

# 36. 모듈, 패키지

---

2018.12

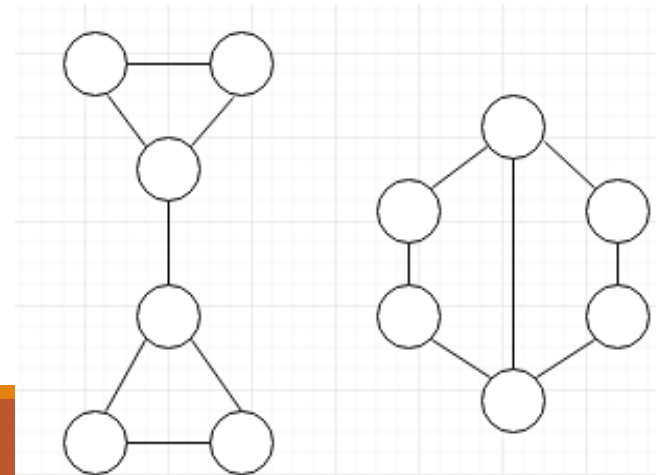
일병 김재형

# 모듈

---

## 모듈이란?

- 프로그램 내에는 여러 구성요소가 존재
- 각 구성요소는 상호작용 한다.
- 여러 요소가 다른 요소와 동일하게 상호작용 하는 것 보다는 관련성이 높은 것을 몇 개로 묶어서 보는 것이 이해하기 쉽다

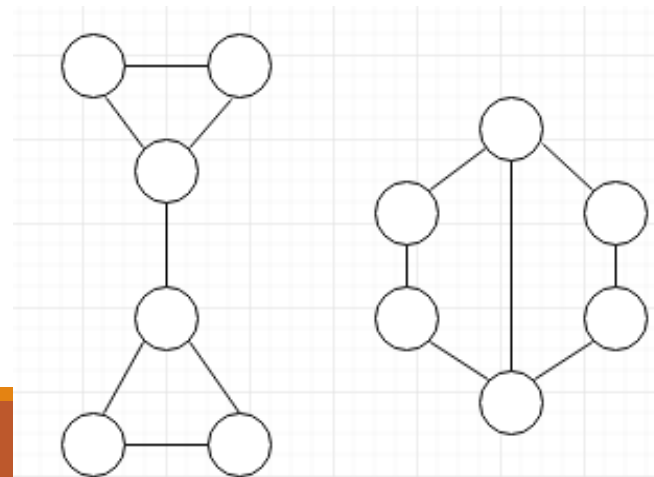


# 모듈

---

## 모듈이란?

- 관련성이 높은 함수나 변수, 클래스의 묶음을 명시하기 위해 모듈(Module)이라는 개념이 생겨났다.
- 비슷한 클래스와 함수를 중복해서 사용하지 않고, 코드를 다시 사용할 수 있다.
- 간단한 기능을 담을 때 사용



# 모듈

---

## 모듈의 이름-pep8

- 짧고, 전부 소문자를 사용
- 모듈의 이름을 읽기 쉽게 언더바(underscore)는 모듈의 이름을 읽기 쉽게 하기 위해서 사용한다.

# 모듈

---

## 모듈의 생성

- Python에서는 모듈과 프로그램을 시작하는 스크립트 파일의 차이가 없다.
- 항상 하던 것 처럼 모듈이름.py로 사용하면 된다.

```
calculator.py - PythonSemin... X
1  def add(a, b):
2      return a + b
3
4
5  def sub(a, b):
6      return a - b
7
8
9  def mul(a, b):
10     return a * b
11
12
13 def div(a, b):
14     return a / b
15
```

# 모듈

---

## 메인 모듈과 하위 모듈

- C언어에서는 메인 함수(프로그램의 시작점)가 존재
  - 처음부터 여러 소스파일을 사용
  - 시작함수(main함수)를 따로 정했다.
- Python에서는 메인 함수가 존재하지 않는다.
  - 처음 개발될 당시 리눅스/유닉스에서 사용하는 스크립트 언어 기반
  - 이 때의 스크립트 파일은 하나의 프로그램
  - 시작점이 따로 필요하지 않았다.

