Python Strings and Character Data

Strings and Character Data in Python

by John Sturtz 4 Comments **basics** **python**

Table of Contents

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**

In the tutorial on Basic Data Types in Python, you learned how to define **strings:** objects that contain sequences of character data. Processing character data is integral to programming. It is a rare application that doesn't need to manipulate strings at least to some extent.

Here's what you'll learn in this tutorial: Python provides a rich set of operators, functions, and methods for working with strings. When you are finished with this tutorial, you will know how to access and extract portions of strings, and also be familiar with the methods that are available to manipulate and modify string data.

You will also be introduced to two other Python objects used to represent raw byte data, the `bytes` and `bytearray` types.

Take the Quiz: Test your knowledge with our interactive “Python Strings and Character Data” quiz. Upon completion you will receive a score so you can track your learning progress over time:

Take the Quiz »

String Manipulation

The sections below highlight the operators, methods, and functions that are available for working with strings.

String Operators

You have already seen the operators + and * applied to numeric operands in the tutorial on Operators and Expressions in Python. These two operators can be applied to strings as well.

The + Operator

The + operator concatenates strings. It returns a string consisting of the operands joined together, as shown here:

>>>

```
>>> s = 'foo'
>>> t = 'bar'
>>> u = 'baz'

>>> s + t
'foobar'
>>> s + t + u
'foobarbaz'

>>> print('Go team' + '!!!')
Go team!!!
```

*The * Operator*

The * operator creates multiple copies of a string. If `s` is a string and `n` is an integer, either of the following expressions returns a string consisting of `n` concatenated copies of `s`:

```
s * n
n * s
```

Here are examples of both forms:

>>>

```
>>> s = 'foo.'

>>> s * 4
'foo.foo.foo.foo.'
>>> 4 * s
'foo.foo.foo.foo.'
```

The multiplier operand `n` must be an integer. You'd think it would be required to be a positive integer, but amusingly, it can be zero or negative, in which case the result is an empty string:

>>>

```
>>> 'foo' * -8
''
```

If you were to create a string variable and initialize it to the empty string by assigning it the value `'foo' * -8`, anyone would rightly think you were a bit daft. But it would work.

The `in` Operator

Python also provides a membership operator that can be used with strings. The `in` operator returns `True` if the first operand is contained within the second, and `False` otherwise:

>>>

```
>>> s = 'foo'

>>> s in 'That\'s food for thought.'
True
>>> s in 'That\'s good for now.'
False
```

There is also a `not in` operator, which does the opposite:

>>>

```
>>> 'z' not in 'abc'
True
>>> 'z' not in 'xyz'
False
```

Built-in String Functions

As you saw in the tutorial on [Basic Data Types in Python](), Python provides many functions that are built-in to the interpreter and always available. Here are a few that work with strings:

Function	Description
`chr()`	Converts an integer to a character
`ord()`	Converts a character to an integer
`len()`	Returns the length of a string
`str()`	Returns a string representation of an object

These are explored more fully below.

`ord(c)`

Returns an integer value for the given character.

At the most basic level, computers store all information as numbers. To represent character data, a translation scheme is used which maps each character to its representative number.

The simplest scheme in common use is called ASCII. It covers the common Latin characters you are probably most accustomed to working with. For these characters, `ord(c)` returns the ASCII value for character `c`:

>>>

```
>>> ord('a')
97
>>> ord('#')
35
```

ASCII is fine as far as it goes. But there are many different languages in use in the world and countless symbols and glyphs that appear in digital media. The full set of characters that potentially may need to be represented in computer code far surpasses the ordinary Latin letters, numbers, and symbols you usually see.

Unicode is an ambitious standard that attempts to provide a numeric code for every possible character, in every possible language, on every possible platform. Python 3 supports Unicode extensively, including allowing Unicode characters within strings.

For More Information: See Python’s Unicode Support in the Python documentation.

As long as you stay in the domain of the common characters, there is little practical difference between ASCII and Unicode. But the `ord()` function will return numeric values for Unicode characters as well:

```
>>> ord('€')
8364
>>> ord('∑')
```

```
8721
```

chr(n)

Returns a character value for the given integer.

`chr()` does the reverse of `ord()`. Given a numeric value `n`, `chr(n)` returns a string representing the character that corresponds to `n`:

>>>

```
>>> chr(97)
'a'
>>> chr(35)
'#'
```

`chr()` handles Unicode characters as well:

>>>

```
>>> chr(8364)
'€'
>>> chr(8721)
'∑'
```

len(s)

Returns the length of a string.

With `len()`, you can check Python string length. `len(s)` returns the number of characters in `s`:

>>>

```
>>> s = 'I am a string.'
>>> len(s)
14
```

str(obj)

Returns a string representation of an object.

Virtually any object in Python can be rendered as a string. `str(obj)` returns the string representation of object `obj`:

```
>>> str(49.2)
'49.2'
>>> str(3+4j)
'(3+4j)'
>>> str(3 + 29)
```

```
'32'
>>> str('foo')
```

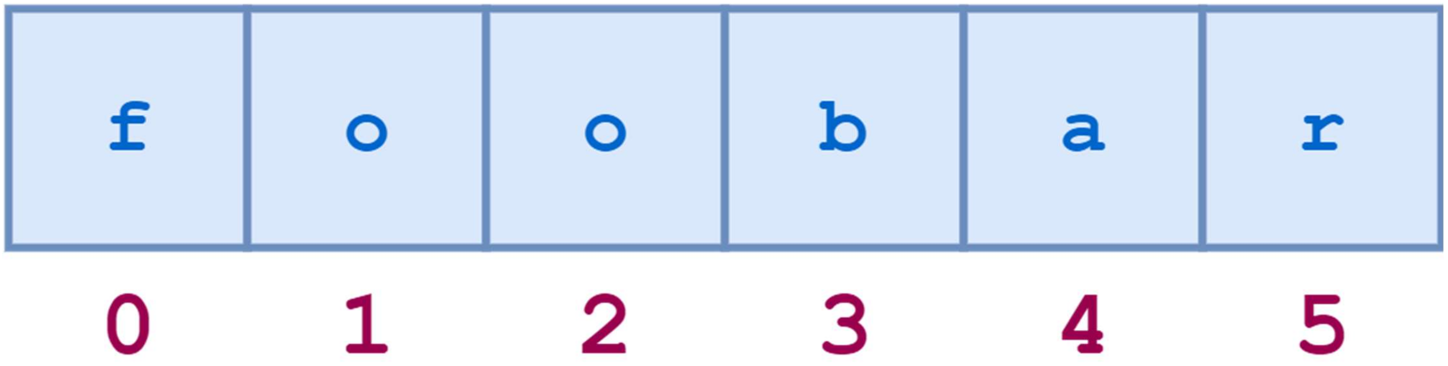
String Indexing

Often in programming languages, individual items in an ordered set of data can be accessed directly using a numeric index or key value. This process is referred to as indexing.

In Python, strings are ordered sequences of character data, and thus can be indexed in this way. Individual characters in a string can be accessed by specifying the string name followed by a number in square brackets (`[]`).

