

파이썬의 조립 블록 구성 요소

Assembly Block Components



서지혜 교수

서울과학기술대학교

jihae@seoultech.ac.kr



계산기 구현하기

- 연산자와 정수 두 개를 입력받아 수치 연산을 수행한 후 결과를 반환하는 **calculator** 함수를 구현
- 함수를 정의할 때 세 개의 매개함수를 아래 순서로 정의
 - operator, integer1, integer2**
 - 만약 **operator**가 **'add'**이면, 두 정수에 더하기 연산을 수행 : **integer1 + integer2**
 - 만약 **operator**가 **'sub'**이면, 두 정수에 빼기 연산을 수행 : **integer1 - integer2**
 - 만약 **operator**가 **'mul'**이면, 두 정수에 곱하기 연산을 수행 : **integer1 * integer2**
 - 만약 **operator**가 **'div'**이면, 두 정수에 나누기 연산을 수행 : **integer1 / integer2**
- 실행 결과 예시

```
>>> calculator('addition', 3, 5)
addition is not supported.
```

```
>>> calculator('add', 3, 5)
8
```

```
>>> calculator('sub', 7, 8)
-1
```

```
>>> calculator('mul', 11, 2)
22
```

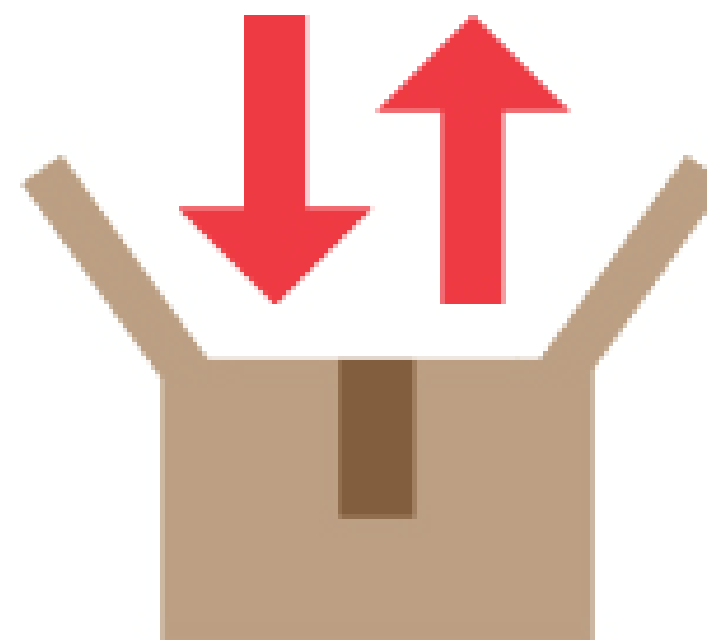
```
>>> calculator('div', 7, 3)
2.3333333333333335
```

```
>>> calculator('add', '3', '5')           # 문자열 더하기
'35'
```

```
>>> calculator('add', [1, 2], [3, 4])     # 리스트 더하기
[1, 2, 3, 4]
```

매개변수와 전달인자

Parameters and Arguments



- 매개변수(parameter)란?
 - 함수를 정의할 때 괄호 안에 선언한 식별자(변수)며,
 - 전달인자를 함수 내부로 가져와 처리할 목적으로 사용
- 매개변수 구분
 - **필수 매개변수**
 - **기본값**이 설정되지 않은 매개변수
 - 기본값이 미리 설정되어 있지 않기 때문에 함수를 실행(호출)할 때 반드시 전달인자를 지정해야 하며, 지정하지 않으면 오류가 발생
 - 예) `myfunction(length, weight)`
 - **선택 매개변수**
 - **기본값**이 미리 설정되어 있는 매개변수
 - 기본값이 미리 설정되어 있기 때문에 함수를 실행(호출)할 때 전달인자를 지정하지 않으면 해당 함수가 기본값으로 명령문을 실행
 - 따라서, 기본값이 아닌 다른 값을 사용하고자 할 때만 지정하면 된다
 - 예) `myfunction(copies=1, color=None)`

함수를 정의할 때 매개변수는 크게 세 가지 종류로 나눌 수 있다

매개변수가 없다

- 매개변수가 0개(즉, 전달인자 없이 함수의 기능을 실행)
- 예) `def myfunction():`

위치 매개변수(positional parameter)

- 쉼표로 구분하는 한 개 이상의 식별자로 정의하는 매개변수
- 예) `def myfunction(length, weight):`

키워드 매개변수(keyword parameter)

- 쉼표로 구분하는 한 개 이상의 식별자=기본값 형태의 쌍으로 정의하는 매개변수
- 예) `def myfunction(copies=1, color=None):`

위치 전달인자가 키워드 전달인자보다 항상 먼저 와야 한다고 했다

따라서 함수를 정의할 때도 모든 위치 매개변수(필수 매개변수)를 키워드 매개변수(선택 매개변수)보다 먼저 선언해야 한다

`def ok(x, y, z=1):` (O)

`def bad(x, y=1, z):` (X)

● 전달인자(argument)란?

- 함수(또는 메소드)를 호출할 때 함수(또는 메소드)의 이름 바로 다음에 오는 소괄호(**()**) 안에 할당해서 함수 안으로 전달하는 값
- 대부분의 파이썬 함수(또는 메소드)는 값을 전달인자로 받아서 작업을 수행

● 전달인자 구분

● 필수 전달인자

- 반드시 값을 전달해야 하며 그렇지 않으면 오류가 난다

● 선택 전달인자

- 선택 사항이라 값을 전달하지 않아도 되며 함수(메소드) 정의 문서에서 일반적으로 대괄호(**[]**)로 표시
- 값을 전달하지 않으면 기본값을 적용

range(**[**시작**,** **]**끝**[****,** **]**폭**)**

- **시작** : 순서 열의 시작 번호(선택 전달인자 : 값을 전달하지 않으면 기본값인 **0**이 주어진다)
- **끝** : 순서 열의 최대 값으로 반드시 값을 전달해야 한다(필수 전달인자 : 값을 전달하지 않으면 **TypeError** 가 난다)
- **폭** : 순서 열에서 서로 붙어 있는 번호 간의 간격(선택 전달인자 : 값을 전달하지 않으면 기본값인 **1**이 주어진다)

