import

다음과 같이 `my_module.py` 라는 파이썬 파일을 생성해본다.

In [1]:

```
%%writefile my_module.py

# my_module.py

_value = 1

class MyClass:
    _num = 10

    def __init__(self, value, **kwargs):
        self.value = value
```

```
Writing my_module.py
```

모듈을 불러올 때는 `import` 키워드를 통해 불러온다.

In [2]:

```
import my_module
```

In [3]:

```
import inspect
```

In [4]:

```
print(inspect.getsource(my_module))
```

```
# my_module.py

_value = 1

class MyClass:
    _num = 10

    def __init__(self, value, **kwargs):
        self.value = value
```

접근 연산자 **dot(.)**을 통해 모듈 안에 있는 **MyClass**를 불러온다.

In [5]:

```
my_class = my_module.MyClass(10)
```

In [6]:

```
my_class.value
```

Out[6]:

```
10
```

모듈안에 포함된 식별자나 함수 앞에 언더스코어(_)를 붙이면 `import` 되지 않는다.

In [7]:

```
dir(my_module)
```

Out[7]:

```
['MyClass',
 '__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 '_value']
```

from

`from` 키워드를 통해서 `my_module` 모듈안에 있는 `MyClass` 만 따로 불러온다.

In [8]:

```
from my_module import MyClass
```

In [9]:

```
my_class = MyClass(20)
```

In [10]:

```
my_class.value
```

Out[10]:

```
20
```

as

`as` 키워드는 별칭을 붙이는 `alias`를 의미한다.

In [11]:

```
import my_module as mm
```

In [12]:

```
mm.MyClass
```

Out[12]:

```
my_module.MyClass
```

언더스코어(_)

앞서 살펴봤듯, 파이썬에서 언더스코어(**underscore,** `_`)는 조금 특별한 의미를 가진다.

아래처럼 언더스코어를 넣어 숫자를 리터럴로 선언 시 단위별로 구분하여 선언하는 것도 가능하다.

In [13]:

```
100_000_000
```

Out[13]:

```
100000000
```

In [14]:

```
1_0000_0000
```

Out[14]:

```
100000000
```

파이쏘닉(`Pythonic`)하게 아래처럼 필요없는 값을 무시하고자 할 때 사용하기도 한다.

In [15]:

```
a, *_, b = [1, 2, 3, 4, 5]
```

In [16]:

```
a
```

Out[16]:

```
1
```

In [17]:

```
b
```

Out[17]:

```
5
```

In [18]:

```
_
```

Out[18]:

```
[2, 3, 4]
```

In [19]:

```
data = zip([1,2,3,4,5], [6,7,8,9,10])
```

In [20]:

```
for i, (_, j) in enumerate(data):
    print(i, j)
```

```
0 6
1 7
2 8
3 9
4 10
```

파이썬 인터프리터에선 마지막으로 실행된 결과값이 `_` 라는 식별자에 저장된다.

```
PS C:\Users\zmfls> python
Python 3.6.8 |Anaconda, Inc.| (default, Feb 21 2019, 18:30:04) [MSC v.1916 64 bit (AMD64)] on win32
```

```
Type "help", "copyright", "credits" or "license" for more information.
>>> '안녕'
'안녕'
>>> _
'안녕'
>>> 10000
10000
>>> _
10000
>>>
```

먼저, 새로운 클래스 `Test`를 정의하였다.

In [21]:

```
class Test:
    a   = 10
    _b  = 20
    __c = 30

    def test(self):
        print('call test()')

    def _f(self):
        print('call _f()')

    def __func(self):
        print('call __func()')
```

In [22]:

```
test = Test()
```

PEP8에 의하면 언더스코어는 클래스 혹은 모듈 안에서만 사용할 수 있도록 외부에서 접근을 하지 못하도록**(=캡슐화)** 하는 접근 지정자 `private`와 같은 용도로써 사용하도록 권장하는 방식이다. 다만, 다른 객체지향 언어들과 같이 해당 멤버 혹은 메소드에 접근하지 못하는 것은 아니다.

언더스코어를 붙인 멤버 혹은 메소드는 그대로 호출이 가능하다. 그렇지만 `test.` 까지 입력하고 자동 완성키 `Tab`을 눌러보면 언더스코어(`_`)가 붙은 멤버 혹은 메소드는 보이지 않는다.

