PythonFunctions

February 15, 2018

0.1 Functions

Functions are the primary and most important method of code organization and reuse in Python. As a rule of thumb, if you anticipate needing to repeat the same or very similar code more than once, it may be worth writing a reusable function. Functions can also help make your code more readable by giving a name to a group of Python statements

Defining Functions A function is defined in Python by the following format:

```
In [ ]: def functionname(arg1, arg2, ...):
            statement1
            statement2
            ...
            return values
```

A function can 'return' a value, You can return multiple values

```
In [1]: def square(x):
            return x*x
        square(5)

        def powers (x):
            return x*x, x**3, x**4
        a,b,c = powers(10)
        print(a,b,c)
```

```
100 1000 10000
```

Default Argument Values If any of the formal parameters in the function definition are declared with the format "arg = value," then you will have the option of not specifying a value for those arguments when calling the function. If you do not specify a value, then that parameter will have the default value given when the function executes.

```
In [2]: def my_function(x, y, z=1.5):
            if z > 1:
                return z * (x + y)
            else:
                return z / (x + y)
        my_function(1,2)
        my_function(1,2,3)
```

```
Out[2]: 9
```

0.1.1 By Value and by Reference

Objects passed as arguments to functions are passed by reference; they are not being copied around. Thus, passing a large list as an argument does not involve copying all its members to a new location in memory. Note that even integers are objects. However, the distinction of by value and by reference present in some other programming languages often serves to distinguish whether the passed arguments can be actually changed by the called function and whether the calling function can see the changes.

Passed objects of mutable types such as lists and dictionaries can be changed by the called function and the changes are visible to the calling function. Passed objects of immutable types such as integers and strings cannot be changed by the called function; the calling function can be certain that the called function will not change them.

```
In [ ]: def appendItem(ilist, item):
          ilist.append(item) # Modifies ilist in a way visible to the caller

        def replaceItems(ilist, newcontentlist):
          del ilist[:]                 # Modification visible to the caller
          ilist.extend(newcontentlist) # Modification visible to the caller
          ilist = [5, 6] # No outside effect; lets the local ilist point to a new list object,
                         # losing the reference to the list object passed as an argument
        def clearSet(iset):
          iset.clear()

        def tryToTouchAnInteger(iint):
          iint += 1 # No outside effect; lets the local iint to point to a new int object,
                    # losing the reference to the int object passed as an argument
          print(" {0:d} iint inside:".format(iint)) # 4 if iint was 3 on function entry

        list1 = [1, 2]
        appendItem(list1, 3)
        print(list1) # [1, 2, 3]
        replaceItems(list1, [3, 4])
        print(list1) # [3, 4]
        set1 = set([1, 2])
        clearSet(set1 )
        print(set1) # set([])
        int1 = 3
        tryToTouchAnInteger(int1)
        print(int1) # 3
```

0.1.2 Preventing Argument Change

An argument cannot be declared to be constant, not to be changed by the called function. If an argument is of an immutable type, it cannot be changed anyway, but if it is of a mutable type such as list, the calling function is at the mercy of the called function. Thus, if the calling function wants to make sure a passed list does not get changed, it has to pass a copy of the list.

```
In [ ]: def evilGetLength(ilist):
            length = len(ilist)
            del ilist[:] # Muhaha: clear the list
            return length

        list1 = [1, 2]
        print(evilGetLength(list1[:])) # Pass a copy of list1
        print(list1)
        list1 = [1, 2]
        print(evilGetLength(list1)) # list1 gets cleared
        print(list1)
        list1 = []
```

0.1.3 Anonymous (Lambda) Functions

Python has support for so-called anonymous or lambda functions, which are a way of writing functions consisting of a single statement, the result of which is the return value. They are defined with the lambda keyword, which has no meaning other than "we are declaring an anonymous function":

```
In [3]: def short_function(x):
            return x * 2
        equiv_anon = lambda x: x * 2

        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)
```

```
Out[3]: [8, 0, 2, 10, 12]
```

0.1.4 Functions Are Objects

Since Python functions are objects, they can be passed into functions too

```
In [4]: def myfun(x, fob):
            return[fun(x) for fun in fob]
        import math
        fobs=[ math.sin, math.exp]
        myfun(1, fobs)
```

```
Out[4]: [0.8414709848078965, 2.718281828459045]
```

0.1.5 Currying: Partial Argument Application

Currying is computer science jargon (named after the mathematician Haskell Curry) that means deriving new functions from existing ones by partial argument application. For example, suppose we had a trivial function that adds two numbers together

```
In [ ]: def add_numbers(x, y):
            return x + y
```

Using this function, we could derive a new function of one variable, add_five, that adds 5 to its argument

```
In [ ]: add_five = lambda y: add_numbers(5, y)
```

The second argument to add_numbers is said to be curried. There's nothing very fancy here, as all we've really done is define a new function that calls an existing function. The built-in functools module can simplify this process using the partial function

```
In [ ]: from functools import partial
        add_five = partial(add_numbers, 5)
        add_five(10)
```

0.1.6 Generators

Having a consistent way to iterate over sequences, like objects in a list or lines in a file, is an important Python feature. This is accomplished by means of the iterator protocol, a generic way to make objects iterable. For example, iterating over a dict yields the dict keys:

```
In [5]: some_dict = {'a': 1, 'b': 2, 'c': 3}
        for key in some_dict:
            print(key)
```

```
a
b
c
```

