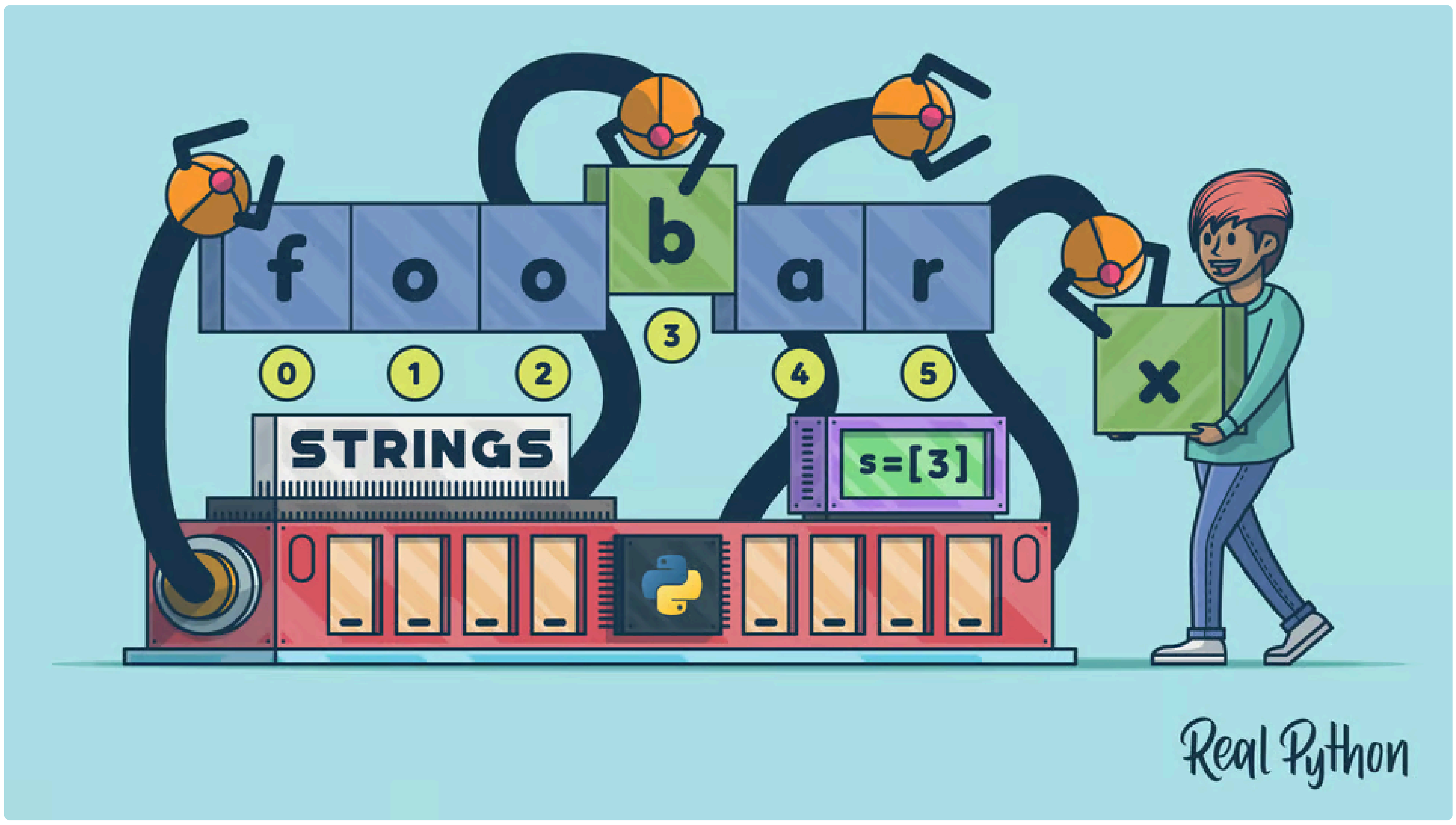


Strings and Character Data in Python

by Leodanis Pozo Ramos Dec 22, 2024 1h 2m 24 Comments basics python

Mark as Completed Share

Table of Contents

Help

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Strings and Character Data in Python**

Python strings are a sequence of characters used for handling textual data. You can create strings in Python using quotation marks or the `str()` function, which converts objects into strings. Strings in Python are immutable, meaning once you define a string, you can't change it.

By the end of this tutorial, you'll understand that:

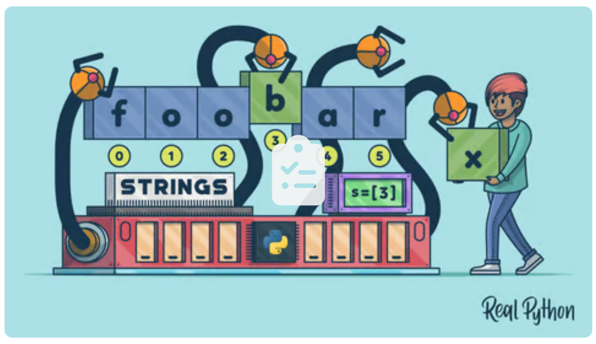
- The `str()` function converts objects to their **string representation**.
- **String interpolation** in Python allows you to insert values into `{}` placeholders in strings using f-strings or the `.format()` method.
- You **access string elements** in Python using **indexing** with square brackets.
- You can **slice a string** in Python by using the syntax `string[start:end]` to extract a substring.
- You **concatenate strings** in Python using the `+` operator or by using the `.join()` method.

You'll explore creating strings with string literals and functions, using operators and built-in functions with strings, indexing and slicing techniques, and methods for string interpolation and formatting. These skills will help you manipulate and format textual data in your Python programs effectively.

To get the most out of this tutorial, you should have a good understanding of core Python concepts, including variables, functions, and operators and expressions.

Get Your Code: Click here to download the free sample code that shows you how to work with strings and character data in Python.

Take the Quiz: Test your knowledge with our interactive "Python Strings and Character Data" quiz. You'll receive a score upon completion to help you track your learning progress:



Interactive Quiz

Python Strings and Character Data

This quiz will test your understanding of Python's string data type and your knowledge about manipulating textual data with string objects. You'll cover the basics of creating strings using literals and the str() function, applying string methods, using operators and built-in functions, and more!

Getting to Know Strings and Characters in Python

Python provides the built-in string (`str`) data type to handle textual data. Other programming languages, such as Java, have a character data type for single characters. Python doesn't have that. Single characters are strings of length one.

In practice, strings are immutable sequences of characters. This means you can't change a string once you define it. Any operation that modifies a string will create a new string instead of modifying the original one.

A string is also a sequence, which means that the characters in a string have a consecutive order. This feature allows you to access characters using integer indices that start with `0`. You'll learn more about these concepts in the section about indexing strings. For now, you'll learn about how to create strings in Python.

Creating Strings in Python

There are different ways to create strings in Python. The most common practice is to use string literals. Because strings are everywhere and have many use cases, you'll find a few different types of string literals. There are standard literals, raw literals, and formatted literals.

Additionally, you can use the built-in `str()` function to create new strings from other existing objects.

In the following sections, you'll learn about the multiple ways to create strings in Python and when to use each of them.

Standard String Literals

A standard string literal is just a piece of text or a sequence of characters that you enclose in quotes. To create single-line strings, you can use single (`''`) and double (`""`) quotes:

Python

```python
>>> 'A single-line string in single quotes'
'A single-line string in single quotes'

>>> "A single-line string in double quotes"
'A single-line string in double quotes'
```

In the first example, you use single quotes to delimit the string literal. In the second example, you use double quotes.

Note: Python's standard REPL displays string objects using single quotes even though you create them using double quotes.

You can define empty strings using quotes without placing characters between them:

Python

```python
>>> ""
''

>>> ''
''

>>> len("")
0
```

An empty string doesn't contain any characters, so when you use the built-in `len()` function with an empty string as an argument, you get `0` as a result.

To create multiline strings, you can use triple-quoted strings. In this case, you can use either single or double quotes:

Python

```python
>>> '''A triple-quoted string
... spanning across multiple
... lines using single quotes'''
'A triple-quoted string\nspanning across multiple\nlines using single quotes'

>>> """A triple-quoted string
... spanning across multiple
... lines using double quotes"""
'A triple-quoted string\nspanning across multiple\nlines using double quotes'
```

The primary use case for triple-quoted strings is to create multiline strings. You can also use them to define single-line strings, but this is a less common practice.

Probably the most common use case for triple-quoted strings is when you need to provide docstrings for your packages, modules, functions, classes, and methods.

If you want to include a quote character within the string, then you can delimit that string with another type of quote. For example, if a string contains a single quote character, then you can delimit it with double quotes, and vice versa:

Python

```
>>> "This string contains a single quote (') character"
"This string contains a single quote (') character"

>>> 'This string contains a double quote (") character'
'This string contains a double quote (") character'
```

In the first example, your string includes a single quote as part of the text. To delimit the literal, you use double quotes. In the second example, you do the opposite.

