**Explore: If, elif, and else statements**

In this module you’ll learn about different data types in Python, how to identify them, and how to convert between them. You’ll also learn how to use variables to assign data and to reference variables. You’ll deep dive into functions: how to define them, pass them parameters, and have them return information. You’ll explore the concepts of code reuse, code style, and refactoring complex code, along with effectively using code comments. Finally, you’ll learn about comparing data using equality and logical operators, and leveraging these to build complex branching scripts using if statements.

**Learning Objectives**

* Differentiate and convert between different data types utilizing variables
* Define and call functions utilizing parameters and return data
* Refactor code and write comments to reduce complexity and enhance code readability and code reuse
* Compare values using equality operators and logical operators
* Build complex branching scripts utilizing if, else and elif statements

**Expressions and Variables**

### 

### Basic Python Syntax introduction

Hi there, welcome back, and well done for completing your first graded assessment. You're doing a great job making it this far. Chances are some topics we've covered may have been a little tricky at times, especially if you're completely new to programming. Don't worry if something wasn't obvious right away. We went through a lot of new concepts and it might take several passes until you feel comfortable with them. And that's totally normal. We all went through it when we were learning how to code. In the previous module we explored some basic concepts, like programming and automation. We called out that each programming language has a specific syntax, which we need to learn so we can tell the computer what to do. We then got a sneak preview of some of the things we could do with Python. Up next, we'll dive deeper into some basic building blocks of Python syntax, things like variables, expressions, functions, and conditional blocks. At first glance these pieces may seem pretty simple, but when we start to combine them they become a lot more powerful. Understanding a programming language's syntax isn't too different from learning a spoken language. For example, the best way to learn Spanish is to visit a Spanish speaking country, immerse yourself in the culture, listen to the people. Then figure out how to arrange the words to form a sentence that another speaker can understand. The same is true for programming. When you immerse yourself in Python programming you'll learn how to formulate statements of code that the computer can understand. This is called syntax. Okay, so as you go through the next few videos keep in mind that our main goal is to learn the language's syntax. So we'll focus on how to tell our computer what to do, not on how to get it to do complicated tasks. Like before, we'll run through some simple exercises to help you see the concepts in action. And as you pick up the new skills and get to grips with different tools we'll start to write more advanced scripts that tackle more challenging problems. Again, if at any point you feel confused or that something just isn't clear, remember you can watch the videos and take the practice quizzes as many times as you need. The key to getting good at programming is practice, practice and practice. You have to keep working your programming muscles in order to get strong, just like building muscles in the gym. Train hard, train regularly, and you'll be tackling more weighty coding problems in no time. All right, ready to jump back in? In the next video we're going to learn all about data types. Let's get started.

### Explore Python syntax

Python is a flexible programming language used in a wide range of fields, including software development, machine learning, and data analysis. Python is one of the most popular programming languages for data professionals, so getting familiar with its fundamental syntax and semantics will be useful for your future career. In this reading, you will learn about Python’s syntax and semantics, as well as where to find resources to further your learning.

The Language of Python

People use language to communicate and give instructions to each other. Computers do the same thing, except computers use languages like Python, C++, and Java. So, in order to communicate instructions to the computer, programmers need to arrange ideas and concepts into a language it will understand.

Python syntax includes words that represent objects and commands, as well as punctuation that gives the words structure, hierarchy, and context. Together, the words and punctuation communicate ideas and processes; this is known as semantics. Semantics is the meaning conveyed by the syntax. The best way to learn syntax and semantics is through exposure. Practice coding and become familiar and comfortable with reading other people’s code. In addition, there are some general conventions that practitioners use to help maintain stylistic uniformity within the language.

Coding languages are similar to spoken languages in that they have a way to classify words according to their function. For example, English sentences are composed of nouns, verbs, prepositions, etc. Here are some of the basics:

* **Variables:** Represent data stored as strings, tuples, dictionaries, lists, and objects (note: future readings explain these categories)
* **Keywords:** Special words that are reserved for specific purposes and that can only be used for those purposes

in

not

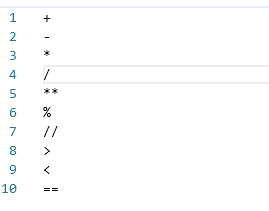
or

for

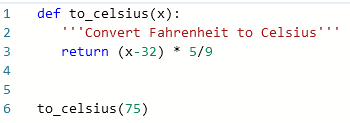
while

return

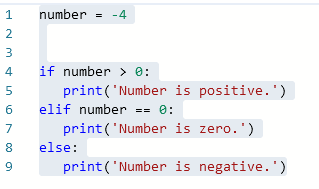
* **Operators:** Symbols that perform operations on objects and values



* **Expressions:** A combination of numbers, symbols, and variables to compute and return a result upon evaluation
* **Functions:** A group of related statements to perform a task and return a value



* **Conditional statements:** Sections of code that direct program execution based on specified conditions



As you’ll surely discover, Python generates syntax errors for incorrectly used keywords and syntax.

Example:

print(This will throw an error because I didn’t make it a string.)

* **Naming rules and conventions**When assigning names to objects, programmers adhere to a set of rules and conventions which help to standardize code and make it more accessible to everyone. Here are some naming rules and conventions that you should know:
  + Names cannot contain spaces.
  + Names may be a mixture of upper and lower case characters.
  + Names can’t start with a number but may contain numbers after the first character.
  + Variable names and function names should be written in snake\_case, which means that all letters are lowercase and words are separated using an underscore.
  + Descriptive names are better than cryptic abbreviations because they help other programmers (and you) read and interpret your code. For example, student\_name is better than sn. It may feel excessive when you write it, but when you return to your code you’ll find it much easier to understand.
* Tim Peters, a Python programmer, wrote this now-famous “poem” of guiding principles for coding in Python:  
  **The Zen of Python**Beautiful is better than ugly.  
  Explicit is better than implicit.  
  Simple is better than complex.  
  Complex is better than complicated.  
  Flat is better than nested.  
  Sparse is better than dense.  
  Readability counts.  
  Special cases aren't special enough to break the rules.  
  Although practicality beats purity.  
  Errors should never pass silently.  
  Unless explicitly silenced.  
  In the face of ambiguity, refuse the temptation to guess.  
  There should be one—and preferably only one—obvious way to do it.  
  Although that way may not be obvious at first unless you're Dutch.  
  Now is better than never.  
  Although never is often better than \*right\* now.  
  If the implementation is hard to explain, it's a bad idea.  
  If the implementation is easy to explain, it may be a good idea.  
  Namespaces are one honking great idea -- let's do more of those!  
  Finally, it’s helpful to bookmark the [PEP 8 Style Guide for Python](https://peps.python.org/pep-0008/) so you can reference it as needed. This reading is limited in scope, and PEP 8 is a more exhaustive resource for style-related matters. PEP stands for Python Enhancement Proposals. These are a running catalog of ways to improve or standardize Python as a language. Because Python is open source, PEP offers a framework to guide developers and build consensus around ideas. It’s a useful and trusted resource.   
  **Key takeaways**Syntax and semantics are what give form and meaning to a language, including Python. A large part of learning a new language is familiarizing yourself with its syntax and semantics. Much of this comes through exposure and practice, but there are a few guiding principles and resources that can help you along the way. If you learn the rules about naming objects and build a bank of resources that you can reference for guidance, you’ll surely make progress as a Python learner. As you get more familiar with Python, you’ll be able to communicate more efficiently with computers and do more with your data analysis tools!  
  **Resources for more information**Here are a few useful resources to help you get more familiar with Python:
  + Python [Reference Library](https://docs.python.org/3/library/)
  + [Built-in Data types](https://docs.python.org/3/library/stdtypes.html)
* [Built-in functions](https://docs.python.org/3/library/functions.html#built-in-functions)
* [Python operators](https://python-reference.readthedocs.io/en/latest/docs/operators/index.html)

### Review: Data types

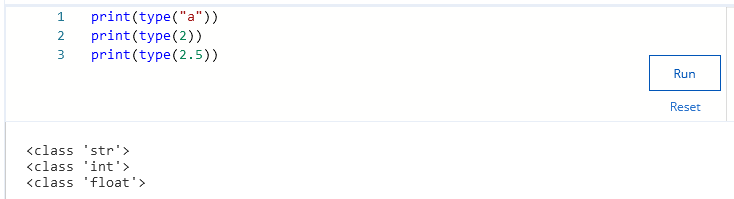
This reading contains the code used in the instructional videos from [**Data Types**](https://www.coursera.org/learn/python-crash-course/lecture/l2J1I/data-types)

**Introduction**

This follow-along reading is organized to match the content in the video that follows. It contains the same code shown in the next video. These code blocks will provide you with the opportunity to see how the code is written, allow you to practice running it, and can be used as a reference to refer back to.

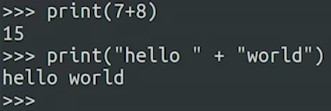
You can follow along in the reading as the instructor discusses the code or review the code after watching the video.



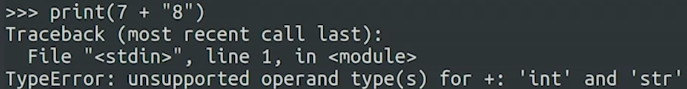


### Data Types

In earlier videos, we called out that text written between quotes and Python is called a string. In programming terminology, a string is known as a data type. Whether it's a mobile game or a script used to automatically create user accounts, most programs need to manipulate some kind of data and that data can come in a lot of different forms or, like we call them, data types. A string is only one kind of data type found in Python. There's a bunch of others, like an integer, which represents whole numbers without a fraction, like one. And float, which represents real numbers or, in other words, a number with a fractional part, like 2.5. Generally, your computer doesn't know how to mix different data types. For example, adding two integers together makes perfect sense to computers, like this. Adding together two strings also makes sense. We just end up with the longer strings that contains the two, like so.

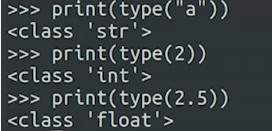


But your computer doesn't know how to add an integer and a string. If you tell it to mix these two different data types, your computer isn't gonna know what to do and will raise an error, check it out.

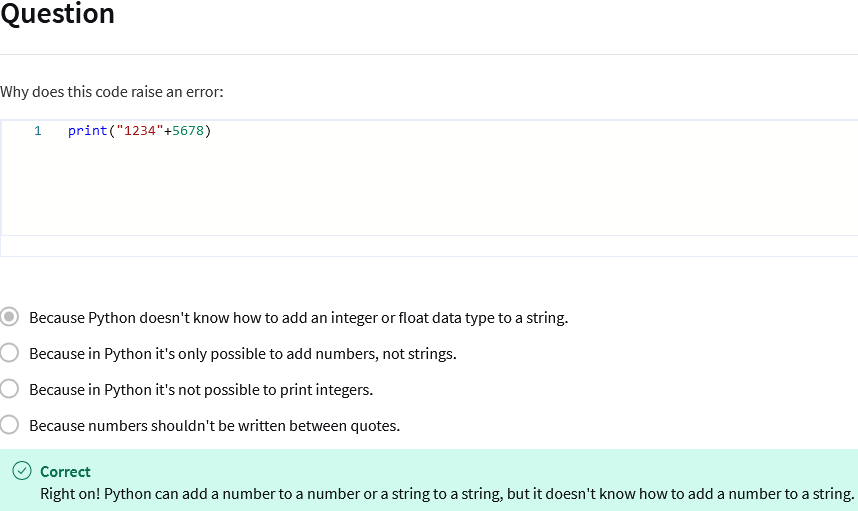


Oh no! Our first error! But don't panic. Errors are a common part of programming and you'll probably have to deal with them a lot. The trick is to think of errors as little clues from your computer to help you improve your programming skills. Read the errors carefully, understand what they're telling you, and then use that new knowledge to help you fix the mistake. In this example, the last line of the error message shows us that we've encountered something called a "type error." When we get a bit of explanatory text that tells us that the plus sign can't be used between an INT type and an STR type, which are short names for integer and string. Thinking about what we've already learned about strings, integers, and mixing data types, can you guess what the error is trying to tell us? The message unsupported operand type tells us that we can't add the integer seven and the string eight because they're different data types. But what if you didn't have an instructor to helpfully point that out? How would you know? You'd need to use your research skills and the resources we called out earlier in the course to do some investigating. For example, you could look for information about the error by pasting the type error message into the search bar of your favorite search engine. This is a common trick used by almost everyone learning to code, and even by experienced developers. You'll usually find that other people on the internet have reported similar errors and solved them, too.

Back to our example, maybe you're thinking, "Aren't we adding two numbers here?" Looks a bit like it, right? Well, look carefully and remember that anything wrapped in quotation marks is considered a string in Python. So eight is a string here while seven is an integer. To the computer, adding seven plus eight is just as strange as adding seven plus A is to us. And seven plus A equals no sense at all. It might be helpful to think about data types in terms of information they can represent. For example, the name of a file would be represented as a string data type, while the size of that file might be an integer data type. If you're ever not 100% sure what data type a certain value is, Python gives you a handy way to find out. You can use the type function to have the computer tell you the type. This might come in handy when dealing with code that someone else wrote and you're not sure what data type it's using. For example…



...pretty neat, right? This tells us that A belongs to STR class, which, like we said earlier, is short for string. The number two belongs to the INT class, which is short for integer, and 2.5 belongs to the float class. We'll talk more about what we mean by class later in the course. For now, you can just use it as a synonym for data type. So now you know three very common data types in Python. There are plenty of others you'll be using soon, but don't worry about them at the moment. As we continue through the course, we'll come across more data types and learn how to interact with each of them. For now, just remember mixing your data types will get your computer, well, all mixed up. So keep your strings with your strings, your integers with your integers, and your floats with your floats, and you shouldn't get in too much of a tangle.



### Categorize: Naming variables

(Note) Provided a quick example of variable names that were effective or ineffective, such as giving example of a function as a variable name and using numbers to begin a variable name with

### Annotating variables by type

Type annotation allows you to clearly communicate the argument types and return type of functions in your code. It’s like giving yourself and other developers hints about what kind of data the variable is supposed to hold. This has several benefits: It reduces the chance of common mistakes, helps in documenting your code for others to reuse, and allows integrated development software (IDEs) and other tools to give you better feedback.

In this reading, you will learn more about annotating variables by type and best practices.

**How to annotate a variable**

Think of annotating a variable as if you were to put a label on a container—and anything in that container should hold what the label is describing. Let’s take a look at an example:

name: str = “Betty”

The variable name is declared using a colon (:) which is annotated with the type *str*, indicating that the name variable should hold a string value. And look, it does! Betty—or any name for that matter—is a string, and we know it’s a string because it is in quotes. Let’s look at another example where a variable holds an integer value.

*age: int = 34*

In this example, *age* is the variable, and *int* is the type annotation that provides you and other developers a hint that the age variable should store an integer value.

**Pro tip:** If a function expects a list of integers, you should annotate it as *List[int]*, not just *List*. Being specific with your types can catch more potential bugs and misunderstandings.

**Dynamic typing**

Many languages, such as C# or Java, require you to declare variable types, but not Python. One of the great things about Python is that the type of variable can change over time as new values are assigned to it. For example:

*a = 3 #a is an integer*

*a = “Hello world” #a is now a string*

Dynamic typing allows programmers to write code more quickly and offers flexibility because you don’t have to explicitly declare the type of variable.

**Note:** Python decides which of the built-in types the variable is and, therefore, how it should behave. For more information, refer to this article on [Built-in types](https://docs.python.org/3/library/stdtypes.html)

.

**Duck typing**

This form of typing comes from the saying, “If it walks like a duck and quacks like a duck, it must be a duck.” Python will infer the variable type at runtime and decide which behaviors are available to the given object.

*a = “Hello world” #looks like a string*

**Annotating variables with type comments**

Another way to annotate variables is to use type comments where the interpreter will ignore the comments.

*captain = “Picard” # type: str*

**Note:** This way of annotating variables might be useful for cases when you need to know what types belong to which variables but do not want the overhead of using a line interpreter (linter) or IDE on this specific variable.

**Annotating variables directly**

Let’s use the same example above to annotate a variable directly.

*captain: str = “Picard”*

**Note:** You might hear annotating variables directly called the more “modern” way to annotate a variable.

Another advantage is that you can use automated tools such as linters, or mypy, to check types to make code more resilient. Most modern IDEs, such as VS Code and JetBrains PyCharm, scan code for type annotations and can use it to help you write better code more quickly!

**How type annotations affect runtime behavior**

Any time a library is called, or an IDE works to scan your code, more computational overhead is required.

**Pro tip:** Be strategic when annotating variables by type. This can add unnecessary overhead when overused.

Type annotation is less common with Python users in data science, as it can be burdensome to manually map data every time a new set of data comes in. On the other hand, when doing object-oriented programming or writing functions, using type annotations becomes extremely important because it helps clarify code since you are dealing with more than just the built-in types.

