

---

# python study

2017.1.10

이영득

---

## 7장. 협력

- 49. 모든 함수, 클래스, 모듈에 docstring 을 작성하자
- 50. 모듈을 구성하고 안정적인 API를 제공하려면 패키지를 사용하자
- 51. 루트 Exception 을 정의해서 API로부터 호출자를 보호하자
- 52. 순환 의존성을 없애는 방법을 알자
- 53. 의존성을 분리하고 재현하려면 가상 환경을 사용하자

모든 함수, 클래스, 모듈에  
docstring 을 작성하자.

# docstring 을 작성하는 이유?

- 대화식 개발이 편해진다. (repr, help function)  
외부에서 설명을 쉽게 찾아볼 수 있음.
  - 소스코드에 적은 내용을 외부에서 바로 꺼낼 수 있기 때문에,  
문서 생성 도구의 활용의 폭이 커진다.
  - 코더 입장에서 문서가 더 잘 정리 되도록 장려한다.
- PEP-0257 을 보면 이미 잘 정리되어 있는 규칙을 확인할 수 있다.

# 문서 작성 컨벤션 (모듈)

- 큰따옴표 세 개를 사용한다 (""")
- 첫 번째 줄은 모듈의 목적을 기술한다
- 그 이후 문단은 모듈의 사용자가 일

```
CH49_docstring.py x CH50_package.py x CH51_root_exception
1 """Library for testing words for various linguistic patterns.
2
3 Testing how words relate to each ...
4
5 Available functions:
6 - palindrome: Determine if a word is a palindrome.
7 - check_anagram: Determine if two words are anagrams.
8 ...
9 """
10 """
```

```
Python Console
>>> help("CH49_docstring")
Help on module CH49_docstring:

NAME
CH49_docstring - Library for testing words for various linguistic patterns.

DESCRIPTION
Testing how words relate to each ...

Available functions:
- palindrome: Determine if a word is a palindrome.
- check_anagram: Determine if two words are anagrams.
...

CLASSES
builtins.object
Player
```

# 문서 작성 컨벤션 (클래스)

- 모듈과 대동소이 하다. 클래스 선언 바로 아래에 docstring 을 적는다.

```
55 class Player(object):
56     """Represents a player of the game.
57
58     Subclasses may override the 'tick' method to provide ...
59
60     Public attributes:
61     - power: Unused power-ups (float between 0 and 1)
62     - coins: ...(integer)
63     """
64     def __init__(self):
65         self.power = 0.0
66         self.coins = 1
67
```

Help on class Player in module \_\_main\_\_:

```
class Player(builtins.object)
| Represents a player of the game.
|
| Subclasses may override the 'tick' method to provide ...
|
| Public attributes:
| - power: Unused power-ups (float between 0 and 1)
| - coins: ...(integer)
|
| Methods defined here:
|
| __init__(self)
```

**help(Player)**

# 문서 작성 컨벤션 (함수)

- 마찬가지로 비슷하다. 필요하다면 args, returns, exceptions 등을 기술한다.

```
72 def find_anagrams(word, dictionary):  
73     """Find all anagrams for a word.  
74  
75     This function only runs as fast as the test for ...  
76  
77     Args:  
78         word: String of the target word.  
79         dictionary: Container with all strings that ...  
80  
81     Returns:  
82         List of anagrams that were found. ...  
83     """  
84     return word
```

```
Help on function find_anagrams in module __main__:  
  
find_anagrams(word, dictionary) help(find_anagrams)  
    Find all anagrams for a word.  
  
    This function only runs as fast as the test for ...  
  
    Args:  
        word: String of the target word.  
        dictionary: Container with all strings that ...  
  
    Returns:  
        List of anagrams that were found. ...
```

# 핵심 정리

- 모든 모듈, 클래스 함수를 `docstring`으로 문서화하자.  
코드를 업데이트할 때마다 관련 문서도 업데이트하자.
- 모듈: 모듈의 내용과 모든 사용자가 알아둬야 할  
중요한 클래스와 함수를 설명한다.
- 클래스: `class` 문 다음의 `docstring`에서 클래스의 동작,  
중요한 속성, 서브클래스의 동작을 설명한다.
- 함수와 메서드: `def` 문 다음의 `docstring`에서 모든 인수,  
반환 값, 일어나는 예외, 다른 동작들을 문서화한다.



