

# Python Textbook

Nicole Oni\*

February 2026

## Contents

<b>1 Basics</b>	<b>3</b>
1.1 Introduction . . . . .	3
1.2 Python Syntax . . . . .	4
1.2.1 Python Indentation . . . . .	4
1.2.2 Python Statements . . . . .	4
1.2.3 Challenge . . . . .	5
1.3 Python Output / Print . . . . .	5
1.3.1 Print Without a New Line . . . . .	5
1.3.2 Print Numbers . . . . .	6
1.3.3 Mixing Text and Numbers . . . . .	6
1.3.4 Challenge . . . . .	6
1.4 Comments . . . . .	6
1.4.1 Multiline Comments . . . . .	7
1.4.2 Challenge . . . . .	7
1.5 Python Variables . . . . .	8
1.5.1 Get the Type . . . . .	8
1.5.2 Other . . . . .	8
1.5.3 Variable Names . . . . .	9
1.5.4 Assign Multiple Values . . . . .	9
1.5.5 Unpack a Collection . . . . .	10
1.5.6 Output Variables . . . . .	10
1.5.7 Global Variables . . . . .	11
1.5.8 Challenge . . . . .	12
1.6 Python Data Types . . . . .	12
1.6.1 Built-in Data Types . . . . .	12
1.6.2 Getting the Data Type . . . . .	13
1.6.3 Setting the Specific Data Type . . . . .	13
1.6.4 Challenge . . . . .	14
1.7 Python Numbers . . . . .	14
1.7.1 Random Number . . . . .	15
1.7.2 Challenge . . . . .	15
1.8 Python Casting . . . . .	15
1.8.1 Challenge . . . . .	16
1.9 Python Strings . . . . .	17

---

\*Using the w3schools tutorials.

1.9.1	Quotes Inside Quotes . . . . .	17
1.9.2	Assign String to a Variable . . . . .	17
1.9.3	Multiline Strings . . . . .	17
1.9.4	Strings are Arrays . . . . .	18
1.9.5	Looping Through a String . . . . .	18
1.9.6	String Length . . . . .	18
1.9.7	Check String . . . . .	18
1.9.8	Slicing Strings . . . . .	19
1.9.9	String Concatenation . . . . .	19
1.9.10	String Format . . . . .	20
1.9.11	Escape Characters . . . . .	21
1.9.12	String Methods . . . . .	21
1.9.13	Challenge . . . . .	23
1.10	Python Booleans . . . . .	23
1.10.1	Evaluate Values and Variables . . . . .	24
1.10.2	Most Values are True . . . . .	24
1.10.3	Some Values are False . . . . .	24
1.10.4	Functions can Return a Boolean . . . . .	25
1.10.5	Challenge . . . . .	25
1.11	Python Operators . . . . .	26
1.11.1	Arithmetic Operators . . . . .	26
1.11.2	Assignment Operators . . . . .	27
1.11.3	Comparison Operators . . . . .	27
1.11.4	Logical Operators . . . . .	28
1.11.5	Identity Operators . . . . .	28
1.11.6	Membership Operators . . . . .	29
1.11.7	Bitwise Operators . . . . .	29

# 1 Basics

## 1.1 Introduction

Python is a popular programming language. It was created by Guido van Rossum, and released in 1991.

It is used for:

- web development (server side),
- software development,
- mathematics,
- system scripting

Python can:

- be used on a server to create web applications.
- be used alongside software to create workflows.
- connect to database systems. It can also read and modify files.
- be used to handle big data and perform complex mathematics.
- be used for rapid prototyping, or for production-ready software development.

Python Syntax compared to other programming languages:

- Python was designed for readability, and has some similarities to the English language with influence from mathematics.
- Python uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses.
- Python relies on indentation, using whitespace, to define scope; such as the scope of loops, functions and classes. Other programming languages often use curly-brackets for this purpose.

Other:

- The extension for Python files is .py.
- The command line syntax for checking if Python is installed on your computer (and also to check the python version) is `python --version`.
- Use `exit()` to exit the Python command line interface.

## 1.2 Python Syntax

### 1.2.1 Python Indentation

- Indentation refers to the spaces at the beginning of a code line.
- Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important.
- Python uses indentation to indicate a block of code.

For example, Python will give you an error if you skip the indentation:

```
1 if 5 > 2:  
2     print("Five is greater than two!")
```

Correct version:

```
1 if 5 > 2:  
2     print("Five is greater than two!")
```

### 1.2.2 Python Statements

- A **computer program** is a list of “instructions” to be “executed” by a computer.
- In a programming language, these programming instructions are called **statements**.
- In Python, a statement usually ends when the line ends. You do *not* need to use a semicolon (;) like in many other programming languages (for example, Java or C).

Example:

```
1 print("Python is fun!")
```

Many Statements:

- Most Python programs contain many statements.
- The statements are executed one by one, in the same order as they are written.