String indexing in Python is zero-based: the first character in the string has index `0`, the next has index `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

The individual characters can be accessed by index as follows:

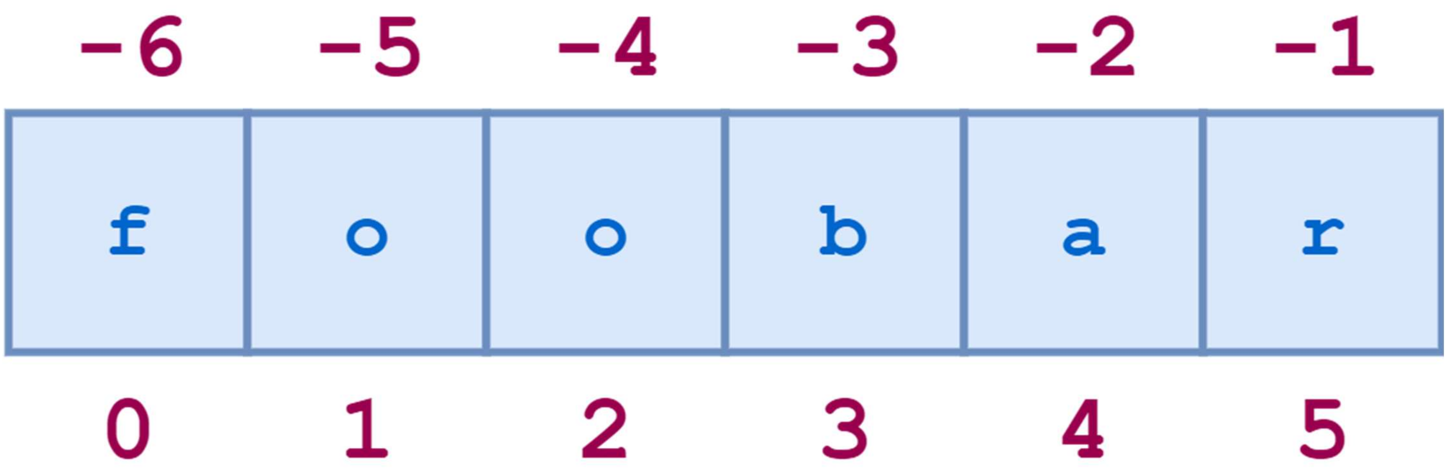
```
>>> s = 'foobar'

>>> s[0]
'f'
>>> s[1]
'o'
>>> s[3]
'b'
>>> len(s)
6
>>> s[len(s)-1]
'r'
```

Attempting to index beyond the end of the string results in an error:

>>>

```
>>> s[6]
Traceback (most recent call last):
  File "<pyshell#17>", line 1, in <module>
    s[6]
IndexError: string index out of range
```

String indices can also be specified with negative numbers, in which case indexing occurs from the end of the string backward: `-1` refers to the last character, `-2` the second-to-last character, and so on. Here is the same diagram showing both the positive and negative indices into the string `'foobar'`:



Positive and Negative String Indices

Here are some examples of negative indexing:

```
>>> s = 'foobar'

>>> s[-1]
'r'
>>> s[-2]
'a'
>>> len(s)
6
>>> s[-len(s)]
'f'
```

Attempting to index with negative numbers beyond the start of the string results in an error:

```
>>> s[-7]
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    s[-7]
IndexError: string index out of range
```

For any non-empty string `s`, `s[len(s)-1]` and `s[-1]` both return the last character. There isn't any index that makes sense for an empty string.

String Slicing

Python also allows a form of indexing syntax that extracts substrings from a string, known as string slicing. If `s` is a string, an expression of the form `s[m:n]` returns the portion of `s` starting with position `m`, and up to but not including position `n`:

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

Remember: String indices are zero-based. The first character in a string has index `0`. This applies to both standard indexing and slicing.

Again, the second index specifies the first character that is not included in the result—the character `'r'` (`s[5]`) in the example above. That may seem slightly unintuitive, but it produces this result which makes sense: the expression `s[m:n]` will return a substring that is `n - m` characters in length, in this case, `5 - 2 = 3`.

If you omit the first index, the slice starts at the beginning of the string. Thus, `s[:m]` and `s[0:m]` are equivalent:

```
>>> s = 'foobar'

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

Similarly, if you omit the second index as in `s[n:]`, the slice extends from the first index through the end of the string. This is a nice, concise alternative to the more cumbersome `s[n:len(s)]`:

```
>>> s = 'foobar'

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

```
'obar'
```

For any string `s` and any integer `n` (`0 ≤ n ≤ len(s)`), `s[:n] + s[n:]` will be equal to `s`:

```
>>> s = 'foobar'


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

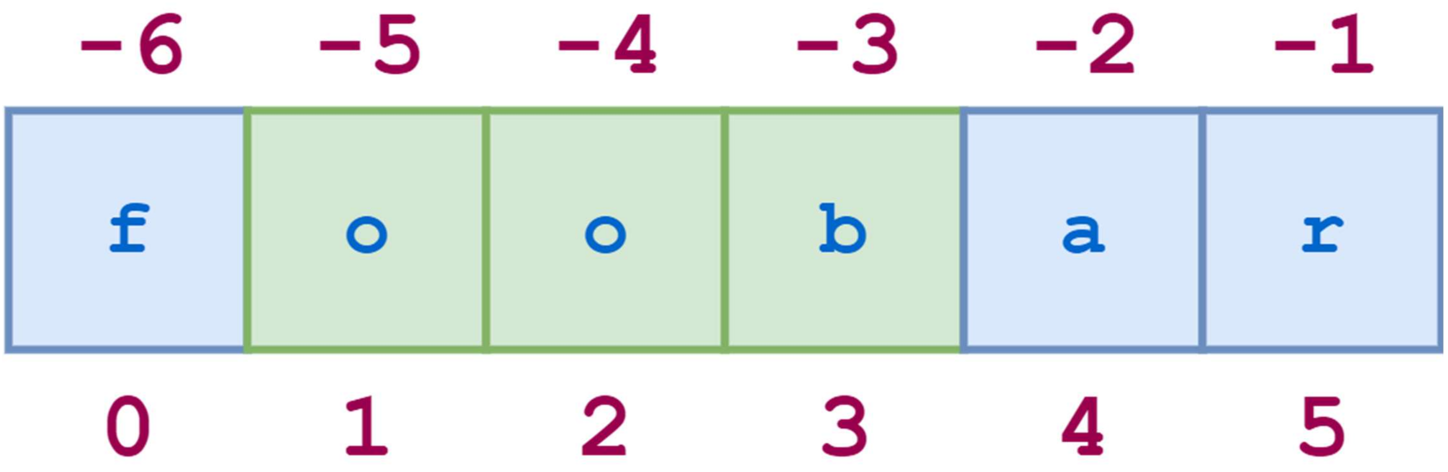
Omitting both indices returns the original string, in its entirety. Literally. It’s not a copy, it’s a reference to the original string:

```
>>> s = 'foobar'
>>> t = s[:]
>>> id(s)
59598496
>>> id(t)
59598496
>>> s is t
True
```

If the first index in a slice is greater than or equal to the second index, Python returns an empty string. This is yet another obfuscated way to generate an empty string, in case you were looking for one:

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

Negative indices can be used with slicing as well. `-1` refers to the last character, `-2` the second-to-last, and so on, just as with simple indexing. 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 is the corresponding Python code:

>>>

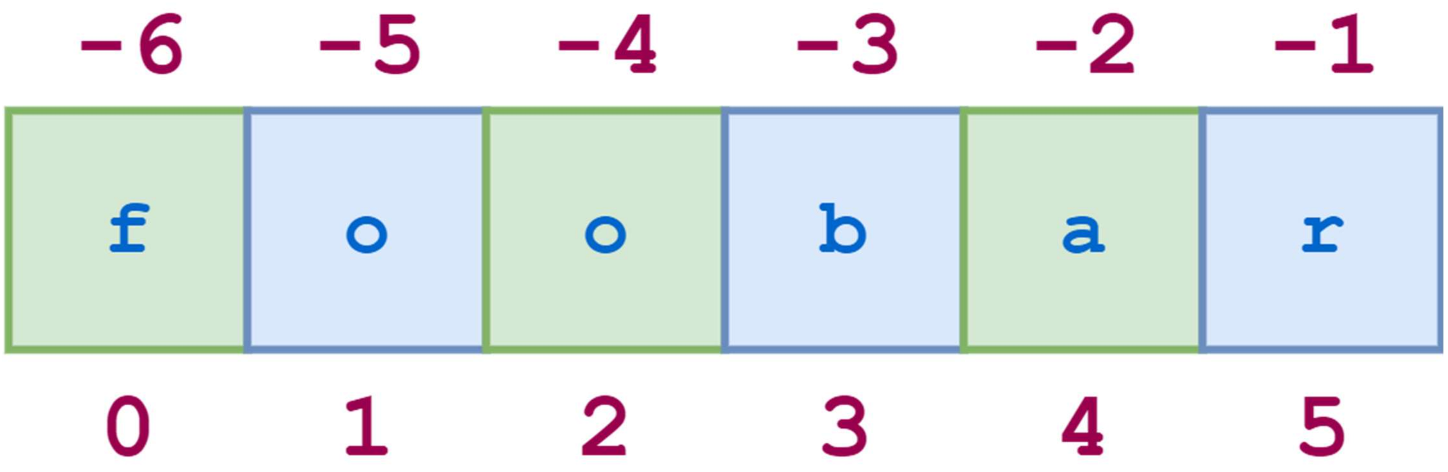
```
>>> s = 'foobar'

>>> s[-5:-2]
'oob'
>>> s[1:4]
'oob'
>>> s[-5:-2] == s[1:4]
True
```

Remove ads

Specifying a Stride in a String Slice

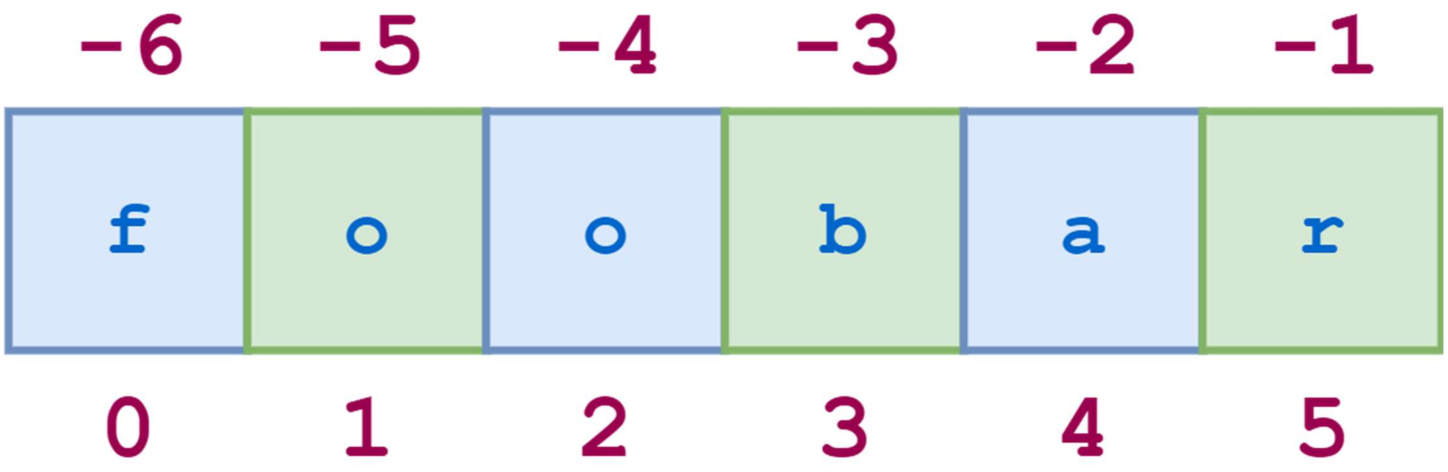
There is one more variant of the slicing syntax to discuss. Adding an additional `:` and a third index designates a stride (also called a step), which indicates how many characters to jump after retrieving each character in the slice.

For example, for the string `'foobar'`, the slice `0:6:2` starts with the first character and ends with the last character (the whole string), and every second character is skipped. This is shown in the following diagram:



String Indexing with Stride

Similarly, `1:6:2` specifies a slice starting with the second character (index `1`) and ending with the last character, and again the stride value `2` causes every other character to be skipped:



Another String Indexing with Stride

The illustrative REPL code is shown here:

```
>>>
```

```
>>> s = 'foobar'

>>> 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:

```
>>>
```

```
>>> s = '12345' * 5
>>> s
'1234512345123451234512345'
>>> s[::5]
'11111'
>>> s[4::5]
'55555'
```

You can specify a negative stride value as well, in which case Python steps backward through the string. In that case, the starting/first index should be greater than the ending/second index:

>>>

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

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

When you are stepping backward, if the first and second indices are omitted, the defaults are reversed in an intuitive way: the first index defaults to the end of the string, and the second index defaults to the beginning. Here is an example:

>>>

```
>>> s = '12345' * 5
>>> s
'1234512345123451234512345'
>>> s[::-5]
'55555'
```

This is a common paradigm for reversing a string:

>>>

```
>>> s = 'If Comrade Napoleon says it, it must be right.'
>>> s[::-1]
'.thgir eb tsum ti ,ti syas noelopaN edarmoC fI'
```

Interpolating Variables Into a String

In Python version 3.6, a new string formatting mechanism was introduced. This feature is formally named the Formatted String Literal, but is more usually referred to by its nickname **f-string**.

The formatting capability provided by f-strings is extensive and won't be covered in full detail here. If you want to learn more, you can check out the Real Python article Python 3's f-Strings:

An Improved String Formatting Syntax (Guide). There is also a tutorial on Formatted Output coming up later in this series that digs deeper into f-strings.

One simple feature of f-strings you can start using right away is variable interpolation. You can specify a variable name directly within an f-string literal, and Python will replace the name with the corresponding value.

For example, suppose you want to display the result of an arithmetic calculation. You can do this with a straightforward `print()` statement, separating numeric values and string literals by commas:

>>>

```
>>> n = 20
>>> m = 25
>>> prod = n * m
>>> print('The product of', n, 'and', m, 'is', prod)
The product of 20 and 25 is 500
```

But this is cumbersome. To accomplish the same thing using an f-string:

- Specify either a lowercase `f` or uppercase `F` directly before the opening quote of the string literal. This tells Python it is an f-string instead of a standard string.
- Specify any variables to be interpolated in curly braces (`{}`).

