Editor

Functions

Section 22.6

Completed

< back to module

Skip straight to editor/terminal

Our tool box is nearly full of the basic tools we need to start building complex programs. Now that we can use conditionals and loops as well as manipulate variables, lists and dictionaries we can start to see how nearly anything is possible.

Functions are another tool in our toolbox, and an important one. Functions allow us to split our code apart into smaller chunks that we can call on at any point in our program. This allows us to avoid repetition and make our code more human-readable.

Defining and calling functions

Let's start with a very simple function that says 'hello' to Agent J.

```
def greet_agent():
    print('Hello Agent J!')

greet_agent()
```

When we want to create a function we use the *def* keyword followed by what we want to name the function. In this case, we've named our function `greet_agent()`. Now we use indentation to create the *body* of our function: in this case, we want it to print the string "Hello Agent J!" when the function is called.

Defining the function itself doesn't actually print the text 'Hello Agent J' to the screen. In order to execute the code inside the function, we use a *function call*. To *call* a function, we write the name of the function, followed by any necessary information in round brackets. In this case there isn't any needed additional information, so we can just call the function with `greet_agent()`.

The code above will have the below output.

```
Hello Agent J!
```

It's important to remember that a function must be *defined* before it can be *called*. If we try to do things out of order and call a function before we define it, we'll get an error.

```
greet_agent()

def greet_agent():
    print('Hello Agent J!')
```

Running the above code will give us the following error.

```
$ python program.py
Traceback (most recent call last):
  File "program.py", line 1, in <module>
    greet_agent()
NameError: name 'greet_agent' is not defined
```

Function parameters

We can modify this function to allow us to greet any of our agents by using a function *parameter*. This lets us pass information into our function at the time we call it, then have the function use that information within it when it executes.

```
def greet_agent(letter):
    print('Hello Agent ' + letter + '!')

greet_agent('J')
greet_agent('M')
greet_agent('Q')
greet_agent('S')
```

In this example, we define our function again, but this time we give it a parameter, `letter`. Think of parameters like placeholders in our function: we use them in the body of our function where we'd like to be able to control data at the time we call the function.

Here we've used our placeholder parameter `letter` inside our print method, `print('Hello Agent ' + letter + '!')`.

Then, having declared and created our function, we call it 4 times: once each for each of our known agents. Here's how our output looks:

```
Hello Agent J!
Hello Agent M!
Hello Agent Q!
Hello Agent S!
```

When we call a function that uses a parameter, the information we pass at the time we call it is called an *argument*. So when we call `greet_agent('J')` the `'J'` is the *argument*. In this case, the *argument* `'J'` is passed into the function `greet_agent()` and *assigned to the parameter* `letter`. The terms *parameter* and *argument* are often used interchangeably: there are technical definitions, but if you're chatting casually with other programmers don't be surprised if you hear arguments referred to as parameters and parameters referred to as arguments.

We can declare as many parameters as we need when we define our function. For example, here's a more complex function that accepts 3 parameters and prints out a more complex output for our 4 agents.

```
def greet_agent(letter, total_cases, solved_cases):
    print('Hello Agent ' + letter + '!')
    print('\tYou have solved ' + str(solved_cases) + ' cases.')

    percent_solved = solved_cases * 100 / total_cases
    print('\tThat\'s ' + str(percent_solved) + '% of your total cases marked as solved,
great job!\n')

greet_agent('J', 11, 8)
greet_agent('M', 15, 12)
greet_agent('Q', 20, 12)
greet_agent('S', 20, 15)
```

This code gives us the below output.

```
Hello Agent J!
    You have solved 8 cases.
    That's 72% of your total cases marked as solved, great job!

Hello Agent M!
    You have solved 12 cases.
    That's 80% of your total cases marked as solved, great job!

Hello Agent Q!
    You have solved 12 cases.
    That's 60% of your total cases marked as solved, great job!

Hello Agent S!
    You have solved 15 cases.
    That's 75% of your total cases marked as solved, great job!
```

Notice here that when we call our function, we put our arguments in the same order as parameters were declared in our function definition: this is using *positional arguments*. If we are using positional arguments and don't get our order correct – say if we mix up the order of `total_cases` and `solved_cases` – then we get some pretty strange output.

```
def greet_agent(letter, total_cases, solved_cases):
    print('Hello Agent ' + letter + '!')
    print('\tYou have solved ' + str(solved_cases) + ' cases.')

    percent_solved = solved_cases * 100 / total_cases
    print('\tThat\'s ' + str(percent_solved) + '% of your total cases marked as solved,
great job!\n')

greet_agent('J', 8, 11)
```

This will give us an incorrectly inflated percent solved rate for Agent J in the below output.

```
Hello Agent J!
    You have solved 11 cases.
    That's 137% of your total cases marked as solved, great job!
```

We can also use *keyword arguments* in our function call, where we directly associate a value to a particular parameter. This lets us mix up our order because we're being clear about which value we're assigning to which parameter.

```
def greet_agent(letter, total_cases, solved_cases):
    print('Hello Agent ' + letter + '!')
    print('\tYou have solved ' + str(solved_cases) + ' cases.')

    percent_solved = solved_cases * 100 / total_cases
    print('\tThat\'s ' + str(percent_solved) + '% of your total cases marked as solved,
great job!\n')

greet_agent(letter='J', solved_cases=8, total_cases=11)
```

This will give us our expected output and the correct percent solved number for Agent J.

```
Hello Agent J!
    You have solved 8 cases.
    That's 72% of your total cases marked as solved, great job!
```

Default parameter values

Sometimes when defining functions, it's useful to set *default values* for some or all of our parameters. If there is a frequently used value as an argument, it can make sense to set it as a default assumed value to make your function calls simpler to use.

Both Agent S and Agent Q have 20 total cases: it turns out that's the largest number of cases any agent can get assigned in a month. Knowing that, and assuming enough agents regularly reach the 20 cases to make it worthwhile, we might want to modify our function to include a default value for the `total_cases` parameter.

```
def greet_agent(letter, solved_cases, total_cases=20):
    print('Hello Agent ' + letter + '!')
    print('\tYou have solved ' + str(solved_cases) + ' cases.')

    percent_solved = solved_cases * 100 / total_cases
    print('\tThat\'s ' + str(percent_solved) + '% of your total cases marked as solved,
great job!\n')

greet_agent('S', 15)
```

This code will generate the expected output below.

```
Hello Agent S!
    You have solved 15 cases.
    That's 75% of your total cases marked as solved, great job!
```

Notice in the function definition for `greet_agent()` in the above example that we swapped the `solved_cases` and `total_cases` parameters around? That's because if we have a mixture of parameters with default values and some without, you *must* have the parameters with default values at the end of the list.

Why? Because then we can use positional arguments when we call our function. If we didn't organize our parameters this way, then the computer wouldn't know when to use the default values and when not to.

Single Responsibility Principle

The most useful thing about functions is you can call them anywhere, even *inside* other functions. This allows us to break our code apart into smaller, more readable and reusable chunks.

A good rule of thumb for functions is they should only do 1 thing. This makes them much more reusable, and – when we get to testing our functions – much easier to test. This is called the *Single Responsibility Principle*.

Let's revise our code to use a series of smaller functions called by one primary function. When we rewrite or revise our code to do the exact same thing in a different (and hopefully better) way, we call it *refactoring* our code. So let's *refactor* our previous code to break it up into smaller functions, each with a single responsibility.

```
def greet_agent(letter):
    print('Hello Agent ' + letter + '!')

def solved_case_rate(total_cases, solved_cases):
    percent_solved = solved_cases * 100 / total_cases
    print('\tYou have solved ' + str(solved_cases) + ' cases.')
    print('\tThat\'s ' + str(percent_solved) + '% of your total cases marked as solved,
great job!\n')

def agent_status(letter, total_cases, solved_cases):
    greet_agent(letter)
    solved_case_rate(total_cases, solved_cases)

agent_status(letter='J', total_cases=11, solved_cases=8)
agent_status(letter='M', total_cases=15, solved_cases=12)
agent_status(letter='Q', total_cases=20, solved_cases=12)
agent_status(letter='S', total_cases=20, solved_cases=15)
```

Here we have 3 functions. The first is `greet_agent()`, a function which takes a single parameter, `letter`, and prints out a nice greeting for our agent.

The second function is called `solved_case_rate()` and takes 2 parameters: `total_cases` and `solved_cases`. This function's responsibility is to print out some text about our agent's current case load status.

Finally, we have `agent_status` which takes 3 parameters: `letter`, `total_cases`, and `solved_cases`. This function does nothing but call our other 2 functions in the correct order.

So when we call `agent_status()` for each individual agent, for each call 2 more function calls are generated per agent, allow us to print out the information as before.

Return values

So far, we've had our functions use the `print()` method to create output directly, but we may not always want this. Fortunately, instead of generating output, we can instead use *return values* to send information from inside our function to the line where the function was called.

Here's a very simple example of how a return value works.

```
def addition(x, y):
    total = x + y
    return total

calculation = addition(31, 11)
print(calculation)
```

In the above example, we've created a function called `addition()` that takes 2 parameters. It adds these parameters together and saves that value in a variable called `total`, and then it *returns* the value assigned to `total`.

When we call the function later, passing in 31 and 11, our function dutifully does the arithmetic and returns the total – in this case 42 – which we store in the variable `calculation`. Finally, we print out the value stored in `calculation`.