# 모듈

---

## 메인 모듈과 하위 모듈

- 맨 처음 실행되는 최상위 수준(Top Level)만 존재
  - 실행에 따라서 메인 모듈이 결정
- 메인모듈의 확인
  - 내장전역변수의 `__name__`이 `__main__`으로 지정된다.
  - 이를 통해 메인 프로그램으로 사용될 때와 모듈로 사용될 때를 구분할 수 있다.
  - 이렇게 구분할 경우, 파일을 `import`하여 디버깅을 좀 더 쉽게 할 수 있다.

# 모듈

---

## 메인 모듈과 하위 모듈

—모듈에 실행문을 놓았을 때,

직접 실행

```
17 print("계산기 모듈입니다.")
18 print(add(10, 10))
19 print(mul(10, 10))
```

```
계산기 모듈입니다.
20
100
```

모듈 실행

```
1 import calculator_exec
2
3 print("main모듈입니다.")
4 print(calculator_exec.add(2, 10))
5 print(calculator_exec.div(2, 5))
```

```
계산기 모듈입니다.
20
100
main모듈입니다.
12
0.4
```



# 모듈

---

## 메인 모듈과 하위 모듈

—if `__name__ == "__main__"`:을 사용하였을 때,

직접 실행

```
17 def main():
18     print("계산기 모듈입니다.")
19     print(add(10, 10))
20     print(mul(10, 10))
21
22
23 if __name__ == "__main__":
24     main()
```

```
계산기 모듈입니다.
20
100
```

모듈 실행

```
1 import calculator
2
3 print("main모듈입니다.")
4 print(calculator.add(2, 10))
5 print(calculator.div(2, 5))
```

```
main모듈입니다.
12
0.4
```

# 내장라이브러리 사용하기

## -review

---

import

- import 모듈

(모듈은 .py로 만드나, 확장자는 없이 사용한다.)

- 모듈이름.변수/함수이름으로 사용

ex) random 모듈 import

```
>>> import random
>>> random.randint(1, 100)
77
```

# 내장라이브러리 사용하기

## -review

---

import

- from 모듈 import 변수나 함수
  - 모듈 이름 없이 변수나 함수를 사용할 때 사용
- Ex) random모듈에서 randint함수 가져오기

```
>>> from random import randint
>>> randint(1, 100)
29
```

# 내장라이브러리 사용하기

## -review

---

import

- from 모듈 import 변수나 함수
  - 여러 줄을 쓰거나 ';'로 한번에 쓸 수도 있다.
- Ex) random모듈에서 randint함수 가져오기

```
>>> from random import randint
```

```
>>> from random import random
```

```
>>> randint(1, 100)
```

```
82
```

```
>>> random()
```

```
0.041474812251078186
```

```
>>> from random import randint, random
```

```
>>> randint(1, 100)
```

```
70
```

```
>>> random()
```

```
0.00447503626478063
```

# 내장라이브러리 사용하기

## -review

---

### import

- from 모듈 import \*
- 모듈내에 있는 변수와 함수를 전부 불러온다.
- 사용하지 않는 것을 '강력히' 추천한다.
- 많은 모듈을 사용시 어떤 코드가 어떤 모듈에 들어 있는지 파악하기 어렵고, 내부에서 같은 이름을 쓸 경우 오류가 발생할 수 있다.

# 내장라이브러리 사용하기

## -review

---

import

- import 모듈 as 별칭
- 모듈을 자주 사용하면, 전체 이름을 쓰는 것이 귀찮기 때문에 별칭을 사용하여 import한다.
- 관례적인 별칭이 존재(matplotlib.pylab -> plt, tensorflow -> tf)
- ex) tensorflow를 tf로 import

```
>>> import tensorflow as tf
>>> tf.__version__
'1.8.0'
```

# 모듈

---

## import

- from 모듈 import 변수(함수, 클래스) as 별칭
- 가져온 변수나 함수, 클래스를 가져온 뒤, 이름을 지정한다.
- ,를 통해 여러 개를 가져오며 이름을 지정할 수 있다.

