In Python, text in between quotes -- either single or double quotes -- is a string data type. An integer is a whole number, without a fraction, while a float is a real number that can contain a fractional part. For example, 1, 7, 342 are all integers, while 5.3, 3.14159 and 6.0 are all floats. When attempting to mix incompatible data types, you may encounter a **TypeError**. You can always check the data type of something using the *type()* function.

As we saw earlier in the video, some data types can be mixed and matched due to implicit conversion. Implicit conversion is where the interpreter helps us out and automatically converts one data type into another, without having to explicitly tell it to do so.

By contrast, explicit conversion is where we manually convert from one data type to another by calling the relevant function for the data type we want to convert to. We used this in our video example when we wanted to print a number alongside some text. Before we could do that, we needed to call the *str()* function to convert the number into a string. Once the number was explicitly converted to a string, we could join it with the rest of our textual string and print the result.

We looked at a few examples of built-in functions in Python, but being able to define your own functions is incredibly powerful. We start a function definition with the def keyword, followed by the name we want to give our function. After the name, we have the parameters, also called arguments, for the function enclosed in parentheses. A function can have no parameters, or it can have multiple parameters. Parameters allow us to call a function and pass it data, with the data being available inside the function as variables with the same name as the parameters. Lastly, we put a colon at the end of the line.

After the colon, the function body starts. It’s important to note that in Python the function body is delimited by indentation. This means that all code indented to the right following a function definition is part of the function body. The first line that’s no longer indented is the boundary of the function body. It’s up to you how many spaces you use when indenting -- just make sure to be consistent. So if you choose to indent with four spaces, you need to use four spaces everywhere in your code.

Sometimes we don't want a function to simply run and finish. We may want a function to manipulate data we passed it and then return the result to us. This is where the concept of return values comes in handy. We use the return keyword in a function, which tells the function to pass data back. When we call the function, we can store the returned value in a variable. Return values allow our functions to be more flexible and powerful, so they can be reused and called multiple times.

Functions can even return multiple values. Just don't forget to store all returned values in variables! You could also have a function return nothing, in which case the function simply exits.

In Python, we can use comparison operators to compare values. When a comparison is made, Python returns a boolean result, or simply a True or False.

- To check if two values are the same, we can use the equality operator: **==**
- To check if two values are not the same, we can use the not equals operator: **!=**

We can also check if values are greater than or lesser than each other using **>** and **<**. If you try to compare data types that aren't compatible, like checking if a string is greater than an integer, Python will throw a **TypeError**.

We can make very complex comparisons by joining statements together using logical operators with our comparison operators. These logical operators are **and**, **or**, and **not**. When using the **and** operator, both sides of the statement being evaluated must be true for the whole statement to be true. When using the **or** operator, if either side of the comparison is true, then the whole statement is true. Lastly, the **not** operator simply inverts the value of the statement immediately following it. So if a statement evaluates to True, and we put the **not** operator in front of it, it would become False.

We can use the concept of **branching** to have our code alter its execution sequence depending on the values of variables. We can use an *if* statement to evaluate a comparison. We start with the *if* keyword, followed by our comparison. We end the line with a colon. The body of the *if* statement is then indented to the right. If the comparison is ***True***, the code inside the *if* body is executed. If the comparison evaluates to ***False,*** then the code block is skipped and will not be run.

We just covered the *if* statement, which executes code if an evaluation is true and skips the code if it's false. But what if we wanted the code to do something different if the evaluation is false? We can do this using the *else* statement. The *else* statement follows an *if* block, and is composed of the keyword *else* followed by a colon. The body of the *else* statement is indented to the right, and will be expected if the above *if* statement doesn't execute.

We also touched on the modulo operator, which is represented by the percent sign: **%**. This operator performs integer division, but only returns the remainder of this division operation. If we're dividing 5 by 2, the quotient is 2, and the remainder is 1. Two 2s can go into 5, leaving 1 left over. So 5%2 would return 1. Dividing 10 by 5 would give us a quotient of 2 with no remainder, since 5 can go into 10 twice with nothing left over. In this case, 10%2 would return 0, as there is no remainder.

Building off of the *if* and *else* blocks, which allow us to branch our code depending on the evaluation of one statement, the *elif* statement allows us even more comparisons to perform more complex branching. Very similar to the *if* statements, an *elif* statement starts with the *elif* keyword, followed by a comparison to be evaluated. This is followed by a colon, and then the code block on the next line, indented to the right. An *elif* statement must follow an *if* statement, and will only be evaluated if

the *if* statement was evaluated as false. You can include multiple *elif* statements to build complex branching in your code to do all kinds of powerful things!