모듈을 구성하고 안정적인 API를  
제공하려면 패키지를 사용하자.

# Package?

패키지란? 다른 모듈을 포함하는 모듈을 총칭.

대부분은 디렉터리 안에 `__init__.py`라는 빈 파일을 넣는 방법으로 패키지를 정의.

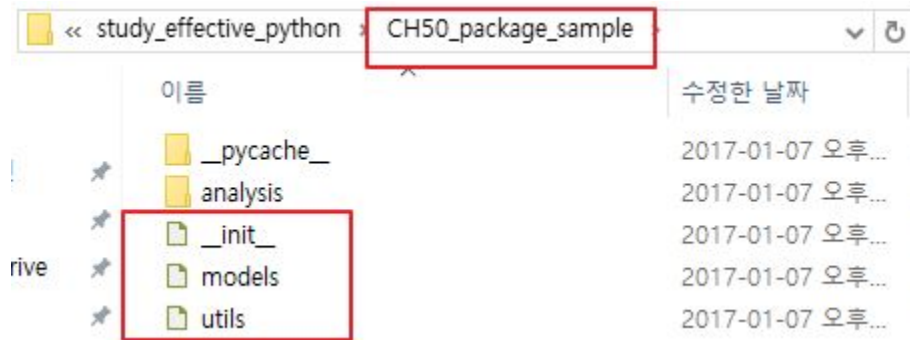
패키지를 활용해서 얻는 이점

- 네임스페이스의 역할을 한다.
- 안정적인 API를 제공할 수 있다. (원하는 부분만 `import`로 보낼 수 있다.)

# 패키지 기본 사용법

패키지를 만드는 방법

- Main.py  
CH50\_package\_sample/\_\_init\_\_.py  
CH50\_package\_sample/models.py  
CH50\_package\_sample/utils.py



사용자가 import 하는 방법

**from** CH50\_package\_sample **import** utils **as** new\_name  
(**from** Directory **import** module\_name **as** new\_name으로 생각하면 편하다)

# 네임스페이스 충돌 케이스

동일한 함수/클래스가 A 모듈과 B 모듈 모두에 있을 때,

```
40 from CH50_package_sample import utils
41 from CH50_package_sample.analysis import utils # 위 import 를 애가 덮어씀!
42
43 print(utils.util_func_a())
```

이 경우 as 절을 이용해서 해결할 수 있다.

```
49 from CH50_package_sample import utils as root
50 from CH50_package_sample.analysis import utils as analysis
51
52 print(root.util_func_a())
53 print(analysis.util_func_a())
```

(혹은 전체 import 한 후, 사용할 때마다 풀네임을 적을 수도 있다.)

# 모듈 전체가 아닌 일부만 사용이 가능하게 허용하기

`__all__` 속성을 이용한다.

```
models.py ×
1 """Models.py
2
3
4 __all__ = ['Projectile']
5
6
7 class Projectile(object):
8     def __init__(self, mass, velocity):
9         self.mass = mass
10        self.velocity = velocity
11
12
13 def models_func_a():
14     return 0
```

```
__init__.py ×
1 __all__ = []
2 from . models import *
3 __all__ += models.__all__
4 from . utils import *
5 __all__ += utils.__all__
6
```

```
55 from CH50_package_sample import *
56 a = Projectile(1.5, 3)
57 b = Projectile(4, 1.7)
58 result = simulate_collision(a, b)
```

```
utils.py ×
1 """Utils.py
2
3
4 from . models import Projectile
5
6 __all__ = ['simulate_collision']
7
8
9 def _dot_product(a, b):
10     return 2
11
12
13 def simulate_collision(a, b):
14     return 3
15
16
17 def util_func_a():
18     return 0
```

