Chapter 5. Text Strings

I always liked strange characters.

—Tim Burton

Computer books often give the impression that programming is all about math. Actually, most programmers work with *strings* of text more often than numbers. Logical (and creative!) thinking is often more important than math skills.

Strings are our first example of a Python *sequence*. In this case, they're a sequence of *characters*. But what's a character? It's the smallest unit in a writing system, and includes letters, digits, symbols, punctuation, and even white space or directives like linefeeds. A character is defined by its meaning (how it's used), not how it looks. It can have more than one visual representation (in different *fonts*), and more than one character can have the same appearance (such as the visual `H`, which means the `H` sound in the Latin alphabet but the Latin `N` sound in Cyrillic).

This chapter concentrates on how to make and format simple text strings, using ASCII (basic character set) examples. Two important text topics are deferred to Chapter 12: *Unicode* characters (like the `H` and `N` issue I just mentioned) and *regular expressions* (pattern matching).

Unlike other languages, strings in Python are *immutable*. You can't change a string in place, but you can copy parts of strings to another string to get the same effect. We look at how to do this shortly.

Create with Quotes

You make a Python string by enclosing characters in matching single or double quotes:

```
>>> 'Snap'
```

```
'Snap'
>>> "Crackle"
'Crackle'
```

The interactive interpreter echoes strings with a single quote, but all are treated exactly the same by Python.

NOTE

Python has a few special types of strings, indicated by a letter before the first quote. `f` or `F` starts an *f string*, used for formatting, and described near the end of this chapter. `r` or `R` starts a *raw string*, used to prevent *escape sequences* in the string (see "Escape with \" and Chapter 12 for its use in string pattern matching). Then, there's the combination `fr` (or `FR`, `Fr`, or `fR`) that starts a raw f-string. A `u` starts a Unicode string, which is the same as a plain string. And a `b` starts a value of type `bytes` (Chapter 12). Unless I mention one of these special types, I'm always talking about plain old Python Unicode text strings.

Why have two kinds of quote characters? The main purpose is to create strings containing quote characters. You can have single quotes inside double-quoted strings, or double quotes inside single-quoted strings:

```
>>> "'Nay!' said the naysayer. 'Neigh?' said the horse."
"'Nay!' said the naysayer. 'Neigh?' said the horse."
>>> 'The rare double quote in captivity: ".'
'The rare double quote in captivity: ".'
>>> 'A "two by four" is actually 1 ½" × 3 ½".'
'A "two by four" is actually 1 ½" × 3 ½".'
>>> "'There's the man that shot my paw!' cried the limping hound."
"'There's the man that shot my paw!' cried the limping hound."
```

You can also use three single quotes (`'''`) or three double quotes (`"""`):

```
>>> '''Boom!'''
'Boom'
>>> """Eek!"""
'Eek!'
```

Triple quotes aren't very useful for short strings like these. Their most common use is to create *multiline strings*, like this classic poem from Edward Lear:

```
>>> poem =  '''There was a Young Lady of Norway,
... Who casually sat in a doorway;
... When the door squeezed her flat,
... She exclaimed, "What of that?"
... This courageous Young Lady of Norway.'''
>>>
```

(This was entered in the interactive interpreter, which prompted us with `>>>` for the first line and continuation prompts `...` until we entered the final triple quotes and went to the next line.)

If you tried to create that poem without triple quotes, Python would make a fuss when you went to the second line:

```
>>> poem = 'There was a young lady of Norway,
  File "<stdin>", line 1
    poem = 'There was a young lady of Norway,
                                             ^
SyntaxError: EOL while scanning string literal
>>>
```

If you have multiple lines within triple quotes, the line ending characters will be preserved in the string. If you have leading or trailing spaces, they'll also be kept:

```
>>> poem2 = '''I do not like thee, Doctor Fell.
...     The reason why, I cannot tell.
...     But this I know, and know full well:
...     I do not like thee, Doctor Fell.
... '''
>>> print(poem2)
I do not like thee, Doctor Fell.
    The reason why, I cannot tell.
    But this I know, and know full well:
    I do not like thee, Doctor Fell.
```

```
>>>
```

By the way, there's a difference between the output of `print()` and the automatic echoing done by the interactive interpreter:

```
>>> poem2
'I do not like thee, Doctor Fell.\n    The reason why, I cannot tell.\n    But
this I know, and know full well:\n    I do not like thee, Doctor Fell.\n'
```

`print()` strips quotes from strings and prints their contents. It's meant for human output. It helpfully adds a space between each of the things it prints, and a newline at the end:

```
>>> print('Give', "us", '''some''', """space""")
Give us some space
```

If you don't want the space or newline, Chapter 14 explains how to avoid them.