In [23]:

```
test._b
```

Out[23]:

```
20
```

In [24]:

```
test._f()
```

```
call _f()
```

맹글링 (Mangling)

파이썬에서 맹글링은 코드에 선언된 식별자명 혹은 함수명을 인터프리터에서 한번 변형하는 것을 의미한다.

In [25]:

```
Test.__c
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-25-a126715e3230> in <module>
----> 1 Test.__c

AttributeError: type object 'Test' has no attribute '__c'
```

In [26]:

```
test.__func()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-26-c4ca12aa8d4a> in <module>
----> 1 test.__func()

AttributeError: 'Test' object has no attribute '__func'
```

`dir` 함수를 통해 클래스에 정의된 멤버와 메소드를 살펴보면 클래스 안에 멤버명 혹은 메소드명 앞에 더블 언더스코어(`__`)를 붙이면 다른 이름으로 `맹글링 (Mangling)` 되어진 것을 확인할 수 있다.

In [27]:

```
dir(test)
```

Out[27]:

```
['_Test__c',
 '_Test__func',
 '__class__',
 '__delattr__',
 '__dict__',
```

```
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 '_b',
 '_f',
 'a',
 'test']
```

In [28]:

```
test._Test__c
```

Out[28]:

```
30
```

In [29]:

```
test._Test__func()
```

```
call __func()
```

몽키 패치(Monkey patch)

예시를 위해 먼저 `matplotlib` 패키지를 **import** 해본다. `matplotlib`은 데이터를 시각화**(Visualization)** 하기 위해 차트**(chart)**나 플롯**(plot)** 등 그래프를 그려주는 패키지이다.

In [30]:

```
import matplotlib
```

In [31]:

```
matplotlib.__version__
```

Out[31]:

```
'3.0.3'
```

`matplotlib` 패키지 안에는 아래의 코드를 실행한 결과에서 볼 수 있듯, **109**가지 모듈, 함수 및 클래스들이 정의되어있다.

In [32]:

```
len(dir(matplotlib))
```

Out[32]:

```
109
```

`몽키 패치(Monkey Patch)` 는 몽키 패칭 혹은 게릴라 패치라고도 불리며, 런타임중에 프로그램의 메모리를 직접 건드려 소스를 변형하는 행위를 말한다. 따라서, 몽키 패칭은 일반적인 경우에는 `안티 패턴(Anti pattern)` 으로 여겨진다.

이번에는 `matplotlib` 패키지의 `pyplot` 모듈을 **import** 해본다.

In [33]:

```
import matplotlib.pyplot as plt
```

`pyplot` 을 **import** 하게되면 몽키 패칭이 되어 사용할 수 있는 함수 및 클래스들이 더 추가되었음을 확인할 수 있다.

In [34]:

```
len(dir(matplotlib))
```

Out[34]:

```
172
```

---

type hint (형 힌트)

`type hint`는 식별자, 클래스 어트리뷰트 및 함수 매개변수나 반환 값의 기대되는 형을 지정하는 어노테이션이다.

형 힌트는 선택 사항이며 파이썬에서 강제되지는 않는다. 하지만, 정적 형 분석 도구에 유용하며 **IDE(Integrated Development Environment, 통합 개발 환경)**의 코드 완성 및 리팩토링(외부동작을 바꾸지 않으면서 내부 구조를 개선하는 방법)에 도움을 줄 수 있도록 한다.

In [35]:

```
def sum_two_numbers(a: int, b: int) -> int:
    return a + b
```

`Shift` + `Tab`키를 눌러 함수에 대한 설명을 확인해보면 시그니처에 파라미터는 어떤 타입이며, 반환 타입은 어떤 타입인지 알 수 있다.

In [36]:

```
?sum_two_numbers
```

하지만, 굳이 타입을 맞추지 않더라도 에러가 나진 않는다.

In [37]:

```
sum_two_numbers('안', '녕')
```

Out[37]:

```
'안녕'
```

annotation (어노테이션)

관습에 따라 `형 힌트`로 사용되는 식별자, 클래스 어트리뷰트 또는 함수 매개변수나 반환 값과 연결된 레이블이다.

지역**(local)** 변수의 어노테이션은 실행 시간에 액세스할 수 없지만, 전역**(global)** 변수, 클래스 어트리뷰트 및 함수의 어노테이션은 각각 모듈, 클래스, 함수의 `__annotations__` 특수 어트리뷰트에 저장된다.

In [38]:

```
sum_two_numbers.__annotations__
```

Out[38]:

```
{'a': int, 'b': int, 'return': int}
```