Programming in Python

04 – 함수, 클래스, 모듈

2016년 8월, 국민대학교 컴퓨터공학부

함수 (Functions)

- Python 함수란?
  - Python 문장들을 묶어 둔 단위
  - 코드 재사용성을 높이고, 불필요한 중복을 최소화하기 위한 프로그램 구조

- 왜 함수를 이용하는가?
  - 반복적으로 이용되는 기능을 한 곳에 모아 둠으로서,

    같은 코드를 반복하여 입력할 필요가 없어진다.
  - 특정한 기능에 대한 수정이 필요해지면, 해당 함수만 고치면 된다.

- Python 함수가 C/C++ 함수와 다른 점들은?

Python 함수를 이용한 코딩

```
def <name>(arg1, arg2, ···, argN):
    <statements>
```

```
def <name>(arg1, arg2, ···, argN):
    ···
    return <value>
```

- def 는 실행되는 코드이다!
  - 이것이 실행되기 전까지는 함수 객체는 존재하지 않음
  - if 블록 안에서, while 순환문 안에서 등 실행되는 도중에 def 가 함수 생성하는 것이 가능
- def 는 객체를 생성하여 이것을 이름에 연결한다.
  - 함수는 다른 객체에 비하여 특별할 것이 없는 Python 객체
  - 함수 이름 또한 다른 객체들과 마찬가지로 취급이 가능 (예: 리스트의 원소)
- 함수 내에서는 별도의 namespace 가 생긴다.

def 는 함수의 선언이 아니라 실행되는 문장

```
if test:
        def func():          # 함수를 이렇게 정의하거나
                ...
else:
        def func():          # 조건에 따라 다르게 정의하고
                ...
...
func()                       # 위 조건에 따라 정의된 함수를 호출

othername = func             # 함수 객체를 다른 이름에 대입
othername()                  # 이 이름으로 동일한 함수 (객체) 를 호출

def func(): ...              # 함수 객체를 생성
func()                       # 호출
func.attr = value            # 속성을 부여
```

함수의 생성과 호출

```
>>> def times(x, y):                    # 함수를 생성하여 times 라는 이름을 부여
...         return x * y
...

>>> times(2, 4)                         # 괄호 안에 인자를 나열하여 호출, 정해진 연산 결과가 리턴됨
8

>>> x = times(3.14, 4)                  # 리턴된 결과에 이름을 부여하여 객체를 저장
>>> x
12.56

>>> times('Ni', 4)                      # 함수는 인자와 리턴값에 대하여 자료형이 정해져 있지 않음
'NiNiNiNi'
```

함수의 생성과 호출

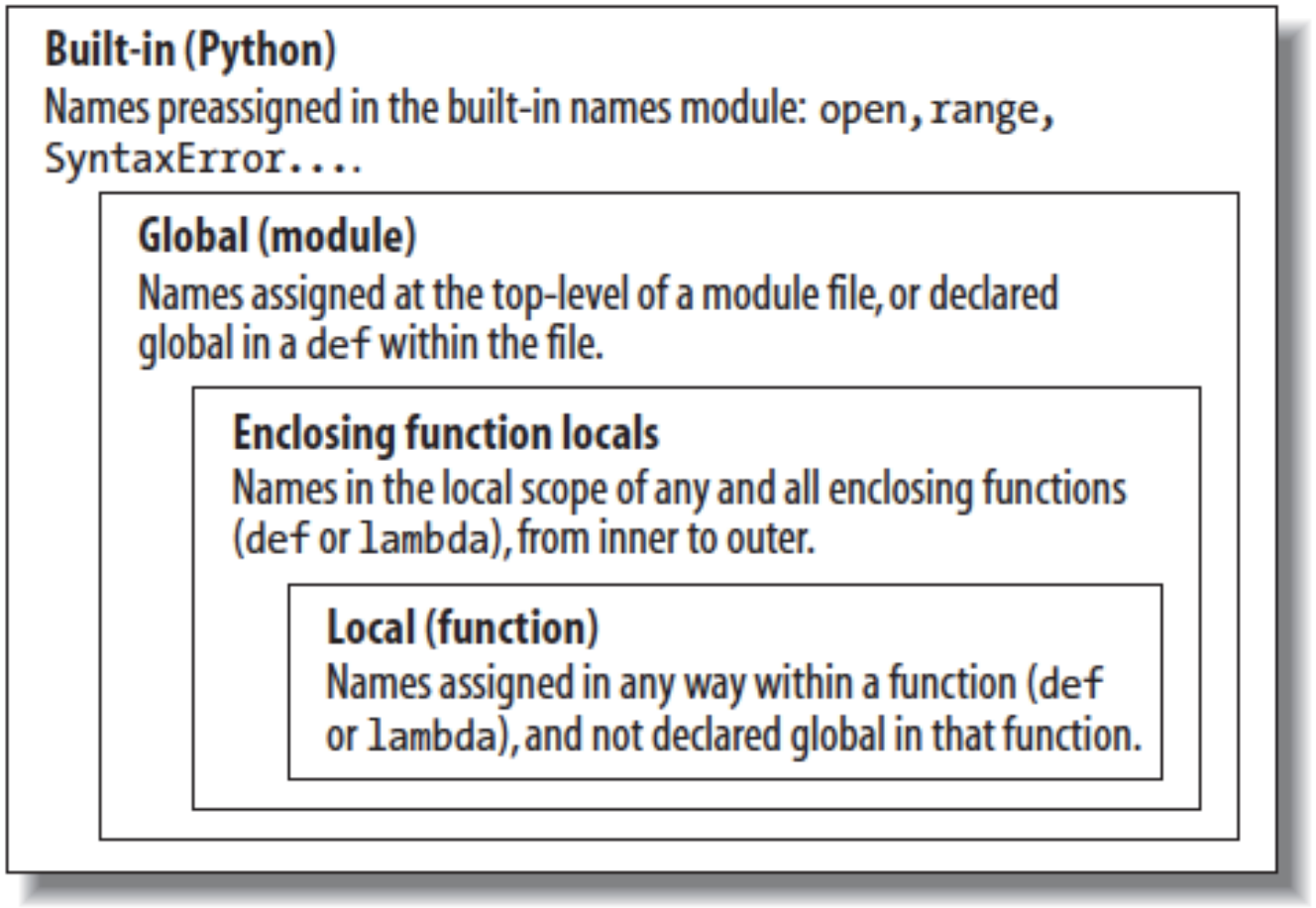
```
def intersect(seq1, seq2):
    res = []                        # 비어 있는 리스트에서 시작
    for x in seq1:                  # seq1 에 들어있는 객체를 스캔하여
        if x in seq2:               # 이것이 seq2 에도 있다면
            res.append(x)           # res 에 추가한다.
    return res

>>> s1 = "SPAM"
>>> s2 = "SCAM"
>>> intersect(s1, s2)               # [x for x in s1 if x in s2]
['S', 'A', 'M']

>>> x = intersect([1, 2, 3], (1, 4)]    # 자료형이 섞여 있어도 상관하지 않음
>>> x                               # 리턴되는 객체는 리스트임
[1]
```

범위 규칙 (Scope Rules)

- LEGB 규칙
  - Local
  - Enclosing
  - Global
  - Built-in



범위 규칙 (Scope Rules)

```
# Global scope
X = 99                  # X 는 global

def func(Y):            # func 는 global
    # Local scope
    Z = X + Y           # Y 와 Z 는 local
    return Z

func(1)                 # 결과는? (100)
```

```
def hider():
    open = 'spam'       # built-in 범위의 객체는 가려짐
    …                     (이제 open 은 local)
    open('data.txt')    # 파일을 열지 못함
```

```
X = 88                  # Global X

def func():
    X =99               # local X: global 은 가려짐

func()
print(X)                # 결과는? 88
```

global

```
X = 88                  # Global X

def func():
    global X
    X = 99              # 여기서의 X 는 global scope 를 가짐

>>> X
88

>>> func()
>>> X                   # 함수의 실행 결과 X 의 값이 바뀌었음
99
```

```
y, z  = 1, 2            # 모듈 내에서 global

def all_global():
    global x            # x 에 대하여 global 선언
    x = y + z           # 모든 변수가 global
```

OOP (Object-Oriented Programming) 와 class

- 객체지향 프로그래밍을 이용하는 이유는?
  - 프로그램을 만드는 데 있어서 복잡한 것을 나누어 단순하게 생각할 수 있다.
  - 객체를 모아 라이브러리 형태로 제공할 수 있다.
- Python 에서 유용한 객체지향 개념
  - Inheritance
    - 상위 클래스에서 만들어진 속성이나 메서드를 하위 클래스가 이어받음
  - Composition
    - 함께 기능하는 객체들과 그들 사이의 상호작용을 묶어서 관리할 수 있음
  - Multiple instances
    - Class 는 같은 종류의 객체를 생산해 내는 공장 (factory)
  - Customization via inheritance
    - 상속을 통하여 베이스 클래스의 속성을 이어받지만, 목적에 맞추어 개별화할 수 있음
  - Operator overloading
    - 런타임에 결정되는 자료형에 따라 서로 다른 연산을 적용할 수 있음

class 예제

cat.py

```
class Cat:
    def speak(self):
        print("야옹!")

    def drink(self):
        print("고양이가 우유를 마십니다.")
        print("고양이가 낮잠을 잡니다.")
```

```
% python cat.py
???
```

main.py

```
import cat

# 로미오라는 이름의 Cat 인스턴스 생성
romeo = cat.Cat()

# 로미오 실행
romeo.speak()
romeo.drink()
```

```
% python main.py
야옹!
고양이가 우유를 마십니다.
고양이가 낮잠을 잡니다.
```

__init__()

cat.py

```
class Cat:
    # 생성자 (constructor)
    def __init__(self, name):
        self.name = name

    def speak(self):
        print(self.name, "가 야옹합니다.")

    def drink(self):
        print(self.name, "가 우유를 마십니다.")
        print(self.name, "가 낮잠을 잡니다.")
```

Class method 는 self 를 첫번째 인자로 취하는데,
이는 이 클래스에 의하여 생성된 객체 자신을 가리키며
호출에서 명시적으로 괄호 안에 포함시키지 않음

```
>>> romeo = cat.Cat("Romeo")
>>> romeo.speak()
???
>>> romeo_drink()
???
```

같은 클래스의 복수 인스턴스 생성

main.py

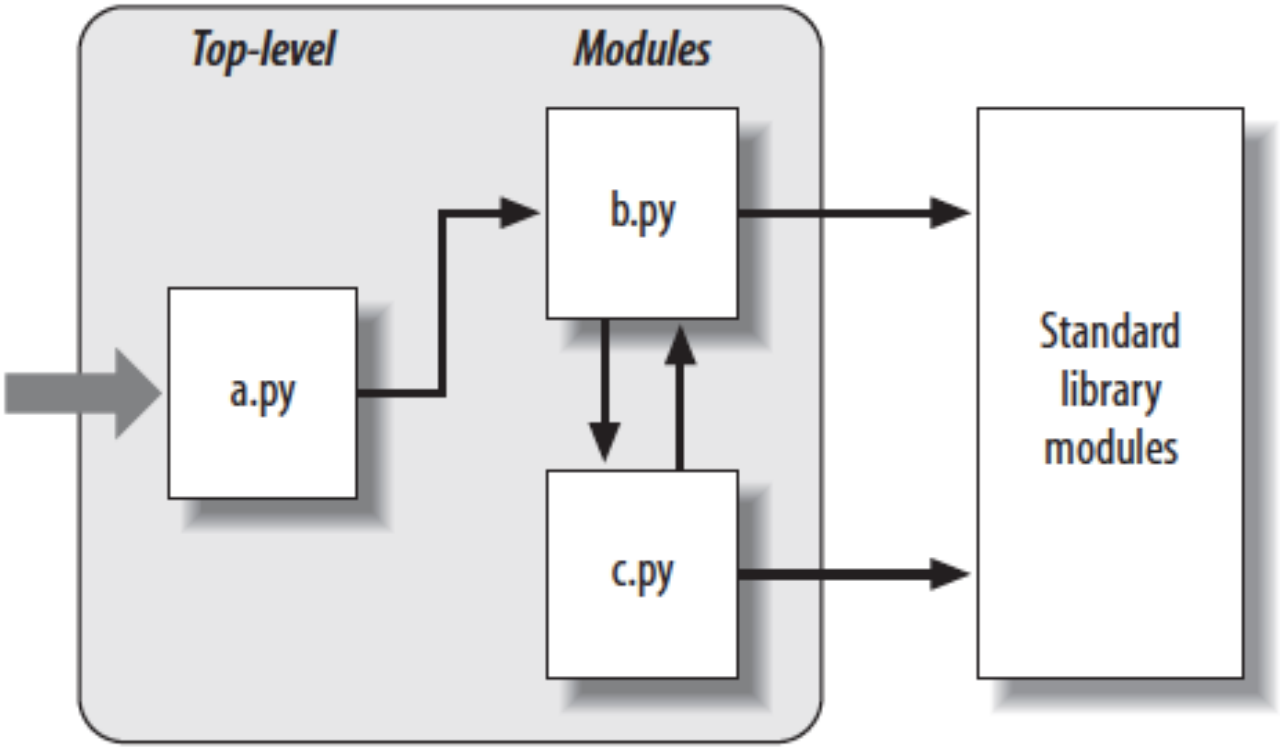
```
import cat

# 두 마리의 Cat 인스턴스 생성
romeo = cat.Cat("Romeo")
juliet = cat.Cat("Juliet")

# 로미오와 놀아준다.
romeo.speak()
romeo.drink()

# 줄리엣과 놀아준다.
juliet.speak()
juliet.drink()
```