Escape Sequences in String Literals

Sometimes, you want Python to interpret a character or sequence of characters within a string differently. This may occur in one of two ways. You may want to:

1. Apply special meaning to characters
2. Suppress special character meaning

To achieve these goals, you can use a backslash (`\`) character combined with other characters. The combination of a backslash and a specific character is called an **escape sequence**. That's because the backslash causes the subsequent character to *escape* its usual meaning.

For example, you can escape a single quote character using the `\'` escape sequence in a string delimited by single quotes:

Python

```
>>> 'This string contains a single quote (\') character'
"This string contains a single quote (') character"
```

In this example, the backslash escapes the single quote character by suppressing its usual meaning, which is delimiting string literals. Now, Python knows that your intention isn't to terminate the string but to embed the single quote.

The following table shows sequences that escape the default meaning of characters in string literals:

Character	Usual Interpretation	Escape Sequence	Escaped Interpretation
`'`	Delimits a string literal	`\'`	Literal single quote (`'`) character
`"`	Delimits a string literal	`\"`	Literal double quote (`"`) character
`<newline>`	Terminates the input line	`\<newline>`	Newline is ignored
`\`	Introduces an escape sequence	`\\`	Literal backslash (`\`) character

You already have an idea of how the first two escape sequences work. Now, how does the newline escape sequence work? Usually, a newline character terminates a physical line of input. To add a newline character, you just press `Enter ↩` in the middle of a string. This will raise a `SyntaxError` exception:

Python

```
>>> "Hello
  File "<input>", line 1
    "Hello
    ^
SyntaxError: incomplete input
```

When you press `Enter ↩` after typing `"Hello`, you get a `SyntaxError`. If you need to break up a string over more than one line, then you can include a backslash before each newline or before pressing the `Enter ↩` key:

Python

```
>>> "Hello\
... , World\
... !"
'Hello, World!'
```

When you add a backslash before you press Enter ↩, Python ignores the newline and interprets the whole construct as a single physical line.

Sometimes, you need to include a literal backslash character in a string. If that backslash doesn't precede a character with a special meaning, then you can insert it right away:

Python

```
>>> "This string contains a backslash (\) character"
'This string contains a backslash (\\) character'
```

In this example, the character after the backslash doesn't match any known escape sequence, so Python inserts the actual backslash for you. Note how the resulting string automatically doubles the backslash. Even though this example works, the best practice is to always double the backslash when you need this character in a string.

However, you may have the need to include a backslash right before a character that makes up an escape sequence:

Python

```
>>> "In this string, the backslash should be at the end \"
  File "<input>", line 1
    "In this string, the backslash should be at the end \"
    ^
SyntaxError: incomplete input
```

Because the sequence `\"` matches a known escape sequence, your string fails with a `SyntaxError`. To avoid this issue, you can double the backslash:

Python

```
>>> "In this string, the backslash should be at the end \\"
'In this string, the backslash should be at the end \\'
```

In this update, you double the backslash to escape the character and prevent Python from raising an error.

Note: When you use the built-in `print()` function to print a string that includes an escaped backslash, then you won't see the double backslash in the output:

Python

```
>>> print("In this string, the backslash should be at the end \\")
In this string, the backslash should be at the end \
```

In this example, the output only displays one backslash, producing the desired effect.

Up to this point, you've learned how to suppress the meaning of a given character by escaping it. Suppose you need to create a string containing a tab character. Some text editors may allow you to insert a tab character directly into your code. However, this is considered a poor practice for several reasons:

- Computers can distinguish between tabs and a sequence of spaces, but human beings can't because these characters are visually indistinguishable.
- Some text editors automatically eliminate tabs by expanding them to an appropriate number of spaces.
- Some Python REPL environments won't insert tabs into code.

In Python, you can specify a tab character with the `\t` escape sequence:

Python

```
>>> print("Before\tAfter")
Before	After
```

The `\t` escape sequence changes the usual meaning of the letter `t`, making Python interpret the combination as a tab character.

Here's a list of escape sequences that cause Python to apply special meaning to some characters instead of interpreting them literally:

Escape Sequence	Escaped Interpretation
`\a`	ASCII Bell (`BEL`) character
`\b`	ASCII Backspace (`BS`) character
`\f`	ASCII Formfeed (`FF`) character
`\n`	ASCII Linefeed (`LF`) character
`\N{<name>}`	Character from Unicode database with given `<name>`
`\r`	ASCII Carriage return (`CR`) character
`\t`	ASCII Horizontal tab (`TAB`) character
`\uxxxx`	Unicode character with 16-bit hex value `xxxx`
`\Uxxxxxxxx`	Unicode character with 32-bit hex value `xxxxxxxx`
`\v`	ASCII Vertical tab (`VT`) character
`\ooo`	Character with octal value `ooo`
`\xhh`	Character with hex value `hh`

The newline or linefeed character (`\n`) is probably the most popular of these escape sequences. This sequence is commonly used to create nicely formatted text outputs that span multiple lines.

Here are a few examples of the escape sequences in action:

Python

```
>>> # Tab
>>> print("a\tb")
a	b

>>> # Linefeed
>>> print("a\nb")
a
b

>>> # Octal
>>> print("\141")
a

>>> # Hex
>>> print("\x61")
a

>>> # Unicode by name
>>> print("\N{rightwards arrow}")
→
```

These escape sequences are useful when you need to insert characters that aren't readily generated from the keyboard or aren't easily readable or printable.

Raw String Literals

With raw string literals, you can create strings that don't translate escape sequences. Any backslash characters are left in the string.

Note: To learn more about raw strings, check out the What Are Python Raw Strings? tutorial.

To create a raw string, you have to prepend the string literal with the letter `r` or `R`:

Python

```
>>> print("Before\tAfter")  # Regular string
Before	After

>>> print(r"Before\tAfter")  # Raw string
Before\tAfter
```

The raw string suppresses the meaning of the escape sequence, such as `\t`, and presents the characters as they are.

Raw strings are commonly used to create regular expressions because they allow you to use several different characters that may have special meanings without restrictions.

Note: To learn more about regular expressions in Python, check out the tutorials on regular expressions in Python Part 1 and Part 2.

For example, say that you want to create a regular expression to match email addresses. To do this, you can use a raw string to create the regular expression like in the code below:

Python

```
>>> import re

>>> pattern = r"\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b"
>>> pattern
'\\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\\.[A-Z|a-z]{2,}\\b'

>>> regex = re.compile(pattern)

>>> text = """
...     Please contact us at support@example.com
...     or sales@example.com for further information.
... """

>>> regex.findall(text)
['support@example.com', 'sales@example.com']
```

The `pattern` variable holds a raw string that makes up a regular expression to match email addresses. Note how the string contains several backslashes that are escaped and inserted into the resulting string as they are. Then, you compile the regular expression and check for matches in a sample piece of text.

Formatted String Literals

Formatted string literals, or f-strings for short, allow you to interpolate values into your strings and format them as you need.

To create a string with an f-string literal, you must prepend the literal with an `f` or `F` letter. F-strings let you interpolate values into **replacement fields** in your string literal. You create these fields using curly brackets.

Here's a quick example of an f-string literal:

Python

```
>>> name = "Jane"

>>> f"Hello, {name}!"
'Hello, Jane!'
```

In this example, you interpolate the `name` variable into your string using an f-string literal and a replacement field.

You'll learn more about using f-strings for string interpolation and formatting in the Doing String Interpolation and Formatting section.

The Built-in `str()` Function

You can create new strings using the built-in `str()` function. However, a more common use case for this function is to convert other data types into strings, which is also a way to create new strings.

Consider the following examples:

Python

```
>>> str()
''

>>> str(42)
'42'

>>> str(3.14)
'3.14'

>>> str([1, 2, 3])
'[1, 2, 3]'

>>> str({"one": 1, "two": 2, "three": 3})
"{'one': 1, 'two': 2, 'three': 3}"

>>> str({"A", "B", "C"})
"{'B', 'C', 'A'}"
```

In the first example, you call the `str()` function without an argument to create an empty string. Then, you use the function to convert objects from different built-in types into strings.

Using Operators on Strings

You can also use a few operators with string values as operands. For example, the `+` and `*` operators allow you to perform string concatenation and repetition, respectively. In the following sections, you'll learn how these and other operators work with strings.

Concatenating Strings: The `+` Operator

The `+` operator lets you concatenate strings. So, in this context, you can call it the **concatenation operator**. Concatenation involves joining two or more string objects to create a single new string:

Python

```
>>> greeting = "Hello"
>>> name = "Pythonista"

>>> greeting + ", " + name + "!!!"
'Hello, Pythonista!!!'
```

In this example, you use the plus operator to concatenate several string objects. You start with the `greeting` variable and add a comma followed by a space. Next, you add the name variable and finally, some exclamation points. Note that you've combined variables holding strings and string literals to build the final string.

Note: Often you'd use f-strings instead of + to concatenate strings.

The concatenation operator also has an augmented variation denoted by `+=`. This operator allows you to do something like `string = string + another_string` but in a shorter way:

Python

```
>>> file = "app"

>>> file += ".py"
>>> file
'app.py'
```

In this example, you create a filename incrementally. You start with the filename without the file extension. Then, you use the augmented concatenation operator to add the file extension to the existing filename.

Note that the augmented operator doesn't change the initial string because strings are immutable. Instead, it creates a new string and reassigns it to the name `file`.

Repeating Strings: The * Operator

The `*` operator allows you to repeat a given string a certain number of times. In this context, this operator is known as the **repetition operator**. Its syntax is shown below:

Python

```
new_string = string * n
new_string = n * string
```

The repetition operator takes two operands. One operand is the string that you want to repeat, while the other operand is an integer number representing the number of times you want to repeat the target string:

Python

```
>>> "=" * 10
'=========='

>>> 10 * "Hi!"
'Hi!Hi!Hi!Hi!Hi!Hi!Hi!Hi!Hi!Hi!'
```

In the first example, you repeat the `=` character ten times. In the second example, you repeat the text `Hi!` ten times as well. Note that the order of the operands doesn't affect the result.

The multiplier operand, `n`, is usually a positive integer. If it's a zero or negative integer, the result is an empty string:

Python

```
>>> "Hi!" * 0
''

>>> "Hi!" * -8
''
```

You need to be aware of this behavior in situations where you compute `n` dynamically, and `0` or negative values can occur. Otherwise, this operator comes in handy when you need to present some output to your users in a table format.

For example, say that you have the following data about your company's employees:

Python

```
>>> data = [
...     ["Alice", "25", "Python Developer"],
...     ["Bob", "30", "Web Designer"],
...     ["Charlie", "35", "Team Lead"],
... ]
```

You want to create a function that takes this data, puts it into a table, and prints it to the screen. In this situation, you can write a function like the following:

Python

```
>>> def display_table(data, headers):
...     max_len = max(len(header) for header in headers)
...     print(" | ".join(header.ljust(max_len) for header in headers))
...     sep = "-" * max_len
...     print("-|-".join(sep for _ in headers))
...     for row in data:
...         print(" | ".join(header.ljust(max_len) for header in row))
...
```

This function takes the employees' data as an argument and the headers for the table. Then, it gets the maximum header length, prints the headers using the pipe character (|) as a separator, and justifies the headers to the left.

To separate the headers from the table's content, you build a line of hyphens. To create the line, you use the repetition operator with the maximum header length as the multiplier operand. The effect of using the operator here is that you'll get a visual line under each header that's always as long as the longest header field. Finally, you print the employees' data.

Here's how the function works in practice:

Python

```
>>> data = [
...     ["Alice", "25", "Python Developer"],
...     ["Bob", "30", "Web Designer"],
...     ["Charlie", "35", "Team Lead"],
... ]
>>> headers = ["Name", "Age", "Job Title"]

>>> display_table(data, headers)
Name      | Age       | Job Title
----------|-----------|----------
Alice     | 25        | Python Developer
Bob       | 30        | Web Designer
Charlie   | 35        | Team Lead
```

Again, creating tables like this one is a great use case for the repetition operator because it allows you to create separators dynamically.

The repetition operator also has an augmented variation that lets you do something like `string = string * n` in a shorter way:

Python

```
>>> sep = "-"

>>> sep *= 10

>>> sep
'----------'
```

In this example, the `*=` operator repeats the string in `sep` ten times and assigns the result back to the `sep` variable.

Finding Substrings in a String: The `in` and `not in` Operators

Python also provides the `in` and `not in` operators that you can use with strings. The `in` operator returns `True` if the left-hand operand is contained within the right-hand one and False otherwise. This type of check is known as a **membership** test.

You can take advantage of a membership test when you need to determine if a substring appears in a given string:

Python

```
>>> "food" in "That's food for thought."
True

>>> "food" in "That's good for now."
False
```

In these examples, you check whether the string `"food"` is a substring of the strings on the right-hand side.

Note: To learn more about how to find substrings in existing strings, check out the How to Check if a Python String Contains a Substring tutorial.

The `not in` operator does the opposite check:

Python

```
>>> "z" not in "abc"
True

>>> "z" not in "xyz"
False
```

With the `not in` operator, you can check if a substring isn't found in a given string. If the substring isn't in the target string, then you get `True`. Otherwise, you get `False`.

Exploring Built-in Functions for String Processing

Python provides many functions that are built into the language and, therefore, are always available to you. Here are a few of these functions that are especially useful when you're processing strings:

Function	Description
`len()`	Returns the length of a string
`str()`	Returns a user-friendly string representation of an object
`repr()`	Returns a developer-friendly string representation of an object
`format()`	Allows for string formatting
`ord()`	Converts a character to an integer
`chr()`	Converts an integer to a character

In the following sections, you'll learn how to use these functions to work with your strings. To kick things off, you'll start by determining the length of an existing string with `len()`.

Finding the Number of Characters: `len()`

A common operation that you'll perform on strings is determining their number of characters. To complete this task, you can use the built-in `len()` function.

Note: To learn more about the `len()` function, check out the Using the `len()` Function in Python tutorial.

With a string as an argument, the `len()` function returns the length of the input string or the number of characters in that string. Here are a couple of examples of how to use `len()`:

Python

```
>>> len("Python")
6

>>> len("")
0
```

The word `"Python"` has six letters, so calling `len()` with this word as an argument returns `6`. Note that `len()` returns `0` when you call it with empty strings. The function calculates the number of characters in a string object, independent of how you constructed the string:

Python

```
>>> "\N{snake}"
'🐍'

>>> len("\N{snake}")
1

>>> len("🐍")
1
```

You use the `\N` escape sequence to define a string with a single snake emoji. Even though you wrote nine characters in the string literal, the string object consist of a single 🐍 character. And `len()` reports its length as one. You get the same result when you create the string with a literal emoji character.

Converting Objects Into Strings: `str()` and `repr()`

To convert objects into their string representation, you can use the built-in `str()` and `repr()` functions. With the `str()` function, you can convert a given object into its **user-friendly representation**. This type of string representation is targeted at end users.

For example, if you pass an object of a built-in type to `str()`, then you get a representation that typically consists of a string containing the literal that defines the object at hand:

Python

```
>>> str(42)
'42'

>>> str(3.14)
'3.14'

>>> str([1, 2, 3])
'[1, 2, 3]'

>>> str({"one": 1, "two": 2, "three": 3})
"{'one': 1, 'two': 2, 'three': 3}"

>>> str({"A", "B", "C"})
"{'B', 'C', 'A'}"
```

In these examples, the outputs show user-friendly representations of the input objects. These representations coincide with the objects' literals.

You can provide a user-friendly string representation for your custom classes through the `.__str__()` special method, as you'll learn in a moment.

Similarly, the built-in `repr()` function gives you a **developer-friendly representation** of the object at hand:

Python

```
>>> repr(42)
'42'

>>> repr(3.14)
'3.14'

>>> repr([1, 2, 3])
'[1, 2, 3]'

>>> repr({"one": 1, "two": 2, "three": 3})
"{'one': 1, 'two': 2, 'three': 3}"

>>> repr({"A", "B", "C"})
"{'B', 'C', 'A'}"
```

Ideally, you should be able to copy the output of `repr()` to re-create the original object. Also, this representation lets you know how the object is built. It's because of these two distinctions that you can say `repr()` returns a developer-friendly string representation of the object at hand.

As you can see in the example above, the output of `str()` and `repr()` is the same for built-in data types. Both functions return the object's literal as a string.

> **Note:** Behind the `str()` function, you have the `.__str__()` special method. Similarly, behind `repr()`, you have the `.__repr__()` method. To learn more about these special methods, check out the When Should You Use `.__repr__()` vs `.__str__()` in Python? tutorial.

To see the difference between `str()` and `repr()`, consider the `Person` class in the following example:

Python person.py

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __repr__(self):
        return f"{type(self).__name__}(name='{self.name}', age={self.age})"

    def __str__(self):
        return f"I'm {self.name}, and I'm {self.age} years old."
```

The `.__repr__()` method gives you a developer-friendly string representation of specific `Person` objects. You'll be able to re-create the object using the output of the `repr()` function. In contrast, the string representation that you get from calling `.__str__()` should aim to be readable and informative for end users. You can access this representation with the `str()` function.

Here's how the two functions work with an instance of your class:

Python

```
>>> john = Person("John Doe", 35)

>>> repr(john)
"Person(name='John Doe', age=35)"

>>> str(john)
"I'm John Doe, and I'm 35 years old."
```

In this example, you can use the output of `repr()` to re-create the object stored in `john`. You also have all the information about how this specific object was built. Meanwhile, the output of `str()` gives you no clue about how to build the object. However, it does provide useful information for the end user.

Formatting Strings: `format()`

You can use the built-in `format()` function to format strings. This function converts an input value into a formatted representation. To define the desired format, you can use a format specifier, which should be a string that follows the syntax defined in the string formatting mini-language.

Note: To learn more about format specifiers, check out the Python's Format Mini-Language for Tidy Strings tutorial.

Consider the following examples of using the `format()` function with different input values and format specifiers:

Python

```python
>>> import math
>>> from datetime import datetime

>>> format(math.pi, ".4f")  # Four decimal places
'3.1416'