# 핵심 정리

- 파이썬의 패키지는 다른 모듈을 포함하는 모듈이다. 패키지를 이용하면 모듈 이름으로 코드를 분리하고, 충돌하지 않는 네임스페이스를 만들 수 있다.
- 간단한 패키지는 다른 소스 파일을 포함하는 디렉터리에 `__init__.py` 파일을 추가하는 방법으로 정의한다.
- `__all__` 이라는 속성에 공개하려는 이름을 나열하여 모듈의 명시적인 API를 제공할 수 있다.
- 공개할 이름만 패키지의 `__init__.py` 파일에서 임포트하거나 내부 전용 멤버의 이름을 밑줄로 시작하게 만들면 패키지의 내부 구현을 숨길 수 있다.
- 주의: 사용처에서 `from x import *` 식의 모든 코드를 임포트하는 것은 자제하자.
  - 다른 사람이 참조를 추적하기 어렵다
  - 만일 이름이 겹치게 되면, 나중에 임포트 된 모듈의 내용으로 덮어 써버린다.

루트 Exception을 정의해서 API로부터  
호출자를 보호하자.

# 예외 재정의의 필요성

- 루트 예외가 있으면 호출자가 API를 사용할 때 문제점을 이해할 수 있다.  
(호출자가 별도의 분기로 처리할 수 있다.)
- API 모듈의 코드 버그를 찾는데 도움이 된다.  
(의도하지 않은 예외는 API 코드에 있는 버그라고 본다.)
- 시간이 지나 특정 환경에서 더 구체적인 예외를 제공하여 API를 확장할 수 있다.



# 사용 예제

사용법은 C#의 그것과 매우 흡사하다.

```
23 # Exception 을 상속받은 커스텀 클래스는 사용자가 try/catch 에서 잡아낼 수 있다.
24 class Error(Exception):
25     """Base-class for all exceptions raised by this module."""
26
27
28 class InvalidDensityError(Error):
29     """There was a problem with a provided density value."""
30
31
32 # 여기서 새로운 예외가 추가되어도 유연하게 대처할 수 있다.
33
34 class NegativeDensityError(InvalidDensityError):
35     """A provided density value was negative."""
36
```

```
78
79
80 try:
81     weight = my_module.determine_weight(1, -1)
82 except NegativeDensityError as e:
83     raise ValueError('Must supply non-negative density') from e
84 except InvalidDensityError:
85     weight = 0
86 except Error as e:
87     logging.error('Bug in the calling code: %s', e)
88 except Exception as e:
89     logging.error('Bug in the API code: %s', e)
90
91 raise
```


# 핵심 정리

- 작성 중인 모듈에 루트 예외를 정의하면 API로부터 API 사용자를 보호할 수 있다.
- 루트 예외를 잡으면 API를 사용하는 코드에 숨은 버그를 찾는 데 도움이 될 수 있다.
- 파이썬 Exception 기반 클래스를 잡으면 API 구현에 있는 버그를 찾는 데 도움이 될 수 있다.
- 중간 루트 예외를 이용하면 API를 사용하는 코드에 영향을 주지 않고 나중에 더 구체적인 예외를 추가할 수 있다.

순환 의존성을 없애는 방법을 알자

# 순환 의존성의 대표적인 예

```
23 # app.py
24 import dialog
25
26 class Prefs(object):
27     def get(self, name):
28         pass
29
30 prefs = Prefs()
31 dialog.show()
```



```
23 # dialog.py
24 import app
25
26 class Dialog(object):
27     def __init__(self, save_dir):
28         self.save_dir = save_dir
29
30 save_dialog = Dialog(app.prefs.get('save_dir'))
31
```

```
save_dialog = Dialog(app.prefs.get('save_dir'))
AttributeError: module 'app' has no attribute 'prefs'
```

# Import가 실행되면 어떤 일이 일어날까?

1. `sys.path`에 들어있는 위치에서 모듈을 검색한다.
2. 모듈에서 코드를 로드하고 코드가 컴파일되게 한다.
3. 대응하는 빈 모듈 객체를 생성한다.
4. 모듈을 `sys.modules`에 삽입한다.
5. 모듈 객체에 있는 코드를 실행하여 모듈의 내용을 정의한다.

위의 케이스에서 상황은?