The interactive interpreter prints the string with individual quotes and *escape characters* such as `\n`, which are explained in "Escape with \".

```
>>> """'Guten Morgen, mein Herr!'
... said mad king Ludwig to his wig."""
"'Guten Morgen, mein Herr!'\nsaid mad king Ludwig to his wig."
```

Finally, there is the *empty string*, which has no characters at all but is perfectly valid. You can create an empty string with any of the aforementioned quotes:

```
>>> ''
''
>>> ""
''
>>> ''''''
''
>>> """"""
```

```
''
>>>
```

Create with str()

You can make a string from another data type by using the `str()` function:

```
>>> str(98.6)
'98.6'
>>> str(1.0e4)
'10000.0'
>>> str(True)
'True'
```

Python uses the `str()` function internally when you call `print()` with objects that are not strings and when doing *string formatting*, which you'll see later in this chapter.

Escape with \

Python lets you *escape* the meaning of some characters within strings to achieve effects that would otherwise be difficult to express. By preceding a character with a backslash (`\`), you give it a special meaning. The most common escape sequence is `\n` , which means to begin a new line. With this you can create multiline strings from a one-line string:

```
>>> palindrome = 'A man,\nA plan,\nA canal:\nPanama.'
>>> print(palindrome)
A man,
A plan,
A canal:
Panama.
```

You will see the escape sequence `\t` (tab) used to align text:

```
>>> print('\tabc')
    abc
>>> print('a\tbc')
a    bc
>>> print('ab\tc')
ab      c
>>> print('abc\t')
abc
```

(The final string has a terminating tab which, of course, you can't see.)

You might also need `\'` or `\"` to specify a literal single or double quote inside a string that's quoted by the same character:

```
>>> testimony = "\"I did nothing!\" he said. \"Or that other thing.\""
>>> testimony
'"I did nothing!" he said. "Or that other thing."'
>>> print(testimony)
"I did nothing!" he said. "Or that other thing."
```

```
>>> fact = "The world's largest rubber duck was 54'2\" by 65'7\" by 105'"
>>> print(fact)
The world's largest rubber duck was 54'2" by 65'7" by 105'
```

And if you need a literal backslash, type two of them (the first escapes the second):

```
>>> speech = 'The backslash (\\) bends over backwards to please you.'
>>> print(speech)
The backslash (\) bends over backwards to please you.
>>>
```

As I mentioned early in this chapter, a *raw string* negates these escapes:

```
>>> info = r'Type a \n to get a new line in a normal string'
>>> info
'Type a \\n to get a new line in a normal string'
>>> print(info)
```

```
Type a \n to get a new line in a normal string
```

(The extra backslash in the first `info` output was added by the interactive interpreter.)

A raw string does not undo any real (not `'\n'`) newlines:

```
>>> poem = r'''Boys and girls, come out to play.
... The moon doth shine as bright as day.'''
>>> poem
'Boys and girls, come out to play.\nThe moon doth shine as bright as day.'
>>> print(poem)
Boys and girls, come out to play.
The moon doth shine as bright as day.
```

Combine by Using `+`

You can combine literal strings or string variables in Python by using the `+` operator:

```
>>> 'Release the kraken! ' + 'No, wait!'
'Release the kraken! No, wait!'
```

You can also combine *literal strings* (not string variables) just by having one after the other:

```
>>> "My word! " "A gentleman caller!"
'My word! A gentleman caller!'
>>> "Alas! ""The kraken!"
'Alas! The kraken!'
```

If you have a lot of these, you can avoid escaping the line endings by surrounding them with parentheses:

```
>>> vowels = ( 'a'
... "e" '''i'''
... 'o' """u"""
```

```
... )
>>> vowels
'aeiou'
```

