FUNCTIONS

Shannon Turner

Twitter: [@svt827]

Github: [http://github.com/shannonturner]

OBJECTIVE

- Review Lesson Three
- Learn what functions are
- Learn how and when and why we use functions
- Using everything we've learned so far: strings, slicing, conditionals, lists, loops, file handling, dictionaries

LIGHTNING REVIEW

- File handling lets us make Python open files (and retrieve the contents)
- The syntax is a bit tricky at first - lots of small parts we hadn't seen before
- We get the contents of the file as a string
- We can turn strings into lists
- We can loop over a list to do something with each line in this file

LIGHTNING REVIEW

- We opened text and comma-separated-values (CSV) files
- Text files: like Word, but without the formatting
- CSV files: like Excel, but without the formatting
- CSV files were a little trickier to get in the format we wanted, since we needed to use .split( ) multiple times.

LIGHTNING REVIEW

- Dictionaries are another way of storing information
- Think of it like a contact list
- Dictionaries have a key and a value
- If you know the key, you can see the value
- If you know my name, you can see my phone #

LIGHTNING REVIEW

- Dictionary **keys** must be a string, but dictionary **values** can be anything! Strings, lists, other dictionaries, anything!
- Think of all of the problems we'd run into if we created a contact list using lists.
  - How do you keep information that belongs together, stuck together?
  - This is why we use dictionaries.

FUNCTIONS: WE'VE SEEN THIS BEFORE

```
bread = raw_input("How many slices of
bread do you have? ")

bread = int(bread)

print len(speech)

for number in range(10):
     print number
```

METHODS ARE FUNCTIONS, TOO

```
phone = phone.replace("-", "")
at_symbol = email_address.find('@')

attendees.append("Amanda")
attendees.pop("Erica")

contacts.get('Lizzie')
contacts.keys()
```

YEAH, BUT WHAT *ARE* FUNCTIONS?

Functions are:

- Reusable code
- They do something specific
- They're flexible
    - **`len()`** tells us the length of anything we put inside the parentheses
- Written by people like us

FUNCTIONS: DEFINE ONCE, CALL AS NEEDED

Functions are:

- Defined once (like setting a variable)
- Then, the function waits to be used until it's called
- **`phone = '202-555-1234'`**
- phone is now a variable with a value, but nothing "happens" to it until you do something with it, like **`print phone`**
- Functions are the same way

HOW TO CALL (USE) A FUNCTION

Syntax:

return_value = function(parameter, other_parameter)

```
phone = phone.replace("-", "")
bread = int(bread)
print len(article)
attendees.append('Amanda')
for index, word in enumerate(article): …
```

FLEXIBILITY

Functions are flexible. When you change the parameters, you change what the function does!

```
phone = phone.replace("-", ".")
age = int(age)
print len(phone)
attendees.append('Reserved')
for index, person in enumerate(waitlist):
```

PARAMETERS / ARGUMENTS

Functions are powerful, useful, and reusable because they're so flexible.

Parameters (arguments) are the key to their flexibility — they control the "how" or "what" the function does.

```
for person in attendees:
    contacts.get(person, 'No info')
```

DEFINING OUR FUNCTIONS

Syntax to create a function:

```
def something(parameters_go_here):

    # your code goes here

    return
```

WHAT DOES THIS FUNCTION DO?

Head over to

https://github.com/shannonturner/python-lessons/blob/master/section_09_(functions)/remove_duplicates.py

EXERCISE: STARTING SMALL

Create a function that will return a string containing 'Hello, <name>!'

So when you call the function like this:

```
print greeting('Shannon')
```

This should happen in response:

```
Hello, Shannon!
```

EXERCISE: THROWBACK

Now turn your PB&J while loop program into a function!

So when you call the function like this:

```
print pbj_while(20)
```

This should happen in response:

```
I am making a sandwich! I have enough bread
for 8 more sandwiches.
```

… **`(and on and on)`**

SOME FUNCTIONS TAKE NO PARAMETERS

Most functions are made useful because of their parameters.

But some are useful solely because they're code you want to repeat often!

Head over to **https://github.com/shannonturner/python-lessons/blob/master/section_09_(functions)/dropdown_states.py**

LET'S CONVERT SOME CODE

From now on, most of the code you write, you'll want to write it as a function.

Then you can make it flexible (using parameters) and re-usable.

Let's convert some code from Lesson 3 into a function.

