SPLITTING OUR CODE

Day 13: Scope and Returning Values from Functions

Hello, and welcome to day 13 of the 30 Days of Python series! Today we're going to start to learn about the very important topic of *scope*.

We can think of scope as a description of where a given name can be referenced in our application. Understanding how this works in Python is a vital step in our learning journey.

We're also going to be expanding the ways we can use our functions. So far all of our functions have just printed something to the console, but there are cases where we want to get something back from our functions as well. We're going to talk about how to do that in today's post.

A demonstration of scope

As I mentioned already, scope is a concept describing where a given name can be referenced in our application.

We haven't really encountered any cases where a name we've defined hasn't been accessible yet, so let's look at an example.

First, we're going to define a simple function called `greet`.

```
def greet(name):
        greeting = f"Hello, {name}!"
        print(greeting)
```

All `greet` does is take in a name and print a greeting to the user. Inside `greet` we've defined a variable called `greeting` which is where we assign our formatted string, and we pass this `greeting` string to `print` as an argument.

The question is, can we access this `greeting` string outside of the function?

```
def greet(name):
        greeting = f"Hello, {name}!"
        print(greeting)

print(greeting)
```

In this case, it looks like we get an error:

```
Traceback (most recent call last):
  File "main.py", line 6, in <module>
    print(greeting)
NameError: name 'greeting' is not defined
```

But maybe this isn't a fair example. After all, we never actually called `greet`, so the code where `greeting` is defined was never really run. How could it? We haven't even told it what `name` means.

Maybe this will work after we call `greet`.

```
def greet(name):
        greeting = f"Hello, {name}!"
        print(greeting)

greet("Phil")
print(greeting)
```

Now our output looks like this:

```
Hello, Phil!
Traceback (most recent call last):
  File "main.py", line 7, in <module>
    print(greeting)
NameError: name 'greeting' is not defined
```

As we can see, `"Hello, Phil!"` printed when `greet` was called, but still `greeting` is undefined. What's going on?

The issue is actually that `greeting` is "out of scope".

Namespaces

To better understand what's going on in the example above, we need to think about what happens when we define a variable.

Python actually keeps a record of the variables we've defined, and the values that are associated with those names. We call this record a namespace, and you can think of it as being a dictionary. We can get a little peek at this dictionary by calling the `globals` function and printing the result.

First, let's look at what happens when we call `globals` in an empty file.

```
print(globals())
```

If you run this code, it may surprise you to find that this dictionary is not empty. We have all kinds of interesting things in it.

```
{
        '__name__': '__main__',
        '__doc__': None,
        '__package__': None,
        '__loader__': <_frozen_importlib_external.Sour
ceFileLoader object at 0x7f2b30020bb0>,
        '__spec__': None,
        '__annotations__': {},
        '__builtins__': <module 'builtins' (built-in)
>,
        '__file__': 'main.py',
        '__cached__': None
}
```

This makes some sense though. After all, we don't have to define every single name we use in Python. We never defined `print`, `input`, or `len`, for example.

Don't worry too much about the actual content of this dictionary right now. Just take note of the things that already exist, so that we can see what happens when we define some names of our own.

Let's change our file so that it actually has some content. I'm going to define a couple of variables and a function, just so we have a bit of variety:

```
names = ["Mike", "Fiona", "Patrick"]
x = 53657

def add(a, b):
        print(a, b)

print(globals())
```

Now let's looks at what we have in our `globals` dictionary. Pay particular attention to the last three items.

```
{
        '__name__': '__main__',
        '__doc__': None,
        '__package__': None,
        '__loader__': <_frozen_importlib_external.Sour
ceFileLoader object at 0x7fae511ffbb0>,
        '__spec__': None,
        '__annotations__': {},
        '__builtins__': <module 'builtins' (built-in)
>,
        '__file__': 'main.py',
        '__cached__': None,
        'names': ['Mike', 'Fiona', 'Patrick'],
        'x': 53657,
        'add': <function add at 0x7fae512021f0>
}
```

We can see the names we defined (`names` , `x` , and `add`) have been written as keys, and associated with each of our names is the value we assigned to that name.

`add` looks a little bit strange, but this `<function add at 0x7fae512021f0>` is just a representation of the function we defined. We can see that it's a function, its name is `add` , and the last number is a memory address for this function.

When we use a name in our application, Python just looks in the namespace to see if it's defined. If it is, it can just reference the value associated with that name. If it can't find the name we requested, Python says that the variable is undefined.

Functions and namespaces

Looking back at our `globals` output, there are a couple of names which are notably absent: the parameters `a` and `b` that we defined for `add` . Maybe you expected those to be there alongside the other names we defined. What happened to those?

Python actually has more than one namespace. The one we're looking at here is just the *global namespace*, but the parameters for a function are not part of this global namespace.

When we call a function, Python creates a new namespace. In other words, it creates a new dictionary to store any names we want to use while running this function. Once the function finishes running, this namespace is destroyed, so that when we run the function next time, we're working with a blank slate.

Just like when we used `globals`, we can get a peek at this function namespace by calling `locals`. Let's make a small change to `add` to see this:

```
def add(a, b):
        print(locals())
        print(a, b)

add(7, 25)
```

The output we get in this case is as follows:

```
{'a': 7, 'b': 25}
7 25
```

The first line is the `locals` dictionary, and contains the names located within the namespace for this function call.

Now we can better explain what's going on in the `greet` example from earlier in the post.

```
def greet(name):
        greeting = f"Hello, {name}!"
        print(greeting)
```

The global namespace doesn't contain the name, `greeting`, so when we try to reference it outside of the function, Python can't find

it. Inside `greet`, however, we're working with a second local namespace that was created when we called the function. This namespace contains `greeting`, because we added it when we defined `greeting` in the function body.

We can see that in more detail by doing something like this:

```
def greet(name):
        print(locals())
        greeting = f"Hello, {name}!"
        print(locals())
        print(greeting)

greet("Phil")
```

If we run this code, we see the following:

```
{'name': 'Phil'}
{'name': 'Phil', 'greeting': 'Hello, Phil!'}
Hello, Phil!
```

We can see that when we enter the function body for this function call, we only have `name` defined. We then move onto the second line where `greeting` gets assigned the result of our string interpolation. When we print `locals()` again on the third line of the function body, we can see that `greeting` has been added to this local namespace for the function call.

I think this makes some intuitive sense. The function has some private variables that only it knows about, and we have a table of variable names and values that we create when we're running the function to keep track of what's going on inside. When we're done with the function, we wipe the table clean, because we no longer need any of those values.

Getting values out of a function

Since all of the variables we define in our functions only exist inside of our functions, the question is, how do we get a value back out of the function? The `input` function is able to provide us the user's response to our prompt, for example. We want to be able to do something similar.

The way that we get a value out of a function is using a `return` statement.

The `return` statement actually has two roles. First, it's used to end the execution of a function. When Python encounters the `return` keyword, it immediately terminates the function.

For example, if we write something like this:

```
def my_func():
        return
        print("This line will never run")
```

We'll never run the `print` call on the second line of the function body. The `return` keyword will cause the function to terminate before we reach that line with the `print` call.

We can also put an expression directly after the `return` keyword, and this is how we get values out of the function. We put the values we want to get out on the same line as the `return` keyword.

For example, let's write a new version of our `add` function from yesterday's exercises. Instead of printing the result, we're going to return the result instead.

```
def add(a, b):
        return a + b
```

If we run this function, nothing seems to happen. We're no longer printing anything, after all. However, we can now assign the return value to a variable if we want.

```
def add(a, b):
        return a + b

result = add(5, 12)
print(result)  # 17
```

So, what's actually happening here?

Remember that a function call is an expression. It evaluates to some value. What value? A function call evaluates to the value that was returned by the function when we called it.

In the example above, we returned the result of the expression, `a + b`. Since `a` was `5` and `b` was `12`, the result of that expression was the integer `17`. This is what was returned by the function, and this is the value the function call evaluated to.

By the way, because `add(5, 12)` is just an expression, we could have passed the function call to `print` directly:

```
def add(a, b):
        return a + b

print(add(5, 12))  # 17
```

But what about when we don't have a `return` statement? If we don't specify a value to return, Python *implicitly* returns `None` for us.

For example, if we try to find the value of our `greet` function call, we get `None`:

```
def greet(name):
        greeting = f"Hello, {name}!"
        print(greeting)

print(greet('Phil'))
```

The output we get is:

```
Hello, Phil!
None
```

First we get the result of calling the function, because Python needs to call the function to figure out what the value of `greet('Phil')` is; then we print the return value of the function, which is `None`.

The same thing happens if you type `return` without a value. Python will return `None`.

```
def greet(name):
        greeting = f"Hello, {name}!"
        print(greeting)
        return

print(greet('Phil'))
```

Finally, remember we can use the return value of the function anywhere we might use a plain old value. For example, in variable assignment or as part of a string:

```
def greet(name):
        greeting = f"Hello, {name}!"
        print(greeting)

print(f"The value of greet('Phil') is {greet('Phil')}
.")
```

Here we get the string telling us that the value of `greet('Phil')` is `None`:

```
Hello, Phil!
The value of greet('Phil') is None.
```

Multiple `return` statements

Sometimes a function definition might have more than one `return` statement. This is totally legal, but only useful if we have some kind of conditional logic that directs us towards just one of the `return` statements. Remember that a function is going to terminate as soon as we encounter any `return` statement, so if we have more than one in series, we'll never hit the ones after the first.

An example where multiple `return` statements makes sense is with our `divide` function:

```
def divide(a, b):
        if b == 0:
                return "You can't divide by 0!"
        else:
                return a / b
```

This makes sense, because while we have multiple `return` statements, we're being directed to just one `return` statement by our conditional logic.

Because `return` will cause a function call to terminate, we can make a slight modification to the code above:

```
def divide(a, b):
        if b == 0:
                return "You can't divide by 0!"

        return a / b
```

This still works, because in any cases where `b` is `0`, we hit this `return` statement and break out of the function. We therefore never hit the point where we perform the calculation. The only way we get there is if `b` is not `0`.

This is a very common pattern that people use to save writing this `else` clause. There's no harm putting on in though, and feel free to

include it in your own code. Whatever is more comfortable for you.

Exercises

1) Define a `exponentiate` function that takes in two numbers. The first is the base, and the second is the power to raise the base to. The function should return the result of this operation. Remember we can perform exponentiation using the `**` operator.

2) Define a `process_string` function which takes in a string and returns a new string which has been converted to lowercase, and has had any excess whitespace removed.

3) Write a function that takes in a tuple containing information about an actor and returns this data as a dictionary. The data should be in the following format:

```
("Tom Hardy", "English", 42)  # name, nationality, age
```

You can choose whatever key names you like for the dictionary.

4) Write a function that takes in a single number and returns `True` or `False` depending on whether or not the number is prime. If you need a refresher on how to calculate if a number is prime, we show one method in day 8 of the series.

You can find our solutions to the exercises here.