CHAPTER 3

Working with Strings

You've seen strings before, and know how to make them. You've also looked at how to access their individual characters by indexing and slicing. In this chapter, you see how to use them to format other values (for printing, for example), and take a quick look at the useful things you can do with string methods, such as splitting, joining, searching, and more.

Basic String Operations

All the standard sequence operations (indexing, slicing, multiplication, membership, length, minimum, and maximum) work with strings, as you saw in the previous chapter. Remember, however, that strings are immutable, so all kinds of item or slice assignments are illegal:

```
>>> website = 'http://www.python.org'
>>> website[-3:] = 'com'
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in ?
    website[-3:] = 'com'
TypeError: object doesn't support slice assignment
```

String Formatting: The Short Version

If you are new to Python programming, chances are you won't need all the options that are available in Python string formatting, so I'll give you the short version here. If you are interested in the details, take a look at the section "String Formatting: The Long Version," which follows. Otherwise, just read this and skip down to the section "String Methods."

String formatting uses the (aptly named) string formatting operator, the percent (%) sign.

Note As you may remember, % is also used as a modulus (remainder) operator.

To the left of the %, you place a string (the format string); to the right of it, you place the value you want to format. You can use a single value such as a string or a number, you can use a tuple of values (if you want to format more than one), or, as I discuss in the next chapter, you can use a dictionary. The most common case is the tuple:

```
>>> format = "Hello, %s. %s enough for ya?"
>>> values = ('world', 'Hot')
>>> print format % values
Hello, world. Hot enough for ya?
```

■ **Note** If you use a list or some other sequence instead of a tuple, the sequence will be interpreted as a single value. Only tuples and dictionaries (discussed in Chapter 4) will allow you to format more than one value.

The %s parts of the format string are called *conversion specifiers*. They mark the places where the values are to be inserted. The s means that the values should be formatted as if they were strings; if they aren't, they'll be converted with str. This works with most values. For a list of other specifier types, see Table 3-1 later in the chapter.

■ **Note** To actually include a percent sign in the format string, you must write %% so Python doesn't mistake it for the beginning of a conversion specifier.

If you are formatting real numbers (floats), you can use the f specifier type and supply the *precision* as a . (dot), followed by the number of decimals you want to keep. The format specifier always ends with a type character, so you must put the precision before that:

```
>>> format = "Pi with three decimals: %.3f"
>>> from math import pi
>>> print format % pi
Pi with three decimals: 3.142
```

TEMPLATE STRINGS

The string module offers another way of formatting values: template strings. They work more like variable substitution in many UNIX shells, with $foo being replaced by a keyword argument called foo (for more about keyword arguments, see Chapter 6), which is passed to the template method substitute:

```
>>> from string import Template
>>> s = Template('$x, glorious $x!')
>>> s.substitute(x='slurm')
'slurm, glorious slurm!'
```

If the replacement field is part of a word, the name must be enclosed in braces, in order to clearly indicate where it ends:

```
>>> s = Template("It's ${x}tastic!")
>>> s.substitute(x='slurm')
"It's slurmtastic!"
```

In order to insert a dollar sign, use $$:

```
>>> s = Template("Make $$ selling $x!")
>>> s.substitute(x='slurm')
'Make $ selling slurm!'
```

Instead of using keyword arguments, you can supply the value-name pairs in a dictionary (see Chapter 4):

```
>>> s = Template('A $thing must never $action.')
>>> d = {}
>>> d['thing'] = 'gentleman'
>>> d['action'] = 'show his socks'
>>> s.substitute(d)
'A gentleman must never show his socks.'
```

There is also a method called safe_substitute that will not complain about missing values or incorrect uses of the $ character.[1]

1. For more information, see Section 4.1.2, “Template strings,” of the Python Library Reference (http://python.org/doc/lib/node40.html).

String Formatting: The Long Version

The right operand of the formatting operator may be anything; if it is either a tuple or a mapping (like a dictionary), it is given special treatment. We haven't looked at mappings (such as dictionaries) yet, so let's focus on tuples here. We'll use mappings in formatting in Chapter 4, where they're discussed in greater detail.

If the right operand is a tuple, each of its elements is formatted separately, and you need a conversion specifier for each of the values.

■ **Note** If you write the tuple to be converted as part of the conversion expression, you must enclose it in parentheses to avoid confusing Python:

```
>>> '%s plus %s equals %s' % (1, 1, 2)
'1 plus 1 equals 2'
>>> '%s plus %s equals %s' % 1, 1, 2 # Lacks parentheses!
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: not enough arguments for format string
```

A basic conversion specifier (as opposed to a full conversion specifier, which may contain a mapping key as well; see Chapter 4 for more information) consists of the items that follow. Note that the order of these items is crucial.

- **The `%` character**: This marks the beginning of the conversion specifier.
- **Conversion flags**: These are optional and may be `-`, indicating left alignment; `+`, indicating that a sign should precede the converted value; " " (a space character), indicating that a space should precede positive numbers; or `0`, indicating that the conversion should be zero-padded.
- **The minimum field width**: This is also optional and specifies that the converted string will be at least this wide. If this is an * (asterisk), the width will be read from the value tuple.
- **A `.` (dot) followed by the precision**: This is also optional. If a real number is converted, this many decimals should be shown. If a string is converted, this number is the *maximum field width*. If this is an * (asterisk), the precision will be read from the value tuple.
- **The conversion type**: This can be any of the types listed in Table 3-1.

Table 3-1. *String Formatting Conversion Types*

Conversion Type	Meaning
`d`, `i`	Signed integer decimal
`o`	Unsigned octal
`u`	Unsigned decimal

Conversion Type	Meaning
`x`	Unsigned hexadecimal (lowercase)
`X`	Unsigned hexadecimal (uppercase)
`e`	Floating-point exponential format (lowercase)
`E`	Floating-point exponential format (uppercase)
`f`, `F`	Floating-point decimal format
`g`	Same as `e` if exponent is greater than –4 or less than precision; `f` otherwise
`G`	Same as `E` if exponent is greater than –4 or less than precision; `F` otherwise
`c`	Single character (accepts an integer or a single character string)
`r`	String (converts any Python object using `repr`)
`s`	String (converts any Python object using `str`)

The following sections discuss the various elements of the conversion specifiers in more detail.

Simple Conversion

The simple conversion, with only a conversion type, is really easy to use:

```
>>> 'Price of eggs: $%d' % 42
'Price of eggs: $42'
>>> 'Hexadecimal price of eggs: %x' % 42
'Hexadecimal price of eggs: 2a'
>>> from math import pi
>>> 'Pi: %f...' % pi
'Pi: 3.141593...'
>>> 'Very inexact estimate of pi: %i' % pi
'Very inexact estimate of pi: 3'
>>> 'Using str: %s' % 42L
'Using str: 42'
>>> 'Using repr: %r' % 42L
'Using repr: 42L'
```

Width and Precision

A conversion specifier may include a field width and a precision. The width is the minimum number of characters reserved for a formatted value. The precision is (for a numeric conversion) the number of decimals that will be included in the result or (for a string conversion) the maximum number of characters the formatted value may have.

These two parameters are supplied as two integer numbers (width first, then precision), separated by a `.` (dot). Both are optional, but if you want to supply only the precision, you must also include the dot:

```
>>> '%10f' % pi      # Field width 10
'  3.141593'
```

```
>>> '%10.2f' % pi    # Field width 10, precision 2
'      3.14'
>>> '%.2f' % pi      # Precision 2
'3.14'
>>> '%.5s' % 'Guido van Rossum'
'Guido'
```

You can use an * (asterisk) as the width or precision (or both). In that case, the number will be read from the tuple argument:

```
>>> '%.*s' % (5, 'Guido van Rossum')
'Guido'
```

Signs, Alignment, and Zero-Padding

Before the width and precision numbers, you may put a "flag," which may be either zero, plus, minus, or blank. A zero means that the number will be zero-padded:

```
>>> '%010.2f' % pi
'0000003.14'
```

It's important to note here that the leading zero in `010` in the preceding code does *not* mean that the width specifier is an octal number, as it would in a normal Python number. When you use `010` as the width specifier, it means that the width should be 10 and that the number should be zero-padded, not that the width should be 8:

```
>>> 010
8
```

A minus sign (-) left-aligns the value:

```
>>> '%-10.2f' % pi
'3.14      '
```

As you can see, any extra space is put on the right-hand side of the number.

