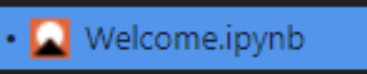
# Python Basics Fundamentals for BQuant

# Access:BQIQ 🡪

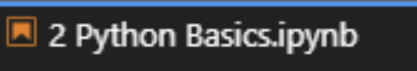
🡪

# 

🡪



🡪



Markdown & HTML

HTML is an acronym standing for "Hyper Text Markup Language" and is the standard language for creating Web pages. HTML is ubiquitous on the Web - every website you've ever visited has rendered HTML to your browser. Markdown is a lightweight markup language with user-friendly syntax. For BQuant purposes, you can think of Markdown as a kind-of simplified version of HTML.

As you develop your Jupyter Notebooks, Markdown and HTML are handy tools for adding descriptive text and documenting your code. If you're already familiar with HTML, feel free to use it in your Markdown cells. You can even use in-line CSS styling to further customize the look and layout of your projects.

Markdown cells are typically followed by the Python cells that they describe, as in the example screenshot below.

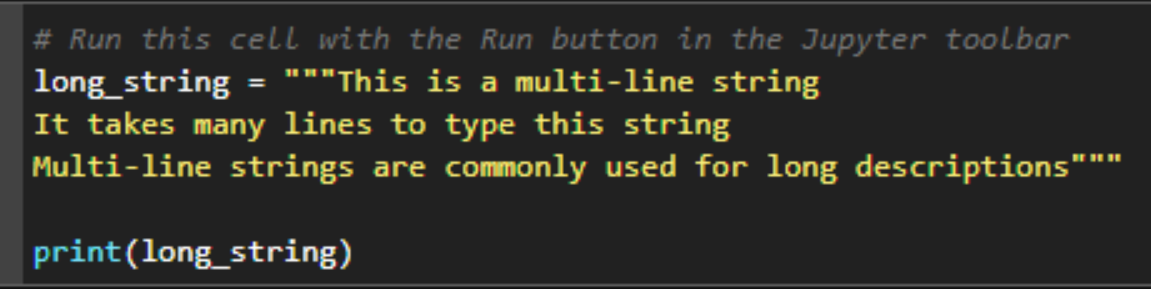
# Python Data Types Introduction

While programming in Python, you'll work with several different types of data. A person's name ( "Jane Doe" ) is a type of data, but not the same type as their age ( 35 ). The person's age is a different type of data than their height ( 1.6 meters ), which is a different type than whether or not they hold a university degree ( True/False). All of these pieces of data may be necessary descriptors even though they fall into different classifications.

* strings - text data  ( name: "Jane Doe" )
* integers - whole numbers  ( age: 35 )
* floats - floating decimal points  ( height: 1.6 meters )
* booleans - binary  ( degree: True/False )

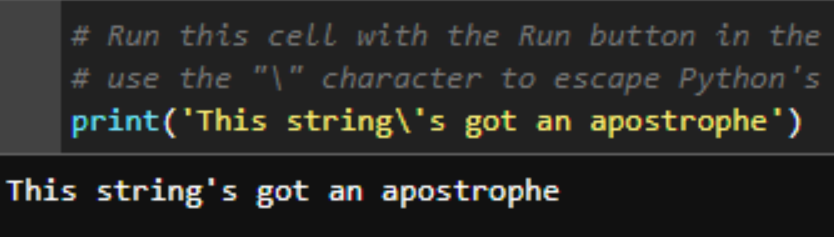
#### Multi-line Strings

If you need to write a long string occupying several lines of code, you can use triple quotes (""") at the beginning and end of the string to indicate to Python that everything between them should be considered one string. Triple single quotes (''') will also work.



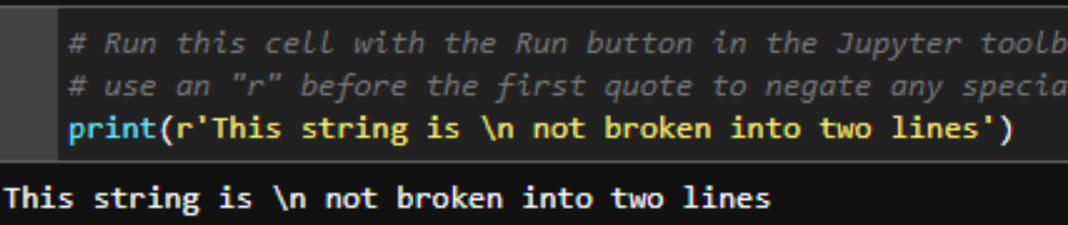
#### Using the escape character

The backslash character (\) in a Python string is special. It allows you to "escape" the normal function of the next character. In the code example below, the single quote after the word "string" would normally close the string. By using a backslash before the single quote, we can escape this behavior and use the single quote as part of the string.



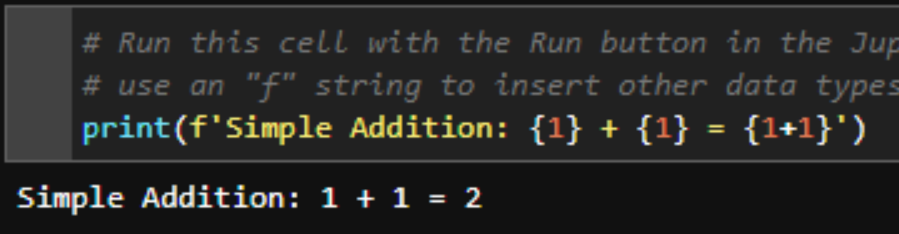
#### Use "r" strings for literals

Sometimes you may want Python to just accept the raw version of a string, ignoring any special characters that may appear in it. You could use the backslash for each special character, but a more efficient way would be to use an "r" string. A string preceeded by the letter "r" will tell Python to ignore any special characters that may appear as part of the text.



#### Use "f" strings to insert other data types

"F" strings are a handy addition to newer versions of the Python language. They allow you to mix data types inside a string and perform operations. Use the letter "f" before a string and wrap any non-strings in curly braces { }. See the code snippet below as an example.



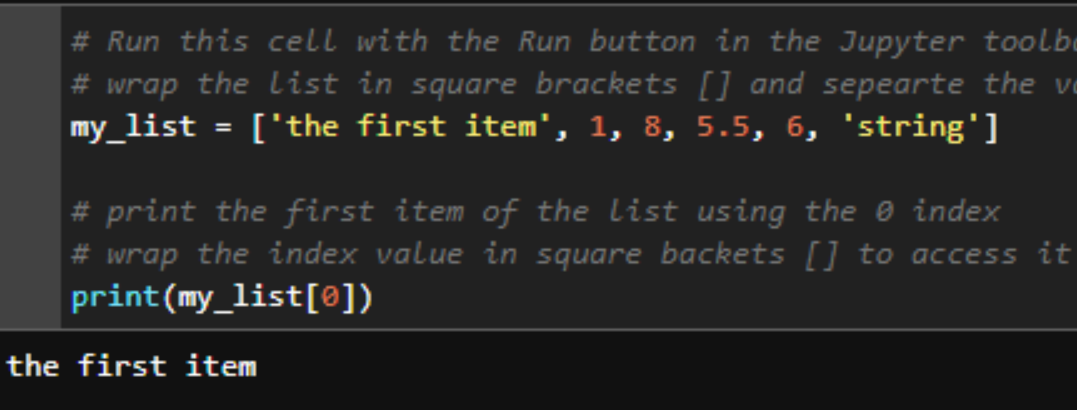
# Sequence Types

Python offers two main sequence data types: list and tuple. Lists are wrapped in square brackets [ ], and tuples are commonly declared with parentheses ( ). Lists and Tuples store an ordered sequence of values seperated by commas. The values inside a list or tuple can be of any data type: numbers, strings, and even other lists or tuples.

