Python Built-in Data Structures, Functions, and Files

Part 5

Functions

Part 1

- Functions are declared with the `def` keyword and returned from with the `return` keyword:

  -
    ```
    def my_function(x, y, z=1.5):
        if z > 1:
            return z * (x + y)
        else:
            return z / (x + y)
    ```
- There is no issue with having multiple `return` statements.
- If Python reaches the end of a function without encountering a `return` statement, `None` is returned automatically.

- Each function can have *positional* arguments and *keyword* arguments.
- Keyword arguments are most commonly used to specify default values or optional arguments.
- In the preceding function, `x` and `y` are positional arguments while `z` is a keyword argument.
- This means that the function can be called in any of these ways:

    -
      ```
      my_function(5, 6, z=0.7)
      my_function(3.14, 7, 3.5)
      my_function(10, 20)
      ```

- The main restriction on function arguments is that the keyword arguments *must* follow the positional arguments (if any).
- You can specify keyword arguments in any order; this frees you from having to remember which order the function arguments were specified in and only what their names are.

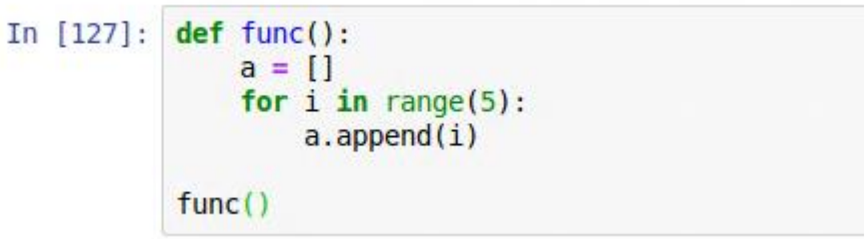
- It is possible to use keywords for passing positional arguments as well.
- In the preceding example, we could also have written:

  -
    ```
    my_function(x=5, y=6, z=7)
    my_function(y=6, x=5, z=7)
    ```
- In some cases this can help with readability.

Namespaces, Scope, and Local Functions

- Functions can access variables in two different scopes: *global* and *local*.
- An alternative and more descriptive name describing a variable scope in Python is a *namespace*.
- Any variables that are assigned within a function by default are assigned to the local namespace.
- The local namespace is created when the function is called and immediately populated by the function’s arguments.
- After the function is finished, the local namespace is destroyed

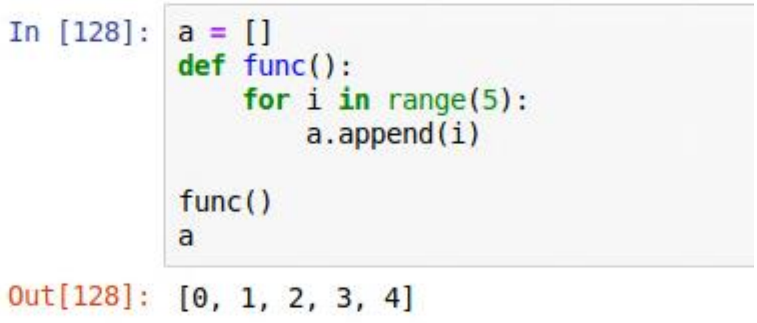
- Consider the following function:



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

func()
```

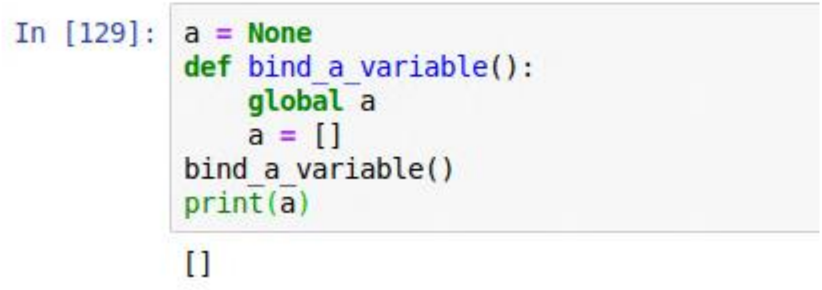
- When `func()` is called, the empty list `a` is created, five elements are appended, and then `a` is destroyed when the function exits.

- Suppose instead we had declared `a` as follows:



```
In [128]: a = []
          def func():
              for i in range(5):
                  a.append(i)

          func()
          a

Out[128]: [0, 1, 2, 3, 4]
```

- Assigning variables outside of the function’s scope is possible, but those variables must be declared as global via the `global` keyword:



```
In [129]: a = None
          def bind_a_variable():
              global a
              a = []
          bind_a_variable()
          print(a)

          []
```

Returning Multiple Values

- Here's an example:

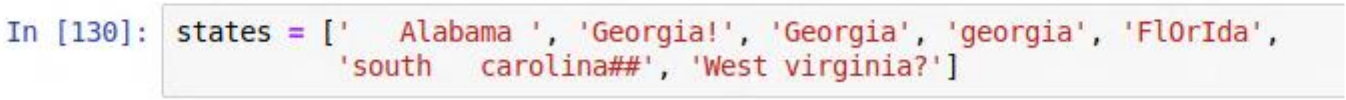
  -
    ```
    def f():
        a = 5
        b = 6
        c = 7
        return a, b, c
    a, b, c = f()
    ```
- In data analysis and other scientific applications, you may find yourself doing this often.
- What's happening here is that the function is actually just returning *one* object, namely a tuple, which is then being unpacked into the result variables.
- In the preceding example, we could have done this instead:
  - `return_value = f()`
- In this case, `return_value` would be a 3-tuple with the three returned variables.

- A potentially attractive alternative to returning multiple values like before might be to return a dict instead:

  -
    ```
    def f():
        a = 5
        b = 6
        c = 7
        return {'a' : a, 'b' : b, 'c' : c}
    ```
- This alternative technique can be useful depending on what you are trying to do.

Functions Are Objects

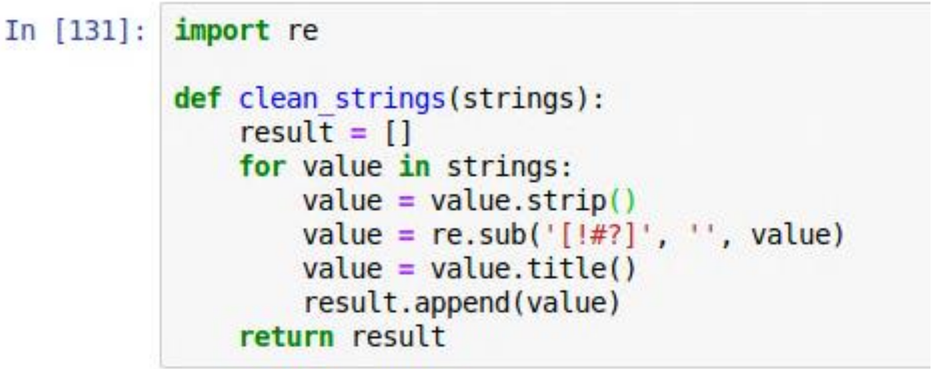
- Since Python functions are objects, many constructs can be easily expressed that are difficult to do in other languages.
- Suppose we were doing some data cleaning and needed to apply a bunch of transformations to the following list of strings:



```
In [130]: states = ['   Alabama ', 'Georgia!', 'Georgia', 'georgia', 'FlOrIda',
                    'south   carolina##', 'West virginia?']
```

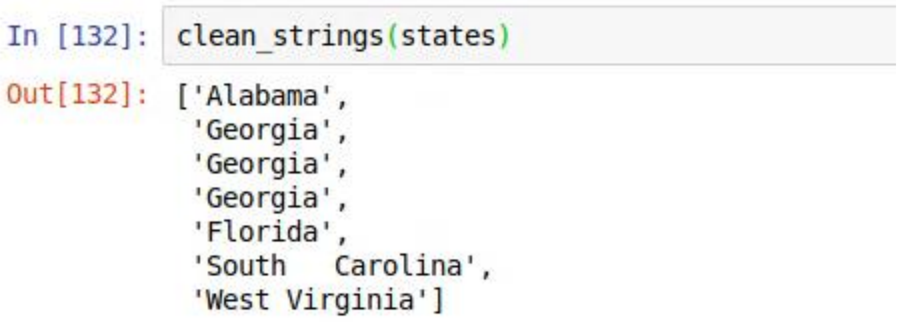
- Anyone who has ever worked with user-submitted survey data has seen messy results like these.
- Lots of things need to happen to make this list of strings uniform and ready for analysis: stripping whitespace, removing punctuation symbols, and standardizing on proper capitalization.

- One way to do this is to use built-in string methods along with the `re` standard library module for regular expressions:



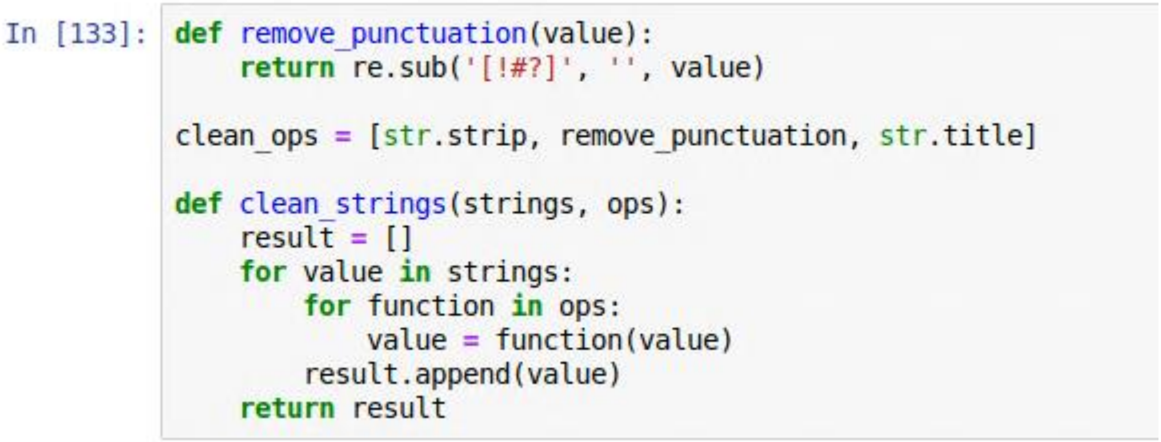
```
In [131]: import re

          def clean_strings(strings):
              result = []
              for value in strings:
                  value = value.strip()
                  value = re.sub('[!#?]', '', value)
                  value = value.title()
                  result.append(value)
              return result
```

- The result looks like this:



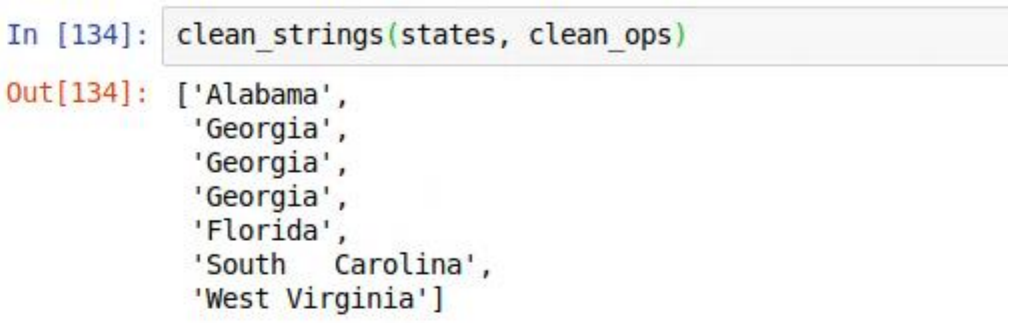
```
In [132]: clean_strings(states)

Out[132]: ['Alabama',
           'Georgia',
           'Georgia',
           'Georgia',
           'Florida',
           'South   Carolina',
           'West Virginia']
```

- An alternative approach that you may find useful is to make a list of the operations you want to apply to a particular set of strings:



```python
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
```

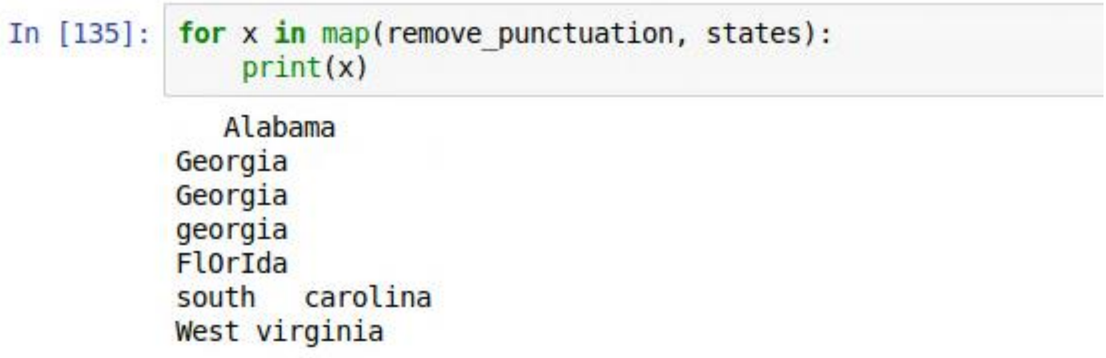
- Then we have the following:



```
In [134]: clean_strings(states, clean_ops)

Out[134]: ['Alabama',
           'Georgia',
           'Georgia',
           'Georgia',
           'Florida',
           'South   Carolina',
           'West Virginia']
```

- A more *functional* pattern like this enables you to easily modify how the strings are transformed at a very high level.
- The `clean_strings` function is also now more reusable and generic.

- You can use functions as arguments to other functions like the built-in `map` function, which applies a function to a sequence of some kind:



In [135]:
```
for x in map(remove_punctuation, states):
    print(x)
```

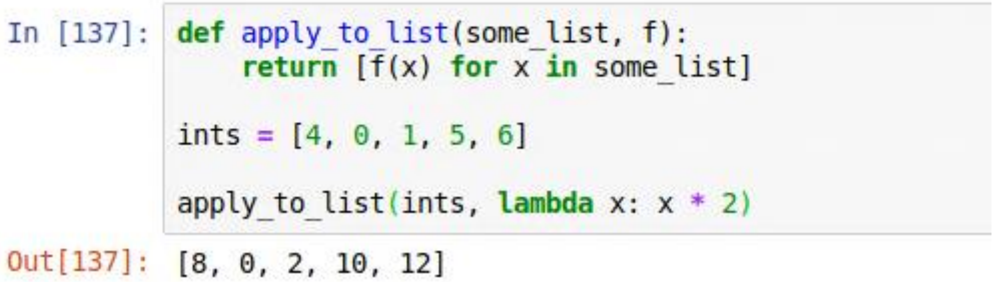
```
   Alabama
Georgia
Georgia
georgia
FlOrIda
south   carolina
West virginia
```

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”:

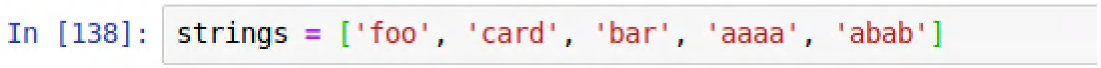
  -
    ```
    def short_function(x):
        return x * 2

    equiv_anon = lambda x: x * 2
    ```

- Lambda functions are especially convenient in data analysis because, as you’ll see, there are many cases where data transformation functions will take functions as arguments.
- It’s often less typing (and clearer) to pass a lambda function as opposed to writing a full-out function declaration or even assigning the lambda function to a local variable.
- For example, consider this silly example:



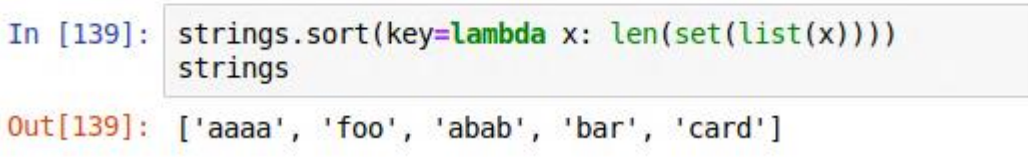
```
In [137]: def apply_to_list(some_list, f):
              return [f(x) for x in some_list]

          ints = [4, 0, 1, 5, 6]

          apply_to_list(ints, lambda x: x * 2)
Out[137]: [8, 0, 2, 10, 12]
```

- As another example, suppose you wanted to sort a collection of strings by the number of distinct letters in each string:



```
In [138]: strings = ['foo', 'card', 'bar', 'aaaa', 'abab']
```

- Here we could pass a lambda function to the list’s `sort` method:



```
In [139]: strings.sort(key=lambda x: len(set(list(x))))
          strings

Out[139]: ['aaaa', 'foo', 'abab', 'bar', 'card']
```