A blank (" ") means that a blank should be put in front of positive numbers. This may be useful for aligning positive and negative numbers:

```
>>> print ('% 5d' % 10) + '\n' + ('% 5d' % -10)F
   10
  -10
```

Finally, a plus (+) means that a sign (either plus or minus) should precede both positive and negative numbers (again, useful for aligning):

```
>>> print ('%+5d' % 10) + '\n' + ('%+5d' % -10)
  +10
  -10
```

In the example shown in Listing 3-1, I use the asterisk width specifier to format a table of fruit prices, where the user enters the total width of the table. Because this information is supplied by the user, I can't hard-code the field widths in my conversion specifiers. By using the asterisk, I can have the field width read from the converted tuple.

Listing 3-1. *String Formatting Example*

```
# Print a formatted price list with a given width

width = input('Please enter width: ')

price_width = 10
item_width = width - price_width

header_format = '%-*s%*s'
format        = '%-*s%*.2f'

print '=' * width

print header_format % (item_width, 'Item', price_width, 'Price')

print '-' * width

print format % (item_width, 'Apples', price_width, 0.4)
print format % (item_width, 'Pears', price_width, 0.5)
print format % (item_width, 'Cantaloupes', price_width, 1.92)
print format % (item_width, 'Dried Apricots (16 oz.)', price_width, 8)
print format % (item_width, 'Prunes (4 lbs.)', price_width, 12)

print '=' * width
```

The following is a sample run of the program:

```
Please enter width: 35
===================================
Item                          Price
———————————————————————————————————
Apples                         0.40
Pears                          0.50
Cantaloupes                    1.92
Dried Apricots (16 oz.)        8.00
Prunes (4 lbs.)               12.00
===================================
```

String Methods

You have already encountered methods in lists. Strings have a much richer set of methods, in part because strings have "inherited" many of their methods from the `string` module where they resided as functions in earlier versions of Python (and where you may still find them, if you feel the need).

Because there are so many string methods, only some of the most useful ones are described here. For a full reference, see Appendix B. In the description of the string methods, you will find references to other, related string methods in this chapter (marked "See also") or in Appendix B.

BUT STRING ISN'T DEAD

Even though string methods have completely upstaged the `string` module, the module still includes a few constants and functions that *aren't* available as string methods. The `maketrans` function is one example and will be discussed together with the `translate` method in the material that follows. The following are some useful constants available from `string`.[2]

- `string.digits`: A string containing the digits 0–9
- `string.letters`: A string containing all letters (uppercase and lowercase)
- `string.lowercase`: A string containing all lowercase letters
- `string.printable`: A string containing all printable characters
- `string.punctuation`: A string containing all punctuation characters
- `string.uppercase`: A string containing all uppercase letters

Note that the string constant letters (such as `string.letters`) are *locale-dependent* (that is, their exact values depend on the language for which Python is configured).[3] If you want to make sure you're using ASCII, you can use the variants with `ascii_` in their names, such as `string.ascii_letters`.

find

The `find` method finds a substring within a larger string. It returns the leftmost index where the substring is found. If it is *not* found, –1 is returned:

```
>>> 'With a moo-moo here, and a moo-moo there'.find('moo')
7
>>> title = "Monty Python's Flying Circus"
>>> title.find('Monty')
0
```

2. For a more thorough description of the module, check out Section 4.1 of the Python Library Reference (`http://python.org/doc/lib/module-string.html`).
3. In Python 3.0, `string.letters` and friends will be removed. You will need to use constants like `string.ascii_letters` instead.

```
>>> title.find('Python')
6
>>> title.find('Flying')
15
>>> title.find('Zirquss')
-1
```

In our first encounter with membership in Chapter 2, we created part of a spam filter by using the expression `'$$$' in subject`. We could also have used `find` (which would also have worked prior to Python 2.3, when `in` could be used only when checking for single character membership in strings):

```
>>> subject = '$$$ Get rich now!!! $$$'
>>> subject.find('$$$')
0
```

Note The string method `find` does *not* return a Boolean value. If `find` returns 0, as it did here, it means that it *has* found the substring, at index zero.

You may also supply a starting point for your search and, optionally, an ending point:

```
>>> subject = '$$$ Get rich now!!! $$$'
>>> subject.find('$$$')
0
>>> subject.find('$$$', 1) # Only supplying the start
20
>>> subject.find('!!!')
16
>>> subject.find('!!!', 0, 16) # Supplying start and end
-1
```

Note that the range specified by the start and stop values (second and third parameters) includes the first index but not the second. This is common practice in Python.