Python does *not* add spaces for you when concatenating strings, so in some earlier examples, we needed to include spaces explicitly. Python *does* add a space between each argument to a `print()` statement and a newline at the end.

```
>>> a = 'Duck.'
>>> b = a
>>> c = 'Grey Duck!'
>>> a + b + c
'Duck.Duck.Grey Duck!'
>>> print(a, b, c)
Duck. Duck. Grey Duck!
```

Duplicate with *

You use the `*` operator to duplicate a string. Try typing these lines into your interactive interpreter and see what they print:

```
>>> start = 'Na ' * 4 + '\n'
>>> middle = 'Hey ' * 3 + '\n'
>>> end = 'Goodbye.'
>>> print(start + start + middle + end)
```

Notice that the `*` has higher precedence than `+`, so the string is duplicated before the line feed is tacked on.

Get a Character with []

To get a single character from a string, specify its *offset* inside square brackets after the string's name. The first (leftmost) offset is 0, the next is 1, and so on. The last (rightmost) offset can be specified with –1, so you don't have to count; going to the left are –2, –3, and so on:

```
>>> letters = 'abcdefghijklmnopqrstuvwxyz'
>>> letters[0]
'a'
>>> letters[1]
'b'
>>> letters[-1]
'z'
>>> letters[-2]
'y'
>>> letters[25]
'z'
>>> letters[5]
'f'
```

If you specify an offset that is the length of the string or longer (remember, offsets go from 0 to length–1), you'll get an exception:

```
>>> letters[100]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Indexing works the same with the other sequence types (lists and tuples), which I cover in Chapter 7.

Because strings are immutable, you can't insert a character directly into one or change the character at a specific index. Let's try to change `'Henny'` to `'Penny'` and see what happens:

```
>>> name = 'Henny'
>>> name[0] = 'P'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Instead you need to use some combination of string functions such as `replace()` or a *slice* (which we look at in a moment):

```
>>> name = 'Henny'
```

```
>>> name.replace('H', 'P')
'Penny'
>>> 'P' + name[1:]
'Penny'
```

We didn't change the value of `name`. The interactive interpreter just printed the result of the replacement.

Get a Substring with a Slice

You can extract a *substring* (a part of a string) from a string by using a *slice*. You define a slice by using square brackets, a *`start`* offset, an *`end`* offset, and an optional *`step`* count between them. You can omit some of these. The slice will include characters from offset *`start`* to one before *`end`*:

- `[:]` extracts the entire sequence from start to end.
- `[` *`start`* `:]` specifies from the *`start`* offset to the end.
- `[:` *`end`* `]` specifies from the beginning to the *`end`* offset minus 1.
- `[` *`start`* `:` *`end`* `]` indicates from the *`start`* offset to the *`end`* offset minus 1.
- `[` *`start`* `:` *`end`* `:` *`step`* `]` extracts from the *`start`* offset to the *`end`* offset minus 1, skipping characters by *`step`*.

As before, offsets go 0, 1, and so on from the start to the right, and –1,–2, and so forth from the end to the left. If you don't specify *`start`*, the slice uses 0 (the beginning). If you don't specify *`end`*, it uses the end of the string.

Let's make a string of the lowercase English letters:

```
>>> letters = 'abcdefghijklmnopqrstuvwxyz'
```

Using a plain `:` is the same as `0:` (the entire string):

```
>>> letters[:]
'abcdefghijklmnopqrstuvwxyz'
```

Here's an example from offset 20 to the end:

```
>>> letters[20:]
'uvwxyz'
```

Now, from offset 10 to the end:

```
>>> letters[10:]
'klmnopqrstuvwxyz'
```

And another, offset 12 through 14. Python does not include the end offset in the slice. The start offset is *inclusive*, and the end offset is *exclusive*:

```
>>> letters[12:15]
'mno'
```

The three last characters:

```
>>> letters[-3:]
'xyz'
```

In this next example, we go from offset 18 to the fourth before the end; notice the difference from the previous example, in which starting at –3 gets the `x`, but ending at –3 actually stops at –4, the `w`:

```
>>> letters[18:-3]
'stuvw'
```

In the following, we extract from 6 before the end to 3 before the end:

```
>>> letters[-6:-2]
'uvwx'
```

If you want a step size other than 1, specify it after a second colon, as shown in the next series of examples.