You can access the values stored inside a list/tuple using an "index." Each item is assigned a location number starting with 0 for the first position, 1 for the second position, and so on. This is known as "zero-based" indexing since the first number is represented by the index number 0.

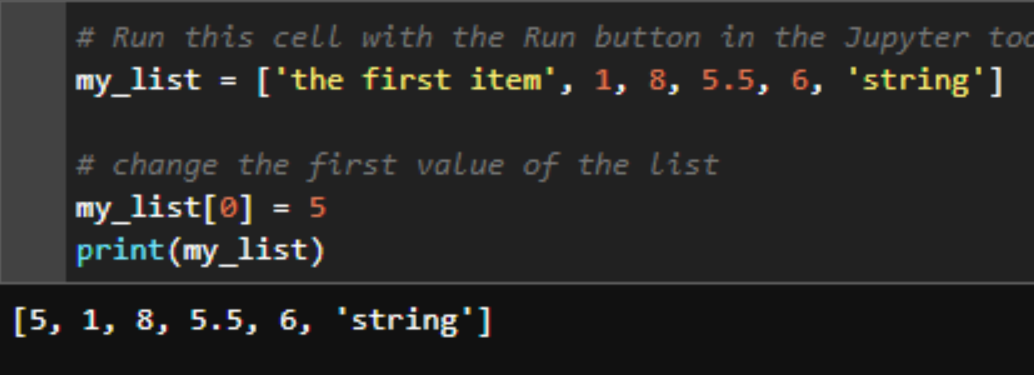
#### Lists

The code below will declare a list and print the first item of that list. Note that lists can mix and match data types. The list below contains integers, floats, and strings.



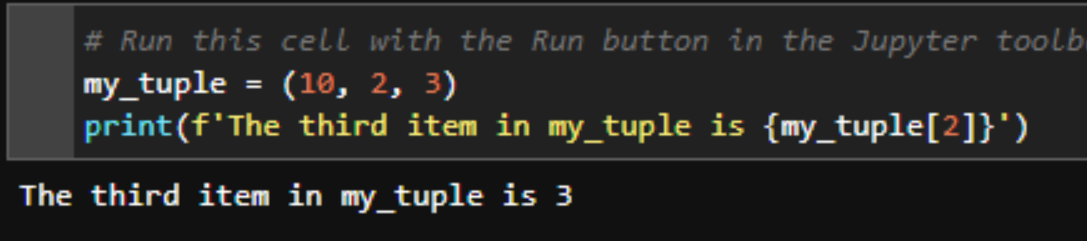
#### Change an item in a list

A list is an example of a "mutable" data type. This means that once the list is declared, the items inside can be changed. In the code snippet below, we change the first item in the sequence to 5 and print out the changed list.



#### Tuples

The code below will declare a tuple and use an "f" string to print the third value. Remember that Python uses "zero-based" indexing, meaning that the first item in the list is indexed to 0, the second item is indexed to 1, and so we call my\_tuple[2] to access the third value.

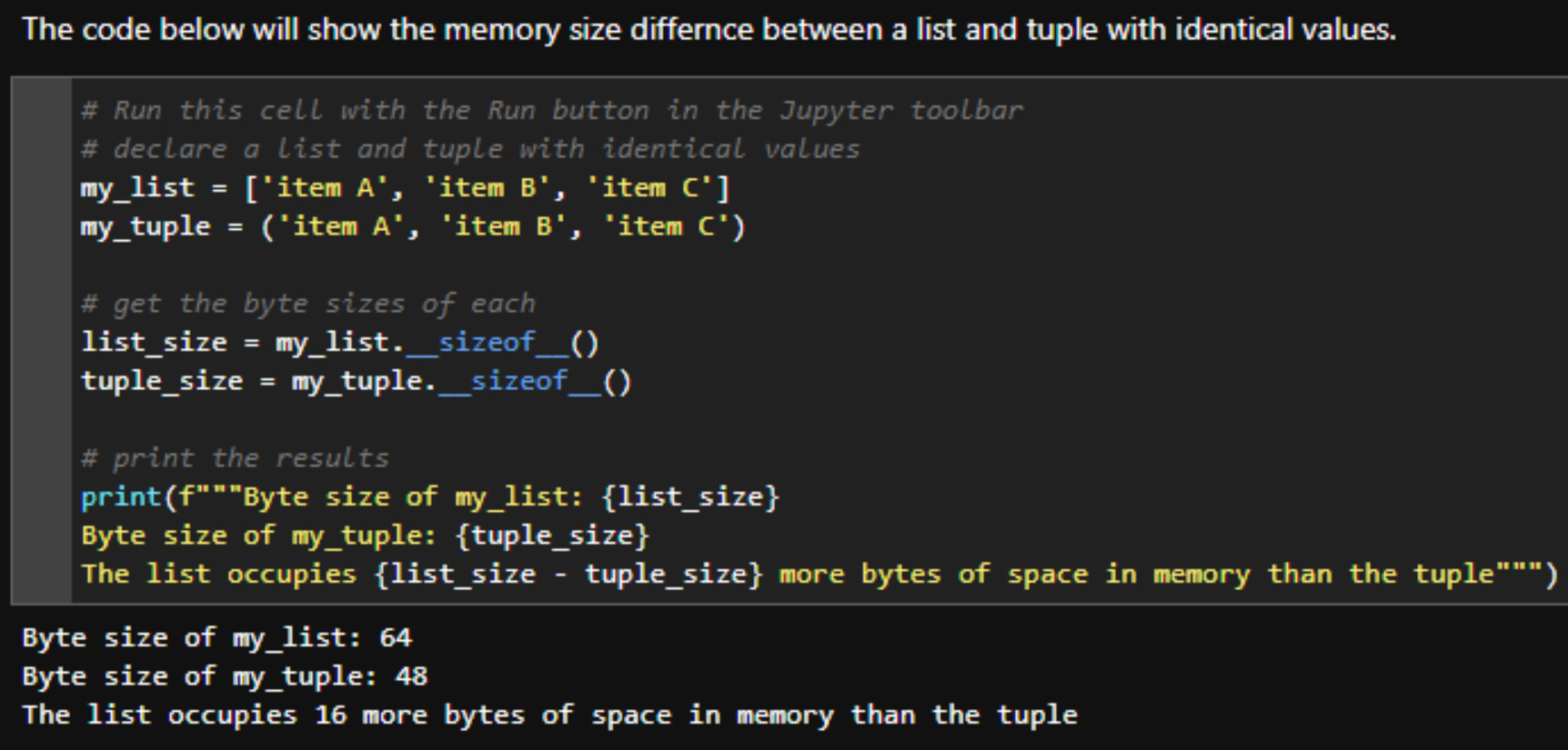


#### Tuples are not "mutable"

The main difference between a tuple and a list is that tuples are not "mutable." This means that once the tuple has been populated with items, those items cannot be changed. Tuples also have a fixed length after they are declared, so you cannot add more items to a tuple later on. If you try to run the code below, Python will display an error message.

