Python_04

함수 응용

내장함수(Built-in Functions)

- 파이썬 인터프리터에는 항상 사용할 수 있는 많은 함수와 형(type)이 내장되어있음

map

```
map(function, iterable)
```

- 순회 가능한 데이터구조(iterable)의 모든 요소에 함수(function)적용하고, 그 결과를 map object로 반환

```
def my_magic_func(n):
    return n * 10


my_list = [1, 2, 3, 4]

map_object = map(my_magic_func, my_list)
print(list(map_object))
#[10, 20, 30, 40]
```

- 알고리즘 문제 풀이시 input 값들을 숫자로 바로 활용하고 싶을 때

```
n, m = map(int, input().split())
print(n, m)   # 3 5
print(type(n), type(m)) # <class 'int'> <class 'int'>
```

filter

```
filter(function, iterable)
```

- 순회 가능한 데이터구조(iterable)의 모든 요소에 함수(function)적용하고, 그 결과가 True인 것들을 filter object로 반환

```
def odd(n):
    return n % 2


numbers = [1, 2, 3]
result = filter(odd, numbers)
print(result, type(result)) # <filter object at 0x00000205981F13D0> <class 'filter'>
print(list(result)) # [1, 3]
```

zip

```
zip(*iterables)
```

- 복수의 iterable을 모아 튜플을 원소로 하는 zip object를 반환

```
girls = ['jane', 'ashley']
boys = ['justin', 'eric']
pair = zip(girls, boys)
print(pair, type(pair)) # <zip object at 0x000001DFAD5B6140> <class 'zip'>
print(list(pair)) # [('jane', 'justin'), ('ashley', 'eric')]
```

lambda (== 익명함수)

```
lambda[parameter] : 표현식
```

- 람다 함수
  - 표현식을 계산한 결괏값을 반환하는 함수로, 이름이 없는 함수여서 익명함수라고도 불림
- 특징
  - return문을 가질 수 없음
  - 간편 조건문 외 조건문이나 반복문을 가질 수 없음
- 장점
  - 함수를 정의해서 사용하는 것보다 간결하게 사용 가능
  - def를 사용할 수 없는 곳에서도 사용가능

```
def triangle_area(b, h):
    return 0.5 * b * h

print(triangle_area(5, 6)) # 15.0

triangle_area = lambda b, h : 0.5 * b * h
print(triangle_area(5, 6)) # 15.0
```

```
(lambda x : x * x)(4)

def pow(x):
    return x * x

pow(4)
```

```
print(list(map(lambda n : n * 10, [1,2,3])))
```

```
arr = [('길동', 80), ('철수', 20), ('남길', 20)]

# method 1
arr = sorted(arr, key=lambda x: x[1])
print(arr)

# method 2
def getter(x):
    return x[1]

arr = sorted(arr, key=getter)
print(arr)

# method 3
from operator import itemgetter

arr = sorted(arr, key=itemgetter(1))
print(arr)

# [('남길', 20), ('철수', 20), ('길동', 80)]
```

```
arr = [('길동', 80), ('철수', 20), ('가희', 80), ('남길', 20)]

arr = sorted(arr, key=lambda x: (x[1], x[0]))
print(arr)

#[('남길', 20), ('철수', 20), ('가희', 80), ('길동', 80)]
```

재귀 함수(recursive function)

- 자기 자신을 호출하는 함수
- 무한한 호출을 목표로 하는 것이 아니며, 알고리즘 설계 및 구현에서 유용하게 활용
  - 알고리즘 중 재귀함수로 로직을 표현하기 쉬운 경우가 있음(예 - 점화식)
  - 변수의 사용이 줄어들며, 코드의 가독성이 높아짐
- 1개 이상의 base case(종료되는 상황)가 존재하고, 수렴하도록 작성

```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n-1)


print(factorial(4))  # 24
```

재귀 함수 주의 사항

- 재귀 함수는 base case에 도달할 때까지 함수를 호출함
- 메모리 스택이 넘치게 되면(stack overflow) 프로그램이 동작하지 않게 됨
- 파이썬에서는 최대 재귀 깊이(maximum recursion depth)가 1000번으로, 호출 횟수가 이를 넘어가게 되면 Recursion Error 발생

반복문과 재귀 함수 비교

- 알고리즘 자체가 재귀적인 표현이 자연스러운 경우 재귀함수를 사용함
- 재귀 호출은 변수 사용을 줄여줄 수 있음
- 재귀 호출은 입력 값이 커질 수록 연산 속도가 오래 걸림

```
def factorial(n):
    result = 1
    while n > 1:
        result *= n
        n -= 1
    return result

print(factorial(4))  # 24
```

함수 가변 입력(패킹/언패킹)

패킹/언패킹 연산자(Packing/ Unpacking Operator)*

- 모든 시퀀스형(리스트, 튜플 등)은 패킹/언패킹 연산자*를 사용하여 객체의 패킹 또는 언패킹이 가능

```
x, *y = i, j, k ...
```

- 패킹

- 대입문의 좌변 변수에 위치
- 우변의 객체 수가 좌변의 변수 수보다 많은 경우 객체를 순서대로 대입
- 나머지 항목들은 모두 별 기호로 표시된 변수에 리스트로 대입

```
x, *y = 1, 2, 3, 4
print(x) # 1
print(type(x)) # <class 'int'>
print(y) # [2, 3, 4]
print(type(y)) # <class 'list'>
```

- 언패킹
  - argument 이름이 *로 시작하는 경우, argument unpacking이라 함
    - *패킹의 경우, 리스트로 대입
    - *언패킹의 경우 튜플 형태로 대입

```
def multiply(x, y, z):
    return x * y * z


numbers = [1, 2, 3]
print(multiply(*numbers)) # 6
```

```
def test(x,y,z):
    return [x,y,z]

numbers = {'a': 1, 'b' : 2, 'c' : 3}

print(test(*numbers)) # ['a', 'b', 'c']
print(test(*numbers.values())) # [1, 2, 3]
print(test(*numbers.items())) # [('a', 1), ('b', 2), ('c', 3)]
```

- 별표(*) 연산자가 곱셈을 의미하는지 Packing/Unpacking 연산자인지 구분
  - Packing/Unpacking 연산자 *
    - *가 대입식의 좌측에 위치하는 경우
    - *가 단항 연산자로 사용되는 경우
      - 단항 연산자 : 하나의 항을 대상으로 연산이 이루어지는 연산자
  - 산술연산자로서의 *
    - *가 이항연산자로 사용되는 경우
      - 이항 연산자: 두 개의 항을 대상으로 연산이 이루어지는 연산자

정해지지 않은 여러 개의 Arguments 처리

Q . print 함수의 Arguments 개수가 변해도 잘 동작하는 이유는?

A. 애스터리스크(Asterisk)혹은 언패킹 연산자라고 불리는 *덕분

```
print(*values: object, sep: '', end: '\n', file: sys.stdout, flush: False)
```

가변 인자(*args)

- 가변인자란?
  - 여러 개의 Positional Argument를 하나의 필수 parameter로 받아서 사용
- 가변인자는 언제 사용하는가?

- 몇 개의 Positional Argument를 받을지 모르는 함수를 정의할 때 유용

```
def add(*args):
    for arg in args:
        print(arg)
add(2)
add(2,3,4,5)
```

- 가변 인자를 데이터를 묶어서 변수에 할당하는 것
- 패킹
  - 여러개의 데이터를 묶어서 변수에 할당하는 것

```
numbers = (1, 2, 3, 4, 5)  # 패킹
print(numbers)  # (1,2,3,4,5)
```

- 언패킹
  - 시퀀스 속의 요소들을 여러 개의 변수에 나누어 할당하는 것

```
numbers = (1, 2, 3, 4, 5) # 패킹
a, b, c, d, e = numbers  # 언패킹
print(a, b, c, d, e) # 1 2 3 4 5
```

- 언패킹시 변수의 개수와 할당하고자 하는 요소의 갯수가 동일해야함

```
numbers = (1, 2, 3, 4, 5) # 패킹
a, b, c, d, e, f = numbers  # 언패킹
```

- 언패킹시 왼쪽의 면수에 asterisk(*)를 붙이면, 할당하고 남은 요소를 리스트에 담을 수 있음

```
numbers = (1, 2, 3, 4, 5) # 패킹
a, b, *rest = numbers  # 1, 2를 제외한 나머지 rest에 대입
print(a,b,rest) # 1 2 [3, 4 ,5]

a, *rest, e = numbers  # 1, 5를 제외한 나머지 rest에 대입
print(rest) # [2, 3, 4]
```

```
my_list = list(range(1, 123))
for i in range(0,len(my_list), 10):
    print(*my_list[i:i+10], sep=',')
```

```
def sum_all(*numbers):
    result = 0
    for number in numbers:
        result += number
    return result

print(sum_all(1,2,3)) # 6
print(sum_all(1,2,3,4,5)) # 21
```

```
def print_family_name(father, mother, *pets):
    print(f'아버지 : {father}')
    print(f'어머니 : {mother}')
    print(f'반려동물들...')
    for name in pets:
        print(f'반려동물 : {name}')


print_family_name('아빠', '엄마', '하루', '메이')
```

```
'''
아버지 : 아빠
어머니 : 엄마
반려동물들...
반려동물 : 하루
반려동물 : 메이
'''
```

가변 키워드 인자(**kwargs)

- 몇 개의 키워드 인자를 받을지 모르는 함수를 정의할 때 유용
- **kwargs는 딕셔너리로 묶여 처리되며, parameter에 **를 붙여 표현

```
def family(**kwargs):
    for key, value in kwargs.items():
        print(key, ':', value)

family(father='아빠', mother='엄마', baby='아기')

'''
father : 아빠
mother : 엄마
baby : 아기
'''
```

```
def print_family_name(father, mother, **pets):
    print(f'아버지 : {father}')
    print(f'어머니 : {mother}')
    if pets:
        print('반려동물들...')
        for species, name in pets.items():
            print(f'{species} : {name}')


print_family_name('아빠', '엄마', cat1 = '하루', cat2 = '메이')

'''
아버지 : 아빠
어머니 : 엄마
반려동물들...
cat1 : 하루
cat2 : 메이
'''
```

가변 인자(*args)*와 가변 키워드 인자*(**kwargs) 동시 사용

```
def print_family_name(*parents, **pets):
    print(f'아버지 : {parents[0]}')
    print(f'어머니 : {parents[1]}')
    if pets:
        print('반려동물들...')
        for species, name in pets.items():
            print('{} : {}'.format(species, name))


print_family_name('아빠', '엄마', cat1 = '하루', cat2 = '메이')
```

모듈과 패키지

```
import module
from module import var, function, Class
from module import *

from package import module
from package.module import var, function, Class
```

모듈 == (다양한 기능을 하나의 파일로)

- 특정 기능을 하는 코드를 파이썬 파일(.py) 단위로 작성한 것

패키지 == (다양한 파일을 하나의 폴더로)

- 특정 기능과 관련된 여러 모듈의 집합
- 패키지 안에는 또 다른 서브 패키지를 포함

라이브러리 == (다양한 패키지를 하나의 묶음으로)

- 파이썬에 기본적으로 설치된 모듈과 내장함수
- ex) random.py

pip == (이걸을 관리하는 관리자)

파이썬 패키지 관리자

- PyPI(Python Package Index)에 저장된 외부 패키지들을 설치하도록 도와주는 패키지 관리 시스템
- 패키지 설치
  - 최신 버전/ 특정 버전 / 최소 버전을 명시하여 설치 할 수 있음
  - 이미 설치되어 있는 경우 이미 설치되어 있음을 알리고 아무것도 하지 않음

```
$pip install SomePackage
$pip install SomePackage==1.0.5
$pip install SomePackage>=1.0.4
```

- 패키지 삭제

```
$pip uninstall SomePackage
```

- 패키지 목록 및 특정 패키지 정보

```
$pip list
$pip show SomePackage
```

- 패키지 관리하기
  - 아래의 명령어들을 통해 패키지 목록을 관리하고 설치할 수 있음
  - 일반적으로 패키지를 기록하는 파일의 이름은 requirements.txt로 정의

```
$pip freeze > requirements.txt
$pip install -r requirements.txt
```

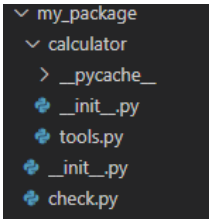
패키지 활용

- 패키지는 여러 모듈/하위 패키지로 구조화
  - 활용 예시 : package.module
- 모든 폴더에는 `__init__.py` 를 만들어 패키지로 인식
  - Python 3.3부터는 파일이 없어도 되지만, 하위 버전 호환 및 프레임워크 등에서의 동작을 위해 파일을 생성하는 것을 권장

패키지 만들기

- 계산 기능이 들어간 calculator패키지를 아래와 같이 구성

- check.py에서 calculator의 tools.py의 기능을 사용
- 폴더 구조




```
# my_package / calculator / tools.py
def add(num1, num2):
    return num1 + num2

def minus(num1, num2):
    return num1 - num2
```

```
# my_package / check.py
from calculator import tools

print(dir(tools))
'''
['__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__',
'add', 'minus']
'''
print(tools.add(3,5))  # 8
print(tools.minus(3,5)) # -2
```

가상환경== (패키지의 활용 공간)

- 파이썬 표준 라이브러리가 아닌 외부 패키지와 모듈을 사용하는 경우 모두 pip를 통해 설치해야 함
- 복수의 프로젝트를 하는 경우 버전이 상이할 수 있음
- 이러한 경우 가상환경을 만들어 프로젝트별로 독립적인 패키지를 관리할 수 있음
- 가상 환경을 만들고 관리하는데 사용되는 모듈(Python 버전 3.5부터)
- 특정 디렉토리에 가상환경을 만들고, 고유유한 패키지 집합을 가질 수 있음
  - 특정 폴더에 가상 환경(패키지 집합 폴더 등) 있고
  - 실행환경에(ex) bash)에서 가상환경을 활성화 시켜
  - 해당 폴더에 있는 패키지를 관리/사용함
- 가상환경을 생성하면, 해당 디렉토리에 별도의 파이썬 패키지가 설치됨

```
$python -m venv <폴더명>
```

가상환경 활성화/비활성화

- 아래의 명령어를 통해 가상환경을 활성화
  - <venv>는 가상환경을 포함하는 디렉토리의 경로

```
(for mac) $ source <venv> /bin/activate
(for window) C:\> <venv>\Scripts\activate.bat
```

- 가상환경 비활성화는 `$ deactivate` 명령어 사용