



# Python Programming for Beginners

## 08 함수와 모듈

2023학년도 2학기

Suk-Hwan Lee

Computer Engineering  
Artificial Intelligence

Creating the Future

Dong-A University

Division of Computer Engineering &  
Artificial Intelligence

## Section01 함수 기본

### ■ 함수의 개념과 필요성

- **함수(Function)** : ‘무엇’을 넣으면, ‘어떤 것’을 돌려주는 요술 상자
- **메서드(Method)**와 차이점 : 함수는 외부에 별도로 존재, 메서드는 클래스 안에 존재
- **함수의 형식**

함수명()

- **print()** 함수

```
print("CookBook-파이썬")
```

### ■ 함수의 개념과 장점

- **함수(function)** : 어떤 일을 수행하는 코드의 덩어리, 또는 코드의 묶음
- **함수의 장점**
  - ① 필요할 때마다 호출 가능
  - ② 논리적인 단위로 분할 가능
  - ③ 코드의 캡슐화

# Section01 함수 기본

## 함수의 선언

```
def 함수 이름 (매개변수 #1 …):  
    수행문 1  
    수행문 2  
    return <반환값>
```

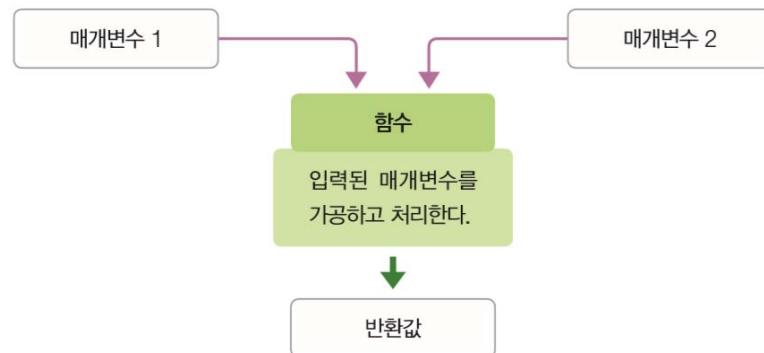
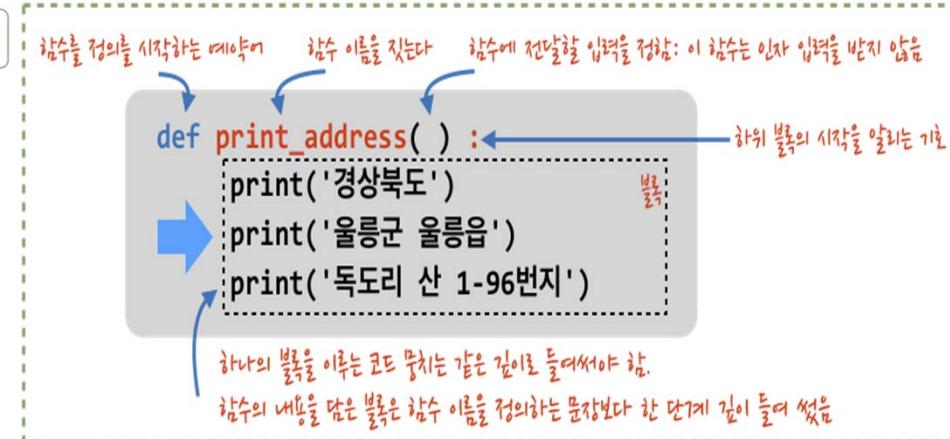


그림 9-3 함수의 기본 형식



- ① `def` : 'definition'의 줄임말로, 함수를 정의하여 시작한다는 의미이다.
- ② **함수 이름** : 함수 이름은 개발자가 마음대로 지정할 수 있지만, 파이썬에서는 일반적으로 다음과 같은 규칙을 사용한다.
  - 소문자로 입력한다.
  - 띄어쓰기를 할 경우에는 \_ 기호를 사용한다. ex) `save_model`
  - 행위를 기록하므로 동사와 명사를 함께 사용하는 경우가 많다. ex) `find_number`
  - 외부에 공개하는 함수일 경우, 줄임말을 사용하지 않고 짧고 명료한 이름을 정한다.

- ③ **매개변수(parameter)** : 함수에서 입력값으로 사용하는 변수를 의미하며, 1개 이상의 값을 적을 수 있다.
- ④ **수행문** : 수행문은 반드시 들여쓰기한 후 코드를 입력해야 한다. 수행해야 하는 코드는 일반적으로 작성하는 코드와 같다. `if`나 `for` 같은 제어문을 사용할 수도 있고, 고급 프로그래밍을 하게 되면 함수 안에 함수를 사용하기도 한다.

# Section01 함수 기본

## ■ 함수의 선언

- 함수 선언 작성 예시를 간단한 코드로 살펴보자

```
def calculate_rectangle_area(x, y)
    return x * y
```

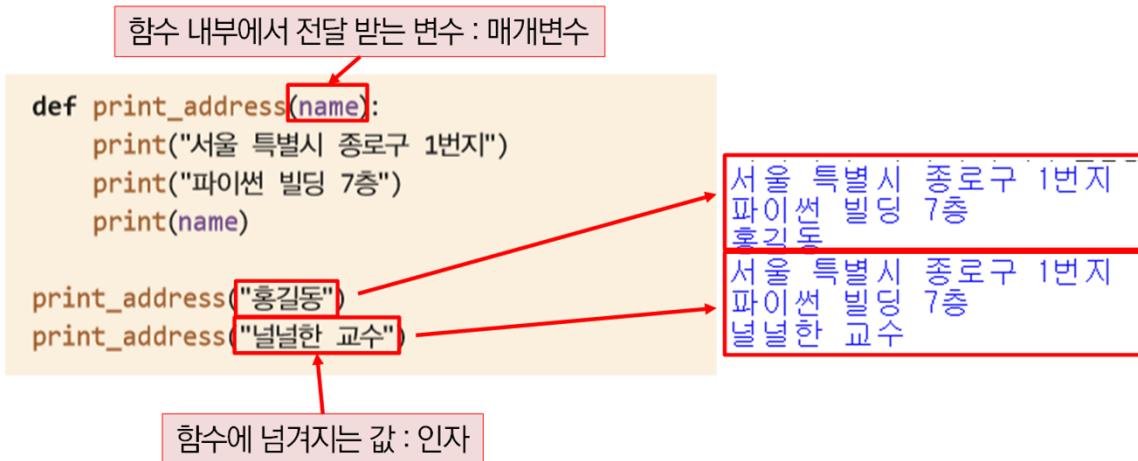
- 먼저 선언된 함수를 확인할 수 있다.
- 함수 이름은 calculate\_rectangle\_area이고, x와 y라는 2개의 매개변수를 사용하고 있다.
- return의 의미는 값을 반환한다는 뜻으로, x와 y를 곱한 값을 반환하는 함수로 이해한다.

## 여기서 잠깐! 반환

- 약간 어렵게 느껴질 수 있는 부분이 바로 '반환'이라는 개념이다. 이는 수학에서의 함수와 같은 개념이라고 생각하면 된다. 예를 들어, 수학에서  $f(x) = x + 10$ 라고 한다면  $f(1)$ 의 값은 얼마일까? 중학교 정도의 수학을 이해하고 있다면  $f(1) = 2$ 라는 것을 알 것이다. 즉, 함수  $f(x)$ 에서  $x$ 에 10이 들어가면 2가 반환되는 것이다. 파이썬의 함수도 같은 개념이다. 수학에서  $x$ 에 해당하는 것이 매개변수, 즉 입력값이고,  $x + 1$ 의 계산 과정이 함수 안의 코드이며, 그 결과가 출력값이다.

## Section01 함수 기본

### ■ 매개변수와 인자



- 함수 정의부를 살펴보면 이전과 달리 함수 이름 뒤의 소괄호 안에 변수 name이 있다. 이 변수 name을 통하여 함수로 값이 전달된다. 메인 프로그램에서 print\_address()를 호출할 때 "홍길동"이라는 문자열을 괄호 안에 넣어 주었는데. 이것이 함수 내부의 name 변수로 전달되는 것이다.
- 함수 호출시 전달되는 실제 값을 인자 argument라고 하고, 함수 내부에서 전달받는 변수를 매개변수 parameter라고 한다.

## Section01 함수 기본



### 매개변수와 인수

- 매개변수는 함수의 인터페이스 정의에 있어 어떤 변수를 사용하는지를 정의하는 것이다. 그에 반해 인수는 실제 매개변수에 대입되는 값을 뜻한다.

코드 5-3 parameter.py

```
1 def f(x):
2     return 2 * x + 7
3
4 print(f(2))
```

```
- 11
```

- [코드 5-3]에서 ‘def f(x):’의 x를 매개변수라고 한다. 일반적으로 함수의 입력값에 대한 정의를 함수 사용에 있어 인터페이스를 정의한다고 한다. 매개변수는 함수의 인터페이스 정의에 있어 어떤 변수를 사용하는지를 정의하는 것이다. 즉, 위 함수에서는 x가 해당 함수의 매개변수이다. 그에 반해, 인수는 실제 매개변수에 대입되는 값을 뜻한다. 매개변수가 설계도라면 인수는 그 설계도로 지은 건물 같은 것이다. 위 코드에서는 f(2)에서 2가 인수에 해당한다.

## Section01 함수 기본

### ■ plus() 함수

Code09-04.py

```
1 ## 함수 선언 부분 ##
2 def plus(v1, v2) :
3     result = 0
4     result = v1 + v2
5     return result
6
7 ## 전역 변수 선언 부분 ##
8 hap = 0
9
10 ## 메인 코드 부분 ##
11 hap = plus(100, 200)
12 print("100과 200의 plus() 함수 결과는 %d" % hap)
```

• 2~5행 : plus() 함수를 정의  
• 4행 : 매개변수로 받은 두 값의 합계를 구함  
• 5행 : 반환  
• 11행 : 100, 200 두 값을 전달하면서 plus() 함수를 호출해 hap에 대입  
• 12행 : plus() 함수에서 반환된 값 출력

### 출력 결과

100과 200의 plus() 함수 결과는 300

### ■ plus() 함수의 형식과 호출 순서

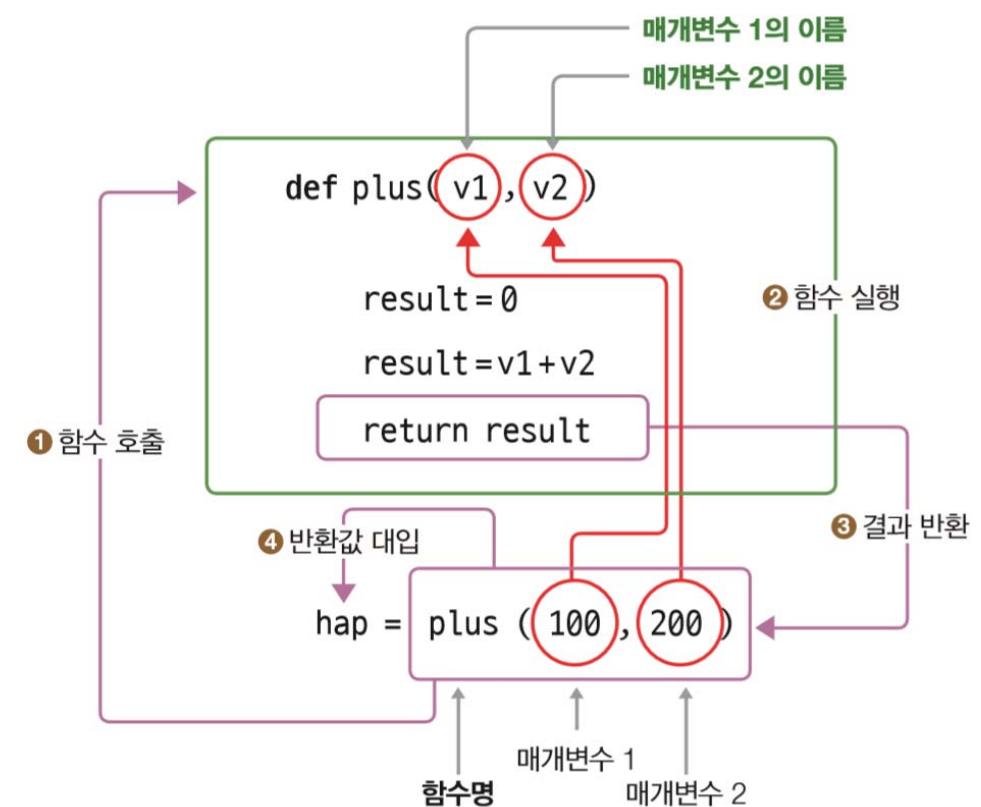


그림 9-4 plus() 함수의 형식과 호출 순서

## Section01 함수 기본

### ■ 여러 개의 값을 넘겨주고 여러 개의 값을 돌려받자

```
def sort_num(n1, n2):          # 2개의 값을 받아오는 함수
    if n1 < n2:
        return n1, n2          # n1이 더 작으면 n1, n2 순서로 반환
    else:
        return n2, n1          # n2가 더 작으면 n2, n1 순서로 반환

print(sort_num(110, 210))      # 110과 210을 함수의 인자로 전달하고 반환되는 값을 출력
print(sort_num(2100, 80))
```

(110, 210)  
(80, 2100)

항상 작은 수, 큰 수 쌍이 반환된다

```
def calc(n1, n2):
    return n1 + n2, n1 - n2, n1 * n2, n1 / n2 # 덧셈, 뺄셈, 곱셈, 나눗셈 결과를 반환
```

n1, n2 = 200, 100

t1, t2, t3, t4 = calc(n1, n2) # 네 개의 값을 반환받기 위해 4개의 변수를 사용함

```
print(n1, '+', n2, '=', t1)
print(n1, '-', n2, '=', t2)
print(n1, '*', n2, '=', t3)
print(n1, '/', n2, '=', t4)
```

사칙연산의 결과를 반환받는다.

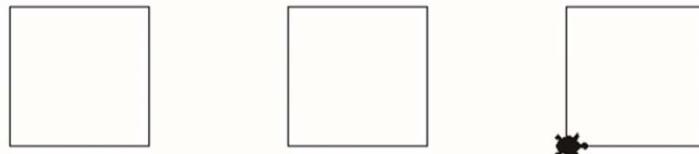
200 + 100 = 300  
200 - 100 = 100  
200 \* 100 = 20000  
200 / 100 = 2.0

## Section01 함수 기본

### LAB<sup>6-1</sup> 사각형을 그리는 함수 만들어보기

터틀 그래픽에서는 원을 그리는 함수는 제공하지만 정사각형을 그리는 함수는 제공하지 않는다. 이 상한 일이지만 어떻게 하겠는가? 우리가 직접 만들어서 사용하자. 일단 함수의 이름은 `square()`라고 하자. 터틀 그래픽에서는 어떻게 정사각형을 그릴 수 있을까? 거북이를 주어진 길이만큼 전진시키고  $(360/n)$ 각도로 방향을 전환하는 작업을 n번 반복하면 된다. n-각형을 그리는 함수를 작성하고 이 함수를 호출하여서 다음과 같은 그림을 그려보자.

원하는 결과



```
import turtle

t = turtle.Turtle()
t.shape("turtle")

def square(length):      # length는 한 변의 길이
    for i in range(4):
        t.forward(length)
        t.left(90)

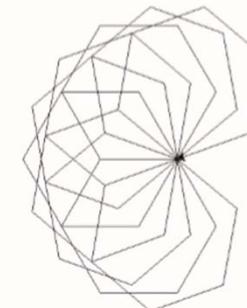
square(100)             # square() 함수를 호출한다.
square(200)             # 호출시 인자값을 100, 200, 300으로 다르게 한다.
square(300)
```

인자가 다르므로 한 변의 길이  
가 다른 사각형을 그린다

### LAB<sup>6-2</sup> n각형을 그리는 함수 만들어보기

n-각형을 그리는 함수를 작성하여 사용해보자. 함수의 이름은 `n_polygon(n, length)`라고 하자. 터틀 그래픽에서는 어떻게 n-각형을 그릴 수 있을까? 거북이를 주어진 길이만큼 전진시키고  $(360/n)$ 각도로 방향을 전환하는 작업을 n번 반복하면 된다. n-각형을 그리는 함수를 작성하고 이 함수를 호출하여서 다음과 같은 그림을 그려보자.

원하는 결과



```
import turtle
t = turtle.Turtle()

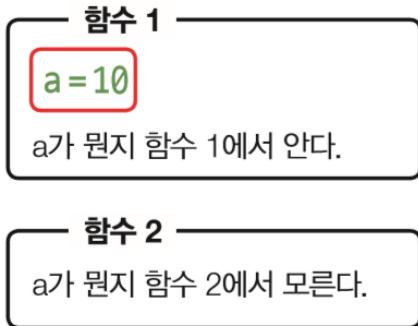
# n-각형을 그리는 함수를 정의한다.
def n_polygon(n, length):
    for i in range(n):
        t.forward(length)
        t.left(360//n)          # 정수 나눗셈은 //으로 한다.

for i in range(10):
    t.left(20)
    n_polygon(6, 100)
```

### ■ 지역 변수와 전역 변수의 이해

- 지역 변수 : 한정된 지역에서만 사용
- 전역 변수 : 프로그램 전체에서 사용

#### ① 지역 변수의 생존 범위



#### ② 전역 변수의 생존 범위

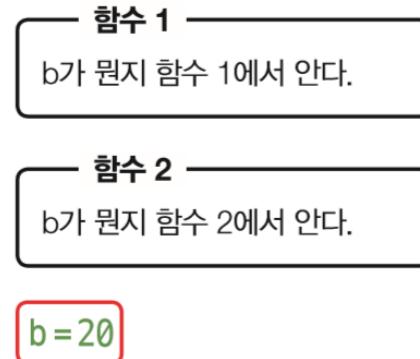


그림 9-6 지역 변수와 전역 변수의 생존 범위

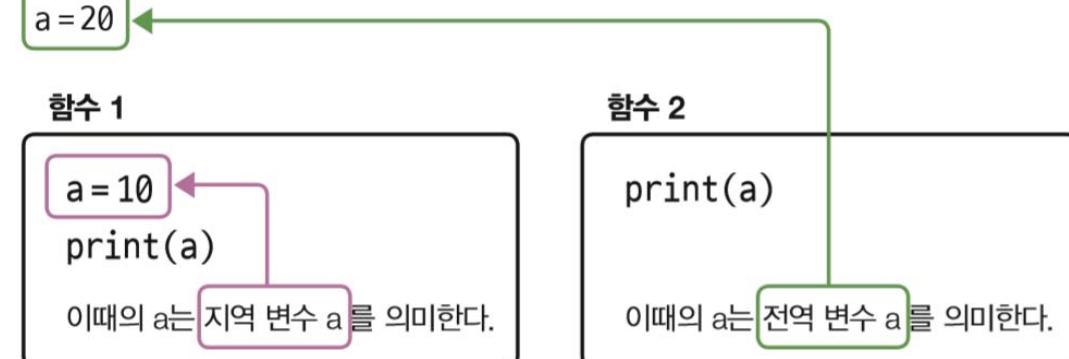


그림 9-7 지역 변수와 전역 변수의 공존

## Section02 지역 변수, 전역 변수

Code09-06.py

```
1 ## 함수 선언 부분 ##
2 def func1() :
3     a = 10      # 지역 변수
4     print("func1()에서 a값 %d" % a)
5
6 def func2() :
7     print("func2()에서 a값 %d" % a)
8
9 ## 전역 변수 선언 부분 ##
10 a = 20       # 전역 변수
11
12 ## 메인 코드 부분 ##
13 func1()
14 func2()
```

- 2~4행 : 한 func1() 함수 정의
- 3행 : a를 선언(지역 변수)
- 10행 : a는 선언(전역 변수)
- 13행 : func1() 함수 호출
- 14행 : func2() 함수 호출

출력 결과

func1()에서 a값 10  
func2()에서 a값 20

- 10행의 전역 변수가 없다면 7행은?

출력 결과

func1()에서 a의 값 10  
Traceback (most recent call last):

```
File "C:/파이썬코드/09-06.py", line 14, in <module>
    func2()
  File "C:/파이썬코드/09-06.py", line 7, in func2
    print("func2()에서 a값 %d" % a)
NameError: name 'a' is not defined
```

## Section02 지역 변수, 전역 변수

### ■ global 예약어 (함수 안에 전역 변수화)

Code09-07.py

```
1 ## 함수 선언 부분 ##
2 def func1() :
3     global a    # 이 함수 안에서 a는 전역 변수
4     a = 10
5     print("func1()에서 a값 %d" % a)
6
7 def func2() :
8     print("func2()에서 a값 %d" % a)
9
10 ## 함수 변수 선언 부분 ##
11 a = 20          # 전역 변수
12
13 ## 메인 코드 부분 ##
14 func1()
15 func2()
```

#### 출력 결과

```
func1()에서 a값 10
func2()에서 a값 10
```

- 3행 : global 예약어로 a 변수를 전역 변수로 지정
- 4행 : 전역 변수 a값을 10으로 변경하므로 func1()과 func2() 함수에서 모두 전역 변수 a값을 10으로 출력

- 지역변수가 아닌 전역변수 counter를 사용하게 된다.
- 함수 내부에서 값을 바꾸면 외부에서도 그 값이 바뀌게 된다

- 만일 함수 내부에서 새로 변수를 만들지 않고 전역변수인 counter를 불러서 사용하려면 어떻게 해야할까? 아래와 같이 global이라는 키워드를 사용하는 것이 바람직하다. **global counter는 함수 외부의 전역변수 counter를 사용하겠다는 선언**이다.

```
def print_counter():
    global counter
    counter = 200
    print('counter =', counter) # 함수 내부의 counter 값

counter = 100
print_counter()
print('counter =', counter) # 함수 외부의 counter 값
```

counter = 200  
counter = 200

- 이와 같은 선언을 할 경우 아래의 그림과 같이 print\_counter()는 외부의 전역변수 counter를 사용하므로 counter = 200을 적용하면 전역변수의 값이 200으로 변경된다.

## Section02 지역 변수, 전역 변수

출처 : 파이썬 프로그램 입문서(가제), <https://python.bakyeono.net/chapter-3-4.html>

### 지역 변수

```
def stamp():
    """쿠폰 스탬프가 찍힌 횟수를 증가시키고, 화면에 출력한다."""
    num_stamp = 2
    num_stamp = num_stamp + 1
    print(num_stamp)

stamp()
```

### 함수 안에서 전역변수 수정하기 - 오류발생

```
num_stamp = 0 # 쿠폰 스탬프가 찍힌 횟수 (전역변수)

def stamp():
    """쿠폰 스탬프가 찍힌 횟수를 증가시키고, 화면에 출력한다."""
    num_stamp = num_stamp + 1 # ❶ 전역변수를 수정하려고 시도함
    print(num_stamp)

stamp()
=====
RESTART: C:\Users\sky\Documents\Python Scripts\test.py =====
Traceback (most recent call last):
  File "C:\Users\sky\Documents\Python Scripts\test.py", line 8, in <module>
    stamp()
  File "C:\Users\sky\Documents\Python Scripts\test.py", line 5, in stamp
    num_stamp = num_stamp + 1 # ❶ 전역변수를 수정하려고 시도함
UnboundLocalError: local variable 'num_stamp' referenced before assignment
```

## Section02 지역 변수, 전역 변수

출처 : 파이썬 프로그램 입문서(가제), <https://python.bakyeono.net/chapter-3-4.html>

### 지역 변수

```
def stamp():
    """쿠폰 스템프가 찍힌 횟수를 증가시키고, 화면에 출력한다."""
    num_stamp = 2
    num_stamp = num_stamp + 1
    print(num_stamp)

stamp()
```

### 함수 안에서 전역변수 수정하기 - 오류발생

num\_stamp = 0 # 쿠폰 스템프가 찍힌 횟수 (전역변수)

지역변수 num\_stamp로 인식

```
def stamp():
    """쿠폰 스템프가 찍힌 횟수를 증가시키고, 화면에 출력한다."""
    num_stamp = num_stamp + 1 # ❶ 전역변수를 수정하려고 시도함
    print(num_stamp)
```

전역변수 num\_stamp로 인식

stamp()

```
===== RESTART: C:\Users\sky\Documents\Python Scripts\test.py =====
Traceback (most recent call last):
  File "C:\Users\sky\Documents\Python Scripts\test.py", line 8, in <module>
    stamp()
  File "C:\Users\sky\Documents\Python Scripts\test.py", line 5, in stamp
    num_stamp = num_stamp + 1 # ❶ 전역변수를 수정하려고 시도함
UnboundLocalError: local variable 'num_stamp' referenced before assignment
```

## Section02 지역 변수, 전역 변수

출처 : 파이썬 프로그램 입문서(가제), <https://python.bakyeono.net/chapter-3-4.html>

```
num_stamp = 0 # 쿠폰 스템프가 찍힌 횟수 (전역변수)
```

```
def stamp():
    """쿠폰 스템프가 찍힌 횟수를 증가시키고, 화면에 출력한다."""
    global num_stamp      # ❶ num_stamp는 전역변수다
    num_stamp = num_stamp + 1 # 이제 오류가 발생하지 않는다
    print(num_stamp)
```

=====

```
stamp() # 화면에 1이 출력된다
stamp() # 화면에 2가 출력된다
```

1  
2

### global 문은 사용하지 않는 것이 좋다

global 문을 배웠지만, 역시 이 명령은 사용하지 않는 것이 좋다.

global 문을 사용하는 것은 함수가 매개변수와 반환값을 이용해 외부와 소통하는 자연스러운 흐름을 깨트리는 일이다.