#### Should I use a Tuple or List?

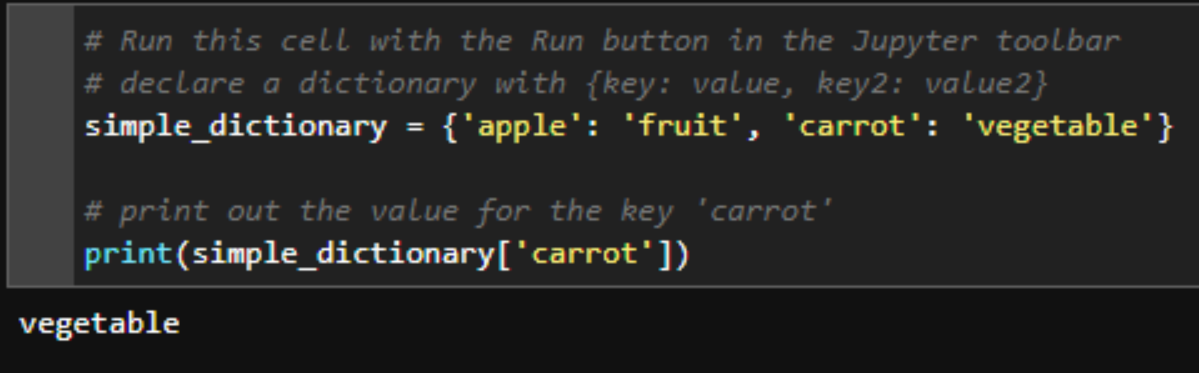
If tuples are immutable, why would we ever use them instead of a list? The main reason is that immutable objects like tuples occupy less space in memory, and operations can therefore be performed faster on tuples than on lists. This might not make much of a difference when working with a small amount of data, but it will be important to remember when working with vast amounts of information. In general, when your data sequence has a fixed length and the values inside will never change, use a tuple. Otherwise use a list!  
  
The code below will show the memory size differnce between a list and tuple with identical values.



# Dictionaries

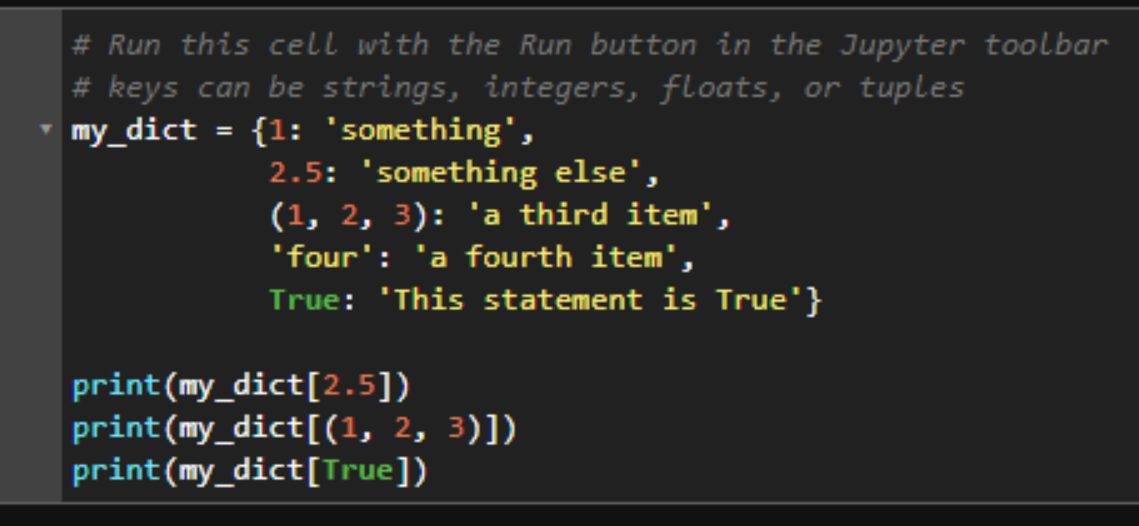
Dictionaries are Python objects that allow you to store pairs of data in a key/value relationship. They are particularly useful for mapping relationships between items. You can declare a dictionary by separating a key/value pair with a colon and surrounding the object with curly braces { }. If a dictionary has multiple key/value pairs, each pair is separated with a comma.

Dictionaries are unordered, meaning that the order in which you declare the key/value pairs is irrelevant. You can access a dictionary's value by calling its key wrapped in square brackets. See the code snippet below for an example.



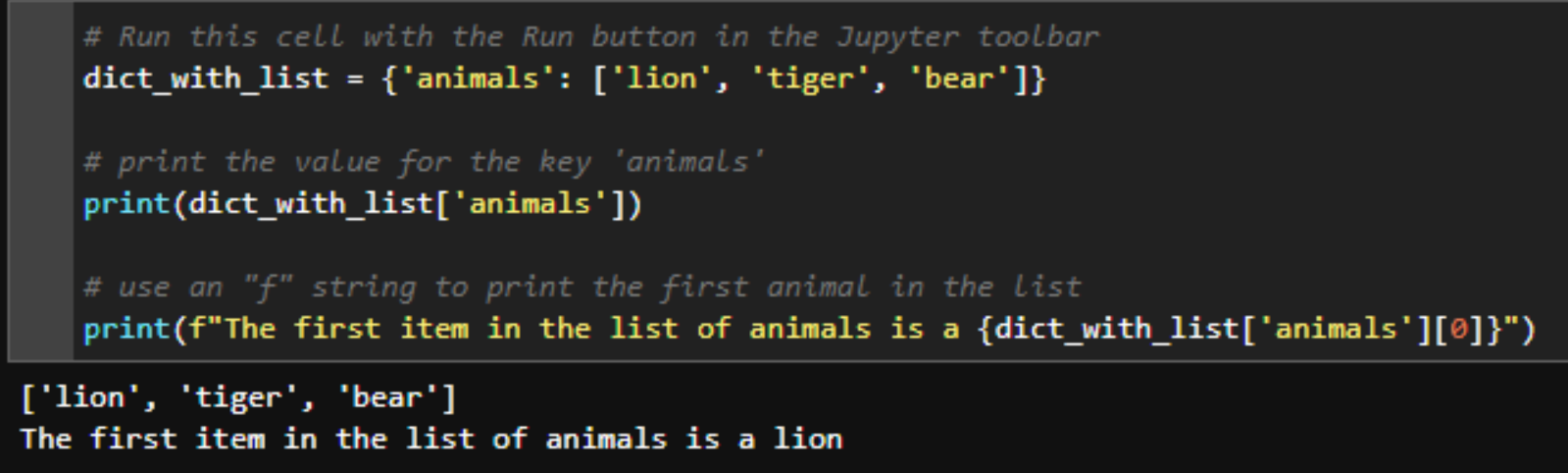
#### Use numbers, strings, and tuples as dictionary keys

Strings are the most common choice for dictionary keys, but any immutable data type can be used. This includes integers, floats, tuples, and Booleans. Dictionaries and lists are mutable types, and cannot be used as dictionary keys.