Example:

```
1 print("Hello World!")  
2 print("Have a good day.")  
3 print("Learning Python is fun!")
```

Semicolons:

- Semicolons are optional in Python.

- You can write multiple statements on one line by separating them with ; but this is rarely used because it makes it hard to read.
- If you put two statements on the same line without a separator (newline or ; , Python will give an error.

Example:

```
1 print("Hello"); print("How are you?"); print("Bye bye!")
```

**Best practice:** Put each statement on its own line so your code is easy to understand.

### 1.2.3 Challenge

Inside the editor, complete the following steps:

1. Write a statement that prints “Hello World!”
2. Write a statement that prints “Have a good day.”
3. Write a statement that prints “Learning Python is fun!”

Solution:

```
1 print("Hello World!")
2 print("Have a good day.")
3 print("Learning Python is fun!")
```

## 1.3 Python Output / Print

- The `print()` function can be used to display text or output values.
- You can use the `print()` function as many times as you want. Each cell prints text on a new line by default.
- Text in Python must be inside quotes. You can use either " double quotes or ' single quotes.
- If you forget to put the text inside quotes. Python will give an error.

```
1 print("Hello World!")
2 print("This will work!")
3 print('This will also work!')
4 print(This will cause an error)
```

### 1.3.1 Print Without a New Line

- By default, the `print()` function ends with a new line.
- If you want to print multiple words on the same line, you can use the `end` parameter.

Example:

```
1 print("Hello World!", end=" ")
2 print("I will print on the same line.")
```

Note that we add a space after `end=" "` for better readability.

### 1.3.2 Print Numbers

- You can also use the `print()` function to display numbers.
- However, unlike text, we don't put numbers inside double quotes.
- You can also do maths inside the `print()` function.

Example:

```
1 print(3)
2 print(358)
3 print(2*5)
```

### 1.3.3 Mixing Text and Numbers

You can combine text and numbers in one output by separating them with a comma. For example:

```
1 print("I am", 35, "years old.")
```

### 1.3.4 Challenge

Inside the editor, complete the following steps:

1. Print the text “**I am**” and the number **25** in one print statement.

Solution:

```
1 print("I am", 25)
```

## 1.4 Comments

- Python has commenting capability for the purpose of in-code documentation.
- Uses of comments:
  - Comments can be used to explain Python code.
  - Comments can be used to make the code more readable.
  - Comments can be used to prevent execution when testing code.
- Comments start with a `#`, and Python will render the rest of the line as a comment.

- Comments can be placed at the end of a line, and Python will ignore the rest of the line.

Example:

```

1 # This is a comment
2 print("Hello, World!")
3
4 print("Hello, World!") # This is a comment

```

#### 1.4.1 Multiline Comments

Option 1:

- Python does not really have a syntax for multiline comments.
- To add a multiline comment you could insert a # for each line.

Option 2:

- Or, not quite as intended, you can use a **multiline string**.
- Since Python will ignore string literals that are not assigned to a variable, you can add a multiline string (triple quotes) in your code, and place your comment inside it.
- As long as the string is not assigned to a variable, Python will read the code, but then ignore it, and you have made a multiline comment.

Example:

```

1 # This is a comment
2 # written in
3 # more than just one line
4
5 print("Hello, World!")
6
7 """
8 This is a comment
9 written in
10 more than just one line
11 """

```

#### 1.4.2 Challenge

Inside the editor, complete the following steps:

- Add a single-line comment that says **This is a comment**
- Comment out the line `print("This should not run")` so it does not execute.
- Add a multiline comment (using triple quotes) that says **This is a multiline comment**

Solution:

```
1 # This is a comment
2 # print("This should not run")
3 """
4 This is
5 a multiline
6 comment
7 """
```

## 1.5 Python Variables

- Variables are containers for storing data values.
- Python has no command for declaring a variable.
- A variable is created the moment you first assign a value to it.

Example:

```
1 x = 5
2 y = "John"
3 print(x)
4 print(y)
```

Variables do not need to be declared with any particular *type* and can even change type after they have been set. For example:

```
1 x = 4 # x is of type int
2 x = "Sally" # x is now of type str
3 print(x)
```

### 1.5.1 Get the Type

You can get the data type of a variable with the `type()` function.

Example:

```
1 x = 5
2 y = "John"
3 print(type(x))
4 print(type(y))
```

### 1.5.2 Other

- String variables can be declared either by using single or double quotes.
- Variable names are case-sensitive.

Example:

```
1 x = "John"
2 # is the same as
3 x = 'John'
4
5 a = 4
6 A = "Sally"
7 # A will not overwrite a
```

### 1.5.3 Variable Names

A variable can have a short name (like `x` and `y`) or a more descriptive name (`age`, `carname`, `total_volume`).

