ch.5 모듈, 패키지, 프로그램

모듈과 import문,

모듈 : 파이썬 코드의 파

```
#main.py
import module1

module1.hi()
```

```
#module1.py
def hi():
    print("hi")
```

같은 디렉터리에 위치시, import main.py 하면 된다.

▼ import된 코드의 사용이 내부에만 제한되는 경우, import 문을 내부에 놓는다

```
#main.py
def using_module1():
    import module1
    module1.hi()

using_module1()
module1.hi()



#output
NameError: name 'module1' is not defined
```

▼ 필요한 모듈만 임포트 하기

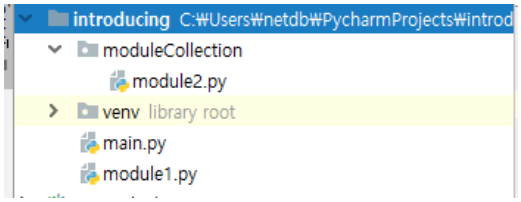
```


def using_module1():
    from module1 import bye
    bye()
    hi()

using_module1()

#output
NameError: name 'hi' is not defined
bye
```

▼ 하위 디렉터리에 위치하고 있는 module import



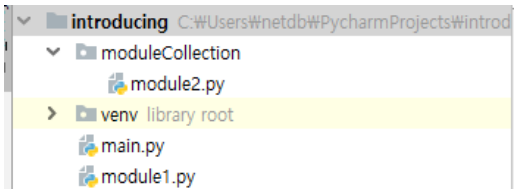

```
def using_module1():
    from module1 import bye
    bye()
    from moduleCollection.module2 import parasite
    parasite()

using_module1()
```

▼ 상위 디렉터리에 위치하고 있는 module import

현재 모듈의 절대경로를 알아내어 상위 폴더 절대경로를 참조 path에 추가하는 방식.

참고: https://brownbears.tistory.com/296




```
import os, sys

def parasite():
    print("parasite")

print(sys.path.append(os.path.dirname(os.path.abspath(os.path.dirname(__file__))))) #None
print(os.path.dirname(os.path.abspath(os.path.dirname(__file__)))) #C:\Users\netdb\PycharmProjects\introducing
print(os.path.abspath(os.path.dirname(__file__))) # C:\Users\netdb\PycharmProjects\introducing\moduleCollection
print(os.path.dirname(__file__)) #C:/Users/netdb/PycharmProjects/introducing/moduleCollection

sys.path.append(os.path.dirname(os.path.abspath(os.path.dirname(__file__))))
import module1
module1.hi()


#output
None
C:\Users\netdb\PycharmProjects\introducing
C:\Users\netdb\PycharmProjects\introducing\moduleCollection
C:/Users/netdb/PycharmProjects/introducing/moduleCollection
hi
```

파이썬 표준 라이브러리

제네릭을 사용할 수 있는 일부 모듈

누락된 키 처리하기: setdefault(), defaultdict()

- 존재하지 않는 키로 딕셔너리에 접근하려 하면 예외가 발생→ 기본값을 반환하는 딕셔너리의 get()함수를 사용하면 이 예외를 피할 수 있다.

- setdefault() : get()과 같지만, 키가 누락된 경우 딕셔너리에 항목을 할당할 수 있다.
- defaultdict() : 딕셔너리를 생성할 때 모든 새 키에 대한 기본값을 먼저 지정. 인자는 함수

▼

```
s =['a','b','c','d','e']
a = {i:e for i, e in enumerate(s)} # {0: 'a', 1: 'b', 2: 'c', 3: 'd', 4: 'e'}

#get()함수 이용
print(a.get('f','없다'))
```

```
s =['a','b','c','d','e']
a = {i:e for i, e in enumerate(s)} # {0: 'a', 1: 'b', 2: 'c', 3: 'd', 4: 'e'}

a.setdefault('f',5) #반환값은 5
print(a) # {0: 'a', 1: 'b', 2: 'c', 3: 'd', 4: 'e', 'f': 5}
#존재하는 키에 다른 기본 값을 할당하려 하면 키에 원래 값이 반환되고 아무것도 바뀌지x
```

defaultdict()의 인자는 값을 누락된 키에 할당하여 반환하는 함수이다.