함수 안에서 전역변수를 수정하지 않고, 매개변수와 반환값만 이용하더라도 함수의 실행 결과를 누적하는 데 부족함이 없다. 다음 예제는 전역변수를 직접 수정하는 대신, 매개변수와 반환값을 이용하도록 stamp() 함수를 수정한 버전이다.

## Section02 지역 변수, 전역 변수

출처 : 파이썬 프로그램 입문서(가제), <https://python.bakyeono.net/chapter-3-4.html>

### global문 사용하지 않고, 매개변수와 반환값만 이용

함수 안에서 전역변수를 수정하지 않고, 매개변수와 반환값만 이용하더라도 함수의 실행 결과를 누적하는 데 부족함이 없다.

다음 예제는 전역변수를 직접 수정하는 대신, 매개변수와 반환값을 이용하도록 stamp() 함수를 수정한 버전이다.

```
num_stamp = 0 # ❶ 쿠폰 스템프가 찍힌 횟수 (전역변수)

def stamp(num_stamp): # ❷ 지역변수(매개변수) num_stamp
    """쿠폰 스템프가 찍힌 횟수를 증가시키고, 화면에 출력한다."""
    num_stamp = num_stamp + 1
    print(num_stamp)
    return num_stamp

num_stamp = stamp(num_stamp) # ❸ 전역변수에 함수의 반환값을 대입한다
num_stamp = stamp(num_stamp)
```

## Section03 함수의 반환값과 매개변수

### ■ 함수의 반환값

- Tip • 반환값은 return 문으로 반환되므로 리턴값이라고도 함. 매개변수는 파라미터라고도 함
- 반환값이 있는 함수
  - 반환값이 없는 함수

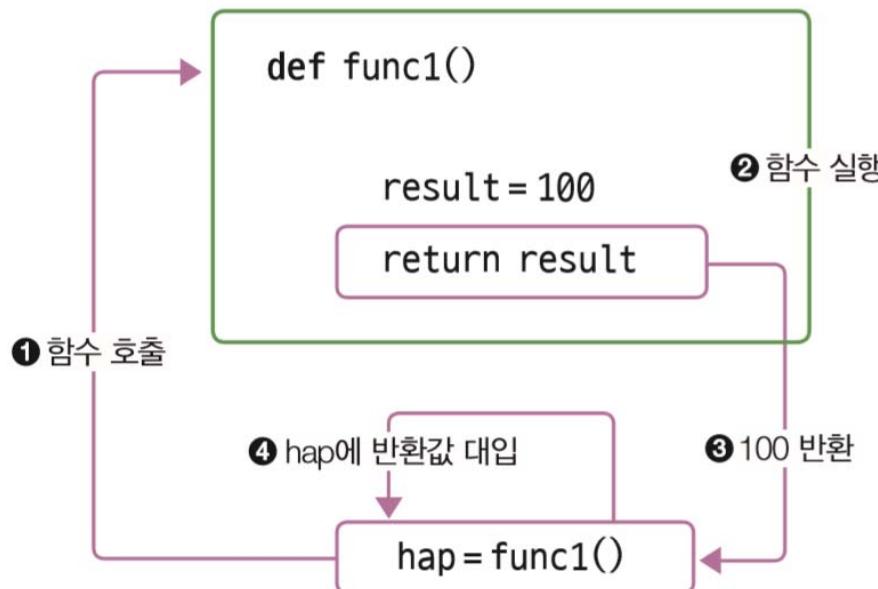


그림 9-8 값의 반환

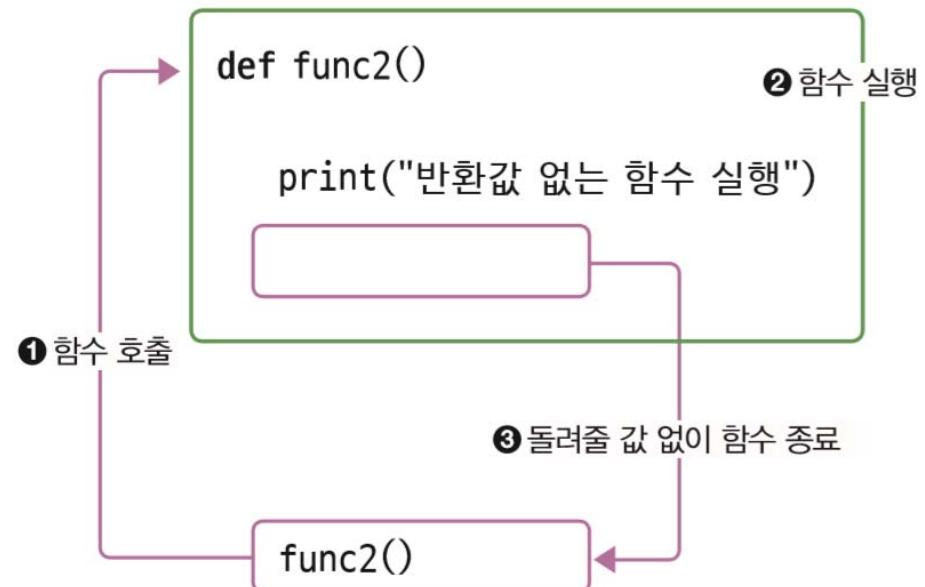


그림 9-9 반환값이 없는 함수의 작동

## Section03 함수의 반환값과 매개변수

### ■ 반환값이 없는 함수

Code09-08.py

```
1 ## 함수 선언 부분 ##
2 def func1() :
3     result = 100
4     return result
5
6 def func2() :
7     print("반환값이 없는 함수 실행")
8
9 ## 전역 변수 선언 부분 ##
10 hap = 0
11
12 ## 메인 코드 부분 ##
13 hap = func1()
14 print("func1()에서 돌려준 값 ==> %d" % hap)
15 func2()
```

#### 출력 결과

```
func1()에서 돌려준 값 ==> 100
반환값이 없는 함수 실행
```

- 13행 : 반환값이 있는 함수인 func1()을 호출하면 func1() 실행 후 func1()의 반환값을 hap에 넣고
- 14행 : 출력
- 15행 : 반환값이 없는 함수인 func2()를 호출하면 반환 않음

### ■ 반환값이 여러 개인 함수

Code09-09.py

```
1 ## 함수 선언 부분 ##
2 def multi(v1, v2) :
3     retList = []          # 반환할 리스트
4     res1 = v1 + v2
5     res2 = v1 - v2
6     retList.append(res1)
7     retList.append(res2)
8     return retList
9
10 ## 전역 변수 선언 부분 ##
11 myList = []
12 hap, sub = 0, 0
13
14 ## 메인 코드 부분 ##
15 myList = multi(100, 200)
16 hap = myList[0]
17 sub = myList[1]
18 print("multi()에서 돌려준 값 ==> %d, %d" % (hap, sub))
```

## Section03 함수의 반환값과 매개변수

- pass 예약어                      pass문 : True/False 실행할 문장 또는 동작을 정의할 때, 아무런 일도 하지 않게 설정하는 것

```
def myFunc():
    pass
```

- True일 때 아무런 할 일이 없다고 빈 줄로 둘 때 오류 발생

```
if True:

else:
    print('거짓이네요')
```

- 오류 해결

```
if True:
    pass
else:
    print('거짓이네요')
```

## Section03 함수의 반환값과 매개변수

### ■ 함수의 매개변수 전달

- 매개변수의 개수를 지정해 전달하는 방법
  - 숫자 2개의 합과 숫자 3개의 합을 구 하는 코드

Code09-10.py

```
1 ## 함수 선언 부분 ##
2 def para2_func( v1, v2 ) :
3     result = 0
4     result = v1 + v2
5     return result
6
7 def para3_func( v1, v2, v3 ) :
8     result = 0
9     result = v1 + v2 + v3
10    return result
11
12 ## 전역 변수 선언 부분 ##
13 hap = 0
```

- 2~5행은 매개변수를 2개,
- 7~10행은 매개변수를 3개 받아 합계를 반환하는 함수 정의

```
14
15 ## 메인 코드 부분 ##
16 hap = para2_func(10, 20)
17 print("매개변수가 2개인 함수를 호출한 결과 ==> %d" % hap)
18 hap = para3_func(10, 20, 30)
19 print("매개변수가 3개인 함수를 호출한 결과 ==> %d" % hap)
```

#### 출력 결과

매개변수가 2개인 함수를 호출한 결과 ==> 30  
매개변수가 3개인 함수를 호출한 결과 ==> 60

## Section03 함수의 반환값과 매개변수

- 매개변수에 기본값을 설정해 놓고 전달하는 방법

Code09-11.py

```
1 ## 함수 선언 부분 ##
2 def para_func( v1, v2, v3 = 0 ) :
3     result = 0
4     result = v1 + v2 + v3
5     return result
6
7 ## 전역 변수 선언 부분 ##
8 hap = 0
9
10 ## 메인 코드 부분 ##
11 hap = para_func(10, 20)
12 print("매개변수가 2개인 함수를 호출한 결과 ==> %d" % hap)
13 hap = para_func(10, 20, 30)
14 print("매개변수가 3개인 함수를 호출한 결과 ==> %d" % hap)
```

디폴트 인자 (다음 페이지)

## Section03 함수의 반환값과 매개변수

### ■ 디폴트 인자

- 파이썬에서는 함수의 매개변수가 기본값을 가질 수 있다. 이것을 **디폴트 인자 default argument**라고 한다.

```
def order(num, pickle = True, onion = True) :
    print('햄버거 {0} 개 - 피클 {1}, 양파 {2}'.format(num, pickle, onion))
```

```
order(1, pickle = False, onion = True)
order(2)      # 햄버거 2개를 주문, 디폴트로 pickle, onion 값은 True임

햄버거 1 개 - 피클 False, 양파 True
햄버거 2 개 - 피클 True, 양파 True
```

- 인자가 부족한 경우에 기본값을 넣어주는 메커니즘이 있다면 편리할 것이다. 바로 이러한 목적으로 사용하는 것이 디폴트 인자이다.

- 위의 코드를 살펴보면 pickle과 onion 매개변수에 True라는 디폴트 인자를 넣어 주었다. 따라서 order(2)와 같이 호출해도 프로그램은 오류없이 잘 수행된다. 반면 이 프로그램에서는 num에 대한 디폴트 값을 주지 않았으며 따라서 order()과 같은 방식으로 함수를 호출하면 프로그램은 오류를 출력할 것이다.



order(2) 만하면 자동으로 order(3, True, True) 가 된다

## Section03 함수의 반환값과 매개변수

### ■ 가변 매개변수

- 매개변수의 개수를 지정하지 않고 전달하는 방법 : **가변 매개변수(Arbitrary Argument)**

Code09-12.py

```
1 ## 함수 선언 부분 ##
2 def para_func (*para) :
3     result = 0
4     for num in para :
5         result = result + num
6
7     return result
8
9 ## 전역 변수 선언 부분 ##
10 hap = 0
11
12 ## 메인 코드 부분 ##
13 hap = para_func(10, 20)
14 print("매개변수가 2개인 함수를 호출한 결과 ==> %d" % hap)
15 hap = para_func(10, 20, 30)
16 print("매개변수가 3개인 함수를 호출한 결과 ==> %d" % hap)
```

• 2행 : \*para로 가변 매개변수 설정  
• 13행 : 호출한 매개변수는 (10, 20) 형식의 튜플로 전달  
• 15행 : 호출한 매개변수는 (10, 20, 30) 형식의 튜플로 전달

- (10, 20, 30)을 매개 변수로 받았을 때 4~5행의 반복
  - 1회 : num에 10을 저장한 후 result = result + 10 result에 10 저장됨
  - 2회 : num에 20을 저장한 후 result = result + 20 result에 30 저장됨
  - 3회 : num에 30을 저장한 후 result = result + 30 result에 60 저장됨
- 매개변수 10개 이상일 때

```
hap = para_func(10, 20, 30, 40, 50, 60, 70, 80, 90, 100)
print(hap)
```

출력 결과

550

## Section03 함수의 반환값과 매개변수

### ❖ 가변 매개변수(Arbitrary Argument)

매개변수의 수가 상황에 따라서 변화

- `*args` : Positional argument (매개변수)로 tuple로 반환      함수에서 정의한 위치대로 대입
- `**kwargs` : keyword argument (매개변수)로 dictionary로 반환 {'key':value}

순서 대신에 parameter 이름으로 맞추어서 값을 전달

```
def function(*args):  
    print(args)  
  
if __name__=="__main__":  
    function(1,'a','bcd')  
  
[출력결과]  
(1, 'a', 'bcd')
```

```
def function(*args):  
    for i,v in enumerate(args):  
        print(i,v)  
  
if __name__=="__main__":  
    function(1,'a','bcd')  
  
[출력결과]  
0 1  
1 a  
2 bcd
```

`enumerate`는 순서가 있는 자료형(리스트, 튜플, 문자열)을 입력으로 받아 인덱스 값을 포함하는 `enumerate` 객체를 돌려준다. `index`와 `value`를 함께 출력하고 싶을 때 주로 사용

`**kwargs` 예시

```
def function(**kwargs):  
    for k,v in kwargs.items():  
        print(k,v)  
  
if __name__=="__main__":  
    function(n1=1,n2=2,n3=3)  
  
[출력결과]  
n1 1  
n2 2  
n3 3
```

`items` 함수는 Key와 Value의 쌍을 튜플로 묶은 값을 dict\_items 객체로 돌려준다

```
def args_function(*args):  
    for i,v in enumerate(args):  
        print(i,v)
```

```
args_function(1,'a','bcd')  
0 1  
1 a  
2 bcd
```

```
args_function(1,2,'a','bcd','efgh')  
0 1  
1 2  
2 a  
3 bcd  
4 efgh
```

```
def kwargs_function(**kwargs):  
    for k,v in kwargs.items():  
        print(k,v)
```

```
kwargs_function(n1=1,n2=2,n3=3)  
n1 1  
n2 2  
n3 3
```

```
kwargs_function(n1=1,n2=2,n3='a',x='bc')  
n1 1  
n2 2  
n3 a  
x bc
```

## Section03 함수의 반환값과 매개변수

### ❖ 가변 매개변수(Arbitrary Argument)

```
def connect(**kwargs):
    print(kwargs, '\n')
    for key, val in kwargs.items():
        print(key, val)

config = {'server': 'localhost',
          'port': 3306,
          'user': 'root',
          'password': 'Py1thon!Xt12'}

connect(**config)

{'server': 'localhost', 'port': 3306, 'user': 'root', 'password': 'Py1thon!Xt12'}
```

```
def fn(*args, **kwargs):
    print(args)
    print(kwargs)

fn(1, 2, x=10, y=20)
(1, 2)
{'x': 10, 'y': 20}
```

## Section03 함수의 반환값과 매개변수

[출처] 유틸 파일썬, “4장 함수와 입출력”

### 가변적인 인자전달

- 인자의 수가 정해지지 않은  
**가변 인자**[arbitrary argument](#)

→ **별표(\*)**를 매개변수의  
앞에 넣어 사용

- 가변적 인자는 [튜플이나 리스트](#)와 비슷하게 `for - in`문에서  
사용 가능

코드 4-26 : 가변 인자를 가지는 함수의 정의와 호출

```
arg_greet.py
def greet(*names):
    for name in names:
        print('안녕하세요', name, '씨')
```

```
greet('홍길동', '양만춘', '이순신') # 인자가 3개
greet('James', 'Thomas') # 인자가 2개
```

실행결과

```
안녕하세요 홍길동 씨
안녕하세요 양만춘 씨
안녕하세요 이순신 씨
안녕하세요 James 씨
안녕하세요 Thomas 씨
```

코드 4-27 : 가변 인자를 가지는 함수에서 [len\(\)](#) 함수 활용

```
arg_foo.py
def foo(*args):
    print('인자의 개수:', len(args))
    print('인자들 :', args)
foo(10, 20, 30)
```

실행결과

```
인자의 개수: 3
인자들 : (10, 20, 30)
```

- `len()` 함수를 이용하여 다음과 같이 가변적으로 전달된 인자의 개수를 출력하는 것도 가능

## Section03 함수의 반환값과 매개변수

[출처] 유틸 파이썬, “4장 함수와 입출력”

- 숫자의 합을 구하는 프로그램
- sum\_num() 함수에 전달될 인자의 개수를 미리 알 수 없는 경우, 가변인자를 받는 \*numbers라는 매개변수를 사용하여 전체 인자를 튜플 형식으로 받을 수 있음

코드 4-28 : 가변 인자를 가지는 함수를 이용한 합계 구하기

```
arg_sum_nums.py

def sum_nums(*numbers):
    result = 0
    for n in numbers:
        result += n
    return result

print(sum_nums(10, 20, 30))          # 10, 20, 30 인자들의 합을 출력
print(sum_nums(10, 20, 30, 40, 50))  # 10, 20, 30, 40, 50 인자들의 합을 출력
```

실행결과

60

150

## Section03 함수의 반환값과 매개변수

### ❖ argparse 모듈 사용 예시

<https://github.com/davide-cocomini/Combining-EfficientNet-and-Vision-Transformers-for-Video-Deepfake-Detection/blob/main/efficient-vit/train.py>

```
# Main body
if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument('--num_epochs', default=300, type=int,
                        help='Number of training epochs.')
    parser.add_argument('--workers', default=10, type=int,
                        help='Number of data loader workers.')
    parser.add_argument('--resume', default='', type=str, metavar='PATH',
                        help='Path to latest checkpoint (default: none.)')
    parser.add_argument('--dataset', type=str, default='All',
                        help="Which dataset to use (Deepfakes|Face2Face|FaceShifter|FaceSwap|NeuralTextures|All)")
    parser.add_argument('--max_videos', type=int, default=-1,
                        help="Maximum number of videos to use for training (default: all).")
    parser.add_argument('--config', type=str,
                        help="Which configuration to use. See into 'config' folder.")
    parser.add_argument('--efficient_net', type=int, default=0,
                        help="Which EfficientNet version to use (0 or 7, default: 0)")
    parser.add_argument('--patience', type=int, default=5,
                        help="How many epochs wait before stopping for validation loss not improving.")

    opt = parser.parse_args()
    print(opt)

    with open(opt.config, 'r') as ymlfile:
        config = yaml.safe_load(ymlfile)

    if opt.efficient_net == 0:
        channels = 1280
    else:
        channels = 2560

    model = EfficientViT(config=config, channels=channels, selected_efficient_net = opt.efficient_net)
```

## Section03 함수의 반환값과 매개변수

### ❖ argparse 모듈

[출처] <https://wikidocs.net/73785>

- run.py 스크립트가 있을 때 명령 프롬프트에서 다음과 같이 실행

```
$ ./run.py
```

- 만약 어떤 옵션에 따라 스크립트가 다르게 동작하기 위해 인자를 전달함

```
$ ./run.py -d 1 -f
```

- ✓ -d는 추가 인자 값을 하나 받고 -f 옵션은 더 이상 추가 인자는 필요없는 형태
- 명령행을 parsing하기 위해 argparse 모듈을 import함. 그리고 parsing할 인자를 add\_argument 메서드를 통해 추가함. 이때 다음 사항에 주의해야 함

- ✓ 추가 옵션을 받는 경우 action="store"를 사용함
- ✓ 추가 옵션을 받지 않고 단지 옵션의 유/무만 필요한 경우 action="store\_true"를 사용함
- ✓ 사용자가 입력한 옵션 값은 dest 인자로 지정한 변수에 저장됨

```
# run.py
import argparse

parser = argparse.ArgumentParser()
parser.add_argument("-d", "--decimal", dest="decimal", action="store")
# extra value
parser.add_argument("-f", "--fast", dest="fast", action="store_true")
# existence/nonexistence
args = parser.parse_args()

print(args.decimal)
print(args.fast)
```

```
$ ./run.py -d 1 -f
```

```
PS C:\Users\sky\Documents\Python Scripts\Python-Lecture> python run.py
None
False
PS C:\Users\sky\Documents\Python Scripts\Python-Lecture> python run.py -d 1 -f
1
True
```

## Section03 함수의 반환값과 매개변수

### ❖ argparse 모듈

[출처] <https://wikidocs.net/73785>

- args.decimal 과 args.fast 값에 따라서 처리 코드

```
# run.py
import argparse

parser = argparse.ArgumentParser()
parser.add_argument("-d", "--decimal", dest="decimal", action="store")
# extra value
parser.add_argument("-f", "--fast", dest="fast", action="store_true")
# existence/nonexistence
args = parser.parse_args()

if args.decimal == '1':
    print("decimal is 1")

if args.fast:
    print("-f option is used")
```

```
$ ./run.py -d 1 -f
```

<https://docs.python.org/ko/3/library/argparse.html>

- class `argparse.ArgumentParser`(prog=None, usage=None, description=None, epilog=None, parents=[], formatter\_class=`argparse.HelpFormatter`, prefix\_chars='-', fromfile\_prefix\_chars=None, argument\_default=None, conflict\_handler='error', add\_help=True, allow\_abbrev=True, exit\_on\_error=True) : 새로운 ArgumentParser 객체 생성

- `ArgumentParser.add_argument(name or flags..., action][, nargs][, const][, default][, type][, choices][, required][, help][, metavar][, dest]`

- `name or flags` - 옵션 문자열의 이름이나 리스트, 예를 들어 `foo` 또는 `-f`, `--foo`.
- `action` - 명령행에서 이 인자가 발견될 때 수행 할 액션의 기본형.
- `nargs` - 소비되어야 하는 명령행 인자의 수.
- `const` - 일부 `action` 및 `nargs` 를 선택할 때 필요한 상수값.
- `default` - 인자가 명령행에 없고 namespace 객체에 없으면 생성되는 값.
- `type` - 명령행 인자가 변환되어야 할 형.
- `choices` - 인자로 허용되는 값의 컨테이너.
- `required` - 명령행 옵션을 생략 할 수 있는지 아닌지 (선택적일 때만).
- `help` - 인자가 하는 일에 대한 간단한 설명.
- `metavar` - 사용 메시지에 사용되는 인자의 이름.
- `dest` - `parse_args()` 가 반환하는 객체에 추가될 어트리뷰트의 이름.

## Section03 함수의 반환값과 매개변수

### ❖ argparse 모듈

[출처] <https://wikidocs.net/73785>

- command line argument를 사용하지 않기
- 파일에 argument들이 저장되어 있고 해당 파일의 각 라인에 대해서 argument 파싱을 해야 할 때가 있음. 이 경우 parser.parse\_args의 인자로 리스트를 넘겨주면 된다.

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument(dest="width", action="store")
parser.add_argument(dest="height", action="store")
parser.add_argument("--frames", dest="frames", default=30, action="store")
parser.add_argument("--qp", dest="qp", default=0, action="store")
parser.add_argument("--configure", dest="configure", action="store")

args = parser.parse_args(["64", "56", "--frames", "60", "--qp", "1", "--configure", "AI"])
print(args.width, args.height, args.frames, args.qp, args.configure)
```

64 56 60 1 AI

```
args = parser.parse_args(["64", "56", "--configure", "AI"])
print(args.width, args.height, args.frames, args.qp, args.configure)
```

64 56 30 0 AI

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument(dest="width", action="store")
parser.add_argument(dest="height", action="store")
parser.add_argument("--frames", dest="frames", action="store")
parser.add_argument("--qp", dest="qp", action="store")
parser.add_argument("--configure", dest="configure", action="store")
```

```
args = parser.parse_args(["64", "56", "--frames", "60", "--qp", "1", "--configure", "AI"])
print(args.width, args.height, args.frames, args.qp, args.configure)
```

### ■ 모듈의 개념

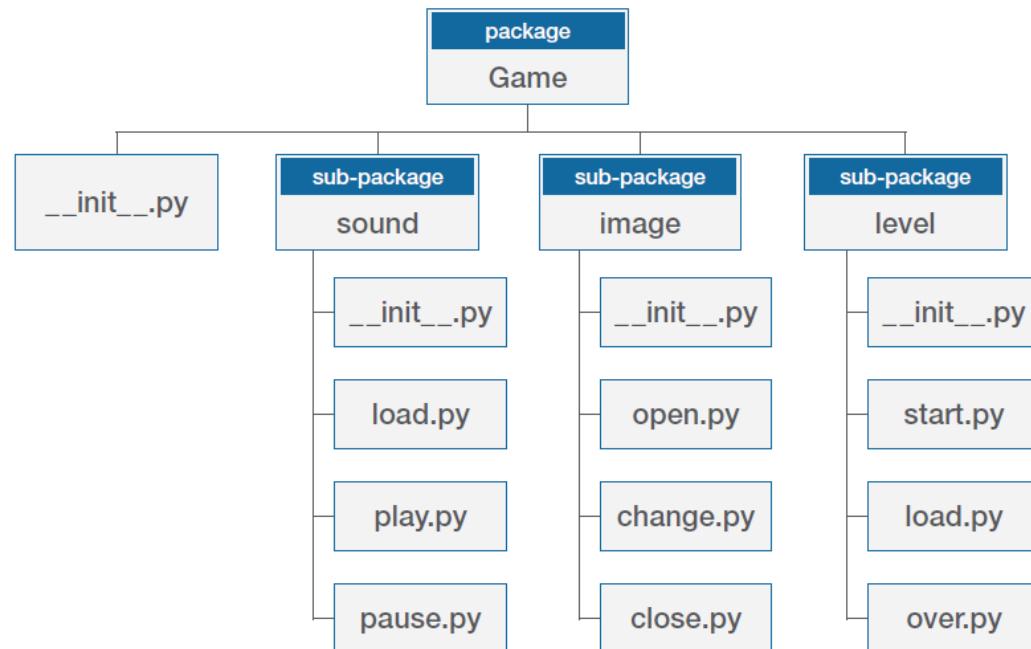
- 내장 모듈이라고 하여 파이썬에서 기본적으로 제공하는 모듈 중 대표적으로 random 모듈이 있다. 이는 난수를 쉽게 생성해 주는 모듈이다. random 모듈을 호출하기 위한 코드는 다음과 같다.

```
>>> import random  
>>> random.randint(1, 1000)  
198
```

- ☞ import 구문이 중요하다. import 구문은 뒤에 있는 모듈, 즉 random을 사용할 수 있도록 호출하라는 명령어이다. 다음으로 해당 모듈의 이름을 사용하여 그 모듈 안에 있는 함수, 여기서는 randint() 함수를 사용할 수 있다. randint() 함수를 사용하기 위해서는 이 randint 함수의 인터페이스, 즉 매개변수의 설정이 어떻게 되어 있는지 알아야 한다.