Rules for Python variables:

- A variable name must start with a letter or the underscore character.
- A variable name cannot start with a number.
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and `_`).
- Variable names are case-sensitive (`age`, `Age` and `AGE` are three different variables).
- A variable name cannot be any of the Python keywords.

Legal variable names:

```
1 myvar = "John"
2 my_var = "John"
3 _my_var = "John"
4 myVar = "John"
5 MYVAR = "John"
6 myvar2 = "John"
```

Illegal variable names:

```
1 2myvar= "John"
2 my-var = "John"
3 my var = "John"
```

Multi Words Variable Names:

Variable names with more than one word can be difficult to read. There are several techniques you can use to make them more readable:

- **Camel Case:** Each word, except the first, starts with a capital letter:  
`myVariableName = "John"`
- **Pascal Case:** Each word starts with a capital letter: `MyVariableName = "John"`
- **Snake Case:** Each word is separated by an underscore character:  
`my_variable_name = "John"`

### 1.5.4 Assign Multiple Values

Many Values to Multiple Variables:

- Python allows you to assign values to multiple variables in one line.
- Make sure the number of variables matches the number of values, or else you will get an error.

Example:

```
1 x, y, z = "Orange", "Banana", "Cherry"
2 print(x)
3 print(y)
4 print(z)
```

One Value to Multiple Variables:

And you can assign the *same* value to multiple variables in one line:

```
1 x = y = z = "Orange"
2 print(x)
3 print(y)
4 print(z)
```

### 1.5.5 Unpack a Collection

If you have a collection of values in a `list`, `tuple` etc. Python allows you to extract the values into variables. This is called *unpacking*.

Example (unpack a list):

```
1 fruits = ["apple", "banana", "cherry"]
2 x, y, z = fruits
3 print(x)
4 print(y)
5 print(z)
```

### 1.5.6 Output Variables

- The `print()` function is often used to output variables.
- In the `print()` function, you output multiple variables, separated by a comma.
  - You can also use the `+` operator to output multiple variables.
    - For strings, add a space at the end otherwise there will be no spaces between words.
    - For numbers, the `+` character works as a mathematical operator.
    - In the `print` function, when you try to combine a string and a number with the `+` operator, Python will give you an error.
- The best way to output multiple variables in the `print()` function is to separate them with commas, which even support different data types.

```
1 # Output multiple variables using a comma
2 x = "Python"
3 y = "is"
4 z = "awesome"
5 print(x, y, z)
6
7 # Output multiple variables using +
```

```

8  x = "Python"
9  y = "is"
10 z = "awesome"
11 print(x + y + z)
12
13 # Using the + character as a mathematical operator
14 x = 5
15 y = 10
16 print(x + y)
17
18 # Combining strings and numbers with the + operator
19 x = 5
20 y = "John"
21 print(x + y)

```

### 1.5.7 Global Variables

- Variables that are created outside of a function (as in all the previous examples) are known as **global variables**.
- Global variables can be used by everyone, both inside of functions and outside.

Example:

Create a variable outside of a function, and use it inside the function.

```

1 x = "awesome"
2
3 def myfunc():
4     print("Python is" + x)
5
6 myfunc()

```

If you create a variable with the same name inside a function, this variable will be **local**, and can only be used inside the function. The global variable with the same name will remain as it was, global and with the original value.

Example:

Create a variable inside a function, with the same name as the global variable.

```

1 x = "awesome"
2
3 def myfunc():
4     x = "fantastic"
5     print("Python is " + x)
6
7 myfunc()
8
9 print("Python is " + x)

```

The `global` keyword

- Normally, when you create a variable inside a function, that variable is local, and can only be used inside that function.
- To create a global variable inside a function, you can use the `global` keyword.

- If you use the `global` keyword, the variable belongs to the global scope.

Example:

```

1 def myfunc():
2     global x
3     x = "fantastic"
4
5 myfunc()
6
7 print("Python is " + x)

```

To change the value of a global variable inside a function, refer to the variable by using the `global` keyword. For example:

```

1 x = "awesome"
2
3 def myfunc():
4     global x
5     x = "fantastic"
6
7 myfunc()
8
9 print("Python is " + x)

```

### 1.5.8 Challenge

Inside the editor, complete the following steps:

1. Create a variable `x` and assign it the value `5`.
2. Create a variable `y` and assign it the value “`John`”.
3. Use the `type()` function to print the type of `x`.