Recast using an f-string, the above example looks much cleaner:

>>>

```
>>> n = 20
>>> m = 25
>>> prod = n * m
>>> print(f'The product of {n} and {m} is {prod}')
The product of 20 and 25 is 500
```

Any of Python's three quoting mechanisms can be used to define an f-string:

>>>

```
>>> var = 'Bark'

>>> print(f'A dog says {var}!')
A dog says Bark!
>>> print(f"A dog says {var}!")
A dog says Bark!
```

```
>>> print(f'''A dog says {var}!''')
A dog says Bark!
```

Modifying Strings

In a nutshell, you can’t. Strings are one of the data types Python considers immutable, meaning not able to be changed. In fact, all the data types you have seen so far are immutable. (Python does provide data types that are mutable, as you will soon see.)

A statement like this will cause an error:

>>>

```
>>> s = 'foobar'
>>> s[3] = 'x'
Traceback (most recent call last):
  File "<pyshell#40>", line 1, in <module>
    s[3] = 'x'
TypeError: 'str' object does not support item assignment
```
In truth, there really isn’t much need to modify strings. You can usually easily accomplish what you want by generating a copy of the original string that has the desired change in place. There are very many ways to do this in Python. Here is one possibility:

>>>

```
>>> s = s[:3] + 'x' + s[4:]
>>> s
'fooxar'
```
There is also a built-in string method to accomplish this:

>>>

```
>>> s = 'foobar'
>>> s = s.replace('b', 'x')
>>> s
'fooxar'
```
Read on for more information about built-in string methods!

Built-in String Methods

You learned in the tutorial on Variables in Python that Python is a highly object-oriented language. Every item of data in a Python program is an object.

You are also familiar with functions: callable procedures that you can invoke to perform specific tasks.

Methods are similar to functions. A method is a specialized type of callable procedure that is tightly associated with an object. Like a function, a method is called to perform a distinct task, but it is invoked on a specific object and has knowledge of its target object during execution.

The syntax for invoking a method on an object is as follows:

```
obj.foo(<args>)
```

This invokes method `.foo()` on object `obj`. `<args>` specifies the arguments passed to the method (if any).

You will explore much more about defining and calling methods later in the discussion of object-oriented programming. For now, the goal is to present some of the more commonly used built-in methods Python supports for operating on string objects.

In the following method definitions, arguments specified in square brackets (`[]`) are optional.

Case Conversion

Methods in this group perform case conversion on the target string.

`s.capitalize()`

Capitalizes the target string.

`s.capitalize()` returns a copy of `s` with the first character converted to uppercase and all other characters converted to lowercase:

>>>

```
>>> s = 'foO BaR BAZ quX'
>>> s.capitalize()
'Foo bar baz qux'
```

Non-alphabetic characters are unchanged:

>>>

```
>>> s = 'foo123#BAR#.'
>>> s.capitalize()
'Foo123#bar#.'
```

`s.lower()`

Converts alphabetic characters to lowercase.

`s.lower()` returns a copy of `s` with all alphabetic characters converted to lowercase:

>>>

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

`s.swapcase()`

Swaps case of alphabetic characters.
`s.swapcase()` returns a copy of `s` with uppercase alphabetic characters converted to lowercase and vice versa:

>>>

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

`s.title()`

Converts the target string to “title case.”
`s.title()` returns a copy of `s` in which the first letter of each word is converted to uppercase and remaining letters are lowercase:

>>>

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

This method uses a fairly simple algorithm. It does not attempt to distinguish between important and unimportant words, and it does not handle apostrophes, possessives, or acronyms gracefully:

>>>

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

`s.upper()`

Converts alphabetic characters to uppercase.
`s.upper()` returns a copy of `s` with all alphabetic characters converted to uppercase:

>>>

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

Find and Replace

These methods provide various means of searching the target string for a specified substring.

Each method in this group supports optional `<start>` and `<end>` arguments. These are interpreted as for string slicing: the action of the method is restricted to the portion of the target string starting at character position `<start>` and proceeding up to but not including character position `<end>`. If `<start>` is specified but `<end>` is not, the method applies to the portion of the target string from `<start>` through the end of the string.

`s.count(<sub>[, <start>[, <end>]])`

Counts occurrences of a substring in the target string.
`s.count(<sub>)` returns the number of non-overlapping occurrences of substring `<sub>` in `s`:

>>>

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

The count is restricted to the number of occurrences within the substring indicated by `<start>` and `<end>`, if they are specified:

>>>

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

`s.endswith(<suffix>[, <start>[, <end>]])`

Determines whether the target string ends with a given substring.
`s.endswith(<suffix>)` returns `True` if `s` ends with the specified `<suffix>` and `False` otherwise:

>>>

```
>>> 'foobar'.endswith('bar')
True
>>> 'foobar'.endswith('baz')
False
```

The comparison is restricted to the substring indicated by `<start>` and `<end>`, if they are specified:

>>>

```
>>> 'foobar'.endswith('oob', 0, 4)
True
>>> 'foobar'.endswith('oob', 2, 4)
False
```

`s.find(<sub>[, <start>[, <end>]])`

Searches the target string for a given substring.
You can use `.find()` to see if a Python string contains a particular substring. `s.find(<sub>)` returns the lowest index in `s` where substring `<sub>` is found:

```
>>>
```

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

This method returns `-1` if the specified substring is not found:

```
>>>
```

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

The search is restricted to the substring indicated by `<start>` and `<end>`, if they are specified:

```
>>>
```

```
>>> 'foo bar foo baz foo qux'.find('foo', 4)
8
>>> 'foo bar foo baz foo qux'.find('foo', 4, 7)
-1
```

`s.index(<sub>[, <start>[, <end>]])`

Searches the target string for a given substring.
This method is identical to `.find()`, except that it raises an exception if `<sub>` is not found rather than returning `-1`:

```
>>>
```

```
>>> 'foo bar foo baz foo qux'.index('grault')
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    'foo bar foo baz foo qux'.index('grault')
ValueError: substring not found
```

`s.rfind(<sub>[, <start>[, <end>]])`

Searches the target string for a given substring starting at the end.
`s.rfind(<sub>)` returns the highest index in `s` where substring `<sub>` is found:

```
>>>
```

```
>>> 'foo bar foo baz foo qux'.rfind('foo')
```

```
16
```
As with `.find()`, if the substring is not found, `-1` is returned:

```
>>>
```

```
>>> 'foo bar foo baz foo qux'.rfind('grault')
-1
```
The search is restricted to the substring indicated by `<start>` and `<end>`, if they are specified:

```
>>>
```

```
>>> 'foo bar foo baz foo qux'.rfind('foo', 0, 14)
8
>>> 'foo bar foo baz foo qux'.rfind('foo', 10, 14)
-1
```

`s.rindex(<sub>[, <start>[, <end>]])`

Searches the target string for a given substring starting at the end.

This method is identical to `.rfind()`, except that it raises an exception if `<sub>` is not found rather than returning `-1`:

```
>>>
```

```
>>> 'foo bar foo baz foo qux'.rindex('grault')
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    'foo bar foo baz foo qux'.rindex('grault')
ValueError: substring not found
```

`s.startswith(<prefix>[, <start>[, <end>]])`

Determines whether the target string starts with a given substring.