We can streamline this code even more by removing some of the work of storing information in the variables `total` and `calculation`, which we don't really need here. If we want to be really concise, we can use the below code to do exactly the same thing.

```
def addition(x, y):
    return x + y

print(addition(31, 11))
```

When using a return value, it's important to know that it will *immediately* end the function execution, similar to how the *break* keyword works in a loop. For example, let's look at a variation of our code above.

```
def addition(x, y):
    return x + y
    print('You will never know I exist!')

print(addition(31, 11))
```

Here we've added a `print()` method to our function `addition()` ... but when we run it, it doesn't print out the line 'You will never know I exist!'. That's because we have a return value above it. The function finishes running before it gets to `print('You will never know I exist!')`, so this line will never execute.

This can be helpful for us if a function includes an if statement or a loop, as it allows us to leave the function execution early if certain conditions are met.

Let's create a program that determines whether or not Agent J has time to wait in line at the cafe for a coffee, or if he has to skip his morning coffee in order to get to his first meeting of the day. The answer depends how many people are in front of him, but also *who* is in front of him. Agent Q currently owes Agent J a favour, and she'll let him swap places with her in the coffee line if he's in a hurry just this one time.

```
# Determine if there is enough time to wait for coffee, given
# the current people in line in front of Agent J.
def can_wait_for_coffee(minutes_available, people_in_line):
    estimated_wait_time = len(people_in_line) * 2

    if estimated_wait_time < minutes_available:
        return 'Yes, plenty of time.'
    else:
        return can_swap_with_agent_q(minutes_available, people_in_line)

# Determine if Agent Q is in the line to swap with and, if so,
# if she's far enough ahead in line to make a swap worthwhile
# given the time available.
def can_swap_with_agent_q(minutes_available, people_in_line):
    if 'Agent Q' not in people_in_line:
        return 'Agent Q isn\'t in line... no coffee for Agent J today.'

    position_agent_q = people_in_line.index('Agent Q')
    wait_time_for_agent_q = (position_agent_q + 1) * 2

    if wait_time_for_agent_q < minutes_available:
        return 'Agent J can swap places with Agent Q and get his coffee.'
    else:
        return 'Not even Agent Q can save Agent J today... no coffee today.'

# Ask our question
print('Does Agent J have enough time to get coffee this morning?')

# Get our answer
people_in_line = ['Unknown Person A', 'Agent M', 'Agent Q', 'Unknown Person B']
print(can_wait_for_coffee(8, people_in_line))
```

We won't go through this example line by line – you should give it a try with the editor below with different values and orders of values in the `people_in_line` list as well as a different number of minutes Agent J can wait to make sure you understand what this program is doing and how it's doing it.

However, let's have a look at the `can_swap_with_agent_q()` function definition. Note here how we first do a check to see if Agent Q isn't in the line: `if 'Agent Q' not in people_in_line`. If she's not found in the `people_in_line` list, then we immediately return out of the function. There's no need to set `position_agent_q` or `wait_time_for_agent_q` variables or run the following checks: she's not in the line, so the rest of the function's logic isn't needed, so we exit the function immediately.

Another thing we should look at in this example is in the `can_wait_for_coffee()` function definition. Here, we have an if statement that checks if the value of `estimated_wait_time` is less than the `minutes_available` value Agent J has available to wait before he's late for his meeting. If this evaluates to `True`, then we return a string. But if it doesn't... we return another function! (Technically, we return the *return value of another function*. And if that function also returns a function, we return the return value of *that* function... and on and on like a game of pass the potato.)

The ability to have functions return other functions, which can then also return other functions, and on and on depending upon layers of logic gives you a glimpse of what's possible with programming. We take a series of small pieces of logic and slowly build up a complex system, piece by piece.

Time Remaining: Loading... **Language:** Python 2 | Python 3 | C **Theme:** Dark | Light **Status:** Editor Terminal Server **Server:** Reset **Log:** Download

Steps

Step 1

Create a program that will track what Agent S likes in their hot drink from the cafe. Try and split apart your program into smaller functions.

- Regular coffee, decaf coffee, lattes, and flat whites always get 2 sugars.
- Regular coffee and decafe coffee get milk.
- Earl Grey tea gets lemon and 1 sugar
- Other types of black tea get milk and 1 sugar
- Green tea doesn't get anything
- Espresso doesn't get anything

Step 2

Modify your program to allow Agent S to have a scone when they order any kind of tea, and a muffin when they order any type of coffee.

Editor – file: ~/output.py

```
1
```

Save

Terminal – user: kPFQpHTUjC

```
kPFQpHTUjC@2d64cb34e9eb:~$
```

Completed

< back to module

Copyright 2020 SANS. Version 1.8.1