- `app` 모듈은 4번 단계를 실행하면서, `dialog` 모듈을 임포트 했다.
- `dialog` 모듈이 `app` 모듈을 참조했을 땐, 아직 `prefs`가 정의되지(5번 단계) 않은 상태이기 때문에 에러가 발생한다.

# 순환 의존성을 해결하는 세 가지 방법(1)

## 임포트 재정렬

: 위 케이스에서 app.py 안에 import dialog를 나중에 호출하는 방법.  
하지만 PEP 8 스타일 위반이다. 별로 바람직 하지 않음.

```
23 | # app
24 | class Prefs(object):
25 |     def get(self, name):
26 |         pass
27 |
28 |     prefs = Prefs()
29 |
30 |     import dialog # Moved
31 |     dialog.show()
```

# 순환 의존성을 해결하는 세 가지 방법(2)

임포트, 설정, 실행

: 임포트 하는 시점에 실제 함수를 실행하지 않는 방법이다. 즉 prefs를 사용하는 시점을 dialog가 임포트 되는 시점이 아닌 사용자의 직접 호출을 받을 때, 호출하도록 변경하는 것이다.

(호출부도 변경되어야 함)

```
23 # main
24 import app
25 import dialog
26
27 app.configure()
28 dialog.configure()
29
30 dialog.show()
```

```
23 # dialog.py
24 import app
25
26 class Dialog(object):
27     def __init__(self):
28         pass
29
30 save_dialog = Dialog()
31
32 def show():
33     print('Showing the dialog!')
34
35 def configure():
36     save_dialog.save_dir = app.prefs.get('save_dir')
```

# 순환 의존성을 해결하는 세 가지 방법(3)

## 동적 임포트

: import 문을 함수나 메서드에서 사용하는 방법. 코드 변경이 적으나, 단점도 있다. Import 문의 비용은 무시하지 못할 정도이다. 또, `SyntaxError` 예외가 프로그램의 실행을 시작한 후에 알 수 있다.

```
27 class Dialog(object):
28     def __init__(self):
29         pass
30
31 # Using this instead will break things
32 # save_dialog = Dialog(app.prefs.get('save_dir'))
33 save_dialog = Dialog()
34
35 def show():
36     import app # Dynamic import
37     save_dialog.save_dir = app.prefs.get('save_dir')
38     print('Showing the dialog!')
```



# 핵심 정리

- 순환 의존성은 두 모듈이 임포트 시점에 서로 호출할 때 일어난다.  
이런 의존성은 프로그램이 시작할 때 동작을 멈추는 원인이 된다.
- 순환 의존성을 없애는 가장 좋은 방법은 순환 의존성을  
의존성 트리의 아래에 있는 분리된 모듈로 리팩토링하는 것이다.
- 동적 임포트는 리팩토링과 복잡도를 최소화해서 모듈 간의 순환 의존성을  
없애는 가장 간단한 해결책이다.(지불하는 비용도 있다.)

의존성을 분리하고 재현하려면 가상  
환경을 사용하자

# 모듈 의존성?

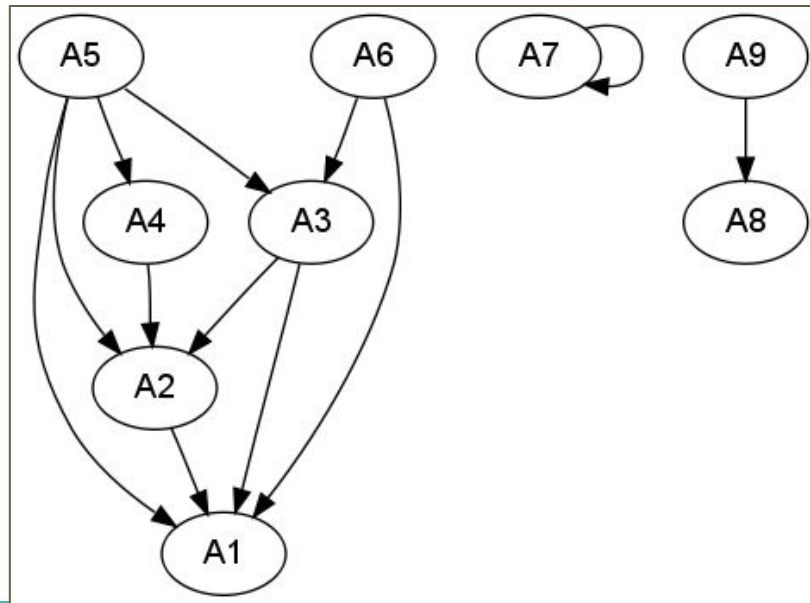
실제 코드를 개발하다 보면, 혹은 오픈소스를 연결하다 보면..