Solution:

```

1 x = 5
2 y = "John"
3 print(type(x))

```

## 1.6 Python Data Types

### 1.6.1 Built-in Data Types

- In programming, data type is an important concept.
- Variables can store data of different types, and different types can do different things.
- Python has the following data types built-in by default, in these categories:

Text Type	<code>str</code>
Numeric Types	<code>int, float, complex</code>
Sequence Types	<code>list, tuple, range</code>
Mapping Type	<code>dict</code>
Set Types	<code>set, frozenset</code>
Boolean Type	<code>bool</code>
Binary Types	<code>bytes, bytearray, memoryview</code>
None Type	<code>NoneType</code>

### 1.6.2 Getting the Data Type

You can get the data type of any object by using the `type()` function.

```

1 x = 5
2 print(type(x))

```

Example	Data Type
<code>x = "Hello World"</code>	<code>str</code>
<code>x = 20</code>	<code>int</code>
<code>x = 20.5</code>	<code>float</code>
<code>x = 1j</code>	<code>complex</code>
<code>x = ["apple", "banana", "cherry"]</code>	<code>list</code>
<code>x = ("apple", "banana", "cherry")</code>	<code>tuple</code>
<code>x = range(6)</code>	<code>range</code>
<code>x = {"name": "John", "age": 36}</code>	<code>dict</code>
<code>x = {"apple", "banana", "cherry"}</code>	<code>set</code>
<code>x = frozenset({"apple", "banana", "cherry"})</code>	<code>frozenset</code>
<code>x = True</code>	<code>bool</code>
<code>x = b"Hello"</code>	<code>bytes</code>
<code>x = bytearray(5)</code>	<code>bytearray</code>
<code>x = memoryview(bytes(5))</code>	<code>memoryview</code>
<code>x = None</code>	<code>NoneType</code>

### 1.6.3 Setting the Specific Data Type

If you want to specify the data type, you can use the following constructor functions:

Example	Data Type
<code>x = str("Hello World")</code>	<code>str</code>
<code>x = int(20)</code>	<code>int</code>
<code>x = float(20.5)</code>	<code>float</code>
<code>x = complex(1j)</code>	<code>complex</code>
<code>x = list(["apple", "banana", "cherry"])</code>	<code>list</code>
<code>x = tuple(("apple", "banana", "cherry"))</code>	<code>tuple</code>
<code>x = range(6)</code>	<code>range</code>
<code>x = dict(name="John" age=36)</code>	<code>dict</code>
<code>x = set({"apple", "banana", "cherry"})</code>	<code>set</code>
<code>x = frozenset({"apple", "banana", "cherry"})</code>	<code>frozenset</code>
<code>x = bool(5)</code>	<code>bool</code>
<code>x = bytes(5)</code>	<code>bytes</code>
<code>x = bytearray(5)</code>	<code>bytearray</code>
<code>x = memoryview(bytes(5))</code>	<code>memoryview</code>

#### 1.6.4 Challenge

Inside the editor, complete the following steps:

1. Create a variable `x` with the value **5**.
2. Create a variable `y` with the value **3.14**.
3. Create a variable `z` with the value “**Hello**”.
4. Print the data type of each variable using `type()`.

Solution:

```

1 x = 5
2 y = 3.14
3 z = "Hello"
4
5 print(type(x))
6 print(type(y))
7 print(type(z))

```

## 1.7 Python Numbers

- Int, or integer, is a whole number, positive or negative, without decimals, of unlimited length.
- Float, or “floating point number” is a number, positive or negative, containing one or more decimals.
- Float can also be scientific numbers with an “e” to indicate the power of 10.
- Complex numbers are written with “j” as the imaginary part.
- You can convert from one type to another with the `int()`, `float()` and `complex()` methods.

```

1 # The three numeric types in Python
2 x = 1 # int
3 y = 2.8 # float
4 y = 35e3
5 z = 1j # complex
6
7 # To verify the type of object
8 print(type(x))
9 print(type(y))
10 print(type(z))
11
12 # Convert from one type to another
13 a = float(x)
14 b = int(y)
15 c = complex(x)

```

You cannot convert complex numbers into another number type.

### 1.7.1 Random Number

Python does not have a `random()` function to make a random number, but Python has a built-in module called `random` that can be used to make random numbers.

```

1 import random
2 print(random.randrange(1,10)) # displays a random number from 1 to 9

```

### 1.7.2 Challenge

Inside the editor, complete the following steps:

1. Create a variable `x` with the value **5**.
2. Create a variable `y` with the value **3.14**.
3. Create a variable `z` with the value  **$2 + 3j$** .
4. Print the type of each variable using `type()`.

Solution:

```

1 x = 5
2 y = 3.14
3 z = 2 + 3j
4
5 print(type(x))
6 print(type(y))
7 print(type(z))

```

## 1.8 Python Casting

If you want to specify the data type of a variable, this can be done with casting. Python is an object-oriented language, and as such it uses classes to define data types, including its primitive types.

