Function

Created by 이 윤 준 (yoonjoon.lee@gmail.com)

May, 2019

Functions

Functions are the primary and most important method of code organization and reuse in Python. declared using the def keyword and returned from using the return keyword:

```
def my_function(x, y, z=1.5):
    if z > 1:
        return z * (x + y)
    else:
        return z / (x + y)
```

Some number of positional arguments and some number of keyword arguments. Keyword arguments are most commonly used to specify default values or optional arguments. The keyword arguments must follow the positional arguments

```
my_function(5, 6, z=0.7)
my_function(3.14, 7, 3.5)
```

Namespaces, Scope, and Local Functions

Functions can access variables in two different scopes: global and local. An alternate and more descriptive name describing a variable scope in Python is a namespace.

```
>>> a = [1]
>>> def func():
...     for i in range(5):
...         a.append(i)
...
>>> print(a)
```

```
>>> def func():
...     a = []
...     for i in range(5):
...         a.append(i)
...
>>> print(a)
```

```
>>> a = None
>>>
>>> def bind_a_variable():
...     global a
...     a = []
...
>>> bind_a_variable()
>>> print(a)
```

Returning Multiple Values

```
def f():
    a = 5
    b = 6
    c = 7
    return a, b, c

a, b, c = f()
```

Functions Are Objects

```
>>> states = [' Alabama ', 'Georgia!', 'Georgia',
...           'georgia', 'FlOrIda, 'south carolina##',
...           'West virginia?']
>>>
>>> import re # Regular expression module
>>>
>>> def clean_strings(strings):
...     result = []
...     for value in strings:
...         value = value.strip()
...         value = re.sub('[!#?]', '', value) # remove pu
...         value = value.title()
...         result.append(value)
...
...     return result
>>>
>>> clean_strings(states)
['Alabama', 'Georgia', 'Georgia', 'Georgia', 'Florida',
'South Carolina', 'West Virginia']
```

```
>>> def remove_punctuation(value):
...     return re.sub('[!#?]', '', value)
>>> clean_ops = [str.strip, remove_punctuation, str.title
>>>
>>> def clean_strings(strings, ops):
...     result = []
...     for value in strings:
...         for function in ops:
...             value = function(value)
...         result.append(value)
...     return result
>>> clean_strings(states)
['Alabama', 'Georgia', 'Georgia', 'Georgia', 'Florida',
'South Carolina', 'West Virginia']
```

```
>>> list(map(remove_punctuation, states))
['Alabama', 'Georgia', 'Georgia', 'Georgia', 'Florida',
'South Carolina', 'West Virginia']
```

Anonymous (lambda) Functions

```
>>> def apply_to_list(some_list, f):
...     return [f(x) for x in some_list]
...
>>> ints = [4, 0, 1, 5, 6]
>>> apply_to_list(ints, lambda x: x * 2)
[8, 0, 2, 10, 12]
```

```
>>> strings = ['foo', 'card', 'bar', 'aaaa', 'abab']
>>> strings.sort(key=lambda x: len(set(list(x))))
```

Closures: Functions that Return Functions

A closure is any dynamically-generated function returned by another function. The returned function has access to the variables in the local namespace where it was created.

```
>>> def make_watcher():
...     have_seen = {}
...
...     def has_been_seen(x):
...         if x in have_seen:
...             return True
...         else:
...             have_seen[x] = True
...             return False
...
...     return has_been_seen
...
>>> watcher = make_watcher()
>>>
>>> vals = [5, 6, 1, 5, 1, 6, 3, 5]
>>> [watcher(x) for x in vals]
[False, False, False, True, True, True, False, True]
```

Extended Call Syntax with *args, **kwargs

The internal function receives a tuple args and dict kwargs and internally does the equivalent of:

```
func(a, b, c, d=some, e=value)

    a, b, c = args
    d = kwargs.get('d', d_default_value)
    e = kwargs.get('e', e_default_value)
```

```
>>> def say_hello_then_call_f(f, *args, **kwargs):
...     print('args is', args)
...     print('kwargs is', kwargs)
...     print("Hello! Now I'm going to call %s" % f)
...     return f(*args, **kwargs)
...
>>> def g(x, y, z=1):
...     return (x + y) / z
...
>>> say_hello_then_call_f(g, 1, 2, z=5.)
('args is', (1, 2))
('kwargs is', {'z': 5.0})
Hello! Now I'm going to call <function g at 0x1029af668>
0.6
```

Currying: Partial Argument Application

Currying is a fun computer science term which means deriving new functions from existing ones by partial argument application. The second argument to add_numbers is said to be curried. The built-in functools module can simplify this process using the partial function:

```
>>> def add_numbers(x, y):
...     return x + y
...
>>> from functools import partial
>>> add_five = partial(add_numbers, 5)
>>>
>>> add_five(3)
8
```

```
>>> add_five = lambda y: add_numbers(5, y)
>>> add_five(3)
8
```