>>> format(1000000, ",.2f")  # Thousand separators
'1,000,000.00'

>>> format("Header", "=^30")  # Centered and filled
'============Header============'

>>> format(datetime.now(), "%a %b %d, %Y")  # Date
'Mon Jul 29, 2024'
```

In these examples, the `".4f"` specifier formats the input value as a floating-point number with four decimal places.

With the `",.2f"` format specifier, you can format a number using commas as thousand separators and with two decimal places, which could be appropriate for formatting currency values.

Next, you use the `"=^30"` specifier to format the string `"Header"` centered in a width of `30` characters using the equal sign as a filler character. Finally, you use `"%a %b %d, %Y"` to format the date.

Processing Characters Through Code Points: `ord()` and `chr()`

The `ord()` function returns an integer value representing the Unicode **code point** for the given character.

At the most basic level, computers store all information as numbers. To represent character data, computers use a translation scheme that maps each character to its associated number, which is its code point.

The most commonly used scheme is ASCII, which covers the familiar Latin characters you're probably most accustomed to working with. For these characters, `ord()` returns the ASCII value that corresponds to the character at hand:

Python

```python
>>> ord("a")
97

>>> ord("#")
35
```

While the ASCII character set is fine for representing English, many natural languages are used worldwide, and countless symbols and glyphs appear in digital media. The complete set of characters that you may need to represent in code surpasses the Latin letters, numbers, and symbols.

Unicode is a generally accepted standard that attempts to provide a numeric code for every possible character in every possible natural language on every platform or operating system. The Unicode character set is the standard in Python.

Note: For more information about Unicode in Python, check out the Unicode & Character Encodings in Python: A Painless Guide tutorial and Python's Unicode Support in the Python documentation.

The `ord()` function will return numeric values for Unicode characters as well:

Python

```
>>> ord("€")
8364

>>> ord("∑")
8721
```

In the Unicode table, the `"€"` character has `8364` as its associated code point, and the `"∑"` character has the `8721` code point.

The `chr()` function does the reverse of `ord()`. It returns the character value associated with a given code point:

Python

```
>>> chr(97)
'a'

>>> chr(35)
'#'

>>> chr(8364)
'€'

>>> chr(8721)
'∑'
```

Given the numeric value `n`, `chr(n)` returns the character that corresponds to `n`. Note that `chr()` handles Unicode characters as well.

Indexing and Slicing Strings

Python's strings are ordered sequences of characters. Because of this, you can access individual characters from a string using the characters' associated **index**. What's a character's index? Each character in a string has an index that specifies its position. Indices are integer numbers that start at `0` and go up to the number of characters in the string minus one.

You can use these indices to perform two common operations:

1. **Indexing**: Consists of retrieving a given character using its associated index.
2. **Slicing**: Allows you to get a portion of an existing string using indices.

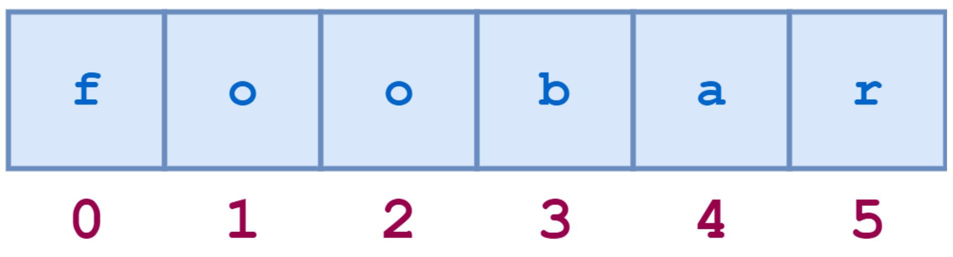
In the following sections, you'll learn how to run these two operations on your Python strings.

Indexing Strings

You can access individual characters in a string by specifying the string object followed by the character's index in square brackets (`[]`). This operation is known as **indexing**.

String indexing in Python is zero-based, which means that the first character in the string has an index of `0`, the next has an index of `1`, and so on. The index of the last character will be the length of the string minus one.

For example, a schematic diagram of the indices of the string `"foobar"` would look like this:



String Indices

You can access individual characters by their index:

Python

```
>>> s = "foobar"

>>> s[0]
'f'
>>> s[1]
'o'
>>> s[2]
'o'
```

Using the indexing operator `[index]`, you can access individual characters in an existing string. Note that attempting to index beyond the end of the string results in an error:

Python

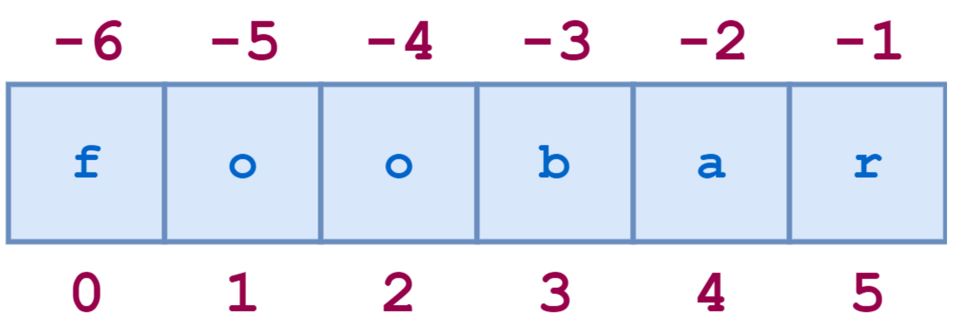
```
>>> len(s)
6

>>> s[6]
Traceback (most recent call last):
    ...
IndexError: string index out of range
```

When you use an index that's beyond the number of characters in the string minus one, you get an `IndexError` exception.

You can also use negative numbers as indices. If you do that, then the indexing occurs from the end of the string backward. Index `-1` refers to the last character, index `-2` refers to the second-last character, and so on.

Here's the diagram showing both positive and negative indices in the string `"foobar"`:



Positive and Negative String Indices

Here are some examples of negative indexing:

Python

```
>>> s[-1]
'r'
>>> s[-2]
'a'
>>> s[-3]
'b'
```

With negative indices, you can access the characters from the end of the string to its beginning. Again, attempting to index with negative numbers beyond the beginning of the string results in an error:

Python

```
>>> s[-7]
Traceback (most recent call last):
    ...
IndexError: string index out of range
```

Because the `"foobar"` string only has six characters, the last negative index that you can use is `-6`, which corresponds to the first character in the string.

In short, for any non-empty string, `s[len(s) - 1]` and `s[-1]` both return the last character. Similarly, `s[0]` and `s[-len(s)]` return the first character. Note that no index makes sense on an empty string.

Finally, you should keep in mind that because strings are immutable, you can't perform index assignments on them. The following example illustrates this fact:

Python

```
>>> greeting = "Hello!"

>>> greeting[5] = "?"
Traceback (most recent call last):
    ...
TypeError: 'str' object does not support item assignment
```

If you try to change a character in an existing string, then you'll get a `TypeError` telling you that strings don't support item assignment.

Slicing Strings

Python allows you to extract substrings from a string. This operation is known as **slicing**. If `s` is a string, then an expression of the form `s[m:n]` returns the portion of `s` starting at index `m`, and up to but not including index `n`:

Python

```
>>> s = "foobar"

>>> s[2:5]
'oba'
```

The first index sets the starting point of the indexing operation. The second index specifies the first character that isn't included in the result—the character `"r"` with index `5` in the example above.

Note: String indices are zero-based. This means that the first character in a string has an index of `0`. This applies to both standard indexing and slicing.

If you omit the first index, then the slicing starts at the beginning of the string:

Python

```
>>> s[:4]
'foob'

>>> s[0:4]
'foob'
```

Therefore, slicings like `s[:m]` and `s[0:m]` are equivalent. Similarly, if you omit the second index as in `s[n:]`, then the slicing extends from the first index to the end of the string. This is an excellent and concise alternative to the more cumbersome `s[n:len(s)]` syntax:

Python

```
>>> s[2:]
'obar'

>>> s[2:len(s)]
'obar'
```

When you omit the last index, the slicing operation extends up to the end of the string.

For any string `s` and any integer `n`, provided that `n ≥ 0`, you'll have `s[:n] + s[n:]` is equal to `s`:

Python

```
>>> s[:4] + s[4:]
'foobar'

>>> s[:4] + s[4:] == s
True
```

The first slicing extracts the characters from the beginning and up to index `4`, which isn't included. The second slicing extracts the characters from `4` to the end of the string. So, the result will contain the same characters as the original string.

Omitting both indices returns the original string in its entirety. It doesn't return a copy but a reference to the original string:

Python

```
>>> t = s[:]

