

LECTURE 2

Python Basics

(원본 자료) <http://www.cs.fsu.edu/~carnahan/cis4930/>

MODULES

```
''' Module fib.py '''
from __future__ import print_function

def even_fib(n):
    total = 0
    f1, f2 = 1, 2
    while f1 < n:
        if f1 % 2 == 0:
            total = total + f1
        f1, f2 = f2, f1 + f2
    return total

if __name__ == "__main__":
    limit = raw_input("Max Fibonacci number: ")
    print(even_fib(int(limit)))
```

fib.py로 저장

Modules이란 간단한 텍스트 파일로 Python 프로그램에 대한 정의와 구문으로 이루어져 있으며, 직접 실행이 가능하고, 다른 모듈에서 import하여 사용이 가능함

MODULES

- 모듈(module)이란 파일로 정의(definitions)들과 구문(statements)으로 구성
- 파일 이름은 모듈 이름에 .py 추가
- 자신의 모듈 이름은 global variable인 `__name__`을 통해서 접근 가능
- 하지만, 모듈이 직접 실행되는 경우 global variable인 `__name__`의 값은 “`__main__`”이 됨
- 모듈은 정의(definition)외에도 수행가능한 구문이 포함될 수 있다. import 구문이 처음으로 불러질 때, 해당 구문이 수행된다.

MODULES

모듈을 직접 command line에서 수행
하면, 모듈의 `__name__` 변수 값은
“`__main__`”을 가진다.

```
$ python fib.py
Max Fibonacci number: 4000000
4613732
```

```
''' Module fib.py '''
from __future__ import print_function

def even_fib(n):
    total = 0
    f1, f2 = 1, 2
    while f1 < n:
        if f1 % 2 == 0:
            total = total + f1
        f1, f2 = f2, f1 + f2
    return total

if __name__ == "__main__":
    limit = raw_input("Max Fibonacci number: ")
    print(even_fib(int(limit)))
```

MODULES

만약 import를 한 경우에는
__name__은 간단히 모듈의 이름 그
자체가 된다.

```
$ python
>>> import fib
>>> fib.even_fib(4000000)
4613732
```

```
''' Module fib.py '''
from __future__ import print_function

def even_fib(n):
    total = 0
    f1, f2 = 1, 2
    while f1 < n:
        if f1 % 2 == 0:
            total = total + f1
        f1, f2 = f2, f1 + f2
    return total

if __name__ == "__main__":
    limit = raw_input("Max Fibonacci number: ")
    print(even_fib(int(limit)))
```

MODULES

만약 import를 한 경우에는
__name__은 간단히 모듈의 이름 그
자체가 된다.

```
$ python
>>> import fib
>>> fib.even_fib(4000000)
4613732
```

그리고 단지 해당 값은 fib 모듈 안에
서만 접근이 가능하다

```
''' Module fib.py '''
from __future__ import print_function

def even_fib(n):
    total = 0
    f1, f2 = 1, 2
    while f1 < n:
        if f1 % 2 == 0:
            total = total + f1
        f1, f2 = f2, f1 + f2
    return total

if __name__ == "__main__":
    limit = raw_input("Max Fibonacci number: ")
    print(even_fib(int(limit)))
```

MODULES

모듈의 일부 정의를 import를 통해 가져올 수 있다

```
$ python
>>> from fib import even_fib
>>> even_fib(4000000)
4613732
```

모듈에 있는 모든 것을 가져올 경우

```
>>> from fib import *
```

```
''' Module fib.py '''
from __future__ import print_function

def even_fib(n):
    total = 0
    f1, f2 = 1, 2
    while f1 < n:
        if f1 % 2 == 0:
            total = total + f1
        f1, f2 = f2, f1 + f2
    return total

if __name__ == "__main__":
    limit = raw_input("Max Fibonacci number: ")
    print(even_fib(int(limit)))
```

