Functions in Python

Leigh J. Little SUNY Brockport November 29, 2021

Contents

1 Introduction

This document will discuss the how to write your own functions in Python.

2 Tuples

Python has three native list-type structures. We previously discussed lists. The two other two list-type structures are *tuples* and *dictionaries*. Before we can discuss functions, we need to briefly touch on tuples. A tuple is similar to a list, except the elements of a tuple are immutable (*i.e.*, they cannot be modified once assigned). A tuple is created using the () constructors:

```
In []: x = ('car',1,4.7)
In []: x
Out[]: ('car', 1, 4.7)
In []: x[0]
Out[]: 'car'
In []: x[1] = 17
Traceback (most recent call last):

  File "g:\temp/ipykernel_14692/3120150804.py", line 1, in <module>
    x[1]= 17

TypeError: 'tuple' object does not support item assignment
```

Like a list, the elements of a tuple can be anything. The first element of the tuple has index number 0. Note that if you attempt to change `x[1]`, an error is generated.

While the elements of a tuple cannot be modified once they are assigned, it is possible to append elements to a tuple. The syntax for this is different than for appending to a list

```
In[]:  x = x + (5.4,)
In[]:  x
Out[]:  ('car', 1, 4.7, 5.4)
In[]:  x = x + ('New Value',)
In []: x
Out[]:  ('car', 1, 4.7, 5.4, 'New Value')
```

2.1 Convert Between Lists and Tuples

It is occasionally necessary to convert a tuple to a list or vice-versa. There are functions that will handle this conversion. For example, if `x` is the list `[1,2,3,4]`, this can be converted to a tuple using the `tuple` function.

```
In[]: x = [1,2,3,4]
In[]: x
Out[]: [1, 2, 3, 4]
In[]:  y = tuple(x)
In[]; y
Out[]: (1, 2, 3, 4)
```

Similarly, you can use the `list` function to convert a tuple to a list

```
In[]; y
Out[]: (1, 2, 3, 4)
In[]: z = list(y)
In[]: z
Out[]: [1, 2, 3, 4]
```

3 Functions

You can create your own functions in Python. This is best exhibited with an example.

```
import math as m              # Any import statements at the top will be available
                              # to all functions in this file.

def testfun(x):               # Function name is testfun and takes in 1
                              # argument.  This argument can be anything
                              # (a scalar, list, etc).  Note the : at the end.
    z = x**2 + 3*m.sin(x)     # Note the indenting
    return z                  # The return statement goes back to the calling
                              # routine
```

The above code is saved in a file called `myfun.py` Some notes about functions:

1) The name of the file containing the function can be anything (as long as it does not conflict with an existing file in your working directory).

2) You can have as many functions in the file as you like.

3) The keyword `def` indicates the start of a new function definition. Note the (`:`) at the end of the `def` line.

4) The entire function body needs to indented (in addition to the usual indenting requirements).

5) The `return` keyword tells the function to return to the calling function.

6) The variable `z` is the return variable. This is the output of the function.

7) A function can have numerous `return` statements.

6) A return statement can be empty (*i.e.*, you do not have to return a variable).

9) There is no keyword that ends the function. The end of a function is implied by the indenting.

10) Because the function uses the sine function, the math module needs to be pulled in at the top of the file. Any other functions in this file that use the math module will also be able to use it.

To use the function, create a driver script, then import the file/s containing the function into the workspace (all of the files should be in the same directory on your computer).

```
import myfun as my
x = 1.5
y = my.testfun(x)
print('x, y = ',x,y)
```

```
Output:
    x, y =  1.5 6.75
```

Note that the function we wrote (`testfun`) needs to be preceded by the `my` prefix.

In the event you need to pull several modules into the workspace, make sure they have unique prefix names. For example, the sequence of statements below will 'work' in the sense that it will not flag an error, but it is a really bad idea:

```
import myfun as m
import math as m
import random as m
```

4 Function with Multiple Inputs/outputs

Functions can have any number of inputs and/or outputs (including none). Multiple inputs are handled in the usual way. If you need multiple outputs, you add them to the return statement

```
def testfun2(x,y):
    a = x**2 + y**2 + x - y
    b = x**2 - y**2 - x - y
    return a,b
```

When you call the function, you can have the output returned in one of two ways.

4.1 Return the Outputs as Individual Variables

The first option when calling a function with multiple outputs is to have each output returned in its own variable.

```
import myfun as my
x = 1.5
y = -5.6
v0,v1 = my.testfun2(x,y)
print('v0, v1 = ',v0,v1)
```

```
Output:
    v0, v1 =  40.71 -22.009999999999998
```

In the above example, the two output values are returned in two separate variables. (`v0` will contain the value of `a` and `v1` will contain the value of `b`).

4.2 Return the Outputs in a Tuple

An alternative to returning the variables individually is to return them in a tuple:

```
import myfun as my
x = 1.5
y = -5.6
t = my.testfun2(x,y)
print('t = ',t)
print('t[0] = ',t[0])
print('t[1] = ',t[1])
```

```
Output:
```

```
t =  (40.71, -22.009999999999998)
t[0] =  40.71
t[1] =  -22.009999999999998
```

Here, both outputs are returned in the single tuple (`t`).

Which of these two methods you use depends on what you intend to do with the output values. Tuples are handy if you have numerous values that you would like to always be grouped together. For example, if the output were a set of polar coordinates, a tuple would make sense because you would want to keep them together.

Note that no matter which method you choose to return the outputs from a function, the function itself remains the same.