>>> id(s)
59598496

>>> id(t)
59598496

>>> s is t
True
```

When you omit both indices, you tell Python that you want to extract a slice containing the whole string. Again, for efficiency reasons, Python returns a reference instead of a copy of the original string. You can confirm this behavior using the built-in `id()` function.

If the first index in a slicing is greater than or equal to the second index, then Python returns an empty string:

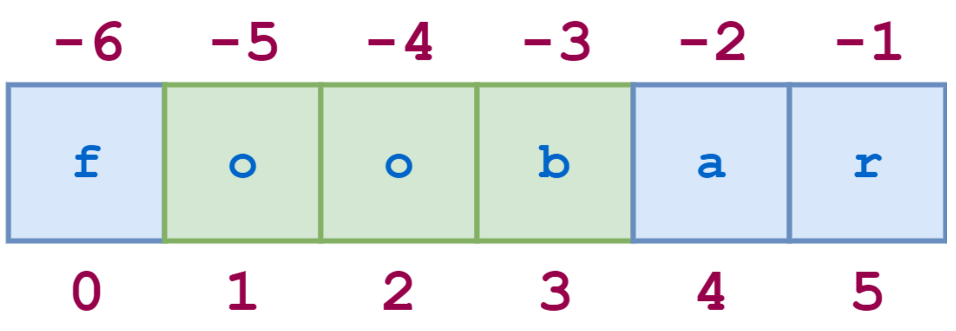
Python

```
>>> s[2:2]
''

>>> s[4:2]
''
```

As you can see, this behavior provides a somewhat obfuscated way to generate empty strings.

You can also use negative indices in slicing. Just as with indexing, `-1` refers to the last character, `-2` to the second-last character, and so on. The diagram below shows how to slice the substring `"oob"` from the string `"foobar"` using both positive and negative indices:



String Slicing With Positive and Negative Indices

Here's the corresponding Python code:

Python

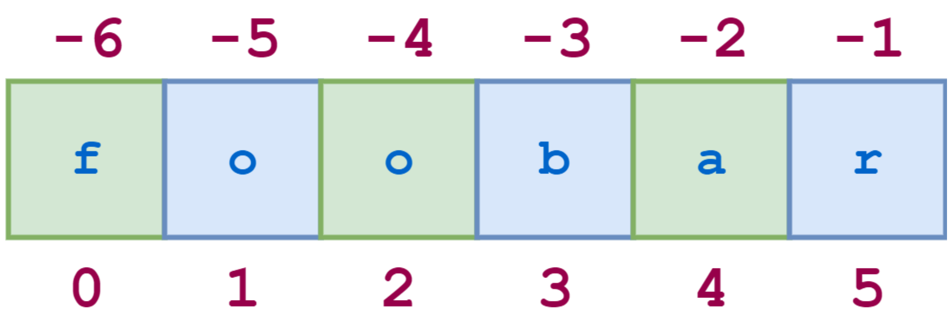
```
>>> s[-5:-2]
'oob'

>>> s[1:4]
'oob'
```

Slicing with negative indices really shines when you need to get slices that have the last character of the string as a reference point.

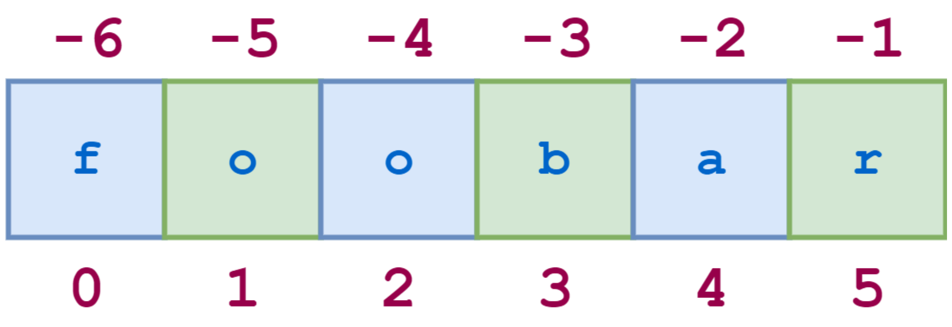
There's one more variant of the slicing syntax to discuss. An additional colon (`:`) and a third index designate a stride, also known as a step. This index indicates how many characters to jump after retrieving each character in the slice.

For example, for the string `"foobar"`, the slice `[0:6:2]` starts with the first character and ends with the last character, skipping every second character. This behavior is shown in the following diagram, where the green items are the ones included in the slice:



String Indexing With Stride

Similarly, `[1:6:2]` specifies a slice that starts with the second character, which is index `1`, and ends with the last character. Again, the stride value `2` skips every other character:



Another String Indexing With Stride

The illustrative REPL code is shown below:

Python

```python
>>> s[0:6:2]
'foa'

>>> s[1:6:2]
'obr'
```

As with any slicing, the first and second indices can be omitted and default to the first and last characters, respectively:

Python

```python
>>> numbers = "12345" * 5
>>> numbers
'1234512345123451234512345'

>>> numbers[::5]
'11111'
>>> numbers[4::5]
'55555'
```

In the first slicing, you omit the first and second indices, making the slice start at index `0` and go up to the end of the string. In the second slicing, you only omit the second index, and the slice goes up to the end of the string.

You can also specify a negative stride value, in which case Python steps backward through the string. Note that the initial index should be greater than the ending index:

Python

```python
>>> s[5:0:-2]
'rbo'
```

In the above example, `[5:0:-2]` means this: *start at the last character and step backward by `2`, up to but not including the first character.*

When you step backward, if the first and second indices are omitted, the defaults are reversed intuitively. In other words, the initial index defaults to the end of the string, and the ending index defaults to the beginning.

Here's an example:

Python

```
>>> numbers[::-5]
'55555'
```

Using a step equal to `-1` provides a common technique for reversing strings:

Python

```
>>> principle = "Beautiful is better than ugly."

>>> principle[::-1]
'.ylgu naht retteb si lufituaeB'
```

To dive deeper into how to reverse strings, check out the Reverse Strings in Python: `reversed()`, Slicing, and More tutorial.

Doing String Interpolation and Formatting

String interpolation refers to building new strings by inserting values into a string that works as a template with replacement fields for each value you insert. **String formatting**, on the other hand, involves applying a desired format to the values you insert in a given string through interpolation.

Note: To dive deeper into string interpolation, check out the String Interpolation in Python: Exploring Available Tools tutorial.

Similarly, if you want to learn more about string formatting, check out the Python String Formatting: Available Tools and Their Features tutorial.

In the following sections, you'll learn the basics of string interpolation and formatting in Python.

Using F-Strings

You can also use f-strings to interpolate values into a string and format them, as you quickly touched on in the section about f-string literals. To do this, you use format specifiers that use the syntax defined in Python's string format mini-language. For example, here's how you can present numeric values using a currency format:

Python

```
>>> income = 1234.1234

>>> f"Income: ${income:.2f}"
'Income: $1234.12'
```

Inside the replacement field delimited by the curly brackets, you have the variable you want to interpolate and the format specifier, which is the string that starts with a colon (`:`). In this example, the format specifier defines a floating-point number with two decimal places.

Note: To learn more about using f-strings for string interpolation and formatting, check out the Python's F-String for String Interpolation and Formatting tutorial.

F-strings have a clean syntax and are quite popular in Python code. However, they're only suitable for those situations where you want to do the interpolation eagerly because the interpolated values are inserted when Python executes the strings.

You can't use f-strings for lazy interpolation. In other words, you can't use f-string for those situations where you need a reusable template that you'll fill with values dynamically in different parts of your code. In this situation, you can use the `str.format()` method.

Using the `.format()` Method

Python’s `str` class has a method called `.format()` that allows for string interpolation and formatting. This method is pretty powerful. Like f-strings, it supports the string formatting mini-language, but unlike f-strings, it allows for both eager and lazy string interpolation.

Consider the following example that uses `.format()` to create a balance report:

Python

```python
>>> debit = 300.00
>>> credit = 450.00

>>> template = """
... Account Report
... Credit:  ${credit:.2f}
... Debit:  -${debit:.2f}
... ________________
... Balance: ${balance:.2f}"""

>>> print(
...     template.format(
...         credit=credit,
...         debit=debit,
...         balance=credit - debit,
...     )
... )

Account Report
Credit:  $450.00
Debit:  -$300.00
________________
Balance: $150.00
```

In this example, you first define two variables to hold the debit and credit of a bank account. Then, you create a template to construct a report showing the account’s credit, debit, and balance. Finally, you call `.format()` with the required argument to create the report.

Using the Modulo Operator (`%`)

Python has another tool you can use for string interpolation and formatting called the modulo operator (`%`). This operator was the first tool Python provided for string interpolation and formatting. It also allows you to do both eager and lazy interpolation.

Note: To learn more about the modulo operator for string formatting and interpolation, check out the Modulo String Formatting in Python tutorial.

While you can use the modulo operator for string formatting, its formatting capabilities are limited and it doesn’t support the string formatting mini-language.

Here’s how you would use the modulo operator to code the example from the previous section:

Python

```
>>> debit = 300.00
>>> credit = 450.00

>>> template = """
... Account Report
... Credit:  $%(credit).2f
... Debit:  -$%(debit).2f
... ________________
... Balance: $%(balance).2f"""

>>> print(
...     template
...     % {
...         "credit": credit,
...         "debit": debit,
...         "balance": credit - debit,
...     }
... )

Account Report
Credit:  $450.00
Debit:  -$300.00
________________
Balance: $150.00
```

In this version of the report template, you use named placeholders with the modulo operator syntax. To provide the values to interpolate, you use a dictionary. The operator unpacks it using the keys that match the replacement fields.

Exploring `str` Class Methods

Methods are functions that you define inside classes. Like a function, you call a method to perform a specific task. Unlike a function, you invoke a method on a specific object or class so it has knowledge of its containing object during execution.

The syntax for invoking a method on an object is shown below:

```
object.method([arg_0, arg_2, ..., arg_n])

class.method([arg_0, arg_2, ..., arg_n])
```

This invokes method `.method()` on object `object`. Inside the parentheses, you can specify the arguments for the method to work, but these arguments are optional. Note that you can have both instance and class methods.

Note: In the above syntax, the arguments specified in square brackets (`[]`) are optional and the brackets aren't part of the syntax. You'll use this same convention throughout the following sections.

Python's `str` class provides a rich set of methods that you can use to format your strings in several different ways. In the next section, you'll learn the basics of some of the more commonly used `str` methods.

Manipulating Casing

The `str` methods you'll look at in this section allow you to perform case conversion on the target string. Note that the methods in this section only affect letters. Non-letter characters remain the same because they don't have uppercase and lowercase variations.

Note: Keep in mind that because strings are immutable, when string methods apply changes to an existing string they return a new string object.

To kick things off, you'll start by capitalizing some strings using the `.capitalize()` method.

`.capitalize()`

The `.capitalize()` method returns a copy of the target string with its first character converted to uppercase, and all other characters converted to lowercase:

Python

```
>>> "foO BaR 123 BAZ quX".capitalize()
'Foo bar 123 baz qux'
```

In this example, only the first letter in the target string is converted to uppercase. The rest of the letters are converted to lowercase.

`.lower()`

The `.lower()` method returns a copy of the target string with all alphabetic characters converted to lowercase:

Python

```
>>> "FOO Bar 123 baz qUX".lower()
'foo bar 123 baz qux'
```

You can see in this example how all the uppercase letters were converted to lowercase.

`.swapcase()`

The `.swapcase()` method returns a copy of the target string with uppercase alphabetic characters converted to lowercase and vice versa:

Python

```
>>> "FOO Bar 123 baz qUX".swapcase()
'foo bAR 123 BAZ Qux'
```

In this example, the call to `.swapcase()` turned the lowercase letters into uppercase and the uppercase letters into lowercase.

`.title()`

The `.title()` method returns a copy of the target string in which the first letter of each word is converted to uppercase, and the remaining letters are lowercase:

Python

```
>>> "the sun also rises".title()
'The Sun Also Rises'
```

Note that this method doesn't attempt to distinguish between important and unimportant words, and it doesn't handle apostrophes, possessives, or acronyms gracefully:

Python

```
>>> "what's happened to ted's IBM stock?".title()
"What'S Happened To Ted'S Ibm Stock?"
```

The `.title()` method converts the first letter of each word in the target string to uppercase. As you can see, it doesn't do a great job with apostrophes and acronyms!

`.upper()`

The `.upper()` method returns a copy of the target string with all alphabetic characters converted to uppercase:

Python

```
>>> "FOO Bar 123 baz qUX".upper()
'FOO BAR 123 BAZ QUX'
```

By calling `.upper()` on a string, you get a new string with all the letters in uppercase.

Finding and Replacing Substrings

The methods in this section provide various ways to search the target string for a specified substring.

Each method supports optional `start` and `end` arguments. These arguments mean that the action of the method is restricted to the portion of the target string starting with the character at index `start` and up to but not including the character at index `end`. If `start` is specified but `end` isn't, then the method applies to the portion of the target string from `start` through the end of the string.

`.count(sub[, start[, end]])`

The `.count(sub)` method returns the number of non-overlapping occurrences of the substring `sub` in the target string:

Python

```
>>> "foo goo moo".count("oo")
3
```

The count is restricted to the number of occurrences within the substring indicated by `start` and `end` if you specify them:

Python

```
>>> "foo goo moo".count("oo", 0, 8)
2
```

In this example, the search starts at index `0` and ends at index `8`. Because of this, the last occurrence of `"oo"` isn't counted.

`.find(sub[, start[, end]])`

You can use `.find()` to check whether a string contains a particular substring. Calling `.find(sub)` returns the lowest index in the target string where `sub` is found:

Python

```
>>> "foo bar foo baz foo qux".find("foo")
0
```

Note that `.find()` returns `-1` if the specified substring isn't found in the target string:

Python

```
>>> "foo bar foo baz foo qux".find("grault")
-1
```

Again, the search is restricted to the substring indicated by `start` and `end` if you specify them:

Python

```
>>> "foo bar foo baz foo qux".find("foo", 4)
8

>>> "foo bar foo baz foo qux".find("foo", 4, 7)
-1
```

With the `start` and `end` indices, you restrict the search to a portion of the target string.

`.index(sub[, start[, end]])`

The `.index()` method is similar to `.find()`, except that it raises an exception rather than returning `-1` if `sub` isn't found in the target string:

Python

```
>>> "foo bar foo baz foo qux".index("foo")
0

>>> "foo bar foo baz foo qux".index("grault")
Traceback (most recent call last):
    ...
ValueError: substring not found
```

If the substring exists in the target string, then `.index()` returns the index at which the first instance of the substring starts. If the substring isn't found in the target string, then you get a `ValueError` exception.

`.rfind(sub[, start[, end]])`

The `.rfind(sub)` call returns the highest index in the target string where the substring `sub` is found:

Python

```
>>> "foo bar foo baz foo qux".rfind("foo")
16
```

As with `.find()`, if the substring isn't found, then `.rfind()` returns `-1`:

Python

```
>>> "foo bar foo baz foo qux".rfind("grault")
-1
```

You can restrict the search to the substring indicated by `start` and `end`:

Python

```
>>> "foo bar foo baz foo qux".rfind("foo", 0, 14)
8

>>> "foo bar foo baz foo qux".rfind("foo", 10, 14)
-1
```

Again, with the `start` and `end` indices, you restrict the search to a portion of the target string.

`.rindex(sub[, start[, end]])`

The `.rindex()` method is similar to `.rfind()`, except that it raises an exception if `sub` isn't found in the target string:

Python

```
>>> "foo bar foo baz foo qux".rindex("foo")
16

>>> "foo bar foo baz foo qux".rindex("grault")
Traceback (most recent call last):
    ...
ValueError: substring not found
```

If the substring isn't found in the target string, then `rindex()` raises a `ValueError` exception.

`.startswith(prefix[, start[, end]])`

The `.startswith(prefix)` call returns `True` if the target string starts with the specified `prefix` and `False` otherwise:

Python

```
>>> "foobar".startswith("foo")
True

>>> "foobar".startswith("bar")
False
```

The comparison is restricted to the substring indicated by `start` and `end` if they're specified:

Python

```
>>> "foobar".startswith("bar", 3)
True

>>> "foobar".startswith("bar", 3, 5)
False
```

In these examples, you use the `start` and `end` indices to restrict the search to a given portion of the target string.

`.endswith(suffix[, start[, end]])`

The `.endswith(suffix)` call returns `True` if the target string ends with the specified `suffix` and `False` otherwise:

Python

```
>>> "foobar".endswith("bar")
True

>>> "foobar".endswith("foo")
False

>>> "foobar".endswith("oob", 0, 4)
True

>>> "foobar".endswith("oob", 2, 4)
False
```

The comparison is restricted to the substring indicated by `start` and `end` if you specify them.

Classifying Strings

The methods in this section classify a string based on its characters. In all cases, the methods are **predicates**, meaning they return `True` or `False` depending on the condition they check.

`.isalnum()`

The `.isalnum()` method returns `True` if the target string isn't empty and all its characters are alphanumeric, meaning either a letter or number. Otherwise, it returns `False`:

Python

```
>>> "abc123".isalnum()
True

>>> "abc$123".isalnum()
False
```

In the first example, you get `True` because the target string contains only letters and numbers. In the second example, you get `False` because of the `"$"` character.

`.isalpha()`

The `.isalpha()` method returns `True` if the target string isn't empty and all its characters are alphabetic. Otherwise, it returns `False`:

Python

```
>>> "ABCabc".isalpha()
True

>>> "abc123".isalpha()
False

>>> "ABC abc".isalpha()
False
```

The `.isalpha()` method allows you to check whether all the characters in a given string are letters. Note that whitespaces aren't considered alpha characters, which makes sense but might be missed if you're working with normal text.

`.isdigit()`

You can use the `.isdigit()` method to check whether your string is made of only digits:

Python

```
>>> "123".isdigit()
True

>>> "123abc".isdigit()
False
```

The `.isdigit()` method returns `True` if the target string is not empty and all its characters are numeric digits. Otherwise, it returns `False`.

`.isidentifier()`

The `.isidentifier()` method returns `True` if the target string is a valid Python identifier according to the language definition. Otherwise, it returns `False`:

Python

```
>>> "foo32".isidentifier()
True

>>> "32foo".isidentifier()
False

>>> "foo$32".isidentifier()
False
```

It's important to note that `.isidentifier()` will return `True` for a string that matches a Python keyword even though that wouldn't be a valid identifier:

Python

```
>>> "and".isidentifier()
True
```

You can test whether a string matches a Python keyword using a function called `iskeyword()`, which is contained in a module called `keyword`. One possible way to do this is shown below:

Python

```
>>> from keyword import iskeyword

>>> iskeyword("and")
True
```

If you want to ensure that a string serves as a valid Python identifier, you should check that `.isidentifier()` returns `True` and that `iskeyword()` returns `False`.

`.islower()`

The `.islower()` method returns `True` if the target string isn't empty and all its alphabetic characters are lowercase. Otherwise, it returns `False`:

Python

```
>>> "abc".islower()
True

>>> "abc1$d".islower()
True

>>> "Abc1$D".islower()
False
```

Note that non-alphabetic characters are ignored when you call the `.islower()` method on a given string.

`.isprintable()`

The `.isprintable()` method returns `True` if the target string is empty or if all its alphabetic characters are printable:

Python

```
>>> "a\tb".isprintable()
False

>>> "a b".isprintable()
True

>>> "".isprintable()
True

>>> "a\nb".isprintable()
False
```

You get `False` if the target string contains at least one non-printable character. Again, non-alphabetic characters are ignored.

Note that `.isprintable()` is one of two `.is*()` methods that return `True` if the target string is empty. The second one is `.isascii()`.

`.isspace()`

The `.isspace()` method returns `True` if the target string isn't empty and all its characters are whitespaces. Otherwise, it returns `False`.

The most commonly used whitespace characters are space (`" "`), tab (`"\t"`), and newline (`"\n"`):

Python

```
>>> " \t \n ".isspace()
True

>>> "   a   ".isspace()
False
```

There are a few other ASCII characters that qualify as whitespace characters, and if you account for Unicode characters, there are quite a few beyond that:

Python

```
>>> "\f\u2005\r".isspace()
True
```

The `"\f"` and `"\r"` combinations are the escape sequences for the ASCII **form feed** and **carriage return** characters. The `"\u2005"` combination is the escape sequence for the Unicode **four-per-em** space.

`.istitle()`

The `.istitle()` method returns `True` if the target string isn't empty, the first alphabetic character of each word is uppercase, and all other alphabetic characters in each word are lowercase. It returns `False` otherwise:

Python

```
>>> "This Is A Title".istitle()
True

>>> "This is a title".istitle()
False

>>> "Give Me The #$#@ Ball!".istitle()
True
```

This method returns `True` if the string is title-cased as would result from calling `.title()`. It returns `False` otherwise.

`.isupper()`

The `.isupper()` method returns `True` if the target string isn't empty and all its alphabetic characters are uppercase. Otherwise, it returns `False`:

Python

```
>>> "ABC".isupper()
True

>>> "ABC1$D".isupper()
True

>>> "Abc1$D".isupper()
False
```

Again, Python ignores non-alphabetic characters when you call the `.isupper()` method on a given string object.

Formatting Strings

The methods in this section allow you to modify or enhance the format of a string in many different ways. You'll start by learning how to center your string within a given space.

`.center(width[, fill])`

The `.center(width)` call returns a string consisting of the target string centered in a field of `width` characters. By default, padding consists of the ASCII space character:

Python

```
>>> "foo".center(10)
'   foo    '
```

If you specify the optional `fill` argument, then it's used as the padding character:

Python

```
>>> "bar".center(10, "-")
'---bar----'

>>> "foo".center(2)
'foo'
```

Note that if the target string is as long as `width` or longer, then it's returned unchanged.

`.expandtabs(tabsize=8)`

The `.expandtabs()` method replaces each tab character (`"\t"`) found in the target string with spaces. By default, spaces are filled in assuming eight characters per tab:

Python

```python
>>> "a\tb\tc".expandtabs()
'a       b       c'

>>> "aaa\tbbb\tc".expandtabs()
'aaa     bbb     c'

>>> "a\tb\tc".expandtabs(4)
'a   b   c'

>>> "aaa\tbbb\tc".expandtabs(tabsize=4)
'aaa bbb c'
```

In the final example, you use the `tabsize` argument, which is an optional argument that specifies an alternate tab size.

`.ljust(width[, fill])`

The `.ljust(width)` call returns a string consisting of the target string left-justified in a field of `width` characters. By default, the padding consists of the ASCII space character:

Python

```python
>>> "foo".ljust(10)
'foo       '
```

If you specify the optional `fill` argument, then it's used as the padding character:

Python

```python
>>> "foo".ljust(10, "-")
'foo-------'

>>> "foo".ljust(2)
'foo'
```

If the target string is as long as `width` or longer, then it's returned unchanged.

`.rjust(width[, fill])`

The `.rjust(width)` call returns a string consisting of the target string right-justified in a field of `width` characters. By default, the padding consists of the ASCII space character:

Python

```python
>>> "foo".rjust(10)
'       foo'
```

If you specify the optional `fill` argument, then it's used as the padding character:

Python

```python
>>> "foo".rjust(10, "-")
'-------foo'
```

If the target string is as long as `width` or longer, then it's returned unchanged:

Python

```python
>>> "foo".rjust(2)
'foo'
```

When the target string is as long as `width` or longer, then `.rjust()` won't have space to justify the text, so you get the original string back.

`.removeprefix(prefix)`

The `.removeprefix()` method returns a copy of the target string with `prefix` removed from the beginning:

Python

```
>>> "http://python.org".removeprefix("http://")
'python.org'

>>> "http://python.org".removeprefix("python")
'http://python.org'
```

The prefix is removed if the target string begins with that exact substring. If the original string doesn't begin with `prefix`, then the string is returned unchanged.

`.removesuffix(suffix)`

The `.removesuffix()` method returns a copy of the target string with `suffix` removed from the end:

Python

```
>>> "http://python.org".removesuffix(".org")
'http://python'

>>> "http://python.org".removesuffix("python")
'http://python.org'
```

The suffix is removed if the target string ends with that exact substring. If the original string doesn't end with `suffix`, then the string is returned unchanged. The `.removeprefix()` and `.removesuffix()` methods were introduced in Python 3.9.

`.lstrip([chars])`

The `.lstrip()` method returns a copy of the target string with any whitespace characters removed from the left end:

Python

```
>>> "    foo bar baz    ".lstrip()
'foo bar baz    '

>>> "\t\nfoo\t\nbar\t\nbaz".lstrip()
'foo\t\nbar\t\nbaz'
```

If you specify the optional `chars` argument, then it's a string that specifies the set of characters to be removed:

Python

```
>>> "http://realpython.com".lstrip("/:htp")
'realpython.com'
```

In this example, you remove the `http://` prefix from the input URL using the `.lstrip()` method with the `"/:htp"` characters as an argument.

Note that this call to `.lstrip()` works only because the URL doesn't start with any of the target characters:

Python

```
>>> "http://python.org".lstrip("/:htp")
'ython.org'
```

In this example, the result isn't correct because the URL starts with a `p`, which is included in the set of letters that you want to remove. You should use `.removeprefix()` instead.

`.rstrip([chars])`

The `.rstrip()` method with no arguments returns a copy of the target string with any whitespace characters removed from the right end:

Python

```python
>>> "   foo bar baz   ".rstrip()
'   foo bar baz'
>>> "foo\t\nbar\t\nbaz\t\n".rstrip()
'foo\t\nbar\t\nbaz'
```

If you specify the optional `chars` argument, then it should be a string that specifies the set of characters to be removed:

Python

```python
>>> "foo.$$$;".rstrip(";$.")
'foo'
```

By providing a string value to `chars`, you can control the set of characters to remove from the right side of the target string.

Note: To learn more about `.rstrip()` and `.lstrip()`, check out the How to Strip Characters From a Python String tutorial.

`.strip([chars])`

The `.strip()` method is equivalent to invoking `.lstrip()` and `.rstrip()` in succession. Without the `chars` argument, it removes leading and trailing whitespace in one go:

Python

```python
>>> "   foo bar baz   ".strip()
'foo bar baz'
```

As with `.lstrip()` and `.rstrip()`, the optional `chars` argument specifies the set of characters to be removed:

Python

```python
>>> "https://realpython.com".strip("htps:/")
'realpython.com'
```

Again, with `chars`, you can control the characters you want to remove from the original string, which defaults to whitespaces.

`.replace(old, new[, count])`

To replace a substring of a string, you can use the `.replace()` method. This method returns a copy of the target string with all the occurrences of the `old` substring replaced by `new`:

Python

```python
>>> "foo bar foo baz foo qux".replace("foo", "grault")
'grault bar grault baz grault qux'
```

If you specify the optional `count` argument, then a maximum of `count` replacements are performed, starting at the left end of the target string:

Python

```python
>>> "foo bar foo baz foo qux".replace("foo", "grault", 2)
'grault bar grault baz foo qux'
```

The `count` argument lets you specify how many replacements you want to perform in the original string.

`.zfill(width)`

The `.zfill(width)` call returns a copy of the target string left-padded with zeroes to the specified `width`:

Python