From the start to the end, in steps of 7 characters:

```
>>> letters[::7]
'ahov'
```

From offset 4 to 19, by 3:

```
>>> letters[4:20:3]
'ehknqt'
```

From offset 19 to the end, by 4:

```
>>> letters[19::4]
'tx'
```

From the start to offset 20 by 5:

```
>>> letters[:21:5]
'afkpu'
```

(Again, the *end* needs to be one more than the actual offset.)

And that's not all! Given a negative step size, this handy Python slicer can also step backward. This starts at the end and ends at the start, skipping nothing:

```
>>> letters[-1::-1]
'zyxwvutsrqponmlkjihgfedcba'
```

It turns out that you can get the same result by using this:

```
>>> letters[::-1]
'zyxwvutsrqponmlkjihgfedcba'
```

Slices are more forgiving of bad offsets than are single-index lookups with `[]`. A slice offset earlier than the beginning of a string is treated as `0`, and one after the end is treated as `-1`, as is demonstrated in this next series of examples.

From 50 before the end to the end:

```
>>> letters[-50:]
'abcdefghijklmnopqrstuvwxyz'
```

From 51 before the end to 50 before the end:

```
>>> letters[-51:-50]
''
```

From the start to 69 after the start:

```
>>> letters[:70]
'abcdefghijklmnopqrstuvwxyz'
```

From 70 after the start to 70 after the start:

```
>>> letters[70:71]
''
```

Get Length with len()

So far, we've used special punctuation characters such as `+` to manipulate strings. But there are only so many of these. Now let's begin to use some of Python's built-in *functions*: named pieces of code that perform certain operations.

The `len()` function counts characters in a string:

```
>>> len(letters)
```

```
26
>>> empty = ""
>>> len(empty)
0
```

You can use `len()` with other sequence types, too, as you'll see in Chapter 7.

Split with split()

Unlike `len()`, some functions are specific to strings. To use a string function, type the name of the string, a dot, the name of the function, and any *arguments* that the function needs: *`string.function(arguments)`*. There's a longer discussion of functions in Chapter 9.

You can use the built-in string `split()` function to break a string into a *list* of smaller strings based on some *separator*. We look at lists in Chapter 7. A list is a sequence of values, separated by commas and surrounded by square brackets:

```
>>> tasks = 'get gloves,get mask,give cat vitamins,call ambulance'
>>> tasks.split(',')
['get gloves', 'get mask', 'give cat vitamins', 'call ambulance']
```

In the preceding example, the string was called `tasks` and the string function was called `split()`, with the single separator argument `','`. If you don't specify a separator, `split()` uses any sequence of white space characters—newlines, spaces, and tabs:

```
>>> tasks.split()
['get', 'gloves,get', 'mask,give', 'cat', 'vitamins,call', 'ambulance']
```

You still need the parentheses when calling `split` with no arguments—that's how Python knows you're calling a function.

Combine by Using join()

Not too surprisingly, the `join()` function is the opposite of `split()`: it collapses a list of strings into a single string. It looks a bit backward because you specify the string that glues everything together first, and then the list of strings to glue: *`string`* `.join(` *`list`* `)`. So, to join the list `lines` with separating newlines, you would say `'\n'.join(lines)`. In the following example, let's join some names in a list with a comma and a space:

```
>>> crypto_list = ['Yeti', 'Bigfoot', 'Loch Ness Monster']
>>> crypto_string = ', '.join(crypto_list)
>>> print('Found and signing book deals:', crypto_string)
Found and signing book deals: Yeti, Bigfoot, Loch Ness Monster
```

Substitute by Using replace()

You use `replace()` for simple substring substitution. Give it the old substring, the new one, and how many instances of the old substring to replace. It returns the changed string but does not modify the original string. If you omit this final count argument, it replaces all instances. In this example, only one string (`'duck'`) is matched and replaced in the returned string:

```
>>> setup = "a duck goes into a bar..."
>>> setup.replace('duck', 'marmoset')
'a marmoset goes into a bar...'
>>> setup
'a duck goes into a bar...'
```