**Key takeaways**

Annotating variables by type provides programmers with benefits to make the code easier to read and understand. Python provides different options on how to annotate variables, so choose how you want to annotate them. Just be cautious of over-annotating, creating unnecessary overhead to your code.

**Resources for more information**

* Python built-in types<https://docs.python.org/3/library/stdtypes.html>

### Review: Expressions, numbers, and type conversions

This reading contains the code used in the instructional videos from [**Expressions, numbers and type conversions**](https://www.coursera.org/learn/python-crash-course/lecture/dKIr2/expressions-numbers-and-type-conversions)

**Introduction**

This follow-along reading is organized to match the content in the video that follows. It contains the same code shown in the next video. These code blocks will provide you with the opportunity to see how the code is written, allow you to practice running it, and can be used as a reference to refer back to.

You can follow along in the reading as the instructor discusses the code or review the code after watching the video.



### Expressions, numbers, and type conversions

In an earlier video, we saw how we can't use the plus operator between an integer and a string because they're different data types. But what happens when we try to operate with an integer and a float instead? Let's find out.



Python has no problem performing this operation. But what's up with that? Aren't integer and a float two different data types? They sure are. But there's a lot happening under the hood here. Behind the scenes, the computer is busy automatically converting our integer seven into a float seven. This lets Python then add together the values to return a result that is also a float. We call this process

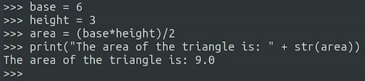
**implicit conversion. The interpreter automatically converts one data type into another.**

We've called this out before, but it's worth highlighting again that Python operations aren't just restricted to numbers. You can also use the plus operator to add together strings.

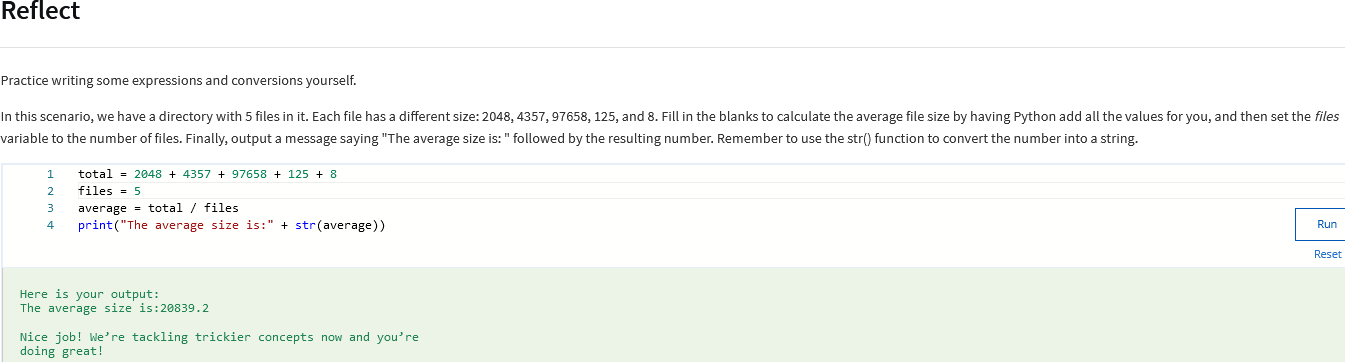


This lets you do things like create sentences from individual words. Just don't forget to add spaces to each word. Otherwise, the computer will run them all together.

So what if you really want to combine a string and a number? Is it possible? It sure is, but only with an **explicit conversion**. In Python to convert between one data type and another, we call a function with the name of the type we're converting to. Let's see how this works.



Now things are getting a little bit more complex. Let's take a moment to unpack this to make sure it all makes sense. In this script, we're first calculating the area of a triangle. And when printing it, we're adding it to a string. To do this, we need to call the str function to convert a number into a string. Let's execute it and check out what happens. Our number got converted to a string and printed together with the message.



We've learned a little bit about variables, values, expressions, and conversions. Next up, we've got a practice quiz to help you solidify your knowledge. As always, take your time and review the content if you need. You've totally got this. I'll see you in the next video once you're finished.

### Implicit vs explicit conversion

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.

### Study guide: Expressions and variables

This study guide provides a quick-reference summary of what you learned in this lesson and serves as a guide for the upcoming practice quiz.

In the Expressions and Variables segment, you learned about expressions, variables, and the data types: string, integer, and float. You learned how to convert a value from one data type to another and you learned how to resolve a few common errors in Python.

**Terms**

* **expression** - a combination of numbers, symbols, or other values that produce a result when evaluated
* **data types** - classes of data (e.g., string, int, float, Boolean, etc.), which include the properties and behaviors of instances of the data type (variables)
* **variable** - an instance of a data type class, represented by a unique name within the code, that stores changeable values of the specific data type
* **implicit conversion** - when the Python interpreter automatically converts one data type to another
* **explicit conversion** - when code is written to manually convert one data type to another using a data type conversion function:
  + *str()* - converts a value (often numeric) to a **string** data type
  + *int()* - converts a value (usually a float) to an **integer** data type
  + *float()* - converts a value (usually an integer) to a **float** data type

**Variables Annotated by Type**

Type annotations are optional in Python. They can be very helpful, though, because they make code easier to read. Annotations make the variable types clear to those reading the code. They can also help you catch errors during compilation. In the example below, we are using the typing module to annotate the different types of variables.

import typing

# Define a variable of type str

z: str = "Hello, world!"

# Define a variable of type int

x: int = 10

# Define a variable of type float

y: float = 1.23

# Define a variable of type list

list\_of\_numbers: typing.List[int] = [1, 2, 3]

# Define a variable of type tuple

tuple\_of\_numbers: typing.Tuple[int, int, int] = (1, 2, 3)

# Define a variable of type dict

dictionary: typing.Dict[str, int] = {"key1": 1, "key2": 2}

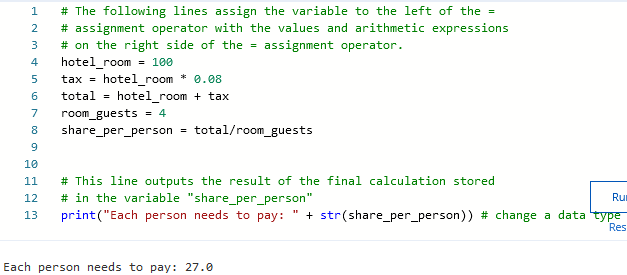
# Define a variable of type set

set\_of\_numbers: typing.Set[int] = {1, 2, 3}

**Coding skills**

**Skill Group 1**

* Use variables to store values
* Use basic arithmetic operators with variables to create expressions
* Use explicit conversion to change a data type from float to string



**Skill Group 2**

* Output multiple string variables on a single line to form a sentence
* Use the plus (+) connector or a comma to connect strings in a *print()* function
* Create spaces between variables in a *print()* function

# The following 5 lines assign strings to a list of variables.

salutation = "Dr."

first\_name = "Prisha"

middle\_name = "Jai"

last\_name = "Agarwal"

suffix = "Ph.D."

print(salutation + " " + first\_name + " " + middle\_name + " " + last\_name + ", " + suffix)

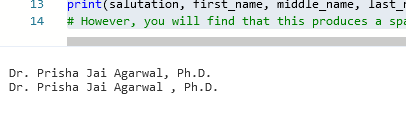
# The comma as a string ", " adds the conventional use of a comma plus a

# space to separate the last name from the suffix.

# Alternatively, you could use commas in place of the + connector:

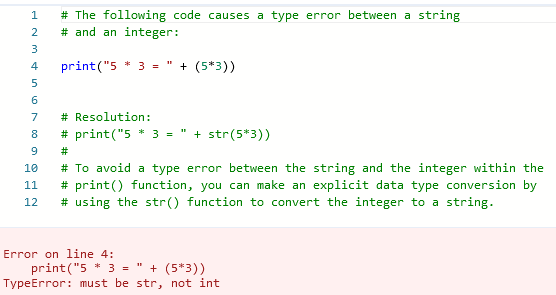
print(salutation, first\_name, middle\_name, last\_name,",", suffix)

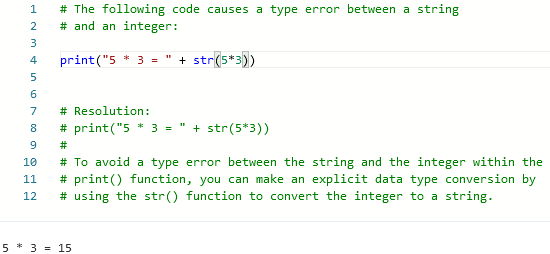
# However, you will find that this produces a space before a comma within a string.



**Skill Group 3**

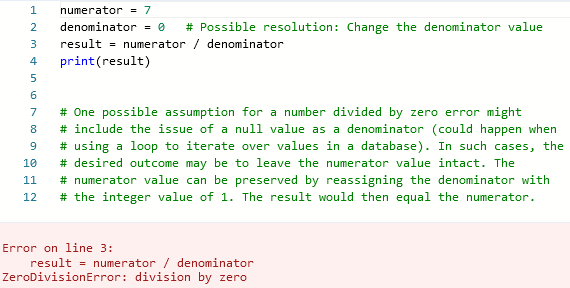
* Resolve TypeError caused by a data type mismatch issue
* Use an explicit conversion function

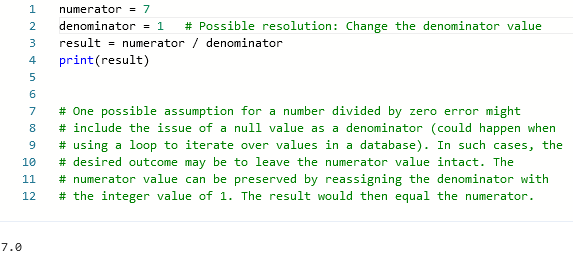




**Skill Group 4**

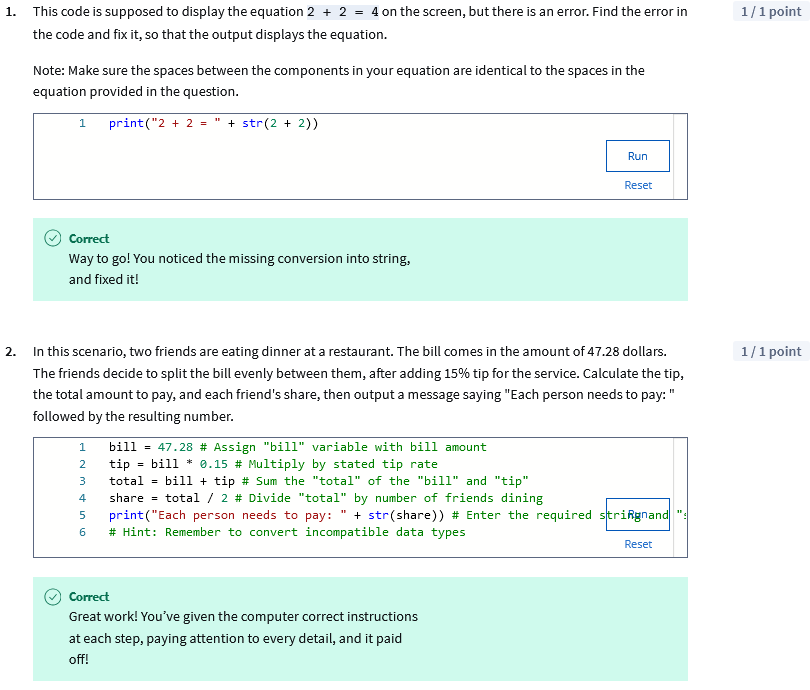
* Resolve a ZeroDivisionError caused by an attempt to divide by 0

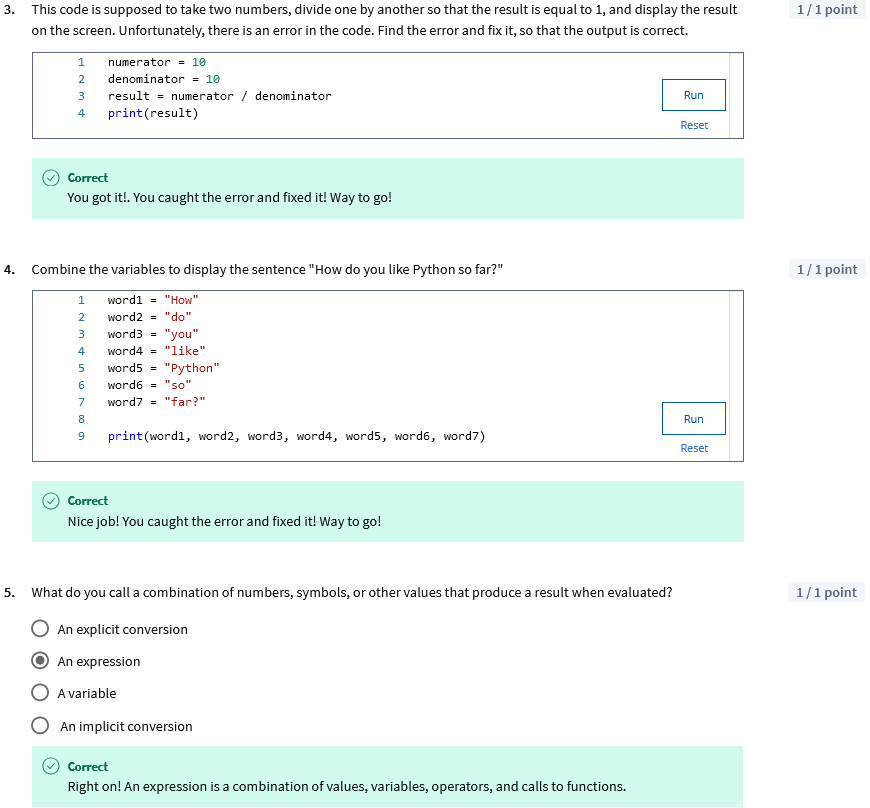




* **Python practice information**For additional Python practice, the following links will take you to several popular online interpreters and codepads:
  + [Welcome to Python](https://www.python.org/shell/)
* [Online Python Interpreter](https://www.onlinegdb.com/online_python_interpreter)
* [Create a new Repl](https://repl.it/languages/python3)
* [Online Python-3 Compiler (Interpreter)](https://www.tutorialspoint.com/execute_python3_online.php)
* [Compile Python 3 Online](https://rextester.com/l/python3_online_compiler)
* [Your Python Trinket](https://trinket.io/python3)

### Practice quiz: Expressions and variables





## **Functions**

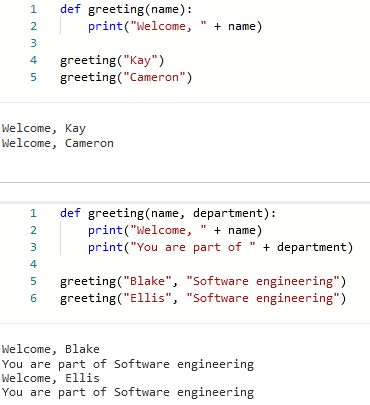
### Review: Defining functions

This reading contains the code used in the instructional videos from [**Defining functions**](https://www.coursera.org/learn/python-crash-course/lecture/Jzoz1/defining-functions)

**Introduction**

This follow-along reading is organized to match the content in the video that follows. It contains the same code shown in the next video. These code blocks will provide you with the opportunity to see how the code is written, allow you to practice running it, and can be used as a reference to refer back to.

You can follow along in the reading as the instructor discusses the code or review the code after watching the video.



### Defining Functions

So far, we've been looking at variables, expressions and operations, which are the smallest components of scripts. Up next, we're gonna look at functions, which are another crucial programming building block. We've come across a few Python functions in our examples so far. The print function that writes text on the screen, the type function, which tells us the type of a certain value, and the STR function, which converts a number into a string. All those functions come as a part of the language, and we'll look into a bunch of other built-in Python functions throughout this course.

But now we're going to see how to define our own functions to tell the computer to do things that the languages built-in functions don't. Let's start with a simple example.



In this piece of code, we're defining a function. Our function takes the parameter. Here, that parameter is name, and prints a greeting for that name. This snippet is small, but it already shows a lot of important points about how we define functions in Python. Let's go through this step by step. To define a function, we use the def keyword. The name of the function is what comes after the keyword. In this example, the function's name is greeting. So to call the function later in the script, we'll use the word greeting.

After the name, we have the parameters of the function, which are written between parentheses. In this example, we only have one parameter, name, name followed by a colon at the end of the line. After the colon, we have the body of the function. That's where we state what we want our function to do. Note how the body is indented to the right. This is a key characteristic of Python, and we'll come across it a bunch. For now, just keep in mind that the body of the function must be to the right of the definition. In this example, the body contains just one line that calls the print function. Looks simple, right? But creating functions can actually be super powerful. The body of a function can have as many lines as we want it to and do all sorts of fun stuff. We'll find out exactly what in later videos, but for now, let's execute our function and see what happens.



That's nice, but it's not too interesting yet. Let's make it do a little more.



Our function now receives two parameters instead of one, name and department, and it writes two separate messages. Again, notice the indentation. We can add as many lines as we'd like to to the body of the function, but each line must be indented the same number of spaces to the right. In this example, we're using four spaces. We could use two or eight or any other number as long as they're all consistent. Let's try calling our new and improved greeting function.



That's more useful. And we're only just scratching the surface of what we can do with functions.

Remember that these are just simple examples, but a function can do a lot more than just print messages. In this course and throughout the upcoming courses, we'll explore a bunch of other tasks that we can do with Python, and usually we'll write them inside functions.

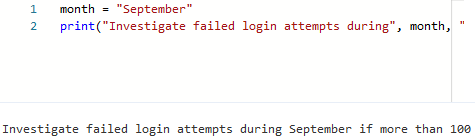
### Built-in functions

In the last video, we explored built-in functions in Python, including *print()*, *type()*, and *str()*. **Built-in functions** are functions that exist within Python and can be called directly. In this reading, you’ll explore these further and also learn about the sorted() function and *max()*and *min()* function. In addition, you'll review how to pass the output of one function into another function.

**print()**

The *print()* function outputs a specified object to the screen. The *print()* function is one of the most commonly used functions in Python because it allows you to output any detail from your code.

To use the *print()* function, you pass the object you want to print as an argument to the function. The *print()* function takes in any number of arguments, separated by a comma, and prints all of them. For example, you can run the following code that prints a string, a variable, another string, and an integer together:



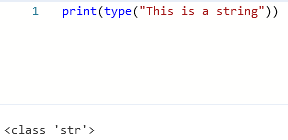
**type()**

The *type()* function returns the data type of its argument. The *type()* function helps you keep track of the data types of variables to avoid errors throughout your code.

To use it, you pass the object as an argument, and it returns its data type. It only accepts one argument. For example, you could specify *type("security")* or *type(7)*.

**Passing one function into another**

When working with functions, you often need to pass them through print() if you want to output the data type to the screen. This is the case when using a function like type(). Consider the following code:

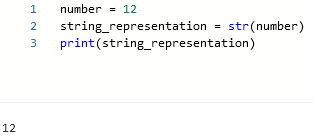


It displays *str()*, which means that the argument passed to the *type()* function is a string. This happens because the *type()* function is processed first and its output is passed as an argument to the *print()* function.

**str()**

The *str()* function can be used to convert any data type to a string. The *str()*function takes a single argument, which is the value that you want to convert to a string. The *str()* function will then return a string representation of the value.

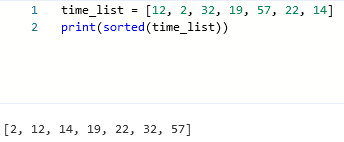
In this example, the *str()* function will convert the number 12 to a string. This will run the code and print the string representation of the number.



**sorted()**

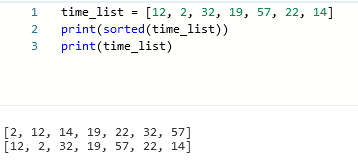
The *sorted()* function sorts the components of a list. The *sorted()* function also works on any iterable, like a string, and returns the sorted elements in a list. By default, it sorts them in ascending order. When given an iterable that contains numbers, it sorts them from smallest to largest; this includes iterables that contain numeric data as well as iterables that contain string data beginning with numbers. An iterable that contains strings that begin with alphabetic characters will be sorted alphabetically.

The *sorted()* function takes an iterable, like a list or a string, as an input. So, for example, you can use the following code to sort the list of login sessions from shortest to longest:



This displays the sorted list.

The *sorted()* function does not change the iterable that it sorts. The following code illustrates this:



The first *print()* function displays the sorted list. However, the second *print()* function, which does not include the *sorted()* function, displays the list as assigned to time\_list in the first line of code.

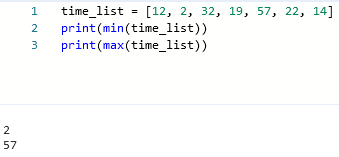
One more important detail about the sorted() function is that it cannot take lists or strings that have elements of more than one data type. For example, you can’t use the list *[1, 2, "hello"]*.

**max() and min()**

The *max()* function returns the largest numeric input passed into it. The *min()* function returns the smallest numeric input passed into it.

The *max()* and *min()* functions accept arguments of either multiple numeric values or of an iterable like a list, and they return the largest or smallest value respectively.

For example, you could use these functions to identify the longest or shortest session that a user logged in for. If a specific user logged in seven times during a week, and you stored their access times in minutes in a list, you can use the *max()* and *min()* functions to find and print their longest and shortest sessions:



**Key takeaways**

Built-in functions are powerful tools in Python that allow you to perform tasks with one simple command. The print() function prints its arguments to the screen, the type() function returns the data type of its argument, the sorted() organizes its argument, and the min() and max() functions return the smallest and largest values of an iterable respectively.

**Resources for more information**

These were just a few of Python's built-in functions. You can continue learning about others on your own:

* [The Python Standard Library documentation](https://docs.python.org/3/library/functions.html) : A list of Python’s built-in functions and information on how to use them

### Review: Returning values

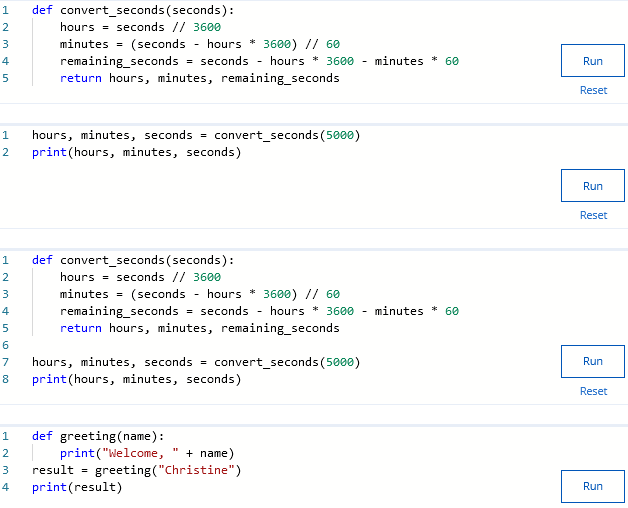
This reading contains the code used in the instructional videos from [**Returning values**](https://www.coursera.org/learn/python-crash-course/lecture/1B4o4/returning-values)

**Introduction**

This follow-along reading is organized to match the content in the video that follows. It contains the same code shown in the next video. These code blocks will provide you with the opportunity to see how the code is written, allow you to practice running it, and can be used as a reference to refer back to.

You can follow along in the reading as the instructor discusses the code or review the code after watching the video.





### Returning Values

We've seen how we can pass values into a function as parameters by passing values like the name or department in the example earlier. But what about getting values out of a function?

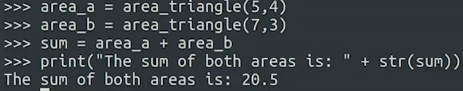
This is where the concept of return values comes into play. The work that functions do can produce new results. Sure, we can print the results on the screen, but what if we wanted to use those results later in our script or didn't want to print them at all? We can do this by returning values from the functions we define ourselves.

Let's go back to calculating the area of a triangle. Do you remember our triangle example from our earlier exercise? The area of the triangle is calculated as

area = (base\*height)/2

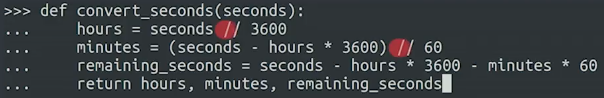
Imagine we need to calculate this value several times in our code. It would be useful to have a function that does this for us. Check out how this would look.

We use the keyword return to tell Python that this is the return value of a function. When we call the function, we store that value in a variable. Let's say we have the two triangles and we want to add up the sum of both areas. Here's what we would do. First, we calculate the two areas separately. Then, we add the sum of both areas together. Finally, we print the result, converting it to a string.



As you can see in this example, the area\_triangle function returns a value which is not surprisingly the area of the triangle. We store that value in a different variable for each call to the function. In this case, area\_a and area\_b. Then we operate with those values adding them into the variable called sum and only printing this final result. This shows the power of the return statement. It allows us to combine calls to functions and to more complex operations which makes your code more reusable.

Return statements in Python are even more interesting because we can use them to return more than one value. Let's say you have a duration of time in seconds and you want to convert that to the equivalent number of hours, minutes, and seconds. Here's how to do that in Python.



Did you spot the new operator in this function? That **//** operator is called **floor division**. A floor division divides a number and takes the integer part of the division as the result. For example, five slash slash two is two instead of 2.5.

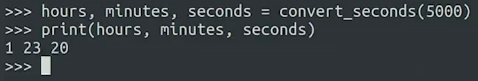
The first operation is calculating how many hours are in a given amount of seconds.

The second works out how many minutes are left once we subtract the hours.

The third shows how many seconds remain after subtracting minutes.

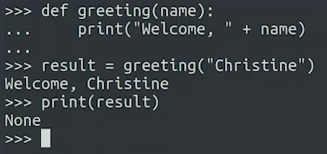
We end up with three numbers as a result. So the function returns all three of them.

Let's see what this looks like when we're calling a function. Because we know that the function returns three values, we assign the result of the function to three different variables.

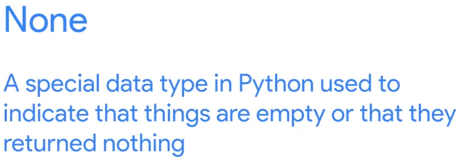


There's one last thing we should call out about returning values. It is possible to return nothing.

Let's look at an example from an earlier video. Here the function just printed a message and didn't return anything. What do you think would happen if we try to assign the value of this function to a variable? Let's try it out and see.



Here when we called the function, it printed a message just like we expected. We stored the return value in the result variable, but there was no return statement in the function. So the value of results is none.



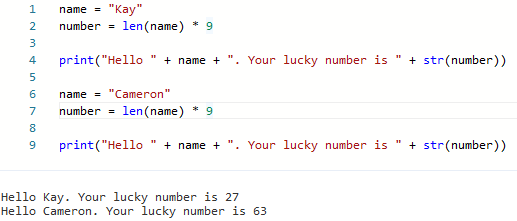
None is a very special data type in Python used to indicate that things are empty or that they return nothing. Functions and return values can be tricky concepts to master, but they let us do a bunch of cool stuff. So put the time and effort in to learn it for some really valuable returns.

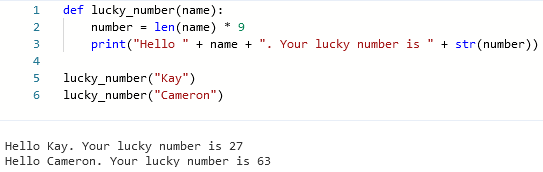
### Review: The principles of code reuse

**Introduction**

This follow-along reading is organized to match the content in the video that follows. It contains the same code shown in the next video. These code blocks will provide you with the opportunity to see how the code is written, allow you to practice running it, and can be used as a reference to refer back to.

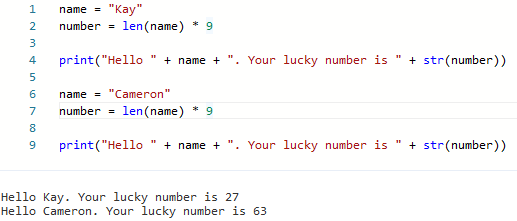
You can follow along in the reading as the instructor discusses the code or review the code after watching the video.





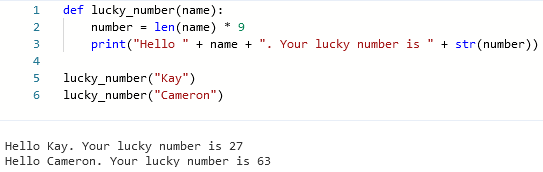
### The principles of code reuse

As we've called out before, functions are powerful because you can create your own. You can use them to organize the code in your scripts into logical blocks, which makes the code you write easier to use and reuse. Check out this example.



This script uses the len function, which returns the length of a string. In this example the script then uses that length to calculate a number, which we're calling the lucky number here. And finally, it prints a message with the name and the number. Each time you want to perform the calculation, we change the values of the variables and write the formula. Then, print a greeting followed by the lucky number.

See how there are exactly two lines that are the same, in the first and second part of the code. When you find code duplication in your scripts, it's a good idea to check if you can clean things up a bit by using a function. How about we rewrite this code creating a function to group all the duplicated code into just one line.



The updated script gives us the exact same result as the original one, but it looks a lot cleaner.

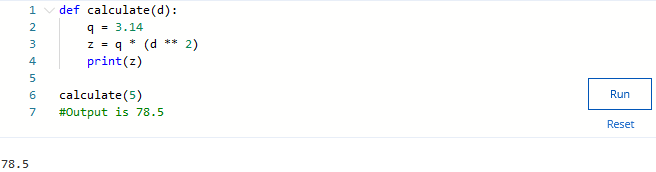
First, we've defined a function called lucky number, which carries out our calculation and prints it for us. Then we call the function twice, once with each name. Since we've grouped the calculation and print statements into a function, our code is not only easier to read but it's also now reusable. We can execute the code inside the lucky number function as many times as we need it, by just calling it with a different name. So we don't have to write it out again and again for each new name.

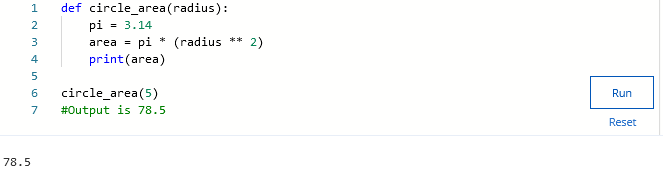
Hopefully, these examples have helped explain how functions are used and defined. And also demonstrated how useful they can be. Did you notice that we're feeding information into our functions through their parameters? This is one of the many ways that we can input data into our code. The values for those parameters may come from different places, like a file on our computer or through a form on a website, but that doesn't impact our code. The result of the function is still the same, no matter where the parameters come from. Functions are your friends. They can help clean up your code and do math so you don't have to. You'll be using them a lot both in this course and in your programming life. So get ready to get real friendly with functions.

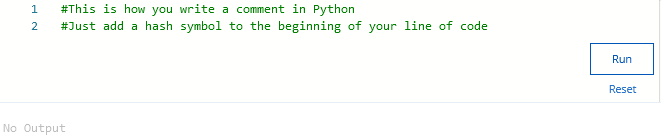
### Review: Code style

This follow-along reading is organized to match the content in the video that follows. It contains the same code shown in the next video. These code blocks will provide you with the opportunity to see how the code is written, allow you to practice running it, and can be used as a reference to refer back to.

You can follow along in the reading as the instructor discusses the code or review the code after watching the video







### Code style

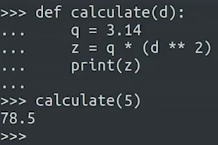
So far, we've looked into how the Python syntax is used for variables, expressions, and defining and using functions.

There's a lot more syntax to come. But before we dive into that, let's talk a bit about a different side of programming: style.

On the whole, having good or bad style when you write code doesn't make much difference between a script succeeding or crashing, but it can make a big difference for the people who use it and contribute to it. Poor programming style can make life difficult for the IT specialists or system administrators who have to read the script after it's written or make changes to it so it works with a new system. Bad style can even give the script's author a headache if it's been awhile since they wrote it. Imagine having to rewrite your own code because it's too messy to understand.

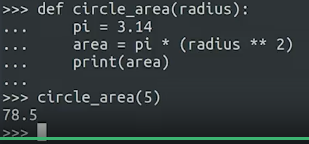
On the flip side, good style can make a script look almost like natural human language. It can make the script's intent and construction immediately clear to the reader. Goods style makes life easier for people who have to maintain the code and helps them understand what it does and how it does it. It can also reduce errors since it makes updating the code easier and more straightforward.

So we agree, our code should be stylish. But what makes the style of a piece of code good or bad? Although there are no hard and fast rules that apply to every programming language and situation, keeping a few principles in mind will go a long way to creating good, well-styled code. First off, you want your code to be as self-documenting as possible. Self-documenting code is written in a way that's readable and doesn't conceal its intent. This principle can be applied to all aspects of writing code from picking your variable names to writing clear concise expressions. Take this code snippet for example.



It's hard to determine the purpose of this code by just looking at it. The names of the variables don't give the reader much information. And, although you can likely work out the result of the calculation, there are no clues to what that result might be used for.

In programming lingo, when we re-write code to be more self-documenting, we call this process refactoring. So how would it look if we refactored this code?



With this refactored code, the intent should now be more clear. The names of the variables and the function reflect their purpose, which helps the reader understand the code more quickly. You should always aim for your code to be self-documenting. But even then, sometimes you may need to use a particularly tricky bit of code in your script. When good naming and clean organization can't make the code clear, you can add a bit of explanatory texts to the code. You do this by adding what we call a comment.

In Python, comments are indicated by the hash character. When your computer sees a hash character, it understands that it should ignore everything that comes after that character on that line. Check out how this looks. Using comments lets you explain why a function does something a certain way. It also allows you to leave notes to your future self or other programmers to remind you of what needs to be improved and why. Obviously, it's much easier to read your own code than someone else's. But in my job, I work on code that was written by lots of different people and everybody designs things a little differently. This is why it's so important to comment and document your code well. More often than not, your code will eventually be used by someone other than you. Use the style guide to structure your code in a way that's readable by others, or by you in six months when you've forgotten why you wrote that code in the first place. In upcoming exercises in this course, we'll use comments to let you know what you need to do with the code. You can always write as many extra comments as you need.

### Study guide: Functions

This study guide provides a quick-reference summary of what you learned in this lesson and serves as a guide for the upcoming practice quiz.

In the Functions segment, you learned how to define and call functions, utilize a function’s parameters, and return data from a function. You also learned how to differentiate and convert between different data types utilizing variables. Plus, you learned a few best practices for writing reusable and readable code.

**Terms**

* **return value** - the value or variable returned as the end result of a function
* **parameter (argument)** - a value passed into a function for use within the function
* **refactoring code** - a process to restructure code without changing functionality

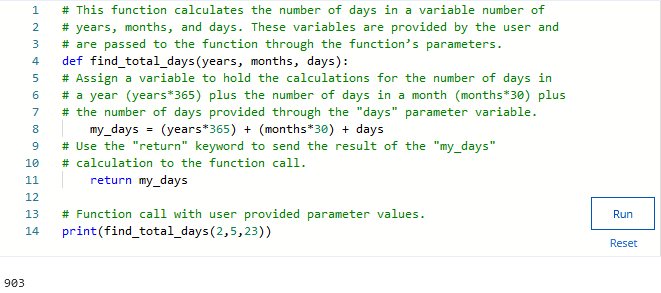
**Knowledge**

* The purpose of the **def()** keywordis to define a new function.
* Best practices for writing code that is readable and reusable:
  + **Create a reusable function** - Replace duplicate code with one reusable function to make the code easier to read and repurpose.
  + **Refactor code** - Update code so that it is self-documenting and the intent of the code is clear.
  + **Add comments** - Adding comments is part of creating self-documenting code. Using comments allows you to leave notes to yourself and/or other programmers to make the purpose of the code clear.

**Coding skills**

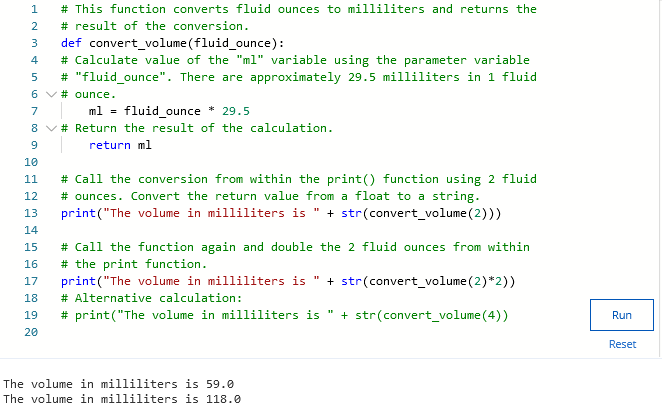
**Skill Group 1**

* Use a function that accepts multiple parameters
* Return a result value



**Skill Group 2**

* Use a function to return the result of a measurement conversion
* Use arithmetic operators to perform a calculation
* Combine text with a function call within a print() statement
* Convert the return value from a float data type to a string for the print() function
* Call the function and perform a calculation on the return value within a print() statement

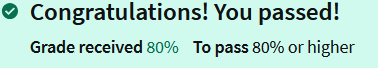


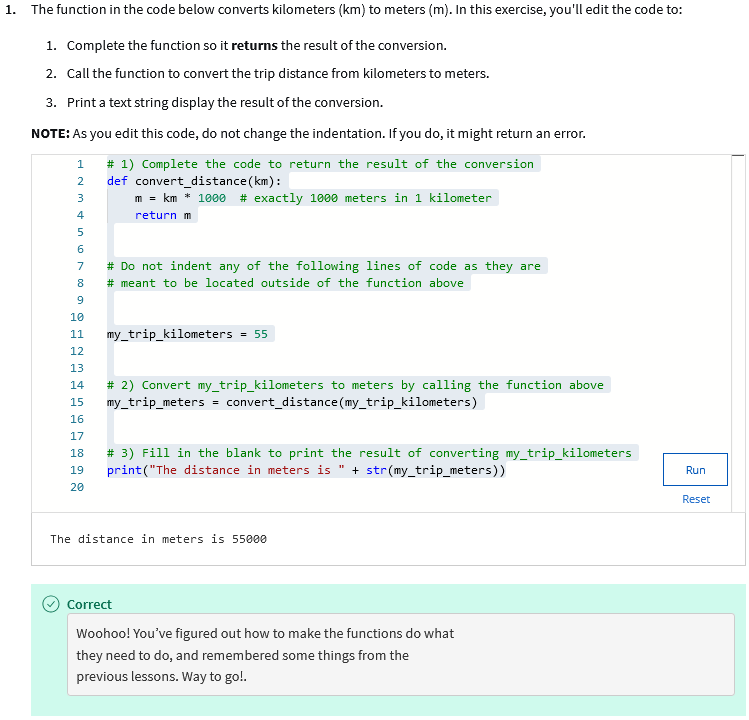
Python practice information

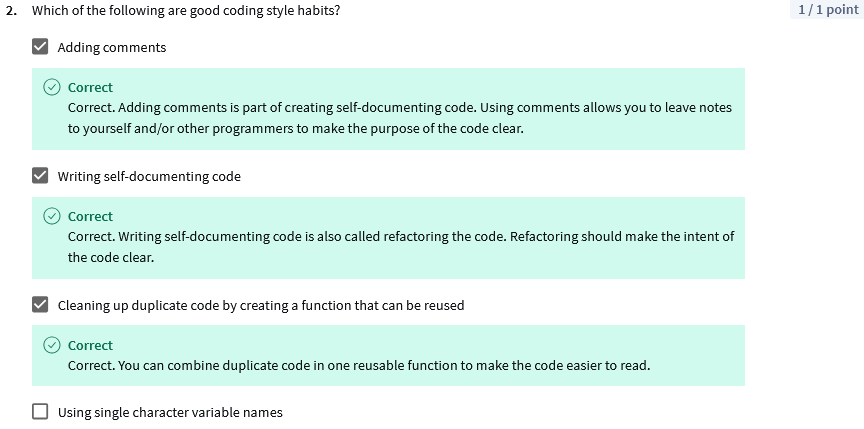
For additional Python practice, the following links will take you to several popular online interpreters and codepads:

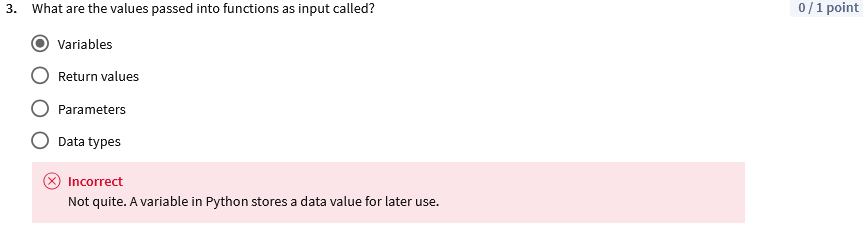
* [Welcome to Python](https://www.python.org/shell/)
* [Online Python Interpreter](https://www.onlinegdb.com/online_python_interpreter)
* [Create a new Repl](https://repl.it/languages/python3)
* [Online Python-3 Compiler (Interpreter)](https://www.tutorialspoint.com/execute_python3_online.php)
* [Compile Python 3 Online](https://rextester.com/l/python3_online_compiler)
* [Your Python Trinket](https://trinket.io/python3)

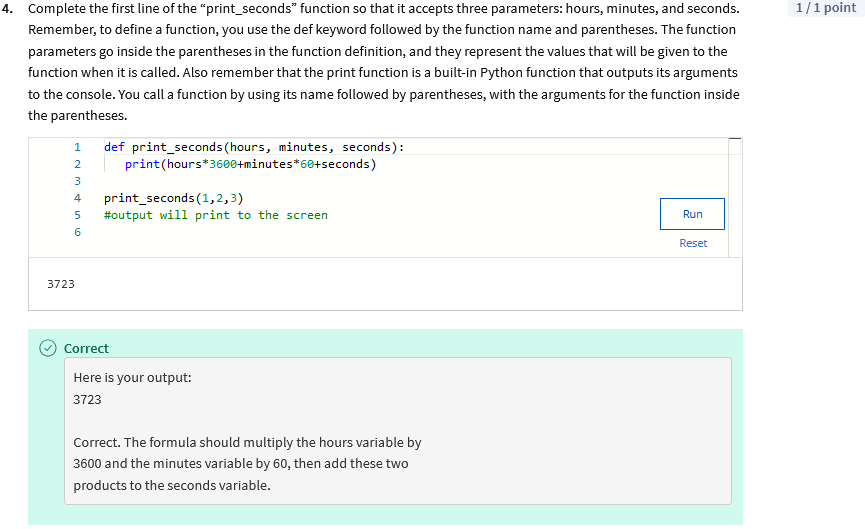
### Practice Quiz: Functions

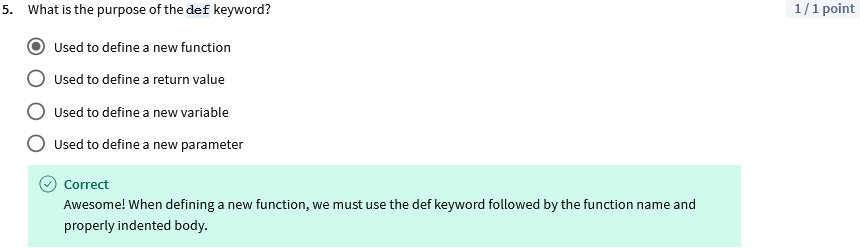












## **Conditionals**

### Review: Comparing things

This reading contains the code used in the instructional videos from [**Comparing things.**](https://www.coursera.org/learn/python-crash-course/lecture/7McMn/comparing-things)

Introduction

This follow-along reading is organized to match the content in the video that follows. It contains the same code shown in the next video. These code blocks will provide you with the opportunity to see how the code is written, allow you to practice running it, and can be used as a reference to refer back to.

You can follow along in the reading as the instructor discusses the code or review the code after watching the video.



### Comparing things

We've seen a few arithmetic expressions so far, like addition, subtraction, and division. Remember when we turned Python into a calculator? Well, Python can also compare values. This lets us check whether something is smaller than, equal to, or bigger than something else. This allows us to take the result of our expressions and use them to make decisions. Check out these three examples.

In the first example 10 is greater than 1. So, the value true is printed as a result.



True is a value that belongs to another data type called the Boolean. Booleans represent one of two possible states, either true or false. Every time you compare things in Python the result is a Boolean of the appropriate value.

In the second example we can see our very first equality operator, which is formed by putting two equal signs == together.



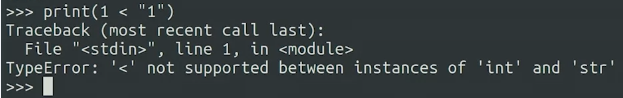
We use this operator to test whether two things are equal to each other. In this example the string cat is not equal to the string dog, so the Boolean that's printed is false.

In our third example we're doing the opposite comparison. By pairing an exclamation mark and an equal sign we're using the not equals operator, which is the negated form of the equality operator.



In this particular line of code the operator checks that 1 isn't equal to 2. We called out before that the plus operator doesn't work between integers and strings.

What do you think will happen if we try to compare an integer and string? Let's find out by seeing if the number 1 is smaller than the string 1.



We get a type error. That's the same error we got before. This happens because Python doesn't know how to check if a number is smaller than a string.

What about the equality operator?



In this case the Interpreter has no problem telling us that the integer 1 and the string 1 aren't the same. So what gives? Basically although they may seem similar to us because they both contain the same number, it's clear to the computer that one is a number and the other is a string. For the computer it's obvious that they are completely different entities.

On top of the comparison and equality operators, Python also has a set of logical operators. These operators allow you to connect multiple statements together and perform more complex comparisons.

In Python the logical operators are the words **and**, **or**, and **not**.

Let's look at some examples.



To evaluate as true the and operator would need both expressions to be true at the same time.

Here we're comparing strings, and the bigger and smaller operators refer to alphabetical order. Yellow comes after cyan, but brown doesn't come after magenta. So this means that the first statement is true, but the second one isn't, which makes the result of the whole expression false.

If we use the or operator, instead, the expression will be true if either of the expressions are true, and false only when both expressions are false.

Let's try it out. 25 is definitely not bigger than 50, but 1 is different than 2. So in the end the whole expression is true.



Last up, the not operator inverts the value of the expression that's in front of it.

If the expression is true, it becomes false. If it's false, it becomes true. Just like this.



Logical operators are important because they help us write more complex expressions.

If this is the first time you've come across these operators it might seem like there's a lot to remember. But don't worry, you'll learn most of them very quickly just by practicing. And in the next reading we have a cheat sheet that lists all the operators available and what each one does. It's a handy resource you're sure to find useful when writing your own scripts.

### Comparison Operators with Equations

The following examples demonstrate how to use comparison operators with the data types **int** (integers, whole numbers) and **float** (number with a decimal point or fractional value). Comparison operators return Boolean results. As you learned previously, Boolean is a data type that can hold only one of two values: **True** or **False**.

The comparison operators include:

* **==** (equality)
* **!=** (not equal to)
* **>** (greater than)
* **<** (less than)
* **>=** (greater than or equal to)
* **<=** (less than or equal to)

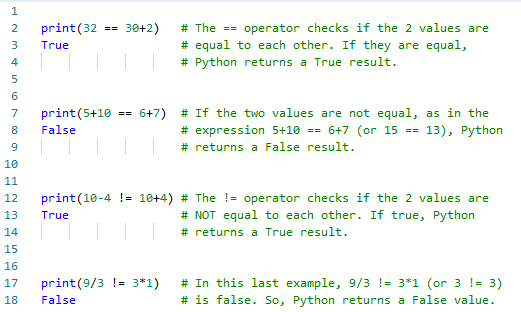
**PART 1: Equality == and Not Equal To != Operators**

In Python, you can use comparison operators to compare values. When a comparison is made, Python returns a Boolean result: **True** or **False**. Note that Boolean data types are not string data types (Boolean **True** is not equal to the string "True").

* To check if two values are the same, use the **equality operator**: **==**
* To check if two values are not the same, use the **not equal to operator**: **!=**

The print() function can be used to display the results of the comparisons.

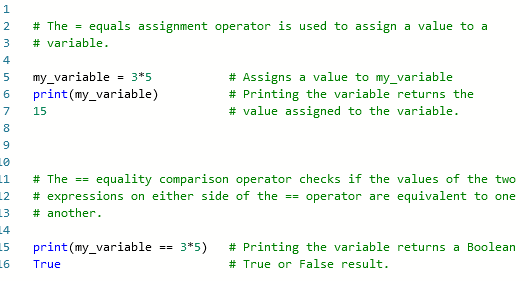
**Examples:**



**The equality == operator versus the equals = operator**

It is important to note that the equality **==** comparison operator performs a different task than the equals **=** assignment operator. The equals **=** operator assigns the value on the right side of the equals **=** to the object (e.g., a variable) on the left side of the equals **=** operator.

**Examples:**

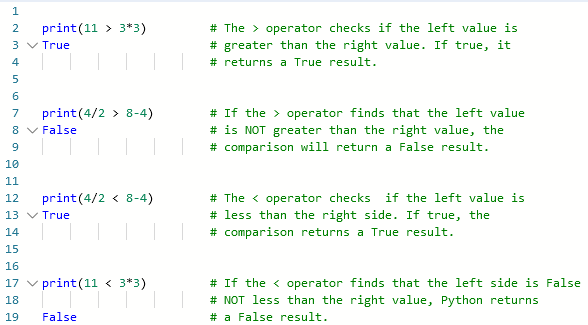


**PART 2: Greater Than > and Less Than < Operators**

The comparison operators greater than **>** and less than **<** also return a **True** or **False** Boolean result after comparing two values.

* To check if one value is larger than another value, use the greater than operator: **>**
* To check if one value is smaller than another value, use the less than operator: **<**

**Examples:**

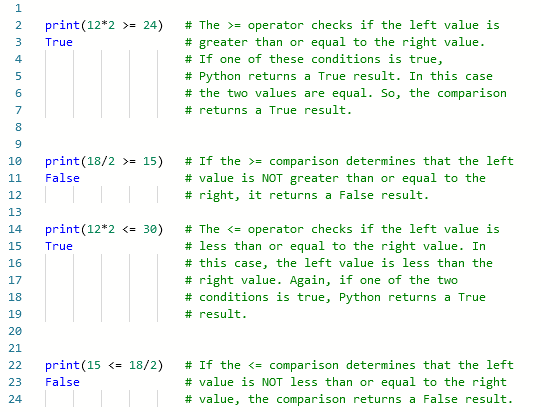


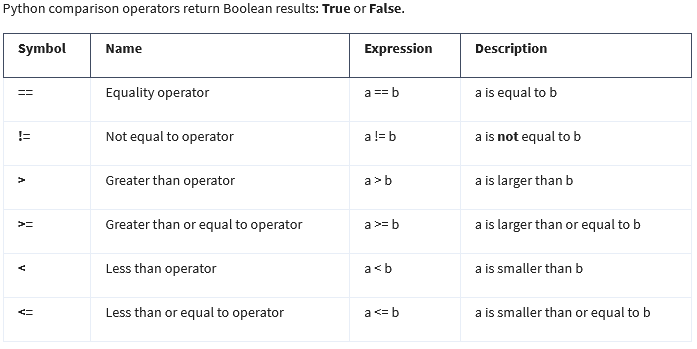
**PART 3: Greater Than or Equal to >= and Less Than or Equal to <= Operators**

Like the other comparison operators, the greater than or equal to **>=** and less than or equal to **<=** operators return a **True** or **False** Boolean result when a comparison is made.

* To check if one value is larger than or equal to another value, use the greater than or equal to operator: **>=**
* To check if one value is smaller than or equal to another value, use the less than or equal to operator: **<=**

**Examples:**



**Key Takeaways  
**  
**Resources for more information**For more information about the concepts covered in these practice exercises, please visit:

* [Order of Operations](https://www.mathsisfun.com/operation-order-pemdas.html) - A refresher on the mathematical Order of Operations.
* [Python Comparison Operators with Syntax and Example](https://data-flair.training/blogs/python-comparison-operators/) - Provides examples of more complex comparisons.
* [Raise numbers to a power: here’s how to exponentiate in Python](https://kodify.net/python/math/exponents/) - Explains multiple methods for calculating exponents in Python.Comparison Operators with Strings

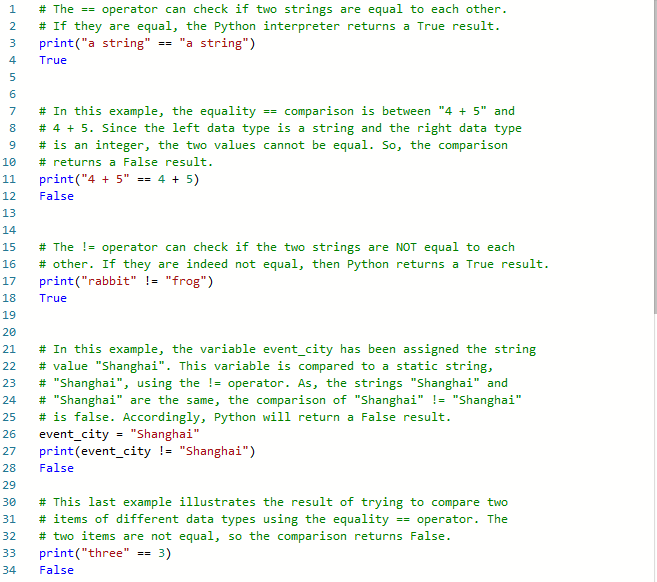
### Comparison Operators with Strings

In this reading, you will learn more about what comparison operators can and cannot do. If you use the **==** (equality) and **!=** (not equal to)operators with strings, you can check if two strings contain the same text or not. You can also alphabetize strings using **>** (greater than), **<** (less than)**, >=** (greater than or equal to)**, <=** (less than or equal to) comparison operators. As with numeric data types, comparison operators used with strings will return Boolean (**True**, **False**) results.

**PART 1: Equality == and Not Equal to != Operators with Strings**

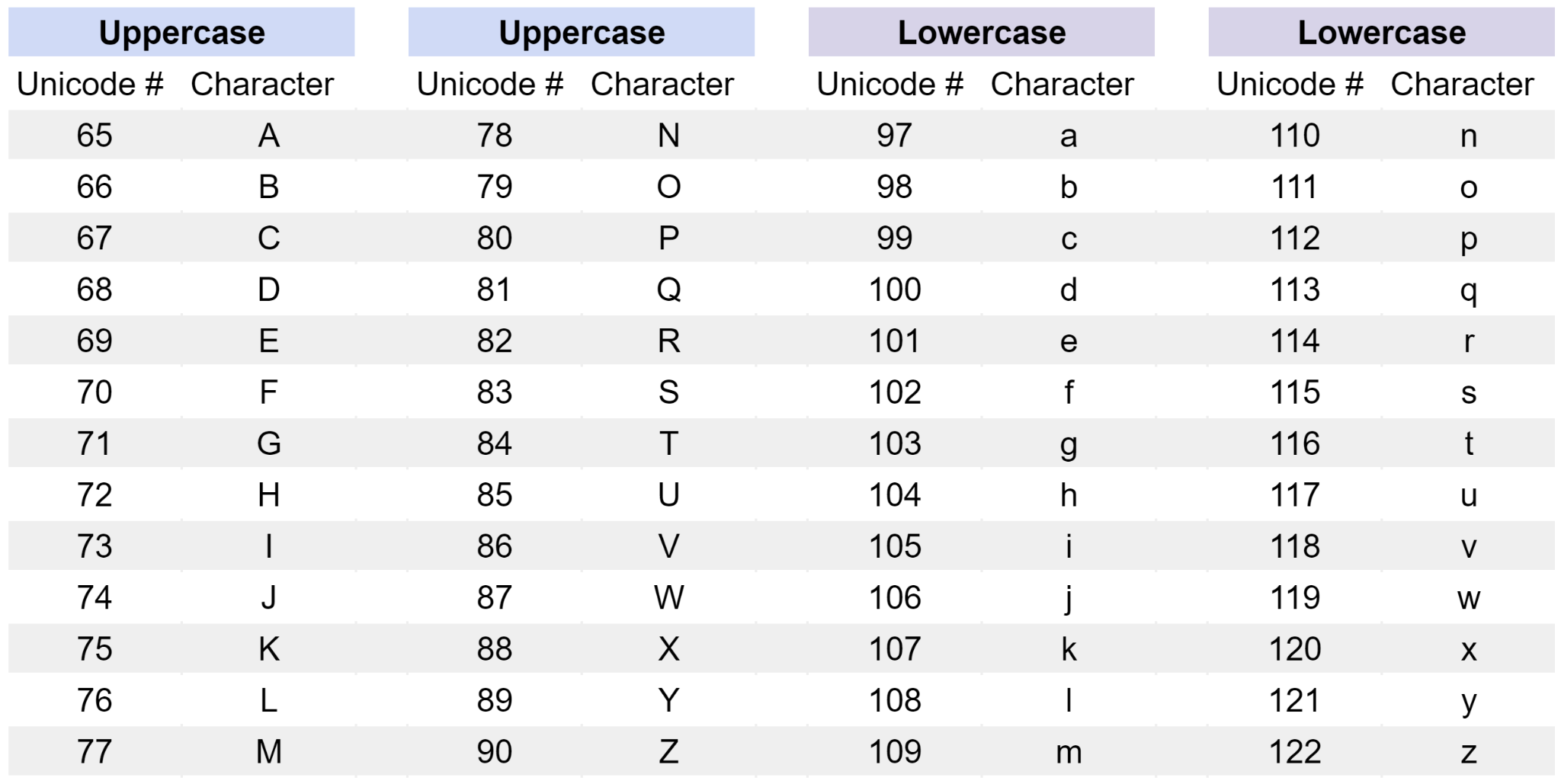
In Python, you can use comparison operators to compare strings. The equality **==** and the not equal to **!=** operators are helpful when you need to search for a specific string in a body of text, a log file, a spreadsheet, a database, and more. You can also check user input strings to compare them to another string. Note that Boolean data types are not string data types (Boolean **True** is not equal to the string "True").

**Examples:**



**PART 2: The Greater Than > and Less Than < Operators**

The comparison operators greater than **>** and less than **<** can be used to alphabetize words in Python. The letters of the alphabet have numeric codes in Unicode (also known as ASCII values). The uppercase letters A to Z are represented by the Unicode values 65 to 90. The lowercase letters a to z are represented by the Unicode values 97 to 122.



* To check if the first letter(s) of a string have a larger Unicode value (meaning the letter is closer to 122 or lowercase z) than the first letter of another string, use the greater than operator: **>**
* To check if the first letter(s) of a string have a smaller Unicode value (meaning the letter is closer to 65 or uppercase A) than the first letter of another string, use the less than operator: **<**

Like numeric comparisons with the greater than **>** and less than **<** operators, comparisons between strings also return Boolean **True** or **False** results.

# The greater than > operator checks if the left string has a higher

# Unicode value than the right string. If true, the Python interpreter

# returns a True result. Since W has a Unicode value of 87, and you can

# easily calculate that F has a Unicode value of 70, this comparison is

# the same as 87 > 70. As this is true, Python will return a True

# result.

print("Wednesday" > "Friday")

True

# The less than < operator checks if the left string has a lower

# Unicode value than the right string. If you reference the Unicode

# chart above, you can see that all lowercase letters have higher

# Unicode values than uppercase letters. We can see that B has a

# Unicode value of 66 and b has a Unicode value of 98. This

# comparison is the same as 66 < 98, which is true. So, Python will

# return a True result.

print("Brown" < "brown")

True

# If the strings have the same first few letters, the comparison will

# cycle through each letter of each string, from left to right until it

# finds two letters that have different Unicode values. In this example,

# both strings share the initial substring "sun", but then have

# different letters with different Unicode values in the fourth place

# in each string. So, the fourth letters 'b' and 't' of the two

# strings are used for the comparison. Since 'b' does not have a higher

# Unicode value than 't', the comparison returns a False result.

print("sunbathe" > "suntan")

False

# If two identical strings are compared using the less than < comparison

# operator, this will produce a False result because they are equal.

print("Lima" < "Lima")

False

# This last example illustrates the result of trying to compare two

# items of different data types using the less than < operator. The

# greater than > and less than operators < cannot be used to compare

# two different data types.

print("Five" < 6)

'''

Error on line 1:

print("Five" < 6)

TypeError: '<' not supported between instances of 'str' and 'int'

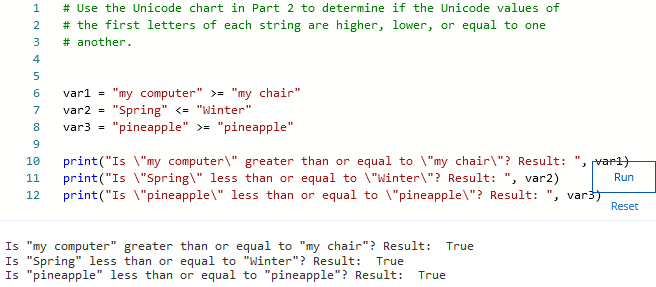
**PART 3: The Greater Than or Equal To >= and Less Than or Equal To <= Operators**

The greater than or equal to **>=** and less than or equal to **<=** operators can be used with strings as well. Like the other comparison operators, they will return a **True** or **False** Boolean result when a comparison is made between two strings.

* To check if a string has a larger or equal Unicode value than the first letter(s) of another string, use the greater than or equal to operator: **>=**
* To check if a string has a smaller or equal Unicode value than the first letter(s) of another string, use the less than or equal to operator: **<=**

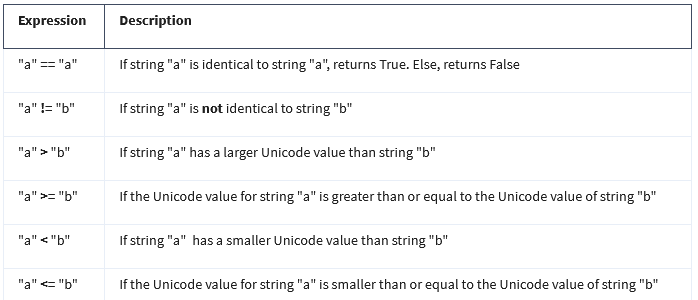
At this point, you should be familiar with how comparison operators work in Python. Can you determine what the results will be from the comparisons listed below? When you are ready to check your answers, click Run.

1. "my computer" >= "my chair"
2. "Spring" <= "Winter"
3. "pineapple" >= "pineapple"



**Key takeaways**

Python comparison operators return Boolean results (**True** or **False**) with strings:



**Resources for more information**

For more information about the concepts covered in these practice exercises, please visit:

[Python String Comparison: A Step-by-Step Guide (with Examples)](https://www.codingem.com/python-string-comparison/) - A quick reference guide to using comparison operators with strings. Includes part of a Unicode table that displays all of the Unicode values for both uppercase and lowercase letters.

[Comparing Strings using Python](https://stackabuse.com/comparing-strings-using-python/) - Provides more advanced examples of using comparison operators with strings.

### Logical Operators

Logical operators are used to construct more complex expressions. You can make complex comparisons by joining comparison statements together using the logical operators: **and**, **or**, **not**. Complex comparisons return a Boolean (**True** or **False**) result.

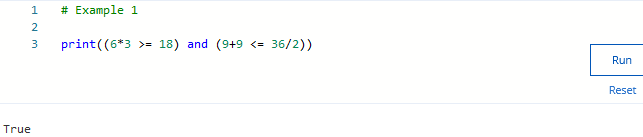
* **and**
  + Both sides of the statement being evaluated must be True for the whole statement to be True.
  + Example: (5 > 1 **and** 5 **<** 10) = **True**
* **or**
  + If either side of the comparison is True, then the whole statement is True.
  + Example: (color = "blue" **or** color = "green") = **True**
* **not**
  + Inverts the Boolean result 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.
  + Example: (**not** "A" **==** "A") = **False**

**PART 1: The and Logical Operator**

In Python, you can use the logical operator **and** to connect more than one comparison. This type of complex comparison is used to check if two comparison statements are both True or not. You might use the **and** operator when you need to execute a block of code, but only if two different conditions are true. For example, you might want to write a script that automates sending you an emergency alert if a server stops responding *and* there is an unusual increase in employees opening trouble tickets.

**Example 1:**

The following model demonstrates the use of the **and** logical operator to join comparisons between two mathematical expressions. The description below the example explains the order in which Python will process the line of code.



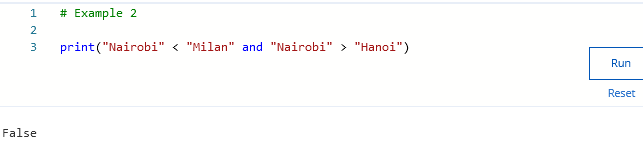
In the example above, the following activities were completed by Python in the following order:

1. Python solves the numerical expressions using the order of operations. **(6\*3 >= 18) and (9+9 <= 36/2) becomes (18 >= 18) and (18 <= 18)**
2. Python compares the results of the numerical expressions using the comparison operators (in this case >= and <=). **(18 >= 18) and (18 <= 18) becomes True and True**
3. Python checks if both sides of the logical operator "and" are true. **True and True become True**
4. Python returns a Boolean value: True or False. **The complex comparison returns a True result.**

**Example 2:**

In this next example, "Nairobi" < "Milan" and "Nairobi" > "Hanoi", the **and** logical operator is connecting two string comparison statements. You learned previously that using the greater than and less than operators on strings will test the alphabetical order (technically Unicode values) of the strings. So, this complex comparison is checking if "Nairobi" is alphabetized before "Milan" (False) AND after "Hanoi" (True).

This comparison returns a False result because both sides of the logical operator are not True. A comparison statement like this might be used to iterate through a list of names to check if they are alphabetized in the correct order.



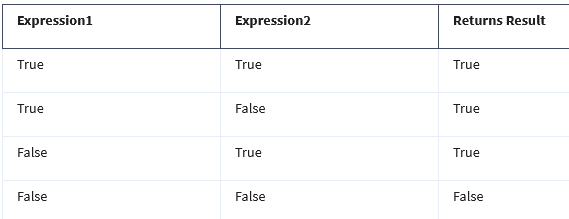
**PART 2: The or Logical Operator**

The **or** logical operator tests two conditions to determine if at least one side of the **or** logical operator is True. The result of the test can be used to trigger a block of code if at least one condition is present.

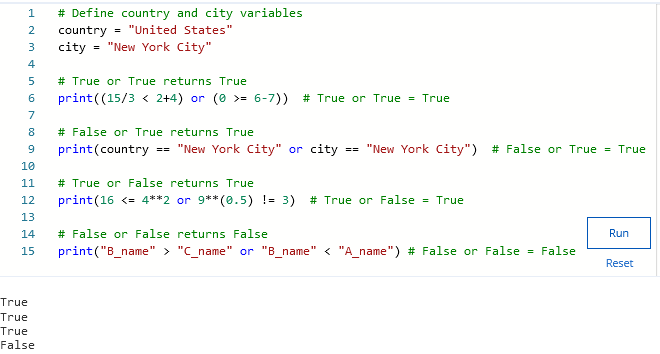
**Syntax:**



**Returns Booleans:**



**Examples:**



**PART 3: The not Logical Operator**

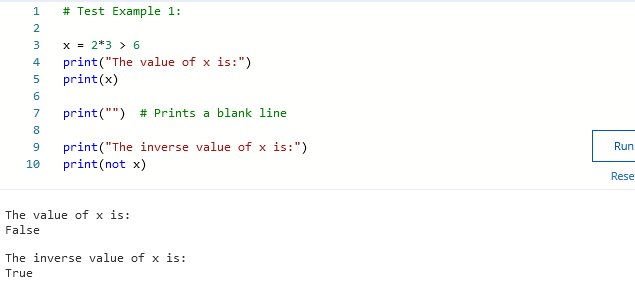
The **not** logical operator inverts the value of the comparison expression. This is a helpful tool when you want to execute a block of code as long as a certain condition is **not** present.

* If the conditional expression is True, the **not** logical operator can be added to make the expression **not** True (False).
* If the conditional expression is False, the **not** logical operator can be added to make the expression **not** False (True).

**Syntax:**

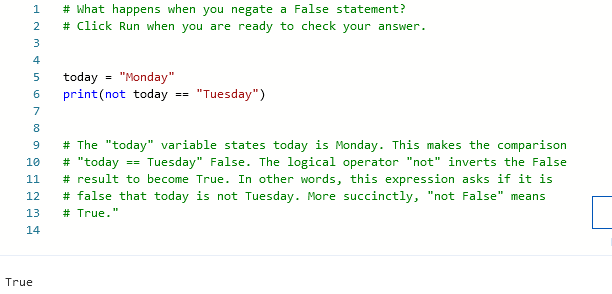
1 not expression

**Example 1:**

****

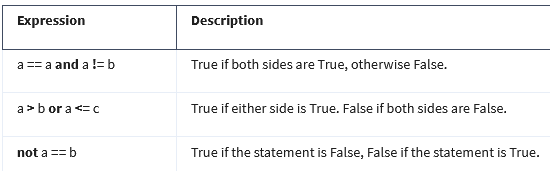
**Example 2:**

Can you determine the result of the following comparison?



**Key takeaways**

When Python logical operators are used with comparison operators, the interpreter will return Boolean results (**True** or **False**):



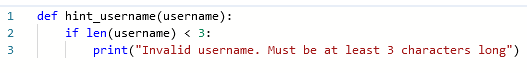
### Review: Branching with if statements

This reading contains the code used in the instructional videos from [**Branching with if statements**](https://www.coursera.org/learn/python-crash-course/lecture/8xZqQ/branching-with-if-statements)

**Introduction**

This follow-along reading is organized to match the content in the video that follows. It contains the same code shown in the next video. These code blocks will provide you with the opportunity to see how the code is written, allow you to practice running it, and can be used as a reference to refer back to.

You can follow along in the reading as the instructor discusses the code or review the code after watching the video.



def hint\_username(username):

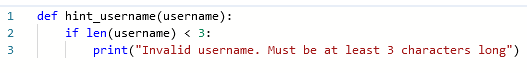
if len(username) < 3:

print("Invalid username. Must be at least 3 characters long")

### Branching with if Statements

Now that we're armed with knowledge of Python's expressions, comparators, and variables, we can dive right into how to use them in our scripts to perform different actions based on their values. The ability of a program to alter its execution sequence is called branching, and it's a key component in making your scripts useful. You probably use the idea of branching a bunch in your everyday life. For example, if it's before noon, you might greet someone by saying good morning instead of good afternoon or good evening. If it's raining outside, you might choose to take an umbrella. If it's cold, you probably wear a jacket.

In your scripts, you can instruct your computer to make decisions based on inputs too. Let's take a look at an IT-focused example. In many companies, new employees can choose the username they'll use to access the company's systems, and usually, the chosen username needs to fit with a given set of guidelines. Companies can set different criteria for what a valid username looks like. For now, let's assume that at your company, a valid username has to have at least three characters. You've been tasked with writing a program that will tell the user if their choices are valid or not. To do that, you could write a function like this.



This function checks whether the length of the username is smaller than three. This function checks whether the length of the username is smaller than three. If it is, the function prints a message saying that the username is invalid.

Look closely at how the if statement is written. We write the keyword if followed by the condition that we want to check for, and then followed by a colon. After that, comes the body of the if block, which is indented further to the right. if block, which is indented further to the right.

You may notice that there are some similarities between how an if block and the function are defined. The keyword, either def or if, indicates the start of a special block. At the end of the first line, we use a colon, and then the body of the function or the if block is indented to the right. But there's also an important difference between how an if block and a function are defined.

The body of the if block will only execute when the condition evaluates to true; otherwise, it skipped.

Of course, you can do a lot more things inside the body of the if block than just printing stuff. As we expand our programming abilities, we'll learn how to do things like shorten text that's too long, delete a file if it exists, start a service if it's not running, and a bunch more. If your code is inside a function, you could also choose to return a value depending on whether a certain condition is met.

By now, you know how to define functions, and inside those functions, you can now make your program do something only when certain conditions are met. Ready to branch out and make our branches even more interesting with else statements? Then hop on over to the next video, or else, you'll miss out.

### Review: else statements

This reading contains the code used in the instructional videos from [**else statements**](https://www.coursera.org/learn/python-crash-course/lecture/k2wDX/else-statements)

**Introduction**

This follow-along reading is organized to match the content in the video that follows. It contains the same code shown in the next video. These code blocks will provide you with the opportunity to see how the code is written, allow you to practice running it, and can be used as a reference to refer back to.

You can follow along in the reading as the instructor discusses the code or review the code after watching the video.

def hint\_username(username):

if len(username) < 3:

print("Invalid username. Must be at least 3 characters long")

else:

print("Valid username")

#This code will not have an output.

**About this code**

This snippet of code defines a function called *hint\_username*. The *if len(username) < 3*: statement checks the length of the string *username*. If the length of the string is less than 3 characters, the code inside the *if* statement is executed. The *print("Invalid username. Must be at least 3 characters long")* statement prints the message "Invalid username. Must be at least 3 characters long". The *else*: statement is executed if the length of the string *username* is not less than 3 characters. In this case, the code inside the *else* statement is executed, which is the *print("Valid username")* statement.

def is\_even(number):

if number % 2 == 0:

return True

return False

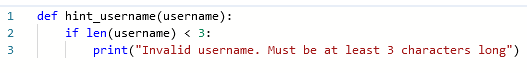
#This code has no ouput

**About this code**

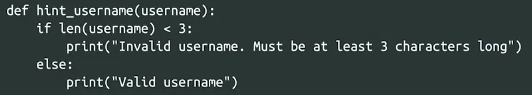
This code snippet defines a function called *is\_even*. It checks whether a number is even. *if number % 2 == 0*: is the part of the code that checks if the number is even. If the number is odd, it will return false. The code does not have any output currently because it has not been provided with a number to check.

### else Statements

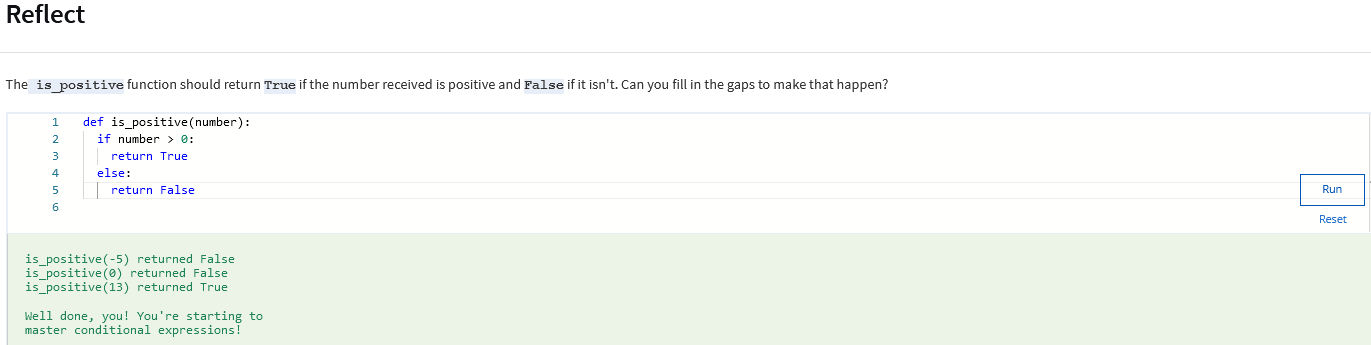
The if statement is already a pretty useful construct, but we can extend it to make it even more powerful. Think about the username example from the last video.



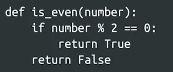
What if we also wanted to print a message when the username was valid? Here, we've included an else statement to achieve this.



The program can now go in one of two directions depending on the length of the username. If it's not long enough, we get a message indicating that the username is invalid. But if the program verifies that the username is long enough, it will print a message saying it is valid. Pay attention to how the else statement is written. It uses the else keyword followed by a colon to indicate the beginning of the else block. Once again, the body of the block is further indented to the right. As we've called out before these blocks can contain multiple lines and do more than just print messages. They can do calculations, modify values, return values, and a lot more. And remember that you can choose to use as many or as few spaces as you want for the indentation, but you always need to indent and you always need to use the same number of spaces.



The else statement is very useful, but we don't always need it. Say we want to have a function that checks if a value is even or odd. We could do that with a piece of code like this.



Here, we're using a new operator so let's first explain that. The modulo operator (%) is represented by the percentage sign and returns the remainder of the integer division between two numbers. The integer division is an operation between integers that yields two results which are both integers, the quotient and the remainder. So if we do an integer division between 5 and 2, the quotient is 2 and the remainder is 1. If we do an integer division between 11 and 3, the quotient is 3 and the remainder is 2.

Even numbers are all multiples of 2 which means the remainder of the integer division between an even number and 2 is always going to be 0. In this function, we're using this principle to decide whether a number is even or not. So how come we have these two return statements, one below the other, without an else statement?

The trick is that when a return statement is executed, the function exits so that the code that follows doesn't get executed. This means that if the number is even, the computer will reach the return true statement and exit the function. Anything that comes after that will only be executed if the condition in the if statement was false. In other words, once the function reaches the return false line, we know for sure that the if condition was false which means the number was odd. At first, you might feel more comfortable including the else statement, even if it's not needed.

It's important to know that both ways of writing this are correct. And remember that this technique can only be used when you're returning a value inside the if statement.

To recap, the if statement allows us to branch the execution based on a specific condition being true. The else statement lets us set a piece of code to run only when the condition of the if statement was false. If you return a value inside an if block then the code after the block will only be executed if the condition was false.

If all these ifs and elses are starting to get a little confusing, that's okay. There's a lot to soak up here and the best way to do that is yeah, you guessed it, practice. So review the content and practice on your own as much as you need. Once you're done, meet me over in the next video.

### else Statements and the Modulo Operator

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 executed 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%5 would return 0, as there is no remainder.

### Review: elif statements

This reading contains the code used in the instructional videos from [**elif statements**](https://www.coursera.org/learn/python-crash-course/lecture/A9mHE/elif-statements)

**Introduction**

This follow-along reading is organized to match the content in the video that follows. It contains the same code shown in the next video. These code blocks will provide you with the opportunity to see how the code is written, allow you to practice running it, and can be used as a reference to refer back to.

You can follow along in the reading as the instructor discusses the code or review the code after watching the video.

def hint\_username(username):

if len(username) < 3:

print("Invalid username. Must be at least 3 characters long")

else:

if len(username) > 15:

print("Invalid username. Must be at most 15 characters long")

else:

print("Valid username")

**About this code**

This snippet of code defines a function called *hint\_username*. The *if len(username) < 3*: statement checks the length of the string username. If the length of the string is less than 3 characters, the code inside the if statement is executed. The *print("Invalid username. Must be at least 3 characters long")* statement prints the message "Invalid username. Must be at least 3 characters long". The *else:* statement is executed if the length of the string username is greater than 15 characters. In this case, the code inside the else statement is executed, which is the *print("Invalid username. Must be at most 15 characters long")* statement. If the username is the correct length, between 3 and 15 characters long, *print("Valid username")* is executed.

def hint\_username(username):

if len(username) < 3:

print("Invalid username. Must be at least 3 characters long")

elif len(username) > 15:

print("Invalid username. Must be at most 15 characters long")

else:

print("Valid username")

**About this code**

This snippet of code works the same way as the code block we just looked at. The difference is that this code uses the *elif* statement. *elif* statements must be used along with an *if* statement. The *elif* statement will only be checked if the condition of the *if* statement was not true.

### elif Statements

The if and else blocks allow us to branch execution depending on whether a condition is true or false. But what if there are more conditions to take into account? This is where the elif statement, which is short for else if, comes into play.

But before we jump into how to use it, let's take a look at why we need it in the first place. Let's go back to our trusty username validation example. Now, what if your company also had a rule that usernames longer than 15 characters aren't allowed? How could we let the user know if their chosen username was too long? We could do it like this.

def hint\_username(username):

if len(username) < 3:

print("Invalid username. Must be at least 3 characters long")

else:

if len(username) > 15:

print("Invalid username. Must be at most 15 characters long")

else:

print("Valid username")

In this case, we're adding an extra if block inside the else block. This works, but the way the code is nested makes it kind of hard to read. To avoid unnecessary nesting and make the code clearer, Python gives us the elif keyword, which lets us handle more than two comparison cases. Take a look.

def hint\_username(username):

if len(username) < 3:

print("Invalid username. Must be at least 3 characters long")

elif len(username) > 15:

print("Invalid username. Must be at most 15 characters long")

else:

print("Valid username")

The elif statement looks very similar to the if statement. It's followed by a condition and a colon, and a block of code indented to the right that forms the body. The condition must be true for the body of the elif block to be executed.

The main difference between elif and if statements is we can only write an elif block as a companion to an if block. That's because the condition of the elif statement will only be checked if the condition of the if statement wasn't true.

So in this example, the program first checks whether the username is less than three characters long, and prints a message if that's the case. If the username has at least three characters, the program then checks if it's longer than 15 characters. If it is, we get a message to tell us that. Finally, if none of the above conditions were met, the program prints a message indicating that the username is valid.

There's no limit to how many conditions we can add, and it's easy to include new ones. For example, say the company decided that the username shouldn't include numbers. We could easily add an extra elif condition to check for this.

You now know how to compare things and use those comparisons for your if, elif, and else statements. And you are using all of them inside functions. Using branching to determine your program's flow opens up a whole new realm of possibilities in your scripts. You can use comparisons to pick between executing different pieces of code, which makes your script pretty flexible.

Branching also helps you do all kinds of practical things like only backing up files with a certain extension, or only allowing login access to a server during certain times of the day. Any time your program needs to make a decision, you can specify its behavior with a branching statement. Are you starting to notice tasks in your day-to-day that could be made more efficient with scripting? There's so many possibilities, and we're only just getting started with all the cool stuff programming can help you do.

We've covered a lot in these last few videos. Remembering all these concepts can take some time, and the best way to learn them is to use them. So we've put together a cheat sheet for you in the next reading. You'll find all these operators and branching blocks listed there in one handy resource. It's super useful when you need a quick refresher. So no skipping the reading.

### Complex Branching with elif Statements

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!

### Explore: If, elif, and else statements

### Study Guide: Conditionals

This study guide provides a quick-reference summary of what you learned in this lesson and serves as a guide for the upcoming practice quiz.

In the Conditionals segment, you learned about the built-in Python operators used for comparing values and the logical operators for making complex comparisons. You also learned how to use operators in if-elif-else blocks.

https://github.com/GaJoDev/Python/blob/main/Coursera%20-%20Google%20IT%20Automation%20with%20Python%20Professional%20Certificate/1.%20Crash%20Course%20on%20Python/Module%202/study\_guide\_conditionals.md

**Knowledge**

**Comparison operators with numerical values**

Comparison expressions return a Boolean result (True or False).

* x **==** y If x is equal to y, return True. Else, return False.
* x **!=** y If x is not equal to y, return True. Else, return False.
* x **<** y If x is less than y, return True. Else, return False.
* x **<=** y If x is less than or equal to y, return True. Else, return False.
* x **>** y If x is greater than y, return True. Else, return False.
* x **>=** y If x is greater or equal to y, return True. Else, return False.

**Comparison operators with strings**

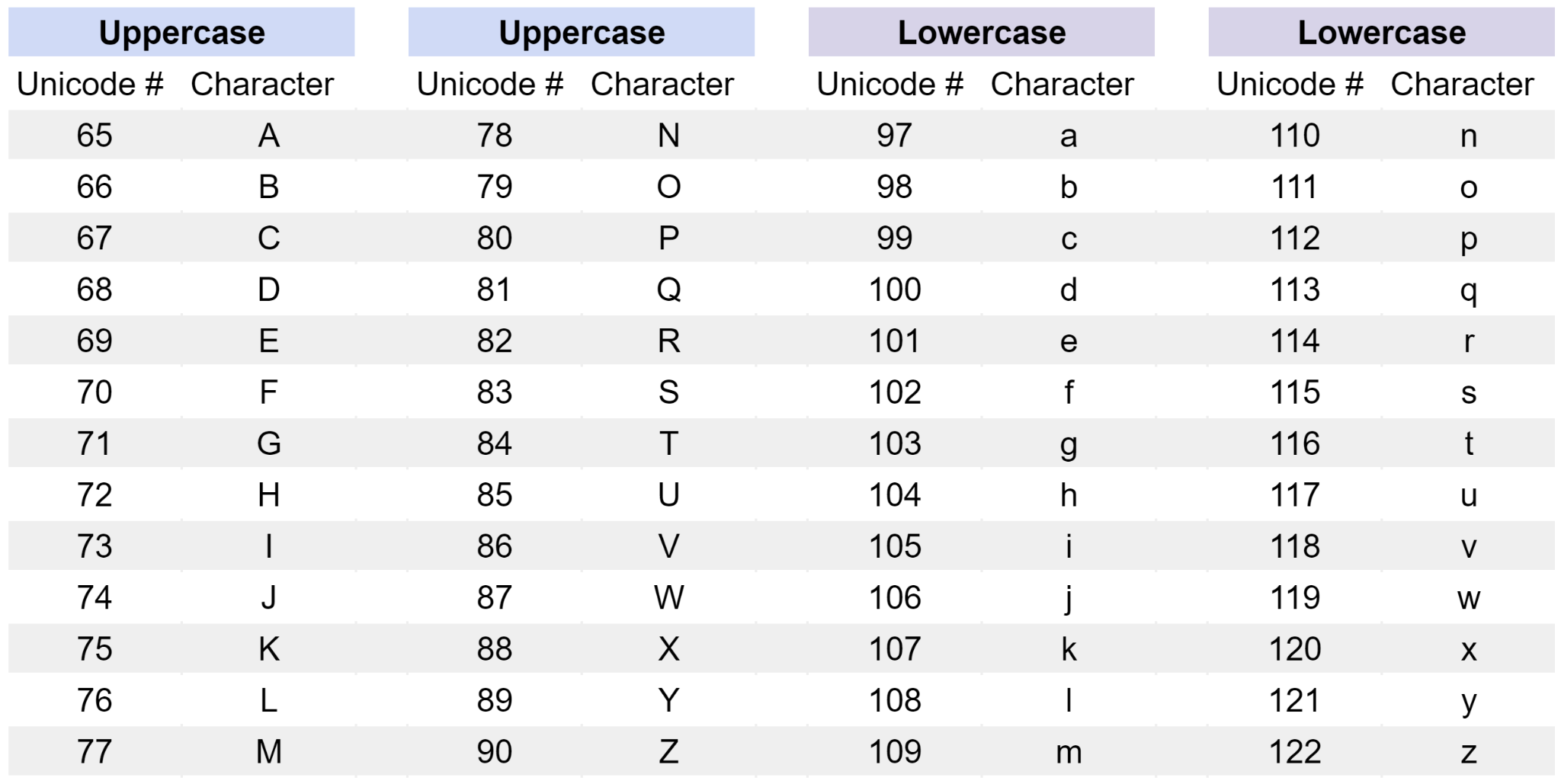
Comparison expressions with strings also return a Boolean result (True or False).

* "x" **==** "y" If the words are the same, return True. Else, return False.
* "x" **!=** "y" If the words are **not** the same, return True. Else, return False.

When used with strings, the following comparison expressions will alphabetize the strings.

* "x" **<** "y" If string "x" has a smaller Unicode value than string "y", return True. Else, return False.
* "x" **<=** "y" If the Unicode value for string "x" is smaller than or equal to the Unicode value of string "y", return True. Else, return False.
* "x" **>** "y" If string "x" has a larger Unicode value than string "y", return True. Else, return False.
* "x" **>=** "y" If the Unicode value for string "x" is greater than or equal to the Unicode value of string "y", return True. Else, return False.

**Unicode values for the alphabet**



**Logical operators**

Logical operators are used to combine comparison expressions and also return Boolean results (True or False).

* comparison1 **and** comparison2
  + Returns a True result if both comparison1 **and** comparison2 are true.
  + If they are not both true, return False.
* comparison1 **or** comparison2
  + Returns a True result if either comparison1 and/or comparison2 are True.
  + If neither comparison is true, return False.
* **not** comparison1
  + Returns the inverse Boolean value of the comparison.
    - Returns a True result if comparison1 is false.
    - If comparison1 is true, then returns False.

**Syntax of an if-elif-else block**

if condition1:

action1

elif condition2:

action2

else:

Action3

* If condition1 is True:
  + Then perform action1 and exit if-elif-else block
* If condition2 is True:
  + Then perform action2 and exit if-elif-else block
* If neither condition1 nor condition2 are True:
  + Then perform action3 and exit if-elif-else block

**Coding skills**

**Skill Group 1**

* Use a comparison operator with numbers
* Use a comparison operator to alphabetize strings

# The value of 10\*4 (40) is greater than 14+23 (37), therefore this

# comparison expression will return the Boolean value of True.

print(10\*4 > 14+23) # Should print True

# The letter "t" has a Unicode value of 116 and the letter "s" has a

# Unicode value of 115. Since 116 is not less than 115, the

# comparison of "tall" < "short" (or 116 < 115) is False.

print("tall" < "short") # Should print False

**Skill Group 2**

* Use a function with the def() keyword
* Pass a parameter to the function
* Use an if-elif-else statement
* Assign strings to variables
* Use conditional operators
* Return a value

# This function accepts one variable as a parameter

def translate\_error\_code(error\_code):

# The if-elif-else block assesses the value of the variable

# passed to the function as a parameter. The if statement uses

# the equality operator == to test the value of the variable.

# This test returns a Boolean (True/False) result.

if error\_code == "401 Unauthorized":

# If the comparison above returns True, then the indented

# line(s) inside the if-statement will run. In this case, the

# action is to assign a string to the translation variable.

# The remainder of the if-elif-else block will not run.

# The Python interpreter will skip to the next line outside of

# the if-elif-else block. In this case, the next line is the

# return value statement.

translation = "Server received an unauthenticated request"

# If the initial if-statement returns a False result, then the

# first elif-statement will run a different test on the value

# of the variable.

elif error\_code == "404 Not Found":

# If the first elif-statement returns a True result, then the

# indented line(s) inside the first elif-statement will run.

# After this line, the remainder of the if-elif-else block will

# not run. The Python interpreter will skip to the next line

# outside of the if-elif-else block.

translation = "Requested web page not found on server"

# If both the initial if-statement and the first elif-statement

# return a False result, then the second elif-statement will

# run.

elif error\_code == "408 Request Timeout":

# If the second elif-statement returns a True result, then the

# indented line(s) inside the second elif-statement will run.

# After this line, the remainder of the if-elif-else block will

# not run. The Python interpreter will skip to the next line

# outside of the if-elif-else block.

translation = "Server request to close unused connection"

# If the conditional tests above do not produce a True result

# then the else-statement will run.

else:

translation = "Unknown error code"

# The if-elif-else block ends.

# The next line outside of the if-elif-else block will run

# after exiting the block. In this case, the next line returns

# the output from the if-elif-else block.

return translation

# The print() function allows us to display the output of the

# function. To call a function in a print statement, the syntax

# is print(name\_of\_function(parameter))

print(translate\_error\_code("404 Not Found"))

# Expected output:

# Requested web page not found on server

**Output:**

Requested web page not found on server

**Skill Group 3**

* Use an if-elif-else statement with:
  + comparison operators
  + logical operators

# Sets value of the "number" variable

number = 25

# The "number" variable will first be compared to 5. Since it is

# False that "number" is not less than or equal to 5, the expression indented

# under this line will be ignored.

if number <= 5:

print("The number is 5 or smaller.")

# Next, the "number" variable will be compared to 33. Since it is

# False that "number" is equal to 33, the expression indented under

# this line will be ignored.

elif number == 33:

print("The number is 33.")

# Then, the "number" variable will be compared to 32 and 6. Since it

# is True that 25 is less than 32 and greater than 6, the Python

# interpreter will print "The number is less than 32 and/or greater

# than 6." Then, it will exit the if-elif-else statement and the remainder

# of the if-elif-else statement will be ignored.

elif number < 32 and number >= 6:

print("The number is less than 32 and greater than 6.")

else:

print("The number is " + str(number))

# Expected output is:

# The number is less than 32 and greater than 6.

**Output:**

The number is less than 32 and greater than 6.

**Skill Group 4**

* Use an if statement to calculate a return value
* Use conditional operators
* Recall the arithmetic operators // and %

# This function rounds a variable number up to the nearest 10x value

def round\_up(number):

x = 10

# The floor division operator will calculate the integer value of

# "number" divided by x: 35 // 10 will return the integer 3.

whole\_number = number // x

# The modulo operator will calculate the remainder value of "number"

# divided by x: 35 % 10 will return the remainder value 5.

remainder = number % x

# If the remainder is greater than or equal to 5:

if remainder >= 5:

# Return x multiplied by the (whole\_number+1) to round up

return x\*(whole\_number+1)

# Else, return x multiplied by the whole\_number to round down

return x\*whole\_number

# Calls the function with the parameter value of 35.

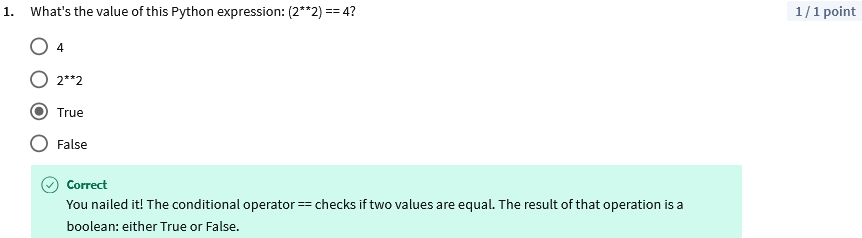
print(round\_up(35)) # Should print 40

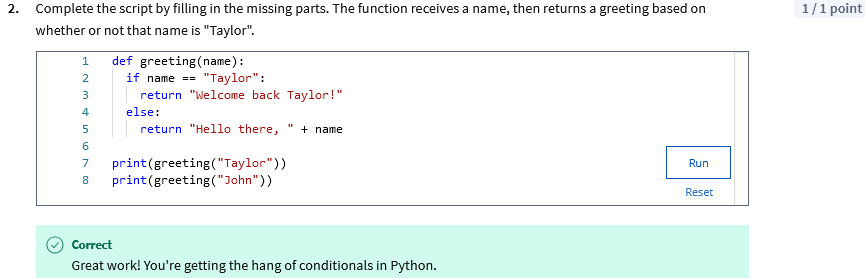
**Output:**

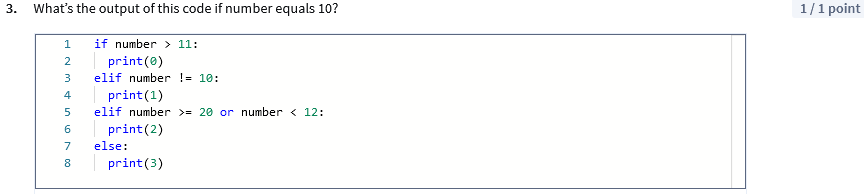
40

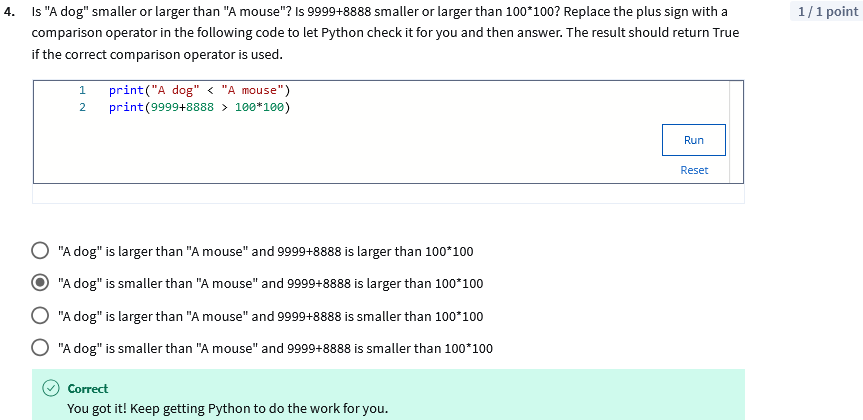
### Practice Quiz: Conditionals

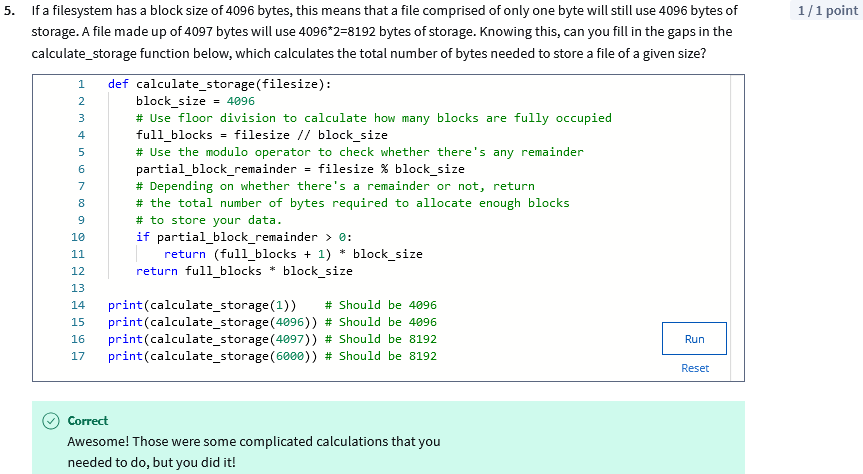












## **Module Review**

### 

### Glossary terms from course 1, module 2

**Built-in functions:** Functions that exist within Python and can be called directly

**Comments:** Notes to yourself and/or other programmers to make the purpose of the code clear

**Data types:** Classes of data (e.g., string, int, float, Boolean, etc.), which include the properties and behaviors of instances of the data type (variables)

**Explicit conversion:** This occurs when code is written to manually convert one data type to another using a data type conversion function

**Expression:** A combination of numbers, symbols, or other values that produce a result when evaluated

**Implicit conversion:** This occurs when the Python interpreter automatically converts one data type to another

**Logical operators:** Operators used to combine or manipulate boolean values (True or False) to create complex conditions for decision-making.

**Parameter (argument):** A value passed into a function for use within the function, controlling the behavior of the CSV reader and writer

**Refactoring:** When a code is updated to be more self-documenting and clarify the intent

**Return value**: This is the value or variable returned as the end result of a function

### Study Guide: Module 2 Graded Quiz

It is time to prepare for the Module 2 graded quiz. Please review the following items from this Module before starting the Module 2 Graded Quiz. If you would like to refresh your memory on these materials, please revisit the Study Guides located before each Practice Quiz in Module 2 : [Study Guide: Expressions and Variables](https://www.coursera.org/learn/python-crash-course/supplement/nLGXk/study-guide-expressions-and-variables), [Study Guide: Functions](https://www.coursera.org/learn/python-crash-course/supplement/SshSU/study-guide-functions), and [Study Guide: Conditionals](https://www.coursera.org/learn/python-crash-course/supplement/5Y1CI/study-guide-conditionals)

.

**Knowledge**

* How to assign values to variables and use them in code
* How to construct a function and use function parameters
* How comparison and logical operators can be used,
* How comparison and logical operators behave with different data types
* What type of results simple and complex comparisons produce
* How to alphabetize strings using comparison operators
* What must appear after the **if** and **elif** keywords
* What the **elif** keyword does
* When an **if, elif,** or **else**-statement will execute
* How to use the floor division // and modulo % operators and why
* How to use logical operators with comparison operators to develop complex conditional statements within an **if-elif-else** block
* Best practices for coding and their benefits
* What “self-documenting code” means

There may be a few questions on the quiz that will ask you about *either* the *output* of a small block of code *or the value of part of the code.* Make sure to read the instructions carefully on those questions.

**Coding skills**

**Skill Group 1**

* Use a function with the def() keyword
* Pass a parameter to the function
* Use an if-elif-else block to set specific conditions for a variety of actions
* Assign strings to variables
* Use comparison operators
* Return a value
* Call the function in a print statement and pass parameter to the function

# A function is created with the def() keyword. The parameter

# variable "time\_as\_string" is passed to the function through a

# call to the function.

def task\_reminder(time\_as\_string):

# The following if-elif-else block assigns various strings to

# the variable "task" depending on specific conditions. The

# test conditions are set using the == equality comparison

# operator. In this case, the time passed through the

# "time\_as\_string" parameter variable is tested as the

# specific condition. So, if the time is "11:30 a.m.", then

# "task" is assigned the value: "Run TPS report".

if time\_as\_string == "8:00 a.m.":

task = "Check overnight backup images"

elif time\_as\_string == "11:30 a.m.":

task = "Run TPS report"

elif time\_as\_string == "5:30 p.m.":

task = "Reboot servers"

# The else statement is a catchall for all other values of

# the "time\_as\_string" parameter variable not listed in the

# if-elif block of code.

else:

task = "Provide IT Support to employees"

# This line returns the value of "task" to the function call.

return task

# This line calls the function and passes a parameter

# ("10:00 a.m.") to the function.

print(task\_reminder("10:00 a.m."))

# Should print "Provide IT Support to employees"

**Output:**

Provide IT Support to employees

**Skill Group 2**

* Predict the output of expressions written with Python’s syntax.
* Requires an understanding of:
  + Arithmetic and logical operators
  + How functions return and print values
  + How if-elif-else statements work
  + Comparison operators

# Example 1

# Evaluate the output of this print statement

def product(a, b):

return(a\*b)

print(product(product(2,4), product(3,5)))

#################################

# Example 2

# Evaluate the output of this print statement

def difference(a, b):

return(a-b)

def sum(a, b):

return(a+b)

print(difference(sum(2,2), sum(3,3)))

#################################

# Example 3

# Evaluate the Boolean output of this comparison

print((5 >= 2\*4) and (5 <= 4\*3))

#################################

# Example 4

# Evaluate the value of the comparison in the if statement

x = 3

if x+5 > x\*\*2 or x % 4 != 0:

print("This comparison is True")

#################################

# Example 5

# Evaluate the output of this if-elif-else statement

number = 6

if number \* 2 < 14:

print(number \* 6 % 3)

elif number > 7:

print(100 / number)

else:

print(7 - number)

# Click Run to check your answers. If you are having trouble

# calculating the correct answers manually, please review the

# Practice Quiz Study Guides, videos, and readings in this Module.

**Output:**

120

-2

False

This comparison is True

0

**Skill Group 3**

* Create an if-elif-else statement with:
  + a complex conditional statement using both comparison and logical operators
  + values assigned to variables
  + arithmetic operators, including the modulo % operator

def get\_remainder(x, y):

if x == 0 or y == 0 or x ==y:

remainder = 0

else:

remainder = (x % y) / y

return remainder

print(get\_remainder(10, 3))

Output:

0.3333333333333333

**Reminder: Correct syntax is critical**

Using precise syntax is critical when writing code in any programming language, including Python. Even a small typo can cause a syntax error and the automated Python-coded quiz grader will mark your code as incorrect. This reflects real life coding errors in the sense that a single error in spelling, case, punctuation, etc. can cause your code to fail. Coding problems caused by imprecise syntax will always be an issue whether you are learning a programming language or you are using programming skills on the job. So, it is critical to start the habit of being precise in your code now.

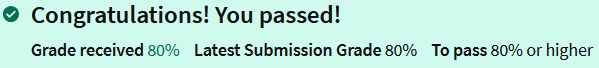
No credit will be given if there are any coding errors on the automated graded quizzes - including minor errors. Fortunately, you have 3 optional retake opportunities on the graded quizzes in this course. Additionally, you have unlimited retakes on practice quizzes and can review the videos and readings as many times as you need to master the concepts in this course.

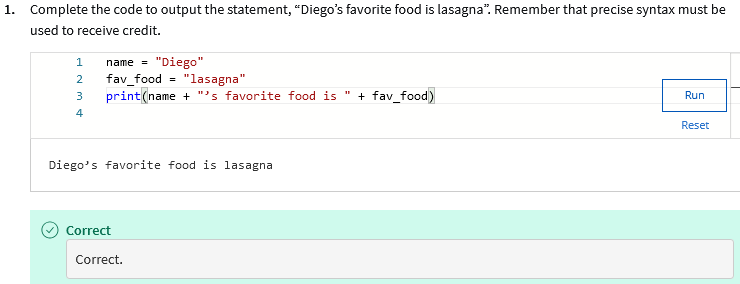
Now, before starting the graded quiz, please review this list of common syntax errors coders make when writing code.

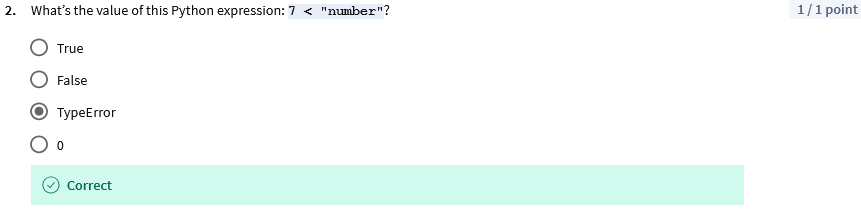
**Common syntax errors:**

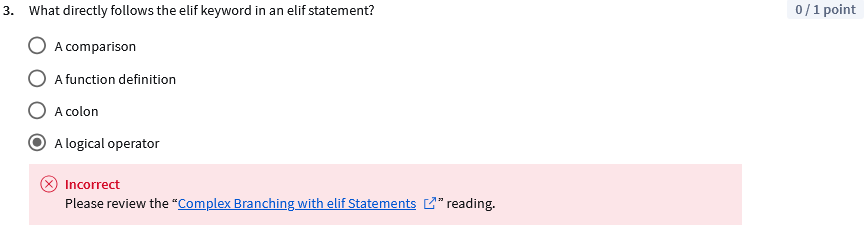
* Misspellings
* Incorrect indentations
* Missing or incorrect key characters:
  + Parenthetical types - ( curved ), [ square ], { curly }
  + Quote types - "straight-double" or 'straight-single', “curly-double” or ‘curly-single’
  + Block introduction characters, like colons - :
* Data type mismatches
* Missing, incorrectly used, or misplaced Python reserved words
* Using the wrong case (uppercase/lowercase) - Python is a case-sensitive language

### Module 2 Graded Assessment

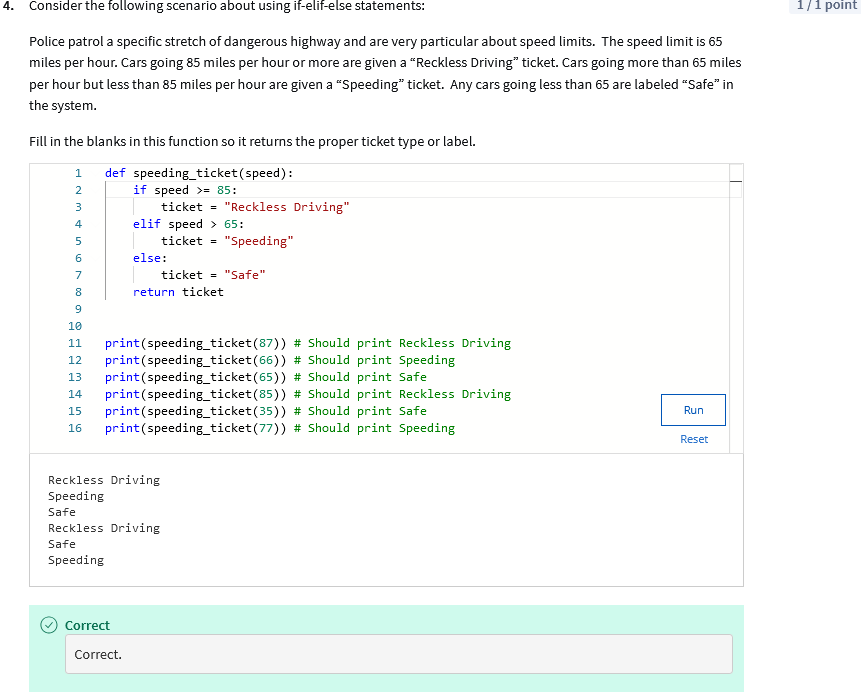


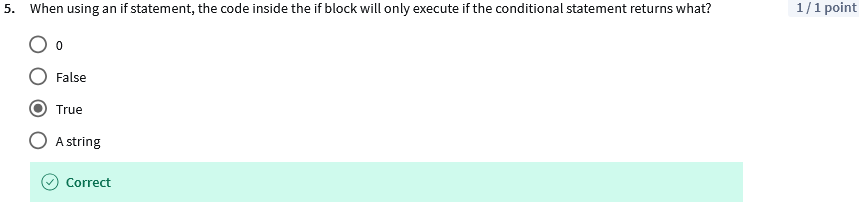


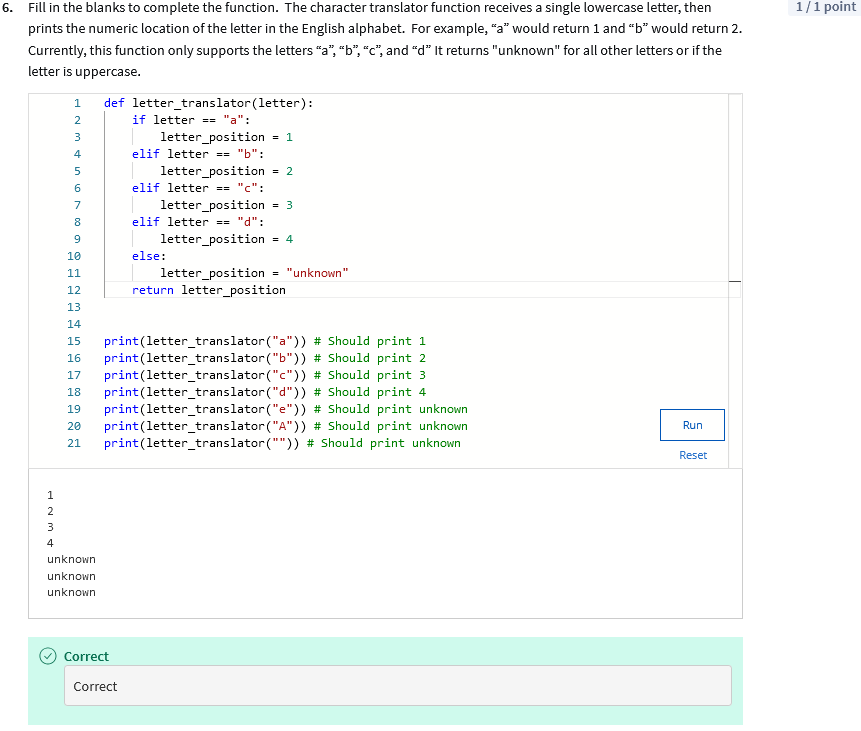


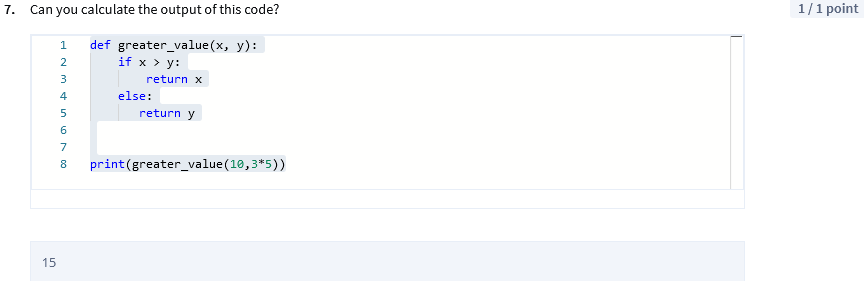


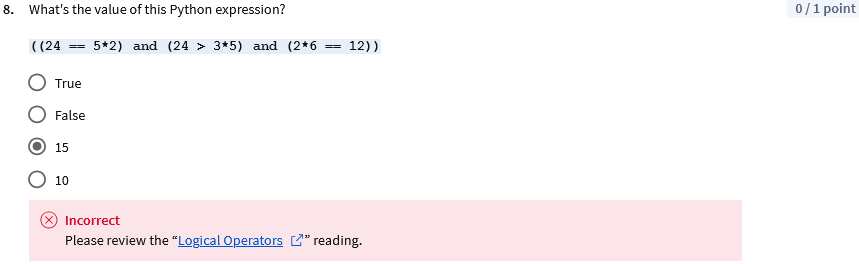
Correct answer is Colon



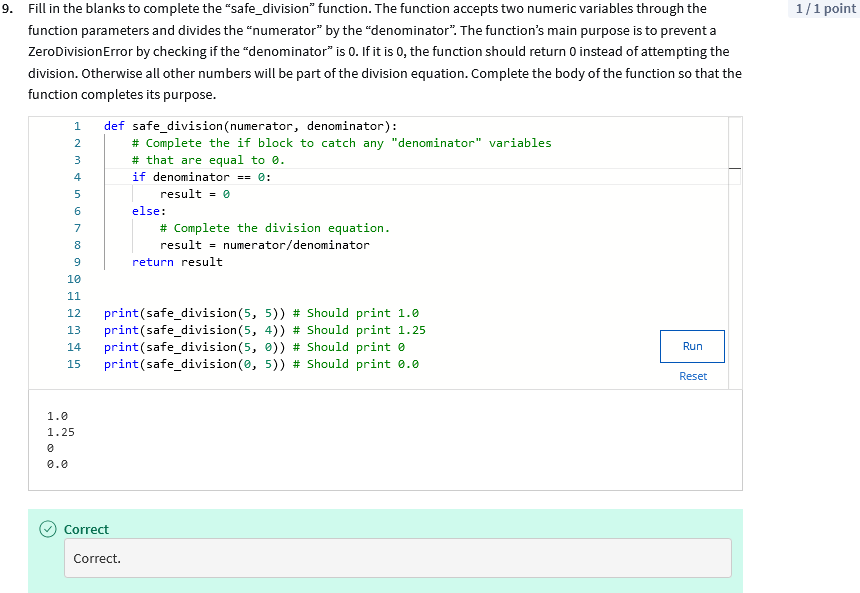


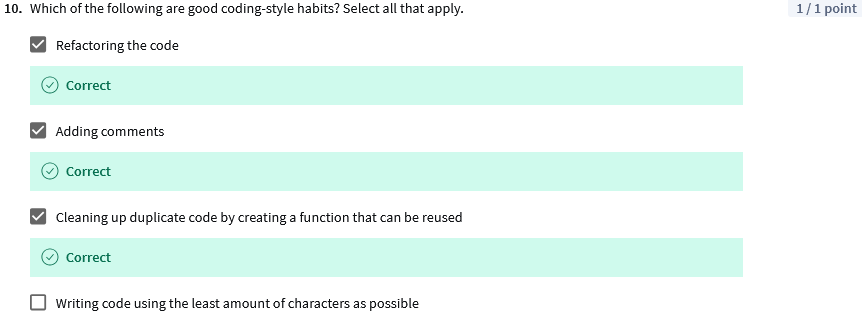






Missclick, obviously this is false





### Basic Syntax Wrap Up

You just completed your second module and learned a whole lot about Python syntax. Congrats. We've learned how to operate with different data types and how to create our own variables and expressions. We've defined our first functions and learned how to make them return values so that they're more reusable. We then dove into creating branches in our scripts which lets them act in different ways depending on the values of our variables. We learned a lot of new and very powerful stuff. Knowing how to structure your code and functions and how to make your code act in different ways depending on values is what allows us to tell our computer what to do. We'll keep using these tools throughout the course as we move on to more complex and interesting things. Next step, you can put everything you've learned to the test in the next graded assessment. Don't worry if you don't feel ready yet. Remember that you can re-watch the videos and do the practice quizzes as many times as you need to make sure you fully understand everything we've covered. When you're ready for the test, take your time and best of luck. I'll catch you after you've finished in the next module, where we'll learn all about loops. See you there.