각 lib들의 requirements 가 동일하게 겹칠 수 있다.

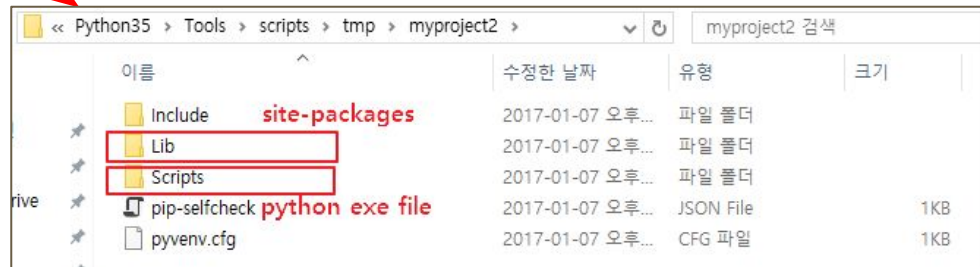
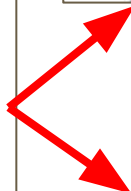
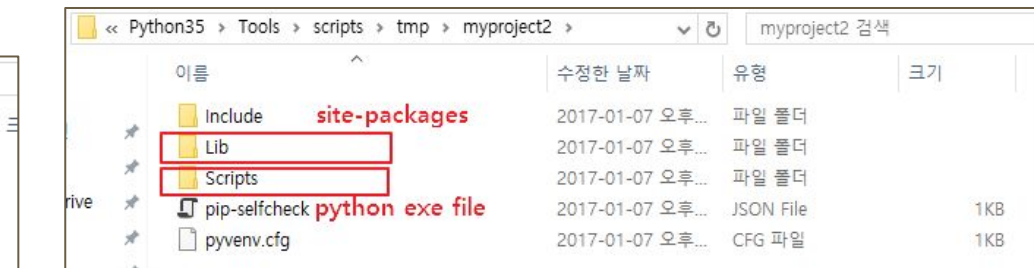
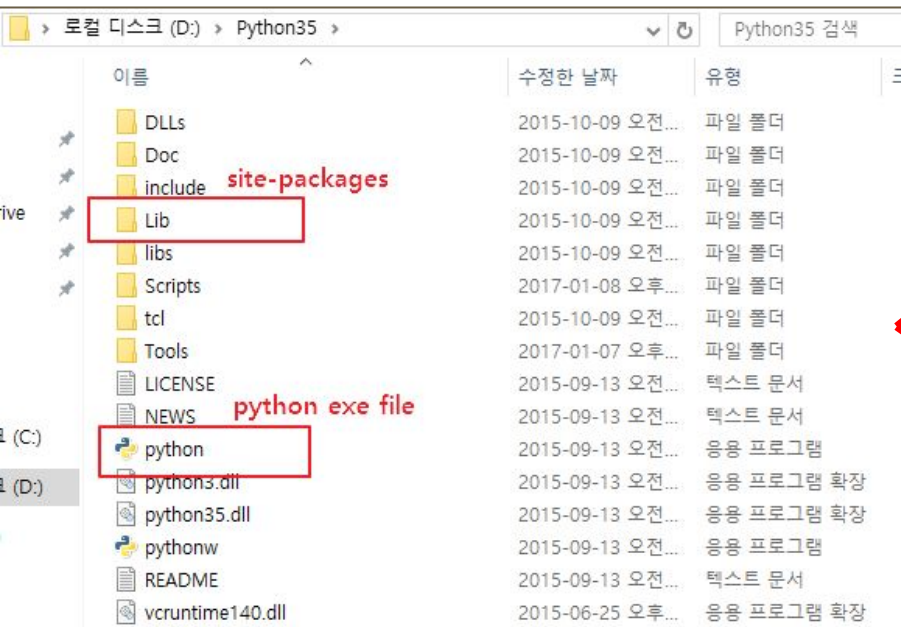
게다가 각 lib들은 호환 버전이 따로 있는 경우도 많다.