defaultdict 잘 모르겠으니까 자세하게

setdefault보다 빠르댜

1. from collections import defaultdict # 콜렉션에서 불러온다
2. d = defaultdict(object) #Default dictionary를 생성한다
3. d = detailtdict(lambda:0) # default값을 0으로 설정한다

```
In [6]: d = defaultdict(lambda: 0)

In [7]: d['one']
Out[7]: 0

In [8]: d['two']
Out[8]: 0

In [9]: d['two'] = 2

In [10]: d['one']
Out[10]: 0

In [11]: d['two']
Out[11]: 2

In [12]: d['three']
Out[12]: 0
```

항목 세기 : Counter()

from collections import Counter

1. 리스트를→ 딕트{element:frequency}로 맹글어준다 → 정확히는 counter객체가 딕트를 싸고있쥬

```
from collections import Counter
```

```
basket = ['a','a','a','b','c','c']
basket_counter = Counter(basket)
print(basket_counter) # Counter({'a': 3, 'c': 2, 'b': 1})
basket_dict = dict(basket_counter)
print(basket_dict) # {'a': 3, 'b': 1, 'c': 2}
```

2. counter.most_common() : 모든 요소를 내림차순으로 반환한다. 혹은 숫자를 입력하는 경우, 그 숫자만큼 상위 요소를 반환한다.

```
from collections import Counter

basket = ['a','a','a','b','c','c']
basket_counter = Counter(basket)
print(basket_counter) # Counter({'a': 3, 'c': 2, 'b': 1})
print(basket_counter.most_common()) #리스트로 반환 ( 튜플로 싸여진)
print(basket_counter.most_common(1))

#output
Counter({'a': 3, 'c': 2, 'b': 1})
[('a', 3), ('c', 2), ('b', 1)]
<class 'list'>
[('a', 3)]
```

3. 카운터끼리 결합 : + 연산자/ -연산자 물론 &(교집합), |(합집합)도 사용 가능

키 정렬하기 : OrderedDict()

from collections import OrderdDIct

딕셔너리의 키 순서는 예측할 수 없지만 얘를 아용하면 키의 추가 순서를 기억하고 이터레이터로 부터 순서대로 키값을 반환한다.

```
from collections import OrderedDict

basket = OrderedDict([ #<class 'collections.OrderedDict'>
    ('a', 1),
    ('b', 2),
    ('c', 3),
])

for e in basket:
    print(e)

#output
for e in basket:
    print(e)
```

스텍 + 큐 = 데크

출입구가 양 끝에 있는 큐

시퀀스의 양 끝으로부터 항목을 추가하거나 삭제할 때 유용하게 쓰인다.

```
```

코드 구조 순회하기: itertools

특수 목적의 이터레이터 함수를 포함하고 있다.

1. Chain() : 순회 가능한 인자들을 하나씩 반환한다

```
import itertools

for item in itertools.chain([1,2],['a','b']):
    print(item)

#output
1
2
a
b
```

2. cycle() : 인자를 순환하는 무한 이터레이터

```
import itertools

for item in itertools.cycle([1,2]):
    print(item)
#output
1
2
1
2
무한반복
```

3. accumulate() 축적된 값을 계산한다.

```
import itertools

for item in itertools.accumulate([1,2,3,4]):
    print(item)

#output
1
3
6
10
```

docstring

- 함수 몸체 시작 부분에 문자열을 포함시켜 함수 정의에 문서를 붙일 수 있다.
- 길게 작성할 수 있으며 서식을 추가할 수도 있다
- docstring을 출력하려면 help()함수를 호출한다. 함수 인자의 리스트와 서식화된 docstring을 읽기 위해 함수 이름을 인자로 전달한다

나중에 멋있는 사람이 되면 함 해보자

```
def fn1(a,b,*args):
    '''
    안녕 이건 fn1 함수이고 args로 받은걸 출력하는 함수야
        1.서식이
        그냥
        이런걸
        말하는 걸까..?
```

```
    '''
    print(args)

help(fn1) #서식이 없는걸 출력하고 싶다면 ech.__doc__ 을 하자

#output
Help on function fn1 in module __main__:

fn1(a, b, *args)
    안녕 이건 fn1 함수이고 args로 받은걸 출력하는 함수야
        1.서식이
        그냥
        이런걸
        말하는 걸까..?
```

decorator

- 소스코드를 바꾸지 않고, 사용하고 있는 함수를 수정하고 싶을 때 사용
- 하나의 함수를 취해서 또 다른 함수를 반환하는 함수
- *args와 &kwargs
- 내부 함수
- 함수 인자

```
def main_function():
    print("안녕 나는 메인 함수얌")

def sub_function1():
    print("안녕 나는 서브 함수 1이얌")

def sub_function2():
    print("안녕 나는 서브 함수 2이얌")
```

이런 main_funnction부터 sub_function1 ~ n개까지 무수히 많을 때, 각 함수에 똑같은 기능을 추가하고 싶다. → 그러면 일일이 모든 함수에 똑같은 코드를 쳐야 하는가?

→놉 ! 데코레이터 라는 걸 추가 해보자!

```
import datetime

def datetime_decorator(func):
    def decorated():
        print(datetime.datetime.now())
        func()
        print(datetime.datetime.now())
    return decorated

@datetime_decorator
def main_function():
    print("안녕 나는 메인 함수얌")
@datetime_decorator
def sub_function1():
    print("안녕 나는 서브 함수 1이얌")
@datetime_decorator
def sub_function2():
    print("안녕 나는 서브 함수 2이얌")

main_function()
```

```
#output
2019-05-31 19:24:42.829966
안녕 나는 메인 함수얌
2019-05-31 19:24:42.829966
```

1. 먼저 decorator 역할을 하는 함수를 정의하고, 이 함수에서 decorator가 적용될 함수를 인자로 받는다. python은 함수의 인자로 다른 함수를 받을 수 있다는 특징을 이용하는 것이다.
2. decorator역할을 하는 함수 내부에 또 한번 함수를 선언(nested function)하여 여기에 추가적인 작업(시간 출력)을 선언해 주는 것이다
3. nested 함수를 리턴해주면 된다

이렇게 해주면 된다!

주의: 대상 함수의 수행 중간에 끼어드는 구문은 할 수 없다. ➷ 원래 작업의 앞 뒤에 추가적인 작업을 손쉽게 사용 가능하도록 도와주는 역할이니까!

또한 한 함수에 여러개의 데코레이터를 할당할 수 있따

▼ class 형태로 decorator를 맹글어 보자

```
import datetime

class Datetime_decorator:
    def __init__(self,f):
        self.func = f

    def __call__(self,*args,**kwargs):
        print(datetime.datetime.now())
        self.func(*args, **kwargs)
        print(datetime.datetime.now())
#         return decorated

class MainClass:
    @Datetime_decorator
    def main_function():
        print("안녕 나는 메인 함수얌")

a =MainClass()
a.main_function()
```

네임스페이스와 스코프

네임스페이스: 특정 이름이 유일하고, 다른 네임스페이스에서의 같은 이름과 관계가 없는 것을 말한다.

```
a = '안녕'

def print_global():
    a = '빠이'
    print('inside print'+a)
    print(id(a))

print(id(a))
print('outside print '+a)
print_global()
```

print_global() 함수에 있는 a는 '안녕'의 전역변수 a와 다른 변수다.

만약 a='빠이'를 print문 다음으로 넣는다면 오류가 뜬다 왜 ? → 전역변수는 static인데 바꾸려고 했으니까!

▼ 함수 내에서 전역변수 a에 접근하려면?

```
a = '안녕'

def print_global():
    global a
    a = '빠이'
    print('inside print'+a)
    print(id(a))

print(a)
print_global()
print(a)

#output
안녕
inside print빠이
2722929030840
빠이
```

근데 global a = '빠이' 이렇게 붙여쓰면 안된다 왜지..?

아 그리고 생각해보니까 저 a가 바뀌기 전 후 id값이 다른건 당연한거임 → str은 immutable이니까

ch.5 모듈,패키지,프로그램

```
In [6]: d = defaultdict(lambda: 0)

In [7]: d['one']
Out[7]: 0

In [8]: d['two']
Out[8]: 0

In [9]: d['two'] = 2

In [10]: d['one']
Out[10]: 0

In [11]: d['two']
Out[11]: 2

In [12]: d['three']
Out[12]: 0
```