Change up to 100 of them:

```
>>> setup.replace('a ', 'a famous ', 100)
'a famous duck goes into a famous bar...'
```

When you know the exact substring(s) you want to change, `replace()` is a good choice. But watch out. In the second example, if we had substituted for the single character string `'a'` rather than the two character

string `'a '` (`a` followed by a space), we would have also changed `a` in the middle of other words:

```
>>> setup.replace('a', 'a famous', 100)
'a famous duck goes into a famous ba famousr...'
```

Sometimes, you want to ensure that the substring is a whole word, or the beginning of a word, and so on. In those cases, you need *regular expressions*, which are described in numbing detail in Chapter 12.

Strip with strip()

It's very common to strip leading or trailing "padding" characters from a string, especially spaces. The `strip()` functions shown here assume that you want to get rid of whitespace characters (`' '`, `'\t'`, `'\n'`) if you don't give them an argument. `strip()` strips both ends, `lstrip()` only from the left, and `rstrip()` only from the right. Let's say the string variable `world` contains the string `"earth"` floating in spaces:

```
>>> world = "    earth   "
>>> world.strip()
'earth'
>>> world.strip(' ')
'earth'
>>> world.lstrip()
'earth   '
>>> world.rstrip()
'    earth'
```

If the character were not there, nothing happens:

```
>>> world.strip('!')
'    earth   '
```

Besides no argument (meaning whitespace characters) or a single character, you can also tell `strip()` to remove any character in a multicharacter string:

```
>>> blurt = "What the...!!?"
>>> blurt.strip('.?!')
'What the'
```

Appendix E shows some definitions of character groups that are useful with `strip()`:

```
>>> import string
>>> string.whitespace
' \t\n\r\x0b\x0c'
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
>>> blurt = "What the...!!?"
>>> blurt.strip(string.punctuation)
'What the'
>>> prospector = "What in tarnation ...??!!"
>>> prospector.strip(string.whitespace + string.punctuation)
'What in tarnation'
```

Search and Select

Python has a large set of string functions. Let's explore how the most common of them work. Our test subject is the following string containing the text of the immortal poem "What Is Liquid?" by Margaret Cavendish, Duchess of Newcastle:

```
>>> poem = '''All that doth flow we cannot liquid name
... Or else would fire and water be the same;
... But that is liquid which is moist and wet
... Fire that property can never get.
... Then 'tis not cold that doth the fire put out
... But 'tis the wet that makes it die, no doubt.'''
```

Inspiring!

To begin, get the first 13 characters (offsets 0 to 12):

```
>>> poem[:13]
'All that doth'
```

How many characters are in this poem? (Spaces and newlines are included in the count.)

```
>>> len(poem)
250
```

Does it start with the letters `All`?

```
>>> poem.startswith('All')
True
```

Does it end with `That's all, folks!`?

```
>>> poem.endswith('That\'s all, folks!')
False
```

Python has two methods (`find()` and `index()`) for finding the offset of a substring, and has two versions of each (starting from the beginning or the end). They work the same if the substring is found. If it isn't, `find()` returns `-1`, and `index()` raises an exception.

Let's find the offset of the first occurrence of the word `the` in the poem:

```
>>> word = 'the'
>>> poem.find(word)
73
>>> poem.index(word)
73
```

And the offset of the last `the`:

```
>>> word = 'the'
>>> poem.rfind(word)
```

```
214
>>> poem.rindex(word)
214
```

But if the substring isn't in there:

```
>>> word = "duck"
>>> poem.find(word)
-1
>>> poem.rfind(word)
-1
>>> poem.index(word)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
>>> poem.rfind(word)
-1
>>> poem.rindex(word)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
```

How many times does the three-letter sequence `the` occur?

```
>>> word = 'the'
>>> poem.count(word)
3
```

Are all of the characters in the poem either letters or numbers?

```
>>> poem.isalnum()
False
```

Nope, there were some punctuation characters.

Case

In this section, we look at some more uses of the built-in string functions.