# 모듈

---

## import

- 모듈 해제하기
  - del(모듈)
- 모듈 다시 가져오기
  - import importlib
  - Importlib.reload(모듈)



# 모듈

---

## import

- 함수 안에 모듈을 import할 수 있다.
- 내부에만 모듈 사용이 제한될 경우 사용할 수 있다.
- 코드 의존성을 명시하기 위해 모듈 맨 앞에 두는 경우가 많다.

```
>>> def rand(a, b):  
...     import random  
...     print(random.randrange(a, b))  
...  
>>> rand(1, 4)  
3  
>>> random.randrange(1, 4)  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
NameError: name 'random' is not defined
```

# 모듈

---

## 모듈의 위치

- import문을 찾기
  - 1. 파이썬 인터프리터의 내장 모듈
    - Cpython에서는 C언어로 프로그래밍되어 파이썬에 내장
  - sys.builtin\_module\_names를 출력하면 내장 모듈 목록을 볼 수 있다.

```
>>> import sys
>>> print(sys.builtin_module_names)
('_ast', '_codecs', '_collections', '_functools', '_
tring', '_symtable', '_thread', '_tracemalloc', '_wa
'gc', 'itertools', 'marshal', 'posix', 'pwd', 'sys',
```

# 모듈

---

## 모듈의 위치

- import문을 찾기
  - 2. sys.path에 정의된 디렉터리
    - 파이썬 모듈이 실행되고 있는 현재 디렉터리
    - PYTHONPATH 환경변수에 정의되어 있는 디렉터리
    - 파이썬과 함께 설치된 기본 라이브러리

# 파일 입/출력-열기-review

---

## 파일 입/출력-열기(open)

- file

- 파일명만 입력할 경우 실행한 위치의 디렉터리에서 파일을 찾는다.

- Path

- 프로그램(python, 운영체제 등)에서 파일이나 모듈(라이브러리)를 찾는 위치
- Python에서는 sys.path를 통해 path를 확인할 수 있다.

# 파일 입/출력-열기-review

---

## 파일 입/출력-열기(open)

### —Path

- Python에서는 sys.path를 통해 path를 확인할 수 있다.
- .py를 실행할 경우 맨 앞에 실행한 파일의 현재 위치가 나온다.
- path에 있는 순서대로 파일이나 모듈을 검색하여 사용한다.
- 인터랙티브 모드(python3만 실행 시)에는 현재 경로를 받지 않는다.

```
1  import sys
2
3  print(sys.path)
```

```
root@goorm:/workspace/PythonSeminar18/TeachingMaterials/insert_file/23_lecture_
aster)# python3 path.py
['/workspace/PythonSeminar18/TeachingMaterials/insert_file/23_lecture_file', '/c
cal/lib/python36.zip', '/usr/local/lib/python3.6', '/usr/local/lib/python3.6/li
oad', '/usr/local/lib/python3.6/site-packages']
```

# 모듈

---

## 모듈의 위치

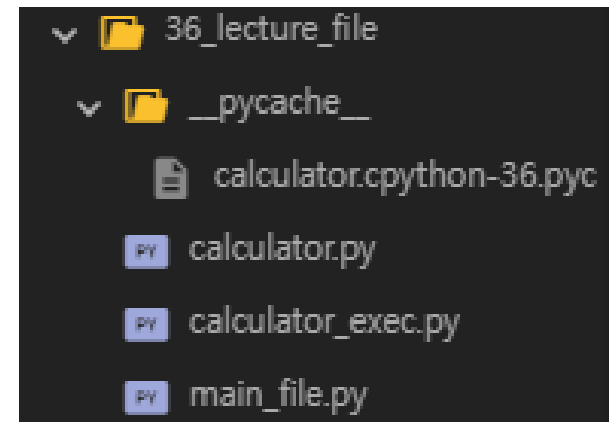
- 자신의 모듈을 넣는 방법은 여러 방법이 있다.
  - 영구적
    - PYTHONPATH 환경변수를 등록하기
    - site-packages 디렉터리에 집어넣기
  - 일시적
    - `sys.path.append("경로명")`를 통해 추가

# 모듈

---

## \_\_pycache\_\_

- 모듈을 import하면 생성된다.
- 성능을 향상시키기 위해 만드는 바이트 코드(byte code)이다.
- Cpython에서 소스코드를 바이트 코드로 컴파일한다.
- 삭제해도 무방하며, 다시 import하면 생성된다.
- python -B 파일명.py로 실행하면 생성하지 않는다.