When you use the Python `.startswith()` method, `s.startswith(<suffix>)` returns `True` if `s` starts with the specified `<suffix>` and `False` otherwise:

```
>>>
```

```
>>> 'foobar'.startswith('foo')
True
>>> 'foobar'.startswith('bar')
False
```
The comparison is restricted to the substring indicated by `<start>` and `<end>`, if they are specified:

>>>

```
>>> 'foobar'.startswith('bar', 3)
True
>>> 'foobar'.startswith('bar', 3, 2)
False
```

Character Classification

Methods in this group classify a string based on the characters it contains.

`s.isalnum()`

Determines whether the target string consists of alphanumeric characters.

`s.isalnum()` returns `True` if `s` is nonempty and all its characters are alphanumeric (either a letter or a number), and `False` otherwise:

>>>

```
>>> 'abc123'.isalnum()
True
>>> 'abc$123'.isalnum()
False
>>> ''.isalnum()
False
```

`s.isalpha()`

Determines whether the target string consists of alphabetic characters.

`s.isalpha()` returns `True` if `s` is nonempty and all its characters are alphabetic, and `False` otherwise:

>>>

```
>>> 'ABCabc'.isalpha()
True
>>> 'abc123'.isalpha()
False
```

`s.isdigit()`

Determines whether the target string consists of digit characters.

You can use the `.isdigit()` Python method to check if your string is made of only digits. `s.digit()` returns `True` if `s` is nonempty and all its characters are numeric digits, and `False` otherwise:

>>>

```
>>> '123'.isdigit()
True
>>> '123abc'.isdigit()
False
```

s.isidentifier()

Determines whether the target string is a valid Python identifier.
`s.isidentifier()` returns `True` if `s` is a valid Python identifier according to the language definition, and `False` otherwise:

>>>

```
>>> 'foo32'.isidentifier()
True
>>> '32foo'.isidentifier()
False
>>> 'foo$32'.isidentifier()
False
```

Note: `.isidentifier()` will return `True` for a string that matches a Python keyword even though that would not actually be a valid identifier:

>>>

```
>>> '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:

>>>

```
>>> from keyword import iskeyword
>>> iskeyword('and')
True
```

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

See Python Modules and Packages—An Introduction to read more about Python modules.

s.islower()

Determines whether the target string's alphabetic characters are lowercase.

`s.islower()` returns `True` if `s` is nonempty and all the alphabetic characters it contains are lowercase, and `False` otherwise. Non-alphabetic characters are ignored:

>>>

```
>>> 'abc'.islower()
True
>>> 'abc1$d'.islower()
True
>>> 'Abc1$D'.islower()
False
```

s.isprintable()

Determines whether the target string consists entirely of printable characters.
`s.isprintable()` returns `True` if `s` is empty or all the alphabetic characters it contains are printable. It returns `False` if `s` contains at least one non-printable character. Non-alphabetic characters are ignored:

>>>

```
>>> 'a\tb'.isprintable()
False
>>> 'a b'.isprintable()
True
>>> ''.isprintable()
True
>>> 'a\nb'.isprintable()
False
```

Note: This is the only `.isxxxx()` method that returns `True` if `s` is an empty string. All the others return `False` for an empty string.

s.isspace()

Determines whether the target string consists of whitespace characters.
`s.isspace()` returns `True` if `s` is nonempty and all characters are whitespace characters, and `False` otherwise.

The most commonly encountered whitespace characters are space `' '`, tab `'\t'`, and newline `'\n'`:

>>>

```
>>> ' \t \n '.isspace()
```

```
True
>>> '   a   '.isspace()
False
```
However, there are a few other ASCII characters that qualify as whitespace, and if you account for Unicode characters, there are quite a few beyond that:

>>>

```
>>> '\f\u2005\r'.isspace()
True
```
(`'\f'` and `'\r'` are the escape sequences for the ASCII Form Feed and Carriage Return characters; `'\u2005'` is the escape sequence for the Unicode Four-Per-Em Space.)

`s.istitle()`

Determines whether the target string is title cased.

`s.istitle()` returns `True` if `s` is nonempty, the first alphabetic character of each word is uppercase, and all other alphabetic characters in each word are lowercase. It returns `False` otherwise:

>>>

```
>>> 'This Is A Title'.istitle()
True
>>> 'This is a title'.istitle()
False
>>> 'Give Me The #$#@ Ball!'.istitle()
True
```
Note: Here is how the Python documentation describes `.istitle()`, in case you find this more intuitive: “Uppercase characters may only follow uncased characters and lowercase characters only cased ones.”

`s.isupper()`

Determines whether the target string’s alphabetic characters are uppercase.

`s.isupper()` returns `True` if `s` is nonempty and all the alphabetic characters it contains are uppercase, and `False` otherwise. Non-alphabetic characters are ignored:

>>>

```
>>> 'ABC'.isupper()
True
>>> 'ABC1$D'.isupper()
True
```

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

String Formatting

Methods in this group modify or enhance the format of a string.

`s.center(<width>[, <fill>])`

Centers a string in a field.

`s.center(<width>)` returns a string consisting of `s` centered in a field of width `<width>`. By default, padding consists of the ASCII space character:

```
>>>
```

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

If the optional `<fill>` argument is specified, it is used as the padding character:

```
>>>
```

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

If `s` is already at least as long as `<width>`, it is returned unchanged:

```
>>>
```

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

`s.expandtabs(tabsize=8)`

Expands tabs in a string.

`s.expandtabs()` replaces each tab character (`'\t'`) with spaces. By default, spaces are filled in assuming a tab stop at every eighth column:

```
>>>
```

```
>>> 'a\tb\tc'.expandtabs()
'a       b       c'
>>> 'aaa\tbbb\tc'.expandtabs()
'aaa     bbb     c'
```

`tabsize` is an optional keyword parameter specifying alternate tab stop columns:

```
>>>
```

```
>>> 'a\tb\tc'.expandtabs(4)
```

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

s.ljust(<width>[, <fill>])

Left-justifies a string in field.

`s.ljust(<width>)` returns a string consisting of `s` left-justified in a field of width `<width>`. By default, padding consists of the ASCII space character:

>>>

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

If the optional `<fill>` argument is specified, it is used as the padding character:

>>>

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

If `s` is already at least as long as `<width>`, it is returned unchanged:

>>>

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

s.lstrip([<chars>])

Trims leading characters from a string.

`s.lstrip()` returns a copy of `s` with any whitespace characters removed from the left end:

>>>

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

If the optional `<chars>` argument is specified, it is a string that specifies the set of characters to be removed:

>>>

```
>>> 'http://www.realpython.com'.lstrip('/:pth')
'www.realpython.com'
```

`s.replace(<old>, <new>[, <count>])`

Replaces occurrences of a substring within a string.
In Python, to remove a character from a string, you can use the Python string `.replace()` method. `s.replace(<old>, <new>)` returns a copy of `s` with all occurrences of substring `<old>` replaced by `<new>`:

>>>

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

If the optional `<count>` argument is specified, a maximum of `<count>` replacements are performed, starting at the left end of `s`:

>>>

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

`s.rjust(<width>[, <fill>])`

Right-justifies a string in a field.
`s.rjust(<width>)` returns a string consisting of `s` right-justified in a field of width `<width>`. By default, padding consists of the ASCII space character:

>>>

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

If the optional `<fill>` argument is specified, it is used as the padding character:

>>>

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

If `s` is already at least as long as `<width>`, it is returned unchanged:

>>>

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

`s.rstrip([<chars>])`

Trims trailing characters from a string.

`s.rstrip()` returns a copy of `s` with any whitespace characters removed from the right end:

>>>

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

If the optional `<chars>` argument is specified, it is a string that specifies the set of characters to be removed:

>>>

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

`s.strip([<chars>])`

Strips characters from the left and right ends of a string.

`s.strip()` is essentially equivalent to invoking `s.lstrip()` and `s.rstrip()` in succession. Without the `<chars>` argument, it removes leading and trailing whitespace:

>>>

