Functions in Python

Vivek K. S., Deepak G.

Information Systems Decision Sciences (ISDS)
MUMA College of Business
University of South Florida
Tampa, Florida

2017

Functions in Python

- Functions are the best way to write reusable code.
- Simply put, its a block of code that may/may not take input(s) and performs a specific operation over it/them.
- A function may or may not return a value.
- Functions improve re-usability and readability.
- Functions have a name and that's how they are referred to by the calling object in the program.
- Functions need to be defined before being called.
- In Python, we define a function using the "def" keyword.

A simple Function in Python

The following function shows a typical Python function.

```python
def add(x,y):
        return(x+y)
```

- Its defined using "def".
- It has a name "add" and takes in two arguments x and y.
- Notice that x and y are not limited by any data type and are mere references that point to an object in memory.
- The function returns the sum of the two variables.
- Notice that there is no need to use curly braces or enclosing mechanism to enclose the function body.
- Instead, Python uses indentation to mark a block of code.

Continued..

In the previous example, the function returned a value. In some situations, there could be functions that don't return anything at all.

```python
def print_something(message):
        print(message)
```

The following function is a special case where there is no return, neither does it do anything. We use a special keyword 'pass' in Python to accomplish this.

```python
def do_nothing():
        pass
```

Enclosing Code Blocks in Function

A real function that does an actual operation looks like this:

```python
import datetime
def meal_time(time):
    print(time)
    if time<=10 and time>= 8:
        print('Time for Breakfast')
    if time<=14 and time>= 12:
        print('Time for Lunch')
    if time<=20 and time>= 18:
        print('Time for Dinner')

hour = datetime.datetime.now().hour
meal_time(hour)
```

Positional Arguments

- Python handles function arguments in an exceptionally flexible manner compared to other languages.
- Having Positional arguments is the most common approach to passing parameters to the function in order. Example,

```python
def language_rating(lang, rating):
        print('I rate {} {}/10.'.\
        format(lang,rating))
```

- The problem with Positional Arguments is that we need to remember the order of the arguments while passing values to them.

Keyword Arguments

To avoid the confusion of Positional arguments, we have keyword arguments.

- Here, we can specify arguments by the names of their corresponding parameters.
- This avoids the confusion of ordering. Example,

```python
def language_rating(lang, rating):
        print('I rate {} {}/10.'.\
        format(lang,rating))

language_rating(rating=10,lang='Python')
```

Mixing Positional and Keyword Arguments

Keyword and Positional arguments could be mixed as well. But it should be noted that Positional arguments should always be specified first.

```python
def language_rating(lang, rating):
        print('I rate {} {}/10.'.\
        format(lang,rating))

language_rating('Python',rating=10)
```

Specifying Default Parameter Values

Arguments can be set with default values as follows:

```python
def language_rating(rating, lang='Python'):
        print('I rate {} {}/10.'.\
        format(lang,rating))


language_rating(10)
```

Note that the default arguments should come in the end. Passing values to the default argument will replace the default value as follows:

```python
def language_rating(rating, lang='Python'):
        print('I rate {} {}/10.'.\
        format(lang,rating))


language_rating(10,'Java')
```

Positional Arguments with *

The asterisk can be used to accept variable number of arguments.

- The asterisk groups a variable number of positional arguments into a tuple of parameter values.

```
def print_args(*args):
    print('Positional argument tuple:', args)

print_args('hello','world',5,6.0,True)

# Calling the method with no arguments:
print_args()
Positional argument tuple: ()
```

Keyword Arguments with **

We could use "**" to gather keyword arguments into a dictionary where the argument names are the keys and values are the corresponding dictionary values:

```python
def print_kwargs(**kwargs):
    print('Keyword arguments:', kwargs)
print_kwargs(NYC='Albany',
 California='Sacramento')
Keyword arguments: {'NYC': 'Albany',
 'California': 'Sacramento'}
```

Passing Functions as Arguments

We can pass Functions as arguments to other functions as functions are objects in Python.

```python
def add(x,y):
        return(x+y)
def arithmetic(func,x,y):
        return(func(x,y))
arithmetic(add,5,10)
15
```

Contd..

Another interesting example.

```python
def edit_text(sentence, func):
    words = sentence.split()
    final_sentence = " ".join(func(word)\
     for word in words)
    return(final_sentence.strip())
def capitalize_sentence(word):
    return word.capitalize()


sentence = 'python was authored by
Guido van Rossum'
edit_text(sentence, capitalize_sentence)
'Python Was Authored By Guido Van Rossum'
```

Lambda Functions

Lambda functions are used to create anonymous "on the go" functions in place of little function definitions that might not be reused later and is an overkill.

Lambda functions can be expressed in a single statement.

```
my_lambda = lambda x : x*2

print(my_lambda(10))
```

Complete Coding Exercise

Complete Coding Exercise is available at –
https://github.com/vivek14632/Python-Workshop/tree/
master/Introducing%20Python/Chapter%204

Summary

- We learned functions in Python and how they help us to enclose blocks of code that perform a defined functionality, thereby improving reusability and readability.
- We learned how to modularize our program.
- We learned how important functions are important to the Object-oriented approach of coding.
- We learned about arguments and parameters and the different type of arguments.
- We experimented with the different types of argument to tackle different requirements.