

Extracted from:

Practical Programming, Third Edition
An Introduction to Computer Science Using Python 3.6

This PDF file contains pages extracted from *Practical Programming, Third Edition*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit <http://www.pragprog.com>.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2017 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The Pragmatic Bookshelf


Raleigh, North Carolina

The
Pragmatic
Programmers

Practical Programming

Third Edition

An Introduction to
Computer Science
Using Python 3.6



Paul Gries
Jennifer Campbell
Jason Montojo
edited by Tammy Coron

Practical Programming, Third Edition
An Introduction to Computer Science Using Python 3.6

Paul Gries
Jennifer Campbell
Jason Montojo

The Pragmatic Bookshelf

Raleigh, North Carolina



Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at <https://pragprog.com>.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Brian MacDonald
Supervising Editor: Jacquelyn Carter
Development Editor: Tammy Coron
Indexing: Potomac Indexing
Copy Editor: Liz Welch
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact support@pragprog.com.

For international rights, please contact rights@pragprog.com.

Copyright © 2017 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Printed in the United States of America.

ISBN-13: 978-1-6805026-8-8

Encoded using the finest acid-free high-entropy binary digits.

Book version: P1.0—December 2017

From email clients and web browsers to calendars and games, text plays a central role in computer programs. This chapter introduces a non-numeric data type that represents text, such as the words in this sentence or the sequence of bases in a strand of DNA. Along the way, we will see how to make programs a little more interactive by printing messages to our programs' users and getting input from them.

Creating Strings of Characters

Computers may have been invented to do arithmetic, but these days, most of them spend a lot of their time processing text. Many programs create text, store it, search it, and move it from one place to another.

In Python, text is represented as a *string*, which is a sequence of *characters* (letters, digits, and symbols). The type whose values are sequences of characters is `str`. The characters consist of those from the Latin alphabet found on most North American keyboards, as well as Chinese morphograms, chemical symbols, musical symbols, and much more.

In Python, we indicate that a value is a string by putting either single or double quotes around it. As we will see in [Using Special Characters in Strings, on page 8](#), single and double quotes are equivalent except for strings that contain quotes. You can use whichever you prefer. (For docstrings, the Python style guidelines say that double quotes are preferred.) Here are two examples:

```
>>> 'Aristotle'
'Aristotle'
>>> "Isaac Newton"
'Isaac Newton'
```

The opening and closing quotes must match:

```
>>> 'Charles Darwin"
      File "<stdin>", line 1
        'Charles Darwin"
            ^
```

SyntaxError: EOL while scanning string literal

EOL stands for “end of line.” The previous error indicates that the end of the line was reached before the end of the string (which should be marked with a closing single quote) was found.

Strings can contain any number of characters, limited only by computer memory. The shortest string is the *empty string*, containing no characters at all:

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

Operations on Strings

Python has a built-in function, `len`, that returns the number of characters between the opening and closing quotes:

```
>>> len('Albert Einstein')
15
>>> len('123!')
4
>>> len(' ')
1
>>> len('')
0
```

We can add two strings using the `+` operator, which produces a new string containing the same characters as in the two operands:

```
>>> 'Albert' + ' Einstein'
'Albert Einstein'
```

When `+` has two string operands, it is referred to as the *concatenation operator*. Operator `+` is probably the most overloaded operator in Python. So far, we've applied it to integers, floating-point numbers, and strings, and we'll apply it to several more types in later chapters.

As the following example shows, adding an empty string to another string produces a new string that is just like the nonempty operand:

```
>>> "Alan Turing" + ''
'Alan Turing'
>>> "" + 'Grace Hopper'
'Grace Hopper'
```

Here is an interesting question: Can operator `+` be applied to a string and a numeric value? If so, would addition or concatenation occur? We'll give it a try:

```
>>> 'NH' + 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

This is the second time that we have encountered a type error. The first time, in [Using Local Variables for Temporary Storage, on page ?](#), the problem was that we didn't pass the right number of parameters to a function. Here, Python took exception to our attempts to combine values of different data types because it didn't know which version of `+` we want: the one that adds numbers or the one that concatenates strings. Because the first operand was a string,

Python expected the second operand to also be a string but instead it was an integer. Now consider this example:

```
>>> 9 + ' planets'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Here, because Python saw a 9 first, it expected the second operand to also be numeric. The order of the operands affects the error message.