```
>>> s = '   foo bar baz\t\t\t'
>>> s = s.lstrip()
>>> s = s.rstrip()
>>> s
'foo bar baz'
```

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

>>>

```
>>> 'www.realpython.com'.strip('w.moc')
'realpython'
```

Note: When the return value of a string method is another string, as is often the case, methods can be invoked in succession by chaining the calls:

>>>

```
>>> '   foo bar baz\t\t\t'.lstrip().rstrip()
'foo bar baz'
>>> '   foo bar baz\t\t\t'.strip()
```

```
'foo bar baz'

>>> 'www.realpython.com'.lstrip('w.moc').rstrip('w.moc')
'realpython'
>>> 'www.realpython.com'.strip('w.moc')
'realpython'
```

`s.zfill(<width>)`

Pads a string on the left with zeros.
`s.zfill(<width>)` returns a copy of `s` left-padded with `'0'` characters to the specified `<width>`:

>>>

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

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

>>>

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

If `s` is already at least as long as `<width>`, it is returned unchanged:

>>>

```
>>> '-42'.zfill(3)
'-42'
```

`.zfill()` is most useful for string representations of numbers, but Python will still happily zero-pad a string that isn't:

>>>

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

Converting Between Strings and Lists

Methods in this group convert between a string and some composite data type by either pasting objects together to make a string, or by breaking a string up into pieces.

These methods operate on or return **iterables**, the general Python term for a sequential collection of objects. You will explore the inner workings of iterables in much more detail in the upcoming tutorial on definite iteration.

Many of these methods return either a list or a tuple. These are two similar composite data types that are prototypical examples of iterables in Python. They are covered in the next tutorial, so you're about to learn about them soon! Until then, simply think of them as sequences of values. A list is enclosed in square brackets (`[]`), and a tuple is enclosed in parentheses (`()`).

With that introduction, let's take a look at this last group of string methods.

`s.join(<iterable>)`

Concatenates strings from an iterable.
`s.join(<iterable>)` returns the string that results from concatenating the objects in `<iterable>` separated by `s`.

Note that `.join()` is invoked on `s`, the separator string. `<iterable>` must be a sequence of string objects as well.

Some sample code should help clarify. In the following example, the separator `s` is the string `', '`, and `<iterable>` is a list of string values:

```
>>>
```

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

The result is a single string consisting of the list objects separated by commas.

In the next example, `<iterable>` is specified as a single string value. When a string value is used as an iterable, it is interpreted as a list of the string's individual characters:

```
>>>
```

```
>>> list('corge')
['c', 'o', 'r', 'g', 'e']

>>> ':'.join('corge')
'c:o:r:g:e'
```

Thus, the result of `':'.join('corge')` is a string consisting of each character in `'corge'` separated by `':'`.

This example fails because one of the objects in `<iterable>` is not a string:

```
>>>
```

```
>>> '---'.join(['foo', 23, 'bar'])
```

```
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    '---'.join(['foo', 23, 'bar'])
TypeError: sequence item 1: expected str instance, int found
```
That can be remedied, though:

```
>>>
```

```
>>> '---'.join(['foo', str(23), 'bar'])
'foo---23---bar'
```
As you will soon see, many composite objects in Python can be construed as iterables, and `.join()` is especially useful for creating strings from them.

`s.partition(<sep>)`

Divides a string based on a separator.

`s.partition(<sep>)` splits `s` at the first occurrence of string `<sep>`. The return value is a three-part tuple consisting of:

- The portion of `s` preceding `<sep>`
- `<sep>` itself
- The portion of `s` following `<sep>`

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

```
>>>
```

```
>>> 'foo.bar'.partition('.')
('foo', '.', 'bar')
>>> 'foo@@bar@@baz'.partition('@@')
('foo', '@@', 'bar@@baz')
```
If `<sep>` is not found in `s`, the returned tuple contains `s` followed by two empty strings:

```
>>>
```

```
>>> 'foo.bar'.partition('@@')
('foo.bar', '', '')
```
Remember: Lists and tuples are covered in the next tutorial.

`s.rpartition(<sep>)`

Divides a string based on a separator.

`s.rpartition(<sep>)` functions exactly like `s.partition(<sep>)`, except that `s` is split at the last occurrence of `<sep>` instead of the first occurrence:

>>>

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

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

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

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

>>>

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

If `<sep>` is specified, it is used as the delimiter for splitting:

>>>

```
>>> 'foo.bar.baz.qux'.rsplit(sep='.')
['foo', 'bar', 'baz', 'qux']
```

(If `<sep>` is specified with a value of `None`, the string is split delimited by whitespace, just as though `<sep>` had not been specified at all.)

When `<sep>` is explicitly given as a delimiter, consecutive delimiters in `s` are assumed to delimit empty strings, which will be returned:

>>>

```
>>> 'foo...bar'.rsplit(sep='.')
['foo', '', '', 'bar']
```

This is not the case when `<sep>` is omitted, however. In that case, consecutive whitespace characters are combined into a single delimiter, and the resulting list will never contain empty strings:

>>>

```
>>> 'foo\t\t\tbar'.rsplit()
['foo', 'bar']
```

If the optional keyword parameter `<maxsplit>` is specified, a maximum of that many splits are performed, starting from the right end of `s`:

>>>

```
>>> 'www.realpython.com'.rsplit(sep='.', maxsplit=1)
['www.realpython', 'com']
```

The default value for `<maxsplit>` is `-1`, which means all possible splits should be performed—the same as if `<maxsplit>` is omitted entirely:

>>>

```
>>> 'www.realpython.com'.rsplit(sep='.', maxsplit=-1)
['www', 'realpython', 'com']
>>> 'www.realpython.com'.rsplit(sep='.')
['www', 'realpython', 'com']
```

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

Splits a string into a list of substrings.

`s.split()` behaves exactly like `s.rsplit()`, except that if `<maxsplit>` is specified, splits are counted from the left end of `s` rather than the right end:

>>>

```
>>> 'www.realpython.com'.split('.', maxsplit=1)
['www', 'realpython.com']
>>> 'www.realpython.com'.rsplit('.', maxsplit=1)
['www.realpython', 'com']
```

If `<maxsplit>` is not specified, `.split()` and `.rsplit()` are indistinguishable.

`s.splitlines([<keepends>])`

Breaks a string at line boundaries.

`s.splitlines()` splits `s` up into lines and returns them in a list. Any of the following characters or character sequences is considered to constitute a line boundary:

Escape Sequence	Character
`\n`	Newline
`\r`	Carriage Return
`\r\n`	Carriage Return + Line Feed
`\v` or `\x0b`	Line Tabulation

Escape Sequence	Character
`\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 is an example using several different line separators:

>>>

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

If consecutive line boundary characters are present in the string, they are assumed to delimit blank lines, which will appear in the result list:

>>>

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

If the optional `<keepends>` argument is specified and is truthy, then the lines boundaries are retained in the result strings:

>>>

