Python String: Working With Text

October 3, 2023

So we've seen numbers, but what about text? This page is about the Python string, the go-to Python data type for storing and using text in Python. So, in Python, a piece of text is called a string, and you can perform all kinds of operations on a string. But let's start with the basics first!

Table of Contents [hide]

Beginners Python Course (2023)

Modules, Packages, And Virtual Environments (2023)

What is a Python string?

The following is a formal definition a Python string:

String
A string in Python is a sequence of characters

In even simpler terms, a string is a piece of text. Strings are not just a Python thing. It's a well-known term in computer science and also means the same thing in most other languages. Now that we know a string, we'll look at how to create one.

How to create a Python string

A Python string needs quotes around it for it to be recognized as such, like this:

```
>>> 'Hello, World'
'Hello, World'
```

Because of the quotes, Python understands this is a sequence of characters and not a command, number, or variable.

And just like with numbers, some of the operators we learned before work on Python strings, too. Try it with the following expressions:

```
>>> 'a' + 'b'
'ab'
>>> 'ab' * 4
'abababab'
>>> 'a' - 'b'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

This is what happens in the code above:

- The plus operator glues two Python strings together.
- The multiplication operator repeats our Python string the given number of times.
- The minus operator doesn't work on a Python string and produces an error. If you want to remove parts of a string, there are other methods that you'll learn about later on.

Single or double quotes?

We've used single quotes, but Python accepts double quotes around a string as well:

```
>>> "a" + "b"
'ab'
```

Note that these are not two single quotes next to each other. The character is often found next to the enter key on your keyboard (US keyboard layout) or on the 2 key (UK layout). It might differ for other countries. You must press shift with this key to get a double quote.

As you can see from its answer, Python itself seems to prefer single quotes. It looks clearer, and Python tries to be as clear and well-readable as possible. So why does it support both? It's because it allows you to use strings that contain a quote.

In the first example below, we use double quotes. Hence, there's no problem with the single quote in the word *it's*. However, in the second example, we try to use single quotes. Python sees the quote in the word *it's* and thinks this is the end of the string! The following letter, "s", causes a syntax error. A syntax error is a character or string incorrectly placed in a command or instruction that causes a failure in execution.

In other words, Python doesn't understand the s at that spot because it expects the string to be ended already and fails with an error:

```
>>> mystring = "It's a string, with a single quote!"
>>> mystring = 'It's a string, with a single quote!'
  File "<stdin>", line 1
    mystring = 'It's a string, with a single quote!'
                    ^
SyntaxError: invalid syntax
```

Python points out the exact location of where it encountered the error with the ^ symbol. Python errors tend to be very helpful, so look closely at them, and you'll often be able to pinpoint what's going wrong.

In case you noticed, the syntax highlighter on this website even gets confused because of the invalid syntax!

Escaping

There's another way around this problem, called escaping. You can escape a special character, like a quote, with a backward slash:

```
>>> mystring = 'It\'s an escaped quote!'
>>> mystring
"It's an escaped quote!"
```

You can also escape double quotes inside a double-quoted string:

```
>>> mystring = "I'm a so-called \"script kiddie\""
>>> mystring
'I\'m a so-called "script kiddie"'
```

Here, again, you see Python's preference for single quotes strings. Even though we used double quotes, Python echoes the string back to us using single quotes. It's still the same string, though; it's just represented differently. Once you start printing strings to the screen, you'll see the evidence of this.

So which one should you use? It's simple: always opt for the option where you need the least amount of escapes because these escapes make your Python strings less readable.

Multiline strings

Python also has syntax for creating multiline strings using triple quotes. By this, I mean three double quotes or three single quotes; both work, but I'll demonstrate with double quotes:

```
>>> """This is line 1,
... this is line 2,
... this is line 3."""
'This is line 1,\nthis is line 2,\nthis is line 3.'
```

As you can see, Python echos the string back as a regular, single-line string. In this string, you might have noticed the `\n` characters: this is how Python and many other programming languages escape special characters such as newlines. The following table lists a couple of the most common escape sequences you will encounter:

Escaped sequence	What is does
\n	A newline (Newlines are generated with your return key). Advances to the next
\r	Carriage return: takes you back to the start of the line, without advancing to the next line
\t	A tab character
\\	The slash character itself: because it is used as the start of escape sequences, we need to escape this character too. Python is quite forgiving if you forget to escape it, though.

Commonly encountered special characters in their escaped form

Interesting fact: Unix-based operating systems like Linux use `\n` for a new line, the carriage return is included automatically, while Windows uses `\r\n`. This has been and will be the cause of many bugs. So if you're on Windows, you will see a lot of `\r\n`.

Using triple quotes for escaping

The nice thing about triple quotes is that you can use both single and double quotes within them. So you can use triple quotes to cleanly create strings that contain both single and double quotes without resorting to escaping:

```
>>> line = """He said: "Hello, I've got a question" from the audience"""
```

String operations

Strings come with several handy, built-in operations you can execute. I'll show you only a couple here since I don't want to divert your attention from the tutorial too much.

In the REPL, you can sometimes use auto-completion. Whether it works or not depends on which installer you used to install Python and which OS you are on. Your best bet is to try!

In the next code fragment, we create a string, `mystring` and on the next line, we type its name, followed by hitting the **TAB** key twice:

```
>>> mystring = "Hello world"
>>> mystring.
mystring.capitalize(    mystring.find(          mystring.isdecimal(     mystring.istitle(       mystring.partition(
mystring.casefold(      mystring.format(        mystring.isdigit(       mystring.isupper(       mystring.replace(
mystring.center(        mystring.format_map(    mystring.isidentifier(  mystring.join(          mystring.rfind(
mystring.count(         mystring.index(         mystring.islower(       mystring.ljust(         mystring.rindex(
mystring.encode(        mystring.isalnum(       mystring.isnumeric(     mystring.lower(         mystring.rjust(
mystring.endswith(      mystring.isalpha(       mystring.isprintable(   mystring.lstrip(        mystring.rpartition(
mystring.expandtabs(    mystring.isascii(       mystring.isspace(       mystring.maketrans(     mystring.rsplit(
```

If all goes well, you should get a big list of operations that can be performed on a string. If it doesn't work, you can review a list of these operations (with an explanation) in the official Python manual. You can try some of these yourself:

```
>>> mystring.lower()
'hello world'
>>> mystring.upper()
'HELLO WORLD'
```

An explanation of each of these operations can be found in the official Python documentation, but we'll also cover a few here.

Getting the string length

A common operation is to get the string length. Unlike the operations above, this can be done with Python's `len()` function like this:

```
>>> len("I wonder how long this string will be...")
40
>>> len(mystring)
11
```

In fact, the `len()` function can be used on many objects in Python, as you'll learn later on. If functions are new to you, you're in luck because our next page will explain exactly what a function in Python is and how you can create one yourself.

Split a string

Another common operation is splitting a string. For this, we can use one of the built-in operations, conveniently called split. Let's start simple by splitting up two words on the space character between them:

```
'Hello world'.split(' ')
['Hello', 'world']
```

The split operation takes one argument: the sequence of characters to split on. The output is a Python list containing all the separate words.

Split on whitespace

A common use case is to split on whitespace. The problem is that whitespace can be a lot of things. Three common ones that you probably know already are:

- space characters
- tabs
- newlines

But there are many more, and to make it even more complicated, whitespace doesn't mean just one of these characters but can also be a whole sequence of them. E.g., three consecutive spaces and a tab character form one piece of whitespace.

Exactly because this is such a common operation among programmers and because it's hard to do it perfectly, Python has a convenient shortcut for it. Calling the split operation without any arguments splits a string on whitespace, as can be seen below:

```
>>> 'Hello \t\n there,\t\t\t stranger.'.split()
['Hello', 'there,', 'stranger.']
```

As you can see, no matter what whitespace character and how many, Python can still split this string for us into separate words.

Replace parts of a string

Let's look at one more built-in operation on strings: the replace function. It's used to replace one or more characters or sequences of characters:

```
>>> 'Hello world'.replace('H', 'h')
'hello world'
>>> 'Hello world'.replace('l', '_')
'He__o wor_d
>>> 'Hello world'.replace('world', 'readers')
'Hello readers'
```

Reversing a string (slicing)

A typical assignment is to reverse a Python string. There's no reverse operation, as you might have noticed when studying the list of operations like `lower() and upper() that comes` with a string. This is not exactly beginner stuff, so feel free to skip this for now if you're going through the tutorial sequentially.

We can treat a string as a list to reverse it efficiently. Lists are covered later on in this tutorial (see for-loop). In fact, you could see a string as a list of characters. And more importantly, you can treat it as such. List index operations like `mystring[2]` work just like they work on lists:

```
>>> mystring = 'Hello world'
>>> mystring[2]
'l'
>>> mystring[0]
'H'
```

Note that in Python, like in all computer languages, we start counting from 0.

What also works the same as in lists is the slicing operator. Details of slicing can be found on the Python list page and won't be repeated here. If you're coming from other languages, you might compare it to an operation like `substring()` in Java, which allows you to retrieve specific parts of a string.

Slicing in Python works with the slicing operator, which looks like this: `mystring[start:stop:step_size]`. The key feature we use from slicing is the step size. We traverse the string from the end to the beginning by giving the slicing operator a negative step size of -1. By leaving the start and end positions empty, Python assumes we want to slice the entire string.

So we can use slicing to reverse a Python string as follows:

```
>>> mystring = 'Hello world'
>>> mystring[::-1]
'dlrow olleH'
```

Slicing is explained in more detail in my article on lists.

Chaining calls

So far, we've used single operations. We can also chain these operations together, however:

```
>>> 'Hello world'.replace('world', 'student').upper()
'HELLO STUDENT'
```

What's happening here? The first operation (replace) results in a string. This string, like all strings, offers us the same operations again. So we can directly call the next operation (upper) on the result. It's a handy shortcut you'll often use and encounter in other code.

You could explicitly do this as well by first assigning the result to a variable:

```
>>> student_greeting = 'Hello world'.replace('world', 'student')
>>> student_greeting.upper()
'HELLO STUDENT'
```

Python string formatting with f-strings

A common pattern is the need to merge some text strings or use a variable or expression inside your string. There are several ways to do so, but the most modern way is to use f-strings, which is short for formatted strings.

Let's first look at an example before we dive into the details:

```
>>> my_age = 40
>>> f'My age is {my_age}'
My age is 40
```

The f-string looks like a regular string with the addition of an f prefix. This f tells Python to scan the string for curly braces. Inside these curly braces, we can put any Python expression we want. In the above case, we just included the variable my_age. F-strings provide an elegant way of including the results of expressions inside strings.

Here are a couple more examples you can try for yourself as well inside the REPL:

```
>>> f'3 + 4 = {3+4}'
'3 + 4 = 7'
>>> my_age = 40
>>> f'My age is, unfortunately, not {my_age-8}'
'My age is, unfortunately, not 32'
```

Quickly printing a variable

The f-string allows us to print the name and content of a variable quickly. This can be super useful when you're debugging your code. The following code crumb shows this in practice:

Learn more

I'm just touching on the basics here. If you're following the tutorial front to back, you can continue with the following topic since you know more than enough for now. If you'd like to find out more about f-strings, try the following resources:

- Official docs on f-strings
- The official guide has some more examples here: formatted string literals

Get certified with our courses

Learn Python properly through small, easy-to-digest lessons, progress tracking, quizzes to test your knowledge, and practice sessions. Each course will earn you a downloadable course certificate.

Beginners Python Course (2023)

Modules, Packages, And Virtual Environments (2023)

NumPy Course: The Hands-on Introduction To NumPy (2023)

‹ Previous: Python Variable

Next: Output Data To Your Screen With Python's print() Function