Functions

You can think of functions as an instruction manual: instead of re-writing all the instructions every single time someone needs them to do something, you can refer that person to the instruction manual (which means you only needed to write the instructions once). Thus, functions are handy for code that is used over and over again: saving yourself a lot of time and making your code much more:

- flexible (just change the function and you can change the whole code)
- readable (less line of code overall)
- debuggable (any bugs can be traced back to the function rather than the specific place you wrote out the code if you were to copy and paste the code over and over)

You may have seen functions before but under a different name: methods. Languages such as Java and C refer to functions as methods (but they really do the same thing conceptually). Before I get into the technical side of things, I want to talk about what a function is. As I said before, it's a premade set of instructions that can be referred to over and over again, so you don't have to rewrite the code everytime.

Now there are some parts we need to be aware about:

- Name (name of the function)
- Parameters (info passed into the function)
- Body (the code the function has)
- Return Statement (returning values)
- Type Hinting (hint at what types the parameters and returned value are)

In real life, an instruction manual has all five of these parts:

- Name: the name of the manual/the name of the thing we are trying to do or build
- Parameters: there's a list of parts you need before you start working
- Body: the actual instructions telling you what to do
- Return Statement: usually that's the end result and giving it to the person that it's meant for
- Type Hinting: tells you exactly what tools you need and what the end product will be

Another Analogy

If that is confusing, imagine you are the Python interpretter. But this time, a friend wants you to deliver them a cake made according to a recipe. This recipe is basically a function: a premade set of instructions for you to follow in order to make your cake. Using the same template above, we can relate a recipe to a function:

- Name: the name of the recipe
- Parameters: the ingredients list
- Body: the instructions list
- Return Statement: how the cake is delivered to your friend
- Type Hinting: tells you what exact ingredients are needed and what the cake should look like at the end

I'll show a quick example of a full function and we'll go piece by piece:

In [ ]:
```python
# Example 9-1: A function example

def add(a: float, b: float) -> float:
    return a + b
```

If you tried running the above, you'll see that nothing gets outputted. All I've done is create a function named `add` that takes two parameters, one named `a` and one named `b`, and I told Python that both will be floats or at least can be changed into floats (like integers). I've also said that the value that gets returned is a float (the `-> float` part). And the `return` is also the body of the function: all the function does is add up `a` and `b` and return that value to wherever the function was called.

Function Breakdown

So, piece by piece, this first part:

`def add(a: float, b: float) -> float:`
The `def` keyword means I'm about to create a function. The next word I use (it's basically a variable name as function names follow variable naming conventions) will be the name of the function (in this case, `add` is the name). Then I put parentheses. These parentheses hold any parameters: if I want to send any information to the function, I need to store them somewhere. Thus parameters exist as local function variables that hold values assigned to it, so Python has the information it needs to properly execute the function. Otherwise, with the example I made, it's gonna be wondering what exactly it's supposed to be adding.

The `: float` is type-hinting: telling Python what type this parameters will be. If Python gets something it can't immediately understand as a float, it will get mad and throw an error. This is optional of course but is very good coding practice and is good for yourself too since you know what type of data should be going through there. The `-> float` is also type-hinting but rather tells Python what type the

returned value will be. If you're returning nothing (usually this is when there is no `return` statement), then the type is `None`.

The next part:

```
return a + b
```

Note that this line is indented. Python doesn't know what belongs in the function and the way to do this is to have the lines indented. Once Python meets a line of code that is unindented, Python understands that the function is finished and that line is part of something else entirely. The body of the program can have several lines, but this is a rather simple program, so it's just one line. `return` literally returns the value back. This is essential if you're using the value from the function for a calculation because there is no other (good) way to get the value. There are some ways but they are extremely bad practice and generally ends up messing up your code, so I won't talk about them. In this case, the value being returned is whatever the value of `a + b` is.

Using the function

Now we gotta use the function. We got the instructions written down, and now we just need Python to go ahead and use those instructions to do something for us (in this case, add two numbers):

In [ ]:
```
# Example 9-2: Calling the function
def add(a: float, b: float) -> float:
    return a + b

add(1, 2)
```

Out[ ]: 3

Now note that in a this type of file, Python will actually auto print the value of a variable written on a line by itself like that. In reality, there is no output! This is because all we've done is add 1 and 2 and return the value, but we haven't done anything with it yet. Let's go ahead and print it two ways (directly and via variable):

In [ ]:
```
# Example 9-3: Printing the results
def add(a: float, b: float) -> float:
    return a + b

print(add(1, 2))
result = add(3, 4)
print(result)
```

```
3
7
```

You can of course have a function not return anything and instead just do something in the function. This is not recommended, but we could just print the result within the function:

In [ ]:
```
# Example 9-4: Modified add function
def new_add(a: float, b: float) -> None:
    print(a + b)

new_add(1, 2)
new_add(3, 4)
```

```
3
7
```

I said earlier that if you have a function without a `return` statement, then your return type is `None`. Let's see what happens if I try to print these function calls out:

In [ ]:
```
# Example 9-5: None return type
def new_add(a: float, b: float) -> None:
    print(a + b)

print(new_add(1, 2))
print(new_add(3, 4))
```

```
3
None
7
None
```

Remember, Python executes commands sequentially. So it will call the `new_add` first with 1 and 2 being passed in and then it will try to print out the value of that function (which is `None`). It repeats again but with 3 and 4 being passed into `new_add`.

These have been functions that were used once or twice, so their practicability hasn't really been seen yet. But imagine you could use functions for Lab 2.10 (More Linear Interpolation): you can write a function that takes in the current time, the initial time and position, and the final time and position, and the function would return the position at the current time. This would save a lot of time copy and pasting and adjusting the values to use the correct variables! Also note that this is just a very basic overview of functions: you will go in more detail in Module 9.

Print Functions

These aren't a special type of function. This is just a common use case of functions since most beginner-level user interfaces are with the command line or the console. More specifically, these functions are made when you want to make the output look fancy and pretty instead of just super plain text. An example of a print function is shown below.

In [ ]:
```
# Example 9-5: Basic print function and usage

def box_print(word):
    print("*" * (len(word) + 4))
    print(f"* {word} *")
    print("*" * (len(word) + 4))

box_print("Alana!")
```

```
**********
* Alana! *
**********
```

The possibilities for print functions are endless! Since it's a function, you also don't have to rewrite all the code every single time you want to print something in your customized way. You can just write the function once and then call the function whenever you need to fancy print. Note that since I am outputting stuff onto the console with the function, there is not really a reason for me to use a `return` statement. You could `return` a value if your print function both outputs text to the console and calculates a value, but arguably you should make two functions (one for printing, one for the calculation). But of course, if you need to use `return`, feel free to do so.