Chapter 4

Strings and String Methods

Many programmers, regardless of their specialty, deal with text on a daily basis. For example, web developers work with text that gets input from web forms. Data scientists process text to extract data and perform things like sentiment analysis, which can help identify and classify opinions in a body of text.

Collections of text in Python are called **strings**. Special functions called **string methods** are used to manipulate strings. There are string methods for changing a string from lowercase to uppercase, removing whitespace from the beginning or end of a string, or replacing parts of a string with different text, and many more.

In this chapter, you will learn how to:

- Manipulate strings with string methods
- Work with user input
- Deal with strings of numbers
- Format strings for printing

Let's get started!

Leave feedback on this section »

4.1 What is a String?

In Chapter 3, you created the string `"Hello, world"` and printed it in IDLE's interactive window using the `print()` function. In this section, you'll get a deeper look into what exactly a string is and the various ways you can create them in Python.

The String Data Type

Strings are one of the fundamental Python data types. The term **data type** refers to what kind of data a value represents. Strings are used to represent text.

Note

There are several other data types built-in to Python. For example, you'll learn about numerical data types in Chapter 5, and Boolean types in Chapter 8.

We say that strings are a **fundamental** data type because they can't be broken down into smaller values of a different type. Not all data types are fundamental. You'll learn about compound data types, also known as **data structures**, in Chapter 9.

The string data type has a special abbreviated name in Python: `str`. You can see this by using the `type()` function, which is used to determine the data type of a given value.

Type the following into IDLE's interactive window:

```
>>> type("Hello, world")
<class 'str'>
```

The output `<class 'str'>` indicates that the value `"Hello, world"` is an instance of the `str` data type. That is, `"Hello, world"` is a string.

Note

For now, you can think of the word “class” as a synonym for “data type,” although it actually refers to something more specific. You’ll see just what a class is in Chapter 10.

`type()` also works for values that have been assigned to a variable:

```
>>> phrase = "Hello, world"
>>> type(phrase)
<class 'str'>
```

Strings have three properties that you’ll explore in the coming sections:

1. Strings contain **characters**, which are individual letters or symbols.
2. Strings have a **length**, which is the number of characters contained in the string.
3. Characters in a string appear in a **sequence**, meaning each character has a numbered position in the string.

Let’s take a closer look at how strings are created.

String Literals

As you’ve already seen, you can create a string by surrounding some text with quotation marks:

```
string1 = 'Hello, world'
string2 = "1234"
```

Either single quotes (`string1`) or double quotes (`string2`) can be used to create a string, as long as both quotation marks are the same type.

Whenever you create a string by surrounding text with quotation marks, the string is called a **string literal**. The name indicates that the string is literally written out in your code. All of the strings you have seen thus far are string literals.

Note

Not every string is a string literal. For example, a string captured as user input isn't a string literal because it isn't explicitly written out in the program's code.

You'll learn how to work with user input in section 4 of this chapter.

The quotes surrounding a string are called **delimiters** because they tell Python where a string begins and where it ends. When one type of quotes is used as the delimiter, the other type of quote can be used inside of the string:

```
string3 = "We're #1!"
string4 = 'I said, "Put it over by the llama."'
```

After Python reads the first delimiter, all of the characters after it are considered a part of the string until a second matching delimiter is read. This is why you can use a single quote in a string delimited by double quotes and vice versa.

If you try to use double quotes inside of a string that is delimited by double quotes, you will get an error:

```
>>> text = "She said, "What time is it?""
  File "<stdin>", line 1
    text = "She said, "What time is it?""
                          ^
SyntaxError: invalid syntax
```

Python throws a `SyntaxError` because it thinks that the string ends after the second `"` and doesn't know how to interpret the rest of the line.

Note

A common pet peeve among programmers is the use of mixed quotes as delimiters. When you work on a project, it's a good idea to use only single quotes or only double quotes to delimit every string.

Keep in mind that there isn't really a right or wrong choice! The goal is to be consistent, because consistency helps make your code easier to read and understand.

Strings can contain any valid Unicode character. For example, the string `"We're #1!"` contains the pound sign (`#`) and `"1234"` contains numbers. `"×Pýŧħøŋ×"` is also a valid Python string!

Determine the Length of a String

The number of characters contained in a string, including spaces, is called the **length** of the string. For example, the string `"abc"` has a length of `3`, and the string `"Don't Panic"` has a length of `11`.