Casting in python is therefore done using constructor functions:

- `int()` - constructs an integer number from:
  - an integer literal
  - a float literal (by removing all decimals)
  - a string literal (providing the string represents a whole number)
- `float()` - constructs a float number from:
  - an integer literal
  - a float literal
  - a string literal (providing the string represents a float or an integer)
- `str()` - constructs a string from a wide variety of data types, including:
  - strings
  - integer literals
  - float literals

Example:

```

1 x = str(3) # x will be '3'
2 y = int (3) # y will be 3
3 z = float(3) # z will be 3.0
4
5 print(int(35.88)) # Output is 35
6 print(float(35)) # Output is 35.0
7 print(str(35.82)) # Output is 35.82

```

### 1.8.1 Challenge

Inside the editor, complete the following steps:

1. Create a variable `x` with integer value `1`.
2. Convert `x` to a float and store it in `a`.
3. Convert `x` to a string and store it in `b`.
4. Print `a` and `b`.

Solution:

```

1 # Create an integer
2 x = 1
3
4 # Convert to float
5 a = float(x)
6
7 # Convert to string
8 b = str(x)
9
10 # Print values
11 print(a)
12 print(b)

```

## 1.9 Python Strings

- Strings in python are surrounded by either single quotation marks, or double quotation marks.
- `'hello'` is the same as `"hello"`.
- You can display a string literal with the `print()` function.

Example:

```
1 print("Hello")
2 print('Hello')
```

### 1.9.1 Quotes Inside Quotes

You can use quotes inside a string, as long as they don't match the quotes surrounding the string.

Example:

```
1 print("It's alright")
2 print("He is called 'Johnny'")
3 print('He is called "Johnny"')
```

### 1.9.2 Assign String to a Variable

Assigning a string to a variable is done with the variable name followed by an equal sign and the string.

Example:

```
1 a = "Hello"
2 print(a)
```

### 1.9.3 Multiline Strings

You can assign a multiline string to a variable by using three single/double quotes.

Example:

```
1 a = """Hello my name is Nicole,
2 I am 18 years old,
3 My birthday is June 2nd,
4 And I am a Gemini."""
5 print(a)
```

Note: in the result, the line breaks are inserted at the same position as in the code.

#### 1.9.4 Strings are Arrays

- Like many other popular programming languages, strings in Python are arrays of unicode characters.
- However, Python does not have a character data type, a single character is simply a string with a length of 1.
- Square brackets can be used to access elements of the string.

Example:

```
1 # Get the character at position 1 (remember that the first character has the
  ↪ position 0)
2 a = "Hello, World!"
3 print(a[1]) # Output would be e
```

#### 1.9.5 Looping Through a String

Since strings are arrays, we can loop through the characters in a string, with a `for` loop.

Example:

```
1 # Loop through the letters in the word "banana"
2 for x in "banana":
3     print(x)
```

#### 1.9.6 String Length

To get the length of a string, use the `len()` function.

Example:

```
1 a = "Hello, world!"
2 print(len(a)) # The len() function returns the length of a string - in this
  ↪ case, 13
```

#### 1.9.7 Check String

- To check if a certain phrase or character is present in a string, we can use the keyword `in`.
- To check if a certain phrase or character is NOT present in a string, we can use the keyword `not in`.

Example:

```
1 txt = "The best things in life are free!"
2 if "free" in txt:
3     print("Yes 'free' is present.")
4 if "expensive" not in txt:
5     print("No, 'expensive' is NOT present.")
```

### 1.9.8 Slicing Strings

- You can return a range of characters by using the slice syntax.
- Specify the start index and the end index, separated by a colon, to return a part of the string.

Example:

```
1 # Get the characters from position 2 to position 5 (not included)
2 b = "Hello, World!"
3 print(b[2:5]) # Output is llo
```

**REMEMBER:** The first character has index 0.

Slice From Start:

By leaving out the start index, the range will start at the first character.

Example:

```
1 # Get the characters from the start to position 5 (not included)
2 b = "Hello, World!"
3 print(b[:5]) # Output is Hello
```

Slice To the End:

By leaving out the end index, the range will go to the end.

Example:

```
1 # Get the characters from position 2 all the way to the end
2 b = "Hello, World!"
3 print(b[2:]) # Output is llo, World!
```

Negative Indexing:

Use negative indexes to start the slice from the end of the string.

Example:

```
1 b = "Hello, World!"
2 print(b[-5:-2]) # Output is orl (remember does not include -2)
```

### 1.9.9 String Concatenation

- To concatenate, or combine, two strings you can use the + operator.
- To add a space between them, add a " "

Example:

```
1 a = "Hello"
2 b = "World"
3 c = a + b
4 d = a + " " + b
5 print(c) # Output is HelloWorld
6 print(d) # Output is Hello World
```

### 1.9.10 String Format

As we learnt in the Python Variables chapter, we **cannot** combine strings and numbers like this:

Example:

```
1 age = 36
2 txt = "My name is John, I am " + age
3 print(txt)
```

But we can combine strings and numbers by using *f-strings* or the `format()` method.

F-Strings:

- F-strings were introduced in Python 3.6, and is now the preferred way of formatting strings.
- To specify a string as an f-string, simply put a `f` in front of the string literal, and add curly brackets `{}` as placeholders for variables and other operations.