`range(끝)`

끝은 필수 전달인자

```
range()
```

`float([x])`

x는 선택 전달인자

```
d1 = float()  
print(d1)
```

```
d2 = float('12')  
print(d2)
```

전달인자는 형식에 따라 '위치 전달인자'와 '키워드 전달인자' 두 종류로 구분

위치 전달인자(positional argument)

- 함수의 괄호 안에 쉼표로 구분하는 한 개 또는 여러 개의 값을 말한다
- 예) `print_setup('A4', 1, False)`

키워드 전달인자(keyword argument)

- 앞에 키워드 식별자가 오는 전달인자를 말한다
- 예) `print_setup(paper='A4', copies=1, color=False)`



주의 (1)

필요한 전달인자보다 적거나 많은 전달인자를 사용하면 **TypeError**가 발생

`len(s)`

`len()`

```
a = [1, 2, 3]
b = [4, 5, 6]
len(a, b)
```




주의 (2)

- 위치 전달인자가 키워드 전달인자보다 항상 먼저 와야한다
- 그렇지 않으면 **SyntaxError**가 발생
- 따라서 함수를 정의할 때도 모든 위치 매개변수(필수 매개변수)가 어떤 키워드 매개변수(선택 매개변수)보다도 앞에 선언해야 한다

`def ok(x, y, z=1):` (O)

`def bad(x, y=1, z):` (X)

```
L = ['c', 'b', 'd', 'a']
```

```
sorted(L, reverse=True)
```

```
sorted(reverse=True, L)
```

예시 : 위치 전달인자

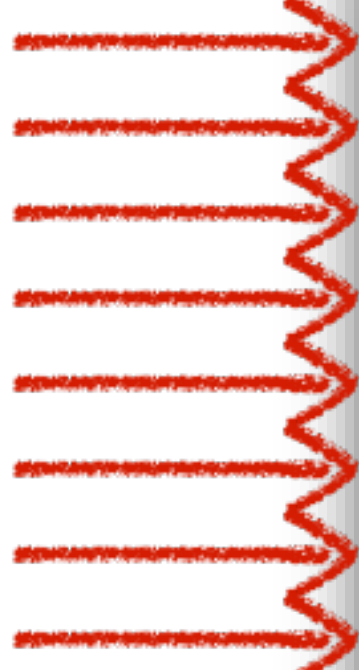
```
def string_format(text, length=10, fills=' '):
    if len(text) < length:
        text = text + fills * (length - len(text))
    return text

s = '파이썬과 빅데이터 분석'           # s의 길이는 12
string_format(s)                       # 필수 전달인자만 사용
string_format(s, 15)                   # length에 15
string_format(s, 5, '*')                # length에 5, fills에 '*' (s보다 length가 짧다)
string_format(s, 15, '!')               # length에 15, fills에 '!'
string_format(s, 25, '~')               # length에 25, fills에 '~'
string_format(s, '~')                   # 두 번째 위치 전달인자로 문자열이 왔다
string_format(s, length=25, '~')        # 키워드 전달인자가 위치 전달인자 앞에 있다
```

예시 : 키워드 전달인자

키워드 전달인자를 사용하면 전달인자의 순서는 중요하지 않다

```
def string_format(text, length=10, fills=' '):
    if len(text) < length:
        text = text + fills * (length - len(text))
    return text
s = '파이썬과 빅데이터 분석' # s의 길이는 12
```



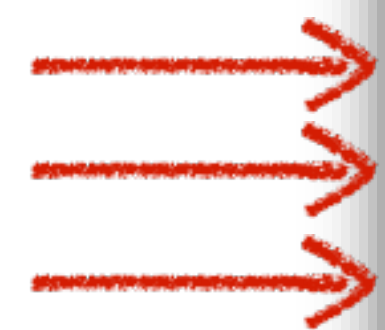
string_format(s, fills='\$', length=15)	# fills와 length의 순서를 바꿔 값을 전달
string_format(text=s)	# 키워드 전달인자를 필수 매개변수에도 사용 가능(나머지는 기본값)
string_format(fills='@', text=s)	# length는 기본값 10 사용
string_format(length=1, text=s)	# fills는 기본값 ' ' 사용
string_format(fills='%!', text=s, length=20)	# 모든 값에 키워드 전달인자를 사용
string_format(s, length=15, '\$')	# 키워드 전달인자가 위치 전달인자 앞에 있다
string_format(s, length=15, fills='\$')	
string_format(text=s, 15, '\$')	# 키워드 전달인자가 위치 전달인자 앞에 있다

* 연산자와 전달인자의 위치

함수 정의 부분에 * 를 선언하면 전달인자들의 위치를 지정하는 역할을 한다

* 뒤에는 위치 전달인자가 올 수 없고 키워드 전달인자만 올 수 있다

자주 사용하는 `sorted()` 함수를 살펴보자 `sorted(iterable, *, key=None, reverse=False)`



```
L = ['a', 'd', 'C', 'E', 'b']
sorted(L)
sorted(L, key=None, reverse=False) # 키워드 전달인자로 기본값을 전달
sorted(L, None, False)             # 위치 전달인자로 기본값을 전달
```

예시 : * 연산자와 전달인자의 위치


```
def product_in_units(i, j, k, *, units='square meters'):
    return f'{i * j * k} {units}'
```

→ product_in_units(2, 3, 4)
 → product_in_units(2, 3, 4, units='square inches')
 → product_in_units(2, 3, 4, 'square inches')

예시 : * 연산자와 전달인자의 위치

*가 매개변수 처음에 위치한다면 어떻게 될까?

```
def print_setup(*, paper='A4', copies=1, color=False):
    print(f"프린터 설정: paper='{paper}', copies={copies}, color={color}")
```



```
print_setup()
print_setup(paper='Letter', color=True)
print_setup(copies=5, paper='Letter', color=True)
print_setup('Letter', 5, True)
print_setup('Letter')
```

```
.. . - . . . . .
.. . - . . . . .
```


함수를 정의할 때 * 연산자를 매개변수 앞에 두면 **시퀀스형 패킹 연산자**다

- * 연산자를 사용하면 콤마로 구분하는 튜플 형태의 불특정 다수 위치 전달인자를 하나의 필수 매개변수로 패킹해서 받아 처리하는 함수를 만들 수 있다

- 따라서, 시퀀스형 패킹 연산자 * 는 몇 개의 위치 전달인자를 넘겨 받을지 모르는 함수를 정의할 때 유용하다 Very flexible!!!