To determine a string's length, you use Python's built-in `len()` function. To see how it works, type the following into IDLE's interactive window:

```
>>> len("abc")
3
```

You can also use `len()` to get the length of a string that's assigned to a variable:

```
>>> letters = "abc"
>>> num_letters = len(letters)
>>> num_letters
3
```

First, the string `"abc"` is assigned to the variable `letters`. Then `len()` is used to get the length of `letters` and this value is assigned to the `num_letters` variable. Finally, the value of `num_letters`, which is `3`, is displayed.

Multiline Strings

The PEP 8 style guide recommends that each line of Python code contain no more than 79 characters—including spaces.

Note

PEP 8's 79-character line-length is recommended because, among other things, it makes it easier to read two files side-by-side. However, many Python programmers believe forcing each line to be at most 79 characters sometimes makes code harder to read.

In this book we will strictly follow PEP 8's recommended line-length. Just know that you will encounter lots of code in the real world with longer lines.

Whether you decide to follow PEP 8, or choose a larger number of characters for your line-length, you will sometimes need to create string literals with more characters than your chosen limit.

To deal with long strings, you can break the string up across multiple lines into a **multiline string**. For example, suppose you need to fit the following text into a string literal:

> "This planet has—or rather had—a problem, which was this: most of the people living on it were unhappy for pretty much of the time. Many solutions were suggested for this problem, but most of these were largely concerned with the movements of small green pieces of paper, which is odd because on the whole it wasn't the small green pieces of paper that were unhappy."
>
> — Douglas Adams, *The Hitchhiker's Guide to the Galaxy*

This paragraph contains far more than 79 characters, so any line of code containing the paragraph as a string literal violates PEP 8. So, what do you do?

There are a couple of ways to tackle this. One way is to break the string up across multiple lines and put a backslash (\) at the end of all but the last line. To be PEP 8 compliant, the total length of the line, including the backslash, must be 79 characters or less.

Here's how you could write the paragraph as a multiline string using the backslash method:

```
paragraph = "This planet has - or rather had - a problem, which was \
this: most of the people living on it were unhappy for pretty much \
of the time. Many solutions were suggested for this problem, but \
most of these were largely concerned with the movements of small \
green pieces of paper, which is odd because on the whole it wasn't \
the small green pieces of paper that were unhappy."
```

Notice that you don't have to close each line with a quotation mark. Normally, Python would get to the end of the first line and complain that you didn't close the string with a matching double quote. With a backslash at the end, however, you can keep writing the same string on the next line.

When you `print()` a multiline string that is broken up by backslashes, the output displayed on a single line:

```
>>> long_string = "This multiline string is \
displayed on one line"
>>> print(long_string)
This multiline string is displayed on one line
```

Multiline strings can also be created using triple quotes as delimiters (`"""` or `'''`). Here is how you might write a long paragraph using this approach:

```
paragraph = """This planet has - or rather had - a problem, which was
this: most of the people living on it were unhappy for pretty much
of the time. Many solutions were suggested for this problem, but
most of these were largely concerned with the movements of small
green pieces of paper, which is odd because on the whole it wasn't
the small green pieces of paper that were unhappy."""
```

Triple-quoted strings preserve whitespace. This means that running `print(paragraph)` displays the string on multiple lines just like it is in the string literal, including newlines. This may or may not be what you want, so you'll need to think about the desired output before you choose how to write a multiline string.

To see how whitespace is preserved in a triple-quoted string, type the following into IDLE's interactive window:

```
>>> print("""An example of a
...     string that spans across multiple lines
...         that also preserves whitespace.""")
An example of a
    string that spans across multiple lines
        that also preserves whitespace.
```

Notice how the second and third lines in the output are indented exactly the same way they are in the string literal.

Note

Triple-quoted strings have a special purpose in Python. They are used to document code. You'll often find them at the top of a `.py` with a description of the code's purpose. They are also used to document custom functions.

When used to document code, triple-quoted strings are called **docstrings**. You'll learn more about docstrings in Chapter 6.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources.

1. Print a string that uses double quotation marks inside the string.
2. Print a string that uses an apostrophe inside the string.
3. Print a string that spans multiple lines, with whitespace preserved.
4. Print a string that is coded on multiple lines but displays on a single line.

Leave feedback on this section »

4.2 Concatenation, Indexing, and Slicing

Now that you know what a string is and how to declare string literals in your code, let's explore some of the things you can do with strings.

In this section, you'll learn about three basic string operations:

1. Concatenation, which joins two strings together
2. Indexing, which gets a single character from a string
3. Slicing, which gets several characters from a string at once

Let's dive in!

