

CSE231 - Lab 04

Functions, Scope, `global`, Defaults, Type Hints

Everyone here has taken calc, right?

Functions are a topic that a lot of new programmers struggle on. Try to think of them just as the functions we know from mathematics.

$$f(x) = 2x+1$$

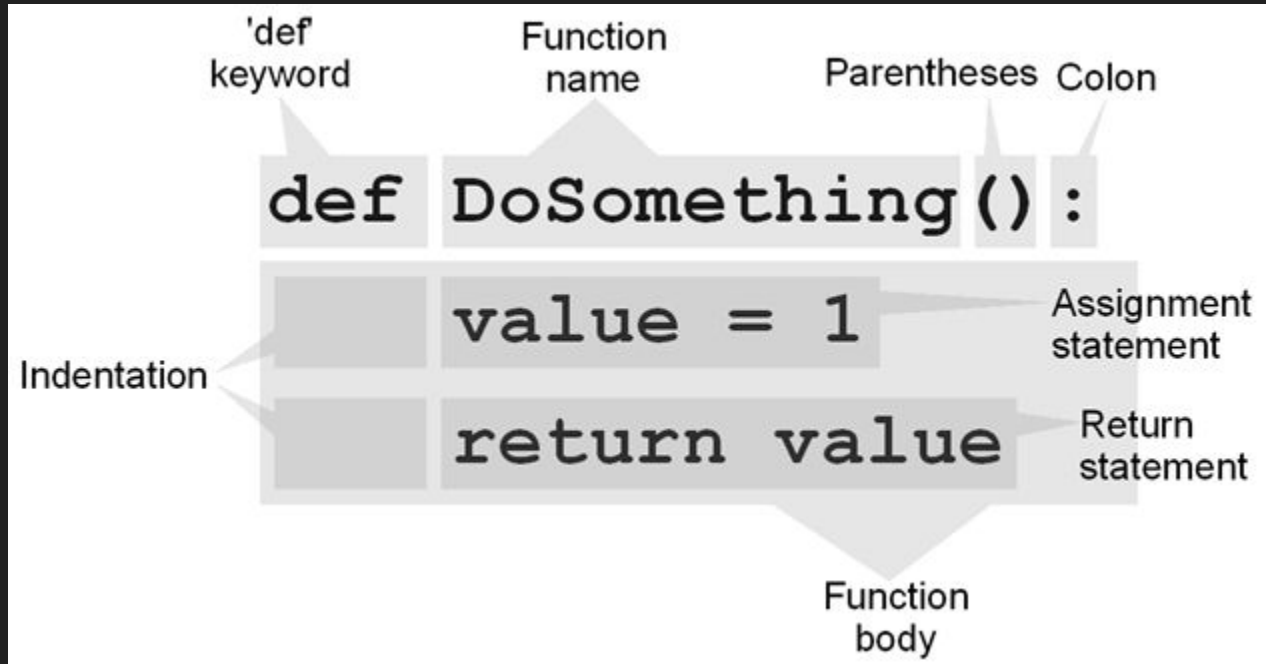
You can think of 'x' as the generalized parameter, where you can plug in any number you want (the domain, $[-\infty, \infty]$ in this case).

You can think of the number that the function spits back out as the “return” (the range).

$$f(2) = 2(2)+1$$

$$f(2) = 5$$


Function Syntax



Python Translation

Let's take our function from earlier, $f(x) = 2x+1$, and translate it to Python:

```
def f(x):  
    return 2*x + 1
```



Simple, right? Again, 'x' is the parameter, it can take on the value of anything we pass into it, and the function will spit back out a different value -- the "return" value.

It's the same concept to functions in mathematics. In programming however, we can do more than just math in our functions.

Why do we need functions?

For the same reason we don't want to copy-paste chunks of code. Functions offer the ability to ***generalize***. We can keep the same algorithm, but we can call it with different parameters to get different outcomes.

It makes code more readable, easier to debug, and highly reusable -- especially if you make your functions error-proof.

We've already seen plenty of functions that are included with Python by default, like `print()` and `input()`, but now we can make our own.

None

As you're working with functions, you'll likely encounter a variable holding the value, 'None'. So what is None?

Like ints, floats and strings, None is a type. You'll pretty commonly see Python refer to it as "NoneType" in its error messages.

You can think of None as representing the absence of a value. If you're familiar with other programming languages, it's functionally the same to the "null" type that many other languages use.

None (cont.)

So why am I talking about None? Where does it appear?

If you create a function that *doesn't return anything*, (whether that be intentional or by mistake), Python will automatically return None for that function.

```
def f(x):  
    pass  
  
var = f(1)  
  
print(var)  # prints: `None`
```

None (cont.)

In certain situations, having your function return `None` can be useful. When assessed as a boolean, `None` equates to `False` -- making it extremely easy to assess if the function was successful in doing whatever it does.

```
def f(x):  
    if x > -1:    # function only returns something if x > -1  
        return 2*x + 1  
  
var = f(-3)    # since -3 is less than -1, no return, i.e. var = None  
if var:  
    print("Success!") # if None -> if False ...  
else:  
    print("Function returned None")    # meaning that this suite runs!
```


Scope

What is scope?

Scope can be defined as *where* the variables you create in your program exist. When you run a program, Python is constantly destroying variables you aren't using in the background to conserve memory on your computer.

Scope becomes extremely relevant when we're talking about functions. Think of functions as existing in a separate world. You can give permission to the function to use certain values (the parameters), and it'll send you values back (the return), but neither you nor the function can access each other's stuff directly without that permission. There is, however, an exception for global scope.

Local Scope

I have a function, and I want the string I initialized inside of it. Why can I not access it?

```
def my_function():  
    my_variable = "Hello, there."  
  
print(my_variable)
```

Things that exist in a function *are for the function*. If you want to obtain a value from it, you would have to take the value from a return.

The variable ``my_variable`` is ***local*** to ``my_function``.

Global Scope

This is where things get tricky. Does this code run?

```
x = 10

def my_function():
    print(x)

my_function()
```

`x`, in this case, is considered to be in the *global namespace*.

Global Scope (cont.)

Let's change up our function a little, does this code run now?

```
x = 10

def my_function():
    x += 1

my_function()
```

Spoiler: it doesn't. We are trying to change the variable `x`, we end up getting an `UnboundLocalError`, meaning that Python thinks this variable doesn't exist... so how come we could print it earlier?

Global Scope (cont.)

Variables that are initialized in the *global namespace*, (the space with no indentation), are accessible everywhere in the program, *including the inside of functions*.

However, their value *cannot be changed by a function*. Functions can “look at” these variables, you can copy the value, print it, whatever, as long as you’re not *changing* what the variable holds.

This *does not* go the other way around -- you cannot access variables declared *inside* functions in the space *outside* of them.

Global Scope (cont.)

“What if I want to change a global variable’s value inside a function?”

Then you need to pass it into the function *as a parameter*, and *overwrite that name with the function’s return*. However, if you need to do this, you likely shouldn’t be considering that variable as global in the first place.

Important: do not use the `global` keyword. More on this in a second.

global

Now is probably a good time to mention that you're *not allowed* to use this keyword. Some of you may be aware of it if you've been Google-searching a lot.

There are a multitude of reasons why you shouldn't use `global`, synopsis is that it's bad practice. I unfortunately have to dock points if I see you use it (item 9 of the coding standard).

I won't be discussing how to use `global` since it's not something you should be using in a vast majority of circumstances, including situations outside of this class.

Defaults

In your function declaration, you can set “parameter defaults”. If a parameter has a default, then the parameter becomes *optional*. You can call the function *without* that parameter, making it so that the default value takes over.

```
def f(a, b=10):          # defaults must follow non-defaults,  
    if b != 0:           # or all arguments must have defaults  
        return a / b  
    else:  
        return a
```

```
var1 = f(4)              # f(a=4, b=10)  
var2 = f(4, 100)         # f(a=4, b=100)
```


Type Hints

Type hinting is a relatively new Python feature (versions 3.5 and above I believe) that allows you to show the expected types of your function parameters. This is purely an aesthetic addition, and doesn't prevent parameters from being invoked with unexpected types.

```
def f(a:int, b:float, c:str) -> str:
```

```
    if c == 'a':
```

```
        return "Hello"
```

```
    elif c == 'b':
```

```
        return a * "Hello" + str(b)
```

(This is just spam-code, the function declaration is the important bit here)

Type Hints (cont.)

When you want a type hint *and* a default argument, the default goes after the type hint. Example:

```
def f(a:int=1, b:float=2.2, c:str='a') -> str:
```

```
    if c == 'a':
```

```
        return "Hello"
```

```
    elif c == 'b':
```

```
        return a * "Hello" + str(b)
```

Even with type hints, you should still include a function docstring as the coding standard explains. The types are covered, but what the parameters are for should be explained.