

12. 모듈과 패키지

12.1 모듈

- 모듈이란

모듈이란 파이썬 프로그램 파일이나 C, Fortran 확장 파일로 프로그램과 데이터를 정의하고 있으며, 고객(Client, 어떤 모듈을 호출하는 측)이 모듈에 정의된 함수나 변수의 이름을 사용하도록 허용하는 것이다. 다시 말하면 파이썬 모듈은 파이썬 프로그램으로 작성된 파일(*.py, *.pyc, *.pyo)이나 C나 Fortran 등으로 만든 파이썬 확장 파일(*.pyd)일수 있다.

모듈 파일은 어떠한 코드로도 작성할 수 있다. 함수를 정의할 수 있고, 뒤에서 배울 클래스를 정의할 수 있으며, 변수도 정의할 수 있다. 이렇게 정의한 내용은 다른 모듈에 의해서 호출되고 사용된다. 모듈은 코드들을 한 단위로 묶어 사용할 수 있게 하는 하나의 단위이다.

모듈은 서로 연관된 작업을 하는 코드들의 모임으로 구성된다. 작성중인 모듈의 크기가 어느 정도 커지게 되면 일반적으로 관리 가능한 작은 단위로 다시 분할한다. 지나치게 큰 모듈은 개념적으로나 실행 효율 면에서 좋지 않다. 이렇게 분리된 모듈은 코드의 독립성을 유지하여 나중에 재사용할수 있게 하는 것이 좋다.

모듈은 누가 제공하느냐에 따라서 표준 모듈, 사용자 생성 모듈, 서드파티 모듈로 나누어질수 있다. 표준 모듈은 파이썬 패키지 안에 포함된 모듈이고, 사용자 생성 모듈은 여러분이 만드는 모듈, 그리고 서드 파티 모듈은 협력 업체나 개인이 만들어서 제공하는 모듈이다.

모듈을 만들고 호출하는 간단한 예를 보자. 모듈을 만들기는 쉽다. 여러분이 필요로 하는 변수나 함수를 정의한 파이썬 파일을 만드는 것이 전부이다. (확장자는 py이어야 한다.) 다음 예를 보자.

```
# file : mymath.py
mypi = 3.14
def add(a, b):
    return a + b

def area(r):
    return mypi * r * r
```

앞의 코드는 mymath.py란 파일 이름으로 저장되었다. 이 모듈을 대화식 모드에서 호출해보자.

```
>>> import mymath
>>> dir(mymath)          # mymath에 정의된 이름을 확인한다.
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__',
'__spec__', 'add', 'area', 'mypi']
>>> mymath.mypi          # mymath 안에 정의된 mypi를 사용한다.
3.14
>>> mymath.area(5)       # mymath 안에 정의된 area를 사용한다.
78.5
```

첫 번째 명령 import는 mymath를 현재의 모듈로 가져온다. 모듈을 사용하기에 앞서서 먼저 실행해야 할 부분

이다. 모듈 이름은 원래 파일 이름에서 확장자를 제외한 것과 동일하다. 두 번째 명령 `dir(mymath)` 함수는 `mymath`에 정의한 각종 이름을 보여준다. 앞뒤에 밑줄(`_`)이 붙은 이름은 시스템에서 자동으로 설정한 것이고 마지막 세 개(`add`, `area`, `mypi`)가 모듈에서 정의한 이름이다. 세 번째 명령 `mymath.mypi` 는 `mymath` 이름 공간안의 `mypi`를 참조한다. 네 번째 명령 `mymath.area(5)`는 `mymath` 이름 공간안의 이름 `area`를 호출한다.

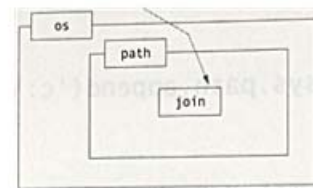
- 자격 변수와 무자격 변수

이름 공간(A)안에 있는 속성들(`x,y`)을 다른 공간(B)의 것들(`x,y`)과 구분하기 위해 다음과 같은 형식을 사용한다.