```
>>> 'foo\nbar\nbaz\nqux'.splitlines(True)
['foo\n', 'bar\n', 'baz\n', 'qux']
>>> 'foo\nbar\nbaz\nqux'.splitlines(1)
['foo\n', 'bar\n', 'baz\n', 'qux']
```

`bytes` Objects

The `bytes` object is one of the core built-in types for manipulating binary data. A `bytes` object is an immutable sequence of single byte values. Each element in a `bytes` object is a small integer in the range `0` to `255`.

Defining a Literal `bytes` Object

A `bytes` literal is defined in the same way as a string literal with the addition of a `'b'` prefix:

>>>

```
>>> b = b'foo bar baz'
>>> b
b'foo bar baz'
>>> type(b)
<class 'bytes'>
```

As with strings, you can use any of the single, double, or triple quoting mechanisms:

>>>

```
>>> b'Contains embedded "double" quotes'
b'Contains embedded "double" quotes'

>>> b"Contains embedded 'single' quotes"
b"Contains embedded 'single' quotes"

>>> b'''Contains embedded "double" and 'single' quotes'''
b'Contains embedded "double" and \'single\' quotes'

>>> b"""Contains embedded "double" and 'single' quotes"""
b'Contains embedded "double" and \'single\' quotes'
```

Only ASCII characters are allowed in a `bytes` literal. Any character value greater than `127` must be specified using an appropriate escape sequence:

>>>

```
>>> b = b'foo\xddbar'
>>> b
b'foo\xddbar'
>>> b[3]
221
>>> int(0xdd)
221
```

The `'r'` prefix may be used on a `bytes` literal to disable processing of escape sequences, as with strings:

>>>

```
>>> b = rb'foo\xddbar'
>>> b
b'foo\\xddbar'
>>> b[3]
```

```
92
>>> chr(92)
'\\'
```

Defining a `bytes` Object With the Built-in `bytes()` Function

The `bytes()` function also creates a `bytes` object. What sort of `bytes` object gets returned depends on the argument(s) passed to the function. The possible forms are shown below.

`bytes(<s>, <encoding>)`

Creates a `bytes` object from a string.

`bytes(<s>, <encoding>)` converts string `<s>` to a `bytes` object, using `str.encode()` according to the specified `<encoding>`:

>>>

```
>>> b = bytes('foo.bar', 'utf8')
>>> b
b'foo.bar'
>>> type(b)
<class 'bytes'>
```

Technical Note: In this form of the `bytes()` function, the `<encoding>` argument is required. "Encoding" refers to the manner in which characters are translated to integer values. A value of `"utf8"` indicates Unicode Transformation Format **UTF-8**, which is an encoding that can handle every possible Unicode character. UTF-8 can also be indicated by specifying `"UTF8"`, `"utf-8"`, or `"UTF-8"` for `<encoding>`.

See the Unicode documentation for more information. As long as you are dealing with common Latin-based characters, UTF-8 will serve you fine.

`bytes(<size>)`

Creates a `bytes` object consisting of null (`0x00`) bytes.

`bytes(<size>)` defines a `bytes` object of the specified `<size>`, which must be a positive integer. The resulting `bytes` object is initialized to null (`0x00`) bytes:

>>>

```
>>> b = bytes(8)
>>> b
b'\x00\x00\x00\x00\x00\x00\x00\x00'
>>> type(b)
<class 'bytes'>
```

`bytes(<iterable>)`

Creates a `bytes` object from an iterable.

`bytes(<iterable>)` defines a `bytes` object from the sequence of integers generated by `<iterable>`. `<iterable>` must be an iterable that generates a sequence of integers `n` in the range `0 ≤ n ≤ 255`:

>>>

```
>>> b = bytes([100, 102, 104, 106, 108])
>>> b
b'dfhjl'
>>> type(b)
<class 'bytes'>
>>> b[2]
104
```

Remove ads

Operations on `bytes` Objects

Like strings, `bytes` objects support the common sequence operations:

- The `in` and `not in` operators:

  >>>

  ```
  >>> b = b'abcde'

  >>> b'cd' in b
  True
  >>> b'foo' not in b
  True
  ```
- The concatenation (`+`) and replication (`*`) operators:

  >>>

  ```
  >>> b = b'abcde'

  >>> b + b'fghi'
  b'abcdefghi'
  >>> b * 3
  b'abcdeabcdeabcde'
  ```
- Indexing and slicing:

```
>>>
```

```
>>> b = b'abcde'

>>> b[2]
99
>>> b[1:3]
b'bc'
```

- Built-in functions:

```
>>>
```

```
>>> len(b)
5
>>> min(b)
97
>>> max(b)
101
```

Many of the methods defined for string objects are valid for `bytes` objects as well:

```
>>>
```

```
>>> b = b'foo,bar,foo,baz,foo,qux'

>>> b.count(b'foo')
3

>>> b.endswith(b'qux')
True

>>> b.find(b'baz')
12

>>> b.split(sep=b',')
[b'foo', b'bar', b'foo', b'baz', b'foo', b'qux']

>>> b.center(30, b'-')
b'---foo,bar,foo,baz,foo,qux----'
```

Notice, however, that when these operators and methods are invoked on a `bytes` object, the operand and arguments must be `bytes` objects as well:

```
>>>
```

```
>>> b = b'foo.bar'

>>> b + '.baz'
Traceback (most recent call last):
  File "<pyshell#72>", line 1, in <module>
    b + '.baz'
TypeError: can't concat bytes to str
>>> b + b'.baz'
b'foo.bar.baz'

>>> b.split(sep='.')
Traceback (most recent call last):
  File "<pyshell#74>", line 1, in <module>
    b.split(sep='.')
TypeError: a bytes-like object is required, not 'str'
>>> b.split(sep=b'.')
[b'foo', b'bar']
```

Although a `bytes` object definition and representation is based on ASCII text, it actually behaves like an immutable sequence of small integers in the range `0` to `255`, inclusive. That is why a single element from a `bytes` object is displayed as an integer:

```
>>>
```

```
>>> b = b'foo\xddbar'
>>> b[3]
221
>>> hex(b[3])
'0xdd'
>>> min(b)
97
>>> max(b)
221
```

A slice is displayed as a `bytes` object though, even if it is only one byte long:

```
>>>
```

```
>>> b[2:3]
b'c'
```

You can convert a `bytes` object into a list of integers with the built-in `list()` function:

>>>

```
>>> list(b)
[97, 98, 99, 100, 101]
```

Hexadecimal numbers are often used to specify binary data because two hexadecimal digits correspond directly to a single byte. The `bytes` class supports two additional methods that facilitate conversion to and from a string of hexadecimal digits.

`bytes.fromhex(<s>)`

Returns a `bytes` object constructed from a string of hexadecimal values.

`bytes.fromhex(<s>)` returns the `bytes` object that results from converting each pair of hexadecimal digits in `<s>` to the corresponding byte value. The hexadecimal digit pairs in `<s>` may optionally be separated by whitespace, which is ignored:

>>>