모듈 (Modules)



a.py 는 최상위 스크립트 파일

여기에서 b.py 와 c.py 를 import 하여 이용

b.py 와 c.py 에서는 표준 라이브러리 모듈을 import 하여 이용

모듈을 이용하는 이유

- 코드 재사용성 향상
  - 잘 만들어진 모듈은 저장해 두고 필요할 때마다 불러서 (import) 이용할 수 있다.
- 시스템 네임스페이스의 관리
  - 모듈 내부에서만 이용되는 이름들은 밖으로 보일 필요 없다.
  - 따라서 모듈은 시스템 컴포넌트를 구성하는 기본적인 방법이다.
- 공유되는 컴포넌트를 구현하는 데 이용
  - 시스템 전체에서 공유되는 기능을 구현하여,

    서로 다른 다수의 클라이언트에서 import 하여 이용할 수 있는 아키텍쳐 제공

모듈 검색 경로

Import 문장을 만나면, Python 은 정해진 위치에서 모듈을 탐색

아래의 순서가 지켜짐

1. 프로그램의 홈 디렉토리
    - 프로그램의 최상위 스크립트가 위치한 디렉토리
2. PYTHONPATH 라는 환경 변수에 지정된 위치
    - 복수의 경로를 지정할 수 있는데 (세미콜론으로 구분), 먼저 지정된 위치가 우선 탐색됨
3. 표준 라이브러리가 위치하는 디렉토리
    - 시스템이 설치될 때 표준 라이브러리가 설치된 디렉토리(들)
4. 존재한다면, .pth 파일이 가리키는 위치
    - 파일에 지정할 수 있음

Quiz

- Global 변수를 많이 이용하는 것이 좋은가, 좋지 않은가?
  - 그 이유는 무엇인가?

- 아래 각각의 실행 결과는?

```
>>> X = 'Spam'
>>> def func():
...     print(X)
...
>>> func()
```

```
>>> X = 'Spam'
>>> def func():
...     X = 'NI!'
...
>>> func()
>>> print(X)
```

```
>>> X = 'Spam'
>>> def func():
...     X = 'NI!'
...     print(X)
...
>>> func()
>>> print(X)
```

```
>>> X = 'Spam'
>>> def func():
...     global X
...     X = 'NI!'
...
>>> func()
>>> print(X)
```

Exercise

(1) 고양이끼리 싸움을 시키는 메서드를 작성해 본다.

romeo.fight(juliet) 하면

“Romeo 에게 Juliet 이 싸움을 걸어 왔습니다.” 를 출력하고

“Juliet 이 이겼습니다.” (이름이 더 긴 쪽) 를 출력하는

fight() 메서드를 새로 Cat class 에 추가해 본다.

Q & A