Example:

```
1 age = 36
2 txt = f"My name is John, I am {age}"
3 print(txt)
```

Placeholders and Modifiers:

- A placeholder can contain variables, operations, functions, and modifiers to format the value.
- A placeholder can include a *modifier* to format the value.
- A modifier is included by adding a `:` followed by a legal formatting type, like `.2f` which means fixed point number with 2 decimals.
- A placeholder can contain Python code, like math operations.

Examples:

```
1 # Add a placeholder for the price variable
2 price = 59
3 txt = f"The price is {price} dollars"
4 print(txt) # Output is "The price is 59 dollars"
5
6 # Display the price with 2 decimals
7 txt = f"The price is {price:.2f} dollars"
8 print(txt) # Output is "The price is 59.00 dollars"
9
10 # Perform a math operation in the placeholder
11 txt = f"The price is {2 * price} dollars"
12 print(txt) # Output is "The price is 118 dollars"
```

### 1.9.11 Escape Characters

- To insert characters that are illegal in a string, use an escape character.
- An escape character is a \ followed by the character you want to insert.
- An example of an illegal character is a double quote inside a string that is surrounded by double quotes:

```
1 txt = "We are the so-called "Vikings" from the North."
```

To fix this problem, use the escape character \\:

```
1 txt = "We are the so-called \"Vikings\" from the North."
```

Other escape characters used in Python:

Code	Result
\'	Single Quote
\\	Backslash
\n	New Line
\r	Carriage Return
\t	Tab
\b	Backspace
\f	Form Feed
\ooo	Octal Value
\xhh	Hex Value

Examples:

```
1 print('It\'s alright') # Output is It's alright
2 print("This will insert one \\ (backslash).") # Output is This will insert one
   ↪ \ backslash
3 print("Hello\nWorld!") # Output is Hello (new line) World!
4 print("Hello\rWorld!") # Output is World!
5 print("Hello\tWorld!") # Output is Hello    World!
6
7
8 # This example erases one character (backspace)
9 print("Hello \bWorld!") # Output is HelloWorld!
10
11 print("Hello\fWorld!") # Output would be Hello (top of next page) World!
12
13 # A backslash followed by three integers will result in an octal value
14 print("\110\145\154\154\157") # Output is Hello
15
16 # A backslash followed by an 'x' and a hex number represents a hex value
17 print("\x48\x65\x6C\x6C\x6F") # Output is Hello
```

### 1.9.12 String Methods

Python has a set of built-in methods that you can use on strings.

**Note:** All string methods return new values. They do not change the original string.

<b>Method</b>	<b>Description</b>
<code>capitalize()</code>	Converts the first character to upper case
<code>casefold()</code>	Converts string into lower case
<code>center()</code>	Returns a centered string
<code>count()</code>	Returns the number of times a specific value occurs in a string
<code>encode()</code>	Returns an encoded version of the string
<code>endswith()</code>	Returns true if the string ends with the specified value
<code>expandtabs()</code>	Sets the tab size of the string
<code>find()</code>	Searches the string for a specified value and returns the position of where it was found
<code>format()</code>	Formats specified values in a string
<code>format_map()</code>	Formats specified values in a string
<code>index()</code>	Searches the string for a specified value and returns the position of where it was found
<code>isalnum()</code>	Returns True if all characters in the string are alphanumeric
<code>isalpha()</code>	Returns True if all characters in the string are in the alphabet
<code>isascii()</code>	Returns True if all characters in the string are ascii characters
<code>isdecimal()</code>	Returns True if all characters in the string are decimals
<code>isdigit()</code>	Returns True if all characters in the string are digits
<code>isidentifier()</code>	Returns True if the string is an identifier
<code>islower()</code>	Returns True if all characters in the string are lower case
<code>isnumeric()</code>	Returns True if all characters in the string are numeric
<code>isprintable()</code>	Returns True if all characters in the string are printable
<code>isspace()</code>	Returns True if all characters in the string are whitespaces
<code>istitle()</code>	Returns True if the string follows he rules of a title
<code>isupper()</code>	Returns True if all characters in the string are upper case
<code>join()</code>	Joins the elements of an iterable to the end of the string
<code>ljust()</code>	Returns a left justified version of the string
<code>lower()</code>	Converts a string into lower case
<code>lstrip()</code>	Returns a left trim version of the string
<code>maketrans()</code>	Returns a translation table to be used in translations
<code>partition()</code>	Returns a tuple where the string is parted into three parts
<code>replace()</code>	Returns a string where a specified value is replaced with a specified value
<code>rfind()</code>	Searches the string for a specified value and returns the last position of where it was found
<code>rindex()</code>	Searches the string for a specified value and returns the last position of where it was found
<code>rjust()</code>	Returns a right justified version of the string
<code>rpartition()</code>	Returns a tuple where the string is parted into three parts
<code>rsplit()</code>	Splits the string at the specified separator, and returns a list
<code>rstrip()</code>	Returns a right trim version of the string
<code>split()</code>	Splits the string at the specified separator, and returns a list
<code>splitlines()</code>	Splits the string at line breaks and returns a list
<code>startswith()</code>	Returns True if the string starts with the specified value
<code>strip()</code>	Returns a trimmed version of the string
<code>swapcase()</code>	Swaps cases,lower case becomes upper case and vice versa
<code>title()</code>	Converts the first character of each word to upper case
<code>translate()</code>	Returns a translated string
<code>upper()</code>	Converts a string into upper case

