

# Python 프로그래밍

## 함수 고급

기술경영전문대학원  
김영민

# Comprehension

- Comprehension (함축, 내포)
  - 하나 이상의 반복자(iterator)를 사용하여 파이썬 자료구조를 만드는 방법
- List comprehension

`[f(xs) for xs in iter]`

반복문 내의 변수      순회 가능한 객체에 대한 반복문  
xs에 대한 함수

`[f(xs) for xs in iter if pred(xs) ]`

이러한 조건을 만족하는 요소들만 저장

- 1부터 5까지의 정수 리스트 만들기

```
number_list = []  
number_list.append(1)  
number_list.append(2)  
number_list.append(3)  
number_list.append(4)  
number_list.append(5)
```

```
number_list = []  
for number in range(1, 6):  
    number_list.append(number)
```

```
number_list = list(range(1, 6))
```

- 1부터 5까지의 정수 리스트 만들기 – List comprehension

```
number_list = [number for number in range(1,6)]
```

```
[1, 2, 3, 4, 5]
```

- 함수 적용

```
number_list = [number-1 for number in range(1,6)]
```

```
[0, 1, 2, 3, 4]
```

- 조건 적용

```
a_list = [number for number in range(1,6) if number % 2 == 1]
```

```
[1, 3, 5]
```

```
a_list = [number for number in range(1,6) if number % 2 == 1]
```

- for 문을 사용한다면

```
a_list = []  
for number in range(1,6):  
    if number%2 == 1:  
        a_list.append(number)
```

```
sentence = ['I', 'Love', 'Python', 'Soooooo', 'MUCH!!!']
```

```
# 함수 적용
```

```
[word.lower() for word in sentence]
```

```
# 조건 적용
```

```
[word for word in sentence if len(word) > 6]
```

```
# 함수 적용하여 튜플로 저장
```

```
[(x, x ** 2, x ** 3) for x in range(10)]
```

```
rows = range(1,4)
cols = range(1,3)
for row in rows:
    for col in cols:
        print(row, col)
```

결과

```
1 1
1 2
2 1
2 2
3 1
3 2
```

- 위를 리스트 내포로 작성해보자.

```
rows = range(1,4)
cols = range(1,3)
cells = [(row, col) for row in rows for col in cols]
for cell in cells:
    print(cell)
```

```
(1, 1)
(1, 2)
(2, 1)
(2, 2)
(3, 1)
(3, 2)
```

```
# 다음은 어떤 결과가?
[(i,j) for i in range(5) for j in range(i)]
```



- Dictionary comprehension

{key\_func(vars):val\_func(vars) for vars in iterable}

key:value 형태로 저장

- 예제

```
word = 'letters'  
letter_counts = {letter: word.count(letter) for letter in word}
```

{ 'l': 1, 'e': 2, 't': 2, 'r': 1, 's': 1 }

*그러나 이 코드에는 비효율적인 부분이 있다.*

```
word = 'letters'  
letter_counts = {letter: word.count(letter) for letter in word}
```

```
{'l': 1, 'e': 2, 't': 2, 'r': 1, 's': 1}
```

- 다른 방식으로 코딩

```
word = 'letters'  
letter_counts = {letter: word.count(letter) for letter in set(word)}
```

```
{'t': 2, 'l': 1, 'r': 1, 's': 1, 'e': 2}
```

- Set comprehension

{func(vars) **for** vars **in** iterable}

- 예제

```
a_set = {number for number in range(1,6) if number % 3 == 1}
```

{1, 4}

```
{v:k for k,v in d.items()}
```

```
{word for word in hamlet if is_palindrome(word.lower())}
```

# Functions and code structure

- return 값을 갖는 함수

```
def function_name(param1, param2):  
    value = do_something()  
    return value
```

- def 라는 키워드로 함수임을 나타냄
- return 은 해도 되고 안해도 됨. 만약 return이 없다면 아무것도 돌려주지 않는 것임
- 매개변수에 변수 타입이 지정되지 않음
- return 하는 값에 대한 타입도 지정하지 않음

- 덧셈 함수

```
>>>  
>>> def sum(a, b):  
...     return a+b  
...  
>>> sum(1, 2)  
3  
>>> sum(1.3, 3.1)  
4.4  
>>> sum('love ', 'python')  
'love python'  
>>>
```

- zip을 이용한 순회
  - zip이라는 함수를 이용하여 여러 시퀀스를 병렬로 순회할 수 있음

```
days = ['Monday', 'Tuesday', 'Wednesday']
fruits = ['banana', 'orange', 'peach']
drinks = ['coffee', 'tea', 'beer']
desserts = ['tiramisu', 'ice cream', 'pie', 'pudding']

for day, fruit, drink, dessert in zip(days, fruits, drinks, desserts):
    print(day, ": drink", drink, "- eat", fruit, "- enjoy", dessert)
```



- zip을 이용하여 영어-프랑스 사전 만들기

```
english = 'Monday', 'Tuesday', 'Wednesday'  
french = 'Lundi', 'Mardi', 'Mercredi'
```

```
list( zip(english, french) )  
# [('Monday', 'Lundi'), ('Tuesday', 'Mardi'), ('Wednesday', 'Mercredi')]
```

```
dict( zip(english, french) )  
# {'Monday': 'Lundi', 'Tuesday': 'Mardi', 'Wednesday': 'Mercredi'}
```

- 파이썬은 함수의 인자(argument)를 유연하게 처리함
- 인자: 함수를 불러서 쓸 때 실제로 집어넣는 값
- 우선 순서대로 상응하는 매개변수(parameter)에 복사하는 위치인자(positional arguments)

```
# wine, entree, dessert를 받아서 딕셔너리로 만들어 반환하는 함수
def menu(wine, entree, dessert):
    return {'wine': wine, 'entree': entree, 'dessert': dessert}
```

함수 호출

```
>>> menu('chardonnay', 'chicken', 'cake')
{'wine': 'chardonnay', 'entree': 'chicken', 'dessert': 'cake'}
```

*각 위치의 의미를 잘 알아야 한다는 단점 존재*

- 매개변수(parameter)에 상응하는 이름을 인자에 지정하여 준다면, 함수에서 **정의된 순서와 다르더라도** 알아서 잘 **매핑**해준다.

```
# wine, entree, dessert를 받아서 딕셔너리로 만들어 반환하는 함수
def menu(wine, entree, dessert):
    return {'wine': wine, 'entree': entree, 'dessert': dessert}
```

함수 호출

```
>>> menu(entree='beef', dessert='bagel', wine='bordeaux')
{'dessert': 'bagel', 'wine': 'bordeaux', 'entree': 'beef'}
```

심지어 섞어서 쓸 수도 있음 ← 위치 인자가 먼저 나와야함

```
>>> menu('frontenac', dessert='flan', entree='fish')
{'entree': 'fish', 'dessert': 'flan', 'wine': 'frontenac'}
```

- 함수 호출자가 인자(값)를 제공하지 않으면 기본값을 사용하게 됨

```
# dessert의 기본값을 정해줌
def menu(wine, entree, dessert='pudding'):
    return {'wine': wine, 'entree': entree, 'dessert': dessert}
```

함수 호출

```
>>> menu('chardonnay', 'chicken')
{'dessert': 'pudding', 'wine': 'chardonnay', 'entree': 'chicken'}
```

dessert 값을 입력한 경우

```
>>> menu('dunkelfelder', 'duck', 'doughnut')
{'dessert': 'doughnut', 'wine': 'dunkelfelder', 'entree': 'duck'}
```

```
def menu(price, wine='chardonnay', entree='chicken', dessert='pudding'):  
    return {'price': price, 'wine': wine, 'entree': entree, 'dessert': dessert}
```

- 다음과 같은 호출이 가능함

menu(100)	# 1 위치 인자
menu(price=100)	# 1 키워드 인자
menu(price=120, entree='beef')	# 2 키워드 인자
menu(dessert='bagel', price=110)	# 2 키워드 인자
menu('eighty' 'saint-pierre', 'fish')	# 3 위치 인자
menu('hundred' wine='saint-pierre')	# 1 위치 인자, 1 키워드 인자

```
def menu(price, wine='chardonnay', entree='chicken', dessert='pudding'):  
    return {'price': price, 'wine': wine, 'entree': entree, 'dessert': dessert}
```

- 다음과 같은 호출은 불가능함

menu()

menu(price=100, 'saint-pierre')

menu(100, price=120)

menu(main='cream pasta')

# price에 인자가 할당이 안됨

# 키워드 인자 할당 후, 위치 인자

# 같은 변수에 두 번 할당

# 정의되지 않은 매개변수

- 함수 시작 부분에 문자열을 포함하여 함수 정의에 문서를 붙일 수 있음

```
def echo(anything):  
    'echo returns its input argument'  
    return anything
```

```
def print_if_true(thing, check):  
    """  
Prints the first argument if a second argument is true.  
The operation is:  
1. Check whether the *second* argument is true.  
2. If it is, print the *first* argument.  
"""  
    if check:  
        print(thing)
```

- `help()` 함수를 호출하여 내용을 출력할 수 있음

```
In[12]: help(echo)
Help on function echo in module __main__:

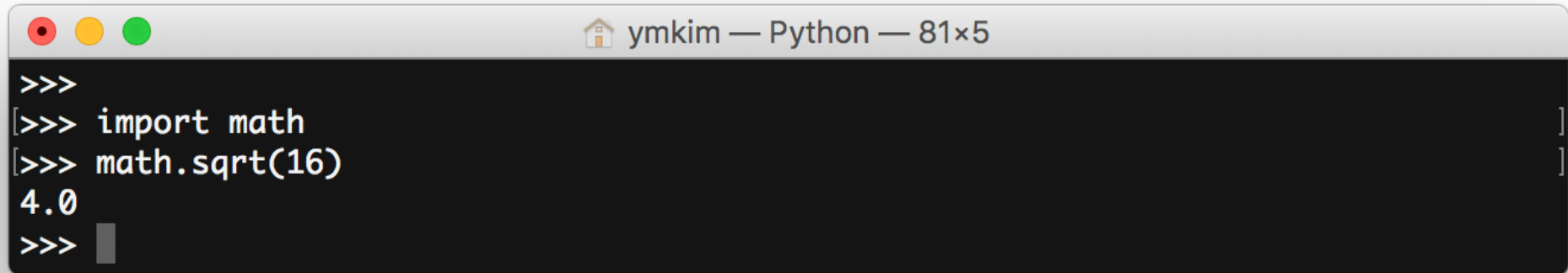
echo(anything)
    echo returns its input argument

In[13]: print(echo.__doc__)
echo returns its input argument
```



# Packages, Module, main

- 유용한 프로그램들을 미리 만들어 놓아 불러서 쓸 수 있게 한 것
- 패키지라고도 함
- 기본으로 설치되지 않는 것들은 필요할 때 설치할 수 있음

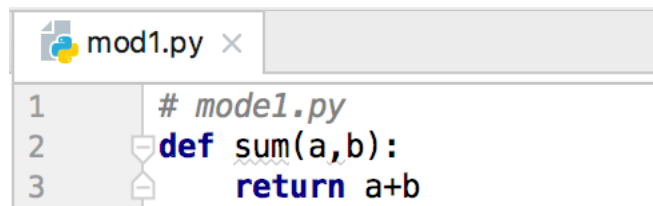


```
ymkim — Python — 81x5
>>>
>>> import math
>>> math.sqrt(16)
4.0
>>>
```

- scikit-learn
  - \$ pip install scikit-learn
  - Machine learning library
  - 기계학습을 위한 다양한 알고리즘, 데이터셋 등이 포함됨
  - 관련된 다른 패키지(Numpy, SciPy)등이 자동으로 설치됨

- NumPy
- Pandas
- seaborn
- matplotlib
- SciPy
- scikit-learn

- 함수나 변수, 클래스들을 모아 놓은 파일
- 다른 파이썬 프로그램에서 불러와 사용할 수 있게끔 만들어진 파이썬 파일
- 모듈만들기
  - mod1.py 라는 파일을 만들어 저장



```

1  # mod1.py
2  def sum(a,b):
3      return a+b
    
```

- 파일이 저장된 디렉토리에서 python 대화형 인터프리터 실행

```

labihem-iMac:test ymkim$ python3
Python 3.6.2 (default, Jul 17 2017, 16:44:45)
[GCC 4.2.1 Compatible Apple LLVM 8.1.0 (clang-802.0.42)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import mod1
>>> print(mod1.sum(2,3))
5
    
```

*import* 모듈이름 으로 로딩

혹은

*from* 모듈이름 *import* 모듈함수 로 로딩

```
test — Python — 81x11
[labihem-iMac:~ ymkim$ cd PycharmProjects/test/
[labihem-iMac:test ymkim$ python3
Python 3.6.2 (default, Jul 17 2017, 16:44:45)
[GCC 4.2.1 Compatible Apple LLVM 8.1.0 (clang-802.0.42)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from mod1 import sum
>>> sum(2,3)
5
>>>
```

만약 모듈에 정의된 모든 함수를 사용하고 싶다면,

*>>> from* 모듈이름 *import* \*

- 패키지(Packages): 도트(.)를 사용하여 파이썬 모듈을 계층적으로 관리할 수 있도록 모듈들을 묶어놓은 것
- 모듈명이 A.B 라면 A는 패키지명, B는 A 패키지의 모듈
- game 패키지 예

```
game/  
  __init__.py  
  sound/  
    __init__.py  
    echo.py  
    wav.py  
  graphic/  
    __init__.py  
    screen.py  
    render.py  
  play/  
    __init__.py  
    run.py  
    test.py
```

- game, sound, graphic, play는 디렉토리 명
- .py 파일은 파이썬 모듈
- game 디렉토리가 이 패키지의 루트 디렉토리
- sound, graphic, play는 서브 디렉토리

- 이 파일을 실행 한다면?

```
mod1.py x
1  # mod1.py
2  def sum(a, b):
3      return a+b
4
5  def safe_sum(a, b):
6      if type(a) != type(b):
7          print("더할수 있는 것이 아닙니다.")
8          return
9      else:
10         result = sum(a, b)
11         return result
12
13 print(safe_sum('a', 1))
14 print(safe_sum(1, 4))
15 print(sum(10, 10.4))
```

```
test — -bash — 47x6
labihem-iMac:test ymkim$ python3 mod1.py
더할수 있는 것이 아닙니다.
None
5
20.4
labihem-iMac:test ymkim$
```

- 근데 이 파일을 import 하면?
  - 원하지 않은 것들이 프린팅됨

```
test — Python — 60x6
[>>> import mod1
더할수 있는 것이 아닙니다.
None
5
20.4
>>>
```



- 앞의 경우를 방지하려면 main 문을 사용하면 됨

```
if __name__ == "__main__":  
    print(safe_sum('a', 1))  
    print(safe_sum(1, 4))  
    print(sum(10, 10.4))
```

- 이 경우, 직접 파일을 실행했을 때는 `__name__ == "__main__"` 이 참이 되어 그 내부가, 다른 파일에서 이 모듈을 부를 때는 거짓이 되어 수행되지 않음