Our test string is again the following:

```
>>> setup = 'a duck goes into a bar...'
```

Remove `.` sequences from both ends:

```
>>> setup.strip('.')
'a duck goes into a bar'
```

NOTE

Because strings are immutable, none of these examples actually changes the `setup` string. Each example just takes the value of `setup`, does something to it, and returns the result as a new string.

Capitalize the first word:

```
>>> setup.capitalize()
'A duck goes into a bar...'
```

Capitalize all the words:

```
>>> setup.title()
'A Duck Goes Into A Bar...'
```

Convert all characters to uppercase:

```
>>> setup.upper()
'A DUCK GOES INTO A BAR...'
```

Convert all characters to lowercase:

```
>>> setup.lower()
'a duck goes into a bar...'
```

Swap uppercase and lowercase:

```
>>> setup.swapcase()
'A DUCK GOES INTO A BAR...'
```

Alignment

Now, let's work with some layout alignment functions. The string is aligned within the specified total number of spaces (`30` here).

Center the string within 30 spaces:

```
>>> setup.center(30)
'  a duck goes into a bar...  '
```

Left justify:

```
>>> setup.ljust(30)
'a duck goes into a bar...    '
```

Right justify:

```
>>> setup.rjust(30)
'    a duck goes into a bar...'
```

Next, we look at more ways to align a string.

Formatting

You've seen that you can *concatenate* strings by using `+`. Let's look at how to *interpolate* data values into strings using various formats. You can use this to produce reports, forms, and other outputs where appearances need to be just so.

Besides the functions in the previous section, Python has three ways of

formatting strings:

- *old style* (supported in Python 2 and 3)
- *new style* (Python 2.6 and up)
- *f-strings* (Python 3.6 and up)

Old style: %

The old style of string formatting has the form *`format_string % data`*. Inside the format string are interpolation sequences. Table 5-1 illustrates that the very simplest sequence is a `%` followed by a letter indicating the data type to be formatted.

Table 5-1. Conversion types

`%s`	string
`%d`	decimal integer
`%x`	hex integer
`%o`	octal integer
`%f`	decimal float
`%e`	exponential float
`%g`	decimal or exponential float
`%%`	a literal `%`

You can use a `%s` for any data type, and Python will format it as a string with no extra spaces.

Following are some simple examples. First, an integer:

```
>>> '%s' % 42
'42'
```

```
>>> '%d' % 42
'42'
>>> '%x' % 42
'2a'
>>> '%o' % 42
'52'
```

A float:

```
>>> '%s' % 7.03
'7.03'
>>> '%f' % 7.03
'7.030000'
>>> '%e' % 7.03
'7.030000e+00'
>>> '%g' % 7.03
'7.03'
```

An integer and a literal `%`:

```
>>> '%d%%' % 100
'100%'
```

Let's try some string and integer interpolation:

```
>>> actor = 'Richard Gere'
>>> cat = 'Chester'
>>> weight = 28


>>> "My wife's favorite actor is %s" % actor
"My wife's favorite actor is Richard Gere"


>>> "Our cat %s weighs %s pounds" % (cat, weight)
'Our cat Chester weighs 28 pounds'
```

That `%s` inside the string means to interpolate a string. The number of `%` appearances in the string needs to match the number of data items after

the `%` that follows the string. A single data item such as `actor` goes right after that final `%`. Multiple data must be grouped into a *tuple* (details in Chapter 7; it's bounded by parentheses, separated by commas) such as `(cat, weight)`.

Even though `weight` is an integer, the `%s` inside the string converted it to a string.

You can add other values in the format string between the `%` and the type specifier to designate minimum and maximum widths, alignment, and character filling. This is a little language in its own right, and more limited than the one in the next two sections. Let's take a quick look at these values:

- An initial `'%'` character.
- An optional *alignment* character: nothing or `'+'` means right-align, and `'-'` means left-align.
- An optional *minwidth* field width to use.
- An optional `'.'` character to separate *minwidth* and *maxchars*.
- An optional *maxchars* (if conversion type is `s`) saying how many characters to print from the data value. If the conversion type is `f`, this specifies *precision* (how many digits to print after the decimal point).
- The *conversion type* character from the earlier table.

This is confusing, so here are some examples for a string:

```
>>> thing = 'woodchuck'
>>> '%s' % thing
'woodchuck'
>>> '%12s' % thing
'   woodchuck'
>>> '%+12s' % thing
'   woodchuck'
>>> '%-12s' % thing
'woodchuck   '
>>> '%.3s' % thing
'woo'
>>> '%12.3s' % thing
'         woo'
>>> '%-12.3s' % thing
```

```
'woo          '
```

Once more with feeling, and a float with `%f` variants:

```
>>> thing = 98.6
>>> '%f' % thing
'98.600000'
>>> '%12f' % thing
'   98.600000'
>>> '%+12f' % thing
'  +98.600000'
>>> '%-12f' % thing
'98.600000   '
>>> '%.3f' % thing
'98.600'
>>> '%12.3f' % thing
'      98.600'
>>> '%-12.3f' % thing
'98.600      '
```

And an integer with `%d`:

```
>>> thing = 9876
>>> '%d' % thing
'9876'
>>> '%12d' % thing
'        9876'
>>> '%+12d' % thing
'       +9876'
>>> '%-12d' % thing
'9876        '
>>> '%.3d' % thing
'9876'
>>> '%12.3d' % thing
'        9876'
>>> '%-12.3d' % thing
'9876        '
```

For an integer, the `%+12d` just forces the sign to be printed, and the format strings with `.3` in them have no effect as they do for a float.

New style: {} and format()

Old style formatting is still supported. In Python 2, which will freeze at version 2.7, it will be supported forever. For Python 3, use the "new style" formatting described in this section. If you have Python 3.6 or newer, *f-strings* ("Newest Style: f-strings") are even better.

"New style" formatting has the form `format_string.format(data)`.

The format string is not exactly the same as the one in the previous section. The simplest usage is demonstrated here:

```
>>> thing = 'woodchuck'
>>> '{}'.format(thing)
'woodchuck'
```

The arguments to the `format()` function need to be in the order as the `{}` placeholders in the format string:

```
>>> thing = 'woodchuck'
>>> place = 'lake'
>>> 'The {} is in the {}.'.format(thing, place)
'The woodchuck is in the lake.'
```

With new-style formatting, you can also specify the arguments by position like this:

```
>>> 'The {1} is in the {0}.'.format(place, thing)
'The woodchuck is in the lake.'
```

The value `0` referred to the first argument, `place`, and `1` referred to `thing`.

The arguments to `format()` can also be named arguments

```
>>> 'The {thing} is in the {place}'.format(thing='duck', place='bathtub')
'The duck is in the bathtub'
```

or a dictionary:

```
>>> d = {'thing': 'duck', 'place': 'bathtub'}
```

In the following example, `{0}` is the first argument to `format()` (the dictionary `d`):

```
>>> 'The {0[thing]} is in the {0[place]}.'.format(d)
'The duck is in the bathtub.'
```

These examples all printed their arguments with default formats. New-style formatting has a slightly different format string definition from the old-style one (examples follow):

- An initial colon (`':'`).
- An optional *fill* character (default `' '`) to pad the value string if it's shorter than *minwidth*.
- An optional *alignment* character. This time, left alignment is the default. `'<'` also means left, `'>'` means right, and `'^'` means center.
- An optional *sign* for numbers. Nothing means only prepend a minus sign (`'-'`) for negative numbers. `' '` means prepend a minus sign for negative numbers, and a space (`' '`) for positive ones.
- An optional *minwidth*. An optional period (`'.'`) to separate *minwidth* and *maxchars*.
- An optional *maxchars*.
- The *conversion type*.

```
>>> thing = 'wraith'
>>> place = 'window'
>>> 'The {} is at the {}'.format(thing, place)
'The wraith is at the window'
>>> 'The {:10s} is at the {:10s}'.format(thing, place)
'The wraith     is at the window    '
>>> 'The {:<10s} is at the {:<10s}'.format(thing, place)
'The wraith     is at the window    '
>>> 'The {:^10s} is at the {:^10s}'.format(thing, place)
'The   wraith   is at the   window  '
>>> 'The {:>10s} is at the {:>10s}'.format(thing, place)
```

```
'The     wraith is at the     window'
>>> 'The {:!^10s} is at the {:!^10s}'.format(thing, place)
'The !!wraith!! is at the !!window!!'
```

Newest Style: f-strings

f-strings appeared in Python 3.6, and are now the recommended way of formatting strings.