LET'S CONVERT SOME CODE

From Lesson 3:

```
with open("states.txt", "r") as
states_file:
    states = states_file.read()
```

LET'S CONVERT SOME CODE

First, let's wrap it in a function:

```
def textfile_to_string():

    with open("states.txt", "r") as states_file:
        states = states_file.read()

    return
```

LET'S CONVERT SOME CODE

Second, let's made it flexible using parameters:

```
def textfile_to_string(filename):

    with open(filename, "r") as
states_file:
        states = states_file.read()

    return
```

LET'S CONVERT SOME CODE

Third, let's remove references to 'states' to make it more general. This could be **any** text file, after all!

```
def textfile_to_string(filename):

    with open(filename, "r") as text_file:
        text = text_file.read()

    return
```

LET'S CONVERT SOME CODE

Last, the function should **return** the text from the file we opened.

```
def textfile_to_string(filename):

     with open(filename, "r") as
text_file:
          text = text_file.read()

     return text
```

NICE FUNCTION YOU'VE GOT THERE

Here's what it looks like:

```
def textfile_to_string(filename):

     with open(filename, "r") as text_file:

          text = text_file.read()

     return text
```

THE MOMENT OF TRUTH

Now let's call it!

```
contents = textfile_to_string('lessons.txt')
print contents
```

```
Lesson 1: Strings and Conditionals
Lesson 2: Lists and Loops
Lesson 3: Dictionaries and File Handling
```

DEFAULT PARAMETERS

Sometimes, it makes sense for parameters to have a default value.

```
def open_csvfile(filename, delimiter=','):
```

In this case, **`open_csvfile`** will assume that your values are separated by commas. But sometimes, you'll want your values separated by tabs instead.

DEFAULT PARAMETERS

Sometimes, it makes sense for parameters to have a default value.

```
def open_csvfile(filename, delimiter=','):
    … (assume the code is here) …
```

Here, I'm telling open_csvfile to use a tab instead of a comma.

```
open_csvfile('states.tsv', '\t')
```

DEFAULT PARAMETERS

Sometimes, it makes sense for parameters to have a default value.

```
def open_csvfile(filename, delimiter=','):
    … (assume the code is here) …
```

But if I'm using a CSV file, I don't even have to specify the delimiter, since it already defaults to the comma.

```
open_csvfile('states.csv')
```

DEFAULT PARAMETERS

If your function definition has default values, they must appear at the end!

```
# This will give an error!     D:    D:     D:
def open_csv(delimiter=',', filename):
```

```
# This is correct! :) :) :)
def scan_file(filename, deduplicate=True):
```

POSITION MATTERS!

In which order did you define your parameters?

Whatever order you defined them is the order you'll need to maintain when you call them.

This is why default parameters need to go at the end of a definition.

POSITION MATTERS!

Whatever order you defined your parameters is the order you'll need to maintain when you call them.

```
def upload_events(events_file, location):
    … (assume the code is here) …

upload_events(location, events_file) # Wrong!
```

POSITION MATTERS, EXCEPT WHEN IT DOESN'T

If you don't like this rule, break it! You can tell which parameter belongs to which variable during your function call.

```
upload_events(location, events_file) # Wrong!

upload_events(location=location,
events_file=events_file) # Yay!
```

RETURN VALUES

All functions return some value.

Often, you have a parameter that needs some processing. You pass it through your function, and the function returns it after making some changes.

What makes sense for the type of function you're creating?

RETURN VALUES WE'VE SEEN BEFORE

Syntax:

return_value = function(parameter, other_parameter)

```
phone = phone.replace("-", "")
bread = int(bread)
print len(article)
attendees.append('Amanda')
for index, word in enumerate(article): …
```

RETURN VALUES

The **return** keyword serves two functions: it gives some value back, and it ends the function.

When the function ends, you **return** to the line of code you were at when you called the function.

```
def product(x, y):
    return x*y
```

RETURN VALUES

Some functions can have multiple return values.

If you don't specify anything, it will return **None.**

You can return a number, string, a list, a dictionary, True, False, None, or any other type of thing!

RETURN VALUES

Some functions can have multiple return points — depending on the outcome of the function, you may wish to return a different result.

For an example, see **https://github.com/shannonturner/python-lessons/blob/master/section_09_(functions)/division.py**

RETURN MULTIPLE VALUES

You can even return multiple items, separated by a comma!

Syntax, as demonstrated by a fake function:

```
def transform_file(filename):
    # Some code that doesn't fit on a  slide
    return updated_filename, changes

new_filename, changes = transform_file(filename)
```

EXERCISES!

Head over to **https://github.com/shannonturner/python-lessons/tree/master/playtime**