Functions

If you find yourself doing the same thing over and over again in your code, it might be time to write a function.

Functions are blocks of reusable code -- little boxes that (usually) take inputs and return outputs. In Excel, `=SUM()` is a function. `print()` is one of Python's built-in function.

You can also *define your own functions*. This can save you some typing, and it will help separate your code into logical, easy-to-read pieces.

Syntax

Functions start with the `def` keyword -- short for *define*, because you're defining a function -- then the name of the function, then parentheses (sometimes with the names of any `arguments` your function requires inside the parentheses) and then a colon. The function's code sits inside an indented block immediately below that line. In most cases, a function will `return` a value at the end.

Here is a function that takes a number and returns that number multiplied by 10:

In [1]:
```
def times_ten(number):
    return number * 10
```

The `number` variable is just a placeholder for the values we're going to hand the function as input. We could have called that argument name "banana" and things would be just fine, though it would be confusing for people reading your code.

Calling a function

By itself, a function doesn't do anything. We have built a tiny machine to multiply a number by 10. But it's just sitting on the workshop bench, waiting for us to use it.

Let's use it.

In [2]:
```
two_times_10 = times_ten(2)
print(two_times_10)
```

```
20
```

Arguments

Functions can accept *positional* arguments or *keyword* arguments.

If your function uses *positional* arguments, the order in which you pass arguments to the function matters. Here is a function that prints out a message based on its input (a person's name and their hometown).

In [3]:
```
def greet(name, hometown):
    return f'Hello, {name} from {hometown}!'
```

Now let's call it.

In [4]:
```
print(greet('Cody', 'Midvale, WY'))
```

```
Hello, Cody from Midvale, WY!
```

If we change the order of the arguments, nonsense ensues.

In [5]:
```
print(greet('Midvale, WY', 'Cody'))
```

```
Hello, Midvale, WY from Cody!
```

Using *keyword* arguments requires us to specify what value belongs to what argument, and it allows us to set a default value for the argument -- values that the function will use if you fail to pass any arguments when you call it. We could rewrite our function like this:

In [6]:
```
def greet(name='Cody', hometown='Midvale, WY'):
    return f'Hello, {name} from {hometown}!'
```

And now it doesn't matter what order we pass in the arguments, because we're defining the keyword that they belong to:

In [7]:
```
print(greet(hometown='Pittsburgh, PA', name='Jacob'))
```

```
Hello, Jacob from Pittsburgh, PA!
```

What happens if we call the `greet()` function without any arguments at all, now? It'll use the default arguments.

In [8]:
```
print(greet())
```

```
Hello, Cody from Midvale, WY!
```

Lambda expressions

Sometimes, you'll see code that looks like this:

```
df['new_column'] = df['old_column'].apply(lambda x: x[0])
```

That stuff inside the `apply()` parentheses? That's called a *lambda expression*, a time-saving way to turn loose a simple function on some values without having to write out a function with `def`. (It's a Python thing, not specific to pandas, but for our purposes that's probably where you'll see them most often.)

This code is equivalent but takes longer to write:

```
def take_first_char(value):
    return value[0]

df['new_column'] = df['old_column'].apply(take_first_char)
```

More resources

- TutorialsPoint post on functions
- LearnPython tutorial
- Software Carpentry tutorial
- Hitchhiker's Guide to Python: Function Arguments