```
>>> b = bytes.fromhex(' aa 68 4682cc ')
>>> b
b'\xaahF\x82\xcc'
>>> list(b)
[170, 104, 70, 130, 204]
```

Note: This method is a class method, not an object method. It is bound to the `bytes` class, not a `bytes` object. You will delve much more into the distinction between classes, objects, and their respective methods in the upcoming tutorials on object-oriented programming. For now, just observe that this method is invoked on the `bytes` class, not on object `b`.

`b.hex()`

Returns a string of hexadecimal value from a `bytes` object.

`b.hex()` returns the result of converting `bytes` object `b` into a string of hexadecimal digit pairs. That is, it does the reverse of `.fromhex()`:

>>>

```
>>> b = bytes.fromhex(' aa 68 4682cc ')
>>> b
b'\xaahF\x82\xcc'

>>> b.hex()
'aa684682cc'
>>> type(b.hex())
<class 'str'>
```

Note: As opposed to `.fromhex()`, `.hex()` is an object method, not a class method. Thus, it is invoked on an object of the `bytes` class, not on the class itself.

`bytearray` Objects

Python supports another binary sequence type called the `bytearray`. `bytearray` objects are very like `bytes` objects, despite some differences:

- There is no dedicated syntax built into Python for defining a `bytearray` literal, like the `'b'` prefix that may be used to define a `bytes` object. A `bytearray` object is always created using the `bytearray()` built-in function:

  >>>

  ```
  >>> ba = bytearray('foo.bar.baz', 'UTF-8')
  >>> ba
  bytearray(b'foo.bar.baz')

  >>> bytearray(6)
  bytearray(b'\x00\x00\x00\x00\x00\x00')

  >>> bytearray([100, 102, 104, 106, 108])
  bytearray(b'dfhjl')
  ```
- `bytearray` objects are mutable. You can modify the contents of a `bytearray` object using indexing and slicing:

  >>>

  ```
  >>> ba = bytearray('foo.bar.baz', 'UTF-8')
  >>> ba
  bytearray(b'foo.bar.baz')

  >>> ba[5] = 0xee
  >>> ba
  bytearray(b'foo.b\xeer.baz')

  >>> ba[8:11] = b'qux'
  >>> ba
  bytearray(b'foo.b\xeer.qux')
  ```

A `bytearray` object may be constructed directly from a `bytes` object as well:

>>>

```
>>> ba = bytearray(b'foo')
>>> ba
bytearray(b'foo')
```

Conclusion

This tutorial provided an in-depth look at the many different mechanisms Python provides for **string** handling, including string operators, built-in functions, indexing, slicing, and built-in methods. You also were introduced to the `bytes` and `bytearray` types.

These types are the first types you have examined that are composite—built from a collection of smaller parts. Python provides several composite built-in types. In the next tutorial, you will explore two of the most frequently used: **lists** and **tuples**.

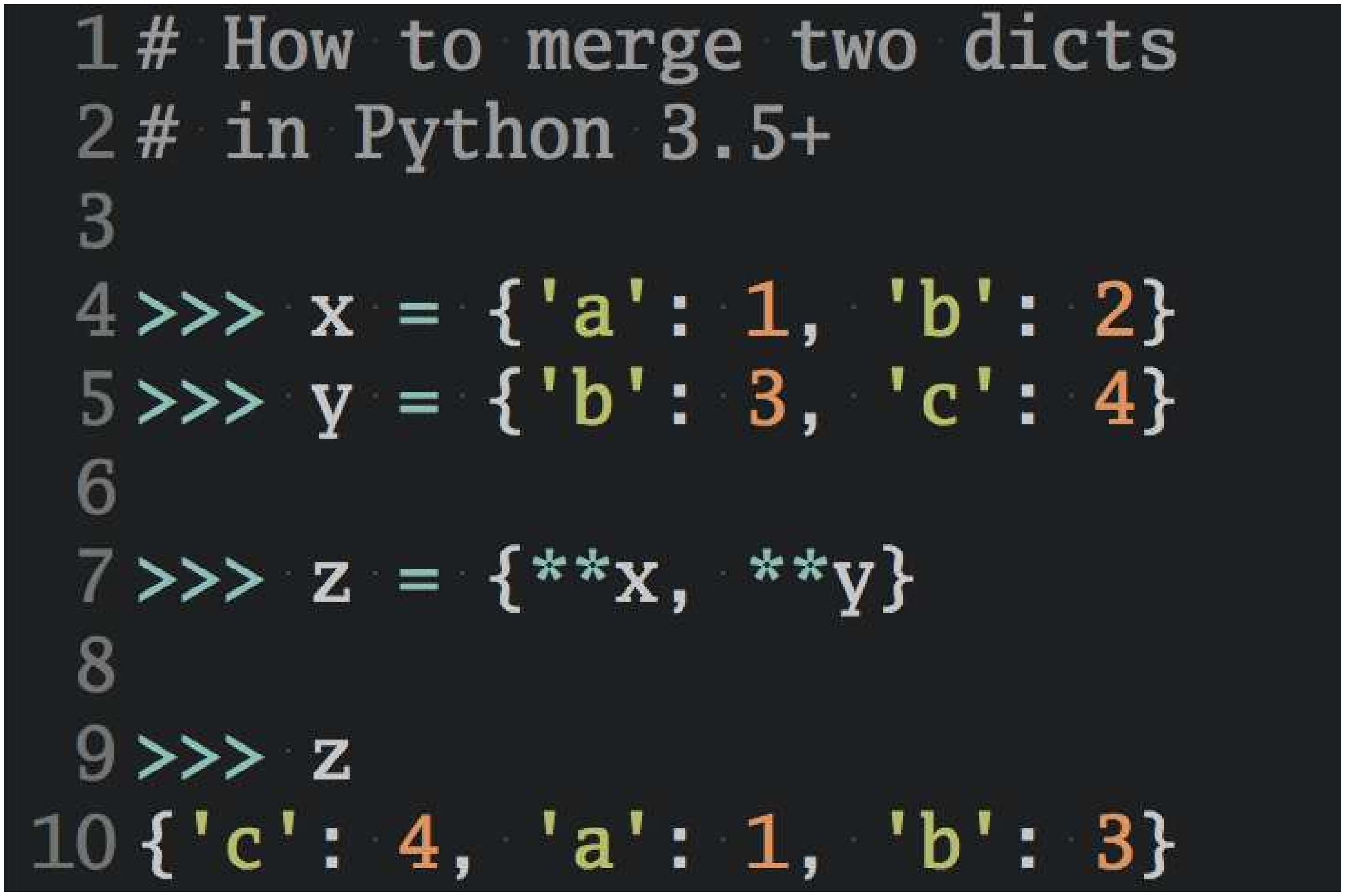
Take the Quiz: Test your knowledge with our interactive "Python Strings and Character Data" quiz. Upon completion you will receive a score so you can track your learning progress over time:

Take the Quiz »

« Operators and Expressions in Python

Strings in Python

Lists and Tuples in Python »



```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```

Send Me Python Tricks »

About **John Sturtz**

John is an avid Pythonista and a member of the Real Python tutorial team.

» More about John

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:

https://realpython.com/python-strings/#.W0P9uWMUJ7U.twitter