

PYTHON CRASH COURSE

3RD EDITION

**A Hands-On, Project-Based
Introduction to Programming**

by Eric Matthes



**no starch
press**

San Francisco

PYTHON CRASH COURSE, 3RD EDITION. Copyright © 2023 by Eric Matthes.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

First printing

26 25 24 23 22 1 2 3 4 5

ISBN-13: 978-1-7185-0270-3 (print)

ISBN-13: 978-1-7185-0271-0 (ebook)

Publisher: William Pollock

Managing Editor: Jill Franklin

Production Editor: Jennifer Kepler

Developmental Editor: Eva Morrow

Cover Illustrator: Josh Ellingson

Interior Design: Octopod Studios

Technical Reviewer: Kenneth Love

Copyeditor: Doug McNair

Compositor: Jeff Lytle, Happenstance Type-O-Rama

Proofreader: Scout Festa

For information on distribution, bulk sales, corporate sales, or translations, please contact No Starch Press, Inc. directly at info@nostarch.com or:

No Starch Press, Inc.

245 8th Street, San Francisco, CA 94103

phone: 1.415.863.9900

www.nostarch.com

The Library of Congress has catalogued the first edition as follows:

Matthes, Eric, 1972-

Python crash course : a hands-on, project-based introduction to programming / by Eric Matthes.

pages cm

Includes index.

Summary: "A project-based introduction to programming in Python, with exercises. Covers general programming concepts, Python fundamentals, and problem solving. Includes three projects - how to create a simple video game, use data visualization techniques to make graphs and charts, and build an interactive web application"-- Provided by publisher.

ISBN 978-1-59327-603-4 -- ISBN 1-59327-603-6

1. Python (Computer program language) I. Title.

QA76.73.P98M38 2015

005.13'3--dc23

2015018135

No Starch Press and the No Starch Press logo are registered trademarks of No Starch Press, Inc. Other product and company names mentioned herein may be the trademarks of their respective owners. Rather than use a trademark symbol with every occurrence of a trademarked name, we are using the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The information in this book is distributed on an "As Is" basis, without warranty. While every precaution has been taken in the preparation of this work, neither the author nor No Starch Press, Inc. shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in it.

This distinction probably won't matter much in your initial programs, but it's worth learning earlier rather than later. At some point, you'll see unexpected behavior from a variable, and an accurate understanding of how variables work will help you identify what's happening in your code.

NOTE

The best way to understand new programming concepts is to try using them in your programs. If you get stuck while working on an exercise in this book, try doing something else for a while. If you're still stuck, review the relevant part of that chapter. If you still need help, see the suggestions in Appendix C.

TRY IT YOURSELF

Write a separate program to accomplish each of these exercises. Save each program with a filename that follows standard Python conventions, using lowercase letters and underscores, such as `simple_message.py` and `simple_messages.py`.

2-1. Simple Message: Assign a message to a variable, and then print that message.

2-2. Simple Messages: Assign a message to a variable, and print that message. Then change the value of the variable to a new message, and print the new message.

Strings

Because most programs define and gather some sort of data and then do something useful with it, it helps to classify different types of data. The first data type we'll look at is the string. Strings are quite simple at first glance, but you can use them in many different ways.

A *string* is a series of characters. Anything inside quotes is considered a string in Python, and you can use single or double quotes around your strings like this:

```
"This is a string."  
'This is also a string.'
```

This flexibility allows you to use quotes and apostrophes within your strings:

```
'I told my friend, "Python is my favorite language!'"  
"The language 'Python' is named after Monty Python, not the snake."  
"One of Python's strengths is its diverse and supportive community."
```

Let's explore some of the ways you can use strings.

Changing Case in a String with Methods

One of the simplest tasks you can do with strings is change the case of the words in a string. Look at the following code, and try to determine what's happening:

```
name.py name = "ada lovelace"  
print(name.title())
```

Save this file as *name.py* and then run it. You should see this output:

```
Ada Lovelace
```

In this example, the variable *name* refers to the lowercase string "ada lovelace". The method `title()` appears after the variable in the `print()` call. A *method* is an action that Python can perform on a piece of data. The dot (`.`) after *name* in `name.title()` tells Python to make the `title()` method act on the variable *name*. Every method is followed by a set of parentheses, because methods often need additional information to do their work. That information is provided inside the parentheses. The `title()` function doesn't need any additional information, so its parentheses are empty.

The `title()` method changes each word to title case, where each word begins with a capital letter. This is useful because you'll often want to think of a name as a piece of information. For example, you might want your program to recognize the input values *Ada*, *ADA*, and *ada* as the same name, and display all of them as *Ada*.