 * 연산자를 변수 이름 바로 앞에 붙여 사용하면 시퀀스형 객체의 패킹 연산자라고 이미 6장(튜플)에서 다루었다

```
def product(*args):
    """임의의 숫자를 전달받은 후 모두 곱한 결과값을 반환하는 함수"""
    result = 1
    for arg in args:
        result *= arg
    return result
```

```
x = product(15)
print(x)
x = product(2, 3, 4)
print(x)
x = product(5, 6, 7, 8)
print(x)
```

```
def sum_of_powers(*args, power=1):
    """임의의 숫자를 전달받은 후 그 숫자를 지정한 수의 거듭제곱으로
    계산한 값들을 모두 합해서 반환하는 함수"""
    result = 0
    for arg in args:
        result += arg ** power
    return result
```

```
x = sum_of_powers(1, 3, 5)
print(x)
x = sum_of_powers(2, 3, power=2)
print(x)
x = sum_of_powers(1, 3, 5, 7, power=3)
print(x)
```

함수의 전달인자 앞에 *를 붙이면 **시퀀스형 언패킹 연산자**가 된다

- 이를 '위치 전달인자 언패킹(positional argument unpacking)'이라고 한다
- * 패킹은 리스트로 할당을 하지만, * 언패킹은 튜플 형태로 할당되었기 때문에 '가변길이 전달인자 튜플(variable-length argument tuples)'이라고도 한다

특징은 다음과 같다

- 전달인자가 두 개 이상이면 * 연산자의 위치는 어디든 상관없다
- 단, * 연산자가 앞에 붙어 있는 전달인자가 언패킹되었을 때 총 객체 개수는 반드시 해당 함수의 전달인자 개수와 일치해야 한다

```
def add5(i, j, k, l, m):
    """5개의 전달인자를 받아 모두 합한 결과값을 반환하는 함수"""
    return i + j + k + l + m

x = add5(1, 2, 3, 4, 5)      # 위치 전달인자가 다섯 개
print(x)
```

```
args = [1, 2, 3, 4, 5]
x = add5(*args)              # args의 다섯 개 모두 언패킹
print(x)

x = add5(1, *args[:3], 1)    # arg의 첫 세 개만 언패킹
print(x)

x = add5(*args[-2:], 1, 2, 3) # arg의 마지막 두 개만 언패킹
print(x)
```


매핑형 패킹 연산자 **

함수를 정의할 때 ** 연산자를 매개변수 앞에 두면 **매핑형 패킹 연산자**다

- ** 연산자를 사용하면 콤마로 구분하는 튜플 형태의 불특정 다수 키워드 전달인자를 하나의 선택 매개변수로 패킹해서 받아 처리하는 함수를 만들 수 있다
- 따라서, 매핑형 패킹 연산자 ** 는 몇 개의 키워드 전달인자를 넘겨 받을지 모르는 함수를 정의할 때 유용하다

Very flexible!!!

```
def add_fruit_detail(fruit_name, date_produced, **kwargs):
    """위치 전달인자 두 개와 0개 이상의 불특정 다수의 매개변수를 키워드 전달인자로 받아 처리하는 함수"""
    print('과일종류 =', fruit_name)
    print('생산일자 =', date_produced)
    for key in sorted(kwargs): # 키워드 전달인자의 키 값으로 오름차순 정렬해서 출력
        print(f'{key} = {kwargs[key]}')
```



```
add_fruit_detail('블루베리', '9일')
add_fruit_detail('바나나', '11일', 원산지='필리핀')
add_fruit_detail('오렌지', '11일', 유통기간='60일', 원산지='제주도', 수량=55)
```

매핑형 언패킹 연산자 **

함수의 전달인자 앞에 **를 붙이면 **매핑형 언패킹 연산자**가 된다

앞서 정의한 `print_setup` 함수를 다시 사용하자

```
def print_setup(*, paper='A4', copies=1, color=False):
    print(f"프린터 셋팅: paper='{paper}', copies={copies}, color={color}")
```

** 를 매핑형 언패킹 연산자로 사용하려면, 딕셔너리 형태의 객체를 참조하는 변수를 하나의 전달인자로 받아 언패킹해서 사용할 수

```
opt1 = dict(copies=35, color=True, paper='Letter')
print_setup(**opt1)
```

일부 전달인자의 값만 지정하고 나머지는 기본값을 사용하면 변경할 값만 딕셔너리로 만들어 전달할 수도 있다

```
opt2 = {'copies': 5}
print_setup(**opt2)
```

패킹 연산자로 함수 정의

* 와 ** 연산자를 매개변수로 함께 사용하여 임의의 위치 전달인자와 임의의 키워드 전달인자를 받아서 처리하는 함수도 만들 수 있다

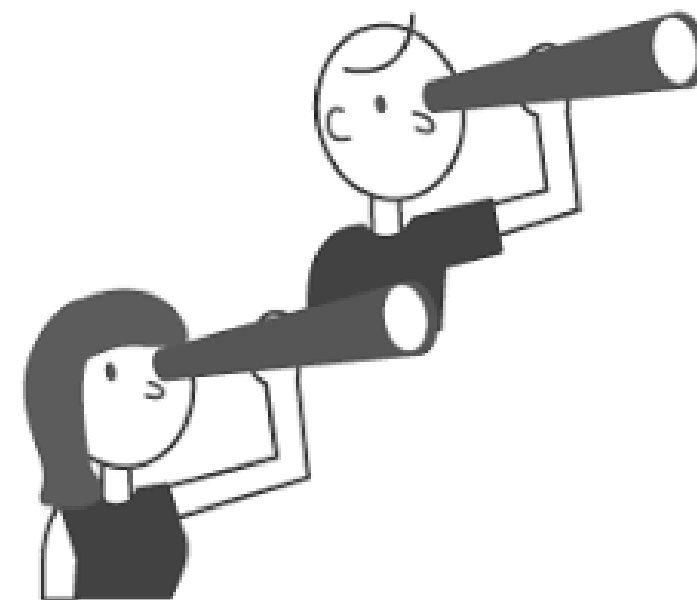
```
def print_args(*args, **kwargs):
    """먼저 위치 전달인자를 순서대로 출력하고,
    그 다음에 키워드 전달인자를 순서대로 출력하는 함수다."""
    for i, arg in enumerate(args):
        print(f'위치 전달인자 {i} = {arg}')
    for key in kwargs:
        print(f'키워드 전달인자 {key} = {kwargs[key]}')

print_args('라이언', -7, 2.5, age=19, email='ryan@korea.kr')
print_args('포도', 유통기한='30일', 원산지='김해', 수량=100)
```



범위와 가시성

Scope and Visibility



- 범위(scope)**란?
- 변수나 함수의 가시성을 나타낸다
 - 범위를 벗어난 변수나 함수는 다른 코드에서 사용할 수 없기 때문에 호출할 수도 없다
 - 모든 변수와 함수는 만들어진 위치에 따라 범위가 정해진다
 - 프로그램 최상위 레벨에서 변수를 정의하면 모든 함수에서 접근할 수 있는데, 이런 변수를 **전역 변수**라고 한다
 - 반면에 함수 안에 정의한 변수인 **지역 변수**는 그 함수 외 다른 함수에서 사용할 수 없도록 범위가 제한되어 있다

전역 변수(global variable)

- 다른 함수를 포함해서 **프로그램 전체**에서 사용 가능

```
x = '나는 전역 변수 x다.'      # 전역 변수를 선언
def myfn1():
    y = '나는 지역 변수 y다.'  # 지역 변수를 선언
    print(y)                  # 지역 변수를 출력 <- in scope
    print(x)                  # 전역 변수를 출력 <- in scope
```

지역 변수(local variable)

- **해당 함수**에서만 사용 가능하며 타 함수에서는 사용할 수 없다
- 지역 변수란 특정 함수 내에서 생성한 변수를 지칭한다

```
myfn1()
print(x)      # 전역 변수에 접근
print(y)      # 지역 변수에 접근
```

- 가시성**(visibility)란? 변수나 함수를 프로그램의 어느 부분에서 볼 수 있는지, 즉 변수나 함수에 합법적으로 접근할 수 있는 프로그램 영역
- 가시성은 어떤 변수나 함수를 호출해서 사용할 수 있는지 여부를 결정한다
 - 가시성이 없으면 그 변수나 함수를 사용할 수 없다

범위와 가시성은 주로 일치하지만,

- 같은 범위 안에 같은 이름의 변수나 함수가 함께 있다면 이 중 하나를 일시적으로 볼 수 없는 상황이 존재할 수도 있다
- 즉, 같은 범위에 있는 변수나 함수라도 가시성이 없을 수 있다
- 예를 들어, 같은 이름의 전역 변수와 지역 변수가 같은 범위(e.g., 함수 안)에 있을 때, 전역 변수의 가시성은 해당 범위에서 일시적으로 사라진다



```
x = '나는 전역 변수 x다.' # 전역 변수 선언
def myfn2():
    x = 'Hi Python' # 같은 이름의 지역 변수 선언
    print(x)

myfn2()
print(x)
```

왜 이런 결과를 가져올까? 이유가 뭘까?

전역 변수에 접근하기

```

x = '나는 전역 변수 x다.'
def myfn3():
    x = 'Hi Python'
    print(x)
    
```

```

myfn3()
print(x)
    
```

not changed

```

x = '나는 전역 변수 x다.'
def myfn3():
    global x
    x = 'Hi Python'
    print(x)
    
```

```

myfn3()
print(x)
    
```

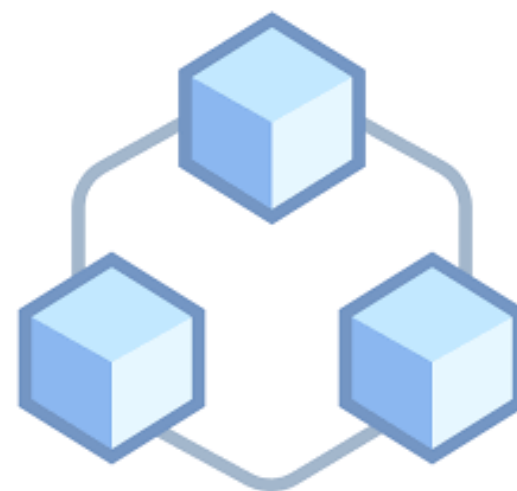
changed



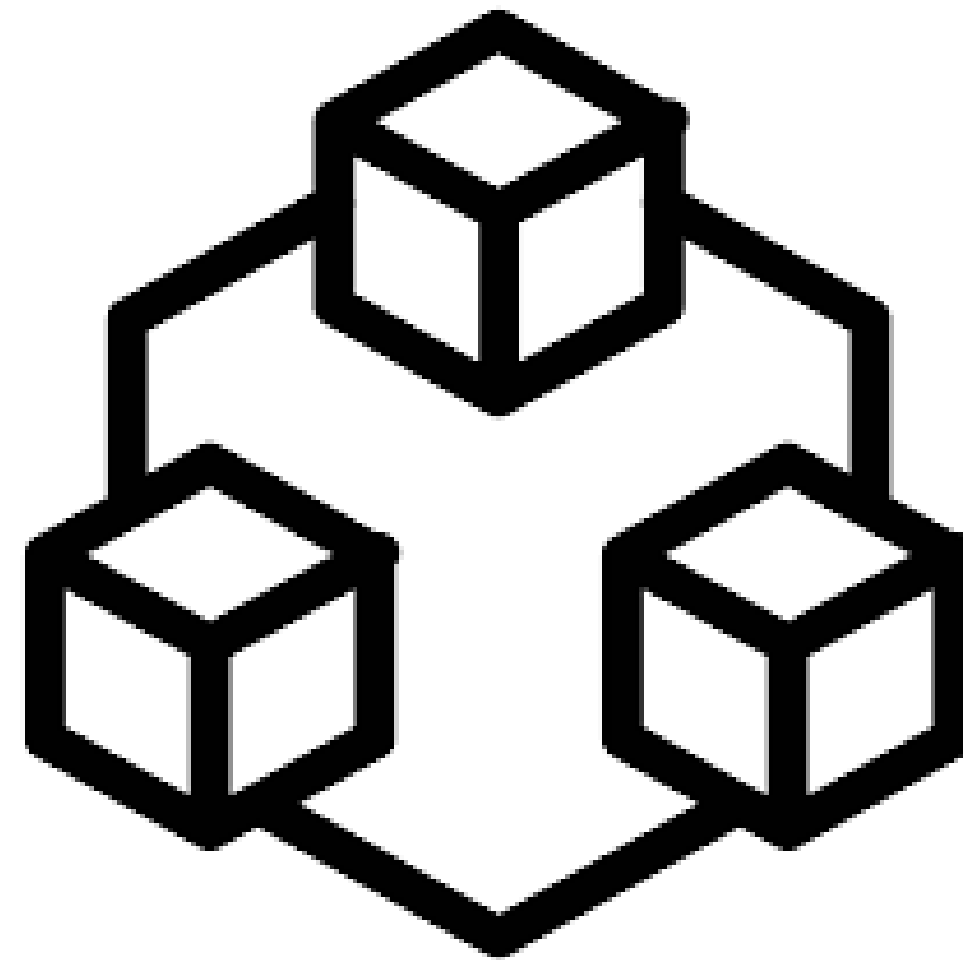
전역 변수의 값이 **가변형**(mutable)(예, 리스트, 딕셔너리 등)이면 **global**을 선언하지 않고
경 가능

• 모듈과 패키지

Modules and Packages



- 모듈화(modularization)란?
 - 하나의 프로그램을 여러 개의 작은 단위로 나누는 것을 뜻한다
 - **모듈**(modules) : 하나의 프로그램을 구성하는 단위
 - 하나의 프로그램을 여러 개의 작은 단위로 쪼개거나
 - 이를 분할정복(divide and conquer)이라고 한다
 - 자주 사용하는 코드를 뽑아 하나의 프로그램으로 저장한 후 다른 여러 프로그램에서 불러와 사용
 - 이 경우 모듈은 주로 여러 함수들의 집합
 - e.g., `import numpy` # `numpy` 모듈을 불러옴
 - ...



● 코드 중복 방지

- 불필요한 코드의 중복을 막고 코드를 단순화
- 코드를 한번만 작성하면 이후 여러 프로그램에서 필요할 때마다 마음껏 가져다 쓸 수 있다
- 코드의 수정과 업데이트가 매우 쉽고 빠르다

● 추상화

- 추상화가 가능
 - **추상화**(abstraction) : 불필요한 세부 내용은 무시하고 중요한 핵심 내용에만 초점을 맞추는 것
 - 예) ‘**밥을 하다**’는 추상화된 작업이라 할 수 있으며, ‘쌀통에서 쌀을 꺼내다’, ‘물을 붓고 쌀을 씻다’, ‘밥 물을 맞춘다’, ‘전기 밥솥에 넣고 취사 버튼을 누른다’ 등의 세부 과정을 굳이 알 필요가 없다
- 모든 컴퓨터 프로그램은 일정 수준의 추상화가 되어 있다

● 재사용

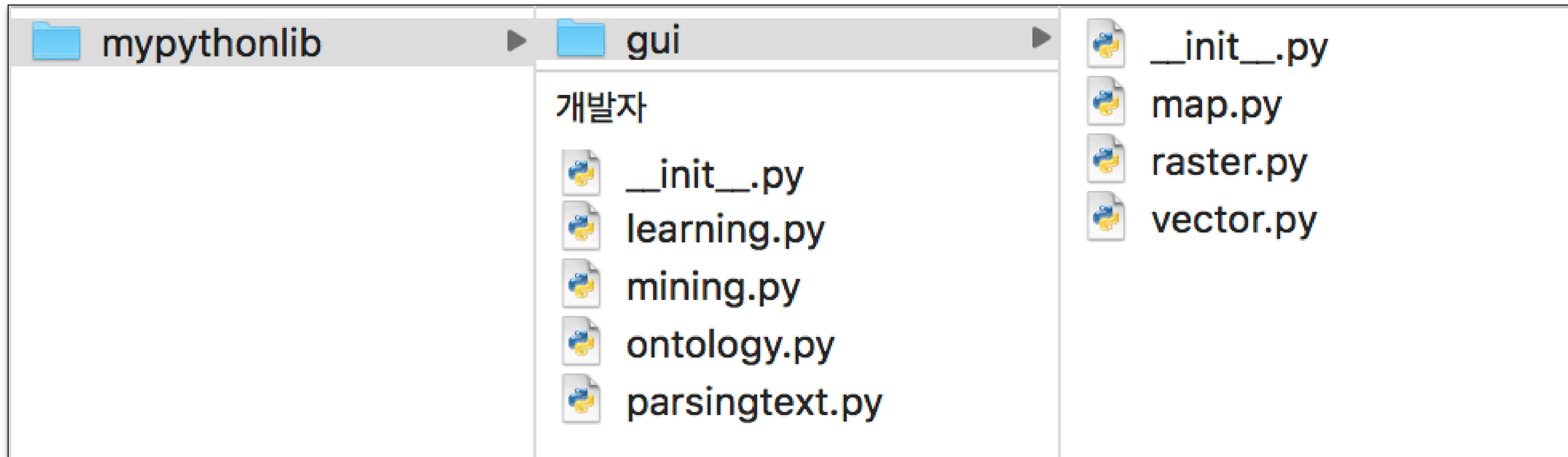
- 재사용(reuse)이 가능해 같은 프로그램 내에서 여러 번 사용할 수 있다
- 불러오기(**import**)를 통해 다른 프로그램에서도 사용 가능
- 하나의 프로그램을 개발할 때 각자 맡은 모듈만 작성하면 되므로 여러 명이 동시에 하나의 프로그램을 개발할 수 있다

- 모듈(module)이란?
 - 서로 연관된 함수나 클래스를 담고 있는 `.py` 라는 확장자를 가진 파이썬 파일
 - 그러나...
 - 프로그램 역시 `py` 라는 확장자를 파이썬 파일에 담겨져 있는 형태

- 모듈과 프로그램의 가장 중요한 차이점
 - **프로그램** : 어떤 문제를 해결하기 위해 파이썬 인터프리터 모드에서 특정 작업을 실행하는 파일
 - **모듈** : 프로그램이 특정 작업을 실행할 때 필요한 함수나 클래스 등을 모아 놓은 파일
 - 따라서 프로그램이 모듈을 사용하려면 먼저 `import`나 `from ... import` 문으로 불러와야 한다

- 모듈에 접근하기
 - 모듈 이름과 함수 이름 또는 속성(변수) 이름을 점(`.`)으로 구분해서 호출(실행)한다
 - 예) `math.pi`, `math.factorial(x)`
 - 이처럼 점 연산자(`.`)로 접근하는 방법을 '점 표기법(dot notation)'이라 한다

- 패키지(package)란?
 - 모듈들을 모아놓은 **디렉토리**(폴더)이며 다음과 같은 아이템을 담고 있다
 - 해당 디렉토리(폴더)에 **`__init__.py`** 파일(파일에 내용이 비어 있어도 됨)이 있다
 - `__init__.py`** 파일의 내용은 비어 있어도 상관없다
 - 위의 두 조건을 만족하면 디렉토리 이름이 패키지 이름이 된다
 - `directory(folder) name = package name`

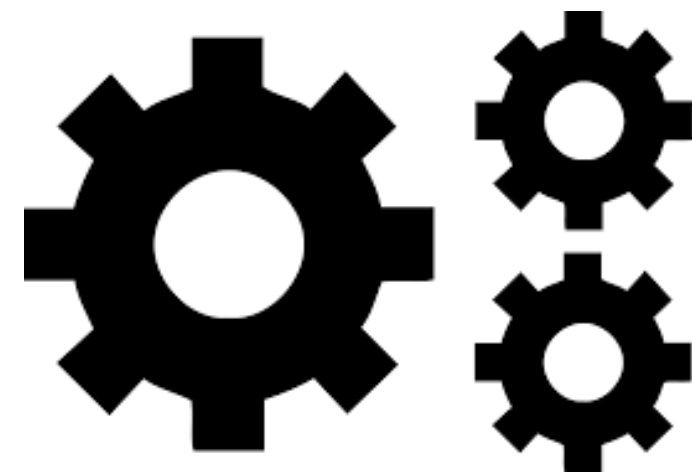


정리 : 모듈 vs. 패키지 vs. 표준 라이브러리

- 모듈(module)이란?
 - 서로 **연관된 함수**들을 담고 있는 파이썬 파일
 - 간단하게 생각하면 **.py** 파일이 모듈
 - 파이썬 프로그램 파일을 인터프리터 모드에서 실행하거나 파이썬 프로그램을 대화형 모드에서 실행할 때 불러오기(**import**)를 해서 사용할 수 있는 또 다른 파이썬 프로그램 파일(**.py**)
- 패키지(package)란?
 - 관련 모듈들을 체계적으로 모아놓은 **디렉토리**(폴더)
 - 패키지에 속해 있는 모듈에 접근하려면 도트 연산사(**.**)를 통해 접근해야한다
 - 예) **패키지이름.모듈이름**
- 표준 라이브러리(library)
 - 내장 함수(built-in functions)와 마찬가지로 파이썬은 기본적으로 유용한 모듈과 패키지를 표준 라이브러리를 통해 제공한다
 - 아나콘다(Anaconda)의 경우에는 데이터 분석에 유용한 대부분의 패키지를 기본적으로 제공하고 있다
 - 예) NumPy, Pandas, Scikit-learn, BeautifulSoup, ...

• 모듈과 패키지 불러오기

Importing Modules and Packages



```
import math                # 모듈 불러오기
```

```
x = math.exp(10)
print(x)
```

```
import math as m          # 식별자(가명)로 모듈 불러오기
```

```
x = m.exp(10)
print(x)
```

```
from math import exp      # 모듈로부터 함수 불러오기
```

```
x = exp(10)
print(x)
```

```
from math import *        # 모듈의 모든 클래스와 함수 불러오기
```

```
x = exp(10)
print(x)
```

```
22026.465794806718
22026.465794806718
22026.465794806718
22026.465794806718
```

```
import X [as 식별자]
```

- **X** 는 **모듈 이름** 또는 **패키지 안의 모듈 이름**이다
 - 예) 모듈 이름 : `import collections`
 - 예) 패키지 안의 모듈 이름(**패키지이름.모듈이름**) : `import os.path`
 - 패키지 이름만 사용할 수도 있지만 별다른 쓰임새가 없다
- **import** 문을 사용하면 항상 패키지 또는 모듈 이름과 함께 함수나 클래스 이름 전부를 사용하기 때문에 이름이 길어져 번거로울 수 있지만 잠재적인 명칭 오류를 피할 수 있다
 - **명칭 오류**(name conflict)란 같은 이름의 모듈이나 함수들 중복 사용해 발생하는 오류를 말한다
- **as 식별자**는 선택 사항
 - 불러오는 **X** 에 **식별자**(변수)를 지정해서 별칭으로 사용할 수 있다
 - 이때 **식별자**로 기존의 변수나 모듈 또는 패키지 이름을 사용하면 명칭 오류가 발생할 수 있으니 주의해야 한다
 - 이론상으로는 명칭 오류가 발생할 수 있지만 실제로는 **as** 문을 통해 명칭 오류를 피할 수 있다

```
import math
math.pi
math.factorial(5)
```

3.141592653589793

120

```
import math as m
m.pi
m.factorial(5)
```

3.141592653589793

120


```
import X1, X2, ..., Xn
```

- 여러 개의 모듈 이름 또는 패키지 안의 모듈 이름을 한 줄에 작성해서 한꺼번에 불러오기 할 때 사용하는 방법
- X** 는 **모듈 이름** 또는 **패키지 안의 모듈 이름**이다
 - 패키지 이름만 사용할 수도 있지만 별다른 쓰임새가 없다
- 여러 줄로 작성해야 할 때는 `\`(역슬래시)를 사용할 수 있지만 권장하지 않는다
- import** 문을 사용하면 항상 패키지 또는 모듈 이름과 함께 함수나 클래스 이름 전부를 사용하기 때문에 잠재적인 명칭 오류를 피할 수 있다

```
import math, random, datetime, os, os.path # 여러 개의 모듈을 한꺼번에 불러온다
math.sqrt(2)                               # 2의 제곱근을 구한다
random.randint(1, 99)                       # 1~99 사이 임의의 정수를 생성한다
datetime.datetime.now().month               # 현재 몇 월인지 알아본다
os.environ['LOGNAME']                        # 현재 로그인 사용자의 ID를 알아본다
os.path.exists('고향의봄.txt')              # 현재 폴더에 지정한 파일이 있는지 확인한다
```

```
1.4142135623730951
```

```
18
```

```
8
```

```
'jinsoopark'
```

```
True
```

```
from X import Y [as 식별자]
```

- **X** 는 **모듈 이름** 또는 **패키지 이름**이거나 **패키지 안의 모듈 이름**이다
- **Y** 는 **모듈 이름** 또는 **모듈 안 함수 이름**(괄호는 생략) 또는 **클래스 이름**(괄호는 생략)이다
- **from ... import** 문은 점 연산자(.)로 패키지와 모듈 이름을 불러오지 않고 직접 함수 이름 등을 불러 사용하기 때문에 명칭 오류가 생길 수 있다
 - 예를 들어, **numpy** 모듈의 **sum** 함수를 **from numpy import sum**으로 불러왔다면, **numpy.sum**이 아니라 **sum**으로 사용하기 때문에, 코드에서 파이썬 표준 라이브러리의 **sum** 함수를 호출한 것인지 **numpy** 모듈의 **sum** 함수를 호출한 것인지 구분하기 어려워진다
- **as 식별자**는 선택 사항
 - 불러오는 모듈이나 함수에 **식별자**(변수)를 지정해서 별칭으로 사용할 수 있다
 - **식별자**로 기존 모듈이나 함수 이름을 사용하면 명칭 오류가 나니 주의해야 한다

```
from math import pi
pi
```

```
3.141592653589793
```

```
from math import factorial
factorial(5)
```

```
120
```

```
from math import pi as mp
mp
```

```
3.141592653589793
```

```
from math import factorial as mf
mf(5)
```

```
120
```

1

from X import Y1, Y2, ..., Yn

2

from X import (Y1, Y2, Y3, Y4, Y5, ..., Yn)

3

from X import *

- **X** 는 **모듈 이름** 또는 **패키지 이름**이거나 **패키지 안의 모듈 이름**이다
- **Yn** 는 **모듈 이름** 또는 **모듈 안 함수 이름**(괄호는 생략) 또는 **클래스 이름**(괄호는 생략)이다
 - 1 여러 개의 모듈이나 함수 또는 클래스를 한 줄에 작성해서 한꺼번에 불러올 때 사용하는 방법
 - 2
 - 3 는 과 같지만 여러 줄에 걸쳐 다수의 모듈을 불러올 때 사용하는 방법
- 모듈 안의 모든 함수를 불러올 때 사용하는 방법
 - 주로 GUI 라이브러리처럼 패키지 안에 불러올 객체가 많을 때 사용할 수 있다
 - 하지만 명칭 오류가 발생하지 않도록 주의해야 한다.
 - 사용자가 만든 패키지나 모듈이 있고, 패키지 폴더에 저장한 `__init__.py` 파일에 모듈의 객체 이름 목록을 담은 `__all__` 전역 변수를 지정했다면 `__all__`에 있는 객체들만 불러온다
 - 단, private 접근 제어자 객체는 제외되기 때문에 다른 프로그램에서 불러 사용하기 원치 않는 변수나 함수 이름은 밑줄(`_`)로 시작하면 된다

1

from math import factorial, isnan, exp, log, sqrt

2

from math import (factorial, isnan, exp, log, sqrt, sin, tan, cos, degrees, radians)

3

from math import *

파이썬 표준 라이브러리

The Python Standard Library



파이썬 표준 라이브러리

- 지난 수년간 다양하고 완성도 높은 모듈들이 개발되었으며 파이썬 표준 라이브러리는 이러한 모듈들을 포함하고 있다 ...
- 그리고 파이썬 배포용 패키지의 용량이 커지다보니 몇몇 뒤편(그러나 포괄적(generic)이지 못한) 모듈들은 파이썬 표준 라이브러리에서 제외 되기도 했다 ...
- 하지만, 해당 모듈들은 개인 개발자 혹은 단체 등 제3자(써드파티, third party)에 의해서 개별적으로 유지보수가 되고 있으며 ...
- 이는 ... 써드파티가 만들어 배포하는 모듈들도 파이썬 표준 라이브러리만큼이나 유용하고 뛰어나다는 의미이다 ...
- 따라서 ... 가급적이면 ... 직접 코드를 작성하기 전에 ...
- 기존에 존재하는 모듈이 있는지 확인해보는 습관을 가지는 것이 좋다 ...
- Do NOT Reinvent the Wheel!!!
- 이어지는 내용에서 보편적으로 많이 사용되는 패키지와 모듈에 대한 간략한 정보를 다루고자 한다

파이썬이 제공하는 유용한 패키지/모듈

- 파이썬 3.x 표준 라이브러리
 - <https://docs.python.org/3/library/index.html>
 - 흔히 사용하는 파이썬 표준 라이브러리(패키지/모듈)
 - `datetime`, `time`, `math`, `collections`, `random`, `os` , `urllib`, `itertools`
- 아나콘다
 - <https://www.anaconda.com>
 - 아나콘다(Anaconda)에 포함되어 있는 라이브러리 중 데이터 분석에 매우 유용한 패키지/모듈
 - `matplotlib`, `nltk`, `networkx`, `numpy`, `pandas`, `scikit-learn`, `scipy`, `xlrd`

`random.random()`

균일 분포(uniform distribution)에 기반하여 0에서 1 사이(1은 포함하지 않는다) `[0.0, 1.0)` 실수 중 하나를 무작위로 선택해 반환

```
import random
random.random()

0.8949032467617718
```

0에서 1사이의 임의의 실수 5개를 만들려면 어떻게 하면 될까?

```
import random
rand_numbers = []
for _ in range(5):
    rand_numbers.append(random.random())
else:
    print(rand_numbers)
```

```
[random.random() for _ in range(5)]
```

random 모듈은 다양한 통계 분포에 기반한 무작위 추출법으로 **의사난수**(pseudo-random number)를 생성하는 함수들을 제공

- 의사난수란 알고리즘으로 생성하는 난수이기 때문에 진정한 난수는 아니라 **유사난수**라고도 한다
- 의사난수로 생성하는 값은 규칙적(deterministic)이긴 하지만 아주 긴 주기를 갖고 생성한 숫자 배열이라 일반적으로 통계나 모델링에서
- 재현 가능한 결과를 얻으려면 **random.seed** 함수를 사용하면 된다

random.seed()

seed 의 전달인자로 같은 정수를 사용하면 매번 같은 결과를 가져온다

```
print('1:', random.random())
print('2:', random.random())
random.seed(15)
print('3:', random.random())
random.seed(15)
print('4:', random.random())
print('5:', random.random())
random.seed(15)
print('6:', random.random())
```

1: 0.18854877350939192

2: 0.07113772341293145

random.randint(시작, 끝)

- 시작과 끝 사이(끝 포함) (시작, 끝) 난수 값 정수를 반환
- randrange(시작, 끝 + 1)와 같다

random.randrange(끝)

random.randrange(시작, 끝[, 폭])

- range 클래스로부터 받은 정수 값 중 무작위로 하나를 선택해서 반환
- choice(range())와 같지만 실제로 range 객체를 생성하지는 않는다

random.randint(1, 99)

random.randrange(1, 100)

```
L = list(range(30))
random.shuffle(L)
print(L)
```

random.shuffle(x[, random])

- 리스트 x의 객체들을 무작위로 섞는다
- 선택 매개변수 random의 기본값은 random 함수

random.choice(seq)

- 시퀀스형(*seq*, sequence data type)으로부터 받은 객체 중 무작위로 하나를 선택해서 반환
- 만약 *seq*가 빈(empty) 자료형이면 **IndexError**가 발생

→ random.choice(range(1, 100))

→ t = 1, 2, 3, 4, 5, 6, 7, 8, 9
random.choice(t)

→ import string
random.choice(string.ascii_letters)

random.sample(population, k)

- 모집단 *population*(리스트, 튜플, 세트 등)에서 *k* 개를 중복 없이(즉, 교체하지 않고) 무작위로 선택해서 리스트로 반환

→ lotto = random.sample(range(1, 46), 6)
print(lotto)

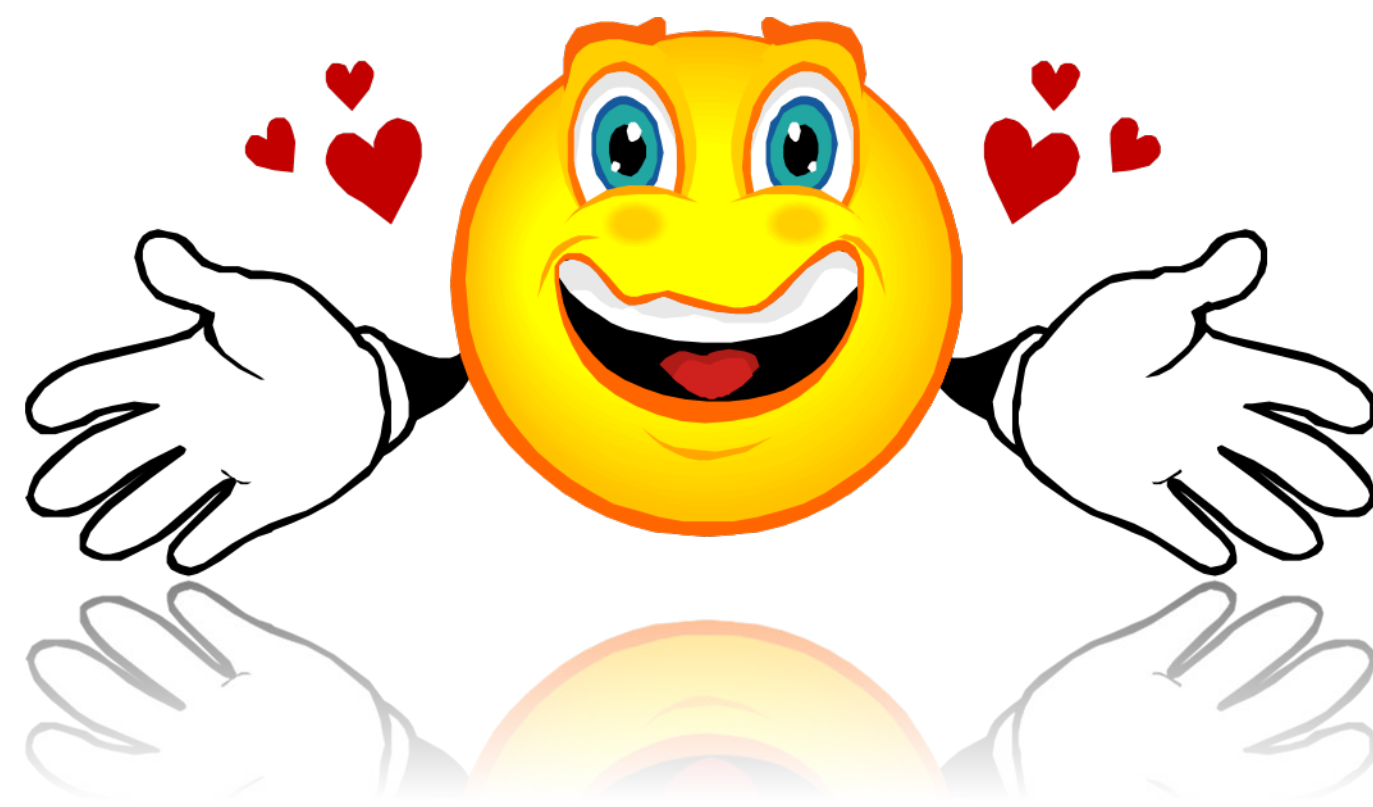
→ random.sample(range(1, 10), 6)

choice vs. sample

- 모집단에서 중복 허용하면서 샘플을 무작위로 선택하려면 **choice** 함수를 샘플 크기만큼 호출
- 모집단에서 중복 없이 샘플을 무작위로 선택하려면 **sample** 함수를 사용

```
[random.choice(range(5)) for _ in range(4)]
```

```
random.sample(range(5), 4)
```



끝

수고 하셨습니다