MINI MODULE QUIZ

두 개의 모듈 foo.py와 bar.py.

```
''' Module bar.py '''

print "Hi from bar's top level!"

def print_hello():
    print "Hello from bar!"

if __name__ == "__main__":
    print "bar's __name__ is __main__"
```

```
''' Module foo.py'''
import bar

print "Hi from foo's top level!"

if __name__ == "__main__":
    print "foo's __name__ is __main__"
    bar.print_hello()
```

일반적으로 import 구문은 파일의 시작에 쓴다. (하지만, 사용되기 전에만 써도 문제는 없다)

MINI MODULE QUIZ

```
''' Module bar.py '''

print "Hi from bar's top level!"

def print_hello():
    print "Hello from bar!"

if __name__ == "__main__":
    print "bar's __name__ is __main__"
```

```
''' Module foo.py'''

import bar

print "Hi from foo's top level!"

if __name__ == "__main__":
    print "foo's __name__ is __main__"
    bar.print_hello()
```

\$ python bar.py

bar 모듈을 직접 실행할 경우의 출력값은?

MINI MODULE QUIZ

```
''' Module bar.py '''

print "Hi from bar's top level!"

def print_hello():
    print "Hello from bar!"

if __name__ == "__main__":
    print "bar's __name__ is __main__"
```

```
''' Module foo.py'''

import bar

print "Hi from foo's top level!"

if __name__ == "__main__":
    print "foo's __name__ is __main__"
    bar.print_hello()
```

```
$ python bar.py
Hi from bar's top level!  b
ar's __name__ is __main__
```

MINI MODULE QUIZ

```
''' Module bar.py '''

print "Hi from bar's top level!"

def print_hello():
    print "Hello from bar!"

if __name__ == "__main__":
    print "bar's __name__ is __main__"
```

```
''' Module foo.py'''

import bar

print "Hi from foo's top level!"

if __name__ == "__main__":
    print "foo's __name__ is __main__"
    bar.print_hello()
```

\$ python foo.py

foo 모듈을 직접 실행하였을 경우의 출력값은?

MINI MODULE QUIZ

```
''' Module bar.py '''

print "Hi from bar's top level!"

def print_hello():
    print "Hello from bar!"

if __name__ == "__main__":
    print "bar's __name__ is __main__"
```

```
''' Module foo.py'''

import bar

print "Hi from foo's top level!"

if __name__ == "__main__":
    print "foo's __name__ is __main__"
    bar.print_hello()
```

```
$ python foo.py
Hi from bar's top level!  H
i from foo's top level!  fo
o's __name__ is __main__
Hello from bar!
```

MINI MODULE QUIZ

```
''' Module bar.py '''

print "Hi from bar's top level!"

def print_hello():
    print "Hello from bar!"

if __name__ == "__main__":
    print "bar's __name__ is __main__"
```

```
''' Module foo.py'''

import bar

print "Hi from foo's top level!"

if __name__ == "__main__":
    print "foo's __name__ is __main__"
    bar.print_hello()
```

```
$ python
>>> import foo
```

interpreter에서 import하였을 경우?

MINI MODULE QUIZ

```
''' Module bar.py '''

print "Hi from bar's top level!"

def print_hello():
    print "Hello from bar!"

if __name__ == "__main__":
    print "bar's __name__ is __main__"
```

```
''' Module foo.py'''

import bar

print "Hi from foo's top level!"

if __name__ == "__main__":
    print "foo's __name__ is __main__"
    bar.print_hello()
```

```
$ python
>>> import foo
Hi from bar's top level!
Hi from foo's top level!
>>> import bar
```

그리고 다시 bar 모듈을 import한 경우

MINI MODULE QUIZ

```
''' Module bar.py '''

print "Hi from bar's top level!"

def print_hello():
    print "Hello from bar!"

if __name__ == "__main__":
    print "bar's __name__ is __main__"
```

```
''' Module foo.py'''

import bar

print "Hi from foo's top level!"

if __name__ == "__main__":
    print "foo's __name__ is __main__"
    bar.print_hello()
```

```
$ python
>>> import foo
Hi from bar's top level!
Hi from foo's top level!
>>> import bar
>>>
```

MODULE SEARCH PATH

모듈을 import하는 경우 python은 그것의 위치를 모른다. 따라서, 다음의 순서대로 해당 모듈의 위치를 찾는다.