Examples:

```
1 a = "Hello, World!"
2 print(a.upper()) # Output is HELLO, WORLD!
3 print(a.lower()) # Output is hello, world!
4 print(a.replace("H", "J")) # Output is Jello, World!
5 print(a.split(",")) # Output is ['Hello', 'World!']
6
7 b = " Hello, World! "
8 print(b.strip()) #Output is "Hello, World!"
```

### 1.9.13 Challenge

Inside the editor, complete the following steps:

1. Create a variable `txt` with the value “Hello, World!”.
2. Print the characters from index **2** to **5** (slicing).
3. Print `txt` converted to **upper case**.
4. Create a variable `name` with the value “Python”.
5. Use a **f-string** to print “I love Python” using the `name` variable.

Solution:

```
1 # Create the variable
2 txt = "Hello, World!"
3
4 # Print characters from index 2 to 5
5 print(txt[2:5])
6
7 # Print in upper case
8 print(txt.upper())
9
10 # Create the name variable
11 name = "Python"
12
13 # Print using an f-string
14 print(f"I love {name}")
```

## 1.10 Python Booleans

- Booleans represents one of two values: `True` or `False`.
- In programming you often need to know if an expression is `True` or `False`.
- You can evaluate any expression in Python, and get one of two answers, `True` or `False`.
- When you compare two values, the expression is evaluated and Python returns the Boolean answer:

```
1 print(10 > 9) # Returns True
2 print (10 == 9) # Returns False
3 print (10 < 9) # Returns False
```

When you run a condition in an if statement, Python returns `True` or `False` :

```
1 # Print a message based on whether the condition is True or False
2 a = 200
3 b = 33
4
5 if b > a:
6     print("b is greater than a")
7 else:
8     print("b is not greater than a")
```

### 1.10.1 Evaluate Values and Variables

The `bool()` function allows you to evaluate any value, and give you `True` or `False` in return:

```
1 x = "Hello"
2 y = 15
3
4 print(bool(x)) # Returns True
5 print(bool(y)) # Returns True
```

### 1.10.2 Most Values are True

Almost any value is evaluated to `True` if it has some content:

- Any string is `True`, except empty strings.
- Any number is `True` , except 0 .
- Any list, tuple, set, and dictionary are `True`, except empty ones.

### 1.10.3 Some Values are False

There are not many values that evaluate to `False`, except empty values, such as:

- ()
- []
- {}
- The number 0
- The value `False`
- The value `None`

Examples:

```

1 # Return True
2 bool("abc")
3 bool(123)
4 bool(["apple", "banana", "cherry"])
5
6 # Return False
7 bool(False)
8 bool(None)
9 bool(0)
10 bool("")
11 bool(())

```

An object that is made from a class with a `__len__` function that returns 0 or `False` evaluates to False. For example:

```

1 class myClass():
2     def __len__(self):
3         return 0
4
5 myobj = myClass()
6 print(bool(myobj)) # Returns False

```

#### 1.10.4 Functions can Return a Boolean

You can create functions that return Boolean values and execute code based on the Boolean answer of the function.

Example:

```

1 def myFunction():
2     return True
3 if myFunction():
4     print("YES!")
5 else:
6     print("NO!")

```

Python also has many built-in functions that return a Boolean value, like the `isinstance()` function, which can be used to determine if an object is of a certain data type.

Example:

```

1 x = 200
2 print(isinstance(x, int)) # Returns True

```

#### 1.10.5 Challenge

Inside the editor, complete the following steps:

1. Print the result of `10 > 9`.
2. Print the result of `10 == 9`.
3. Print the result of `bool("Hello")`.

4. Print the result of `bool(0)`.

Solution:

```
1 print(10 > 9)
2 print(10 == 9)
3 print(bool("Hello"))
4 print(bool(0))
```

## 1.11 Python Operators

Operators are used to perform operations on variables and values.

- The `+` operator can be used to add two values.
- The `+` operator can also be used to add together a variable and two values, or two variables.

