SPLITTING OUR CODE

Day 12: Functions

Welcome to day 12 of the 30 Days of Python series! In this post we're going to be taking a look at functions.

We've used a number of built-in functions throughout the course, such as `print` and `input`, but now we're going to learn how to create some functions of our own.

Also we've created a short recap of Days 10 and 11 that you can view here to refresh your memory.

Why use functions?

Before we start writing functions, we need to understand why they're useful.

Let's take the example of `print` for now. Printing is something we do all the time, both when testing our code and when providing information to users of our programs.

While this is sounds like a fairly simple operation, the code for `print` is actually over 80 lines long! `print` also calls several other functions in order to do its job, so the complete functionality provided by `print` actually requires several hundred lines of code.

This presents a couple of major benefit of using functions:

1. They allow us to cut down on repeating potentially long and complicated code for operations we want to perform multiple times.
2. They make our code more readable. It's much easier to understand `print("Hello, world!")`, than the lengthy implementation of the `print` function.

This brings us to the third major benefit of functions: we don't need to know how the underlying code for a function works. We've been using `print` since the first day of this series, but we don't yet know anything about functions, and we certainly don't know what `print` is doing when we call it.

It's enough that we know how to use the function, and what the function will do with the values we provide. The implementation details can often be safely ignored.

Defining our first function

To start off, we're going to define a simple function called `get_even_numbers`. This function is going to print out the first ten even numbers, starting with `2`.

In case you're unclear on how to get the first ten even numbers, we can use a `for` loop and `range` like so:

```
for number in range(1, 11):
        print(number * 2)
```

We can also pass in a third value to `range` called `step`, which allows us to create sequences with different patterns. For example, specifying a `step` of `2` would allow us to get every second number in the specified range.

```
for number in range(2, 21, 2):
        print(number)
```

Either of these approaches is fine.

So, how do we turn this into a function?

The first step is to write the `def` keyword, short for "define". This indicates to Python that what follows is a function definition. One the same line, we then need to write the name of our function. This name is going to serve as an identifier for the function, just a like a variable name.

Directly after the function name, we need a pair of parentheses which are going to remain empty for now. Finally, we need to end the line with a colon. The first line of our function definition therefore looks like this:

```
def get_even_numbers():
```

However, this isn't legal syntax right now, because all functions need a function body. This body contains the code that should be run when we call the function. In our case, we want to run that for loop we specified above, so we need to include it in our function body.

The body of a function is written directly underneath the line where we define the function name, and must be indented, just like when we use for loops or conditional statements.

```
def get_even_numbers():
        for number in range(1, 11):
                print(number * 2)
```

With that, we have a fully working function, and we can call it just like any of the built-in functions:

```
def get_even_numbers():
        for number in range(1, 11):
                print(number * 2)

get_even_numbers()
```

Run the code for yourself and see!

Style note

As we start writing more and more complex functions, it's going to be common for there to be empty lines within the function body to break up the code. For this reason, all of our function definitions should be followed by **two** empty lines.

This is going to help make it very easy to see where our functions end at a glance.

Function parameters and arguments

The `get_even_numbers` function works great, but many of the functions we've been using throughout this series have been able to accept values when we call them. For example `print` accepts the values we want to output, and `input` accepts a prompt.

There are even some functions we *have to* pass a value to. We can't call `len` without passing it a collection, for example. This is invalid:

```
len([1, 3, 5])  # 3
len()  # Error!
```

In the second line we didn't pass it a collection, so we get a `TypeError` :

```
Traceback (most recent call last):
  File "main.py", line 1, in <module>
    len()
TypeError: len() takes exactly one argument (0 given)
```

If we want to accept values in our own functions, we need to tell Python to expect values. We do this by defining *parameters*.

Remember those parentheses in our function definition that didn't seem to be doing very much? This is where we specify our parameters. In our original `get_even_numbers` function, we didn't have any parameters, so the parentheses were empty:

```
def get_even_numbers():
```

Let's update our `get_even_numbers` function so that the user can specify how many numbers to print out. We want to accept a single value, so we need to provide a single parameter. The value we pass in is called an *argument*.

A parameter is really just a variable, and it will provide us a way to access the arguments a user passes in. They serve as names for the argument values.

In the case of `get_even_numbers` , I'm going to define a parameter called `amount` by writing `amount` between the parentheses. I'm then going to use this parameter in the function body to modify the `range` .

```
def get_even_numbers(amount):
        for number in range(1, amount + 1):
                print(number * 2)
```

Don't forget that the stop value for `range` is non-inclusive, so we have to add `1` if we want to generate the right amount of numbers in this case.

Now if we call our `get_even_numbers` function and pass in an integer as an argument, we're going to be able to vary the amount of numbers we output.

```
def get_even_numbers(amount):
        for number in range(1, amount + 1):
                print(number * 2)

get_even_numbers(5)
```

Give it a try!

One thing you may notice is that we can no longer call the function without passing in any arguments. If we try, we're going to get a `TypeError`, just like we do when trying to call `len` without passing in an argument.

```
Traceback (most recent call last):
  File "main.py", line 6, in <module>
    get_even_numbers()
TypeError: get_even_numbers() missing 1 required posit
ional argument: 'amount'
```

The exception message is very helpful in this case. We get told we're missing a required argument, and the parameter missing a value is called `amount`.

There are ways we can make certain arguments optional, but this is something we're going to look at later in this series. For now it's just important to realise that if we specify a parameter, we need to provide an argument for that parameter when we call the function.

Specifying multiple parameters

Let's define a new function called `x_print`. `x_print` is going to accept two arguments: a string to print, and a number of times to print that string.

So if we call `x_print` like this:

```
x_print("Hello!", 5)
```

We would expect `Hello!` printed to the console 5 times, like so:

```
Hello!
Hello!
Hello!
Hello!
Hello!
```

Defining this function is really no more complicated than the `get_even_numbers` function, but I want to draw your attention to how Python assigns arguments to parameter by default.

Here is one possible implementation of `x_print`:

```
def x_print(requested_output, quantity):
        for _ in range(quantity):
                print(requested_output)
```

Here we have a `for` loop where we grab items from a `range` sequence. However, we don't actually care about the values in this `range`: we're just using the `range` to ensure something happens a certain number of times. To indicate that the loop variable is unimportant, we've use an `_` as a variable name, which is a common convention in cases like this.

For each iteration of the loop, we print out whatever the user provided as their requested output.

One thing we need to be really clear on, is that Python doesn't understand what any of these names we've defined mean. Parameters, like variables, are just names for our convenience as developers, so that we can easily reason about the sorts of values we're dealing with.

When we have multiple parameters, by default Python is going to assign argument values to these parameters in order.

You might have noticed some of the error messages we've been reading have referred to *positional arguments*. That's because these arguments are assigned to parameters based on their position in list of arguments.

This fact means the order we provide arguments to a function can be very important, and if we're not careful, we can end up with some bugs. For example, if we write something like this,

```
def x_print(requested_output, quantity):
        for _ in range(quantity):
                print(requested_output)

x_print(5, "Hello")
```

We're going to have some problems. Now `5` is assigned to `requested_output`, and `"Hello"` is assigned to `quantity`. This is going to give us an error, because we provide `quantity` to `range`, and `range` expects an integer.

What we get in this case is a `TypeError`:

```
Traceback (most recent call last):
  File "main.py", line 6, in <module>
    x_print(5, "Hello")
  File "main.py", line 2, in x_print
    for _ in range(quantity):
TypeError: 'str' object cannot be interpreted as an in
teger
```

Keyword arguments

We've seen keyword arguments a couple of times in this series, but we've not really spoken about them directly. Keyword arguments are an alternative to positional arguments, where we specifically tie an argument's value to a parameter name.

For example, we can call our `x_print` function like this:

```
def x_print(requested_output, quantity):
        for _ in range(quantity):
                print(requested_output)

x_print(requested_output="Hello", quantity=5)
```

Here we've specified that `"Hello"` should be the value of `requested_output`, and `5` should be the value for `quantity`. Because we've specified which value belongs to which parameter, we now don't have to worry about the order we specify these values.

For example, this will work just fine:

```
x_print(quantity=5, requested_output="Hello")
```

We can also mix positional and keyword arguments, but we have to be a little bit careful here as there are several limitations.

We can't provide a positional argument after a keyword argument. This is not valid syntax in Python:

```
x_print(requested_output="Hello", 5)
```

If we provide a positional argument that maps onto a given parameter, we can't then provide a keyword argument for that same parameter. For example, we can't do this:

```
x_print(5, requested_output="Hello")
```

The issue here is that `5` is a positional argument, and so gets assigned to `requested_output` (the first parameter). When we then specify a keyword argument for the same parameter, Python realises that something is wrong, and raises a `TypeError`.

If we want to mix positional and keyword arguments for our `x_print` function, we only have one viable option:

```
x_print("Hello", quantity=5)
```

This satisfies all of the requirements. The keyword arguments comes after the positional arguments, and we haven't got any duplicate assignments caused by the order of our values.

It's generally a good idea to use keyword arguments wherever you can, because they provide a lot of readability benefits. However, it's not always possible to provide keyword arguments for parameters. Many built-in functions don't accept keyword arguments for certain parameters, and in Python 3.8 we gained the ability to define positional only parameters as well.

These parameters will not accept keyword arguments. If you try, you'll just get a `TypeError`.

There's a lot more we need to learn about functions, but for now, let's test what we've learn with some exercises.

Exercises

1) Define four functions: `add`, `subtract`, `divide`, and `multiply`. Each function should take two arguments, and they should print the result of the arithmetic operation indicated by the function name.

When orders matters for an operation, the first argument should be treated as the left operand, and the second argument should be

treated as the right operand. For example, if the user passes in `6` and `2` to subtract, the result should be `4`, not `-4`.

You should also make sure that the user can't pass in `0` as the second argument for `divide`. If the user provides `0`, you should print a warning instead of calculating their division.

2) Define a function called `print_show_info` that has a single parameter. The argument passed to it will be a dictionary with some information about a T.V. show. For example:

```
tv_show = {
        "title": "Breaking Bad",
        "seasons": 5,
        "initial_release": 2008
}

print_show_info(tv_show)
```

The `print_show_info` function should print the information stored in the dictionary, in a nice way. For example:

```
Breaking Bad (2008) - 5 seasons
```

Remember you must define your function *before* calling it!

3) Below you'll find a list containing details about multiple TV series.

```
 series = [
        {"title": "Breaking Bad", "seasons": 5, "initi
al_release": 2008},
        {"title": "Fargo", "seasons": 4, "initial_rele
ase": 2014},
        {"title": "Firefly", "seasons": 1, "initial_re
lease": 2002},
        {"title": "Rick and Morty", "seasons": 4, "ini
tial_release": 2013},
        {"title": "True Detective", "seasons": 3, "ini
```

```
tial_release": 2014},
        {"title": "Westworld", "seasons": 3, "initial_
release": 2016},
 ]
```

Use your function, `print_show_info`, and a `for` loop, to iterate over the `series` list, and call your function once for each iteration, passing in each dictionary. You should end up with each series printed in the appropriate format.

4) Create a function to test if a word is a palindrome. A palindrome is a string of characters that are identical whether read forwards or backwards. For example, "was it a car or a cat I saw" is a palindrome.

In the day 7 project, we saw a number of ways to reverse a sequence, and you can use this to verify whether a string is the same backwards as it is in its original order. You can also use a slicing approach. Once you've found whether or not a word is a palindrome, you should print the result to the user.

Make sure to clean up the argument provided to the function. We should be stripping whitespace from both ends of the string, and we should convert it all to the same case, just in case we're dealing with a name, like `"Hannah"`.

You can find our solutions to the exercises here.

Project

Once you're done with the exercises, it's time to tackle today's project!

Additional resources

If you want to learn about positional-only parameters, there's a small summary in the Python documentation describing new features in Python 3.8. The section on positional-only parameters also describes a few use cases for this feature.

If you're interested in expanding on your palindrome solution to work with whole sentences, and words with punctuation, we have a much more detailed walkthrough on our blog.