String Concatenation

Two strings can be combined, or **concatenated**, using the + operator:

```
>>> string1 = "abra"
>>> string2 = "cadabra"
>>> magic_string = string1 + string2
>>> magic_string
```

```
'abracadabra'
```

In this example, string concatenation occurs on the third line. `string1` and `string2` are concatenated using `+` and the result is assigned to the variable `magic_string`. Notice that the two strings are joined without any whitespace between them.

You can use string concatenation to join two related strings, such as joining a first and last name into a full name:

```
>>> first_name = "Arthur"
>>> last_name = "Dent"
>>> full_name = first_name + " " + last_name
>>> full_name
'Arthur Dent'
```

Here string concatenation occurs twice on the same line. `first_name` is concatenated with `" "`, resulting in the string `"Arthur "`. Then this result is concatenated with `last_name` to produce the full name `"Arthur Dent"`.

String Indexing

Each character in a string has a numbered position called an **index**. You can access the character at the *Nth* position by putting the number *N* in between two square brackets (`[` and `]`) immediately after the string:

```
>>> flavor = "apple pie"
>>> flavor[1]
'p'
```

`flavor[1]` returns the character at position `1` in `"apple pie"`, which is `p`. Wait, isn't `a` the first character of `"apple pie"`?

In Python—and most other programming languages—counting always starts at zero. To get the character at the beginning of a string, you need to access the character at position `0`:

```
>>> flavor[0]
'a'
```

Note

Forgetting that counting starts with zero and trying to access the first character in a string with the index `1` results in an **off-by-one error**.

Off-by-one errors are a common source of frustration for both beginning and experienced programmers alike!

The following figure shows the index for each character of the string `"apple pie"`:

```
|  a  |  p  |  p  |  l  |  e  |     |  p  |  i  |  e  |
   0     1     2     3     4     5     6     7     8
```

If you try to access an index beyond the end of a string, Python raises an `IndexError`:

```
>>> flavor[9]
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    flavor[9]
IndexError: string index out of range
```

The largest index in a string is always one less than the string's length. Since `"apple pie"` has a length of nine, the largest index allowed is `8`.

Strings also support negative indices:

```
>>> flavor[-1]
'e'
```

The last character in a string has index `-1`, which for `"apple pie"` is the letter `e`. The second-to-last character `i` has index `-2`, and so on.

The following figure shows the negative index for each character in the string `"apple pie"`:

a	p	p	l	e		p	i	e
-9	-8	-7	-6	-5	-4	-3	-2	-1

Just like positive indices, Python raises an `IndexError` if you try to access a negative index less than the index of the first character in the string:

```
>>> flavor[-10]
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    flavor[-10]
IndexError: string index out of range
```

Negative indices may not seem useful at first, but sometimes they are a better choice than a positive index.

For example, suppose a string input by a user is assigned to the variable `user_input`. If you need to get the last character of the string, how do you know what index to use?

One way to get the last character of a string is to calculate the final index using `len()`:

```
final_index = len(user_input) - 1
last_character = user_input[final_index]
```

Getting the final character with the index `-1` takes less typing and doesn't require an intermediate step to calculate the final index:

```
last_character = user_input[-1]
```

String Slicing

Suppose you need the string containing just the first three letters of the string `"apple pie"`. You could access each character by index and concatenate them, like this:

```
>>> first_three_letters = flavor[0] + flavor[1] + flavor[2]
>>> first_three_letters
'app'
```

If you need more than just the first few letters of a string, getting each character individually and concatenating them together is clumsy and long-winded. Fortunately, Python provides a way to do this with much less typing.

You can extract a portion of a string, called a **substring**, by inserting a colon between two index numbers inside of square brackets, like this:

```
>>> flavor = "apple pie"
>>> flavor[0:3]
'app'
```

`flavor[0:3]` returns the first three characters of the string assigned to `flavor`, starting with the character with index `0` and going up to, but not including, the character with index `3`. The `[0:3]` part of `flavor[0:3]` is called a **slice**. In this case, it returns a slice of `"apple pie"`. Yum!

String slices can be confusing because the substring returned by the slice includes the character whose index is the first number, but doesn't include the character whose index is the second number.

To remember how slicing works, you can think of a string as a sequence of square slots. The left and right boundary of each slot is numbered from zero up to the length of the string, and each slot is filled with a character in the string.

