Beginning\_Python\_Magnus\_03

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 to the

section “String Methods.”

Formatting values as strings is such an important operation, and one that has to cater to such a diverse

set of requirements, that several approaches have been added to the language over the years. Historically,

the main solution was to use the (aptly named) string formatting operator, the percent sign. The behavior of

this operator emulates the classic printf function from the C language. 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')

>>> format % values

'Hello, world. Hot enough for ya?'

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. Other specifiers lead to other forms of conversion; for example, %.3f will format the

value as a floating-point number with three decimals.

This formatting method still works, and is still very much alive in a lot of code out there, so you might

run into it. Another solution you may encounter is so-called template strings, which appeared a while back

as an attempt to simplify the basic formatting mechanism, using a syntax similar to UNIX shells, for example.

>>> from string import Template

>>> tmpl = Template("Hello, $who! $what enough for ya?")

>>> tmpl.substitute(who="Mars", what="Dusty")

'Hello, Mars! Dusty enough for ya?'

The arguments with the equal signs in them are so-called keyword arguments—you’ll hear a lot about those

in Chapter 6. In the context of string formatting, you can just think of them as a way of supplying values to

named replacement fields.

When writing new code, the mechanism of choice is the format string method, which combines and

extends the strong points of the earlier methods. Each replacement field is enclosed in curly brackets and

may include a name, as well as information on how to convert and format the value supplied for that field.

The simplest case is where the fields have no name, or where each name is just an index.

>>> "{}, {} and {}".format("first", "second", "third")

'first, second and third'

>>> "{0}, {1} and {2}".format("first", "second", "third")

'first, second and third'

The indices need not be in order like this, though.

>>> "{3} {0} {2} {1} {3} {0}".format("be", "not", "or", "to")

'to be or not to be'

Named fields work just as expected.

>>> from math import pi

>>> "{name} is approximately {value:.2f}.".format(value=pi, name="π")

'π is approximately 3.14.'

The ordering of the keyword arguments does not matter, of course. In this case, I have also supplied a format

specifier of .2f, separated from the field name by a colon, meaning we want float-formatting with three

decimals. Without the specified, the result would be as follows:

>>> "{name} is approximately {value}.".format(value=pi, name="π")

'π is approximately 3.141592653589793.'

Finally, in Python 3.6, there’s a shortcut you can use if you have variables named identically to

corresponding replacement fields. In that case, you can use so-called f-strings, written with the prefix f.

>>> from math import e

>>> f"Euler's constant is roughly {e}."

"Euler's constant is roughly 2.718281828459045."

Here, the replacement field named e simply extracts the value of the variable of the same name, as the string

is being constructed. This is equivalent to the following, slightly more explicit expression:

>>> "Euler's constant is roughly {e}.".format(e=e)

"Euler's constant is roughly 2.718281828459045."

String Formatting: The Long Version

The string formatting facilities are extensive, so even this long version falls short of a complete exploration

of all its details, but let’s take a look at the main components. The idea is that we call the format method on

a string, supplying it with values that we want to format. The string contains information on how to perform

this formatting, specified in a template mini-language. Each value is spliced into the string in one of several

replacement fields, each of which is enclosed in curly braces. If you want to include literal braces in the final

result, you can specify those by using double braces in the format string, that is, {{ or }}.

>>> "{{ceci n'est pas une replacement field}}".format()

"{ceci n'est pas une replacement field}"

The most exciting part of a format string is found in the guts of the replacement fields, consisting of the

following parts, all of which are optional:

•A field name. An index or identifier. This tells us which value will be formatted and

spliced into this specific field. In addition to naming the object itself, we may also

name a specific part of the value, such as an element of a list, for example.

•A conversion flag. An exclamation mark, followed by a single character. The

currently supported ones are r (for repr), s (for str), or a (for ascii). If supplied,

this flag overrides the object’s own formatting mechanisms and uses the specified

function to turn it into a string before any further formatting.

•A format specifier. A colon, followed by an expression in the format specification

mini-language. This lets us specify details of the final formatting, including the

type of formatting (for example, string, floating-point or hexadecimal number), the

width of the field and the precision of numbers, how to display signs and thousands

separators, and various forms of alignment and padding.

Let’s look at some of these elements in a bit more detail.

Replacement Field Names

In the simplest case, you just supply unnamed arguments to format and use unnamed fields in the format

string. The fields and arguments are then paired off in the order they are given. You can also provide the

arguments with names, which is then used in replacement fields to request these specific values. The two

strategies may be mixed freely.

>>> "{foo} {} {bar} {}".format(1, 2, bar=4, foo=3)

'3 1 4 2'

The indices of the unnamed arguments may also be used to request them out of order.

>>> "{foo} {1} {bar} {0}".format(1, 2, bar=4, foo=3)

'3 2 4 1'

Mixing manual and automatic field numbering is not permitted, however, as that could quickly get really

confusing.

But you don’t have to use the provided values themselves—you can access parts of them, just as in

ordinary Python code. Here’s an example:

>>> fullname = ["Alfred", "Smoketoomuch"]

>>> "Mr {name[1]}".format(name=fullname)

'Mr Smoketoomuch'

>>> import math

>>> tmpl = "The {mod.\_\_name\_\_} module defines the value {mod.pi} for π"

>>> tmpl.format(mod=math)

'The math module defines the value 3.141592653589793 for π'

As you can see, we can use both indexing and the dot notation for methods, attributes or variables, and

functions in imported modules. (The odd-looking \_\_name\_\_ variable contains the name of a given module.)

Basic Conversions

Once you’ve specified what a field should contain, you can add instructions on how to format it. First, you

can supply a conversion flag.

>>> print("{pi!s} {pi!r} {pi!a}".format(pi="π"))

π 'π' '\u03c0'

The three flags (s, r, and a) result in conversion using str, repr, and ascii, respectively. The str function

generally creates a natural-looking string version of the value (in this case, it does nothing to the input

string); the repr string tries to create a Python representation of the given value (in this case, a string literal),

while the ascii function insists on creating a representation that contains only character permitted in the

ASCII encoding. This is similar to how repr worked in Python 2.

You can also specify the type of value you are converting—or, rather, what kind of value you’d like it to

be treated as. For example, you may supply an integer but want to treat it as a decimal number. You do this

by using the f character (for fixed point) in the format specification, that is, after the colon separator.

>>> "The number is {num}".format(num=42)

'The number is 42'

>>> "The number is {num:f}".format(num=42)

'The number is 42.000000'

Or perhaps you’d rather format it as a binary numeral?

>>> "The number is {num:b}".format(num=42)

'The number is 101010'

There are several such type specifiers. For a list, see Table 3-1.

Table 3-1. String Formatting Type Specifiers

TypeMeaning

bFormats an integer as a binary numeral.

cInterprets an integer as a Unicode code point.

dFormats an integer as a decimal numeral. Default for integers.

eFormats a decimal number in scientific notation with e to indicate the exponent.

ESame as e, but uses E to indicate the exponent.

fFormats a decimal number with a fixed number of decimals.

FSame as f, but formats special values (nan and inf) in uppercase.

gChooses between fixed and scientific notation automatically. Default for decimal numbers,

except that the default version has at least one decimal.

GSame as g, but uppercases the exponent indicator and special values.

nSame as g, but inserts locale-dependent number separator characters.

oFormats an integer as an octal numeral.

sFormats a string as-is. Default for strings.

xFormats an integer as a hexadecimal numeral, with lowercase letters.

XSame as x, but with uppercase letters.

%Formats a number as a percentage (multiplied by 100, formatted by f, followed by %).

Width, Precision, and Thousands Separators

When formatting floating-point numbers (or other, more specialized decimal number types), the default

is to display six digits after the decimal point, and in all cases, the default is to let the formatted value have

exactly the width needed to display it, with no padding of any kind. These defaults may not be exactly what

you want, of course, and you can augment your format specification with details about width and precision

to suit your preferences.

The width is indicated by an integer, as follows:

>>> "{num:10}".format(num=3)

' 3'

>>> "{name:10}".format(name="Bob")

'Bob '

Numbers and strings are aligned differently, as you can see. We’ll get back to alignment in the next section.

Precision is also specified by an integer, but it’s preceded by a period, alluding to the decimal point.

>>> "Pi day is {pi:.2f}".format(pi=pi)

'Pi day is 3.14'

Here, I’ve explicitly specified the f type, because the default treats the precision a bit differently. (See the

Python Library Reference for the precise rules.) You can combine width and precision, of course.

>>> "{pi:10.2f}".format(pi=pi)

' 3.14'

You can actually use precision for other types as well, although you will probably not need that very often.

49

>>> "{:.5}".format("Guido van Rossum")

'Guido'

Finally, you can indicate that you want thousands separators, by using a comma.

>>> 'One googol is {:,}'.format(10\*\*100)

'One googol is 10,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,00

0,000,000,000,000,000,000,000,000,000,000,000,000,000,000'

When used in conjunction with the other formatting elements, this comma should come between the width

and the period indicating precision.1

Signs, Alignment, and Zero-Padding

Quite a bit of the formatting machinery is aimed at formatting numbers, for example, for printing out a table

of nicely aligned values. The width and precision get us most of the way there, but our pretty output may

still be thrown off if we include negative numbers. And, as you’ve seen, strings and numbers are aligned

differently; maybe we want to change that, for example, to include a piece of text in the middle of a column

of numbers? 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}'.format(pi)

'0000003.14'

You specify left, right and centered alignment with <, >, and ^, respectively.

>>> print('{0:<10.2f}\n{0:^10.2f}\n{0:>10.2f}'.format(pi))

3.14

3.14

3.14

You can augment the alignment specifier with a fill character, which is used instead of the space character.

>>> "{:$^15}".format(" WIN BIG ")

'$$$ WIN BIG $$$'

There’s also the more specialized specifier =, which places any fill characters between sign and digits.

>>> print('{0:10.2f}\n{1:10.2f}'.format(pi, -pi))

3.14

-3.14

>>> print('{0:10.2f}\n{1:=10.2f}'.format(pi, -pi))

3.14

- 3.14

If you want to include signs for positive numbers as well, you use the specifier + (after the alignment

specifier, if any), instead of the default -. If you use space character, positive will have a space inserted

instead of a +.

>>> print('{0:-.2}\n{1:-.2}'.format(pi, -pi)) # Default

3.1

-3.1

>>> print('{0:+.2}\n{1:+.2}'.format(pi, -pi))

+3.1

-3.1

>>> print('{0: .2}\n{1: .2}'.format(pi, -pi))

3.1

-3.1

One final component is the hash (#) option, which you place between the sign and width (if they are

present). This triggers an alternate form of conversion, with the details differing between types. For example,

for binary, octal, and hexadecimal conversion, a prefix is added.

>>> "{:b}".format(42)

'101010'

>>> "{:#b}".format(42)

'0b101010'

For various types of decimal numbers, it forces the inclusion of the decimal point (and for g, it keeps decimal

zeros).

>>> "{:g}".format(42)

'42'

>>> "{:#g}".format(42)

'42.0000'

In the example shown in Listing 3-1, I’ve used string formatting twice on the same strings—the first time

to insert the field widths into what is to become the eventual format specifiers. Because this information is

supplied by the user, I can’t hard-code the field widths.

Listing 3-1. String Formatting Example

# Print a formatted price list with a given width

width = int(input('Please enter width: '))

price\_width = 10

item\_width = width - price\_width

header\_fmt = '{{:{}}}{{:>{}}}'.format(item\_width, price\_width)

fmt = '{{:{}}}{{:>{}.2f}}'.format(item\_width, price\_width)

print('=' \* width)

print(header\_fmt.format('Item', 'Price'))

print('-' \* width)

print(fmt.format('Apples', 0.4))

print(fmt.format('Pears', 0.5))

print(fmt.format('Cantaloupes', 1.92))

print(fmt.format('Dried Apricots (16 oz.)', 8))

print(fmt.format('Prunes (4 lbs.)', 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 following are some useful

constants available from string2:

•string.digits: A string containing the digits 0–9

•string.ascii\_letters: A string containing all ASCII letters (uppercase and lowercase)

•string.ascii\_lowercase: A string containing all lowercase ASCII letters

•string.printable: A string containing all printable ASCII characters

•string.punctuation: A string containing all ASCII punctuation characters

•string.ascii\_uppercase: A string containing all uppercase ASCII letters

Despite explicitly dealing with ASCII characters, the values are actually (unencoded) Unicode strings.

center

The center method centers the string by padding it on either side with a given fill character—spaces by

default.

>>> "The Middle by Jimmy Eat World".center(39)

' The Middle by Jimmy Eat World '

>>> "The Middle by Jimmy Eat World".center(39, "\*")

'\*\*\*\*\*The Middle by Jimmy Eat World\*\*\*\*\*'

In Appendix B: ljust, rjust, zfill.

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

>>> 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: islower, istitle, isupper, translate.

In Appendix B: capitalize, casefold, swapcase, title, upper.

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: partition, rpartition, 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

contains information about which Unicode code points should be translated to which. You construct such a

table using the maketrans method on the string type str itself. The method takes two arguments: two strings

of equal length, where each character in the first string should be replaced by the character in the same

position in the second string.3 In the case of our simple example, the code would look like the following:

>>> table = str.maketrans('cs', 'kz')

We can peek inside the table if we wish, though all we’ll see is a mapping between Unicode code points.

>>> table

{115: 122, 99: 107}

Once you have a translation table, you can use it as an argument to the translate method.

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

'thiz iz an inkredible tezt'

An optional third argument can be supplied to maketrans, specifying letters that should be deleted. If you

wanted to emulate a really fast-talking German, for instance, you could delete all the spaces.

>>> table = str.maketrans('cs', 'kz', ' ')

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

'thizizaninkredibletezt'

See also: replace, lower.

Is My String …

There are plenty of string methods that start with is, such as isspace, isdigit, or isupper, that determine

whether your string has certain properties (such as being all whitespace, digits, or uppercase), in which case

the methods return True. Otherwise, of course, they return False.

In Appendix B: isalnum, isalpha, isdecimal, isdigit, isidentifier, islower, isnumeric, isprintable,

isspace, istitle, isupper.

A Quick Summary

In this chapter, you saw 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

FunctionDescription

string.capwords(s[, sep])Splits s with split (using sep), capitalizes items, and joins with a

single space

ascii(obj)Constructs an ASCII representation of the given object

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, we look at how dictionaries support not only integer

indices but other kinds of keys (such as strings or tuples) as well. They also have a few methods, though not

as many as strings.