In Appendix B: `rfind`, `index`, `rindex`, `count`, `startswith`, `endswith`.

join

A very important string method, `join` is the inverse of `split`. It is used to join the elements of a sequence:

```
>>> seq = [1, 2, 3, 4, 5]
>>> sep = '+'
>>> sep.join(seq) # Trying to join a list of numbers
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: sequence item 0: expected string, int found
```

```
>>> seq = ['1', '2', '3', '4', '5']
>>> sep.join(seq) # Joining a list of strings
'1+2+3+4+5'
>>> dirs = '', 'usr', 'bin', 'env'
>>> '/'.join(dirs)
'/usr/bin/env'
>>> print 'C:' + '\\'.join(dirs)
C:\usr\bin\env
```

As you can see, the sequence elements that are to be joined must all be strings. Note how in the last two examples I use a list of directories and format them according to the conventions of UNIX and DOS/Windows simply by using a different separator (and adding a drive name in the DOS version).

See also: `split`.

lower

The `lower` method returns a lowercase version of the string:

```
>>> 'Trondheim Hammer Dance'.lower()
'trondheim hammer dance'
```

This can be useful if you want to write code that is case insensitive—that is, code that ignores the difference between uppercase and lowercase letters. For instance, suppose you want to check whether a user name is found in a list. If your list contains the string `'gumby'` and the user enters his name as `'Gumby'`, you won't find it:

```
>>> if 'Gumby' in ['gumby', 'smith', 'jones']: print 'Found it!'
...
>>>
```

Of course, the same thing will happen if you have stored `'Gumby'` and the user writes `'gumby'`, or even `'GUMBY'`. A solution to this is to convert all names to lowercase both when storing and searching. The code would look something like this:

```
>>> name = 'Gumby'
>>> names = ['gumby', 'smith', 'jones']
>>> if name.lower() in names: print 'Found it!'
...
Found it!
>>>
```

See also: `translate`.

In Appendix B: `islower`, `capitalize`, `swapcase`, `title`, `istitle`, `upper`, `isupper`.

TITLE CASING

One relative of `lower` is the `title` method (see Appendix B), which title cases a string—that is, all words start with uppercase characters, and all other characters are lowercased. However, the word boundaries are defined in a way that may give some unnatural results:

```
>>> "that's all folks".title()
"That'S All, Folks"
```

An alternative is the `capwords` function from the `string` module:

```
>>> import string
>>> string.capwords("that's all, folks")
"That's All, Folks"
```

Of course, if you want a truly correctly capitalized title (which depends on the style you're using—possibly lowercasing articles, coordinating conjunctions, prepositions with fewer than five letters, and so forth), you're basically on your own.

replace

The `replace` method returns a string where all the occurrences of one string have been replaced by another:

```
>>> 'This is a test'.replace('is', 'eez')
'Theez eez a test'
```

If you have ever used the "search and replace" feature of a word processing program, you will no doubt see the usefulness of this method.

See also: `translate`.

In Appendix B: `expandtabs`.

split

A very important string method, `split` is the inverse of `join`, and is used to split a string into a sequence:

```
>>> '1+2+3+4+5'.split('+')
['1', '2', '3', '4', '5']
>>> '/usr/bin/env'.split('/')
['', 'usr', 'bin', 'env']
>>> 'Using   the   default'.split()
['Using', 'the', 'default']
```

Note that if no separator is supplied, the default is to split on all runs of consecutive whitespace characters (spaces, tabs, newlines, and so on).

See also: `join`.

In Appendix B: `rsplit`, `splitlines`.

strip

The `strip` method returns a string where whitespace on the left and right (but not internally) has been stripped (removed):

```
>>> '      internal whitespace is kept      '.strip()
'internal whitespace is kept'
```

As with `lower`, `strip` can be useful when comparing input to stored values. Let's return to the user name example from the section on `lower`, and let's say that the user inadvertently types a space after his name:

```
>>> names = ['gumby', 'smith', 'jones']
>>> name = 'gumby '
>>> if name in names: print 'Found it!'
...
>>> if name.strip() in names: print 'Found it!'
...
Found it!
>>>
```

You can also specify which characters are to be stripped, by listing them all in a string parameter:

```
>>> '*** SPAM * for * everyone!!! ***'.strip(' *!')
'SPAM * for * everyone'
```

Stripping is performed only at the ends, so the internal asterisks are not removed.

In Appendix B: `lstrip`, `rstrip`.

translate

Similar to `replace`, `translate` replaces parts of a string, but unlike `replace`, `translate` works only with single characters. Its strength lies in that it can perform several replacements simultaneously, and can do so more efficiently than `replace`.

There are quite a few rather technical uses for this method (such as translating newline characters or other platform-dependent special characters), but let's consider a simpler (although slightly more silly) example. Let's say you want to translate a plain English text into one with a German accent. To do this, you must replace the character *c* with *k*, and *s* with *z*.

Before you can use `translate`, however, you must make a *translation table*. This translation table is a full listing of which characters should be replaced by which. Because this table (which is actually just a string) has 256 entries, you won't write it out yourself. Instead, you'll use the function `maketrans` from the `string` module.

The `maketrans` function takes two arguments: two strings of equal length, indicating that each character in the first string should be replaced by the character in the same position in the second string. Got that? In the case of our simple example, the code would look like the following:

```
>>> from string import maketrans
>>> table = maketrans('cs', 'kz')
```

WHAT'S IN A TRANSLATION TABLE?

A translation table is a string containing one replacement letter for each of the 256 characters in the ASCII character set:

```
>>> table = maketrans('cs', 'kz')
>>> len(table)
256
>>> table[97:123]
'abkdefghijklmnopqrztuvwxyz'
>>> maketrans('', '')[97:123]
'abcdefghijklmnopqrstuvwxyz'
```

As you can see, I've sliced out the part of the table that corresponds to the lowercase letters. Take a look at the alphabet in the table and that in the empty translation (which doesn't change anything). The empty translation has a normal alphabet, while in the preceding code, the letter *c* has been replaced by *k*, and *s* has been replaced by *z*.

Once you have this table, you can use it as an argument to the `translate` method, thereby translating your string:

```
>>> 'this is an incredible test'.translate(table)
'thiz iz an inkredible tezt'
```

An optional second argument can be supplied to `translate`, specifying letters that should be deleted. If you wanted to emulate a really fast-talking German, for instance, you could delete all the spaces:

```
>>> 'this is an incredible test'.translate(table, ' ')
'thizizaninkredibletezt'
```

See also: `replace`, `lower`.

PROBLEMS WITH NON-ENGLISH STRINGS

Sometimes string methods such as `lower` won't work quite the way you want them to—for instance, if you happen to use a non-English alphabet. Let's say you want to convert the uppercase Norwegian word *BØLLEFRØ* to its lowercase equivalent:

```
>>> print 'BØLLEFRØ'.lower()
bØllefrØ
```

As you can see, this didn't really work because Python doesn't consider *Ø* a real letter. In this case, you can use `translate` to do the translation:

```
>>> table = maketrans('ÆØÅ', 'æøå')
>>> word = 'KÅPESØM'
>>> print word.lower()
kÅpesØm
>>> print word.translate(table)
KåPESøM
>>> print word.translate(table).lower()
kåpesøm
```

Then again, simply using Unicode might solve your problems:

```
>>> print u'ærnæringslære'.upper()
ÆRNÆRINGSLÆRE
```

You might also want to check out the `locale` module for some internationalization functionality.

A Quick Summary

In this chapter, you have seen two important ways of working with strings:

String formatting: The modulo operator (%) can be used to splice values into a string that contains conversion flags, such as `%s`. You can use this to format values in many ways, including right or left justification, setting a specific field width and precision, adding a sign (plus or minus), or left-padding with zeros.

String methods: Strings have a plethora of methods. Some of them are extremely useful (such as `split` and `join`), while others are used less often (such as `istitle` or `capitalize`).

New Functions in This Chapter

Function	Description
`string.capwords(s[, sep])`	Splits `s` with `split` (using `sep`), capitalize items, and join with a single space
`string.maketrans(from, to)`	Makes a translation table for `translate`

What Now?

Lists, strings, and dictionaries are three of the most important data types in Python. You've seen lists and strings, so guess what's next? In the next chapter, you see how dictionaries not only support indices, but other kinds of keys (such as strings or tuples) as well. Dictionaries also support a few methods, although not as many as strings.