```
>>> "42".zfill(5)
'00042'
```

If the target string contains a leading sign, then it remains at the left edge of the result string after zeros are inserted:

Python

```
>>> "+42".zfill(8)
'+0000042'

>>> "-42".zfill(8)
'-0000042'
```

If the target string is as long as `width` or longer, then it's returned unchanged:

Python

```
>>> "-42".zfill(2)
'-42'
```

Python will still zero-pad a string that isn't a numeric value:

Python

```
>>> "foo".zfill(6)
'000foo'
```

The `.zfill()` method is useful for representing numeric values. However, it also works with textual strings.

Joining and Splitting Strings

The methods you'll explore in this section allow you to convert between a string and some composite data types by joining objects together to make a string or by breaking a string up into pieces.

These methods operate on or return iterables, which are collections of objects. For example, many of these methods return a `list` or a `tuple`.

`.join(iterable)`

The `.join()` method takes an iterable of string objects and returns the string that results from concatenating the objects in the input iterable separated by the target string.

Note: To learn more about string concatenation, check out the Splitting, Concatenating, and Joining Strings tutorial.

Note that `.join()` is invoked on a string that works as the separator between each item in the input iterable, which must contain string objects.

In the following example, the separator is a string consisting of a comma and a space. The input iterable is a list of string values:

Python

```
>>> ", ".join(["foo", "bar", "baz", "qux"])
'foo, bar, baz, qux'
```

The result of this call to `.join()` is a single string consisting of string objects separated by commas.

Again, the input iterable must contain string objects. Otherwise, you'll get an error:

Python

```
>>> "---".join(["foo", 23, "bar"])
Traceback (most recent call last):
    ...
TypeError: sequence item 1: expected str instance, int found
```

This example fails because the second object in the input iterable isn't a string object but an integer number.

When you have iterables of values that aren't strings, then you can use the `str()` function to convert them before passing them to `.join()`. Consider the following example:

Python

```
>>> numbers = [1, 2, 3]

>>> "->".join(str(number) for number in numbers)
'1->2->3'
```

In this example, you use the `str()` function to convert the values in `numbers` to strings before feeding them to `.join()`. To do the conversion, you use a generator expression.

`.partition(sep)`

The `.partition(sep)` call splits the target string at the first occurrence of string `sep`. The return value is a tuple with three objects:

1. The portion of the target string that precedes `sep`
2. The `sep` object itself
3. The portion of the target string that follows `sep`

Here are a couple of examples of `.partition()` in action:

Python

```
>>> "foo.bar".partition(".")
('foo', '.', 'bar')

>>> "foo@@bar@@baz".partition("@@")
('foo', '@@', 'bar@@baz')

>>> "foo.bar@@".partition("@@")
('foo.bar', '@@', '')

>>> "foo.bar".partition("@@")
('foo.bar', '', '')
```

Note that if the string ends with the target `sep`, then the last item in the tuple is an empty string. Similarly, if `sep` isn't found in the target string, then the returned tuple contains the string followed by two empty strings, as you see in the final example.

`.rpartition(sep)`

The `.rpartition(sep)` call works like `.partition(sep)`, except that the target string is split at the last occurrence of `sep` instead of the first:

Python

```
>>> "foo@@bar@@baz".partition("@@"")
('foo', '@@', 'bar@@baz')

>>> "foo@@bar@@baz".rpartition("@@")
('foo@@bar', '@@', 'baz')
```

In this case, the string partition is done starting from the right side of the target string.

`.split(sep=None, maxsplit=-1)`

Without arguments, `.split()` splits the target string into substrings delimited by any sequence of whitespace and returns the substrings as a list:

Python

```python
>>> "foo bar baz qux".split()
['foo', 'bar', 'baz', 'qux']

>>> "foo\n\tbar   baz\r\fqux".split()
['foo', 'bar', 'baz', 'qux']
```

If you specify the `sep` argument, then it's used as the delimiter for the splitting:

Python

```python
>>> "foo.bar.baz.qux".split(".")
['foo', 'bar', 'baz', 'qux']
```

When `sep` is explicitly given as a delimiter, consecutive instances of the delimiter in the target string are assumed to delimit empty strings:

Python

```python
>>> "foo...bar".split(".")
['foo', '', '', 'bar']
```

However, consecutive whitespace characters are combined into a single delimiter, and the resulting list will never contain empty strings:

Python

```python
>>> "foo\t\t\tbar".split()
['foo', 'bar']
```

If the optional parameter `maxsplit` is specified, then a maximum of that many splits are performed, starting from the left end of the target string:

Python

```python
>>> "foo.bar.baz.qux".split(".", 1)
['foo', 'bar.baz.qux']
```

In this example, `.split()` performs only one split, starting from the left end of the target string.

`.rsplit(sep=None, maxsplit=-1)`

The `.rsplit()` method behaves like `.split()`, except that if `maxsplit` is specified, then the splits are counted from the right end of the target string rather than from the left end:

Python

```python
>>> "foo.bar.baz.qux".split(".", 1)
['foo.bar.baz', 'qux']
```

If `maxsplit` isn't specified, then the results of `.split()` and `.rsplit()` are indistinguishable.

`.splitlines([keepends])`

The `.splitlines()` method splits the target string into lines and returns them in a list. The following escape sequences can work as line boundaries:

Sequence	**Description**
`\n`	Newline

Sequence	Description
`\r`	Carriage return
`\r\n`	Carriage return + line feed
`\v` or `\x0b`	Line tabulation
`\f` or `\x0c`	Form feed
`\x1c`	File separator
`\x1d`	Group separator
`\x1e`	Record separator
`\x85`	Next line (C1 control code)
`\u2028`	Unicode line separator
`\u2029`	Unicode paragraph separator

Here's an example of using several different line separators:

Python

```
>>> "foo\nbar\r\nbaz\fqux\u2028quux".splitlines()
['foo', 'bar', 'baz', 'qux', 'quux']
```

If consecutive line boundary characters are present in the target string, then they're assumed to delimit blank lines, which will appear in the result list as empty strings:

Python

```
>>> "foo\f\f\fbar".splitlines()
['foo', '', '', 'bar']

>>> "foo\nbar\nbaz\nqux".splitlines(True)
['foo\n', 'bar\n', 'baz\n', 'qux']
```

Note that if you set the optional `keepends` argument to `True`, then the line boundaries are retained in the result strings.

Conclusion

You now know about Python's many tools for **string** creation and manipulation, including string literals, operators, and built-in functions. You've also learned how to extract items and parts of existing strings using the indexing and slicing operators. Finally, you've looked at the methods the `str` class provides for string formatting and processing.

In this tutorial, you've learned how to:

- Create strings using **literals** and the `str()` function
- Use **operators** and **built-in functions** with string objects
- Extract characters and portions of a string through **indexing** and **slicing**
- **Interpolate** and **format** values into your strings
- Use various **string methods**

With this knowledge, you're ready to start creating and processing textual data in your Python code. You can always come back to this tutorial if you ever need a quick reference on how to use string methods.

Get Your Code: Click here to download the free sample code that shows you how to work with strings and character data in Python.

Take the Quiz: Test your knowledge with our interactive “Python Strings and Character Data” quiz. You’ll receive a score upon completion to help you track your learning progress:

Interactive Quiz

Python Strings and Character Data

This quiz will test your understanding of Python's string data type and your knowledge about manipulating textual data with string objects. You'll cover the basics of creating strings using literals and the str() function, applying string methods, using operators and built-in functions, and more!

Frequently Asked Questions

Now that you have some experience with Python strings, you can use the questions and answers below to check your understanding and recap what you’ve learned.

These FAQs are related to the most important concepts you’ve covered in this tutorial. Click the *Show/Hide* toggle beside each question to reveal the answer.

What is a string in Python? Show/Hide

What does str() do in Python? Show/Hide

What is {} in a string in Python? Show/Hide

How do you access string elements in Python? Show/Hide

Mark as Completed Share

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Strings and Character Data in Python**

About **Leodanis Pozo Ramos**

Leodanis is a self-taught Python developer, educator, and technical writer with over 10 years of experience.

» More about Leodanis

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

Aldren

Brenda

Bartosz

Dan

Geir Arne

Joanna

John

Philipp

What Do You Think?

Rate this article:

LinkedIn Twitter Bluesky Facebook Email

What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

Commenting Tips: The most useful comments are those written with the goal of learning from or helping out other students. Get tips for asking good questions and get answers to common questions in our support portal.

Looking for a real-time conversation? Visit the Real Python Community Chat or join the next "Office Hours" Live Q&A Session. Happy Pythoning!

Keep Learning

Related Topics: **basics** **python**

Recommended Video Course: Strings and Character Data in Python

Related Tutorials:

- Lists vs Tuples in Python
- Dictionaries in Python
- Python's list Data Type: A Deep Dive With Examples
- Regular Expressions: Regexes in Python (Part 1)
- Variables in Python: Usage and Best Practices