Python Language Rules

2020.04.30

학부생 연구원 이지헌

배경

- Python은 Google에서 사용되는 주요 동적 언어
- Google Python Style Guide는 파이썬 프로그램에 대한 **할 일**과 **하지 말아야 할 일**의 목록
- http://google.github.io/styleguide/pyguide.html

사용하지 않는 언어의 기능

- 가능한 String module 대신에 **string method**를 사용

Python Shell

```
>>> import string

>>> string.split('Hello,world', ',')   // 문자열을 분리한 리스트 구함 ( X )
```

```
>>> 'Hello,world'.split(',')   ( O )

['Hello', 'world']
```

사용하지 않는 언어의 기능

- apply 함수는 파이썬 2, 3에서 **사용 중단** 안내함

  일련의 동적인 인수에 함수를 호출해야 할 때는 함수 호출(function call) 구문을 사용

Python Shell

```
>>> apply(fn, args, kwargs) ( X )
```

```
>>> fn(*args, *keywords) ( O )
```

사용하지 않는 언어의 기능

- 함수의 인자 값이 **inlined lambda**일 때 filter( ), map( ) 대신에

  list comprehensions와 for문을 사용

[Inlined lambda]

Python Shell

```
>>> li = [1, 2, 3]

>>> list(map(lambda i: i ** 2, li)

[1, 4, 9]
```

사용하지 않는 언어의 기능

- 함수의 인자 값이 **inlined lambda**일 때 filter( ), map( ) 대신에

  list comprehensions와 for문을 사용

[For문 사용]

Python Shell

```
>>> li = [1, 2, 3]

>>> result = [ ]
>>> for i in li:
        temp = i ** 2
        result.append(temp)

>>> result
[1, 4, 9]
```

사용하지 않는 언어의 기능

- reduce 함수는 파이썬 3 부터 **내장 함수에서 제외**

  가독성을 위해 for문을 사용

[reduce 함수 사용]

Python Shell

```
>>> from functools import reduce

>>> a = [1, 2, 3, 4, 5]

>>> reduce(lambda x, y: x + y, a)   // 1부터 5까지 더하기

15
```

사용하지 않는 언어의 기능

- reduce 함수는 파이썬 3 부터 **내장 함수에서 제외**

  가독성을 위해 for문을 사용

[reduce(lambda x, y: x + y, a) for문으로 표현]

Python Shell

```
>>> a = [1, 2, 3, 4, 5]

>>> x = a[0]

>>> for i in range(len(a) - 1):
        x += a[i + 1]
>>> x
15
```

Decorator

- Decorator는 하나의 함수를 취해서 또 다른 함수를 반환하는 함수
- 표기법으로는 @(심볼)
- 주로 log를 남기거나 유저의 로그인 상태를 확인하여 로그인 페이지로 redirect, 프로그램 성능을 위한 테스트에서 사용

Decorator

[decorator 예제]

```
def decorator_function(original_function):
    def wrapper_function():
        print('{} 함수가 호출되기전 입니다.'.format(original_function.__name__))
        return original_function()
    return wrapper_function


def display_1():
    print('display_1 함수가 실행됐습니다.')


def display_2():
    print('display_2 함수가 실행됐습니다.')

display_1 = decorator_function(display_1)  #1
display_2 = decorator_function(display_2)  #2

display_1()
print()
display_2()
```

http://schoolofweb.net/blog/posts/파이썬-데코레이터-decorator/

Python Shell

```
display_1 함수가 호출되기전 입니다.
display_1 함수가 실행됐습니다.

display_2 함수가 호출되기전 입니다.
display_2 함수가 실행됐습니다.
```

Decorator

[decorator @(심볼) 사용 예제]

```
def decorator_function(original_function):
    def wrapper_function():
        print('{} 함수가 호출되기전 입니다.'.format(original_function.__name__))
        return original_function()
    return wrapper_function

@decorator_function  #1
def display_1():
    print('display_1 함수가 실행됐습니다.')

@decorator_function  #2
def display_2():
    print('display_2 함수가 실행됐습니다.')

# display_1 = decorator_function(display_1)  #1
# display_2 = decorator_function(display_2)  #2

display_1()
print()
display_2()
```

http://schoolofweb.net/blog/posts/파이썬-데코레이터-decorator/

Python Shell

```
display_1 함수가 호출되기전 입니다.
display_1 함수가 실행됐습니다.

display_2 함수가 호출되기전 입니다.
display_2 함수가 실행됐습니다.
```

Decorator

➔ 장점

- 변형을 엄밀하게 명시
- 반복적인 코드를 제거
- 불변성을 유지하게 만드는 작업 수행

➔ 단점

- 함수의 인자, 반환 값에 대해 임의의 동작을 수행
- 결과적으로 놀라운 암묵적 행동
- 잘못된 Decorator 코드는 회복이 거의 불가능

Thread

- Thread는 2가지 이상의 일을 동시에 수행
- Python에서의 Thread는 주로 Threading module로 사용
- 내장된 타입의 원자성에 의존하면 안됨

  ex) Dictionary와 같은 Python의 내장된 타입

- Queue module의 Queue 데이터 타입을 사용 권장

강력한 기능들

- Python은 매우 유연한 언어로서 즉각적인 컴파일, 동적 상속,
  import hacks 등 많은 강력한 기능들을 제공

- 하지만 이 기능들이 반드시 필요한 것은 아님
- 읽고 이해 또는 디버그 하는데 어렵기 때문에 되도록 피하는 것을 권장

Modern Python : Python 3 (from, __future__, import)

- Python 3 버전을 사용하지 않아도 모든 코드는 버전 호환이 되도록 작성

  (Python 3 에 따라 테스트 권장)

- 작성된 코드는 명확하고 모든 의존성이 Python 3 에서 실행하기가 더 쉬움

- 코드를 재사용하므로 호환 문제로 지원하지 않는 기능들을

  import 하는 경우도 생김

Modern Python : Python 3 (from, __future__, import)

- 이러한 형태로 다음 사항이 코드에 포함되어야 한다고 명시하며

  호환되도록 업데이트 해야 함

[example]

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
```

감사합니다