공간, 속성

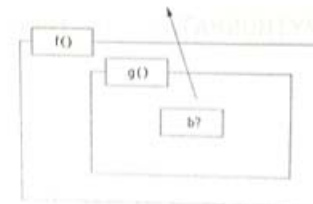
예를 들어, `X.Y.Z`는 이름 공간 `X`에서 `Y`를 찾고, 이름 공간 `Y`에서 `Z`를 찾는다. 다음코드를 보자.

```
>>> import os
>>> os.path.join('a','b')      # path.join은 자격이 있는 이름이다.
'a\\b'
```



점(.)으로 연결되지 않은 이름 공간이 주어지지 않은 이름은 LEGB 규칙에 따라서 먼저 찾아지는 이름을 취한다. 즉, 지역 영역과 내포된 함수 영역, 전역 영역, 내장 영역 순으로 이름 `W`를 찾는다.

```
>>> def f():
...     a = 1
...     def g():
...         print(b)      # b는 자격이 없는 이름이다.
```



`X,Y`와 같이 이름 공간이 명백히 주어진 변수를 자격 변수라고 하며, `W`와 같이 이름 공간이 명확하지 않은 변수를 무자격 변수라고 한다.

- 모듈 검색 경로

파이썬은 가져오기를 한 모듈로 특별히 지정한 폴더에서 찾아 나간다. 이 폴더는 다음과 같이 `sys.path`변수에서 확인할수 있다.

```
>>> import sys
>>> sys.path
['', 'C:\\Program Files\\Python36\\python36.zip', 'C:\\Program Files\\Python36\\DLLs', 'C:\\Program Files\\Python36\\lib', 'C:\\Program Files\\Python36', 'C:\\Program Files\\Python36\\lib\\site-packages']
```

이 폴더에서 모듈을 찾을수 없으면 `ImportError` 에러가 발생한다. `sys.path`변수에 직접 경로를 추가해도 검색 경로에 포함된다.

```
sys.path.append('c:\\myfolder')
```

만일 검색 경로에 사용자가 지정한 폴더를 포함시키고 싶으면 환경 변수 PYTHONPATH에 폴더를 추가하면 된다. 윈도우 7인 경우 [시작]->[제어판]->[시스템 및 보안]->[시스템]->[고급 시스템 설정]을 실행한후 [고급] 탭에서 [환경 변수]를 톨러 설정하면 된다. 예를 들어, 다음과 같이 설정할 수 있다.

```
PYTHONPATH=C:\Program Files\Python36\mypythonlib;
```

리눅스인 경우는 셀마다 다르므로 한마디로 설명하기 어렵지만, C 셸인 경우 ~/.cshrc 파일에 다음과 같이 정의할 수 있다.

```
setenv PYTHONPATH ~/mypythonlib
```

bash 셸인 경우라면 ~/.bash_profile 파일을 편집해서 다음과 같이 줄을 추가하면 된다. 그러나 패키지마다 다를수 있다.

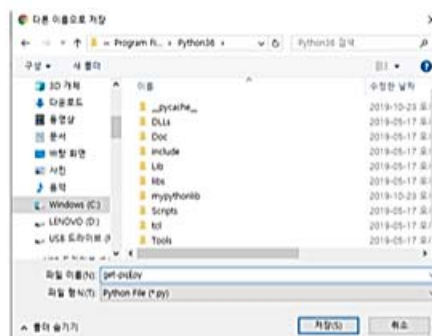
```
export PYTHONPATH=~/mypythonlib
```

- 절대 가져오기

1. pip 설치하기

<https://bootstrap.pypa.io>

Index of /



파이썬이 설치된 경로에 get-pip.py로 저장하고 cmd창에서 파이썬이 설치된 경로로 이동한다.

```
C:\Users\User> cd C:\Program Files\Python36
```

```
C:\Program Files\Python36>python get-pip.py
```



2. numpy사용하기

여기서 <https://www.lfd.uci.edu/~gohlke/pythonlibs/#numpy>

자신의 버전에 맞게 확인하여 다운받는다.

```
C:\Program Files\Python36>python -V
```

Python 3.6.4

파이썬 버전이 3.6이면 cp36

윈도우 64비트이면 amd64

numpy-1.17.3+mkl-cp36-cp36m-win32.whl

numpy-1.17.3+mkl-cp36-cp36m-win_amd64.whl

설치 하기

설치 명령어 : python -m pip install {다운로드 파일 경로}

ex) python -m pip install D:\numpy-1.17.3+mkl-cp36-cp36m-win_amd64.whl

실행 프로그램

```
C:\Program Files\Python36>python -m pip install D:\numpy-1.17.3+mkl-cp36-cp36m-win_amd64.whl
Processing d:\numpy-1.17.3+mkl-cp36-cp36m-win_amd64.whl
Installing collected packages: numpy
Successfully installed numpy-1.17.3+mkl
```

```
>>> import numpy as np
>>> x = np.array([1.0,2.0])
>>> print(x)
[1. 2.]
>>>
>>> A = np.array([[1,2],[3,4]])
>>> print(A)
[[1 2]
 [3 4]]
```

import가 잘 되었다!

파이썬의 가져오기에는 절대 가져오기와 상대 가져오기 두가지가 있다. 절대 가져오기는 항상 `sys.path` 변수에 정해진 순서대로 폴더를 검색해서 모듈을 가져오는 것이고 상대 가져오기는 현재 모듈이 속해 있는 패키지를 기준으로 상대적인 위치에서 가져올 모듈을 찾는다. 점(.)으로 시작하지 않는 것은 모두 절대 가져오기이다. 상대 가져오기는 패키지 절에서 설명한다.

절대 가져오기는 `sys.path` 변수에 기술된 경로만을 따라서 모듈을 가져온다. 다음과 같이 다양한 형식이 될 수 있다.

```
>>> import math # ①
>>> from math import sin, cos, pi # ②
```



```

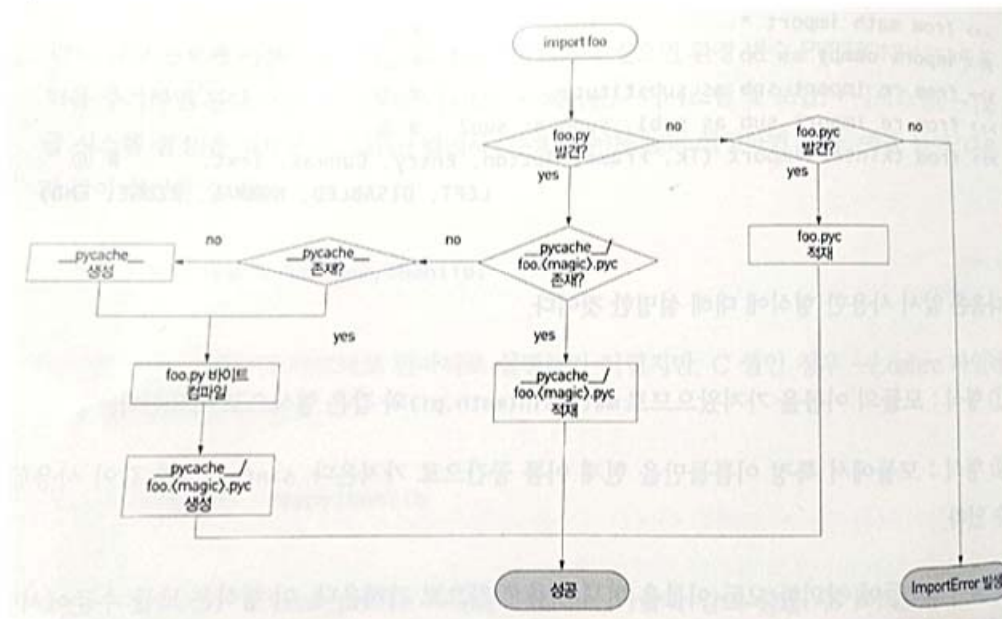
>>> from math import *                # ③
>>> import numpy as np                # ④
>>> from re import sub as substitute  # ⑤
>>> from re import sub as sub1, subn as sub2  # ⑤
>>> from tkinter import (Tk, Frame, Button, Entry, Canvas, Text, LEFT, DISABLED, NORMAL, RIDGE, END)
# ⑥

```

다음은 앞서 사용한 형식에 대해 설명한 것이다.

- ① 형식 : 모듈의 이름을 가져왔으므로 `math.sin(math.pi)`와 같은 형식으로 사용한다.
- ② 형식 : 모듈에서 특정 이름들만을 현재 이름 공간으로 가져온다. `sin(pi/2)`와 같이 사용할수 있다.
- ③ 형식 : 모듈에 정의한 모든 이름을 현재 이름 공간으로 가져온다. 이 형식은 모듈 수준에서만 사용할수 있고 함수내에서는 사용할수 없다.
- ④ 형식 : 모듈 이름을 다른 이름으로 사용하고자 할 때 사용한다. 모듈 이름이 너무 길거나, 사용중인 이름과 충돌이 일어날 때 사용할수 있다.
- ⑤ 형식 : 모듈에 정의한 이름을 다른 이름으로 사용하고자 할 때 사용한다.
- ⑥ 형식 : 하나의 모듈에서 여러개의 이름을 가져올 때 괄호를 사용할수 있다. 여러줄에 걸쳐서 `import` 문을 사용할수 있다.

다음으로 가져오기 과정을 `import foo` 문으로 알아보자. 우선 소스인 `foo.py` 파일을 찾는다. 이 파일이 있으면 `__pycache__` 폴더에서 바이트 코드 `foo,<magic>, pyc` 파일이 있는지 찾는다. 여기서 `<magic>`은 컴파일된 파이썬 버전을 나타내는 문자열이다. 예를 들으, `cpython-32` 이다. 따라서 `foo.cpython-32.pyc`가 바이트 코드인 파일이름일수 있다. 바이트 코드가 있으면 바이트 코드를 적재하고 만일 없으면 `foo,<magic>.pyc`파일을 만들어서 적재한다. 이 파일이 있어도 소스가 없으면 가져오기가 되지 않는다. 만일 소스가 있어야 할 폴더에 `foo.py` 파일이 없고 대신 `foo.pyc` 파일이 있으면 `foo.pyc`파일을 사용한다. 자세한 절차는 그림 12-1에 있다.



- `__name__` 변수

모듈은 모듈의 이름을 나타내는 내장 변수 `__name__`을 갖는다. 이 변수는 일반적으로는 자신의 모듈 이름을 가진다.

```
>>> import math
>>> math.__name__
'math'
```

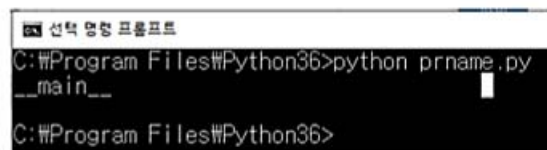
모듈의 이름을 출력하는 다음 예를 보자.

```
#file : prname.py
print(__name__)
```

이 파일을 대화식 인터프리터에서 가져오기를 해보자.

```
>>> import prname
prname ----- 자기 모듈의 이름을 출력한다.
```

이번에는 `prname.py` 파일을 직접 실행해 보자. 이 파일이 최상위 모듈이라면 `__name__` 변수는 `__main__` 형식의 이름을 가진다.



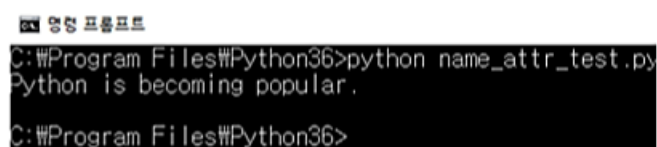
A terminal window titled '선택 명령 프롬프트' (Selected Command Prompt) showing the command `C:\Program Files\Python36>python prname.py` and the output `__main__`.

따라서 이것을 이용하면 최상위 모듈인지 가져온 모듈인지를 구분할 수 있다.

```
# file : name_attr_test.py
def test():
    print("Python is becoming popular.")

if __name__ == "__main__":
    test()
```

앞의 코드에서 `if` 문안의 `test`함수는 가장 먼저 실행되는 최상위 모듈이때만 실행되고,



A terminal window titled '명령 프롬프트' (Command Prompt) showing the command `C:\Program Files\Python36>python name_attr_test.py` and the output `Python is becoming popular.`

다른 모듈에 의해 가져올때는 실행되지 않는다.

```
>>> import name_attr_test
>>> name_attr_test.test()
```

Python is becoming popular.

>>>

따라서 모든 파이썬 모듈은 독립적으로 실행될 수 있으며, 다른 모듈에 의해 라이브러리처럼 실행될 수도 있다. 이것은 파이썬 모듈을 독립적으로 만드는 좋은 특징이다.

- 문자열로 표현된 모듈을 가져오기

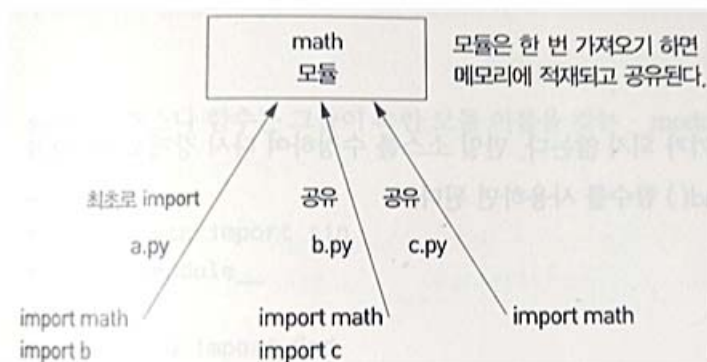
모듈 이름이 문자열로 표현되어 있을 때, 해당 이름의 모듈을 가져오는 방법은 `__import__()` 함수를 사용하는 것이다.

```
>>> modulename = 're'
>>> re = __import__(modulename)
>>> re
<module 're' from 'C:\Python32\lib\re.py' >
```

- 모듈의 공유

한번 가져온 모듈은 다른 모듈에서 가져오기가 요구되어도 이미 가져온 모듈을 공유한다.

(그림) 모듈의 공유



재귀적으로 가져오기를 한 경우를 생각해 보자, 두 개의 모듈 A와 B가 서로를 가져오려 한다.

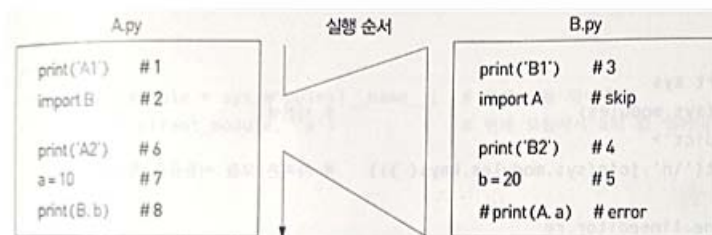
```
# file : A.py
print('A1')
import B
```

```
print('A2')
a=10;
print(B.b)
```

```
# file : B.py
print('B1')
import A

print('B2')
b=20
#print(A.a) # 오류발생
```

(그림) 재귀적인 가져오기



외부에서 A를 가져오려 할 경우 B.py 파일에서의 import A 문은 실행되지 않고 넘어간다. 따라서 재귀적인 무한 반복에 빠지지 않는다.

```
>>> import A
A1
B1
B2
A2
20
```

- 모듈의 재적재

한번 가져온 모듈은 다시 가져오기가 되지 않는다. 만일 소스를 수정하여 다시 강제로 가져오기를 하고 싶으면 imp 모듈의 reload() 함수를 사용하면 된다.

```
>>> import math
>>> import imp
>>> imp.reload(math)
<module 'math' (built-in)>
```

- 함수나 클래스가 속한 모듈 알아내기

파이썬을 실행하고 나서 한번이라도 가져온 모든 모듈은 sys.modules 변수에 남아있다.

```
>>> import sys
>>> type(sys.modules)          # 사전형
<class 'dict'>
>>> print('\n'.join(sys.modules.keys()))    # 가져온 모듈 이름들을 출력한다.
```



```

builtins
sys
_frozen_importlib
_imp
_warnings
_thread
_weakref
_frozen_importlib_external
_io
marshal
nt
winreg
zipimport
- 생략 -
heapq
_heapq
itertools
reprlib
_collections
weakref
collections.abc
warnings
importlib.machinery
importlib.util
- 생략 -
>>>

```

새로 가져오기를 하지않고 sys.modules 변수에서 모듈을 사용하는 것도 가능하다.

```

>>> sys.modules['heapq']
<module 'heapq' from 'C:\Program Files\Python36\lib\heapq.py'>
>>> heapq = sys.modules['heapq']
>>> heapq.merge([1, 2, 3])
<generator object merge at 0x000001F6BAE17360>

```

파이썬 클래스나 함수는 그들이 속한 모듈 이름을 갖는 __module__ 속성을 갖는다.

```

>>> from math import sin
>>> sin.__module__
'math'
>>> from cmd import Cmd
>>> Cmd.__module__

```

```
'cmd'
>>> sys.modules['cmd'] # 모듈 객체를 얻어낸다...
<module 'cmd' from 'C:\\Program Files\\Python36\\lib\\cmd.py'>
>>>
```

__name__ 변수를 이용하면 코드가 실행되는 현재 모듈을 얻어낼수도 있다.

```
>>> current_module = sys.modules[__name__] # 현재 모듈 얻어내기 얻어내기
>>> getattr(current_module, 'a') # 현재 모듈에서 a의 값 얻어내기
1
>>>
```

12.2 패키지

패키지는 모듈을 모아놓은 단위이다. 관련된 여러개의 모듈을 계층적인 몇 개의 디렉터리로 분류해서 저장하고 계층화한다.

- 패키지의 구조

예를 들어, 음성관련 패키지를 만든다고 해보자. 다음은 패키지의 구조이다.

```
Speech/ ----- 최상위 패키지
    __init__.py
    SignalProcessing/ ----- 신호처리 하위 패키지
        __init__.py
        LPC.py
        FilterBank.py
    Recognition/ ----- 음성인식 하위 패키지
        __init__.py
        Adaptation/
            __init__.py
            ML.py
        HMM.py
        NN.py
        DTW.py
    Synthesis/ ----- 음성합성 하위 패키지
        __init__.py
        Tagging.py
        ProsodyControl.py
```

- __init__.py 파일

각 디렉터리에는 __init__.py 파일이 반드시 있어야 한다.

이파일은 패키지를 가져올 때 자동으로 실행되는 초기화 스크립트이다.

이 파일이 없으면 해당 폴더는 파이썬 패키지로 간주하지 않는다.

__init__.py 파일은 패키지를 초기화하는 어떠한 파이썬 코드도 포함할 수 있다.

예를 들어, Speech/__init__.py 파일은 다음과 같다.

```
__all__ = [ 'Recognition' , 'SignalProcessing' , 'Synthesis' ]
__version__ = '1.2'
```

```
from . import Recognition          # 상대 가져오기
from . import SignalProcessing
from . import Synthesis
```

__all__ 변수는 from Speech import *문에 의해서 가져오기를 할 모듈이나 패키지 이름들을 지정한다.

```
>>> from Speech import *
```

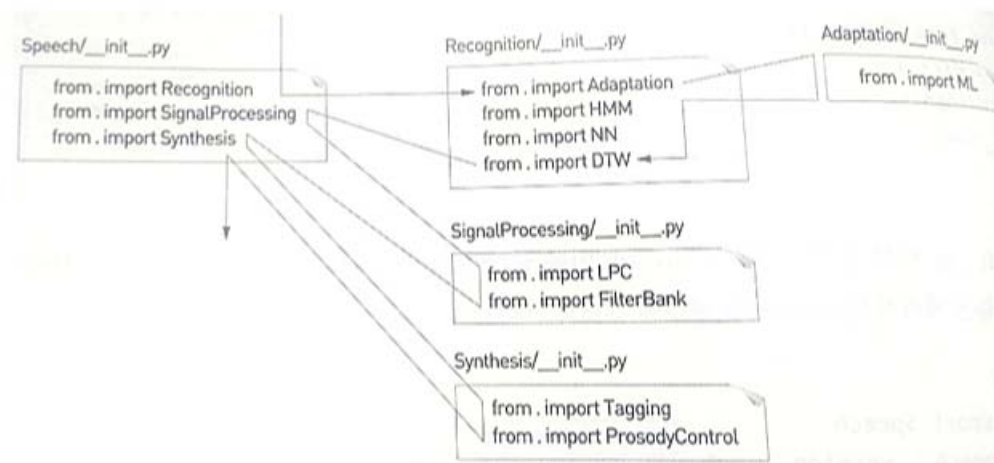
```
>>> dir()
['Recognition', 'SignalProcessing', 'Speech', 'Synthesis', '__all__', '__annotations__',
 '__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__', '__warningregistry__']
>>>
```

`__init__.py` 이름 공간은 패키지 `Speech` 이름 공간이다. 즉, `__init__.py` 이름 공간에 정의하는 이름들은 패키지 `Speech` 이름 공간에 그대로 드러난다.

```
>>> import Speech
>>> Speech.__version__          # 이름 공간 Speech의 __version__
'1.2'
>>> Speech.__all__
['Recognition', 'SignalProcessing', 'Synthesis']
>>> dir(Speech)
['Recognition', 'SignalProcessing', 'Synthesis', '__all__', '__builtins__', '__cached__', '__doc__',
 '__file__', '__loader__', '__name__', '__package__', '__path__', '__spec__', '__version__']
>>> Speech.Recognition
<module 'Speech.Recognition' from 'C:\Program
Files\Python36\lib\Speech\Recognition\__init__.py'>
>>>
```

`from . import Recognition`과 같은 문에 의해서 하위 패키지인 `Recognition`이 가져와 진다.
이 경우 역시 `Recognition/_init_.py` 파일이 실행되어 패키지를 초기화한다.
`Recognition/_init_.py` 파일의 예는 다음과 같다.

```
__all__ = [ 'Adaptation' , 'HMM' , 'NN' , "DTW" ]
from . import Adaptation
from . import HMM
from . import NN
from . import DTW
```



(그림) `import Speech` 문에 의해서 실행되는 코드의 순서

하위 패키지들이 모두 자동으로 초기화되었다. 따라서 다음과 같이 접근할 수 있다.

```
>>> import Speech
>>> Speech.Recognition.HMM.Train()      # 모듈 내 함수 호출
```

- 상대 가져오기

상대 가져오기는 현재 모듈이 속해있는 패키지를 기준으로 상대적인 접근 경로를 기술하는 것이다. 상대 가져오기는 점(.)으로 시작한다. 예를 들어, 앞 절의 패키지 구조에서 Speech/Recognition/NN.py 파일에서 사용할 수 있는 상대 가져오기는 다음과 같다.

```
from . import DTW                # 현재 패키지의 DTW 모듈
from . HMM import Train          # 현재 패키지의 HMM 모듈 안의 Train 함수
from . Adaptation import ML     # 하위 패키지 Adaptation의 ML 모듈
from ..SignalProcessing import LPC # 상위 폴더의 SignalProcessing 패키지의 LPC 모듈
from ..SignalProcessing.LPC import LinearPrediction
```

여기서 점(.)은 NN.py파일이 있는 폴더를 의미한다. 두 개의 점(..)은 한 폴더위, 세 개의 점(...)은 두폴더위를 의미한다. 하지만, 상대 가져오기는 모듈의 `_name_` 변수를 이용하여 패키지 구조안의 모듈 위치를 파악한다. 만일 모듈의 이름이 어떠한 패키지 정보도 갖고 있지 않으면(예를 들어, `_main_` 과 같은 경우)모듈은 모듈이 현재 위치해 있는 패키지 안이 아닌 최상위 수준에 위치하고 있는 것으로 간주하고 상대 가져오기를 하려든다. 따라서 상대 가져오기는 동작하지 않을 것이므로 상대 가져오기는 반드시 패키지 가져오기를 한 상태에서 수행해야 한다.

- `__main__.py` 파일

python의 인수로 *.py 파이썬 프로그램이 아닌 디렉터리를 지정할수 있다.

```
python my_program_dir
```

만일 `my_program_dir` 디렉터리 안에 `_main_.py` 파일이 있으면 이 파일이 자동으로 실행된다. 디렉터리가 아닌 zip 파일인 경우도 `_main_.py` 파일을 포함하고 있으면 `_main_.py` 파일을 실행한다.

```
python my_program.zip
```


12.3 프로그램 배포하기

distutils(Python Distribution Utility) 모듈은 파이썬 프로그램을 배포하고 설치할 때 사용하는 파이썬 표준 도구이다. 소스와 문서, 데이터, 스크립트 등을 한번에 묶어서 배포할수 있다. 배포하는 파일은 작성자의 의도대로 설치된다.

- setup 파일 설정하기

배포와 설치를 하려면 최상위 디렉터리에 setup.py 파일을 만들어야 한다.

```
# setup.py
from distutils.core import setup

setup(name="spam",
      version="1.0",
      description="setup test",
      author="Rhee, Soong Moo",
      author_email="hiland00@gmail.com",
      url="http://pythonworld.net/",

      py_modules = ['A', 'B', 'mymath02'],
      packages = ['Speech',
                  'Speech/Recognition',
                  'Speech/SignalProcessing',
                  'Speech/Synthesis'],
      #package_dir={'Speech': 'Speech2'},
      package_data={'Speech': ['images/*.jpg']},
      data_files=[('data', ['Speech/images/a1.jpg'])],
)
```

setup() 함수에는 다양한 옵션이 있다. 개별적인 모듈을 지정하려면 인수 py_modules를 사용한다. ['A', 'B']와같이 확장자를 뺀 모듈 이름만 추가하면 된다. 파일의 위치가 패키지 안에 있을경우는 package.module과 같이 지정할수 있다.

패키지를 포함하려면 인수 packages를 사용한다. 하위 패키지는 루트 패키지를 지정하면 자동으로 포함되지 않으므로 각각 지정해야 한다. 만일 패키지 이름과 패키지가 저장된 폴더가 다를경우에는 옵션 package_dir을 사용한다.

```
package_dir = { 'Speech' : 'Speech2' }
```

이렇게 지정하면 패키지 Speech는 Speech2 폴더에 저장되어 있다는 의미이다. 패키지 안에 있는 데이터 폴더 등은 자동으로 포함되지 않는다. 따라서 추가로 패키지 데이터를 지정하려면 인수 package_data를 다음과 같은 형식으로 사용한다.

‘패키지이름’ : [패키지에_복사되어야_할_파일의_상대경로 ‘]

패키지에 포함되지 않은 데이터 파일을 지정하려면 인수 `data_files`를 사용하는데 다음과 같은 형식으로 지정한다.

‘설치경로’ : [파일목록]

설치경로가 상대경로라면 `sys/prefix` 변수의 위치를 기준으로 한다.

```
>>> import sys
>>> sys.prefix
'C:\Program Files\Python36' ----- 윈도우
>>> sys.prefix
'/usr/local' ----- 리눅스
```

즉, 앞의 `setup.py` 파일에서 `data`는 `C:\Program Files\Python36\data`를 의미한다.

- 배포판만들기와 설치하기

`setup.py` 파일을 준비하였으면 배포판을 만들어 보자. 소스 배포판은 다음 명령을 실행한다.

```
python setup.py sdist
```

배포판은 기본적으로 `dist` 디렉터리에 `spacm-1.0.zip` 이나 `spam-1.0.tar.gz` 라는 이름으로 만든다. 소스 배포판을 얻어서 내 시스템에 설치하려면 압축을 풀고 다음 명령을 입력하면 된다.

```
python setup.py install
```

바이너리 배포판이란 소스가 아닌 실제로 설치할 결과물을 만들고 해당 결과물을 패키징하는 것을 의미한다. 다음 명령을 실행한다.

```
python setup.py sdist
```

그러면 `build` 디렉터리에 가상으로 설치하고 해당 결과물을 `dist` 디렉터리에 저장한다. 윈도우에서는 `spam-1.0.win32.zip` 파일이 만들어지고 리눅스에서는 `spam-1.0.linux-i686.tar.gz` 파일이 만들어진다. 이 배포판은 설치할 시스템 폴더를 만들므로 올바른 위치에 압축을 풀어야 한다. 파이썬 소스 파일로 구성된 경우에는 소스 배포판을 이용해서 설치하기가 오히려 더 쉽다. 하지만, 확장 모듈이 있으면 컴파일한 결과를 배포판에 넣어주는 편리함은 있다. 윈도우용 바이너리 배포판이라든가 RPM용 배포판을 만드는 것이 사용자에게 더 쉬울수 있다.

윈도우용 바이너리 배포판은 다음 명령으로만든다.

```
pytho setup.py bdist_wininst
```

dist 디렉터리에 spam-1.0.win32.exe 파일이 만들어진 다. 이 파일을 실행하면 GUI 설치 파일이 실행된다.

- 실행 파일 만들기

만일 여러분이 파이썬이 없어도 실행가능한 형태로 배포판을 만들고 싶으면 이에 맞는 서드 파티모듈이 있다.

- py2exe 파이썬 프로그램을 윈도우용 실행파일로 변환해 준다. 파이썬이 시스템에 설치되어 있지 않아도 실행할수 있다. (<http://www.py2exe.org>),

- py2app 맥 OS X에 독립실행형 응용프로그램을 만들어준다.
(<http://cx-freeze.sourceforge.net/>)

이들에 대한 사용법은 이 책의 범위를 벗어난다. 독자들이 스스로 찾아보기 바란다..