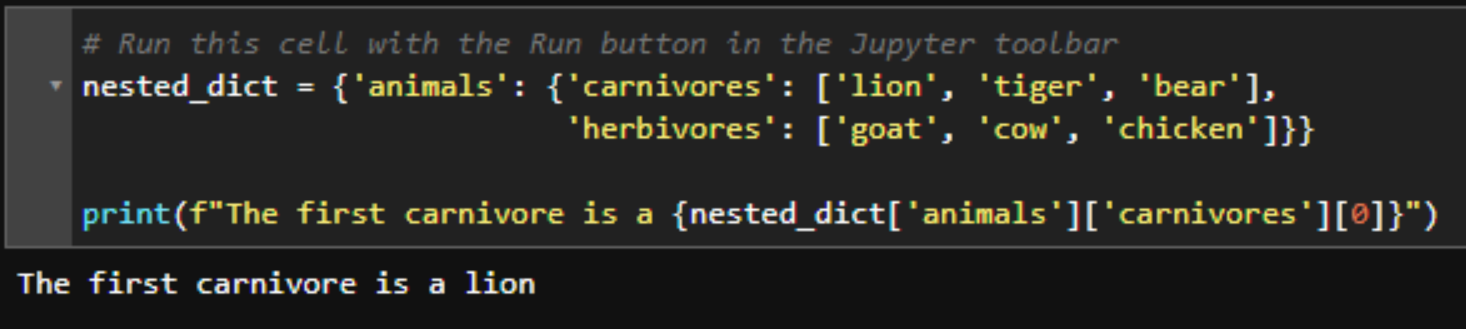
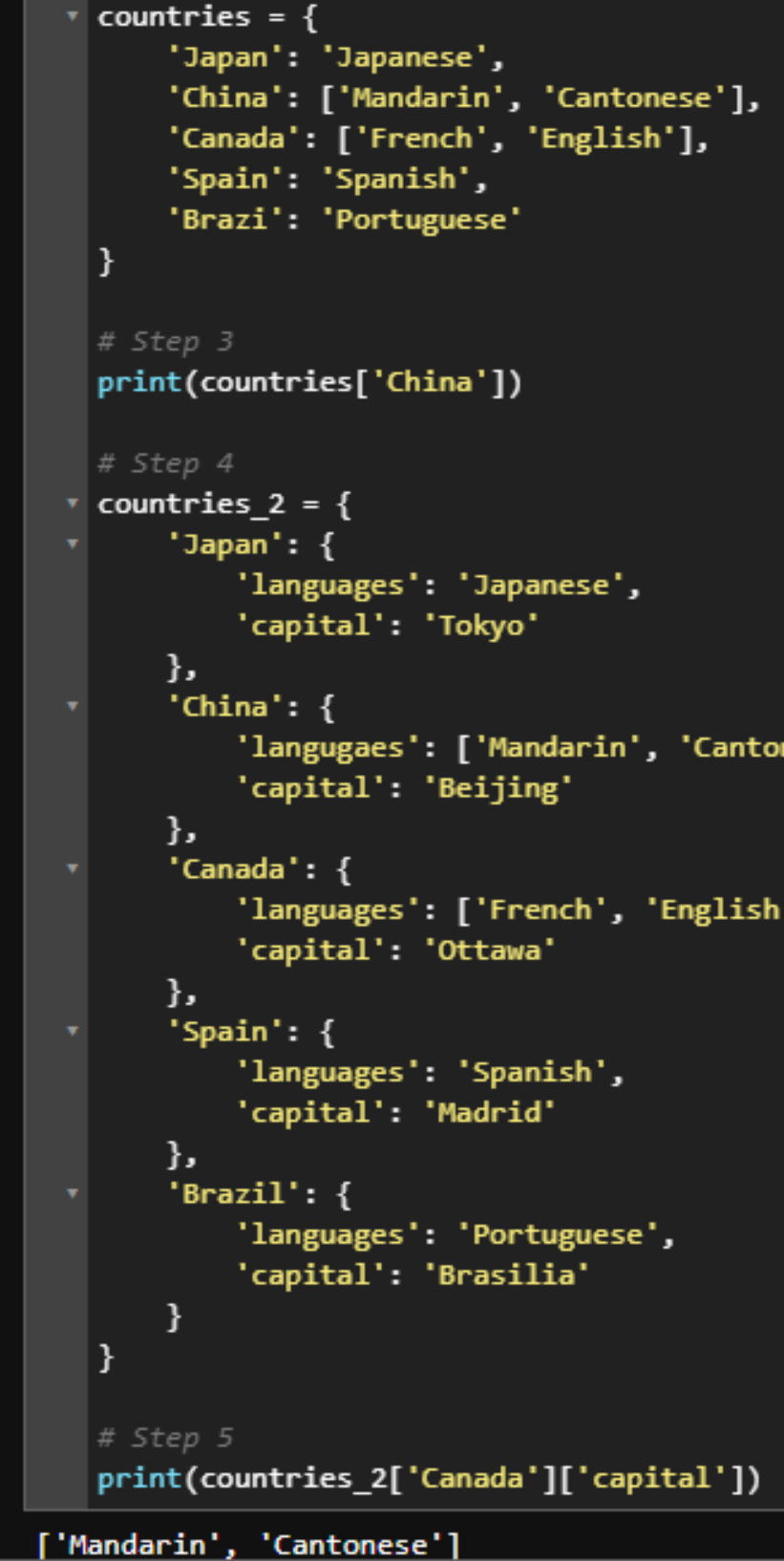
#### Any data type can be a dictionary value

Dictionary values, unlike keys, can be any data type regardless of mutability. The code below uses a list of animals as the value for the key 'animals'. We can access the first animal in that list by calling dict\_with\_list['animals'] for the list and then adding [0] to access the first item in that list.



#### Dictionaries can be nested inside other dictionaries

As we discovered in the example above, dictionary values can be any data type. This includes other dictionaries! Nesting dictionaries is a common technique if there are subsequent relationships that also should be mapped with a key/value pair. The code below segregates lists of animals into carnivores and herbivores. We can access each subsequent level of the dictionary by chaining keys together (i.e. nested\_dict['animals']['carnivores'] will access the list of carnivorous animals.

#### Variables

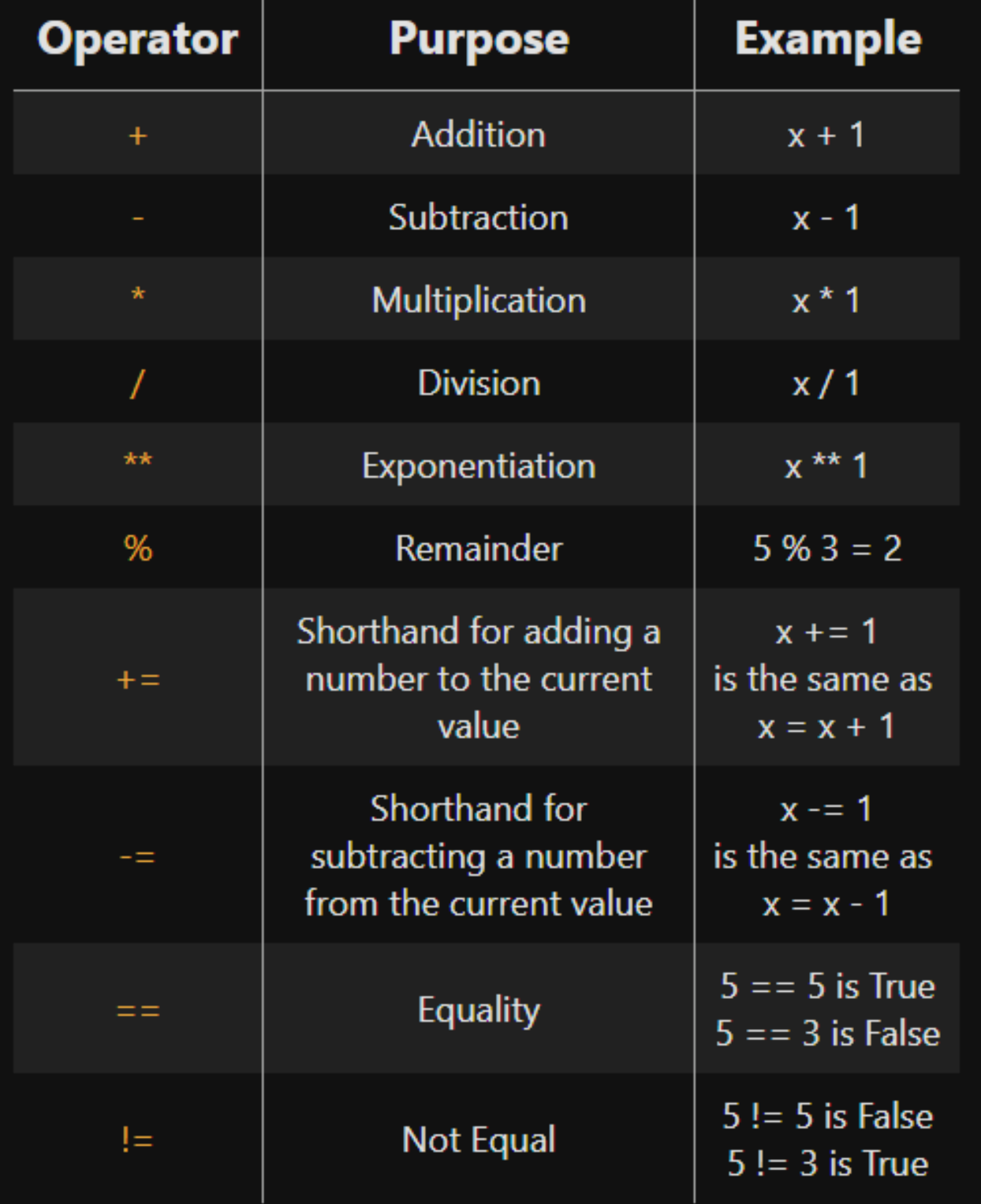
Variables in Python are used to temporarily store a data type. You can think of a variable as an empty box. When "declaring" a variable, you name the box and place a data type inside. You can change what's in the box, or empty it altogether. We declare variables with the = operator. In fact, if you've been following this notebook in order, you've already declared a few variables in the data types section.  
  
There are a few rules to keep in mind when declaring variables.

* The variable name must start with a letter or an underscore. Numbers can be used in the name, just not at the beginning.
* The variable name can only contain alpha-numeric characters and underscores. No other special characters are allowed.
* Variable names are case sensitive. My\_Variable is different than my\_variable which is different than mY\_vArIAble

While not required, it's good practice to use lower case letters, separating words by an underscore.

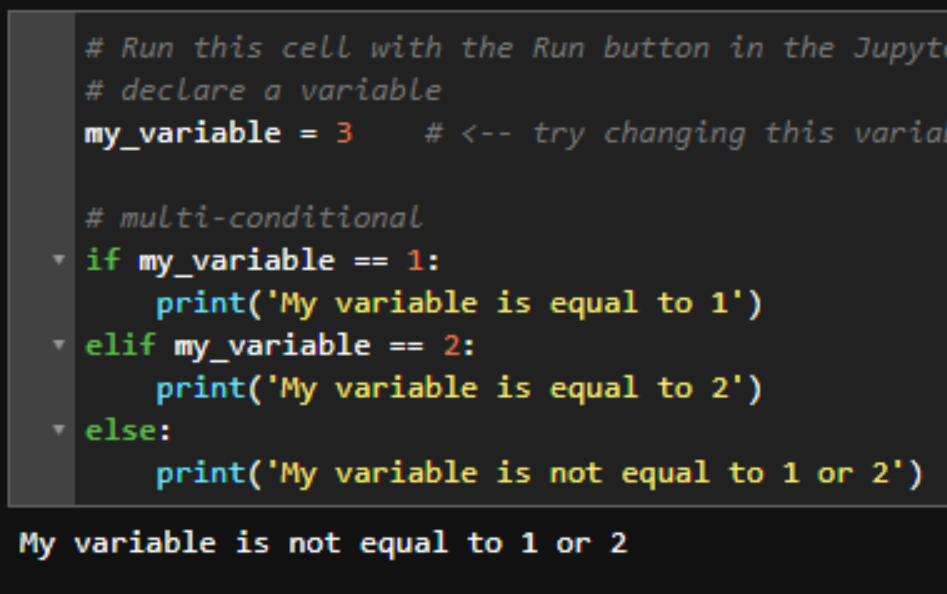
#### Operators

Operators are special symbols in Python that carry out a specific function on a variable or data type. These include some of the mathematical symbols you might expect (+, -, \*, and /), but there are also logical and comparison operators such as and, or, >, and <. The table below displays some of the most commonly used operators. See this link for a [complete list of operators](https://www.w3schools.com/python/python_operators.asp).



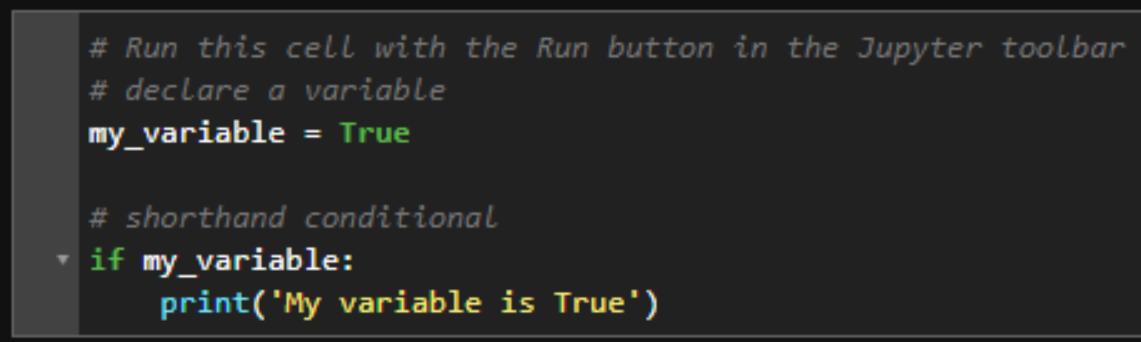
#### Multiple Conditionals

You can use elif ("else if") and else to specify other conditional statements if the first one fails.



#### Conditionals with Booleans

If you're referencing a Boolean variable in a conditional statement, you can exclude an operator from the conditional.



# Loops Automate Repetitive Tasks

Oftentimes in programming you'll need to repeat a certain task over and over again until a condition is met. For example, consider a situation in which you need to add 1 to every item in a list of numbers. The operation x + 1 would need to be carried out for each x in the list, and Python should stop this behavior when it runs out of items in the list. Alternatively, consider a scenario where you want to add 1 to a number until the result is a prime number. Rather than iterating over a list, Python is repeating the operation x += 1 until a given condition is met. Loops are a perfect tool for accomplishing these tasks.

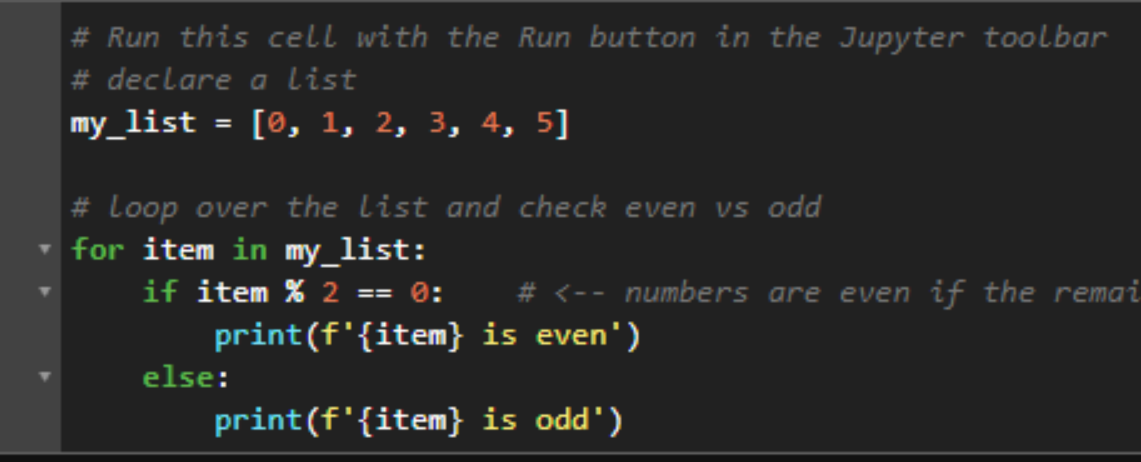
There are two types of loops in Python: for loops and while loops. For loops are best for accomplishing the first type of scenario in which you need to iterate over a given set of values. The while loop is best for the second type of scenario, repeating a task until a given condition is met. The two types of loops are interchangable: any for loop can be written as a while loop and vice versa, but choosing the right loop to apply to a given situation could save you from writing extra lines of code.

#### For Loops

The example below shows the syntax for a basic for loop. This code prints every item in a list. Notice that the syntax is similar to the conditional statements above: a declarative statement starting with for and ending in a colon. All subsequent lines are indented.