Several other useful methods are available for dealing with case as well. For example, you can change a string to all uppercase or all lowercase letters like this:

```
name = "Ada Lovelace"  
print(name.upper())  
print(name.lower())
```

This will display the following:

```
ADA LOVELACE  
ada lovelace
```

The `lower()` method is particularly useful for storing data. You typically won't want to trust the capitalization that your users provide, so you'll convert strings to lowercase before storing them. Then when you want to display the information, you'll use the case that makes the most sense for each string.

Using Variables in Strings

In some situations, you'll want to use a variable's value inside a string. For example, you might want to use two variables to represent a first name and

a last name, respectively, and then combine those values to display someone's full name:

```
full_name.py first_name = "ada"
last_name = "lovelace"
❶ full_name = f"{first_name} {last_name}"
print(full_name)
```

To insert a variable's value into a string, place the letter `f` immediately before the opening quotation mark ❶. Put braces around the name or names of any variable you want to use inside the string. Python will replace each variable with its value when the string is displayed.

These strings are called *f-strings*. The *f* is for *format*, because Python formats the string by replacing the name of any variable in braces with its value. The output from the previous code is:

```
ada lovelace
```

You can do a lot with f-strings. For example, you can use f-strings to compose complete messages using the information associated with a variable, as shown here:

```
first_name = "ada"
last_name = "lovelace"
full_name = f"{first_name} {last_name}"
❶ print(f"Hello, {full_name.title()}!")
```

The full name is used in a sentence that greets the user ❶, and the `title()` method changes the name to title case. This code returns a simple but nicely formatted greeting:

```
Hello, Ada Lovelace!
```

You can also use f-strings to compose a message, and then assign the entire message to a variable:

```
first_name = "ada"
last_name = "lovelace"
full_name = f"{first_name} {last_name}"
❶ message = f"Hello, {full_name.title()}!"
❷ print(message)
```

This code displays the message `Hello, Ada Lovelace!` as well, but by assigning the message to a variable ❶ we make the final `print()` call much simpler ❷.

Adding Whitespace to Strings with Tabs or Newlines

In programming, *whitespace* refers to any nonprinting characters, such as spaces, tabs, and end-of-line symbols. You can use whitespace to organize your output so it's easier for users to read.

To add a tab to your text, use the character combination `\t`:

```
>>> print("Python")
Python
>>> print("\tPython")
    Python
```

To add a newline in a string, use the character combination `\n`:

```
>>> print("Languages:\nPython\nC\nJavaScript")
Languages:
Python
C
JavaScript
```

You can also combine tabs and newlines in a single string. The string `"\n\t"` tells Python to move to a new line, and start the next line with a tab. The following example shows how you can use a one-line string to generate four lines of output:

```
>>> print("Languages:\n\tPython\n\tC\n\tJavaScript")
Languages:
    Python
    C
    JavaScript
```

Newlines and tabs will be very useful in the next two chapters, when you start to produce many lines of output from just a few lines of code.

Stripping Whitespace

Extra whitespace can be confusing in your programs. To programmers, `'python'` and `'python '` look pretty much the same. But to a program, they are two different strings. Python detects the extra space in `'python '` and considers it significant unless you tell it otherwise.

It's important to think about whitespace, because often you'll want to compare two strings to determine whether they are the same. For example, one important instance might involve checking people's usernames when they log in to a website. Extra whitespace can be confusing in much simpler situations as well. Fortunately, Python makes it easy to eliminate extra whitespace from data that people enter.

Python can look for extra whitespace on the right and left sides of a string. To ensure that no whitespace exists at the right side of a string, use the `rstrip()` method:

```
❶ >>> favorite_language = 'python '
❷ >>> favorite_language
'python '
❸ >>> favorite_language.rstrip()
'python'
❹ >>> favorite_language
'python '
```

The value associated with `favorite_language` ❶ contains extra whitespace at the end of the string. When you ask Python for this value in a terminal session, you can see the space at the end of the value ❷. When the `rstrip()` method acts on the variable `favorite_language` ❸, this extra space is removed. However, it is only removed temporarily. If you ask for the value of `favorite_language` again, the string looks the same as when it was entered, including the extra whitespace ❹.

To remove the whitespace from the string permanently, you have to associate the stripped value with the variable name:

```
>>> favorite_language = 'python '
❶ >>> favorite_language = favorite_language.rstrip()
>>> favorite_language
'python'
```

To remove the whitespace from the string, you strip the whitespace from the right side of the string and then associate this new value with the original variable ❶. Changing a variable's value is done often in programming. This is how a variable's value can be updated as a program is executed or in response to user input.