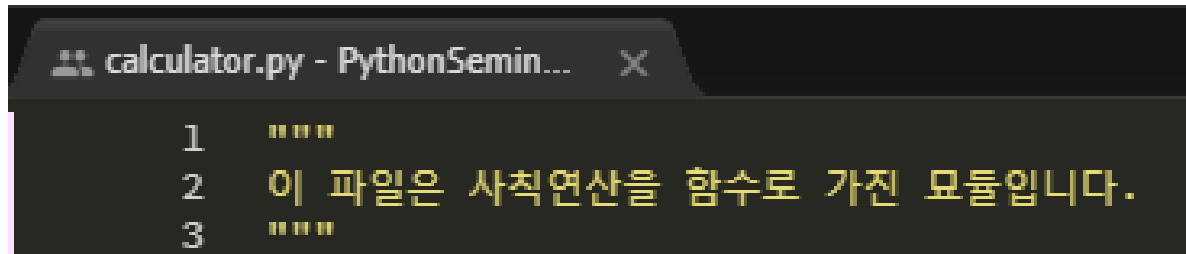


# 모듈

---

## 모듈의 doc\_string

- 모듈 파일의 첫 줄에 doc\_string을 넣는다.

A screenshot of a code editor window titled 'calculator.py - PythonSemin...'. The editor shows three lines of code: a docstring starting with four quotes, a line of Korean text '이 파일은 사칙연산을 함수로 가진 모듈입니다.', and another line of four closing quotes.

```
1  """  
2  이 파일은 사칙연산을 함수로 가진 모듈입니다.  
3  """
```



# 패키지

---

## 패키지란?

- 여러 모듈을 묶은 것
- Package=꾸러미. 즉, 모듈 꾸러미이다.
- 코드가 많고 복잡할 때 사용
- 프로그램에 많은 기능이 있어 모듈이 다양할 때,  
관련된 기능을 가진 모듈을 디렉터리에 모아서 생성

# 패키지

---

## 패키지 이름-pep8

- 모듈과 유사하게 짧고, 전부 소문자이다.
- \_(언더바)는 사용하지 않는다.

# 패키지

---

## 패키지 생성

- 다음과 같이 패키지를 구성

main.py

game

+--\_\_init\_\_.py

+--graphic.py

+--sound.py

# 패키지

---

## 패키지 생성

- 다음과 같이 패키지를 구성

main.py

game

+--\_\_init\_\_.py

+--graphic.py

+--sound.py

# 패키지

---

## 패키지 생성

- `__init__.py`
- python 3.3 이상부터는 이 파일이 없어도 패키지로 인식되나, 호환성을 위해 생성한다.
- 이 파일의 내용은 비워두어도 된다.

# 패키지

---

## 패키지 생성

—graphic.py

```
1  def render():
2      print("그래픽 렌더링")
3
4
5  def draw():
6      render()
7      print("화면에 그리기")
```

—sound.py

```
1  def render():
2      print("음향 만들기")
3
4
5  def play():
6      render()
7      print("음향 출력")
```

# 패키지

---

## 패키지 생성

—main.py

```
1  import game.graphic
2  import game.sound
3
4  game.graphic.render()
5  game.sound.play()
```

그래픽 렌더링  
음향 만들기  
음향 출력

# 패키지

---

## 패키지 가져오기(import)

- 모듈과 유사
  - `import 패키지.모듈1, [패키지.모듈2...]`
- 패키지 모듈이름 지정
  - `import 패키지.모듈 as 이름`
- 패키지 모듈의 일부 가져오기
  - `from 패키지.모듈 import 변수/함수/클래스`



# 패키지

---

## 패키지-\_\_init\_\_.py

- import 패키지로 모듈 가져오기
  - 패키지를 가져올 때, \_\_init\_\_.py를 통해 초기화한다.
  - 은 현재 패키지라는 말이다. (상대 위치)
- \_\_init\_\_.py

```
1 from . import graphic
2 from . import sound
```

- main.py

```
1 import game
2
3 game.graphic.render()
4 game.sound.play()
```

그래픽 렌더링  
음향 만들기  
음향 출력

# 패키지

---

## 하위 패키지

- 패키지내에 다른 패키지를 넣을 수 있다.
- 내부 디렉터리에 새로운 `__init__.py`와 모듈을 넣으면 하위 패키지가 된다.
- `import 패키지.하위패키지.모듈`

# 패키지

---

## 하위 패키지

- 하위패키지에서 옆의 패키지를 사용하기 위해서는  
from ..패키지 import 모듈  
로 가능하다.

# 패키지

---

패키지의 doc\_string

- `__init__.py`파일의 첫 줄에 doc\_string을 넣는다.