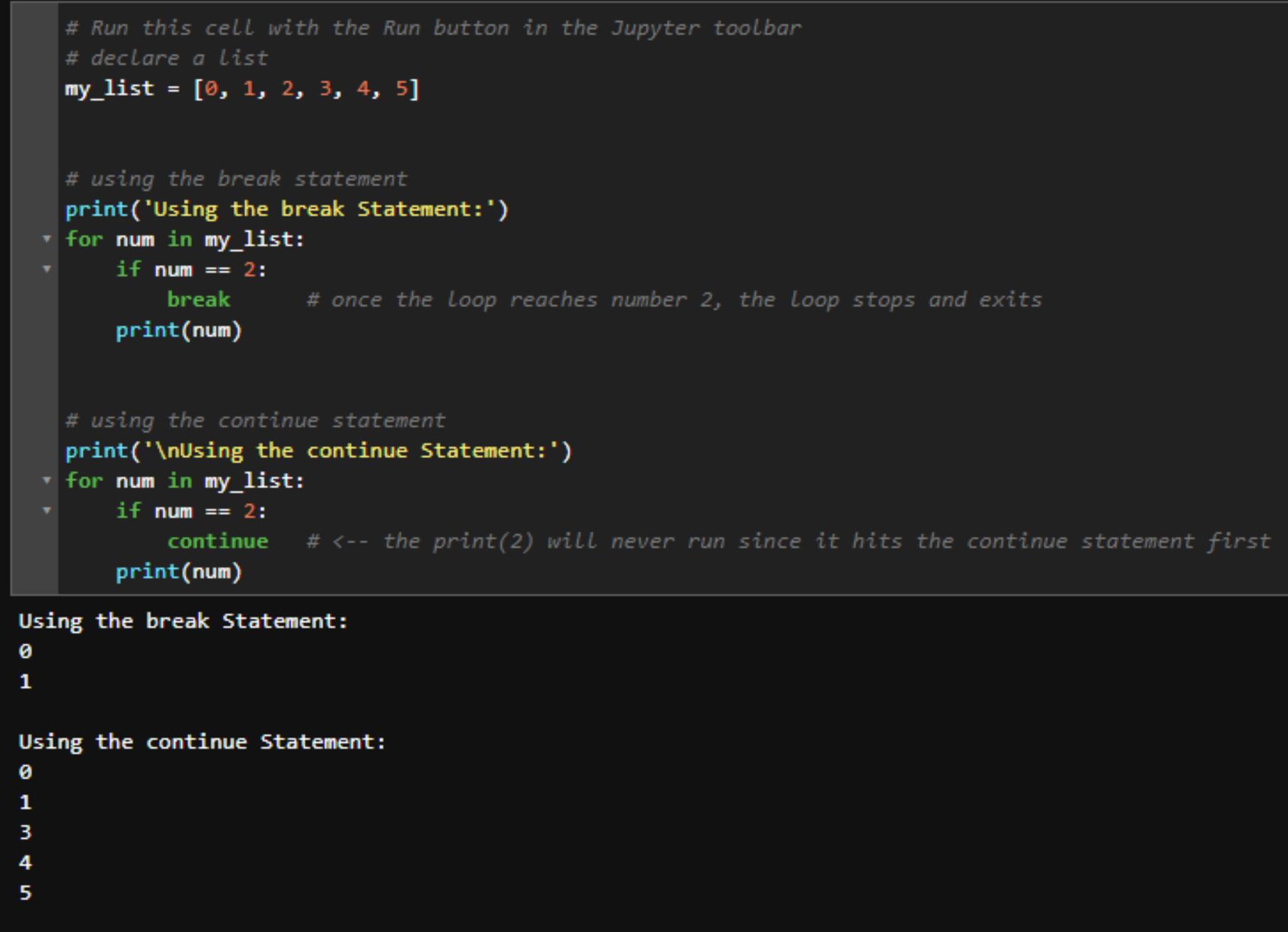
#### Conditionals in loops

Loops are often combined with conditionals to perform some IF → THEN logic for each iteration. The example below uses a conditional to determine if numbers are even or odd and prints that information to the screen using an "f" string. Notice how the for loop and the conditional are indented.



#### Break and Continue Statements

Python provides two statements for controlling the flow of iteration inside the loop. The break statement will stop and exit a loop when it is called. The continue statement will start the next iteration without executing any of the code below it. See how these statements work in the example cell below.

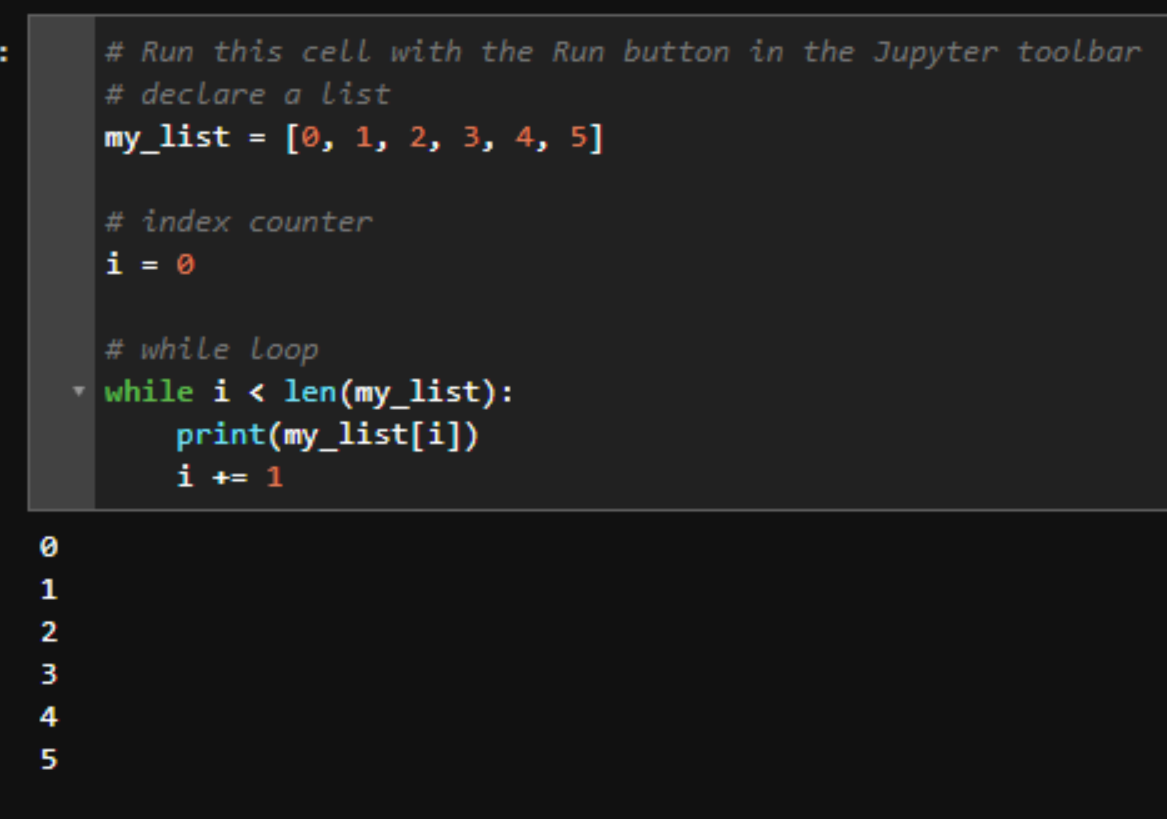


#### The Range Object

range() is a handy object in writing loops. A range object is used to generate a sequence of numbers between an inclusive minimum and exclusive maximum, and has many applications in writing loops. By default, the minimum value of a range is 0 and it increments by 1, so calling list(range(6)) will generate the list of numbers [0, 1, 2, 3, 4, 5].  
  
In our examples so far, my\_list has been an ordered sequence of numbers from 0 to 5. We can replicate our example loops with range(6) without the need to explicitly declare the list.

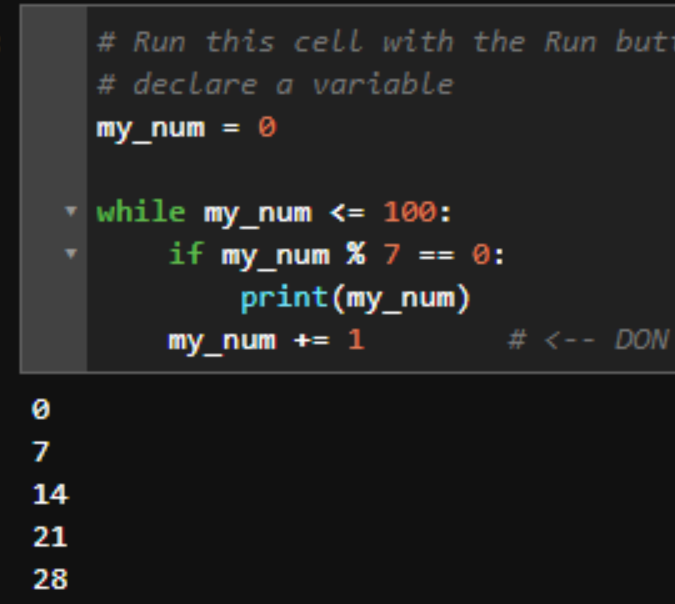
#### While Loops

All the for loops we have learned so far can also be written as while loops. In the example below, we declare a variable i and set it equal to 0. This variable is meant to keep track of our list index. We print each item in the list at the "i"th position, and then increase i by 1 at the end of each iteration. We should only continue this behavior while i is less than the length of the list, since Python uses zero-based indexing.



#### Better Use Cases for While Loops

The syntax for our first while loop above is messier and more complex than using a for loop. Better use cases for while loops often don't involve a given set of values. It will make more sense to use a while loop when working with less tangible data than a list. The example below declares a variable my\_num and sets it equal to 0. We then start counting up until we reach 100. Every time the resulting number is divisible by 7, we print that to the screen. We can't see the number counting up in the background like we can see the items in a list, but that's ok. All that's important is that numbers divisible by 7 are printed to the screen. While loops are better suited to these scenarios than for loops.



#### Nesting Loops

Loops can be nested inside each other. The syntax can be tricky here, but remember that Python completes the inner loop for every turn of the outer loop. See the example below to see how it works. In total, the print() function is called 9 times in the following code.

#### Using Loops to Generate Lists

In Python, loops provide a shorthand way to create a list. We can surround a simple loop statement in square brackets [ ] to generate a list and store it in a variable. Consider the two pieces of code below that have identical outputs. Oftentimes you will see list declarations written this way, but don't be thrown by the shorthand syntax. Once you gain a little experience declaring lists this way, you can save yourself a lot of code.

# Functions Reusable Bits of Code

Python functions are reusable chunks of code that perform a specific task. If you've been following this notebook in order, you've already used the print() and len() functions. These functions are built into the Python language, but you can also create your own. Functions are especially useful when you need to store and reuse a process that manipulates data. Define a function by calling the def keyword ("def" for "define") followed by the name of the function, a pair of parentheses ( ), and a colon.

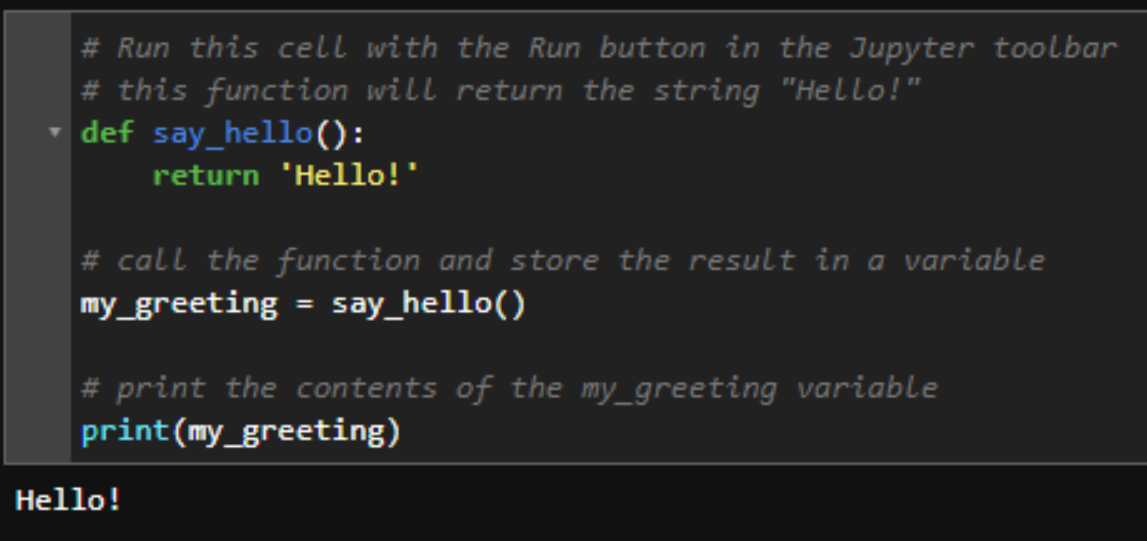
Similar to variables, there are a few rules to keep in mind when naming your functions.

* They must start with a letter or underscore. Numbers are allowed, just not at the beginning.
* They can't be the same as a Python keyword (i.e. don't name your function print since that already exists)
* Function names can contain only alpha-numeric characters and underscores

It's best practice, but not required, that function names be lowercase, short, and descriptive.

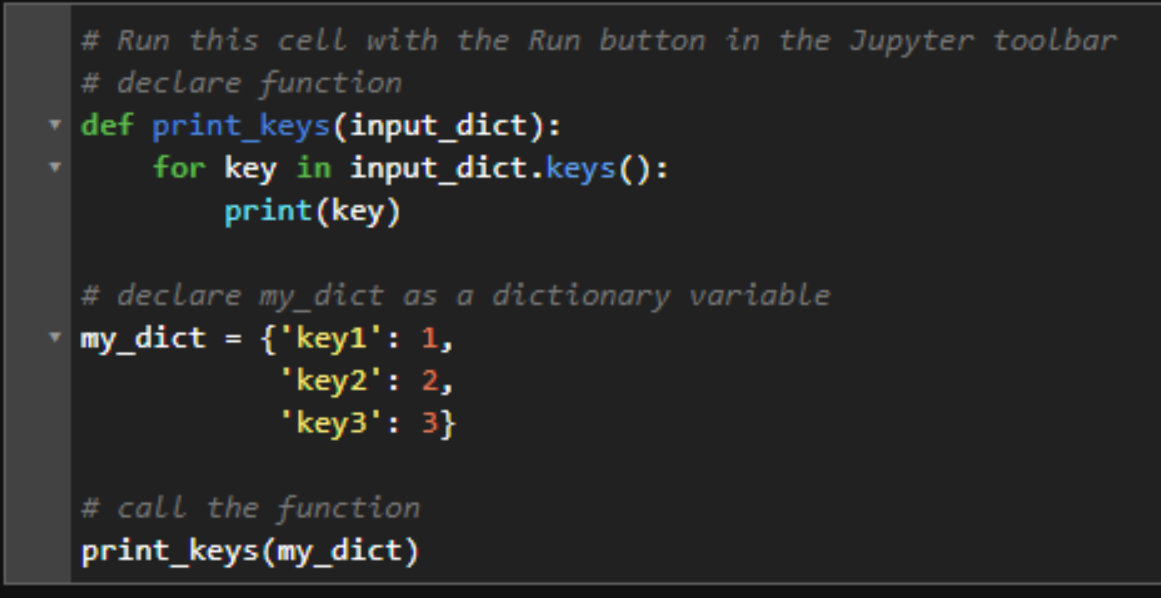
#### The Return Keyword

Functions that print to the screen have limited use. More often, you will want the function to give you a piece of data after it finishes its execution. The return statement will exit the function at the point it's called, "returning" something to the developer. The function below doesn't print "Hello!" to the screen, but rather returns that string as an object which can be stored as a variable.



#### Non-numerical Arguments

Function arguments can be any data type, not just numbers. The function below, for example, takes a dictionary as an argument and prints out the dictionary keys.



#### Keyword Arguments

You can also pass arguments to a function using the syntax key = value. This will allow you to ignore the order in which the arguments are declared in the function. The example below uses the same subtract\_these function as above, but notice how the function is called in the last line of the cell. The result is the same number 3.5 even though the arguments were passed in reverse order.