### ■ 패키지의 개념

- 패키지(package)는 모듈의 묶음이다. 일종의 디렉토리처럼 하나의 패키지 안에 여러 개의 모듈이 있는데, 이 모듈들이 서로 포함 관계를 가지며 거대한 패키지를 만든다.



[모듈과 패키지의 관계]

## Section04 모듈과 패키지

### ■ 모듈 : 함수의 집합

- 모듈이란 함수나 변수 또는 클래스를 모아 놓은 파일임
- 모듈은 다른 파이썬 프로그램에서 불러와 사용할 수 있게끔 만든 파이썬 파일이라고도 함
- 우리는 파이썬으로 프로그래밍을 할 때 굉장히 많은 모듈을 사용한다. 다른 사람들이 이미 만들어 놓은 모듈을 사용할 수도 있고 우리가 직접 만들어서 사용할 수도 있다.

(점프 투 파이썬, <https://wikidocs.net/29>)

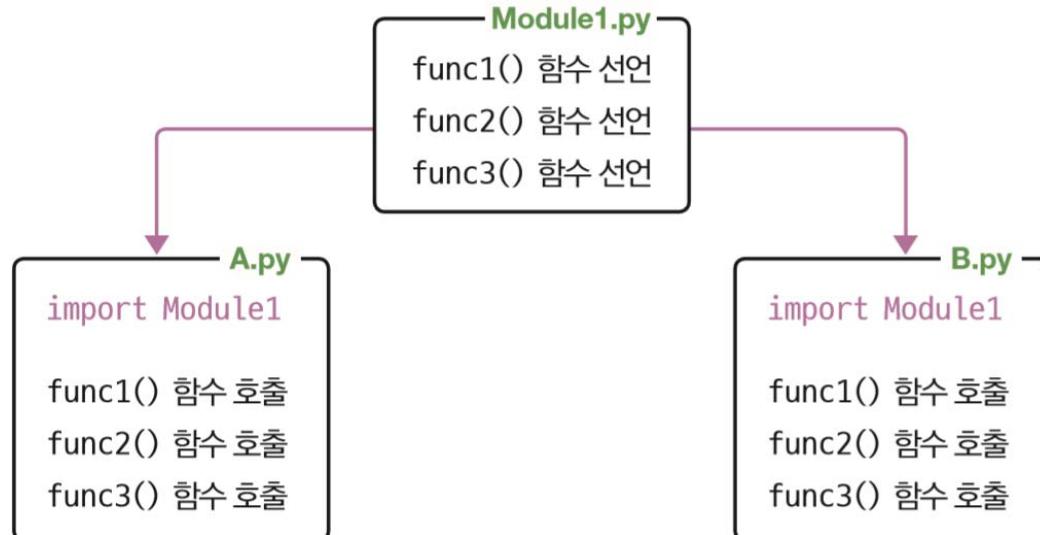


그림 9-10 모듈 사용 예

### • 모듈 module

- 파일에 함수나 변수 또는 클래스들을 모아놓은 스크립트 파일
- 파일은 수많은 개발자들에 의해서 개발된 많은 모듈이 있음
- 만들어진 모듈을 가져올 때에는 'import'와 함께 모듈 이름을 써 줌
- 사용할 때에는 모듈 이름에 점(.)을 찍은 후 모듈 안의 구성요소를 작성

`import 모듈이름1 [, 모듈이름2, ...]`

- 파이썬 설치와 함께 제공되는 모듈을 파일 표준 라이브러리 **python standard library**라고 함
- 문자열과 텍스트 처리를 위한 모듈, 이진 데이터 처리, 날짜, 시간, 배열 등의 자료형 처리를 위한 모듈, 수치 연산과 수학 함수 모듈, 파일과 디렉터리 접근, 유닉스 시스템의 데이터베이스 접근을 위한 모듈, 데이터 압축, 그래픽 모듈 등 100여 가지 이상의 표준 라이브러리들이 있다

## Section04 모듈과 패키지

### ■ 모듈의 생성과 사용

Module1.py

```
1 ## 함수 선언 부분 ##
2 def func1() :
3     print("Module1.py의 func1()이 호출됨.")
4
5 def func2() :
6     print("Module1.py의 func2()가 호출됨.")
7
8 def func3() :
9     print("Module1.py의 func3()이 호출됨.")
```

A.py

```
1 import Module1
2
3 ## 메인 코드 부분 ##
4 Module1.func1()
5 Module1.func2()
6 Module1.func3()
```

출력 결과

Module1.py의 func1()이 호출됨.  
Module1.py의 func2()가 호출됨.  
Module1.py의 func3()이 호출됨.

### ■ 4~6행 함수명으로만 호출

B.py

```
1 from Module1 import func1, func2, func3      # 또는 from Module1 import *
2
3 ## 메인 코드 부분 ##
4 func1()
5 func2()
6 func3()
```

### ■ 모듈명을 생략하고 함수명만 쓸 때 1행 형식

from 모듈명 import 함수1, 함수2, 함수3  
또는  
from 모듈명 import \*

## Section04 모듈과 패키지

### ■ 모듈의 생성과 사용

```
A.py B.py Module1.py X
CookPython(2019.10.15) > Module > Module1.py > ...
1 ## 함수 선언 부분 ##
2 def func1():
3     print("Module1.py의 func1()이 호출됨.")
4
5 def func2():
6     print("Module1.py의 func2()가 호출됨.")
7
8 def func3():
9     print("Module1.py의 func3()이 호출됨.")
10
11 def publicFunc():
12     print('this is public function')
13
14 def _privateFunc():
15     print('this is private function')
16
17
18 class Order:
19     def __init__(self, coffee='Americano', price=3000):
20         self.coffee = coffee
21         self.__price = price
22
23     def printInfo(self):
24         print(f"coffee : {self.coffee}")
25         print(f"price : {self.__price}")
~~
```

```
A.py X B.py Module1.py
CookPython(2019.10.15) > Module > A.py > ...
1 #from Module1 import *
2
3 import Module1
4
5 order1 = Module1.Order()
6
7 print(order1._Order__price)
8 order1.printInfo()
9
10 order2 = Module1.Order('cappuchino', 100000)
11 print(order2._Order__price)
12 order2.printInfo()
13
14 Module1.publicFunc()
15 Module1._privateFunc()
~~
```

```
PS C:\Users\sky\Documents\Python Scripts\Python-Lecture> & C:/User
Lecture/CookPython(2019.10.15)/Module/A.py"
3000
coffee : Americano
price : 3000
100000
coffee : cappuchino
price : 100000
this is public function
this is private function
PS C:\Users\sky\Documents\Python Scripts\Python-Lecture>
```

## Section04 모듈과 패키지

### ■ 모듈의 종류

- 표준 Module, User define module, 3rd Party module로 구분
- 표준 Module : 파이썬에서 제공하는 모듈
- User define Module : 직접 만들어서 사용하는 모듈
- 3rd Party Module : 파이썬이 아닌 외부 회사나 단체에서 제공하는 모듈
  - 파이썬 표준 모듈이 모든 기능을 제공 않음
  - 3rd Party Module 덕분에 파이썬에서도 고급 프로그래밍 가능
  - 게임 개발 기능이 있는 pyGame, 윈도창을 제공 하는 PyGTK, 데이터베이스 기능의 SQLAlchemy 등

## Section04 모듈과 패키지

### ■ 파이썬에서 제공하는 표준 모듈의 목록을 일부 확인

```
import sys  
print(sys.builtin_module_names)
```

```
('__abc', '__ast', '__bisect', '__blake2', '__codecs', '__codecs_cn', '__codecs_hk', '__codecs_iso2022', '__codecs_jp', '__codecs_kr', '__codecs_tw', '__collections', '__contextvars', '__csv', '__datetime', '__functools', '__heapq', '__imp', '__io', '__json', '__locale', '__lsprof', '__md5', '__multibytecodec', '__opcode', '__operator', '__pickle', '__random', '__sha1', '__sha256', '__sha3', '__sha512', '__signal', '__sre', '__stat', '__statistics', '__string', '__struct', '__symtable', '__thread', '__tracemalloc', '__warnings', '__weakref', '__winapi', '__xxsubinterpreters', 'array', 'atexit', 'audioop', 'binascii', 'builtins', 'cmath', 'errno', 'faulthandler', 'gc', 'itertools', 'marshal', 'math', 'mmap', 'msvcrt', 'nt', 'parse', 'sys', 'time', 'winreg', 'xxsubtype', 'zlib')
```

- ❖ sys 모듈은 파이썬 인터프리터가 제공하는 변수와 함수를 직접 제어할 수 있게 해주는 모듈임

Tip • dir(\_\_builtins\_\_) 명령어로도 제공하는 모듈과 예약어 확인

### ■ 수학 계산 모듈인 math 모듈이 제공하는 함수의 목록 보기

```
import math  
dir(math)
```

#### 출력 결과

```
[ '__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin',  
'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e',  
'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum',  
'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp',  
'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'sin',  
'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

- ❖ dir() 내장 함수는 어떤 객체를 인자로 넣어주면 해당 객체가 어떤 변수와 메소드(method)를 가지고 있는지 나열해줍니다.

## 1) import로 모듈 가져오기

import 모듈

### ➤ import 모듈1, 모듈2

- 모듈1.변수, 모듈1.함수(), 모듈1.클래스()

```
>>> import math  
>>> math.pi  
3.141592653589793
```

```
>>> import math  
>>> math.sqrt(4.0)  
2.0  
>>> math.sqrt(2.0)  
1.4142135623730951
```

## 2) import as로 모듈 이름 지정하기

### ➤ import 모듈 as 이름

```
>>> import math as m      # math 모듈을 가져오면서 이름을 m으로 지정  
>>> m.sqrt(4.0)          # m으로 제곱근 함수 사용  
2.0  
>>> m.sqrt(2.0)          # m으로 제곱근 함수 사용  
1.4142135623730951
```

- 긴 모듈 이름을 간략하게 부르는 별명을 붙여주는 방법

### NOTE : import ~ as 문법

모듈 내에 있는 클래스나 메소드를 활용할 때 점으로 연결해주는데, 모듈 이름이 너무 긴 경우 계속해서 이름을 써주는 것이 번거로울 수 있다. 이 때, as를 활용하여 모듈의 새 이름을 지정할 수 있다. 보통 math 모듈은 m, datetime은 dt, random은 rd, turtle은 t와 같은 짧은 이름을 넣어 사용한다.

ex1) import datetime as dt  
ex2) import random as rd  
ex3) import math as m  
ex4) import turtle as t

# 모듈 가져오기

[출처] <https://dojang.io/mod/page/view.php?id=2441>

## 3) from import로 모듈의 일부만 가져오기

### ➤ from 모듈 import 변수 (또는 함수, 클래스)

- 이후 모듈 이름을 붙이지 않고, 변수(또는 함수, 클래스) 바로 사용 가능

```
>>> from math import pi      # math 모듈에서 변수 pi만 가져옴  
>>> pi                      # pi를 바로 사용하여 원주율 출력  
3.141592653589793
```

```
>>> from math import sqrt    # math 모듈에서 sqrt 함수만 가져옴  
>>> sqrt(4.0)              # sqrt 함수를 바로 사용  
2.0  
>>> sqrt(2.0)              # sqrt 함수를 바로 사용  
1.4142135623730951
```

```
>>> from math import pi, sqrt  # math 모듈에서 pi, sqrt를 가져옴  
>>> pi                      # pi로 원주율 출력  
3.141592653589793  
>>> sqrt(4.0)              # sqrt 함수 사용  
2.0  
>>> sqrt(2.0)              # sqrt 함수 사용  
1.4142135623730951
```

### ➤ from 모듈 import \*

- 모듈의 모든 변수, 함수, 클래스를 가져오기

```
>>> from math import *        # math 모듈의 모든 변수, 함수, 클래스를 가져옴  
>>> pi                      # pi로 원주율 출력  
3.141592653589793  
>>> sqrt(4.0)              # sqrt 함수 사용  
2.0  
>>> sqrt(2.0)              # sqrt 함수 사용  
1.4142135623730951
```

### ➤ datetime 모듈 : 뒤에 자세한 설명

: <https://docs.python.org/ko/3/library/datetime.html>

```
from datetime import datetime  
start_time = datetime.now()  
print(start_time)  
start_time.replace(month=12, day=25)
```

2021-10-09 01:30:02.701076

datetime.datetime(2021, 12, 25, 1, 30, 2, 701076)

## 4) from import로 모듈의 일부를 가져온 뒤 이름 지정

### ➤ from 모듈 import 변수 (또는 함수, 클래스) as 이름

```
>>> from math import sqrt as s      # math 모듈에서 sqrt 함수를 가져오면서 이름을 s로 지정
>>> s(4.0)                         # s로 sqrt 함수 사용
2.0
>>> s(2.0)                         # s로 sqrt 함수 사용
1.4142135623730951
```

- 여러 개를 가져올 경우 콤마로 구분하여 as를 여러 개 지정

### ➤ from 모듈 import 변수 as 이름1, 함수 as 이름2, 클래스 as 이름3

```
>>> from math import pi as p, sqrt as s
>>> p      # p로 원주율 출력
3.141592653589793
>>> s(4.0)  # s로 sqrt 함수 사용
2.0
>>> s(2.0)  # s로 sqrt 함수 사용
1.4142135623730951
```

### [Note] 가져온 모듈 해제하기, 다시 가져오기

- import로 가져온 모듈(변수, 함수, 클래스)은 del로 해제할 수 있다.

```
>>> import math
>>> del math
```

- 모듈을 다시 가져오려면 importlib.reload를 사용한다

```
>>> import importlib
>>> import math
>>> importlib.reload(math)
```

## ▶ 모듈/패키지 사용

예제

```
import os
import cv2
import json
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sn
import pandas as pd
from tensorflow.keras.preprocessing.image import ImageDataGenerator, img_to_array, load_img
from tensorflow.keras.utils import to_categorical
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix

X = []
Y = []

for img in real_data:
    X.append(img_to_array(load_img(data_dir+'/real/'+img)).flatten() / 255.0)
    Y.append(1)
for img in fake_data:
    X.append(img_to_array(load_img(data_dir+'/fake/'+img)).flatten() / 255.0)
    Y.append(0)

Y_val_org = Y

#Normalization
X = np.array(X)
Y = to_categorical(Y, 2)

#Reshape
X = X.reshape(-1, 128, 128, 3)

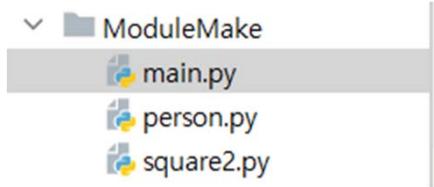
#Train-Test split
X_train, X_val, Y_train, Y_val = train_test_split(X, Y, test_size = 0.2, random_state=5)
```

# 모듈 만들기

[참고] <https://dojang.io/mod/page/view.php?id=2447>

## 1) 모듈 만들기

- square2.py, person.py 모듈 만들고, main.py에서 import하여 가져오기



- square2.py

```
base = 2

def square(n):
    return base ** n

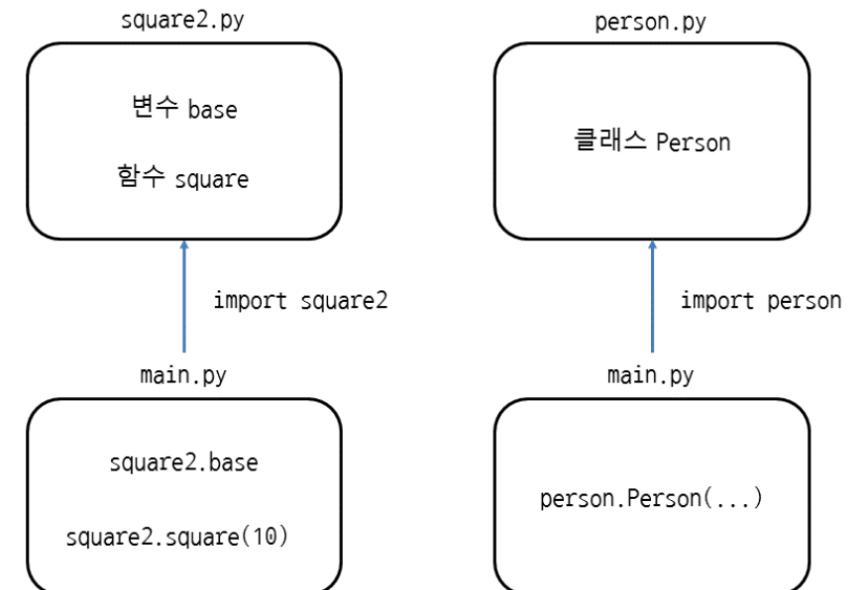
if __name__ == '__main__':
    print(square(3))
```

- person.py

```
class Person: # 클래스
    def __init__(self, name, age, address):
        self.name = name
        self.age = age
        self.address = address

    def greeting(self):
        print('안녕하세요. 저는 {}'.format(self.name))
```

- square2.py에서 실행할 경우 True, 다른 파일에서 import할 경우 False으로 실행이 안 됨
- 테스트용으로 많이 사용



# 모듈 만들기

## 2) 모듈 사용하기1

- main.py

```
import square2, person

def main():
    print(square2.base)          # 모듈.변수 형식으로 모듈의 변수 사용
    print(square2.square(10))    # 모듈.함수() 형식으로 모듈의 함수 사용

# 모듈.클래스()로 person 모듈의 클래스 사용
maria = person.Person('마리아', 20, '서울시 서초구 반포동')
maria.greeting()

if __name__ == "__main__":
    main()
```

2  
1024  
안녕하세요. 저는 마리아입니다.

## 3) 모듈 사용하기2

- from import로 클래스를 가져온 뒤 모듈 이름을 붙이지 않고 사용

```
import square2
from person import Person

def main():
    print(square2.base)
    print(square2.square(10))

    maria = Person('마리아', 20, '서울시 서초구 반포동')
    maria.greeting()

if __name__ == "__main__":
    main()
```

## Section04 모듈과 패키지

### ■ 모듈을 활용해 글자를 쓰는 거북이 프로그램

myTurtle.py

```
1 import random
2 from tkinter.simpledialog import *
3
4 def getString() :
5     retStr = ''
6     retStr = askstring('문자열 입력', '거북이 쓸 문자열을 입력')
7     return retStr
8
9 def getRGB() :
10    r, g, b = 0, 0, 0
11    r = random.random()
12    g = random.random()
13    b = random.random()
14    return(r, g, b)
15
```

- 4~7행 : 문자열을 입력받아 반환하는 함수 생성
- 9~14행 : 무작위로 RGB 색상 추출해서 튜플 반환하는 함수 생성
- tkinter.simpledialog 모듈의 askinteger, askfloat, askstring : 다이얼로그 상의 질문 외에 정수, 실수 또는 문자를 입력받을 때

```
16 def getXYAS(sw, sh) :
17     x, y, angle, size = 0, 0, 0, 0
18     x = random.randrange(-sw / 2, sw / 2)
19     y = random.randrange(-sh / 2, sh / 2)
20     angle = random.randrange(0, 360)
21     size = random.randrange(10, 50)
22     return [x, y, angle, size]
```

- 16~22행 : x, y, 각도, 크기를 무작위로 추출해서 리스트로 묶어 반환하는 함수 생성

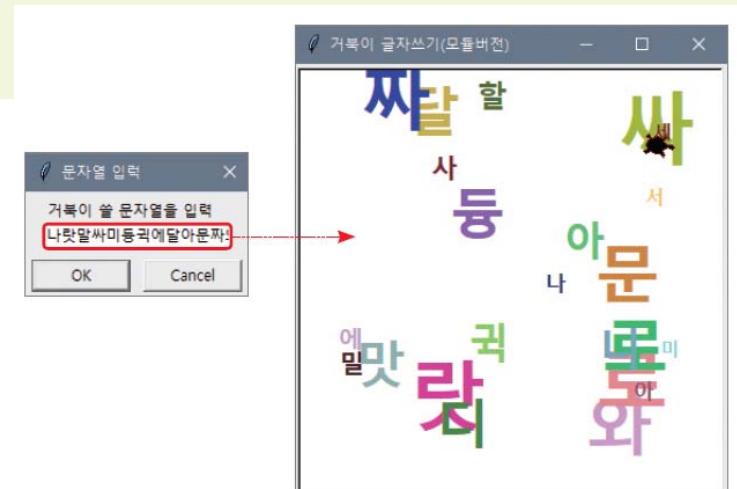
## Section04 모듈과 패키지

Code09-14.py

```
1 from myTurtle import *
2 import turtle
3
4 ## 전역 변수 선언 부분 ##
5 inStr = ''
6 swidth, sheight = 300, 300
7 tX, tY, tAngle, tSize = [0] * 4
8
9 ## 메인 코드 부분 ##
10 turtle.title('거북이 글자쓰기(모듈버전)')
11 turtle.shape('turtle')
12 turtle.setup(width = swidth + 50, height = sheight + 50)
13 turtle.screensize(swidth, sheight)
14 turtle.penup()
15 turtle.speed(5)
16
```

1행 : myTurtle 모듈 임포트

```
17 inStr = getString()           17행과 21~22행 : 모듈의 함수를 사용
18
19 for ch in inStr :
20
21     tX, tY, tAngle, txtSize = getXYAS(swidth, sheight)
22     r, g, b = getRGB()
23
24     turtle.goto(tX, tY)
25     turtle.left(tAngle)
26
27     turtle.pencolor((r, g, b))
28     turtle.write(ch, font = ('맑은고딕', txtSize, 'bold'))
29
30 turtle.done()
```



## Section04 모듈과 패키지



여기서 잠깐

파이썬에서 함수의 매개변수 전달은 값에 의한 전달(Call By Value)과 참조에 의한 전달(Call By Reference) 두 가지가 있다.

함수의  
매개변수 전달  
방법

### ① 값에 의한 전달

값에 의한 전달은 일반 변수나 값을 전달할 때 함수에 동일한 크기의 별도의 메모리 공간이 확보되어 값이 복사되는 방식이다. 지금까지 소개한 방법은 모두 값에 의한 전달이다.

```
def func(p) : # p는 별도의 메모리 공간을 확보함
    p = 222
```

```
v = 111
func(v)
print(v)      # 111이 출력됨
```

```
def func(p):
    p = 222
    print(p, id(p))

y = 111
func(y)
print(y, id(y))
```

```
222 140727261608608
111 140727261605056
```

```
def func(p):
    p[0] = 222
    print(p, id(p))

y = [111]
func(y)
print(y, id(y))
```

```
[222] 2244068323968
[222] 2244068323968
```

### ② 참조에 의한 전달

참조에 의한 전달은 리스트(튜플, 딕셔너리, 세트도 해당)를 매개변수로 전달하므로 주소(Address)가 전달되어 메모리 공간이 공유되는 방식이다. 따라서 함수에서 리스트를 변경하면 메인 코드의 리스트도 변경된다. [그림 7-7]에서 설명한 개념과 비슷하다.

```
def func(p) : # 리스트 p는 리스트 v와 메모리를 공유함
    p[0] = 222

v = [111]
func(v)
print(v[0])    # 222가 출력됨
```

# 날짜와 시간 모듈

[출처] 으뜸 파이썬, “7장 모듈과 활용”

## ◆ datetime 모듈

- 날짜와 시간에 관한 기능을 제공하고 조작할 수 있는 모듈
- 라이브러리 : <https://docs.python.org/ko/3/library/datetime.html>
- 코드 : <https://github.com/python/cpython/blob/3.10/Lib/datetime.py>
- 아래의 `datetime.datetime.now()`에서 앞의 `datetime`은 모듈의 이름, 뒤의 `datetime`은 클래스의 이름

### 대화창 실습 : datetime 모듈의 임포트와 사용

```
>>> import datetime      # 이하 이 문장은 생략함
>>> datetime.datetime.now()
datetime.datetime(2019, 1, 2, 6, 57, 27, 904565)
```

- `date` 클래스의 `today()` 메소드는 현재 날짜를 `today`에 반환
- `today` 값을 프롬프트에서 출력하면 `datetime.date` 클래스가 가진 년, 월, 일을 출력해줌

### 대화창 실습 : datetime 모듈의 사용법

```
>>> today = datetime.date.today()
>>> print(today)
2019-01-02
>>> today
datetime.date(2019, 1, 2)
>>> today.year
2019
>>> today.month
1
>>> today.day
2
```

# 날짜와 시간 모듈

[출처] 유틸 파이썬, “7장 모듈과 활용”

## • dir() 함수

- 모듈이 가진 클래스의 목록을 출력

대화창 실습 : datetime 모듈의 속성과 클래스를 알아보는 dir() 함수

```
>>> dir(datetime)
```

```
['MAXYEAR', 'MINYEAR', '__builtins__', '__cached__', '__doc__', '__file__',  
 '__loader__', '__name__', '__package__', '__spec__', '_divide_and_round', 'date',  
 'datetime', 'datetime CAPI', 'time', 'timedelta', 'timezone', 'tzinfo']
```

- dir() 함수는 datetime 오브젝트에서 사용 가능한 속성을 반환
- MAXYEAR는 datetime 오브젝트가 표현 가능한 최대 년도로 9999 값을 가짐
- MINYEAR는 1 값을 가짐
- date, datetime, datetime CAPI, time, timedelta, timezone, tzinfo 와 같은 클래스들은 날짜, 시간, 시간대, 시간대 정보를 편리하게 이용할 수 있는 기능이 있음

## replace() 메소드

- datetime에서 날짜나 시간 값을 변경하고 싶을 때 사용

대화창 실습 : 날짜 및 시간 변경

```
>>> start_time = datetime.datetime.now()  
>>> start_time.replace(month = 12, day = 25)  
datetime.datetime(2019, 12, 25, 7, 1, 25, 880317)
```

## • dir([object])

- object가 모듈 객체면, 리스트에는 모듈 속성의 이름이 포함됨.
- object가 형 또는 클래스 객체면, 리스트에는 그것의 속성 이름과 베이스의 속성 이름들이 재귀적으로 포함됨.
- 그 밖의 경우, 리스트에는 객체의 속성 이름, 해당 클래스의 속성 이름 및 해당 클래스의 베이스 클래스들의 속성 이름을 재귀적으로 포함함.

- 매번 “[모듈 이름].[클래스 이름].[메소드이름]()”을 점 연산자로 구분해서 적는 것은 매우 번거로움
- as 구문을 사용하여 모듈 이름 datetime을 별칭인 dt로 간단하게 줄일 수 있다.

import [모듈 이름] as [모듈의 별칭]

대화창 실습 : 현재 날짜 및 시간 변경

```
>>> import datetime as dt  
>>> start_time = dt.datetime.now()  
>>> start_time.replace(month = 12, day = 25)  
datetime.datetime(2019, 12, 25, 7, 1, 25, 880317)
```

이 결과는 컴퓨터의 실행시간에 따라  
매번 달라질 수 있음

# 날짜와 시간 모듈

[출처] 으뜸 파이썬, “7장 모듈과 활용”

## 1) 남은 시간 구하기

코드 7-1 : datetime 모듈을 사용하여 크리스마스까지 남은 시간 구하기

xmas\_left\_day.py

```
import datetime as dt

today = dt.date.today()
print('오늘은 {}년 {}월 {}일입니다'.format(today.year, today.month, today.day))

xMas = dt.datetime(2021, 12, 25)
time_gap = xMas - dt.datetime.now()

print('다음 크리스마스 까지는 {}일 {}시간 남았습니다.'.format(\n    time_gap.days, time_gap.seconds // 3600))
```

The screenshot shows the PyCharm interface with a script named 'xmas\_left\_day.py' open. The output window displays the following text:  
"C:\sky\동영상강의\2021학년도2학기\파이썬\참고\..."  
오늘은 2021년 10월 9일입니다  
다음 크리스마스 까지는 76일 21시간 남았습니다.

## 2) 100일 뒤 날짜 구하기

- datetime.timedelta 모듈은 두 날짜의 차이 기간을 나타낼 때 사용하는 모듈이다. timedelta 객체에는 산술 연산자(+, -)를 사용할 수 있기 때문에 어떤 날짜에 특정 기간(일, 시, 분, 초)을 더하거나 뺄 수 있다.
- timedelta 이용하여 100일 후와 100일 전의 날짜를 구해보기

코드 7-2 : 100일 후, 100일 전 날짜 구하기

```
import datetime as dt

print('오늘 =', dt.datetime.now()) # 현재시간을 구한다

hundred = dt.timedelta(days = 100) # 100일 경과시간
print(hundred)

plus100day = dt.datetime.now() + hundred # 현재 시간에서 100일 경과시간을 더함
print('100일 후 =', plus100day)

minus100day = dt.datetime.now() - hundred
print('100일전 =', minus100day)

"C:\sky\동영상강의\2021학년도2학기\파이썬\참
오늘 = 2021-10-09 02:15:04.733571
100 days, 0:00:00
100일 후 = 2022-01-17 02:15:04.733571
100일전 = 2021-07-01 02:15:04.733571

Process finished with exit code 0
```

# 날짜와 시간 모듈

[출처] 유틸 파이썬, “7장 모듈과 활용”

- timedelta에 들어가는 인자

나타내는 날짜	코드
1 주	<code>datetime.timedelta(weeks=1)</code>
1 일	<code>datetime.timedelta(days=1)</code>
1 시간	<code>datetime.timedelta(hours=1)</code>
1 분	<code>datetime.timedelta(minutes=1)</code>
1 초	<code>datetime.timedelta(seconds=1)</code>
1 밀리초	<code>datetime.timedelta(milliseconds=1)</code>
1 마이크로초	<code>datetime.timedelta(microseconds=1)</code>

- strftime(): localtime() 함수가 반환한 struct\_time이라는 형식의 튜플 값을 지정된 포맷의 문자열로 변환

## 3) time 모듈

- 시간에 관련된 정보를 제공하는 모듈
- ✓ 유닉스 시스템 시작 시간인 1970년1월1일0시0분0초 협정 세계시(UTC)를 epoch이라고 함. 유닉스 운영체제에서 표준으로 사용되는 시간 체계는 epoch 시간 혹은 유닉스 시간이라고도 한다.

대화창 실습 : time 모듈을 이용한 에포 이후의 시간 출력

```
>>> import time  
>>> seconds = time.time()  
>>> print('에포 이후의 시간 = ', seconds)  
에포 이후의 시간 = 1555674634.0023367
```

코드 7-3 : time.time()을 통한 현재시간 출력

epoch\_time\_2\_local\_time.py

```
import time  
  
unix_timestamp = time.time()  
local_time = time.localtime(unix_timestamp)  
print(time.strftime('%Y-%m-%d %H:%M:%S', local_time))
```

"C:\sky\동영상강의\2021학년도2학기\파이  
2021-10-09 09:55:41

# 날짜와 시간 모듈

[출처] 유틸 파이썬, “7장 모듈과 활용”

- 쓰레드thread

- 한 프로그램 안에서 실행되는 작은 실행단위

- sleep()

- 일정한 시간동안 현재 실행중인 쓰레드를 일시 중지

코드 7-4 : time 모듈의 sleep() 함수 사용

sleep\_time.py

```
import time

print("바로 출력되는 구문.") # 이 문장은 바로 출력된다
time.sleep(4.5)
print("4.5초 후 출력되는 구문.") # 이 문장은 4.5초 후에 출력된다
```

실행결과

바로 출력되는 구문.

4.5초 후 출력되는 구문.

- 1에서 10까지의 합을 구하여 출력하는데까지 걸리는 시간을 정확하게 알아보기

코드 7-5 : time 모듈을 사용한 경과시간의 출력

elapsed\_time.py

```
import time

start_time = time.time() # 시작시간을 기록
print(1+2+3+4+5+6+7+8+9+10)
```

```
end_time = time.time() # 종료시간을 기록
```

```
gap = end_time - start_time
```

```
print('1에서 10까지의 합을 구하고 출력하는 시간 :{:7.4f}초'.format(gap))
```

실행결과

55

1에서 10까지의 합을 구하고 출력하는 시간 : 0.0008초.

# random 모듈

[출처] 유틸 파이썬, “7장 모듈과 활용”

## 4) random 모듈

- 임의의 수를 생성하거나, 리스트 내의 원소를 무작위로 섞거나 선택하는 함수를 포함

- random() 함수는 0 이상 1 미만의 임의의 실수를 반환함
- randrange(n, m)은 n이상 m 미만의 임의의 정수를 반환

[표 7-3] random 모듈에 포함된 함수들

함수	하는 일
random()	0에서 1사이의 실수를 생성한다.(1은 포함하지 않음)
randrange()	지정된 범위 내의 정수를 반환한다.
randint(a, b)	a <= N <= b 사이의 랜덤 정수 N을 반환한다.
shuffle(seq)	주어진 seq 리스트의 요소를 랜덤하게 섞는다.
choice(seq)	seq 시퀀스 내의 임의의 요소를 선택한다.
sample()	지정된 개수의 요소를 임의로 선택한다.

대화창 실습 : random 모듈을 활용한 값 생성

```
>>> import random as rd
>>> rd.random() # 0 이상 1 미만의 실수를 반환함
0.19452357419514088
>>> rd.random() # 매번 다른 실수를 반환함
0.6947454047320903
>>> rd.randrange(1, 7) # 1 이상 7 미만의 정수를 반환함
6
>>> rd.randrange(0, 10, 2) # 1 이상 10 미만 정수 중 2의 배수를 반환함
2
>>> rd.randint(1, 10) # 1 이상 10 이하의(1, 10이 포함) 임의의 정수를 반환함
3
```

## random 모듈

[출처] 유틸 파일썬, “7장 모듈과 활용”

대화창 실습 : random 모듈을 활용한 섞기와 고르기

```
>>> numlist = [10, 20, 30, 40, 50]
>>> rd.shuffle(numlist) # 리스트의 원소를 랜덤하게 섞는다
>>> numlist
[20, 30, 10, 40, 50]
>>> rd.choice(numlist) # 리스트의 원소들 중에서 랜덤하게 하나를 고른다
20
>>> rd.sample(numlist, 3) # 리스트의 원소들 중에서 랜덤하게 세개를 고른다
[40, 20, 30]
```

- shuffle() : 시퀀스의 원소를 랜덤하게 섞어 반환
- choice() : 인자로 들어온 시퀀스로부터 임의의 원소를 추출
- sample() : 원소를 랜덤하게 반환
  - ✓ 반환할 원소의 개수를 인자로 넣어줄 수 있음

- shuffle()은 매번 다른 순서로 시퀀스를 섞어서 반환함

대화창 실습 : random 모듈을 활용한 섞기

```
>>> a = list(range(1, 11)) # 1에서 10까지의 연속적인 정수를 생성
>>> rd.shuffle(a) # 리스트의 원소를 랜덤하게 섞는다
>>> print(a)
[2, 1, 4, 3, 10, 6, 9, 8, 5, 7]
```

# random 모듈

[출처] 유틸 파일썬, “7장 모듈과 활용”

random – Pseudorandom number generators

[참고] <https://pymotw.com/2/random/>

## Seeding

- `random()` produces different values each time it is called, and has a very large period before it repeats any numbers. This is useful for producing unique values or variations, but there are times when having the same dataset available to be processed in different ways is useful.
- One technique is to use a program to generate random values and save them to be processed by a separate step. That may not be practical for large amounts of data, though, so `random` includes the `seed()` function for initializing the pseudorandom generator so that it produces an expected set of values.

```
import random

for i in range(5):
    print('%04.3f' % random.random())

0.449
0.652
0.789
0.094
0.028
```

```
random.seed(1)

for i in range(5):
    print('%04.3f' % random.random())

0.134
0.847
0.764
0.255
0.495
```

## 현재 시간으로 seeding 예

[참고]  
<https://stackoverflow.com/questions/27276135/python-random-system-time-seed>

```
import random
from datetime import datetime
random.seed(datetime.now())

import random
import time
random.seed(time.time())

import time
import random

t = int(time.time() * 1000.0)
random.seed( ((t & 0xff000000) >> 24) +
            ((t & 0x00ff0000) >> 8) +
            ((t & 0x0000ff00) << 8) +
            ((t & 0x000000ff) << 24) )
```

```
random.seed(10)

for i in range(5):
    print('%04.3f' % random.random())

0.571
0.429
0.578
0.206
0.813
```

## (1) pseudo random과 seed 번호

- 많은 프로그래밍 언어에서 사용하는 랜덤 함수는 의사 랜덤 pseudo random 생성기로 만들어 낸다.
- $X_n$ 은 의사 랜덤 값의 열(sequence)이 되며,  $X_{n+1}$  값은  $X_n$  값을 만드는데 사용될 수 있다

$$X_{n+1} = (aX_n + b) \bmod m$$

**X<sub>n+1</sub>은 X<sub>n</sub> 다음의 랜덤 값**

코드 7-6 : 의사 랜덤 함수를 이용한 난수 만들기

```
def pseudo_rand(x):
    a = 1103515245
    b = 12345
    m = 2 ** 31 # 4byte
    new_x = (a * x + b) % m
    return new_x

x = pseudo_rand(100)
print(x, sys.getsizeof(x))
x = pseudo_rand(101)
print(x, sys.getsizeof(x))
x = pseudo_rand(32)
print(x, sys.getsizeof(x))
x = pseudo_rand(45)
print(x, sys.getsizeof(x))
```

- pseudo\_rand()의 입력으로 100을 넣어주면 829870797와 같은 난수를 얻을 수 있음
- 입력 값을 101로 변경하면 1933386042와 같은 전혀 다른 난수 값이 나옴
- $m = 2^{39} \rightarrow 5\text{byte}$  랜덤수 생성

829870797 28  
1933386042 32  
952761817 28  
266074466 28

코드 7-7 : 의사 랜덤 함수를 이용한 연속적인 난수 만들기

```
def pseudo_rand(x):
    new_x = (1103515245 * x + 12345) % (2 ** 31)
    return new_x

x = 45 # seed 되는 초기 번호
number = []
for _ in range(15):
    x = pseudo_rand(x)
    number.append(x)

print(number)
```

[266074466, 175192819, 1025397680, 763982377, 2030324142, 2145030991, 11]

- 변수 a, b, m을 사용하지 않고 리터럴 상수를 이용하여 함수내부를 간단하게 수정
- 새롭게 얻어진 x 값을 다음 난수를 생성하는데 사용함
- 위 코드의 x와 같이 초기에 난수를 생성하기 위한 값을 seed number 혹은 씨드 값이라고 한다
- 씨드 값으로는 현재의 시간을 사용하면 규칙성이 덜하다

# random 모듈

[출처] 유틸 파일썬, “7장 모듈과 활용”

## (2) Lotto 번호 만들기

- 로또는 1에서 45 사이의 임의의 정수를 6개 맞히는 규칙이 있음
- 파이썬의 random 모듈을 활용하여 로또 번호를 자동으로 생성하여 출력하는 프로그램을 만들어보기

- 1에서 45까지의 번호를 리스트에 넣도록 하자.
- 이 리스트를 임의의 순서대로 섞도록 하자.
- 임의의 순서대로 섞은 리스트에서 최초 6개의 항목만 가져오자.
- 선택된 6개 항목들을 오름차순으로 정렬하자.
- 이 리스트를 출력하자.



코드 7-8 : random 모듈을 이용한 로또 번호 만들기 1

```
import random as rd

lotto_list = list(range(1, 46)) # 1부터 45까지 생성
rd.shuffle(lotto_list) # 임의의 순서로 섞기
lotto_list = lotto_list[:6] # 앞 부분 6개만 선택
lotto_list.sort() # 선택된 번호를 정렬
print('이번 주의 추천 로또번호 :', lotto_list)
```

이번 주의 추천 로또번호 : [2, 15, 18, 20, 25, 32]

코드 7-9 : random 모듈을 이용한 로또 번호 만들기 2

```
import random as rd

lotto_list = list(range(1, 46)) # 1부터 45까지 생성
lotto_list = rd.sample(lotto_list, 6) # 임의의 값 6개를 추출(샘플링)
lotto_list.sort() # 선택된 번호를 정렬
print('이번 주의 추천 로또번호 :', lotto_list)
```

이번 주의 추천 로또번호 : [8, 9, 12, 16, 21, 38]

- random 모듈의 추출(샘플링) 기능을 수행하는 sample() 함수를 사용하여 코드를 간략히 함

# sys 모듈

## sys 모듈

- 파이썬 인터프리터가 제공하는 변수들과 함수들을 직접 제어할 수 있게 해주는 모듈
- prefix 속성은 파이썬이 설치된 경로를 알려줌
- version 속성은 파이썬 인터프리터의 버전을 알려줌
- copyright 속성은 파이썬의 저작권에 관련된 내용을 포함
- sys 모듈 내의 속성들



```
Python 3.8.2 Shell
File Edit Shell Debug Options Window Help
(C) 1991-1995 Stichting Mathematisch Centrum, Amsterdam. All Rights Reserved.
>>> dir(sys)
['__breakpointhook__', '__displayhook__', '__doc__', '__excepthook__', '__interactivehook__', '__loader__', '__name__', '__package__', '__spec__', '__stderr__', '__stdin__', '__stdout__', '__unraisablehook__', '_base_executable', '_clear_type_cache', '_current_frames', '_debugmallocstats', '_enablelegacywindowsfsencoding', '_framework', '_getframe', '_git', '_home', '_xoptions', 'addaudithook', 'api_version', 'argv', 'audit', 'base_exec_prefix', 'base_prefix', 'breakpointhook', 'builtin_module_names', 'byteorder', 'call_tracing', 'callstats', 'copyright', 'displayhook', 'dllhandle', 'dont_write_bytecode', 'exc_info', 'excepthook', 'exec_prefix', 'executable', 'exit', 'flags', 'float_info', 'float_repr_style', 'get_asyncgen_hooks', 'get_coroutine_origin_tracking_depth', 'getallocatedblocks', 'getcheckinterval', 'getdefaultencoding', 'getfilesystemencodingerrors', 'getfilesystemencoding', 'getprofile', 'getrecursionlimit', 'getrefcount', 'getsizeof', 'getswitchinterval', 'gettrace', 'getwindowsversion', 'hash_info', 'hexversion', 'implementation', 'int_info', 'intern', 'is_finalizing', 'maxsize', 'maxunicode', 'meta_path', 'modules', 'path', 'path_hooks', 'path_importer_cache', 'platform', 'prefix', 'pycache_prefix', 'set_asyncgen_hooks', 'set_coroutine_origin_tracking_depth', 'setcheckinterval', 'setprofile', 'setrecursionlimit', 'setswitchinterval', 'settrace', 'stderr', 'stdin', 'stdout', 'thread_info', 'unraisablehook', 'version', 'version_info', 'warnoptions', 'winver']
>>>
```

[출처] 유틸 파일, “7장 모듈과 활용”

- sys 모듈을 이용한 경로 확인

```
import sys
print(sys.prefix)
print(sys.version)
print(sys.copyright)
```

```
C:\Users\sky\.conda\envs\python
3.8.11 (default, Aug 6 2021, 09:57:55) [MSC v.1916 64 bit (AMD64)]
Copyright (c) 2001-2021 Python Software Foundation.
All Rights Reserved.
```

```
Copyright (c) 2000 BeOpen.com.
All Rights Reserved.
```

```
Copyright (c) 1995-2001 Corporation for National Research Initiatives.
All Rights Reserved.
```

```
Copyright (c) 1991-1995 Stichting Mathematisch Centrum, Amsterdam.
All Rights Reserved.
```

- sys 모듈을 이용한 파이썬 경로 정보

```
sys.path
```

```
[C:\\\\Users\\\\sky\\\\Documents\\\\Python Scripts\\\\Python\\\\Basic',
'C:\\\\Users\\\\sky\\\\.conda\\\\envs\\\\python\\\\python38.zip',
'C:\\\\Users\\\\sky\\\\.conda\\\\envs\\\\python\\\\DLLs',
'C:\\\\Users\\\\sky\\\\.conda\\\\envs\\\\python\\\\lib',
'C:\\\\Users\\\\sky\\\\.conda\\\\envs\\\\python',
 '',
'C:\\\\Users\\\\sky\\\\.conda\\\\envs\\\\python\\\\lib\\\\site-packages',
'C:\\\\Users\\\\sky\\\\.conda\\\\envs\\\\python\\\\lib\\\\site-packages\\\\win32',
'C:\\\\Users\\\\sky\\\\.conda\\\\envs\\\\python\\\\lib\\\\site-packages\\\\win32\\\\lib',
'C:\\\\Users\\\\sky\\\\.conda\\\\envs\\\\python\\\\lib\\\\site-packages\\\\Pythonwin',
'C:\\\\Users\\\\sky\\\\.conda\\\\envs\\\\python\\\\lib\\\\site-packages\\\\IPython\\\\extensions',
'C:\\\\Users\\\\sky\\\\.ipython']
```

## Section04 모듈과 패키지

### ■ 패키지

- 모듈이 하나의 \*.py 파일 안에 함수가 여러 개 들어 있으면, 패키지 (Package)는 여러 모듈을 모아 놓은 것으로 폴더의 형태로 나타냄
- 모듈을 주제별로 분리할 때 주로 사용

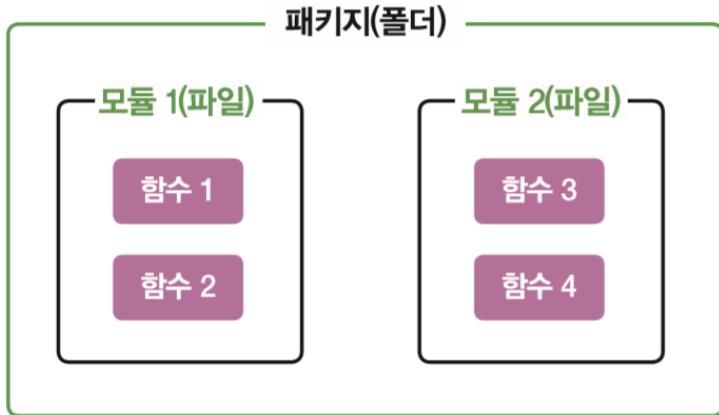


그림 9-11 패키지의 개념

### ▪ import 형식

```
from 패키지명.모듈명 import 함수명
```

### ▪ [그림 9-11]과 같은 형태로 구현 import

```
from package.Module1 import *
```

# 패키지 가져오기

[출처] <https://dojang.io/mod/page/view.php?id=2441>

## 1) import로 패키지 가져오기

- import 패키지.모듈
- import 패키지.모듈1, 패키지.모듈2
  - 패키지.모듈.변수(또는 함수(), 클래스())
  - 예제) 파이썬 표준 라이브러리에서 urllib 패키지의 request 모듈을 가져오기 (urllib은 URL 처리에 관련된 모듈을 모아 놓은 패키지).

```
import urllib.request  
response = urllib.request.urlopen('http://www.google.co.kr')  
response.status
```

200

### [Note] urlopen 함수

- urllib.request.urlopen은 URL을 여는 함수인데 URL 열기에 성공하면 response.status의 값이 200이 나온다. 이 200은 HTTP 상태 코드이며 웹 서버가 요청을 제대로 처리했다는 뜻이다.

## 2) import as로 패키지 모듈 이름 지정하기

- import 패키지.모듈 as 이름

```
import urllib.request as r  
# urllib 패키지의 request 모듈을 가져오면서 이름을 r로 지정  
  
response = r.urlopen('http://www.google.co.kr') # r로 urlopen 함수 사용  
response.status
```

200

# 패키지 가져오기

[출처] <https://dojang.io/mod/page/view.php?id=2441>

## 3) from import로 패키지의 일부만 가져오기

- from 패키지.모듈 import 변수 (또는 함수, 클래스)
- from 패키지.모듈 import 변수, 함수, 클래스

```
# urlopen 함수, Request 클래스를 가져옴
from urllib.request import Request, urlopen

# Request 클래스를 사용하여 req 생성
req = Request('http://www.google.co.kr')

# urlopen 함수 사용
response = urlopen(req)
response.status
```

200

## • 패키지의 모듈에서 모든 변수, 함수, 클래스를 가져오기

- from 패키지.모듈 import \*

```
# urllib의 request 모듈에서 모든 변수, 함수, 클래스를 가져옴
from urllib.request import *

req = Request('http://www.google.co.kr')
response = urlopen(req)
response.status
```

200

## 4) from import로 패키지의 모듈의 일부를 가져온 뒤 이름 지정하기

- from 패키지.모듈 import 변수 as 이름
- from 패키지.모듈 import 변수 as 이름, 함수 as 이름, 클래스 as 이름

```
from urllib.request import Request as r, urlopen as u
req = r('http://www.google.co.kr')      # r로 Request 클래스 사용
response = u(req)                      # u로 urlopen 함수 사용
response.status
```

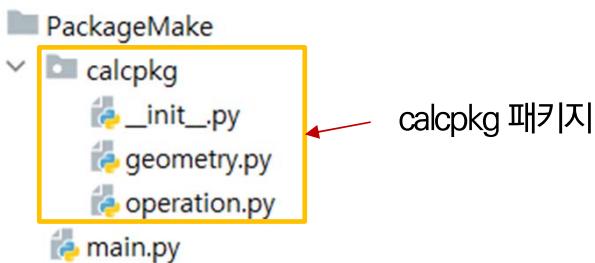
200

# 패키지 만들기

[참고] <https://dojang.io/mod/page/view.php?id=2447>

## 1) 패키지 만들기

- calc pkg 패키지에 geometry.py, operation.py 모듈 만들기



- calc pkg / \_\_init\_\_.py

✓ 폴더(디렉터리) 안에 \_\_init\_\_.py 파일이 있으면 해당 폴더는 패키지로 인식된다.  
그리고 기본적으로 \_\_init\_\_.py 파일의 내용은 비워 둘 수 있다(파이썬 3.3 이상  
부터는 \_\_init\_\_.py 파일이 없어도 패키지로 인식된다. 하지만 하위 버전에도 호  
환되도록 \_\_init\_\_.py 파일을 작성하는 것을 권장한다).

- calc pkg / operation.py
- calc pkg / geometry.py

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

def mul(a, b):
    return a * b

def triangle_area(base, height):
    return base * height / 2

def rectangle_area(width, height):
    return width * height
```

## 2) 패키지 이용하기

- main.py

```
import calc pkg.operation # calc pkg 패키지의 operation 모듈을 가져옴
import calc pkg.geometry # calc pkg 패키지의 geometry 모듈을 가져옴

def main():
    print(calc pkg.operation.add(10, 20))
    print(calc pkg.operation.mul(10, 20))

    print(calc pkg.geometry.triangle_area(30, 40))
    print(calc pkg.geometry.rectangle_area(30, 40))

if __name__ == '__main__':
    main()
```

30
200
600.0
1200

# 패키지 만들기

[참고] <https://dojang.io/mod/page/view.php?id=2447>

## 2) 패키지 이용하기 (2)

- from 패키지 import로 2개 모듈 동시에 import
- main.py

```
from calcpkg import operation, geometry

def main():
    print(operation.add(10, 20))
    print(operation.mul(10, 20))

    print(geometry.triangle_area(30, 40))
    print(geometry.rectangle_area(30, 40))

if __name__ == '__main__':
    main()
```

- ✓ sys 모듈의 path 변수에는 모듈, 패키지를 찾는 경로가 들어있다. 여기서 site-packages 폴더에는 pip로 설치한 패키지가 들어간다. 그리고 자기가 만든 모듈, 패키지도 site-packages 폴더에 넣으면 스크립트 파일이 어디에 있든 모듈, 패키지를 사용할 수 있다.
- ✓ 만약 가상 환경(virtual environment)를 만들어서 모듈과 패키지를 관리한다면 가상 환경/Lib/site-packages 폴더에 모듈과 패키지가 들어간다

### [Note] 패키지의 모듈과 \_\_name\_\_

- 패키지의 모듈에서는 \_\_name\_\_ 변수에 패키지 모듈 형식으로 이름이 들어간다. 즉, calcpkg 패키지의 geometry.py에서 \_\_name\_\_의 값을 출력하도록 만들고, import로 가져오면 'calcpkg.geometry' 가 나온다.

### [Note] 모듈과 패키지를 찾는 경로

- 파이썬에서는 현재 폴더에 모듈, 패키지가 없으면 다음 경로에서 모듈, 패키지를 찾는다.

```
import sys
sys.path
```

```
[ 'C:\\\\Users\\\\sky\\\\Documents\\\\Python Scripts\\\\Python\\\\Basic',
  'C:\\\\Users\\\\sky\\\\.conda\\\\envs\\\\python\\\\python38.zip',
  'C:\\\\Users\\\\sky\\\\.conda\\\\envs\\\\python\\\\DLLs',
  'C:\\\\Users\\\\sky\\\\.conda\\\\envs\\\\python\\\\lib',
  'C:\\\\Users\\\\sky\\\\.conda\\\\envs\\\\python',
  '',
  'C:\\\\Users\\\\sky\\\\.conda\\\\envs\\\\python\\\\lib\\\\site-packages',
  'C:\\\\Users\\\\sky\\\\.conda\\\\envs\\\\python\\\\lib\\\\site-packages\\\\win32',
  'C:\\\\Users\\\\sky\\\\.conda\\\\envs\\\\python\\\\lib\\\\site-packages\\\\win32\\\\lib',
  'C:\\\\Users\\\\sky\\\\.conda\\\\envs\\\\python\\\\lib\\\\site-packages\\\\Pythonwin',
  'C:\\\\Users\\\\sky\\\\.conda\\\\envs\\\\python\\\\lib\\\\site-packages\\\\IPython\\\\extensions',
  'C:\\\\Users\\\\sky\\\\.ipython']
```

## Section05 함수의 심화 내용

### ■ 내부 함수, lambda, map()

- **내부 함수** : 함수 안에 함수가 있는 형태

```
def outFunc(v1, v2) :  
    def inFunc(num1, num2) :  
        return num1 + num2  
    return inFunc(v1, v2)  
print(outFunc(10, 20))
```

출력 결과

30

- outFunc() 함수 밖에서 호출하면 오류 발생

```
print(inFunc(10, 20))
```

출력 결과

```
NameError: name 'inFunc' is not defined
```

## Section05 함수의 심화 내용

- **lambda** 함수 : 함수를 한 줄로 간단하게 만들어 줌

- **lambda 매개변수1, 매개변수2, ... : 매개변수를 이용한 표현식**

```
def hap(num1, num2) :  
    res = num1 + num2  
    return res  
print(hap(10, 20))
```

출력 결과

30

```
hap2 = lambda num1, num2 : num1 + num2  
print(hap2(10, 20))
```

- 매개변수에 기본값을 설정

```
hap3 = lambda num1 = 10, num2 = 20 : num1 + num2  
print(hap3())  
print(hap3(100, 200))
```

출력 결과

30

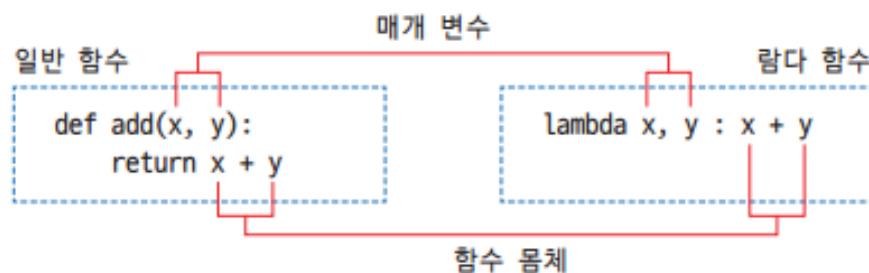
300

- 매개변수를 지정하지 않으면 기본값으로 설정, 매개변수를 넘겨주면 기본값 무시

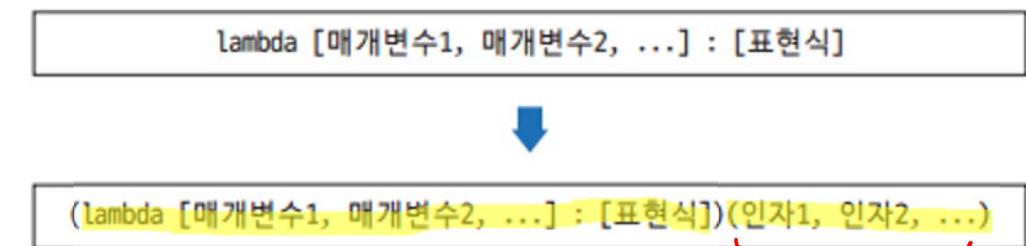
## 람다(lambda) 함수

[출처] 유틸 파이썬, “10장 lambda 함수와 list comprehension (리스트 축약)”

- 람다 함수**lambda function**
  - 1회용의 간단한 함수를 만드는 것
- 람다 표현식**lambda expression**이라고도 불리우는 이름이 없는 함수
- 익명 함수**anonymous function**



일반 함수와 람다 함수: 일반 함수는 def 키워드와 함수명, 매개변수, 콜론, 그리고 몸체로 구성되어 있으나 람다 함수는 매개변수와 함수의 기능만을 가진다.



[그림 10-2] 람다 함수의 정의 방법과 인자 전달 방법

- 람다 함수의 문법은 매우 간결하고 단순하다

# 람다(lambda) 함수

[출처] 으뜸 파이썬, “10장 lambda 함수와 list comprehension (리스트 축약)”

코드 10-1 : add() 함수를 이용한 정수의 덧셈과 결과 반환

function\_add.py

# add() 함수를 이용한 덧셈

```
def add(x, y):
```

```
    return x + y
```

```
print('100과 200의 합 :', add(100, 200))
```

실행결과

100과 200의 합 : 300

## NOTE : 람다 함수의 장점과 단점

- 장점: 람다 함수는 축약된 표현이 가능하기 때문에 코드가 간결해 진다. 일반 함수의 경우 재사용을 위하여 함수를 위한 이름과 이 메모리 공간의 할당이 필요하지만 람다 함수는 메모리 공간을 별도로 할당하지 않기 때문에 메모리를 절약할 수 있다. 즉 람다 함수는 익명 함수이므로 표현식 내에서 사용된 후 다음 코드로 이동하면 힙 메모리에서 사라지게 된다.
- 단점: lambda 선언문 뒤에 쉼표와 콜론 수식으로 구성되어 가독성이 떨어진다. 즉 람다식은 그 자체에 쉼표(,)를 포함하고 있기 때문에 다른 매개변수와 시각적인 구분이 모호해지고, 읽고 이해하기가 어렵다.

코드 10-2 : 람다 함수를 정의하고 add라는 변수를 통해 참조하기

lambda\_add.py

# 람다 함수를 이용한 덧셈

```
add = lambda x, y: x + y
```

```
print('100과 200의 합 :', add(100, 200))
```

실행결과

100과 200의 합 : 300

코드 10-3 : 인라인 람다 함수와 인자를 이용한 덧셈

inline\_lambda\_add.py

```
print('100과 200의 합 :', (lambda x, y: x + y)(100, 200))
```

위와 같이 별도의 라인으로 람다함수를 정의하지 않고  
호출문 내에 람다 함수의 기능을 넣을 수 있다  
이를 인라인 람다 함수라 한다

## Section05 함수의 심화 내용

### ➤ map() 함수

출처: <https://offbyone.tistory.com/73>

- lambda 함수의 장점은 map() 함수와 함께 사용될 때 볼 수 있음.
- map() 은 두 개의 인수를 가지는 함수임

`r = map(function, iterable, ...)`

- ✓ 첫 번째 인자 function 는 함수명
- ✓ 두 번째 인자 iterable은 한번에 하나의 멤버를 반환할 수 있는 객체  
(list, str, tuple)
- ✓ map() 함수는 function을 iterable의 모든 요소에 대해 적용하고  
function에 의해 변경된 iterator를 반환

```
>>> a = [1,2,3,4]
>>> b = [17,12,11,10]
>>> list(map(lambda x, y:x+y, a,b))
[18, 14, 14, 14]
```

## Section05 함수의 심화 내용

### ■ 리스트에 모두 10을 더하는 코드

```
myList = [1, 2, 3, 4, 5]
def add10(num) :
    return num + 10

for i in range(len(myList)) :
    myList[i] = add10(myList[i])
print(myList)
```

#### 출력 결과

```
[11, 12, 13, 14, 15]
```

- lambda 함수와 map() 함수로 간단히

```
1 myList = [1, 2, 3, 4, 5]
2 add10 = lambda num : num + 10
3 myList = list(map(add10, myList))
4 print(myList)
```

### ■ 2행과 3행을 합친 코드

```
myList = [1, 2, 3, 4, 5]
myList = list(map(lambda num : num + 10, myList))
print(myList)
```

### ■ 두 리스트의 각 자릿수를 합쳐서 새로운 리스트로 만들기

```
list1 = [1, 2, 3, 4]
list2 = [10, 20, 30, 40]
hapList = list(map(lambda n1, n2 : n1 + n2, list1, list2))
print(hapList)
```

#### 출력 결과

```
[11, 22, 33, 44]
```

```
list1 = [1, 2, 3, 4]
list2 = [10, 20, 30, 40]

haplist = list1 + list2
print(haplist)

hap = lambda n1, n2 : n1+n2
print(hap(list1, list2))
print(list(map(hap, list1, list2)))

haplist = list(map(lambda n1, n2 : n1+n2, list1, list2))
print(haplist)
```

```
[1, 2, 3, 4, 10, 20, 30, 40]
[1, 2, 3, 4, 10, 20, 30, 40]
[11, 22, 33, 44]
[11, 22, 33, 44]
```

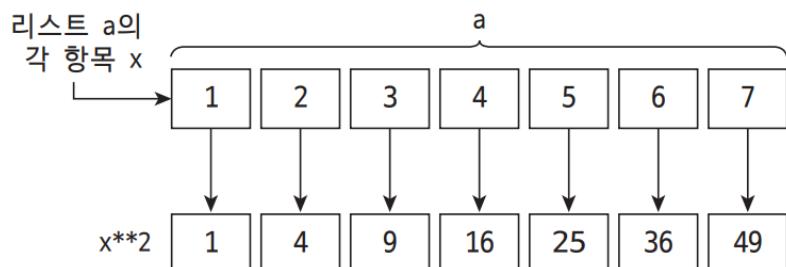
# 맵(map) 함수

코드 10-10 : 맵 함수를 이용한 제곱값 리스트 생성

```
square_map_for.py  
a = [1, 2, 3, 4, 5, 6, 7]  
square_a = []  
for n in a:  
    square_a.append(n**2) # n의 제곱을 square_a 리스트에 추가  
  
print(square_a)
```

실행결과

```
[1, 4, 9, 16, 25, 36, 49]
```



[그림 10-5] 원본 리스트 a와 각 항목 x에 대한 제곱 연산이 적용된 리스트

[출처] 유틸 파이썬, “10장 lambda 함수와 list comprehension (리스트 축약)”

- [코드 10-10]은 map() 함수와 square() 함수를 사용하면 다음 [코드 10-11]과 같이 간단하게 수정할 수 있다.

map(적용시킬\_함수, 반복가능\_객체, ...)

코드 10-11 : 맵 함수와 square() 호출을 이용한 제곱값 리스트 생성

```
square_map_func.py
```

```
def square(x):  
    return x ** 2
```

```
a = [1, 2, 3, 4, 5, 6, 7]
```

```
# a의 각 항목에 대해 square 함수의 반환값을 적용
```

```
square_a = list(map(square, a))
```

```
print(square_a)
```

실행결과

```
[1, 4, 9, 16, 25, 36, 49]
```

## 맵(map) 함수

[출처] 으뜸 파이썬, “10장 lambda 함수와 list comprehension (리스트 축약)”

코드 10-12 : 맵 함수와 람다 함수를 이용한 제곱값 리스트 생성

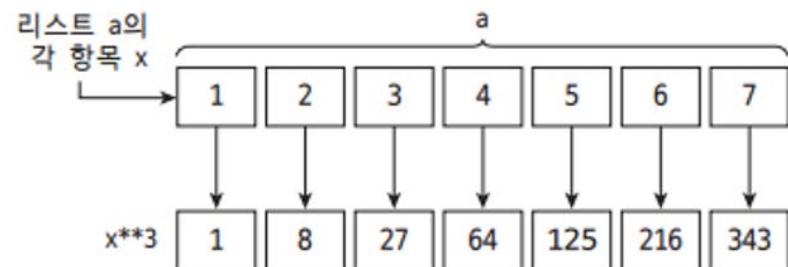
square\_map\_lambda.py

```
a = [1, 2, 3, 4, 5, 6, 7]
square_a = list(map(lambda x: x**2, a))
print(square_a)
```

실행결과

```
[1, 4, 9, 16, 25, 36, 49]
```

```
a = [1, 2, 3, 4, 5, 6, 7]
square_a = list(map(lambda x: x**2, a))
cubic_a = list(map(lambda x: x**3, a))
```



[그림 10-6] 람다 함수를 사용한 매핑 함수의 동작

- map() 함수와 램다 함수를 사용하면 간결한 표현식으로 리스트의 요소들을 변환할 수 있다.

# 맵(map) 함수

[출처] 으뜸 파이썬, “10장 lambda 함수와 list comprehension (리스트 축약)”



## LAB 10-3 : map() 함수의 응용

- ['a', 'b', 'c', 'd']와 같은 영문 소문자가 들어있는 a\_list라는 이름의 리스트를 ['A', 'B', 'C', 'D']와 같이 영문 대문자가 들어있는 upper\_a\_list라는 이름의 리스트로 변환시키는 map() 함수를 작성하여라.

```
upper_a_list = ['A', 'B', 'C', 'D']
```

- 1) 이때 소문자를 매개변수로 받아 대문자를 반환하는 to\_upper()라는 이름의 함수를 정의하여 변환시키시오.
- 2) 1)에서 생성한 to\_upper() 함수를 lambda 함수로 고쳐서 간결하게 변환시키시오(힌트 : str의 upper() 메소드를 사용해 보시오.).
2. [10, 20, 30]과 같은 값을 가진 n\_list라는 이름의 리스트를 map() 함수에 넣은 후의 결과는 각각 다음과 같다.

입력 값의 두 배 : [20, 40, 60]

입력 값의 세 배 : [30, 60, 90]

- 1) twice()와 triple()이라는 이름의 함수를 만들어서 각각 입력 값의 2배와 3배 큰 값을 반환하도록 하시오. 그리고 map() 함수를 이용하여 위의 결과와 같이 나오도록 하시오.
- 2) twice()와 triple() 함수를 lambda 함수로 고쳐서 간결하게 변환시키시오.

## Section05 함수의 심화 내용

### ■ 재귀 함수

☞ **reduce** 함수 예제 참조

- 자신이 자신을 호출

```
def selfCall() :  
    print('하', end = ' ')  
    selfCall()  
selfCall()
```

출력 결과

하하하하하하하하…

- 입력한 숫자를 1까지 세는 함수를 재귀 함수

```
def count(num) :  
    if num >= 1 :  
        print(num, end = ' ')  
        count(num - 1)  
    else :  
        return  
count(10)  
count(20)
```

출력 결과

10 9 8 7 6 5 4 3 2 1  
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

- Factorial 값을 구하는 함수

```
def factorial(num) :  
    if num <= 1 :  
        return num  
    else :  
        return num * factorial(num - 1)  
print(factorial(4))  
print(factorial(10))
```

출력 결과

24  
3628800

## 필터(filter) 함수

[출처] 유틸 파일썬, “10장 lambda 함수와 list comprehension (리스트 축약)”

코드 10-4 : 19 이상의 값을 반환하는 `adult_func()` 함수와 필터 함수의 사용

```
age_filter.py  
# 19세 이상의 값이 들어오면 True, 그렇지 않으면 False를 반환  
  
def adult_func(n):  
    if n >= 19:  
        return True  
    else:  
        return False
```

```
ages = [34, 39, 20, 18, 13, 54]  
print('성년 리스트 :')  
for a in filter(adult_func, ages): # filter() 함수를 사용한 ages의 필터링  
    print(a, end = ' ')
```

실행결과

성년 리스트 :  
34 39 20 54

코드 10-5 : 람다 함수를 이용한 간략화된 필터

```
age_lambda_filter.py  
ages = [34, 39, 20, 18, 13, 54]  
  
print('성년 리스트 :')  
for a in filter(lambda x: x >= 19, ages): # filter() 함수를 사용한 ages의 필터  
    print(a, end = ' ')
```

실행결과

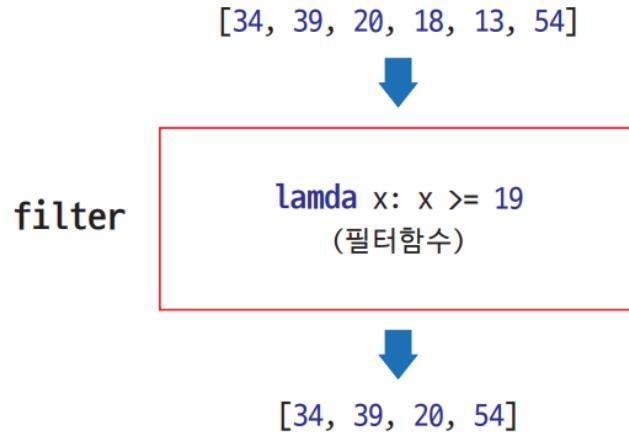
성년 리스트 :  
34 39 20 54

코드 10-4 보다 간결한 표현방법

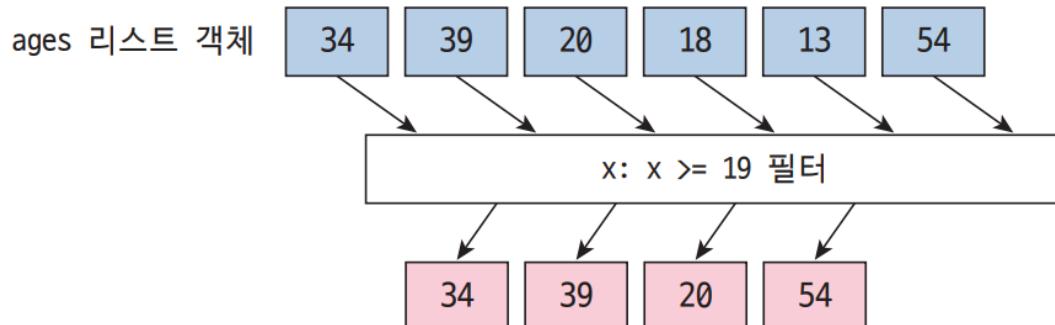
1. 알고리즘이 비교적 단순하다.
2. 별도의 내부 변수가 필요 없다.
3. 성년 리스트를 필터링한 후 더 이상 재사용할 필요가 없다.

## 필터(filter) 함수

[출처] 으뜸 파이썬, “10장 lambda 함수와 list comprehension (리스트 축약)”



[그림 10-3] filter() 함수의 동작과 람다 함수의 역할



[그림 10-4] 반복 가능한 객체와 필터 함수에 의해 변환된 반복자 객체

코드 10-7 : 람다 함수를 이용한 음수 값 추출기능 1

minus\_filter\_func.py

```
def minus_func(n): # n이 음수이면 True, 그렇지 않으면 False를 반환
    if n < 0 :
        return True
    else:
        return False

n_list = [-30, 45, -5, -90, 20, 53, 77, -36]
minus_list = []      # 음수를 저장할 리스트
for n in filter(minus_func, n_list):
    minus_list.append(n)
print('음수 리스트 : ', minus_list)
```

import random

```
rlist = random.sample(range(1,101), 10)
evenlist = list(filter(lambda x: x%2==0, rlist))
print(rlist)
print(evenlist)
```

```
[51, 13, 21, 10, 64, 77, 66, 69, 67, 48]
[10, 64, 66, 48]
```

## 필터(filter) 함수

코드 10-8 : 람다 함수를 이용한 음수 값 추출기능 2

```
lambda_filter_minus1.py
```

```
n_list = [-30, 45, -5, -90, 20, 53, 77, -36]
minus_list = []
for n in filter(lambda x: x < 0, n_list):
    minus_list.append(n)

print('음수 리스트 :', minus_list)
```

실행결과

음수 리스트 : [-30, -5, -90, -36]

코드 10-9 : 람다 함수를 이용한 음수 값 추출기능 3

```
lambda_filter_minus2.py
```

```
n_list = [-30, 45, -5, -90, 20, 53, 77, -36]
minus_list = list(filter(lambda x: x < 0, n_list))
print('음수 리스트 :', minus_list)
```

[출처] 유틸 파일썬, “10장 lambda 함수와 list comprehension (리스트 축약)”



### LAB 10-2 : 람다 함수의 응용

- 정수 요소 값을 가진 `n_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`라는 리스트가 있다. `n_list`에서 짝수 값 항목만을 가진 `even_list`라는 리스트를 람다 함수를 이용하여 아래와 같이 생성해 보자.

```
even_list = [2, 4, 6, 8, 10]
```

- 이때 `even_list`라는 빈 리스트를 만든 후 `append()` 메소드를 이용하여 짝수 값 항목을 추가해 보시오.
- `filter()` 함수가 반환하는 객체를 `list()` 함수를 통하여 리스트 객체로 만들어서 `even_list`에 할당하시오.

- 1번 문제에서 사용한 `n_list` 리스트에 람다 함수를 이용하여 홀수 값 항목만을 가진 다음과 같은 `odd_list`를 생성해 보자.

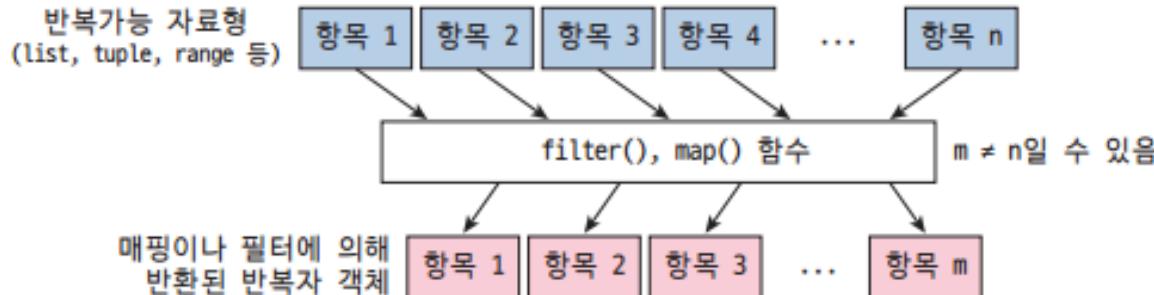
```
odd_list = [1, 3, 5, 7, 9]
```

- 이때 `odd_list`라는 빈 리스트를 만든 후 `append()` 메소드를 이용하여 홀수 값 항목을 추가해 보시오.
- `filter()` 함수가 반환하는 객체를 `list()` 함수를 통하여 리스트 객체로 만들어서 `odd_list`에 할당하시오.

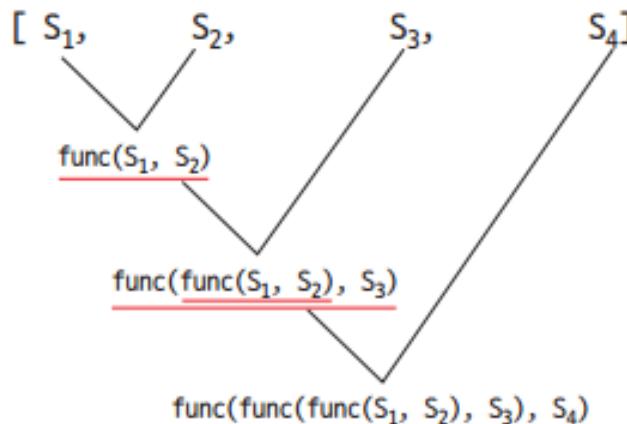
## reduce 함수

[출처] 으뜸 파이썬, “10장 lambda 함수와 list comprehension (리스트 축약)”

[출처] <https://codepractice.tistory.com/86>



[그림 10-7] 반복가능한 자료형과 맵핑 함수에 의해 변환된 반복자 객체



[그림 10-8] reduce() 함수의 동작

- functools 모듈의 reduce 함수는 다음과 같다.

```
def reduce(function, iterable, initializer=None):  
    it = iter(iterable)  
    if initializer is None:  
        value = next(it)  
    else:  
        value = initializer  
    for element in it:  
        value = function(value, element)  
    return value
```

- reduce 함수는 매개 변수로 function, iterable[, initializer]를 갖는다. function 과 iterable 은 반드시 전달되어야 하고, initializer 는 선택적이라고 보면 된다.
- reduce 함수는 iterable 의 요소들을 function 에 대입하여 하나의 결과값을 리턴해 주는 함수이다.
- 만약 function 에는 func 가 iterable 에는 [a<sub>1</sub>, a<sub>2</sub>, a<sub>3</sub>, a<sub>4</sub>] 가 전달되었다고 하면, reduce 함수는 다음의 결과를 리턴한다. func( func( func( a<sub>1</sub>, a<sub>2</sub> ), a<sub>3</sub> ), a<sub>4</sub> )

## reduce 함수

- reduce 함수 예제

```
def reduce(function, iterable, initializer=None):
    it = iter(iterable)
    if initializer is None:
        value = next(it)
    else:
        value = initializer
    for element in it:
        value = function(value, element)
    return value
```

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

reduce(sum, [1, 2, 3, 4, 5])
```

15

```
from functools import reduce

result1 = reduce(lambda x, y: x + y, ['2', '4', '6', '8', '10', '12', '14'])
result2 = reduce(lambda x, y: x + y, ['2', '4', '6', '8', '10', '12', '14'], 'add : ')

print(result1 + "\t" + result2)
```

2468101214      add : 2468101214

from functools import reduce

[출처] 으뜸 파이썬, “10장 lambda 함수와 list comprehension (리스트 축약)”

```
reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])
```

15

```
reduce(lambda x, y: x+y, [1, 2, 3, 4, 5], 10)
```

25

```
reduce(lambda x, y: x*y, range(1, 6))
```

120

```
def factorial(n):
    return reduce(lambda x, y: x*y, range(1, n+1))

factorial(6)
```

720

# list comprehension

- 리스트 축약 표현 **list comprehension**의 문법은 다음과 같다

```
[ {표현식} for {변수} in {반복자/연속열} if {조건 표현식} ]
```

- 리스트 축약 문법에서 다음과 같이 if 조건 표현식은 생략 가능하다

```
[ {표현식} for {변수} in {반복자/연속열} ]
```

코드 10-18 : 문자열 각각을 대문자로 바꾸는 기능

```
upper_chars_from_string.py
```

```
st = 'Hello World'  
s_list = [x.upper() for x in st] # 문자열 각각에 대해 upper() 메소드 적용  
print(s_list)
```

실행결과

```
['H', 'E', 'L', 'L', 'O', ' ', 'W', 'O', 'R', 'L', 'D']
```

[출처] 유틸 파일썬, “10장 lambda 함수와 list comprehension (리스트 축약) ”

코드 10-15 : 맵과 람다 함수를 이용한 리스트의 제곱 구하기

```
list_map_lambda_square.py
```

```
a = [1, 2, 3, 4, 5, 6, 7] # 연속된 값을 가지는 리스트  
a = list(map(lambda x: x**2, a)) # 리스트의 각 요소에 대하여 람다 함수 적용  
print(a)
```

코드 10-16 : 리스트 축약 표현식을 이용한 리스트의 제곱 구하기

```
list_comp_square.py
```

```
a = [1, 2, 3, 4, 5, 6, 7] # 연속된 값을 가지는 리스트  
a = [x**2 for x in a] # 리스트의 각 요소에 대하여 x**2를 적용  
print(a)
```

코드 10-17 : 리스트 축약 표현식과 range를 이용한 리스트의 제곱 구하기

```
list_comp_range_square.py
```

```
a = [x**2 for x in range(1, 8)]  
print(a)
```

- 세 코드의 실행 결과는 다음과 같아야 한다.

실행결과

```
[1, 4, 9, 16, 25, 36, 49]
```

# list comprehension

## ➤ if 조건식을 이용한 필터링

### 코드 10-19 : list() 함수, filter() 함수, 람다 함수를 이용한 필터링

```
age_lambda_filter_list.py
```

```
ages = [34, 39, 20, 18, 13, 54]
adult_ages = list(filter(lambda x: x >= 19, ages))
print('성년 리스트 :', adult_ages)
```

- 위의 코드는 다음 [코드 10-20]과 같은 리스트 축약 표현을 사용할 수 있다.(결과는 위와 동일함)

### 코드 10-20 : 리스트 축약표현을 이용한 필터링

```
age_list_comp_filter.py
```

```
ages = [34, 39, 20, 18, 13, 54]
print('성년 리스트 :', [x for x in ages if x >= 19])
```

실행결과

성년 리스트 : [34, 39, 20, 54]

[출처] 유틸 파일, “10장 lambda 함수와 list comprehension (리스트 축약) ”

### 코드 10-21 : list() 함수, filter() 함수, 람다 함수를 이용한 필터링

```
lambda_filter_minus2.py
```

```
n_list = [-30, 45, -5, -90, 20, 53, 77, -36]
minus_list = list(filter(lambda x: x < 0, n_list))
print('음수 리스트 :', minus_list)
```

- 위의 코드는 다음 [코드 10-20]과 같은 리스트 축약 표현을 사용하여 동일한 결과를 얻을 수 있다.

### 코드 10-22 : 리스트 축약 표현과 if 조건식을 이용한 필터링

```
list_comp_if_condition.py
```

```
n_list = [-30, 45, -5, -90, 20, 53, 77, -36]
minus_list = [x for x in n_list if x < 0]
print('음수 리스트 :', minus_list)
```

실행결과

음수 리스트 : [-30, -5, -90, -36]

## list comprehension

[출처] 유틸 파이썬, “10장 lambda 함수와 list comprehension (리스트 축약)”

- 필터링 된 값에 대해  $x * x$ 를 통해 각 값의 제곱을 구한 후 이 값을 리스트에 넣는 과정이다

대화창 실습 : 리스트의 축약 표현 실습

```
>>> [x for x in range(10)]      # 0에서 9까지 숫자를 포함하는 리스트  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  
  
>>> [x * x for x in range(10)]    # 0에서 9까지 숫자의 제곱 값  
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]  
  
>>> [x for x in range(10) if x % 2 == 0]  # 0에서 9까지 숫자 중 짝수 값  
[0, 2, 4, 6, 8]  
  
>>> [x for x in range(10) if x % 2 == 1]  # 0에서 9까지 숫자 중 홀수 값  
[1, 3, 5, 7, 9]  
  
>>> [x * x for x in range(10) if x % 2 == 0] # 0에서 9까지 숫자 중 짝수의 제곱 값  
[0, 4, 16, 36, 64]  
  
>>> [x * x for x in range(10) if x % 2 == 1] # 0에서 9까지 숫자 중 홀수의 제곱 값  
[1, 9, 25, 49, 81]
```

- 여러 개의 정수 문자열을 입력받아 이를 int() 함수를 사용해서 정수형 값을 가진 리스트로 변환하는 것도 손쉽게 할 수 있다.

대화창 실습 : 여러 개의 정수를 리스트로 입력받는 방법

```
>>> s = input('정수 5개를 입력하세요 : ').split()  
정수 5개를 입력하세요 : 10 20 30 40 50  
>>> lst = [int(x) for x in s]  
>>> lst  
[10, 20, 30, 40, 50]  
  
>>> [int(x) for x in input('정수를 여러개 입력하세요 : ').split()]  
정수를 여러개 입력하세요 : 1 2 3  
[1, 2, 3]
```

## list comprehension

[출처] 유틸 파이썬, “10장 lambda 함수와 list comprehension (리스트 축약) ”

코드 10-23 : 두 리스트를 곱하여 새 리스트를 생성하는 코드

product\_xy\_for\_loop.py

```
product_xy = []
for x in [1, 2, 3]: # 이중 for 루프를 통해 두 리스트 원소의 곱을 모두 구함
    for y in [2, 4, 6]:
        product_xy.append(x * y)
print(product_xy)
```

코드 10-24 : 리스트 축약을 이용한 두 리스트의 곱하기 기능

product\_xy\_comprehension.py

```
product_xy = [x * y for x in [1, 2, 3] for y in [2, 4, 6]]
print(product_xy)
```

리스트 축약 표현을 이용하여  
한 줄의 코딩으로 표현 가능

실행결과

```
[2, 4, 6, 4, 8, 12, 6, 12, 18]
```

대화창 실습 : 리스트의 축약 표현을 사용한 2와 3의 배수 구하기

```
>>> [n for n in range(1, 31) if n % 2 == 0 if n % 3 == 0]
[6, 12, 18, 24, 30]
```

필요에 따라서 더 많은 if를 추가하여 조  
건을 더 줄 수 있다(1에서 30까지의 수  
중에서 2, 3, 5의 배수 구하기)

대화창 실습 : 리스트의 축약 표현을 사용한 2와 3과 5의 배수 구하기

```
>>> [n for n in range(1, 31) if n % 2 == 0 if n % 3 == 0 if n % 5 == 0]
[30]
```

```
[n for n in range(1,31) if (n%2 == 0) and (n%3 == 0)]
```

```
[6, 12, 18, 24, 30]
```

```
[n for n in range(1,31) if (n%2 == 0) or (n%3 == 0)]
```

```
[2, 3, 4, 6, 8, 9, 10, 12, 14, 15, 16, 18, 20, 21, 22, 24,
26, 27, 28, 30]
```

## ➤ Iterable 객체

- iterable 객체 – 반복 가능한 객체
  - 대표적으로 iterable한 타입 – list, dict, set, str, bytes, tuple, range
  - iterable한 타입을 확인하는 방법
- ✓ collections.Iterable에 속한 instance인지 확인 : isinstance 함수는 첫번째 파라미터, 두번째 파라미터 클래스의 instance이면 True를 반환함

```
import collections.abc

# iterable 타입
var_list = [1, 3, 5, 7]
print(isinstance(var_list, collections.Iterable))

var_dict = {"a": 1, "b":1}
print(isinstance(var_dict, collections.Iterable))

var_set = {1, 3}
print(isinstance(var_set, collections.Iterable))

var_str = "abc"
print(isinstance(var_str, collections.Iterable))

var_bytes = b'abcdef'
print(isinstance(var_bytes, collections.Iterable))

var_tuple = (1, 3, 5, 7)
print(isinstance(var_bytes, collections.Iterable))

var_range = range(0,5)
print(isinstance(var_range, collections.Iterable))
```

python 3.3 이후  
collections.abc.Iterable 사용

True  
True  
True  
True  
True  
True  
True  
True

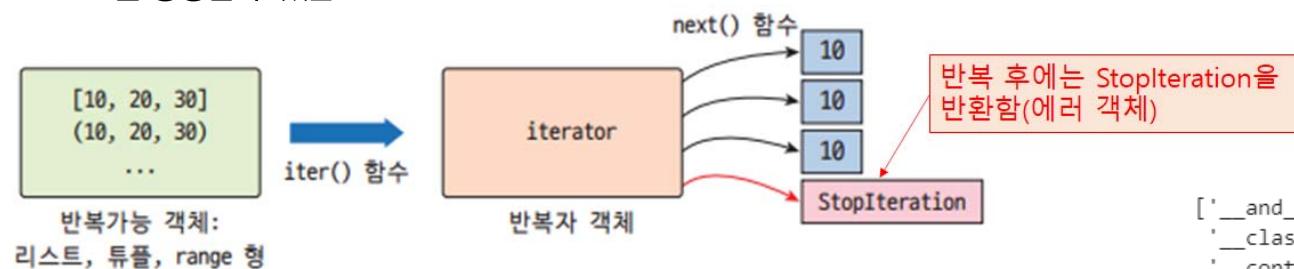
```
var_list = [1, 2, 3]
int_value = 10
print(iter(var_list))
print(iter(int_value))
```

```
<list_iterator object at 0x00000206850FF7F0>
-----
TypeError                                         Tr
~\AppData\Local\Temp\ipykernel_15708/1804861
    2 int_value = 10
    3 print(iter(var_list))
----> 4 print(iter(int_value))

TypeError: 'int' object is not iterable
```

## ➤ Iterator 객체

- iterator 객체 – 하나 이상의 항목이 포함되어 있는 자료구조에서 데이터를 차례대로 꺼낼 수 있는 객체
- iterator는 iterable한 객체를 내장함수 또는 iterable 객체의 메소드로 객체를 생성할 수 있음



- iterable 객체는 매직메서드 `__iter__` 메소드를 가지고 있음. 이 메소드로 iterator를 만들어 보자

`b.__iter__()` 또는 `iter(b)`

```
b = {1, 2, 3}
dir(b)
```

```
b_iter = b.__iter__()
type(b_iter)
```

set\_iterator

<code>'__and__'</code> ,	<code>'__le__'</code> ,	<code>'add'</code> ,
<code>'__class__'</code> ,	<code>'__len__'</code> ,	<code>'clear'</code> ,
<code>'__contains__'</code> ,	<code>'__lt__'</code> ,	<code>'copy'</code> ,
<code>'__delattr__'</code> ,	<code>'__ne__'</code> ,	<code>'difference'</code> ,
<code>'__dir__'</code> ,	<code>'__new__'</code> ,	<code>'difference_update'</code> ,
<code>'__doc__'</code> ,	<code>'__or__'</code> ,	<code>'discard'</code> ,
<code>'__eq__'</code> ,	<code>'__rand__'</code> ,	<code>'intersection'</code> ,
<code>'__format__'</code> ,	<code>'__reduce__'</code> ,	<code>'intersection_update'</code> ,
<code>'__ge__'</code> ,	<code>'__reduce_ex__'</code> ,	<code>'isdisjoint'</code> ,
<code>'__getattribute__'</code> ,	<code>'__repr__'</code> ,	<code>'issubset'</code> ,
<code>'__gt__'</code> ,	<code>'__ror__'</code> ,	<code>'issuperset'</code> ,
<code>'__hash__'</code> ,	<code>'__rsub__'</code> ,	<code>'pop'</code> ,
<code>'__iand__'</code> ,	<code>'__rxor__'</code> ,	<code>'remove'</code> ,
<code>'__init__'</code> ,	<code>'__setattr__'</code> ,	<code>'symmetric_difference'</code> ,
<code>'__init_subclass__'</code> ,	<code>'__sizeof__'</code> ,	<code>'symmetric_difference_update'</code> ,
<code>'__ior__'</code> ,	<code>'__str__'</code> ,	<code>'union'</code> ,
<code>'__isub__'</code> ,	<code>'__sub__'</code> ,	<code>'update'</code>
<code>'__iter__'</code> ,	<code>'__subclasshook__'</code> ,	
<code>'__ixor__'</code> ,	<code>'__xor__'</code> ,	

```
a = [1, 2, 3]
a_iter = iter(a)
type(a_iter)
```

list\_iterator

```
n = 10          반복 불가능 자료형 : 변환 불가능
n_iter = iter(n)
```

```
-----  
TypeError
~\AppData\Local\Temp\ipykernel_15708/269:
  1 n = 10
----> 2 n_iter = iter(n)

TypeError: 'int' object is not iterable
```

# 반복자 iterator

[출처] <https://wikidocs.net/16068>

## ➤ Iterator

- iterator 객체는 `next` 내장 함수를 사용할 때마다 첫번째, 두번째, 세번째 값이 출력된다.
- 네번째 실행에서는 `StopIteration`이라는 예외가 발생함.
- iterator 매직 메소드 '`__next__`'를 통해 하나씩 값을 꺼내보자

```
a = [1, 2, 3]
a_iter = iter(a)

for _ in range(0,4):
    print(next(a_iter))
```

```
1
2
3
```

```
-----
StopIteration
~\AppData\Local\Temp\ipykernel_15708/376
    3
    4 for _ in range(0,4):
-----> 5     print(next(a_iter))

StopIteration:
```

```
a = [1, 2, 3]
a_iter = iter(a)

for i in a_iter:
    print(i)
```

```
1
2
3
```

```
b = [1, 2, 3]
b_iter = b.__iter__()
#b_iter = iter(b)
type(b_iter)
```

list\_iterator

```
for _ in range(len(b)):
    #print(next(b_iter))
    print(b_iter.__next__())
```

```
1
2
3
```

대화창 실습 : `next()` 함수를 사용한 반복자 객체의 요소 추출

```
>>> lst = [10, 20, 30]
>>> l_iter = iter(lst) # 리스트 형 객체를 반복자로 만들
>>> type(l_iter)
<class 'list_iterator'>
>>> next(l_iter) ← next () 내장 함수를 사용하여 순회 함
10
>>> next(l_iter)
20
>>> next(l_iter) ← 마지막 원소 반환 후에는 StopIteration을 반환함
30
>>> next(l_iter)
...
StopIteration:
```

대화창 실습 : `__next__()` 메소드를 사용한 반복자 객체의 요소 추출

```
>>> lst = [10, 20, 30]
>>> l_iter = iter(lst) # 리스트 형 객체를 반복자로 만들
>>> l_iter.__next__() ← __next__() 특수 메소드를 호출함
10
>>> l_iter.__next__()
20
>>> l_iter.__next__()
30
>>> l_iter.__next__() ← 마지막 원소 반환 후에는 StopIteration을 반환함
...
StopIteration
```

## 반복자 iterator

[출처] 유틸 파이썬, “10장 lambda 함수와 list comprehension (리스트 축약)”

- 코드 : 리스트와 튜플, 정수 값을 반복자 객체로 변환

```
# 리스트가 반복 가능 객체인가 검사
try:
    l = [10, 20, 30]
    iterator = iter(l)
except TypeError:
    print('list는 iterable 객체가 아닙니다.')
else:
    print('list는 iterable 객체입니다.')

# 튜플이 반복 가능 객체인가 검사
try:
    t = ('홍길동', 22, 79.7)
    iterator = iter(t)
except TypeError:
    print('tuple은 iterable 객체가 아닙니다.')
else:
    print('tuple은 iterable 객체입니다.')

# 정수형이 반복 가능 객체인가 검사
try:
    n = 100
    iterator = iter(n)
except TypeError:
    print('n은 iterable 객체가 아닙니다.')      list는 iterable 객체입니다.
else:                                         tuple은 iterable 객체입니다.
    print('n은 iterable 객체입니다.')          n은 iterable 객체가 아닙니다.
```

# 반복자(iterator) 클래스

[참고] <https://engineer-mole.tistory.com/64>, <https://wikidocs.net/74399>

## ➤ Iterable 객체와 Iterator 객체

- 파이썬의 어떤 객체가 `_iter_` 메소드를 포함하고 있다면 우리는 해당 객체를 **Iterable 객체**라고 부른다.
- 앞서 `iter`라는 내장 함수를 호출하면 내부적으로 해당 객체의 `_iter_` 메소드를 호출하게 됩니다. `_iter_` 메소드는 Iterator 객체를 반환해주는데 여기서 Iterator 객체는 `_next_` 메소드를 반드시 구현하고 있어야 한다.

- 다음 코드를 살펴봅시다.
- `MyIterable` 클래스는 `_iter_` 메소드를 포함하고 있으며 **내부적으로 iterator** 객체를 리턴한다. 따라서 iterable한 객체입니다.
- `MyIterator` 클래스는 `_next_` 메소드를 포함하고 있으므로 `next` 내장 함수에 대해서 동작한다. 따라서 iterator 객체이다. Iterable 객체로부터 Iterator 객체를 얻었으면 우리는 `next` 내장 함수를 호출하여 계속해서 값을 얻어올 수 있다.

```
class MyIterator:  
    def __next__(self):  
        return 1  
  
class MyIterable:  
    def __iter__(self):  
        obj = MyIterator()  
        return obj  
  
m = MyIterable()  
r = iter(m)  
print(next(r))  
print(next(r))  
print(next(r))  
1  
1  
1
```

- `Iterable`을 상속받는 클래스는, 즉 클래스가 `Iterable`이기 위해서는:
  - `_iter_` 추상메소드를 실제로 구현해야 하며 이 메소드는 **호출될 때마다** 새로운 `Iterator`를 반환해야 한다.
- 자료구조나 클래스가 `Iterable`이기 위해서는 이 조건만 만족하면 된다. 즉, 클래스에 `_iter_` 메소드가 구현되었으면 해당 객체에 `iter`라는 내장 함수를 적용해 `Iterator`를 생성할 수 있다.

- 클래스가 `Iterator`이기 위한 필수적인 메소드는 `_next_`인 것 같다.
- 어떤 클래스가 `Iterator`이기 위해서는 다음과 같은 조건을 만족해야 한다:
  - 클래스는 `_iter_`를 구현하되 자기 자신(`self`)을 반환해야 한다.
  - 클래스는 `_next_` 메소드를 구현해서 `Iterator`를 `next` 내장 함수의 인자로 줬을 때 다음에 반환할 값을 정의해야 한다.
  - `Iterator`가 더 이상 반환할 값이 없는 경우는 `_next_` 메소드에서 `StopIteration` 예외를 일으키도록 한다.
- `next` 내장 함수는 인자가 되는 `Iterator`의 다음 인자를 반환하고 위치를 다음으로 옮기는 **기능**을 한다.

# 반복자(iterator) 클래스

[참고] <https://engineer-mole.tistory.com/64>, <https://wikidocs.net/74399>

## ➤ Iterator 객체 만들기

- 우리가 어떤 타입을 만드려고 하는데 이 타입이 반복 가능한 성질을 갖고 있도록 설계하고자 할 때 바로 Iterator를 사용한다. 클래스 내에서 `_next_` 와 `_iter_` 메소드를 적절히 구현만 해주면 된다.
- Iterator 클래스 통한 객체 만들기

```
class Season: 출력값
    # 구현 생략

s = Season()
for i in s:
    print(i)
```

봄  
여름  
가을  
겨울

- Season 클래스 : 봄, 여름, 가을, 겨울 출력
- Iterator 객체를 만들 때 `_iter_` 와 `_next_` 메소드를 구현해야 함

- 1) `_iter_` 메소드 구현 : `_iter_` 메소드는 Iterator 객체를 리턴해야 하는데 Season 클래스 자기 자신이 Iterator 객체 (`_next_`를 포함함)이므로 자기 자신(self)를 리턴하도록 구현한다.

```
# Season 클래스
class Season:
    def __init__(self):
        pass

    def __iter__(self):
        return self      # 자기 자신이 iterator로 self 리턴
```

- 2) `_next_` 메소드 구현 : for 문에서 각 반복마다 계절의 이름이 순서대로 출력되는 기능을 `_next_` 메소드에 구현한다.
- 먼저 생성자에서 계절 이름을 리스트로 저장하고 출력할 계절 이름의 위치를 기억하기 위한 변수를 하나 만든다.
  - 내부적으로 유지되는 `self.index` 값이 4를 넘지 않으면 현재의 계절의 이름을 리턴해주고 `self.index` 값을 1 증가시킨다. 만약 `self.index` 값이 4보다 크거나 같다면 `StopIteration` 예외를 발생시킨다.

```
class Season:
    def __init__(self):
        self.data = ["봄", "여름", "가을", "겨울"]
        self.index = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.index < len(self.data):
            cur_season = self.data[self.index]
            self.index += 1
            return cur_season
        else:
            raise StopIteration
```

# 반복자(iterator) 클래스

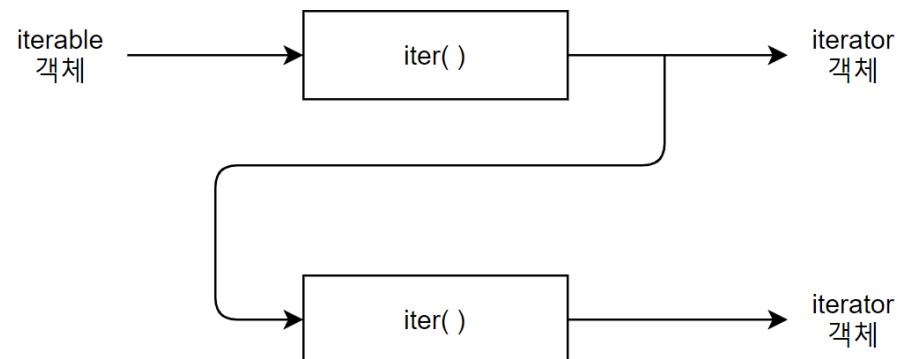
[참고] <https://engineer-mole.tistory.com/64>, <https://wikidocs.net/74399>

## ➤ Iterator 객체 만들기

- 3) Season 클래스의 객체를 생성한 후 **iter 내장함수**를 호출하여 Iterator 객체를 생성한다. 그 다음 Iterator 객체에 대해서 다시 **next 내장 함수**를 호출하여 반복적으로 값을 얻는다.

```
s = Season()      또는      s = Season()      # 클래스 객체 생성
ir = iter(s)          for i in s:
print(next(ir))                  print(i)
print(next(ir))
print(next(ir))
print(next(ir))      for 문을 사용하여 반복할 때마다 내부적으로 next(ir) 호출
```

- 보통 iterator 객체를 얻기 위해 iter 내장 함수에 iterable 객체를 입력한다. 그런데 iter 내장함수의 리턴 값인 iterator 객체를 다시 iter에 넣어도 전달된 iterator 객체를 그대로 리턴해준다.



## 반복자(iterator) 클래스

[출처] 유틸 파일썬, “10장 lambda 함수와 list comprehension (리스트 축약)”

### OddCounter 클래스의 정의와 객체 생성

```
class OddCounter:  
    """ 1부터 증가하는 허수를 반환하는 클래스 """  
    def __init__(self, n = 1): # 초기화 메소드 n을 1로 둔다  
        self.n = n  
  
    def __iter__(self): # 반복자는 __iter__() 함수를 가져야 함  
        return self  
  
    def __next__(self): # 반복자는 __next__() 함수를 가져야 함  
        t = self.n # self.n을 임시 변수 t에 저장해 두고  
        self.n += 2 # self.n을 2증가시킨다  
        return t  
  
my_counter = OddCounter()  
print(next(my_counter))  
print(my_counter.__next__())  
print(my_counter.__next__())  
print(my_counter.__next__())
```

1  
3  
5  
7

### OddCounter 클래스의 정의와 20보다 작은 허수를 출력

```
class OddCounter:  
    # 1부터 증가하는 허수를 반환하는 클래스  
    def __init__(self, n = 1): # 초기화 메소드, n를 1로 둔다  
        self.n = n  
  
    def __iter__(self): # 반복자는 __iter__() 함수를 가져야 함  
        return self  
  
    def __next__(self): # 반복자는 __next__() 함수를 가져야 함  
        t = self.n # self.n을 임시 변수 t에 지정해 두고  
        self.n += 2 # self.n을 2씩 증가  
        return t  
  
my_counter = OddCounter()  
for x in my_counter:  
    if x > 20:  
        break  
    print(x, end = ' ')
```

1 3 5 7 9 11 13 15 17 19  
Process finished with exit code 0

## 반복자(iterator) 클래스

[출처] 유틸 파이썬, “10장 lambda 함수와 list comprehension (리스트 축약)”

OddCounter 클래스와 StopIteration 예외 생성 가능

```
class OddCounter:  
    def __init__(self, n = 1):  
        self.n = n  
  
    def __iter__(self):  
        return self  
  
    def __next__(self):  
        if self.n < 20:      # 반복자가 수행되는 조건  
            t = self.n  
            self.n += 2  
            return t  
  
        raise StopIteration # 조건을 만족하지 않으면 StopIteration을 raise함  
  
my_counter = OddCounter()  
for x in my_counter:  
    print(x, end = ' ')  
  
1 3 5 7 9 11 13 15 17 19  
Process finished with exit code 0
```

## 반복가능 객체를 위한 내장함수

[출처] 유틸 파이썬, “10장 lambda 함수와 list comprehension (리스트 축약)”

- 파이썬의 반복가능 iterable 객체는 다양한 내장함수들을 적용할 수 있다.
- 앞서 살펴봤던 min()이나 max()와 같은 함수는 반복가능 객체를 인자로 받아서 최솟값과 최댓값을 반환하는데 이들 외에도 all(), any(), ascii(), bool(), filter(), iter()와 같은 고급 내장함수도 제공되고 있다
- all() 함수는 반복 가능한 항목들이 모두 참일 때 참을 반환한다.

```
11 = [1,2,3,4]    # 모든 요소가 0이 아님  
12 = [0,2,4,8]    # 한 요소만 0임  
13 = [0,0,0,0]    # 모든 요소가 0임
```

```
all(11) # 모든 요소가 true(0이 아님 값)일 때만 True를 반환함
```

True

```
print(all(12), all(13))
```

False False

- any()는 임의의 반복 가능한 항목들 중에서 참이 하나라도 있을 경우 참을 반환

```
print(any(11)) # 항목들 중 하나라도 true(0이 아님 값)일 경우 True를 반환함  
print(any(12))  
print(any(13)) # 항목들이 모두 false(0의 값)이므로 False를 반환함
```

True  
True  
False

- bool()은 값(리스트)을 부울값으로 변환한다. 즉 리스트의 항목 유무를 True와 False로 알려준다

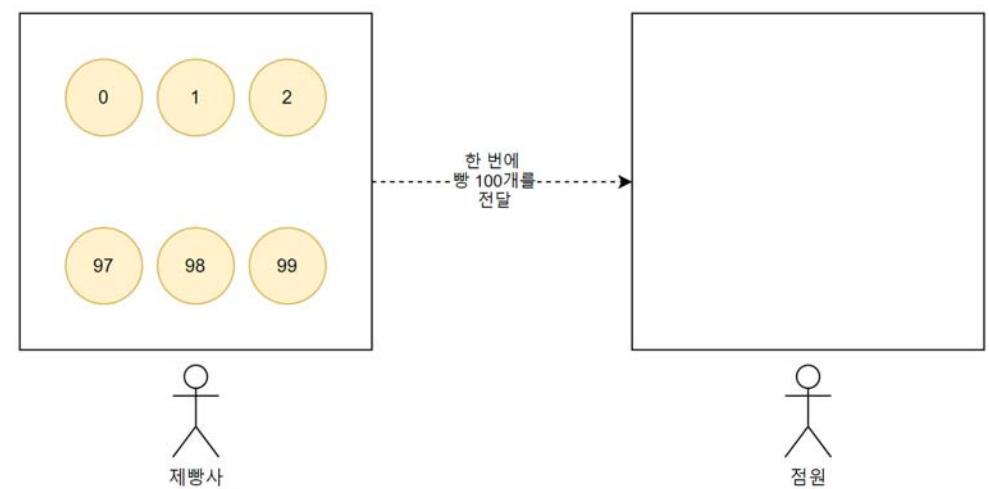
```
print(bool(11)) # 리스트에 원소가 있으면 True를 반환함  
print(bool(12))  
print(bool(13))  
14 = []  
print(bool(14)) # 리스트에 원소가 없으면 False를 반환함
```

True  
True  
True  
False

## ➤ Generator

- generator 함수 : iterator를 생성해주는 함수, 함수 안에 return 키워드 대신 yield 키워드를 사용함
- yield 문으로 값을 반환한 후 계속 진행
  - ✓ return 키워드는 결과값을 돌려주고 함수도 끝나버리지만 yield 키워드는 결과값을 돌려주지만 실행 상태를 그대로 저장해 둔 채로 잠시 멈춘다. 따라서 나중에 다시 이전에 멈춘 상태에서부터 이어서 다음 코드를 실행할 수 있게 된다.
- generator 특징
  - ✓ iterable한 순서가 지정됨(모든 generator는 iterator)
  - ✓ 느슨하게 평가된다.(순서의 다음 값은 필요에 따라 계산됨)
  - ✓ 함수의 내부 로컬 변수를 통해 내부상태가 유지된다.
  - ✓ 무한한 순서가 있는 객체를 모델링할 수 있다. (명확한 끝이 없는 데이터 스트림) → 모든 값을 반환하는 대신 호출할 때마다 값을 리턴하므로, 아주 작은 메모리로 대용량의 반복 가능한 구조를 순차적으로 순회함

- 잘못된 일처리 방식 : 빵이 100개 될 때까지 순서대로 하나씩 빵을 구은 후 100개가 준비되면 이를 점원에게 한 번에 전달
- ✓ 파이썬의 함수는 호출하면 호출이 끝날 때까지 호출부가 대기하고 있다가 함수가 리턴한 값을 받아서 사용하게 됨.



- 빵을 낱개로 포장하는 거라면 제빵사가 빵을 한 번에 100개를 전달하는 것이 아니라 2개나 5개처럼 빵이 일부라도 준비가 될 때 전달해주는 것.
- ✓ 파이썬 함수도 그때 그때 준비된 값을 리턴하고 다시 돌아와서 함수 내 코드를 실행하고 다시 준비되면 리턴해야 함

## Generator와 yield 문

[출처] 유틸 파이썬, “10장 lambda 함수와 list comprehension (리스트 축약)”

- 파이썬은 반복자 말고도 generator라는 객체를 제공하는데 이 객체는 모든 값을 메모리에 올려두고 이용하는 것이 아니라 필요할 때마다 생성해서 반환하는 일을 한다
- 메모리를 효율적으로 활용할 수 있다는 장점이 있다.

```
my_generator = (x for x in range(1,4))
for n in my_generator:
    print(n)

type(my_generator)
```

```
1
2
3

generator
```

- tuple comprehension이 아니고, generator 객체 생성

```
my_generator = (x for x in range(1,4))
print(type(my_generator))

my_tuple = tuple(x for x in range(1,4))
print(type(my_tuple))
```

```
<class 'generator'>
<class 'tuple'>
```

- 반복자와 다른 점은 여기에서 생성된 1, 2, 3을 미리 메모리에 만들어 두는 것이 아니라 for 문에서 필요로 할 때마다 반환해 주고 메모리에서 보관하지 않는다는 점이다.

```
def create_gen():
    alist = range(1, 4)
    for x in alist:
        yield x

my_generator = create_gen()
print(my_generator)
for n in my_generator:
    print(n)
```

```
<generator object create_gen at 0x000001A3853B7F90>
1
2
3
```

# Generator

[참고] <https://engineer-mole.tistory.com/64>, <https://wikidocs.net/74399>

## ■ generator와 yield

- yield 문 : 함수를 종결하지 않으면서 값을 계속 반환

```
def genFunc():
    yield 1
    yield 2
    yield 3

print(list(genFunc()))
gen = genFunc()
type(gen)

[1, 2, 3]
generator

next(gen)
1

next(gen)
2

next(gen)
3

next(gen)
0 1 2

StopIteration
~\AppData\Local\Te
-----> 1 next(gen)

StopIteration:
```

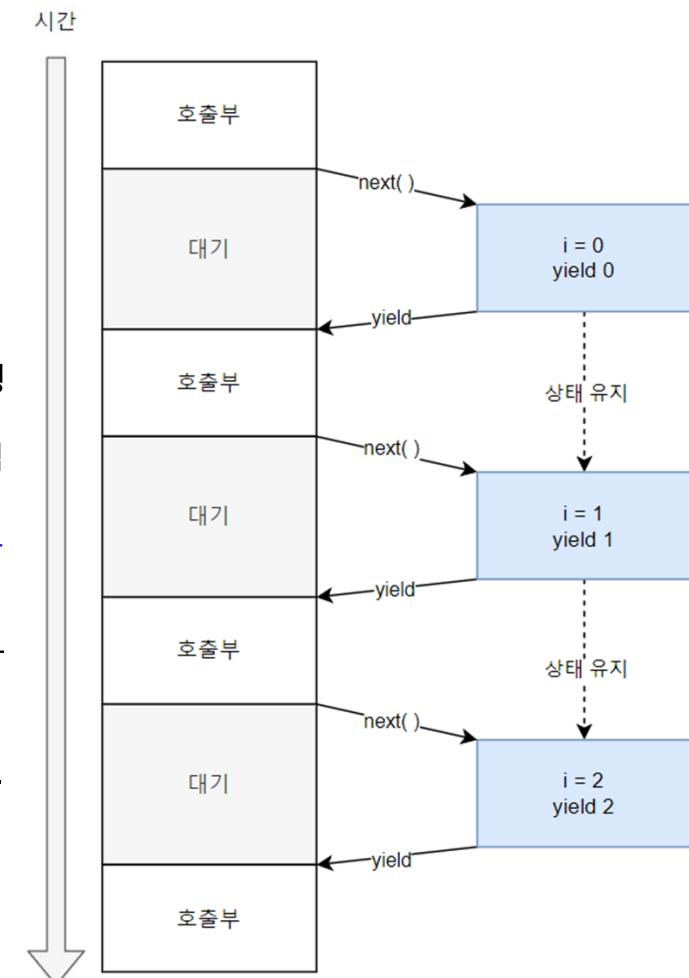
```
def genFunc():
    for i in range(3):
        yield i

gen = genFunc()

num1 = next(gen)
num2 = next(gen)
num3 = next(gen)

print(num1, num2, num3)
```

- yield가 사용된 함수는 바로 실행되지 않고, 대신 generator라는 객체가 생성되고 코드가 실행될 준비를 한다.
- generator 객체를 생성했다면 next() 함수 호출을 통해 generator 객체 내의 코드를 실행할 수 있는데 이때 yield 구문을 만나면 파이썬 함수의 return처럼 yield 키워드에 있는 값(여기서는 i)을 호출부로 리턴하고 실행의 흐름도 호출부로 이동하게 된다. 다만 파이썬 함수와 달리 현재 상태를 그대로 유지할 수 있다. 코드가 중지된 시점의 상태를 유지하고 있기 때문에 해당 상태로부터 다시 코드를 이어서 실행할 수 있다.



- yield 가 호출되면 암시적으로 return이 호출되며, 한번 더 실행되면 실행되었던 'yield' 다음 코드가 실행된다.

# Generator

[참고] <https://engineer-mole.tistory.com/64>, <https://wikidocs.net/74399>

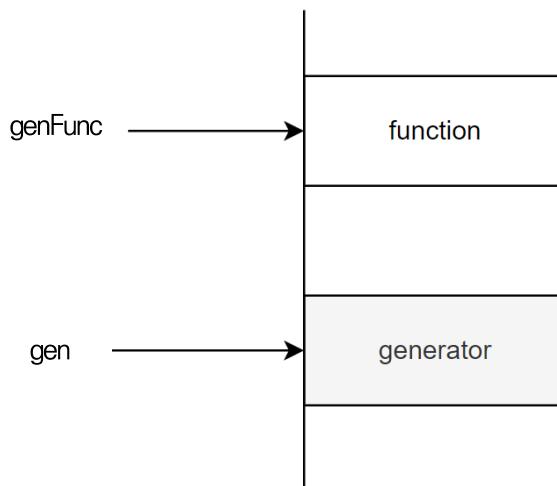
- generator 함수를 호출하면, 다음과 같이 generator 클래스의 객체가 생성됨

```
def genFunc():
    for i in range(3):
        yield i

gen = genFunc()
print(gen, type(gen))

<generator object genFunc at 0x0000020686098040> <class 'generator'>
```

- gen 변수가 바인딩하는 generator 객체는 next(g)를 호출할 때 0, 1, 2라는 값을 순서대로 돌려준다. 이때 한 번더 next(g)를 호출하면 더이상 돌려줄 값이 없기 때문에 StopIteration이라는 에러가 발생한다.



- next(g)를 사용하지 않고 generator로부터 값을 가져오는 쉬운 방법
- for 문을 사용하면 반복할 때마다 내부적으로 next(g)를 호출하는데 이는 StopIteration이 발생할 때 까지 계속된다. 즉, 다음과 같이 generator 객체로부터 순차적으로 값을 가져오게 됩니다.

- 이렇게 생성한 generator는 iterable한 객체가 되며 for문에서 사용할 수 있다.

```
def genFunc(num):
    for i in range(0, num):
        yield i

for data in genFunc(5):
    print(data)

0
1
2
3
4
```

```
import collections

def getFunc(num):
    for i in range(0, num):
        yield i

gen = getFunc(5)
print(isinstance(gen, collections.abc.Iterable))

for data in getFunc(5):
    print(data, end=' ')
```

```
True
0 1 2 3 4
```

```
def genFunc():
    for i in range(3):
        yield i

gen = genFunc()
for i in gen:
    print(i)

0
1
2
```



## Generator

[참고] <https://engineer-mole.tistory.com/64>, <https://wikidocs.net/74399>

- generator를 동시에 두개 생성하면, 서로가 다른 객체이며, 각기 따로 동작한다.

```
x = genFunc(5)
y = genFunc(5)

print(x == y)
print(x is y)

False
False

for _ in range(0, 5):
    print(next(x), next(y))

0 0
1 1
2 2
3 3
4 4
```

- 그 때 그 때 생성되는 무한한 순서있는 객체를 모델링할 수 있다.
- 이렇게 무한하게 숫자를 리턴할 수 있는 이유는 generator를 실행할 때마다 느슨하게 평가되며 내부의 변수가 유지되고 있기 때문이다.

```
def infinite_generator():
    count = 0
    while True:
        count += 1
        yield count

gen = infinite_generator()
for i in range(50):
    print(next(gen), end=' ')
```

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23  
24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43  
44 45 46 47 48 49 50

- List Set, Dictionary의 표현식의 내부도 사실 generator이다.

```
type(x*x for x in [2, 4, 6])
```

generator

## ➤ yield from

- yield문은 한번씩 값을 바깥으로 전달했습니다. 여러번 바깥으로 전달할려면 for문을 아래와 같이 사용해야 한다.

```
def three_generator():
    a = [1, 2, 3]
    for i in a:
        yield i

gen = three_generator()
list(gen)
```

[1, 2, 3]

- for문 대신에 iterable한 객체를 yield할 때는 **yield from iterable**로 값을 전달할 수 있다

```
def three_generator():
    a = [1, 2, 3]
    yield from a

gen = three_generator()
list(gen)
```

[1, 2, 3]

# iterator과 generator 정리

[참고] <https://engineer-mole.tistory.com/64>, <https://wikidocs.net/74399>

- iterator : 요소가 복수인 컨테이너(리스트, 튜플, 셋, 사전, 문자열)에서 각 요소를 하나씩 꺼내 어떤 처리를 수행할 수 있도록 하는 간편한 방법을 제공하는 객체
- generator : iterator의 한 종류로, 하나의 요소를 꺼내려고 할 때마다 요소 generator를 수행하는 타입으로, Python에서는 yield문을 통해 구현하는 것이 보통이다.

- python의 내장 컬렉션 (list, tuple, set, dic 등)의 어떤 것인든 iterable을 상속받고, 내장 컬렉션을 사용한 반복문 처리로 미리 컬렉션 값을 입력해 둬야 할 필요가 있으므로 아래와 같은 경우는 iterator 또는 generator를 직접 구현하고 싶다고 생각하는 경우가 있을 것이다.

- 무한히 반복하는 iterator
- 모든 요소를 미리 계산하는 것이 혹은 가져 오는 것이 계산 비용, 처리 시간, 메모리 사용량 등의 측면에서 힘든 경우

## ❖ (cf) 각 자료구조의 iterable, Generator, Iterator 상속 여부

- list는 Iterable를 상속 받나요? True
- list는 Iterator를 상속 받나요? False
- list는 Generator를 상속 받나요? False
- tuple는 Iterable를 상속 받나요? True
- tuple는 Iterator를 상속 받나요? False
- tuple는 Generator를 상속 받나요? False
- set는 Iterable를 상속 받나요? True
- set는 Iterator를 상속 받나요? False
- set는 Generator를 상속 받나요? False
- dict는 Iterable를 상속 받나요? True
- dict는 Iterator를 상속 받나요? False
- dict는 Generator를 상속 받나요? False

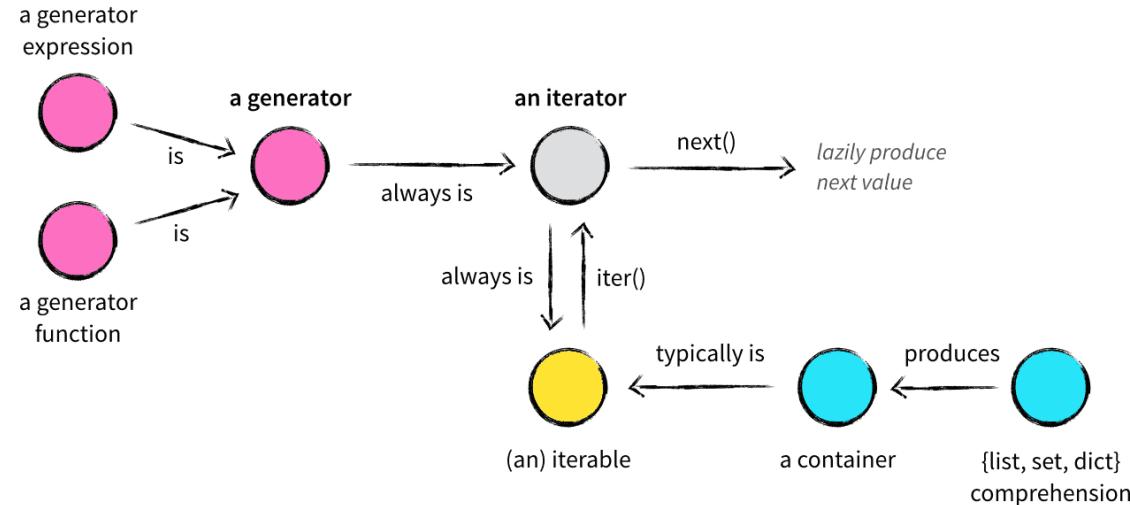


그림. iterator와 generator의 관계도

## 참고자료

# if \_\_name\_\_ == '\_\_main\_\_'

[참고] <https://dojang.io/mod/page/view.php?id=2448>, <https://wikidocs.net/29>

if \_\_name\_\_ == "\_\_main\_\_"은 인터프리터에서 직접 실행했을 경우에만 if문 내의 코드를 실행하는 명령

\_\_name\_\_ 변수란?

- 파이썬의 \_\_name\_\_ 변수는 interpreter가 실행 전에 만들어 둔 글로벌 변수
- 만약 C:\doit>python mod1.py처럼 직접 mod1.py 파일을 실행할 경우 mod1.py의 \_\_name\_\_ 변수에는 \_\_main\_\_ 값이 저장된다.
- 하지만 파이썬 셸이나 다른 파이썬 모듈에서 mod1을 import 할 경우에는 mod1.py의 \_\_name\_\_ 변수에는 mod1.py의 모듈 이름 값 mod1이 저장된다.

```
# mod1.py
def add(a, b):
    return a+b

def sub(a, b):
    return a-b

if __name__ == "__main__":
    print(add(1, 4))
    print(sub(4, 2))
```

- if \_\_name\_\_ == "\_\_main\_\_"을 사용하면 C:\doit>python mod1.py처럼 직접 이 파일을 실행했을 때는 \_\_name\_\_ == "\_\_main\_\_"이 참이 되어 if문 다음 문장이 수행된다.
- 반대로 대화형 인터프리터나 다른 파일에서 이 모듈을 불러서 사용할 때는 \_\_name\_\_ == "\_\_main\_\_"이 거짓이 되어 if문 다음 문장이 수행되지 않는다.

예제) excuteThisModule.py

```
# excuteThisModule.py
def func():
    print("function working")

if __name__ == "__main__":
    print("직접 실행")
    print(__name__)
else:
    print("임포트되어 사용됨")
    print(__name__)
```

1) 인터프리터에서 직접 실행할 때 : python3 executeThisModule.py

- \_\_name\_\_ 변수에 "\_\_main\_\_"이 담겨서 print

실행결과

직접 실행  
\_\_main\_\_.

2) 다른 모듈에 임포트해서 실행 :

import executeThisModule.py  
executeThisModule.func()

실행결과

임포트되어 사용됨  
executeThisModule

- \_\_name\_\_ 변수에 "executeThisModule"이 담겨서 print

# if \_\_name\_\_ == '\_\_main\_\_'

[참고] <https://dojang.io/mod/page/view.php?id=2448>, <https://wikidocs.net/29>

hello.py

```
hello.py  X  hello_main.py  A.py  B.py  Module1.py
CookPython(2019.10.15) > Module > hello.py
1 # __name__ 변수 알아보기 : https://dojang.io/mod/page/view.php?id=2448
2 # hello.py
3
4 print('hello 모듈 시작')
5 print('hello.py __name__: ', __name__)    # __name__ 변수 출력
6 print('hello 모듈 끝')
7
```

ecture/CookPython(2019.10.15)/Module/hello.py"
hello 모듈 시작
hello.py \_\_name\_\_: main\_
hello 모듈 끝

hello\_main.py

```
hello.py  X  hello_main.py  A.py  B.py  Module1.py
CookPython(2019.10.15) > Module > hello_main.py > ...
1 # __name__ 변수 알아보기 : https://dojang.io/mod/page/view.php?id=2448
2 # hello_main.py
3
4 import hello    # hello 모듈을 가져옴
5
6 print('hello_main.py __name__: ', __name__)    # __name__ 변수 출력
```

ecture/CookPython(2019.10.15)/Module/hello\_main.py"
hello 모듈 시작
hello.py \_\_name\_\_: hello
hello 모듈 끝
hello\_main.py \_\_name\_\_: main\_

파이썬에서 import로 모듈을 가져오면 해당 스크립트 파일이 한 번 실행된다.

hello 모듈을 가져오면 hello.py 안의 코드가 실행된다.

hello.py의 \_\_name\_\_ 변수에는 'hello'가 들어가고, hello\_main.py의 \_\_name\_\_ 변수에는 '\_\_main\_\_'이 들어간다.

# 언더스코어('\_')

[출처] <https://mingrammer.com/underscore-in-python/>, <https://ebbnflow.tistory.com/255>

- 파이썬에서 언더스코어(\_)는 다양한 의미를 가짐

## Case1. 언더스코어가 독립적으로 사용

- 인터프리터에서 마지막 실행 결과 값을 가지는 변수로 '\_' 사용

```
1+3
```

```
4
```

```
x = _  
print(x)
```

```
4
```

```
y = _ + 10  
print(y)
```

```
14
```

```
# Unpacking / 특정값 무시  
x, _, y = (1, 2, 3) # x = 1, y = 3  
  
# 여러 개의 값 무시  
x, *_ , y = (1, 2, 3, 4, 5) # x = 1, y = 5  
  
# 인덱스 무시  
for _ in range(10):  
    do_something()  
  
# 특정 위치의 값 무시  
for _, val in list_of_tuple:  
    do_something()
```

## Case2. 변수로 사용될 경우

- 변수 값을 굳이 사용할 필요가 없을 때 사용
- 변수를 문자로 할당해주어도 되지만 굳이 사용하지 않을 경우에 '\_'로 치환함. 변수명이 낭비될 필요가 없기 때문임.

```
for _ in range(5): # 단순히 5번 반복할 목적으로  
    print('hello world')  
  
for _ in range(3): # _를 i로 바꿔서 보면 우리에게 익숙한 표현식일 것이다.  
    print(_)  
  
def values():  
    return (1,2,3,4) # 튜플을 반환  
  
a, b, _, _ = values() # 반환된 튜플 값중 2개만 필요할 경우  
print(a, b)  
  
hello world  
hello world  
hello world  
hello world  
hello world  
0  
1  
2  
1 2
```

## 언더스코어('\_')

### Case3. \_\_이름\_\_ (더블 언더스코어)

#### double\_leading\_and\_trailing\_underscores

- 내장된 특수한 함수와 변수를 나타낸다.

```
class vector:  
    def __init__(self, x,y,z):  
        self.x = x  
        self.y = y  
        self.z = z  
  
    def __add__(self, other):  
        x_ = self.x + other.x  
        y_ = self.y + other.y  
        z_ = self.z + other.z  
  
        return vector(x_,y_,z_)  
  
    def show(self):  
        print(f"x:{self.x}, y:{self.y}, z:{self.z}")  
  
v1 = vector(1,2,3)  
v2 = vector(4,5,6)  
v3 = v1 + v2  
v3.show()
```

- \_\_init\_\_은 클래스의 생성자 함수이며, \_\_add\_\_ 함수는 연산자 오버로드용으로 사용

- 스페셜 변수나 메서드(매직 메서드라고도 부른다.)에 사용되는 컨벤션이며, \_\_init\_\_, \_\_len\_\_과 같은 메서드들이 있다.
- 이런 형태의 메서드들은 어떤 특정한 문법적 기능을 제공하거나 특정한 일을 수행한다. 가령, \_\_file\_\_은 현재 파일의 위치를 나타내는 스페셜 변수이며, \_\_eq\_\_은 a == b라는 식이 수행될 때 실행되는 스페셜 메서드이다. 물론 사용자가 직접 만들 수도 있지만 그런 경우는 정말 거의 없으며, 일부 스페셜 메서드의 경우 직접 수정하거나 하는 일은 빈번히 있을 수 있다. \_\_init\_\_의 경우 클래스의 인스턴스가 생성될 때 처음으로 실행되는 메서드인데 인스턴스의 초기화 작업을 이 메서드의 내용으로 작성할 수 있다.

```
class A:  
    def __init__(self, a): # 스페셜 메서드 __init__에서 초기화 작업을 한다.  
        self.a = a  
  
    def __custom__(self): # 커스텀 스페셜 메서드. 이런건 거의 쓸 일이 없다.  
        pass
```

x:5, y:7, z:9

## 언더스코어('\_')

### Case4. 네이밍

- 가장 많이 사용하는 경우로
- 변수 이름 뒤에 \_가 붙는 경우, 함수 이름 왼쪽 \_를 붙이는 경우, 변수 왼쪽에 \_\_를 붙이는 경우, 그리고 *mangling*할 때 사용한다.

#### 1) 변수명\_(single\_trailing\_underscore)

- 예약어를 변수명으로 사용할 수 없을 때 사용한다. 즉, 파이썬 키워드와의 충돌을 피하기 위해 사용하는 컨벤션 (*PEP8 - 파이썬 코드 스타일 가이드*)으로 그 리 많이 사용하지는 않을 것이다.

```
class Order:  
    def __init__(self):  
        self.coffee = 'Americano'  
        self.price = 3000  
  
def printClassName(class_):  
    print(class_)  
  
order = Order()  
printClassName(order.__class__)  
  
Tkinter.Toplevel(master, class_='ClassName') # class_와의 충돌을 피함  
list_ = List.objects.get(1) # list와의 충돌을 피함
```

## 언더스코어('\_')

### Case4. 네이밍

#### 2) \_함수 (\_single\_leading\_underscore)

- 주로 한 모듈 내부에서만 사용하는 **private 클래스/함수/변수/메서드를 선언할 때 사용하는 컨벤션이다.**
- 이 컨벤션으로 선언하게 되면 from module import \*시 \_로 시작하는 것들은 모두 import에서 무시된다. 그러나, 파이썬은 진정한 의미의 private 을 지원하고 있지는 않기 때문에 private을 완전히 강제할 수는 없다. 즉, 위와 같은 임포트문에서는 무시되지만 직접 가져다 쓰거나 호출을 할 경우엔 사용이 가능하다. 그래서 “**weak internal use indicator**”라고 부르기도 한다.

Module1.py

```
def publicFunc():
    print('this is public function')

def __privateFunc():
    print('this is private function')
```

A.py

```
from Module1 import *

publicFunc()
__privateFunc()  this is public function
Traceback (most recent call last):
  File "C:/sky/동영상강의/2021학년도2학기/파이썬/Pyt
      _privateFunc()
NameError: name '_privateFunc' is not defined
```

```
import Module1
Module1.publicFunc()
Module1.__privateFunc()
```

```
this is public function
this is private function
```

- 모듈명을 가져와서 호출할 수 있으므로, 이를 약한 private라고 함.

- 함수 이름 왼쪽에 \_를 붙이면 다른 파일을 import할 때 \_를 붙인 함수는 가져오지 않는다
- 보안성이 약한 private 의미로 사용하는 경우이다. 파이썬은 진정한 private를 지원하지 않기 때문에 import문에서는 누락이 가능하지만 직접 가져다 쓰거나 호출을 할 때는 사용이 가능하다.

# 언더스코어('\_')

## Case4. 네이밍

### 2) \_함수 (\_single\_leading\_underscore)

```
_internal_name = 'one_module' # private 변수  
_internal_version = '1.0' # private 변수  
  
class _Base: # private 클래스  
    _hidden_factor = 2 # private 변수  
  
    def __init__(self, price):  
        self._price = price  
  
    def _double_price(self): # private 메서드  
        return self._price * self._hidden_factor  
  
    def get_double_price(self):  
        return self._double_price()
```

- 외부에서 Module1.py의 \_price 변수에 접근하려면 클래스명\_변수명 형식으로 변경해서 사용할 수 있다. 이러한 작업을 ' 맹글링(Mangling)'이라고 한다.

### 3) 변수 왼쪽에 \_\_(더블스코어)를 붙인 경우 (\_double\_leading\_underscores) → Mangling

- 언더스코어를 두개(\_\_)붙인 변수는 선언된 클래스 안에서만 해당 이름으로 사용 가능하다. 다음 예제에서는 Module1.py 안에서만 \_\_price 변수가 \_\_price이름으로 사용이 가능하고, 외부 모듈인 A.py 안에서는 \_\_price이름으로 사용이 불가능하다.

#### Module1.py

```
class Order:  
    def __init__(self):  
        self.coffee = 'Americano'  
        self.__price = 3000  
  
    def printInfo(self):  
        print(f"coffee : {self.coffee}")  
        print(f"price : {self.__price}")
```

#### A.py

```
import Module1  
  
order1 = Module1.Order()  
print(order1.coffee)  
print(order1.__price)
```

Traceback (most recent call last):

```
File "C:/sky/동영상강의/2021학년도2학기/파이썬/Python for Beg."  
    print(order1.__price)  
AttributeError: 'Order' object has no attribute '__price'
```

#### Module1

```
import Module1  
  
order1 = Module1.Order()  
  
print(order1.__Order__price)  
order1.printInfo()
```

```
Americano  
3000  
coffee : Americano  
price : 3000  
this is public function  
this is private function
```

# 언더스코어('\_')

## Case4. 네이밍

### 3) 변수 왼쪽에 \_(더블스코어)를 붙인 경우 (\_double\_leading\_underscores) → Mangling

- 이는 컨벤션이라기보단 하나의 문법적인 요소이다.
- 더블 언더스코어는 클래스 속성명을 mangling하여 클래스간 속성명의 충돌을 방지하기 위한 용도로 사용된다. (mangling이란, 컴파일러나 인터프리터가 변수/함수명을 그대로 사용하지 않고 일정한 규칙에 의해 변형시키는 것을 말한다.)
- 파이썬의 mangling 규칙은 더블 언더스코어로 지정된 속성명 앞에 \_ClassName을 결합하는 방식이다. 즉, ClassName이라는 클래스에서 \_method라는 메서드를 선언했다면 이는 ClassName\_method로 mangling 된다.
- 더블 언더스코어로 지정된 속성명은 위와 같이 맹글링이 되기 때문에 일반적인 속성 접근인 ClassName.\_method으로 접근이 불가능하다.
- 간혹, 이러한 특징으로 더블 언더스코어를 사용해 진짜 private처럼 보이게 하는 경우가 있는데 이는 private을 위한 것이 아니며 private으로의 사용도 권장되지 않는다. 이에 대한 자세한 내용은 Python Naming을 참고하면 좋을 것 같다.

```
class A:  
    def __single_method(self):  
        pass  
  
    def __double_method(self): # Mangling을 위한 메서드  
        pass  
  
class B(A):  
    def __double_method(self): # Mangling을 위한 메서드  
        pass  
  
print(dir(A())) # ['__A__double_method', ..., '__single_method']  
print(dir(B())) # ['__A__double_method', '__B__double_method', ..., '__single_method']  
  
# 서로 같은 이름의 메서드를 가지지만 오버라이드가 되지 않는다.  
['__A__double_method', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__',  
['__A__double_method', '__B__double_method', '__class__', '__delattr__', '__dict__', '__']
```

## 더블 언더스코어('\_\_') : Magic Method

[참고] <https://corikachu.github.io/articles/python/python-magic-method>

# Python imports, `__init__.py` and pythonpath

[참고] <https://towardsdatascience.com/understanding-python-imports-init-py-and-pythonpath-once-and-for-all-4c5249ab6355>

PYTHON BASICS

## Understanding Python imports, `__init__.py` and pythonpath — once and for all

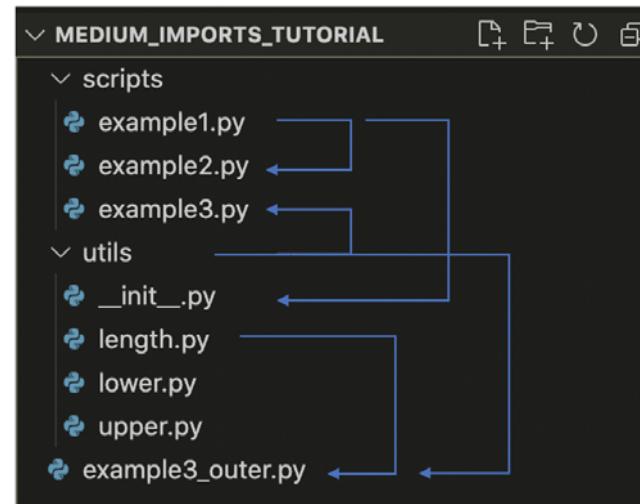
Learn how to import packages and modules (and the difference between the two)

 Dr. Varshita Sher 6 days ago · 12 min read \*



By the end of the tutorial, this is the directory structure (for the `Medium_Imports_Tutorial` project) that you would be comfortable working with — in terms of importing any script(s) from one subdirectory into another (arrows in blue).

*Note: If you'd like to play along, here is the [Github](#) repo.*



Module: A single python script.

Package: A collection of modules.

## 함수와 메소드 method

- 함수와 메소드는 특정한 일을 하도록 정의된 동작을 수행하는 점에서 동일함
- 함수 - 호출될 때 매개변수를 전달받아(생략 가능) 특정한 동작을 수행하고 필요한 경우 그 결과를 반환하는 모든 프로그래밍 모듈
- 메소드 - 객체지향 프로그래밍에서 다루는 객체에 부속되는 함수
- 메소드는 인스턴스 instance라고 하는 각각 다른 값을 가진 객체들마다 호출할 수 있음
- 클래스에서 상세히 다루게 됨
- 템플릿 문자열 template string 혹은 베이스 문자열 base string
  - 코드를 대화창에 입력할 때 사용되는 문자열
  - 출력 메소드 format을 호출하는 문자열
- 플레이스홀더 placeholder
  - 인자의 출력을 목적으로 사용되는 중괄호

대화창 실습 : format() 메소드와 플레이스홀더의 인덱스  
플레이스 홀더에 들어갈 내용

```
>>> '{} Python!'.format('Hello')  
'Hello Python!'  
플레이스 홀더
```

```
>>> '{} Python!'.format('Hi')  
'Hi Python!'
```

대화창 실습 : format() 메소드와 플레이스홀더  
베이스 문자열, 템플릿 문자열

```
>>> '{0} Python!'.format('Hello')  
'Hello Python!'
```

대화창 실습 : format() 메소드와 플레이스홀더  
인덱스에 따라 Python, Java가 각각 들어감

```
>>> 'I like {} and {}'.format('Python', 'Java')  
'I like Python and Java'  
>>> 'I like {0} and {1}'.format('Python', 'Java')  
'I like Python and Java'
```

## 함수와 메소드

[출처] 유틸 파이썬, “4장 함수와 입출력”

- 플레이스홀더 내에 필요한 정수 값(인덱스)을 할당하여 출력 순서를 제어할 수 있다(디폴트로 0, 1, .. 이 할당됨)

```
'I like {} and {}'.format('Python', 'Java')  
      ↑   ↑  
      {0} {1}  
  
'I like [0] and [1]'.format('Python', 'Java')  
      ↑   ↑  
      {0} {1}
```

[그림 4-15] 플레이스홀더와 인덱스를 이용한 출력제어

```
'I like [1] and [0]'.format('Python', 'Java')  
      ↑   ↑  
      {1} {0}
```

[그림 4-16] 플레이스홀더와 전달인자

- {0}, {1}, {2}와 같이 플레이스홀더의 번호를 이용하여 다양한 출력을 할 수 있다
- 플레이스홀더에는 문자열뿐만 아니라 100, 200과 같은 정수형이나 실수형등 임의의 자료형도 올 수 있음

대화창 실습 : format() 메소드와 플레이스홀더의 인덱스

```
>>> 'I like {1} and {0}'.format('Python', 'Java')  
'I like Java and Python'
```

인덱스를 중복할 수 있음

대화창 실습 : format() 메소드와 플레이스홀더의 인덱스 사용법

```
>>> '{0}, {0}, {0}! Python'.format('Hello')  
'Hello, Hello, Hello! Python'  
  
>>> '{0}, {0}, {0}! Python'.format('Hello', 'Hi')  
'Hello, Hello, Hello! Python'  
  
>>> '{0} {1}, {0} {1}, {0} {1}!'.format('Hello', 'Python')  
'Hello Python, Hello Python, Hello Python!'  
  
>>> '{0} {1}, {0} {1}, {0} {1}!'.format(100, 200)  
'100 200, 100 200, 100 200!'
```

## 함수와 메소드

- `format()` 내부에는 문자열 리터럴 뿐만 아니라 다음과 같이 변수나 객체를 넣을 수 있음

대화창 실습 : 플레이스홀더 내의 객체 출력

```
>>> greet = 'Hello'  
>>> '{} World!'.format(greet)  
'Hello World!'
```

코드 4-33 : 플레이스홀더와 `format()` 메소드의 사용

```
print_format1.py  
name = 'Hong GilDong'  
print('My Name is {}!'.format(name))
```

실행결과

```
My Name is Hong GilDong!
```

[출처] 유틸 파일썬, “4장 함수와 입출력”

코드 4-34 : 플레이스홀더와 `format()` 메소드의 사용

`print_format2.py`

```
name = input('당신의 이름을 입력해주세요 : ')  
age = input('나이를 입력해주세요 : ')  
job = input('직업을 입력해주세요 : ')
```

```
print('당신의 이름은 {}, 나이는 {}살, 직업은 {}입니다.'.format(name, age, job))
```

실행결과

```
당신의 이름을 입력해주세요 : 김철수  
나이를 입력해주세요 : 21  
직업을 입력해주세요 : 학생
```

```
당신의 이름은 김철수, 나이는 21살, 직업은 학생입니다.
```

- `print()` 함수 내부에 있는 {}의 의미는 {}가 있는 곳에 `format()` 함수의 인자 값 name을 출력하라는 뜻
- 사용자의 이름과 나이를 입력으로 받아서 name, age라는 이름의 변수에 저장한 후 `format()` 메소드를 이용하여 출력이 가능
- 문자열의 `format()` 메소드를 사용해 {} 필드에 각각 이름(name)과 나이(age), 직업(job)이 위치하도록 문자열 포매팅을 함
- `format()` 메소드를 사용하지 않는다면 출력문은 복잡한 형태로 나타남. 이 경우 쉼표가 너무 많고 따옴표도 많아서 코드를 읽기도 힘들고 오류가 날 가능성도 매우 높다

```
print('당신의 이름은', name, '나이는', age, '살, 직업은', job, '입니다.')
```

## format() 메소드

[출처] 유틸 파이썬, “4장 함수와 입출력”

### 고급 format() 메소드

- 문자열 포매팅을 더욱 세밀하게 하기 위하여 format() 메소드의 고급 기능에 대하여 알아보기
- format() 메소드는 플레이스홀더 내에 콜론(:)을 찍고 출력의 크기와 형식 지정을 할 수 있다

대화창 실습 : 정수 표현을 위한 기본 포매팅

```
>>> for i in range(2, 15, 2):
....   print('{0} {1} {2}'.format(i, i*i, i*i*i))
....
```

2 4 8  
4 16 64  
6 36 216  
8 64 512  
10 100 1000  
12 144 1728  
14 196 2744

보기 불편한 출력

별도의 정렬기능이 없음

대화창 실습 : 출력 간의 크기 지정을 통한 정수 포매팅

```
>>> for i in range(2, 15, 2):
....   print('{0:3d} {1:4d} {2:5d}'.format(i, i*i, i*i*i))
....
```

오른쪽 정렬기능이 없음

2 4 8  
4 16 64  
6 36 216  
8 64 512  
10 100 1000  
12 144 1728  
14 196 2744

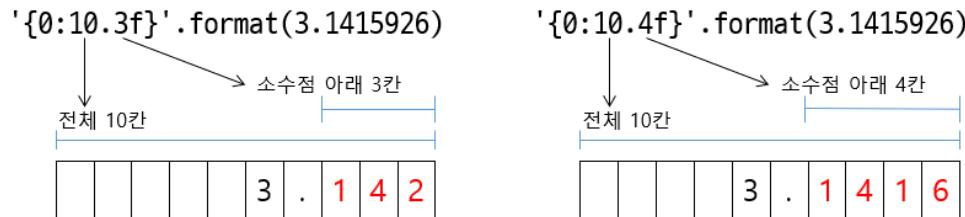
오른쪽으로 정렬된 보기 좋은 출력

대화창 실습 : 소수점 아래 자리수를 조절하는 실수 포매팅

```
>>> print('소수점 두 자리로 표현한 원주율 = {0:10.2f}'.format(3.1415926))
소수점 두 자리로 표현한 원주율 =      3.14
>>> print('소수점 세 자리로 표현한 원주율 = {0:10.3f}'.format(3.1415926))
소수점 세 자리로 표현한 원주율 =      3.142
>>> print('소수점 네 자리로 표현한 원주율 = {0:10.4f}'.format(3.1415926))
소수점 네 자리로 표현한 원주율 =      3.1416
```

## format() 메소드

[출처] 유틸 파이썬, “4장 함수와 입출력”



[그림 4-19] {0:10.3f}과 {0:10.4f} 포맷팅 출력의 예

- 플레이스홀더에 출력을 할 때는 key=value와 같이 키와 값을 인자로 넘겨주고 이 키를 이용한 출력도 가능
- 부산시의 위도 35.17N와 경도 129.07E를 출력하는 경우
- 순서에 상관없이 키와 값 형식으로 입력된 정보가 유지되지만 하면 정상적인 출력을 얻을 수 있다

대화창 실습 : 실수 표현을 위한 포맷팅에서 소수점

```
>>> print('1/3은 {:.3f}'.format(1/3))      # 소수점 아래 출력만 지정  
1/3은 0.333
```

대화창 실습 : 1000 단위 쉼표 출력방법

```
>>> print('{:,}'.format(1234567890))      # 1000단위 쉼표 출력  
1,234,567,890
```

대화창 실습 : 플레이스홀더 내에 키-값 형식으로 인자를 전달하는 방법

```
>>> print('위도 : {0}, 경도: {1}'.format('35.17N', '129.07E'))  
위도 : 35.17N, 경도: 129.07E  
  
>>> print('위도 : {lat}, 경도: {long}'.format(lat='35.17N', long='129.07E'))  
위도 : 35.17N, 경도: 129.07E  
  
>>> print('위도 : {lat}, 경도: {long}'.format(long='129.07E', lat='35.17N'))  
위도 : 35.17N, 경도: 129.07E
```

키워드 인자와 유사

키워드 인자와 유사

## built-in function (내장 함수)

[출처] 유틸 파이썬, “4장 함수와 입출력”

- 프로그래밍에서 함수는 흔히 **블랙박스**라는 비유를 함
- 함수를 호출하는 사용자는 제대로 된 입력 값을 주고 그 결과인 출력(결과 값)을 사용만 하면 되기 때문



[그림 4-18] 입력에 대해 정해진 출력을 내보내는 블랙박스 역할을 하는 함수



[그림 4-19] 내장함수 len()의 동작

- len() 함수가 어떤 값을 반환하는지 알고 있기 때문에 자세한 동작 과정을 몰라도 사용함
- print(), input() 함수도 이런 함수에 속함
- 파이썬에서 기본으로 구현되어 있어 제공하는 함수를 **파이썬의 내장함수**라고 한다

상세 설명 참조

<https://docs.python.org/ko/3/library/functions.html>

<https://wikidocs.net/32>

[표 4-2] 파이썬의 내장함수 목록

파이썬의 내장 함수				
abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	_import_()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

## built-in function (내장 함수)

대화창 실습 : 대화형 모드를 통한 여러 가지 내장 함수 실습

```
>>> abs(-100)                      # 절대값을 반환하는 함수  
100  
>>> min(200, 100, 300, 400)       # 여러 원소들 중 최솟값을 반환하는 함수  
100  
>>> max(200, 100, 300, 400)       # 여러 원소들 중 최댓값을 반환하는 함수  
400  
>>> str1 = "FOO"                  # "Foo" 혹은 'FOO' 형식으로 문자열 객체를 생성함  
>>> len(str1)                     # 문자열의 길이를 반환  
3  
>>> eval("100+200+300")          # 문자열을 수치값과 연산자로 변환하여 평가  
600  
>>> sorted("EABFD")              # 문자열을 정렬  
['A', 'B', 'D', 'E', 'F']  
>>> list = [200, 100, 300, 400]  
>>> sorted(list)  
[100, 200, 300, 400]  
>>> sorted(list, reverse=True)  
[400, 300, 200, 100]
```

[출처] 유틸 파이썬, “4장 함수와 입출력”

- `abs()` 함수는 -100이라는 정수를 입력 받아서 그 절댓값인 100을 반환
- `min()` 함수는 200, 100, 300, 400 의 값을 가지는 정수들 중에서 가장 작은 값을 반환
- `max()` 함수는 200, 100, 300, 400 의 값을 가지는 정수들 중에서 가장 큰 값을 반환
- `id()` 함수는 “FOO”라는 문자열이 저장된 변수 `str1`의 `identity`를
- `type()` 함수는 해당 변수의 자료형을 반환
- `len()` 함수는 변수 `str1`에 저장된 문자열의 길이를 반환
- `eval()` 함수는 문자열을 받아와 해당 문자열의 내용을 수식화해서 평가(evaluate)한 다음 평가한 값을 반환
- `sorted()` 함수는 문자열을 받아와 해당 문자열을 구성하는 문자들을 알파벳순으로 정렬해 반환

### • 파이썬은 객체지향 프로그래밍 언어 object oriented programming language

- 다양한 속성과 기능을 가진 객체들이 프로그램을 구성하는 패러다임이 객체지향 언어의 핵심
- 파이썬의 객체는 다른 객체와 구별되는 고유한 식별값identity를 가지고 있으며 `id()` 함수는 이 객체의 식별 값을 정수형으로 반환