Function_*args_**kwargs_lambda

February 23, 2017

1 Functions

There are a number of bits of 'advanced' Python syntax that relate directly to how we use and define functions. Let's dive in first by looking at how we deal with complex arguments. First, the arguments list of a function defintion take the form we're used to:

function(arg1, arg2)

But one of Python's strengths is in its flexibility in arguments that may be more complex (or at least less expressive) in other languages. We can easily define a function that takes any number of arguments, such as a function that prints each of its inputs, no matter how many you give it. Let's see this below:

```
In [1]: def deg(x):
            return x + "is awesome"

        def my_func(*args):
            for arg in args:
                print(arg)

        my_func('one', 'two', 'three')
        print('-----')
        my_func('or just one')
```

```
one
two
three
-----
or just one
```

This *args syntax allows us to take a flexible number of arguments, which the function basically sees as a list. Conveniently, we can use this to provide some optional arguments and some required ones. In this case, the *args must come last.

```
In [2]: def my_func(arg1, arg2, *args):
            print(arg1, ' and ', arg2)
            print('But you also gave ', len(args), ' more arguments!')
```

```
        my_func('one', 'two')
        print('-----')
        my_func('or one', 'two', 'and three', 'maybe a fourth')
```

```
one  and  two
But you also gave  0  more arguments!
-----
or one  and  two
But you also gave  2  more arguments!
```

There's an inverse operation to this, however, called argument unpacking. Say we have a function that takes three arguments, and we have a list of three items. We can use this syntax to intelligently feed each element of the list (or tuple, etc) to the function, as shown below:

```
In [3]: def my_func(arg1, arg2, arg3):
            return arg1 + arg2 + arg3

        my_vals = [1, 2, 3]
        my_func(*my_vals)
```

```
Out[3]: 6
```

The syntax, as expected, is basically the same as when we provide optional arguments. This is convenient for a number of reasons, especially when you feed data into a variable-argument function.

We can use `zip()` as its own inverse function, simply by using this argument unpacking syntax.

```
In [4]: l1, l2 = [1,2,3,4], [4,5,6,7]
        zipped = list(zip(l1, l2))
        print(zipped)

        unzipped = list(zip(*zipped))
        print(unzipped)
```

```
[(1, 4), (2, 5), (3, 6), (4, 7)]
[(1, 2, 3, 4), (4, 5, 6, 7)]
```

Similar to how we can provide optional arguments using this `*args` syntax, we can provide optional arguments as key-value pairs using a dict-like syntax. Let's define a function that uses all of these together, and look at some clever ways to take advantage of it.

```
In [5]: def complex_func(arg1, arg2, *args, **kwargs):
            print(arg1, arg2)
            print('variable length = ', args)
            print('keywords = ', kwargs)

        complex_func(1, 2, 4, 5, 6, 7, my_var='x', other_var='y')
```

```
1 2
variable length =  (4, 5, 6, 7)
keywords =  {'my_var': 'x', 'other_var': 'y'}
```

Notice that using `**kwargs` provides a dict-like interface to access named parameters of our function. These are inherently optional, and we can iterate over them just as we would in a dict. We can provide default values for keyword arguments in a function definition as well:

```
In [6]: def func_defaults(key=5, value=1):
            print(key, ' = ', value)

        func_defaults()
        print('----')
        func_defaults(1, 2)
        print('----')
        func_defaults(key=10, value=5)
        print('----')
        func_defaults(value=5, key=10)
```

```
5  =  1
----
1  =  2
----
10  =  5
----
10  =  5
```

And as expected, argument unpacking works on keyword arguments just the same:

```
In [7]: d = {'key': 5, 'value': 15}
        func_defaults(**d) # key=5, value=15
```

```
5  =  15
```

Python supports a number of more useful features related to functions. Below we'll look at a few examples. The first note to make is that Python has a concept of first-class function (or higher-order functions) - that is, functions are values just like numbers are, and functions can accept functions as arguments or return functions just as they can values.

As a result, there are a number of times we need a quick little function defined without necessarily giving it a name. This concept is known as 'anonymous functions', but Python implements the `lambda` operator to support this. The `lamba` syntax basically provides a quick way to define a 0-N argument anonymous function in-line. Let's use this `map` function for an example: `map()` takes two arguments - a function to apply, and an iterable object on which to apply it.

```
In [8]: def multiply2(x):
            return x * 2
```

```
        data = [1,2,4,5,6]
        list(map(multiply2, data))
```

```
Out[8]: [2, 4, 8, 10, 12]
```

Using a lambda expression, we can encode this simple behavior in one line:

```
In [9]: s = ["this is my string", "this is my other string"]
        list(map(lambda x: x[:7], s))
```

```
Out[9]: ['this is', 'this is']
```

Lambda expressions are very useful, but their use can be a little tricky. Lambda expressions consider their scope, so they can lead to some complicated scenarios, for better or worse. Thankfully, you can't assign values from within a lambda expression, so you can't do too much harm, but you have to be mindful of values in scope:

```
In [10]: some_value = 10

         # pull in a variable from our current scope
         list(map(lambda x: x + some_value, data))
```

```
Out[10]: [11, 12, 14, 15, 16]
```