To make an f-string:

- Type the letter `f` or `F` directly before the initial quote.
- Include variable names or expressions within curly brackets (`{}`) to get their values into the string.

It's like the previous section's "new-style" formatting, but without the `format()` function, and without empty brackets (`{}`) or positional ones (`{1}`) in the format string.

```
>>> thing = 'wereduck'
>>> place = 'werepond'
>>> f'The {thing} is in the {place}'
'The wereduck is in the werepond'
```

As I already mentioned, expressions are also allowed inside the curly brackets:

```
>>> f'The {thing.capitalize()} is in the {place.rjust(20)}'
'The Wereduck is in the             werepond'
```

This means that the things that you could do inside `format()` in the previous section, you can now do inside a `{}` in your main string. This seems easier to read.

f-strings use the same formatting language (width, padding, alignment) as new-style formatting, after a `':'` .

```
>>> f'The {thing:>20} is in the {place:.^20}'
'The             wereduck is in the ......werepond......'
```

Starting in Python 3.8, f-strings gain a new shortcut that's helpful when you want to print variable names as well as their values. This is handy when debugging. The trick is to have a single `=` after the name in the `{}`-enclosed part of the f-string:

```
>>> f'{thing =}, {place =}'
thing = 'wereduck', place = 'werepond'
```

The name can actually be an expression, and it will be printed literally:

```
>>> f'{thing[-4:] =}, {place.title() =}'
thing[-4:] = 'duck', place.title() = 'Werepond'
```

Finally, the `=` can be followed by a `:` and the formatting arguments like width and alignment:

```
>>> f'{thing = :>4.4}'
thing = 'were'
```

More String Things

Python has many more string functions than I've shown here. Some will turn up in later chapters (especially Chapter 12), but you can find all the details at the standard documentation link.

Coming Up

You'll find Froot Loops at the grocery store, but Python loops are at the first counter in the next chapter.

Things to Do

5.1 Capitalize the word starting with `m`:

```
>>> song = """When an eel grabs your arm,
... And it causes great harm,
... That's - a moray!"""
```

5.2 Print each list question with its correctly matching answer, in the form:

Q: *question*
A: *answer*

```
>>> questions = [
...     "We don't serve strings around here. Are you a string?",
...     "What is said on Father's Day in the forest?",
...     "What makes the sound 'Sis! Boom! Bah!'?"
...     ]
>>> answers = [
...     "An exploding sheep.",
...     "No, I'm a frayed knot.",
...     "'Pop!' goes the weasel."
...     ]
```

5.3 Write the following poem by using old-style formatting. Substitute the strings `'roast beef'`, `'ham'`, `'head'`, and `'clam'` into this string:

```
My kitty cat likes %s,
My kitty cat likes %s,
My kitty cat fell on his %s
And now thinks he's a %s.
```

5.4 Write a form letter by using new-style formatting. Save the following string as `letter` (you'll use it in the next exercise):

```
Dear {salutation} {name},

Thank you for your letter. We are sorry that our {product}
{verbed} in your {room}. Please note that it should never
be used in a {room}, especially near any {animals}.

Send us your receipt and {amount} for shipping and handling.
We will send you another {product} that, in our tests,
is {percent}% less likely to have {verbed}.

Thank you for your support.

Sincerely,
{spokesman}
{job_title}
```

5.5 Assign values to variable strings named `'salutation'`, `'name'`, `'product'`, `'verbed'` (past tense verb), `'room'`, `'animals'`, `'percent'`, `'spokesman'`, and `'job_title'`. Print `letter` with these values, using `letter.format()`.

5.6 After public polls to name things, a pattern emerged: an English submarine (Boaty McBoatface), an Australian racehorse (Horsey McHorseface), and a Swedish train (Trainy McTrainface). Use `%` formatting to print the winning name at the state fair for a prize duck, gourd, and spitz.

5.7 Do the same, with `format()` formatting.

5.8 Once more, with feeling, and *f strings*.