Here's what this looks like for the string `"apple pie"`:

```
_________________________________________________________
|  a  |  p  |  p  |  l  |  e  |     |  p  |  i  |  e  |
```

0	1	2	3	4	5	6	7	8	9

The slice `[x:y]` returns the substring between the boundaries `x` and `y`. So, for `"apple pie"`, the slice `[0:3]` returns the string `"app"`, and the slice `[3:9]` returns the string `"le pie"`.

If you omit the first index in a slice, Python assumes you want to start at index `0`:

```
>>> flavor[:5]
'apple'
```

The slice `[:5]` is equivalent to the slice `[0:5]`, so `flavor[:5]` returns the first five characters in the string `"apple pie"`.

Similarly, if you omit the second index in the slice, Python assumes you want to return the substring that begins with the character whose index is the first number in the slice and ends with the last character in the string:

```
>>> flavor[5:]
' pie'
```

For `"apple pie"`, the slice `[5:]` is equivalent to the slice `[5:9]`. Since the character with index `5` is a space, `flavor[5:9]` returns the substring that starts with the space and ends with the last letter, which is `" pie"`.

If you omit both the first and second numbers in a slice, you get a string that starts with the character with index `0` and ends with the last character. In other words, omitting both numbers in a slice returns the entire string:

```
>>> flavor[:]
'apple pie'
```

It's important to note that, unlike string indexing, Python won't raise an `IndexError` when you try to slice between boundaries before or after the beginning and ending boundaries of a string:

```
>>> flavor[:14]
'apple pie'
>>> flavor[13:15]
''
```

In this example, the first line gets the slice from the beginning of the string up to but not including the fourteenth character. The string assigned to `flavor` has length nine, so you might expect Python to throw an error. Instead, any non-existent indices are ignored and the entire string `"apple pie"` is returned.

The second shows what happens when you try to get a slice where the entire range is out of bounds. `flavor[13:15]` attempts to get the thirteenth and fourteenth characters, which don't exist. Instead of raising an error, the **empty string** `""` is returned.

You can use negative numbers in slices. The rules for slices with negative numbers are exactly the same as slices with positive numbers. It helps to visualize the string as slots with the boundaries labeled by negative numbers:

```
 |   a   |   p   |   p   |   l   |   e   |      |   p   |   i   |   e   |
-9      -8      -7      -6      -5      -4     -3      -2      -1
```

Just like before, the slice `[x:y]` returns the substring between the boundaries `x` and `y`. For instance, the slice `[-9:-6]` returns the first three letters of the string `"apple pie"`:

```
>>> flavor[-9:-6]
'app'
```

Notice, however, that the right-most boundary does not have a negative index. The logical choice for that boundary would seem to be the number `0`, but that doesn't work:

```
>>> flavor[-9:0]
''
```

Instead of returning the entire string, `[-9:0]` returns the **empty string** `""`. This is because the second number in a slice must correspond to a boundary that comes after the boundary corresponding to the first number, but both `-9` and `0` correspond to the left-most boundary in the figure.

If you need to include the final character of a string in your slice, you can omit the second number:

```
>>> flavor[-9:]
'apple pie'
```

Strings Are Immutable

To wrap this section up, let's discuss an important property of string objects. Strings are **immutable**, which means that you can't change them once you've created them. For instance, see what happens when you try to assign a new letter to one particular character of a string:

```
>>> word = "goal"
>>> word[0] = "f"
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    word[0] = "f"
TypeError: 'str' object does not support item assignment
```

Python throws a `TypeError` and tells you that `str` objects don't support item assignment.

Note

The term `str` is Python's internal name for the string data type.

If you want to alter a string, you must create an entirely new string. To change the string `"goal"` to the string `"foal"`, you can use a string slice to concatenate the letter `"f"` with everything but the first letter of the word `"goal"`:

```
>>> word = "goal"
>>> word = "f" + word[1:]
>>> word
'foal'
```

First assign the string `"goal"` to the variable `word`. Then concatenate the slice `word[1:]`, which is the string `"oal"`, with the letter `"f"` to get the string `"foal"`. If you're getting a different result here, make sure you're including the `:` colon character as part of the string slice.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources.

1. Create a string and print its length using the `len()` function.
2. Create two strings, concatenate them, and print the resulting string.
3. Create two strings and use concatenation to add a space in-between them. Then print the result.
4. Print the string `"zing"` by using slice notation on the string `"bazinga"` to specify the correct range of characters.

Leave feedback on this section »

4.3 Manipulate Strings With Methods

Strings come bundled with special functions called **string methods** that can be used to work with and manipulate strings. There are numerous string methods available, but we'll focus on some of the most commonly used ones.

In this section, you will learn how to: