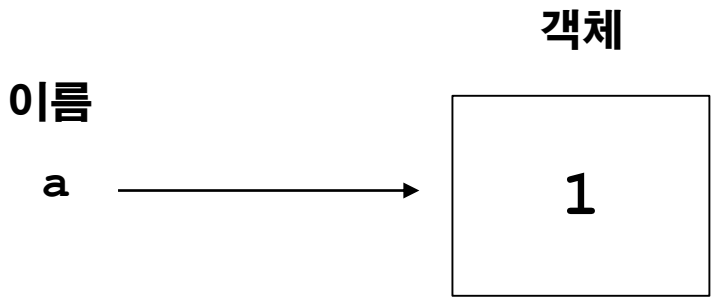
Python 프로그래밍 기초

객체

객체
: 이름과 객체

- 파이썬에서 모든 자료(데이터)들은 객체의 형태로 저장된다
- 파이썬의 변수는 컴파일러 언어처럼 변수에 할당된 값을 저장하는 저장공간(메모리)의 주소 심볼릭이 아니다
- 변수는 단지 객체의 이름(심볼)일 뿐이다
- 파이썬의 객체 이름(변수)과 객체의 ID(주소)는 심볼 테이블에 함께 저장되어 관계를 갖게 된다

```
a = 1
```



객체
: 심볼테이블

- 변수의 이름과 저장된 데이터의 주소를 저장하는 테이블
- 심볼테이블의 내용을 살펴보기 위해 globals(), locals() 내장 함수를 이용
- 두 함수는 해당 스코프 내 심볼 테이블의 내용을 dict 타입의 객체로 반환한다

```
>>> # 글로벌 변수 선언
>>> g_a = 1
>>> g_b = "symbol"
>>>
>>> def f(): # 로컬 변수 확인을 위한 함수 선언
...     l_a = 2
...     l_b = "table"
...     print(locals()) # 로컬 심볼테이블 확인
>>> f()
{'l_a': 2, 'l_b': 'table'}
>>> globals() # 글로벌 심볼테이블 확인
{'__doc__': None, '__loader__': <class
'_frozen_importlib.BuiltinImporter'>, '__package__': None,
'f': <function f at 0x104d06bf8>, 'g_a': 1, 'g_b':
'symbol', '__spec__': None, '__name__': '__main__',
'__builtins__': <module 'builtins' (built-in)>}
```

객체
: 레퍼런스 카운트와 쓰레기 수집

- 레퍼런스 카운트(Reference Count) : 객체를 참조하는 참조 수
- 레퍼런스 카운트가 0이 되면 사용하지 않는 객체로 판단, 자동으로 사라짐
- 이러한 작업을 쓰레기 수집(Garbage Collection)이라 함

```
>>> import sys
>>> x = object()
>>> sys.getrefcount(x)
2
>>> y = x
>>> sys.getrefcount(x)
3
>>> sys.getrefcount(y)
3
>>> del(x) # 레퍼런스 값이 줄어든다
>>> sys.getrefcount(y)
2
```

객체
: 객체 ID

- id() 함수를 이용하면 객체의 주소를 식별할 수 있다
  - 만일 두 객체의 ID가 동일하면, 같은 객체를 참조하고 있는 것

```
>>> i1 = 10
>>> i2 = 10
>>> print(hex(id(i1)), hex(id(i2)))
0x1067fb710 0x1067fb710
>>> l1 = [1, 2, 3]
>>> l2 = [1, 2, 3]
>>> print(hex(id(l1)), hex(id(l2)))
0x106a0be88 0x106b36b08
>>> s1 = "hello"
>>> s2 = "hello"
>>> print(hex(id(s1)), hex(id(s2)))
0x106f39110 0x106f39110
>>> i1 is i2
True
>>> l1 is l2
False
>>> s1 is s2
True
```

객체
: 객체의 복사

- 레퍼런스 복사
  - 객체를 참조하는 주소만 복사하는 것

```
>>> x = [1, 2, 3]
>>> y = x # 객체 참조 주소만 복사된다
>>> y
[1, 2, 3]
>>> hex(id(x)), hex(id(y))
('0x106b36d48', '0x106b36d48')
>>> x[1] = 4
>>> y
[1, 4, 3]
```

객체
: 객체의 복사

- [:] 이용한 복사
  - 객체 전체를 가리키는 [:] 를 이용하여 복사한다

```
>>> x = [1, 2, 3]
>>> y = x[:]
>>> y
[1, 2, 3]
>>> x is y
False
>>> x[1] = 4
>>> y
[1, 2, 3]
```

객체
: 객체의 복사

- copy 함수 이용
  - copy 모듈의 copy 함수를 사용하여 복사한다

```
>>> import copy
>>> x = [1, 2, 3]
>>> y = copy.copy(x)
>>> x is y
False
>>> x[1] = 4
>>> y
[1, 2, 3]
```

객체
: 객체의 복사

- deepcopy 함수 이용
  - copy 모듈의 deepcopy 함수를 사용하여 복사한다
  - deepcopy는 복합객체를 재귀적으로 생성하고 복사한다

```
>>> a = [1, 2, 3]
>>> b = [4, 5, a]
>>> x = [a, b, 100]
>>> import copy
>>> y = copy.deepcopy(x)
```

Python 프로그래밍 기초

제어문 (조건문과 반복문)

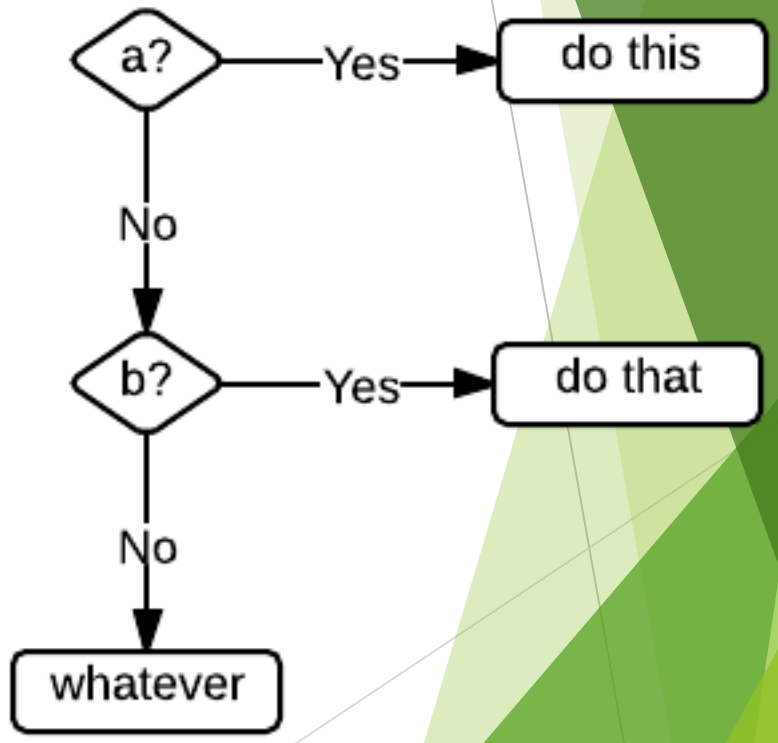
조건문
: if - elif - else

```
if( 조건식1 ) :
    구문1
    구문2

elif( 조건식2 ) :
    구문3
    구문4

else :
    구문 5
```

```
if a:
    do this
elif b:
    do that
else:
    whatever
```



조건문
: if – elif - else

```
>>> n = -2
>>> if n > 0:
…     print("양수")
... elif n < 0:
…     print("음수")
... else:
…     print("0")
…
음수
```

조건문
: 조건 표현식(Conditional Expression)

- C 또는 Java의 3항 연산자와 같은 역할을 한다

```
value = {true-expr} if {condition} else {false-expr}
```

```
>>> money = 8500
>>> print("by taxi" if money > 10000 else "by bus")
by bus
```

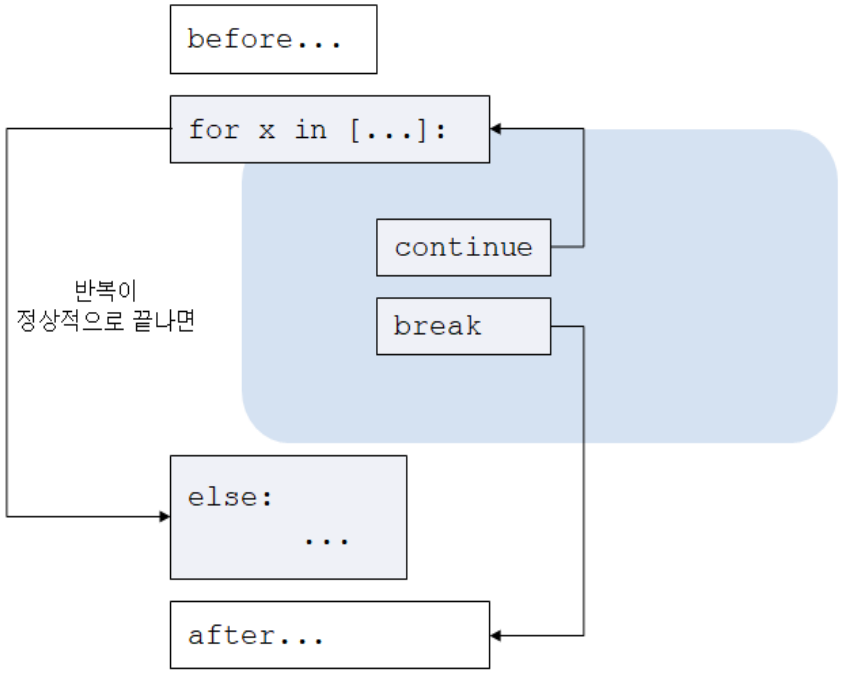
조건문
: in, not in

- 파이썬은 리스트, 튜플, 문자열에 in, not in 등 편리한 조건식을 제공

in	not in
x in 리스트	x not in 리스트
x in 튜플	x not in 튜플
x in 문자열	x not in 문자열

```
>>> 1 in [1, 2, 3]
True
>>> 1 not in [1, 2, 3]
False
>>> "y" in "Python"
True
>>> "y" not in "Python"
False
```

반복문
: for




```
for {타킷} in {객체}:
    구문1
    구문2
else :
    구문 3
```

- {객체}는 list, str, tuple, bytes, bytearray, range 등 시퀀스 자료형
- 반복횟수는 {객체}의 크기
- {객체} 안의 객체를 하나씩 순차적으로 꺼내어 구문1,2가 실행됨
- 반복이 정상적으로 끝나면 else 블록이 실행
- for문 내에서 break로 빠져나오면 else 블록은 실행되지 않음

반복문
: for

```
>>> # list 객체를 이용한 for 문
>>> animals = ['cat', 'cow', 'tiger']
>>> for animal in animals:
...     print(animal, end = " ")
cat cow tiger

>>> # range 객체를 이용한 for 문
>>> for x in range(1, 10, 3):
...     print(x, end = " ")
1 4 7

>>> # for ~ else 문의 활용
>>> for x in data:
...     if x > 10:
...         break
... else:
...     print("10보다 큰 수 없음")
```

반복문
: enumerate

- 요소의 값은 물론 인덱스가 필요할 경우 enumerate() 함수를 이용한다

```
>>> colors = ['red', 'orange', 'yellow', 'green', 'pink', 'blue']
>>> for index, color in enumerate(colors):
...     print(index, color)
...
0 red
1 orange
2 yellow
3 green
4 pink
5 blue
```

반복문
: break

- 어떤 조건에서 반복을 중지하고 빠져나가야 하는 경우 break문

```
>>> l = [1, 3, 5, 7, 9, 11, 12, 13, 15, 17]
>>> for x in l:
...     if x % 2 == 0:
...         break;
...     print(x)
1
3
5
7
9
11
```

반복문
: continue

- Continue문을 만나면, 이후 구문은 실행하지 않고 처음으로 이동한다

```
>>> for x in range(10):
...     if x % 2 == 0:
...         continue
...     print(x, end = " ")

1 3 5 7 9
```

반복문
: List 내포

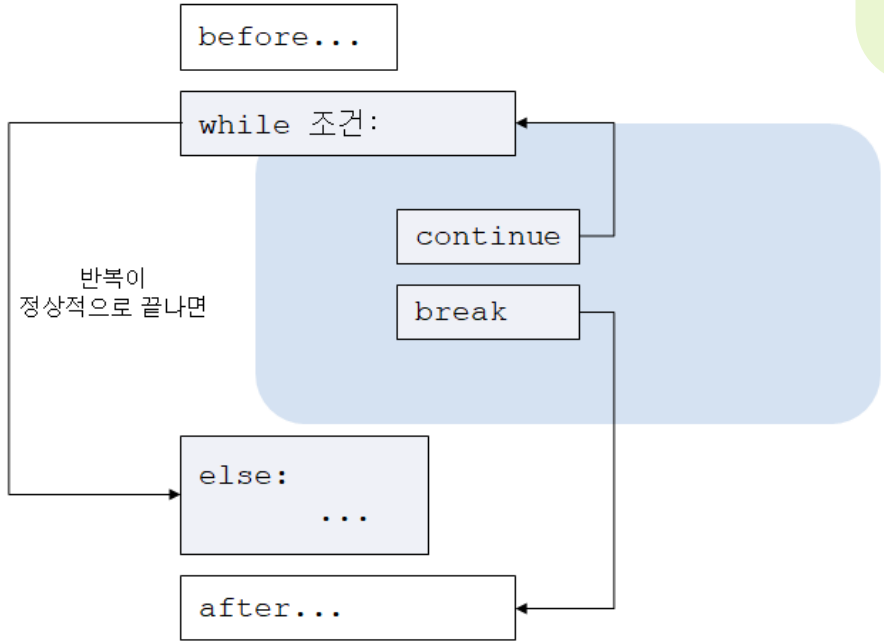
- 리스트 내포(List Comprehension)를 이용하면 좀더 직관적인 프로그램을 만들 수 있다

```
[{표현식} for {항목} in {객체} if {조건}]
```

```
>>> a = [1, 6, 4, 13, 8, 3]
>>> a 리스트 항목 중, 짝수인 것만 2배 하여 result에 저장
>>> result = [num * 2 for num in a if num % 2 == 0]
>>> result
[12, 8, 16]
```

반복문
: while

```
while {조건식}:
    구문1
    구문2
else:
    구문 3
```



- {조건}이 참인 동안 구문1, 2가 반복 실행
- Else 블록은 while문을 빠져나올 때 실행
- 단, break로 while문을 빠져나올 경우 else 블록은 실행되지 않는다
- 무한루프(실행이 종료되지 않고 계속 실행되는 반복)에 빠지지 않도록 유의
- 경우에 따라서는 의도적으로 무한루프를 돌리기도 한다

반복문
: while

```
>>> counter = 1
>>> while counter < 11:
...     print(counter, end = " ")
...     counter += 1
... else:
...     print("")
...
1 2 3 4 5 6 7 8 9 10
```

```
>>> sum, i = 0, 1
>>> while i <= 100: # 1~100까지의 정수 합 구하기
...     sum += i
...     i += 1
...
>>> sum
5050
```

반복문
: while 내부에서 break, continue, else 사용하기

```
>>> i = 0
>>> while i < 100:
...     i += 1
...     if i < 5:
...         continue
...     print(i, end = " ")
...     if i > 10:
...         break
... else:
...     print("else block") # 이 블록은 실행되지 않을 것임. WHY?
...
5 6 7 8 9 10 11
```

반복문
: 무한루프

- 조건을 True로 주면 무한루프를 구성할 수 있다
- break 문으로 루프를 탈출할 수 있는 조건이 있어야 한다

```
>>> while True:
...     print("Ctrl+C를 눌러 루프를 종료하십시오.")
...
Ctrl+C를 눌러 루프를 종료하십시오.
Ctrl+C를 눌러 루프를 종료하십시오.
...
```

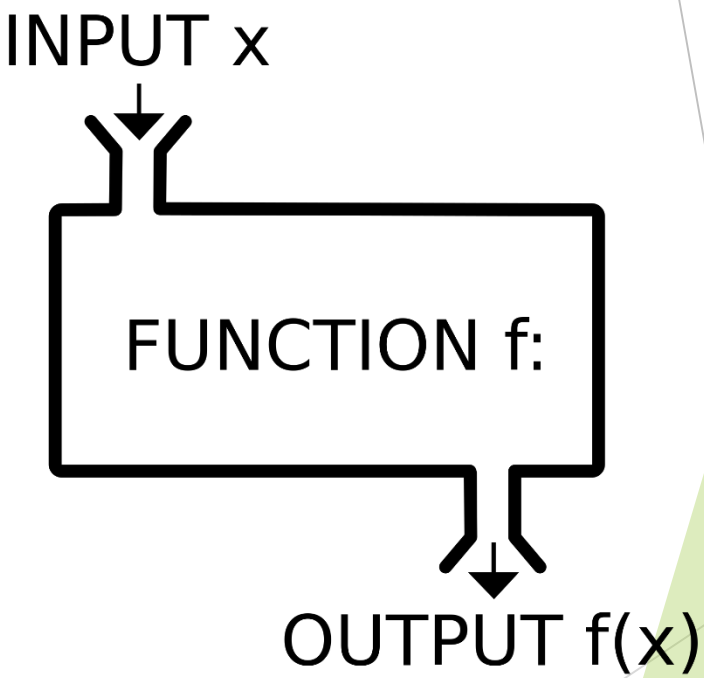
Python 프로그래밍 기초

함수(Function)

함수
: 함수란

- 입력값을 가지고 어떤 일을 수행한 다음 그 결과물을 내 놓는 것

- 함수를 사용하는 이유
  - 반복되는 부분이 있을 경우 재활용을 위해
  - 프로그램의 흐름을 일목요연하게 볼 수 있다



함수 정의하기

- def 키워드를 이용하여 정의
- 함수 이름과 인수들이 기술
- 함수 선언부는 :로 끝난다
- 들여쓰기 규칙이 적용
- 함수의 끝은 들여쓰기가 적용 안되는 라인에서 끝난다

함수 정의하기

```
def dummy():
    pass   # 실행할 내용이 없을때는 pass

def my_function():
    print("Hello World")

def times(a, b):
    return a * b # 결과값을 돌려줘야 할 때는 return 문으로 반환

def do_nothing():
    return # return 문만 썼을 경우, None이 반환

dummy()
my_function()
print(times(10, 10))
print(do_nothing())
```

함수
: 함수도 객체다

- 함수도 객체이므로 다음과 같은 호출도 가능하다

```
t = times
print(t(100, 100))
print(t, times, sep = ",")
```

함수
: return

- 함수를 종료시키고, 해당 함수를 호출한 곳으로 되돌아 가게 한다
- 파이썬에서는 어떤 종류의 객체도 반환할 수 있다
- 여러 객체를 return하면 튜플로 반환한다
- return문을 만나면 함수는 종료한다
- return문만 사용하면 None을 반환한다
- 함수가 끝날 때까지 종료할 필요가 없고 반환할 값이 없을 때는 return문이 없어도 된다

함수
: return문 활용

```
# 인수 없이 반환하기
def do_nothing():
    return # None을 반환

# return 문이 필요없는 경우도 있다
def say_hello():
    print("Life is too short, You need Python")

# 한 개의 값을 반환
def max_value(a, b):
    if a > b:
        return a
    else:
        return b
```

함수
: return문 활용

```
# 여러 값을 반환할 때
def swap(a, b):
    return b, a

print(swap(10, 20))

# 결과값은 튜플로 반환된다
```

함수
: 인수의 전달 방법

- 기본적으로 참조에 의한 호출(Call-by-reference)이다
- 하지만 인수의 타입이 변경가능(mutable), 변경불가(immutable)에 따라 처리 방식이 달라진다

```
# 변경 가능 객체를 인수로 전달할 경우
def g(t):
    t[0] = 0

a = [1, 2, 3]
g(a)
print(a)
```

함수
: 인수의 전달 방법

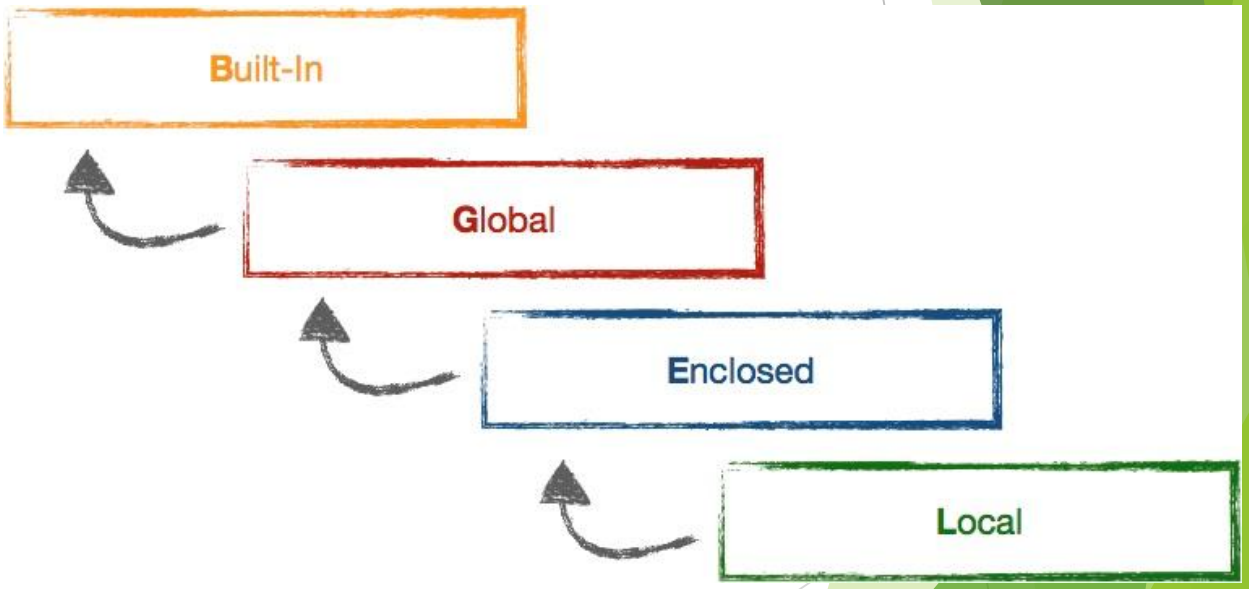
- 기본적으로 참조에 의한 호출(Call-by-reference)이다
- 하지만 인수의 타입이 변경가능(mutable), 변경불가(immutable)에 따라 처리 방식이 달라진다

```
# 변경 불가 객체를 인수로 전달했을 때
def h(t):
    t = (10, 20, 30)

a = (1, 2, 3)
h(a)
print(a) # a 객체는 변경되지 않는다
```

함수
: 스코핑 룰(Scope)

- 이름공간(Namespace)
  - 프로그램에서 사용되는 이름들이 저장되는 공간
- 이름은 값을 치환할 때 해당 값의 객체와 함께 생겨나고 이름공간에 저장
- 이름공간에 저장된 이름을 통해 객체를 참조
- 이름공간의 종류
  - Local Scope: 함수 내부
  - Enclosed: function in function
  - Global: 모듈 내부
  - Built-in: 내장 영역
- 동일한 이름이 여러 영역에 있다면 LEGB 순으로 찾는다



함수
: 스코핑 룰 예제

```
x = 1
def func(a):
    return a + x # Local 스코프에 x 가 없으므로 Global x를 사용한다

def func2(a):
    x = 2
    return a + x # Local 스코프에 x가 있으므로 Local x를 사용한다

print(func(10))
print(func2(10))
print(x)
```

함수
: 스코핑 룰 예제

- 함수 내부에서 전역 객체를 사용해야 하는 경우 **global** 선언문을 이용한다
- 가능하면 함수 내부에서 글로벌 객체를 직접 사용하는 것은 피한다

```
g = 1
def func3(a):
    global g
    g = 3 # 본 객체는 글로벌 객체이다
    return a + g

print(func3(10))
print(g)
```

함수
: 함수의 인수

- 인수는 필요한 개수만큼 선언할 수 있다
- 기본값이 필요하면 함수 선언시 지정할 수 있다

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

def incr(a, step = 1): # 두 번째 인자의 기본값은 1
    return a + step

print(sum(2, 3))
print(incr(10)) # 두 번째 인자의 기본값을 사용한다
print(incr(10, 2)) # 두 번째 인자를 직접 지정한다
```

함수
: 함수의 인수 - 키워드 인수

- 인수값을 인수 이름으로 전달할 수 있다(함수의 정의에 따른다)

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

print(area(10, 12))
print(area(height = 4, width = 3)) # == area(3, 4)
```

함수
: 함수의 인수 - 가변인수

- 정해지지 않은 개수의 인수값을 받을 때 사용한다
- 선언시 인수명 앞에 *를 붙여 선언한다

```
def get_total(*args):
    sum = 0
    for x in args:
        sum += x
    return sum

print(get_total(1, 3, 5, 7, 9))
```

함수
: 함수의 인수 - 사전 키워드 전달

- 정해지지 않은 키워드 인수는 모두 dict로 받을 수 있다
- 선언시 인수명 앞에 **를 붙인다
- 사전 키워드 인수는 선언의 맨 마지막에 있어야 한다

```
def f(a, b, *args, **kwd):
    print(a, b)
    print(args)
    print(kwd)

f(10, 20, 30, 40, depth = 10, dimension = 3)
```

함수
: 함수 객체를 인수로 전달

- 파이썬에서는 함수도 객체이다
- 따라서 인수로 함수를 전달하는 것도 가능하다

```
states = ['Alabama', ' Georgia', 'Georgia ', 'georgia', 'FlOrIda', 'south carolina
', 'West virginia']

def clean_strings(strings, *funcs): # 함수 목록을 가변인수로 전달
    results = []
    for string in strings:
        for func in funcs: # 전달된 함수들을 순차적으로 적용
            string = func(string)
        results.append(string)
    return results

states = clean_strings(states, str.strip, str.title)
print(states)
```

함수
: 익명 함수(Lambda)

- 이름이 정의되지 않은 '익명 함수'를 선언
- 데이터 분석/변형 함수에서 파라미터로 처리 함수를 인자로 받는 경우가 많다

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

for i in range(10):
    print("{0}:{1}".format(i, square(i)), end = " ")
else:
    print()

# Same as above with Lambda
for i in range(10):
    print("{0}:{1}".format(i, (lambda x: x * 2)(i)), end = " ")
else:
    print()
```

함수
: Lambda를 이용한 정렬

- 정렬할 때, key 함수로 정의하기에도 편리한 경우가 많다

```
strings = ['foo', 'card', 'bar', 'abab', 'aaaa', 'abab', 'foo']

strings.sort(key=lambda x: len(x))
print(strings)

strings.sort(key=lambda x: strings.count(x))
print(strings)
```

파이썬 프로그래밍 기초

예외처리 (Handling Exceptions)

예외 처리

- 오류는 프로그램이 잘못 작동되는 것을 막기 위한 파이썬의 처리
- 이 오류를 회피하기 위한 동작을 예외 처리라 한다

```
>>> list = []
>>> list[0] # 내부에 0 인덱스에 매칭되는 값이 없으므로 IndexError를 발생
IndexError: list index out of range

>>> text = "Try convert me to int"
>>> number = int(text) # 정수 형식의 문자열이 아니어서 변환이 불가: ValueError
ValueError: invalid literal for int() with base 10: 'Try convert me to int’

>>> result = 4/0 # 0으로는 나눗셈을 할 수 없다. ZeroDivisonError 발생
ZeroDivisionError: division by zero
```

예외 처리
: 오류의 회피 - try, except

```
try:
    # Do Something
except:
    # Do Something when Error occured
```

```
>>> try:
...     4 / 0
... except:
...     print("오류 발생")

오류 발생
```

예외 처리
: 특정 오류의 회피 - try, except

```
try:
    # Do Something
except {발생오류}:
    # Do Something when {발생오류} occured
```

```
>>> try:
...     4 / 0
... except ZeroDivisionError:
...     print("오류 발생")

오류 발생
```

예외 처리

: 특정 오류의 회피 - try, except

```
try:
    # Do Something
except {발생오류} as {오류변수}:
    # Do Something when {발생오류} occured
```

```
>>> try:
...     4 / 0
... except ZeroDivisionError as e: # 발생한 오류의 정보를 e 에 담아 사용한다
...     print(e)

division by zero
```

예외 처리

: - try, except, else

```
try:
    # Do Something
except {발생오류} as {오류변수}:
    # Do Something when {발생오류} occurred
else:
    # 예외가 발생하지 않은 경우 실행
```

```
>>> try:
...     f = open("notexists.txt", "r")
... except FileNotFoundException as e:
...     print(e)
... else:
...     data = f.read()
...     f.close()
```

예외 처리

: try ~ finally

```
try:
    # Do Something
except {발생오류} as {오류변수}:
    # Do Something when {발생오류} occurred
finally:
    # 예외 발생 여부와 상관 없이 마지막에 항상 수행
```

```
>>> f = open("result.txt", "r")
>>> try:
...     print(f.read())
... finally:
...     f.close() # 오류 여부와 상관 없이 파일 close 작업을 수행한다
```

Python 프로그래밍 기초

파일 입출력

파일 입출력 개요
: 파일의 생성과 파일 모드

```
파일객체 = open({파일명}, {파일모드}[, encoding='인코딩'])
```

파일 모드	설명
r (default)	읽기 모드 - 파일을 읽기만 할 때 사용
w	쓰기 모드 - 파일에 내용을 기록할 때 사용
a	추가 모드 - 파일의 마지막에 사로운 내용을 추가할 때 사용

파일 모드	설명
t (default)	텍스트 모드
b	바이너리 모드

파일 입출력 개요
: 파일 제어 기본 함수

함수명	설명
open	파일을 생성한다
write	파일에 내용을 기록한다
read	파일에서 내용을 읽어온다
close	파일 사용을 끝낸다. 파일을 열었으면(open) 반드시 사용후 닫아주도록 한다

```
>>> # File Write Sample
>>>
>>> f = open('text.txt', 'w', encoding='utf-8') # text.txt, 쓰기모드
>>> write_size = f.write("Life is too short, You need Python")
>>> print(write_size)
33
>>> f.close() # 반드시 닫아주자
```

파일 입출력 개요
: 텍스트 파일 예제

```
# File Write

f = open('test.txt', 'w', encoding='utf-8')
for i in range(1, 10):
    f.write("%d: Life is too short, You need Python\n" % i)
f.close()

# File Read

f = open('test.txt', 'r', encoding='utf-8')
text = f.read()
print(text)
f.close()
```

파일 입출력 개요
: 텍스트 파일 예제 - write and read

```
# File Write

f = open('multilines.txt', 'w', encoding='utf-8')
for i in range(1, 10):
    f.write("%d: Life is too short, You need Python\n" % i)
f.close()

# File Read

f = open('multilines.txt', 'r', encoding='utf-8')
text = f.read()
print(text)
f.close()
```

파일 입출력 개요
: readline 함수를 이용한 텍스트 파일 읽기

- readline 함수를 이용하면 텍스트 파일을 줄 단위로 읽어올 수 있다

```
f = open('multilines.txt', 'r')

while True:
    line = f.readline()
    if not line:
        break # 무한루프 탈출
    print(line)

f.close()
```

파일 입출력 개요
: readlines 함수를 이용한 텍스트 파일 읽기

- readlines 함수를 이용하면 모든 라인을 불러 리스트로 제공한다

```
f = open('multilines.txt', 'r')

lines = f.readlines()
# print(lines)

for line in lines:
    print(line)

f.close()
```

파일 입출력 개요
: 바이너리(Binary) 파일 다루기

- 바이너리 파일을 다루려면 모드를 바이너리로 지정해야 한다

```
>>> f = open('python.png')
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/namsk/.pyenv/versions/3.4.3/lib/python3.4/codecs.py", line
319, in decode
    (result, consumed) = self._buffer_decode(data, self.errors, final)
UnicodeDecodeError: 'utf-8' codec can't decode byte 0x89 in position 0:
invalid start byte
```

파일 입출력 개요
: 바이너리(Binary) 파일 다루기

- 바이너리 파일을 다루려면 모드를 바이너리로 지정해야 한다

```
>>> # copy binary sample
>>>
>>> f_src = open('python.png', 'rb') # 바이너리 읽기 모드
>>> data = f_src.read()
>>> f_src.close()
>>>
>>> f_dest = open('python_copy.png', 'wb') # 바이너리 쓰기 모드
>>> f_dest.write(data)
11155
>>> f_dest.close()
```

파일 입출력 개요
: 그 외 파일 관련 함수

함수명	설명
seek	사용자가 원하는 위치로 파일 포인터 이동
tell	현재 파일에서 어디까지 읽고 썼는지 위치를 반환

```
f = open('multilines.txt', 'r', encoding='utf-8')

text = f.readline()
print(text)
pos = f.tell() # 현재의 파일 포인터를 얻어옴
print(pos)

f.seek(16)
text = f.read()
print(text)
```

파일 입출력 개요
: with ~ as – 자동 자원 정리

- with ~ as 를 이용, 파일 입출력을 수행하면 수동으로 파일을 close 하지 않아도 된다

```
with open('multilines.txt', 'r') as f_as:
    for line in f_as.readlines():
        print(line, end = "")

print(f_as.closed) # 파일이 close 되었는지 점검
```

Using Pickle

- 객체의 내용을 파일에 저장하거나 복원해야 할 경우에 Pickle 모듈을 사용하면 편리
- Pickle 모듈은 객체를 파일에 썼다가 나중에 복원할 수 있도록 객체를 바이트 스트림으로 직렬화
  - 모든 파이썬의 객체를 저장하고 읽을 수 있음
  - 원하는 객체를 형태 변환 없이 쉽게 쓰고 읽을 수 있다
- Pickle 모듈을 사용하려면 import pickle 을 이용, 모듈을 로드해야 한다
- Pickle 모듈 주요 메서드

메서드	설명
dump(data, file [, protocol])	data 객체를 [protocol 을 이용해] file에 저장
load(file)	File로부터 저장된 객체를 불러옴

Using Pickle
: 객체의 저장:Pickling - dump

- file에 객체를 저장하고자 할 때에는 dump 메서드를 이용한다

```
import pickle
f = open("players.bin", "wb")
data = {"baseball": 9}
pickle.dump(data, f)
f.close()
```

- dump 메서드에 프로토콜 버전을 정의해 줄 수 있다
  - 최신 프로토콜 버전을 확인하려면 pickle.HIGHEST_PROTOCOL로 확인

```
...
pickle.dump(data, f, pickle.HIGHEST_PROTOCOL)
...
print(pickle.HIGHEST_PROTOCOL)
```

Using Pickle
: 객체의 복원:Unpickling - load

- file에 객체로부터 객체를 불러올 때에는 load 메서드를 이용한다

```
import pickle
f = open("players.bin", "rb")
data = pickle.load(f)
f.close()
print(data)
```

- dump시에 PROTOCOL을 지정했다 하더라도 load할 때는 지정해주지 않아도 된다
  - pickle 파일에 PROTOCOL 버전이 저장되어 있음

Using Pickle
: 복수 객체의 저장

- 기본적으로 Pickle은 단일 객체를 저장하는 포맷이지만, dump 메서드를 중복하여 사용하면 복수 개의 객체를 저장할 수 있다

```
import pickle
with open("players.bin", "wb") as f:
    pickle.dump({"baseball": 9}, f)
    pickle.dump({"soccer": 11}, f)
    pickle.dump({"basketball": 5}, f)
```

Using Pickle
: 복수 객체의 복원

- 저장된 객체를 복원하려면 load 메서드를 이용
  - load가 수행될 때마다 한줄씩 불러들이며 더 이상 불러올 객체가 없을 때 EOFError 발생
  - 다음 코드를 수행해 보고 무엇이 문제인지 확인해 봅니다

```
>>> import pickle
>>> with open("players.bin", "rb") as f:
...     print(pickle.load(f))
...     print(pickle.load(f))
...     print(pickle.load(f))
...     print(pickle.load(f))
...
{'baseball': 9}
{'soccer': 11}
{'basketball': 5}
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
EOFError: Ran out of input
```

Using Pickle
: 복수 객체의 복원

- [Solution]

```
>>> import pickle
>>> with open("players.bin", "rb") as f:
...     data_list = []
...     while True:
...         try:
...             data = pickle.load(f)
...         except EOFError:
...             break
...         data_list.append(data)
...
>>> print(data_list)
[{'baseball': 9}, {'soccer': 11}, {'basketball': 5}]
```

Using Pickle

- Pickle 사용시 유의사항
  - Pickle은 단순 텍스트 저장이 아닌 바이트 스트림 직렬화를 이용한 것이므로 파일 모드는 반드시 "b" 모드 (wb / rb) 로 지정해야 한다
  - Pickle에 사용되는 데이터 포맷은 파이썬에 특화되어 있기 때문에 다른 언어로 작성된 응용프로그램과의 데이터 교환에는 사용하지 않는 것이 좋다
  - 저장된 데이터에 대한 보안을 제공하지 않는 점에 유의하여 사용

파이썬 프로그래밍 기초

날짜와 시간

날짜와 시간

- 날짜와 시간은 파이썬에서 기본으로 제공하는 자료형에는 포함되어 있지 않지만 중요한 자료형
- 날짜와 시간은 각각 date 객체, time 객체를 이용하며 이들 객체는 datetime 모듈에 포함되어 있다
  - import datetime 을 이용, 해당 모듈을 불러올 수 있다
  - 주요 클래스
    - datetime : 날짜와 시간을 모두 포함하는 클래스
    - date : 날짜 관련 클래스
    - time : 시간 관련 클래스

날짜와 시간
: 날짜와 시간의 획득

- datetime 모듈 내 datetime 클래스의 now() 메서드를 이용하면 현재 날짜와 시간을 획득할 수 있다

```
>>> import datetime     # datetime 모듈 불러오기
>>> dt = datetime.datetime.now()
>>> dt
datetime.datetime(2017, 5, 2, 14, 59, 56, 411440)
(year, month, day, hour, minute, second, microsecond)
```

- datetime 클래스 생성자로 직접 날짜를 지정할 수 있다
  - 년, 월, 일, 시, 분, 초, 마이크로초 순으로 매개변수를 전달하면 되며, 최소 년월일은 지정해 주어야 한다

```
>>> import datetime     # datetime 모듈 불러오기
>>> dt = datetime.datetime(1999, 12, 24)
>>> dt
datetime.datetime(1999, 12, 24, 0, 0)
```

날짜와 시간
: datetime 클래스

- datetime 클래스의 주요 속성

속성	설명
year	년
month	월
day	일
hour	시
minute	분
second	초
microsecond	마이크로초

```
>>> import datetime
>>> dt = datetime.datetime.now()
>>> dt.year, dt.month, dt.day, dt.hour, dt.minute, dt.second, dt.microsecond
(2017, 5, 2, 15, 15, 22, 625369)
```

날짜와 시간
: datetime 클래스

- datetime 클래스의 주요 메서드

메서드	설명
weekday()	요일을 반환
strftime()	datetime 을 문자형으로 포맷
date()	날짜 정보만 가지는 date 클래스로 변환
time()	시간 정보만 가지는 time 클래스로 변환

```
>>> import datetime
>>> dt = datetime.datetime.now()
>>> dt.weekday() # 월:0, 화:1, 수:2, 목:3, 금:4, 토:5, 일:6
2
>>> dt.date()
datetime.date(2018, 5, 2)
>>> dt.time()
datetime.time(15, 29, 42, 821062)
```

- date는 datetime 클래스의 날짜 관련 속성과 요일 메서드를 이용 가능
- time은 datetime 클래스의 시간 관련 속성 이용 가능

날짜와 시간
: datetime의 비교와 두 datetime의 차이 구하기

- 두 개의 날짜는 대소 비교, 차이 값을 구할 수 있다

```
>>> import datetime
>>> current = datetime.datetime.now()
>>> past = datetime.datetime(1999, 12, 31)
>>> current > past # 현재 날짜가 과거보다 큰가?
True
>>> diff = current - past #
>>> diff
datetime.timedelta(6697, 56169, 742005)
>>> diff.days, diff.seconds, diff.microseconds
(6697, 56169, 742005)
```

- 두 날짜의 차이는 datetime.timedelta 클래스로 반환, 날짜의 차이를 구할 수 있다
  - days, seconds, microseconds

날짜와 시간
: timedelta

- timedelta 클래스의 주요 속성
  - days : 일수
  - seconds: 초 (0 ~ 863999)
  - microseconds: 마이크로초 (0 ~ 999999)
- timedelta 클래스의 메서드
  - total_seconds() : 모든 속성을 초단위로 모아서 변환

- 두 datetime의 차이가 timedelta로 반환되었던 것과 반대로 datetime 클래스에 timedelta 값을 더해 새로운 날짜를 만들 수도 있다

날짜와 시간
: timedelta

- datetime과 timedelta의 합산

```
>>> import datetime
>>> current = datetime.datetime.now()
>>> current
datetime.datetime(2018, 5, 2, 15, 55, 19, 406471)
>>> diff = datetime.timedelta(days=30, seconds=0, microseconds=0)
>>> current + diff
datetime.datetime(2018, 6, 1, 15, 55, 19, 406471)
```

날짜와 시간
: datetime을 문자열로 변환

- datetime의 strftime 메서드를 이용하면 datetime 형식을 문자열로 변환할 수 있다
- strftime의 출력 형식 기호

기호	표시되는 형식
%Y	서기 년도 4자리 표시
%y	서기 년도 2자리 표시
%m	달을 두 자리로 표시
%d	일을 두 자리로 표시
%B	영어로 달을 표시
%b	영어로 달을 단축 표시
%A	영어로 요일을 표시
%a	영어로 요일을 단축 표시

기호	표시되는 형식
%H	시간 표시(24시간)
%I	시간 표시(12시간)
%p	AM/PM 표시
%M	분을 두 자리로 표시
%S	초를 두 자리로 표시
%f	마이크로초를 6자리수로

날짜와 시간
: datetime을 문자열로 변환

- 다양한 형식으로 datetime을 문자열로 포맷하기

```
>>> import datetime
>>> current = datetime.datetime.now()
>>> current.strftime('%Y/%m/%d')
'2018/05/02'
>>> current.strftime('%Y년 %m월 %d일') # 하단 참고
'2018년 05월 02일'
>>> current.strftime('%Y-%m-%d %H-%M-%S')
'2018-05-02 16-09-31'
```

- 참고 : Windows에서는 Locale로 인해 UnicodeEncodeError가 발생할 수 있다 다음과 같이 조치한다

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'ko_KR.UTF-8')
'ko_KR.UTF-8'
>>> current.strftime('%Y년 %m월 %d일')
'2018년 05월 02일'
```

날짜와 시간
: 문자열을 datetime으로 변환

- datetime 클래스의 strptime 메서드를 이용하면 문자열로부터 날짜와 시간 정보를 읽어서 datetime 클래스로 변환할 수 있다
  - strptime의 인수
    - 첫 번째 인수 : 날짜와 시간 정보를 가진 문자열
    - 두 번째 인수 : 문자열 해독을 위한 형식 문자열

```
>>> import datetime
>>> str = '2017-12-24 23:59'
>>> datetime.datetime.strptime(str, '%Y-%m-%d %H:%M')
datetime.datetime(2017, 12, 24, 23, 59)
```