- 내부 모듈(Built-in modules)
- `sys.path` 변수에 저장된 디렉토리 리스트. `sys.path` 변수는 다음과 같이 초기화 된다.
 - 현재 디렉토리 (current directory)
 - `PYTHONPATH` (a list of directory names, with the same syntax as the shell variable `PATH`).
 - The installation-dependent default.

`sys.path` 변수는 python 프로그램 안에서 변경 가능하다.

MODULE SEARCH PATH

`sys.path` 변수는 `sys` 모듈의 멤버변수로 다음과 같이 접근 가능하다.

```
>>> import sys
>>> sys.path
['', '/usr/local/lib/python2.7/dist-packages/D_Wave_One_Python_Client-1.4.1-py2.6-linux-x86_64.egg', '/usr/local/lib/python2.7/dist-packages/PyOpenGL-3.0.2a5-py2.7.egg', '/usr/local/lib/python2.7/dist-packages/pip-1.1-py2.7.egg', '/usr/local/lib/python2.7/dist-packages/Sphinx-....']
```

FUNCTIONS

만약, 파이썬을 이용하여 원격 머신에 접속하는 통신 프로그램을 작성한다고 하자.

```
def connect(uname, pword, server, port):  
    print "Connecting to", server, ":", port, "..."  
    # Connecting code here ...
```

FUNCTIONS

```
def connect(uname, pword, server, port):  
    print "Connecting to", server, ":", port, "..."  
    # Connecting code here ...
```

다음과 같이 함수가 불릴 수 있다

- `connect('admin', 'ilovecats', 'shell.cs.fsu.edu', 9160)`
- `connect('jdoe', 'r5f0g87g5@y', 'linprog.cs.fsu.edu', 6370)`

FUNCTIONS

함수 호출 시 동일한 파라미터를 반복적으로 쓰는 것이 불편할 수 있다.
그래서 파이썬에서는 파라미터의 default값을 설정할 수 있다.

```
def connect(uname, pword, server = 'localhost', port = 9160):  
    # connecting code
```

FUNCTIONS

```
def connect(uname, pword, server = 'localhost', port = 9160):  
    # connecting code
```

다양한 개수의 파라미터로 호출하여 다음의 예는 모두 유효한 호출이다.

- `connect('admin', 'ilovecats')`
- `connect('admin', 'ilovecats', 'shell.cs.fsu.edu')`
- `connect('admin', 'ilovecats', 'shell.cs.fsu.edu', 6379)`

SURPRISING BEHAVIOR

다음 예를 살펴보자. `add_item`이라는 함수는 `item`과 `item_list`를 파라미터로 받는 함수이다. 그리고 `item_list`의 default값은 `[]` (empty list)이다.

```
''' Module adder.py '''  
  
def add_item(item, item_list = []):  
    item_list.append(item) # Add item to end of list  
    print item_list
```

SURPRISING BEHAVIOR

다음 예를 살펴보자. `add_item`이라는 함수는 `item`과 `item_list`를 파라미터로 받는 함수이다. 그리고 `item_list`의 default값은 `[]` (empty list)이다.

```
''' Module adder.py '''  
  
def add_item(item, item_list = []):  
    item_list.append(item)  
    print item_list
```

```
$ python  
>>> from adder import *  
>>> add_item(3, [])  
[3]  
>>> add_item(4)  
[4]  
>>> add_item(5)  
[4, 5]
```

SURPRISING BEHAVIOR

이상한 양상을 보이지만, 이는 파이썬이 어떻게 동작하는지 보여주는 좋은 예

```
''' Module adder.py '''  
  
def add_item(item, item_list = []):  
    item_list.append(item)  
    print item_list
```

파이썬의 default argument는 함수가 정의될 때 단 한번만 evaluation 된다. (즉, 매번 부를 때 마다 가 아니다)

만약, 변경 가능한 default argument로 정의하게 되면, 이후 해당 변수의 변화되는 값들은 함수를 부를 때 반영이 된다.

```
$ python  
>>> from adder import *  
>>> add_item(3, [])  
[3]  
>>> add_item(4)  
[4]  
>>> add_item(5)  
[4, 5]
```


SURPRISING BEHAVIOR

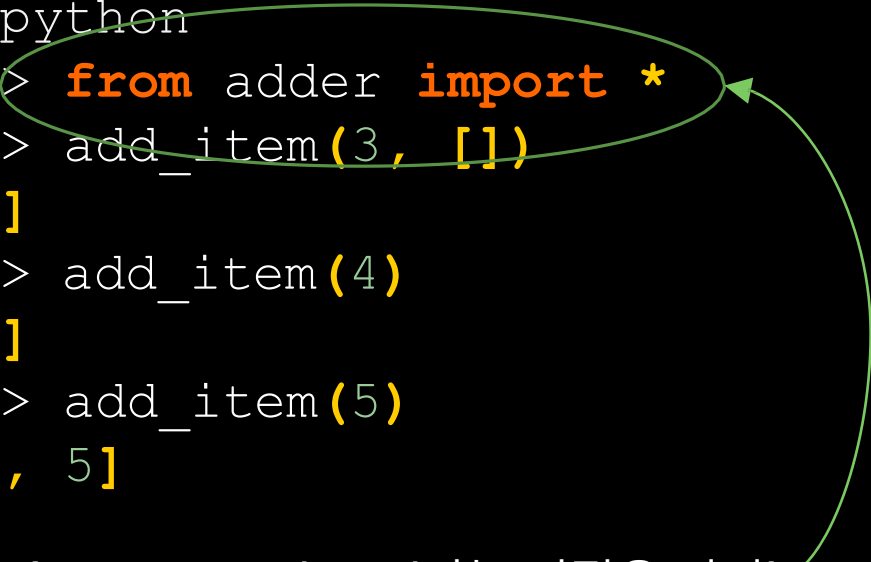
이상한 양상을 보이지만, 이는 파이썬이 어떻게 동작하는지 보여주는 좋은 예

```
''' Module adder.py '''  
  
def add_item(item, item_list = []):  
    item_list.append(item)  
    print item_list
```

파이썬의 default argument는 함수가 정의될 때 단 한번만 evaluation 된다. (즉, 매번 부를 때 마다 가 아니다)

만약, 변경 가능한 default argument로 정의하게 되면, 이후 해당 변수의 변화되는 값들은 함수를 부를 때 반영이 된다.

```
$ python  
>>> from adder import *  
>>> add_item(3, [])  
[3]  
>>> add_item(4)  
[4]  
>>> add_item(5)  
[4, 5]
```



Arguments evaluated되는 시점은 여기!

SURPRISING BEHAVIOR

다음과 같이 쉽게 고칠 수 있다.

```
''' Module adder.py '''  
  
def add_item(item, item_list = None):  
    if item_list == None:  
        item_list = []  
    item_list.append(item)  
    print item_list
```

```
$ python  
>>> from adder import *  
>>> add_item(3, [])  
[3]  
>>> add_item(4)  
[4]  
>>> add_item(5)  
[5]
```

FUNCTIONS

다시 connect함수의 경우를 살펴보자.

```
def connect(uname, pword, server = 'localhost', port = 9160):  
    # connecting code
```

다음의 함수 call은 argument의 위치를 가지고 binding을 수행한다.

```
connect('admin', 'ilovecats', 'shell.cs.fsu.edu', 6379)
```

FUNCTIONS

비 명시적으로 위치를 이용할 수도 있지만, 명시적으로 특정 argument를 지정할 수도 있다.

```
connect (uname='admin', pword='ilovecats', se  
        rver='shell.cs.fsu.edu', port=6379)
```

이 경우 argument 의 위치는 중요하지 않다

FUNCTIONS

다음과 같은 함수가 주어졌을 경우 유효한 함수 call은?

```
def connect(uname, pword, server = 'localhost', port = 9160):  
    # connecting code
```

1. `connect('admin', 'ilovecats', 'shell.cs.fsu.edu')`
2. `connect(uname='admin', pword='ilovecats', 'shell.cs.fsu.edu')`
3. `connect('admin', 'ilovecats', port=6379, server='shell.cs.fsu.edu')`

FUNCTIONS

다음과 같은 함수가 주어졌을 경우 유효한 함수 call은?

```
def connect(uname, pword, server = 'localhost', port = 9160):  
    # connecting code
```

1. `connect('admin', 'ilovecats', 'shell.cs.fsu.edu')` -- VALID
2. `connect(uname='admin', pword='ilovecats', 'shell.cs.fsu.edu')` -- INVALID
3. `connect('admin', 'ilovecats', port=6379, server='shell.cs.fsu.edu')` -- VALID

FUNCTIONS

함수에서 positional arguments는 args로 제공되고, 키워드 기반의 arguments는 dictionary 형식으로 제공된다.

```
def connect(uname, *args, **kwargs):  
    # connecting code here
```

이러한 방식은 packing으로 알려져 있다.

FUNCTIONS

```
def connect(uname, *args, **kwargs):  
    print uname  
    for arg in args:  
        print arg  
    for key in kwargs.keys():  
        print key, ":", kwargs[key]
```

```
connect('admin', 'ilovecats', server='localhost', port=9160)
```

Output: ?

FUNCTIONS

```
def connect(uname, *args, **kwargs):  
    print(uname)  
    for arg in args:  
        print(arg)  
    for key in kwargs.keys():  
        print(key, ":", kwargs[key])
```

```
connect('admin', 'ilovecats', server='localhost', port=9160)
```

Output:

```
admin  ilovecats  
port : 9160  
server : localhost
```

FUNCTIONS

*args 와 **kwargs 는 함수를 정의할 때도 사용가능하지만, 함수를 호출할 경우에도 사용가능하다. 다음 함수의 예를 살펴보자.

```
def func(arg1, arg2, arg3):  
    print "arg1:", arg1  
    print "arg2:", arg2  
    print "arg3:", arg3
```

FUNCTIONS

tuple을 만들어서 `*args` 형식으로 전달한다. 단 tuple의 값들은 argument의 순서대로 적혀져 있어야 한다.

```
>>> args = ("one", 2, 3)
>>> func(*args)
arg1: one
arg2: 2
arg3: 3
```

이를 argument tuple을 *unpacking* 한다고 한다.

FUNCTIONS

유사하게 dictionary 형식으로 해서 `**kwargs`를 전달하는 방식이다.

```
>>> kwargs = {"arg3": 3, "arg1": "one", "arg2": 2}
>>> func(**kwargs)
arg1: one
arg2: 2
arg3: 3
```

LAMBDA FUNCTIONS

파이썬의 특징 중에 하나는 lambda 함수를 정의할 수 있다는 것이다.

- `def`가 아닌 *lambda* 키워드를 통해 함수 선언
- 제약 조건은 하나의 expression만 수행 가능
- 일반적으로 functional programming 시에 활용된다. (추후 설명)

```
>>> def f(x):  
...     return x**2  
...  
>>> print f(8)  
64  
>>> g = lambda x: x**2  
>>> print g(8)  
64
```

LIST COMPREHENSIONS

List comprehensions은 어떤 연산의 결과를 list로 생성할 때 좋은 방법이다
가장 간단한 형식으로는

```
[expr for x in sequence]
```

for 구문 이후에 자유롭게 for나 if 구문 추가가 가능하다.

```
>>> squares = [x**2 for x in range(0,11)]  
>>> squares  
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

LIST COMPREHENSIONS

tuple 들의 list를 생성하는 예

```
>>> squares = [(x, x**2, x**3) for x in range(0,9) if x % 2 == 0]
>>> squares
[(0, 0, 0), (2, 4, 8), (4, 16, 64), (6, 36, 216), (8, 64, 512)]
```

특pecially 초기화식에 대한 제약 조건은 없다.(list comprehension도 가능하다.)

```
>>> [[x*y for x in range(1,5)] for y in range(1,5)]
[[1, 2, 3, 4], [2, 4, 6, 8], [3, 6, 9, 12], [4, 8, 12, 16]]
```