You can also strip whitespace from the left side of a string using the `lstrip()` method, or from both sides at once using `strip()`:

```
❶ >>> favorite_language = ' python '
❷ >>> favorite_language.rstrip()
' python'
❸ >>> favorite_language.lstrip()
'python '
❹ >>> favorite_language.strip()
'python'
```

In this example, we start with a value that has whitespace at the beginning and the end ❶. We then remove the extra space from the right side ❷, from the left side ❸, and from both sides ❹. Experimenting with these stripping functions can help you become familiar with manipulating strings. In the real world, these stripping functions are used most often to clean up user input before it's stored in a program.

Removing Prefixes

When working with strings, another common task is to remove a prefix. Consider a URL with the common prefix `https://`. We want to remove this prefix, so we can focus on just the part of the URL that users need to enter into an address bar. Here's how to do that:

```
>>> nostarch_url = 'https://nostarch.com'
>>> nostarch_url.removeprefix('https://')
'nostarch.com'
```

Enter the name of the variable followed by a dot, and then the method `removeprefix()`. Inside the parentheses, enter the prefix you want to remove from the original string.

Like the methods for removing whitespace, `removeprefix()` leaves the original string unchanged. If you want to keep the new value with the prefix removed, either reassign it to the original variable or assign it to a new variable:

```
>>> simple_url = nostarch_url.removeprefix('https://')
```

When you see a URL in an address bar and the `https://` part isn't shown, the browser is probably using a method like `removeprefix()` behind the scenes.

Avoiding Syntax Errors with Strings

One kind of error that you might see with some regularity is a syntax error. A *syntax error* occurs when Python doesn't recognize a section of your program as valid Python code. For example, if you use an apostrophe within single quotes, you'll produce an error. This happens because Python interprets everything between the first single quote and the apostrophe as a string. It then tries to interpret the rest of the text as Python code, which causes errors.

Here's how to use single and double quotes correctly. Save this program as *apostrophe.py* and then run it:

```
apostrophe.py message = "One of Python's strengths is its diverse community."  
print(message)
```

The apostrophe appears inside a set of double quotes, so the Python interpreter has no trouble reading the string correctly:

```
One of Python's strengths is its diverse community.
```

However, if you use single quotes, Python can't identify where the string should end:

```
message = 'One of Python's strengths is its diverse community.'  
print(message)
```

You'll see the following output:

```
File "apostrophe.py", line 1  
  message = 'One of Python's strengths is its diverse community.'  
                                     ❶ ^  
SyntaxError: unterminated string literal (detected at line 1)
```

In the output you can see that the error occurs right after the final single quote ❶. This syntax error indicates that the interpreter doesn't recognize

something in the code as valid Python code, and it thinks the problem might be a string that's not quoted correctly. Errors can come from a variety of sources, and I'll point out some common ones as they arise. You might see syntax errors often as you learn to write proper Python code. Syntax errors are also the least specific kind of error, so they can be difficult and frustrating to identify and correct. If you get stuck on a particularly stubborn error, see the suggestions in Appendix C.

NOTE

Your editor's syntax highlighting feature should help you spot some syntax errors quickly as you write your programs. If you see Python code highlighted as if it's English or English highlighted as if it's Python code, you probably have a mismatched quotation mark somewhere in your file.

TRY IT YOURSELF

Save each of the following exercises as a separate file, with a name like `name_cases.py`. If you get stuck, take a break or see the suggestions in Appendix C.

2-3. Personal Message: Use a variable to represent a person's name, and print a message to that person. Your message should be simple, such as, "Hello Eric, would you like to learn some Python today?"

2-4. Name Cases: Use a variable to represent a person's name, and then print that person's name in lowercase, uppercase, and title case.

2-5. Famous Quote: Find a quote from a famous person you admire. Print the quote and the name of its author. Your output should look something like the following, including the quotation marks:

Albert Einstein once said, "A person who never made a mistake never tried anything new."

2-6. Famous Quote 2: Repeat Exercise 2-5, but this time, represent the famous person's name using a variable called `famous_person`. Then compose your message and represent it with a new variable called `message`. Print your message.

2-7. Stripping Names: Use a variable to represent a person's name, and include some whitespace characters at the beginning and end of the name. Make sure you use each character combination, `"\t"` and `"\n"`, at least once.

Print the name once, so the whitespace around the name is displayed. Then print the name using each of the three stripping functions, `lstrip()`, `rstrip()`, and `strip()`.

2-8. File Extensions: Python has a `removesuffix()` method that works exactly like `removeprefix()`. Assign the value `'python_notes.txt'` to a variable called `filename`. Then use the `removesuffix()` method to display the filename without the file extension, like some file browsers do.