When you write for key in some_dict, the Python interpreter first attempts to create an iterator out of some_dict:

```
In [6]: dict_iterator = iter(some_dict)
        print(type(dict_iterator))
```

```
<class 'dict_keyiterator'>
```

An iterator is any object that will yield objects to the Python interpreter when used in a context like a for loop. Most methods expecting a list or list-like object will also accept any iterable object. This includes built-in methods such as min, max, and sum, and type constructors like list and tuple

```
In [7]: list(dict_iterator)
```

```
Out[7]: ['a', 'b', 'c']
```

A generator is a concise way to construct a new iterable object. Whereas normal functions execute and return a single result at a time, generators return a sequence of multiple results lazily, pausing after each one until the next one is requested. To create a generator, use the yield keyword instead of return in a function:

```
In [10]: def squares(n=10):
             print('Generating squares from 1 to {0}'.format(n ** 2))
             for i in range(1, n + 1):
                 yield i ** 2
```

When you actually call the generator, no code is immediately executed

```
In [11]: gen = squares()
         gen

Out[11]: <generator object squares at 0x0000003BD8104CA8>
```

It is not until you request elements from the generator that it begins executing its code:

```
In [12]: for x in gen:
             print(x, end=' ')

Generating squares from 1 to 100
1 4 9 16 25 36 49 64 81 100
```

0.1.7 Generator expresssions

Another even more concise way to make a generator is by using a generator expression. This is a generator analogue to list, dict, and set comprehensions; to create one, enclose what would otherwise be a list comprehension within parentheses instead of brackets:

```
In [13]: gen = (x ** 2 for x in range(100))
         gen

Out[13]: <generator object <genexpr> at 0x0000003BD81045C8>
```

This is completely equivalent to the following more verbose generator:

```
In [14]: def _make_gen():
             for x in range(100):
                 yield x ** 2
         gen = _make_gen()
```

Generator expressions can be used instead of list comprehensions as function arguments in many cases

```
In [16]: sum(x ** 2 for x in range(100))
         dict((i, i **2) for i in range(5))

Out[16]: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

0.1.8 itertools module

The standard library itertools module has a collection of generators for many common data algorithms. For example, groupby takes any sequence and a function, grouping consecutive elements in the sequence by return value of the function. Here's an example:

```
In [17]: import itertools
         first_letter = lambda x: x[0]
         names = ['Alan', 'Adam', 'Wes', 'Will', 'Albert', 'Steven']
         for letter, names in itertools.groupby(names, first_letter):
             print(letter, list(names)) # names is a generator
```

```
A ['Alan', 'Adam']
W ['Wes', 'Will']
A ['Albert']
S ['Steven']
```

0.2 Python Files Operations

we will read from and write into files. To do so, we need to maintain some steps. Those are

1. Open a file
2. Take input from that file / Write output to that file
3. Close the file

We will also learn some useful operations such as copy file and delete file.

Python Open File

According to the previous discussion, the first step we have to perform in Python File Operation is opening that file. You can open a file by using open() function. This function take two arguments. The first one is file address and the other one is opening mode. There are some mode to open a file. Most common of them are listed below:

1. 'r' : This mode indicate that file will be open for reading only
2. 'w' : This mode indicate that file will be open for writing only. If file containing containing that name does not exists, it will create a new one
3. 'a' : This mode indicate that the output of that program will be append to the previous output of that file
4. 'r+' : This mode indicate that file will be open for both reading and writing

Additionally, for Windows operating system you can append 'b' for accessing the file in binary. As Windows makes difference between binary file and text file.

Suppose, we place a text file name 'file.txt' in the same directory where our code is placed. Now we want to open that file. However, the open(filename, mode) function returns a file object. With that file object you can proceed your further operation.

```
In [ ]: #directory:   /home/imtiaz/code.py
        text_file = open('file.txt','r')

        #Another method using full location
        text_file2 = open('/home/imtiaz/file.txt','r')
        print('First Method')
        print(text_file)

        print('Second Method')
        print(text_file2)
```

0.2.1 Python Read File, Python Write File

There are some methods to read from and write to file. The following list are the common function for read and write in python. Note that, to perform read operation you need to open that file in read mode and for writing into that file, you need to open that in write mode. If you open a file in write mode, the previous data stored into that fill will be erased.

1. read() : This function reads the entire file and returns a string
2. readline() : This function reads lines from that file and returns as a string. It fetch the line n, if it is been called nth time.
3. readlines() : This function returns a list where each element is single line of that file.
4. write() : This function writes a fixed sequence of characters to a file.
5. writelines() : This function writes a list of string.
6. append() : This function append string to the file instead of overwriting the file.

The following code will guide you to read from file using Python File Operation. We take 'file.txt' as our input file.

```
In [ ]: #open the file
        text_file = open('/Users/pankaj/abc.txt','r')

        #get the list of line
        line_list = text_file.readlines();

        #for each line from the list, print the line
        for line in line_list:
            print(line)

        text_file.close() #don't forget to close the file
```

```
In [ ]: #open the file
        text_file = open('file.txt','w')

        #initialize an empty list
```

```
word_list= []

#iterate 4 times
for i in range (1, 5):
    print("Please enter data: ")
    line = input() #take input
    word_list.append(line) #append to the list


text_file.writelines(word_list) #write 4 words to the file

text_file.close() #dont forget to close the file
```