The concatenation operator must be applied to two strings. If you want to join a string with a number, you could apply function `str` to the number to get its string representation, and then apply the concatenation:

```
>>> 'Four score and ' + str(7) + ' years ago'
'Four score and 7 years ago'
```

Function `int` can be applied to a string whose contents look like an integer, and `float` can be applied to a string whose contents are numeric:

```
>>> int('0')
0
>>> int("11")
11
>>> int('-324')
-324
>>> float('-324')
-324.0
>>> float("56.34")
56.34
```

It isn't always possible to get an integer or a floating-point representation of a string, and when an attempt to do so fails, an error occurs:

```
>>> int('a')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'a'
>>> float('b')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: could not convert string to float: 'b'
```

In addition to `+`, `len`, `int`, and `float`, operator `*` can be applied to strings. A string can be repeated using operator `*` and an integer, like this:

```
>>> 'AT' * 5
'ATATATATAT'
>>> 4 * '-'
'----'
```

If the integer is less than or equal to zero, the operator yields the empty string:

```
>>> 'GC' * 0
''
>>> 'TATATATA' * -3
''
```

Strings are values, so you can assign a string to a variable. Also, operations on strings can be applied to those variables:

```
>>> sequence = 'ATTGTCCCC'
>>> len(sequence)
10
>>> new_sequence = sequence + 'GGCCTCCTGC'
>>> new_sequence
'ATTGTCCCCGGCCTCCTGC'
>>> new_sequence * 2
'ATTGTCCCCGGCCTCCTGCATTGTCCCCGGCCTCCTGC'
```

Using Special Characters in Strings

Suppose you want to put a single quote inside a string. If you write it directly, an error occurs:

```
>>> 'that's not going to work'
File "<stdin>", line 1
  'that's not going to work'
    ^
```

SyntaxError: invalid syntax

When Python encounters the second quote—the one that is intended to be part of the string—it thinks the string is ended. It doesn't know what to do with the text that comes after the second quote.

One simple way to fix this is to use double quotes around the string; we can also put single quotes around a string containing a double quote:

```
>>> "that's better"
"that's better"
>>> 'She said, "That is better."'
'She said, "That is better."'
```

If you need to put a double quote in a string, you can use single quotes around the string. But what if you want to put both kinds of quote in one string? You could do this:

```
>>> 'She said, "That' + "'" + 's hard to read."'
'She said, "That\'s hard to read."'
```

The result is a valid Python string. The backslash is called an *escape character*, and the combination of the backslash and the single quote is called an *escape sequence*. The name comes from the fact that we’re “escaping” from Python’s usual syntax rules for a moment. When Python sees a backslash inside a string, it means that the next character represents something that Python normally uses for other purposes, such as marking the end of a string.

The escape sequence `'` is indicated using two symbols, but those two symbols represent a single character:

```
>>> len('\')
1
>>> len('it\'s')
4
```

Python recognizes several escape sequences. Here are some common ones:

Escape Sequence	Description
<code>'</code>	Single quote
<code>"</code>	Double quote
<code>\</code>	Backslash
<code>\t</code>	Tab
<code>\n</code>	Newline
<code>\r</code>	Carriage return

Table 4—Escape Sequences

In order to see how they are used, we will introduce multiline strings and also revisit built-in function `print`.

Creating a Multiline String

If you create a string using single or double quotes, the whole string must fit onto a single line.

Here's what happens when you try to stretch a string across multiple lines:

```
>>> 'one
File "<stdin>", line 1
  'one
  ^
```

SyntaxError: EOL while scanning string literal

As we saw in [Creating Strings of Characters, on page 5](#), EOL stands for “end of line”. So in this error report, Python is saying that it reached the end of the line before it found the end of the string.

To span multiple lines, put three single quotes or three double quotes around the string instead of one. The string can then span as many lines as you want:

```
>>> '''one
... two
... three'''
'one\ntwo\nthree'
```

Notice that the string Python creates contains a `\n` sequence everywhere our input started a new line. Each newline is a character in the string.

Normalizing Line Endings

Each of the three major operating systems uses a different set of characters to indicate the end of a line. This set of characters is called a *newline*. On Linux and macOS, a newline is one `\n` character; on version 9 and earlier of Mac OS, it is one `\r`; and on Windows, the ends of lines are marked with both characters as `\r\n`.

Python always uses a single `\n` to indicate a newline, even on operating systems like Windows that do things other ways. This is called *normalizing* the string; Python does this so that you can write exactly the same program no matter what kind of machine you're running on.