예> A2 경우 같은 것..

이를 효과적으로 제어하기 위해 버전을 clone 시켜서 별도의 repository를 갖게 해주는 것이 Pyenv라는 툴이다.

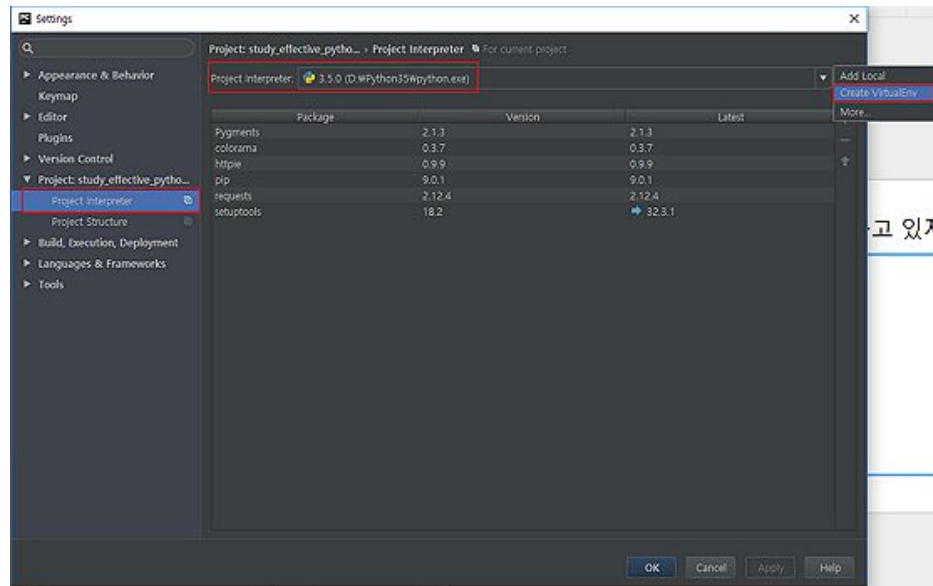


# Concept of pyvenv



# 편한 방법

- 책에선 cmd(or shell) 환경에서 설명하고 있지만... (cmd로 짧은 시연하기)
- 그런데 굳이 cmd 창에서 할 필요가 없다  
왜냐? 파이참이 다 지원하니까.  
그것도 GUI로.  
이것도 직접 보여 dream.



# 이건 내 생각

로컬에서 여러 버전을 사용하려면 확실히 `pyenv`가 편하다.

근데 아예 `docker`로 분리하는 것도 고려해볼만 하다. (특히 서버 플머)

어차피 무엇을 하건 내 로컬에서만 돌리려는 목적이 아니라면,  
`Docker`로 묶어 주는 것이 사용자 입장에서도 매우 편하다.

`Pyenv`는 어쨌든 로컬에서 파일을 별도로 보관하는 것이기 때문에,  
`Cmd`나 수동으로 관리한다면 귀찮은 타이핑을 많이 해야할 것 같다.

# 핵심 정리

- 가상 환경은 pip를 사용하여 같은 머신에서 같은 패키지의 여러 버전을 충돌 없이 설치할 수 있게 해준다.
- 가상 환경은 pyvenv로 생성하며, source bin/activate로 활성화하고 deactivate로 비활성화 한다.(리눅스 얘기, 윈도우는 batch 파일 실행)
- Pip freeze로 환경에 대한 모든 요구 사항을 덤프할 수 있다. Requirements.txt 파일을 pip install -r 명령의 인수로 전달하여 환경을 재현할 수 있다.
- 파이썬 3.4 이전 버전에서는 pyvenv 도구를 별도로 다운로드해서 설치해야 한다. 명령줄 도구의 이름은 pyvenv가 아닌 virtualenv다.