Examples:

```
1 sum1 = 100 + 50 # 150 (100 + 50)
2 sum2 = sum1 + 250 # 400 (150 + 250)
3 sum3 = sum1 + sum2 # 550 (150 + 400)
```

Python divides the operators in the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators
- Membership operators
- Bitwise operators

### 1.11.1 Arithmetic Operators

Arithmetic operators are used with **numeric values** to perform common mathematical operations:

Operator	Name	Example
<code>+</code>	Addition	<code>x + y</code>
<code>-</code>	Subtraction	<code>x - y</code>
<code>*</code>	Multiplication	<code>x * y</code>
<code>/</code>	Division	<code>x / y</code>
<code>%</code>	Modulus	<code>x % y</code>
<code>**</code>	Exponentiation	<code>x ** y</code>
<code>//</code>	Floor division	<code>x // y</code>

Example:

```

1 x = 15
2 y = 4
3
4 print(x + y) # 19
5 print(x - y) # 11
6 print(x * y) # 60
7 print(x / y) # 3.75 (returns a float)
8 print(x % y) # 3 (remainder)
9 print(x ** y) # 50625
10 print(x // y) # 3 (returns an integer, rounds DOWN to the nearest integer)

```

### 1.11.2 Assignment Operators

Assignment operators are used to assign values to variables:

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
=	x  = 3	x = x   3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3
:=	print(x := 3)	x = 3 print(x)

The Walrus Operator:

- Python 3.8 introduced the `:=` operator, known as the "walrus operator".
- It assigns values to variables as part of a larger expression.

Example:

```

1 numbers = [1, 2, 3, 4, 5]
2 if (count := len(numbers)) > 3:
3     print(f"List has {count} elements") # Output is "List has 5 elements"

```

### 1.11.3 Comparison Operators

Comparison operators are used to compare two values:

Operator	Name	Example
<code>==</code>	Equal	<code>x == y</code>
<code>!=</code>	Not equal	<code>x != y</code>
<code>&gt;</code>	Greater than	<code>x &gt; y</code>
<code>&lt;</code>	Less than	<code>x &lt; y</code>
<code>&gt;=</code>	Greater than or equal to	<code>x &gt;= y</code>
<code>&lt;=</code>	Less than or equal to	<code>x &lt;= y</code>

Comparison operators return `True` or `False` based on the comparison. // Example:

```

1 x = 5
2 y = 3
3 print(x == y) # Returns False
4 print(x != y) # Returns True
5 print(x > y) # Returns True
6 print(x < y) # Returns False
7 print(x >= y) # Returns True
8 print(x <= y) # Returns False

```

#### Chaining Comparison Operators:

Python allows you to chain comparison operators. For example:

```

1 x = 5
2 print(1 < x < 10) # Returns True
3 print(1 < x and x < 10) # Returns True

```

#### 1.11.4 Logical Operators

Logical operators are used to combine conditional statements:

Operator	Description	Example
<code>and</code>	Returns True if both statements are true	<code>x &lt; 5 and x &lt; 10</code>
<code>or</code>	Returns True if one of the statements is true	<code>x &lt; 5 or x &lt; 4</code>
<code>not</code>	Reverse the result, returns False if the result is true	<code>not(x &lt; 5 and x &lt; 10)</code>

Examples:

```

1 x = 5
2
3 print(x > 0 and x < 10) # Returns True
4 print(x < 5 or x > 10) # Returns False
5 print(not(x > 3 and x < 10)) # Returns False

```

#### 1.11.5 Identity Operators

Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location:

Operator	Description	Example
<code>is</code>	Returns True if both variables are the same object	<code>x is y</code>
<code>is not</code>	Returns True if both variables are NOT the same object	<code>x is not y</code>

Difference between `is` and `==`:

- `is` checks if both variables point to the same object in memory
- `==` checks if the values of both variables are equal

Examples:

```
1 x = ["apple", "banana"]
2 y = ["apple", "banana"]
3 z = x
4
5 # The is operator returns True if both variables point to the same object
6 print(x is z) # Returns True
7 print(x is y) # Returns False
8 print(x == y) # Returns True
9
10 # The is not operator returns True if both variables do not point to the same
11 # object
11 print(x is not y) # Returns True
```

### 1.11.6 Membership Operators

Membership operators are used to test if a sequence is presented in an object:

Operator	Description	Example
<code>in</code>	Returns True if a sequence with the specified value is present in the object	<code>x in y</code>
<code>not in</code>	Returns True if a sequence with the specified value is NOT present in the object	<code>x not in y</code>

Examples:

```
1 fruits = ["apple", "banana", "cherry"]
2 print("banana" in fruits) # Returns True
3 print("pineapple" not in fruits) # Returns True
```

Membership in Strings:

The membership operators also work with strings. For example:

```
1 txt = "Hello World"
2 print("H" in txt) # Returns True
3 print("hello" in txt) # Returns False
4 print("z" not in txt) # Returns True
```

**REMEMBER:** Membership operators are case-sensitive when used with strings.

### 1.11.7 Bitwise Operators

Bitwise operators are used to compare (binary) numbers: