learn enough PYTHON TO BE DANGEROUS

SOFTWARE DEVELOPMENT,
FLASK WEB APPS,
AND BEGINNING
DATA SCIENCE
WITH PYTHON

MICHAEL HARTL

Praise for Learn Enough Tutorials

"Just started the #100DaysOfCode journey. Today marks day 1. I have completed @mhartl's great Ruby tutorial at @LearnEnough and am looking forward to starting on Ruby on Rails from tomorrow. Onwards and upwards."

—Optimize Prime (@_optimize), Twitter post

"Ruby and Sinatra and Heroku, oh my! Almost done with this live web application. It may be a simple palindrome app, but it's also simply exciting! #100DaysOfCode #ruby @LearnEnough #ABC #AlwaysBeCoding #sinatra #heroku"

—Tonia Del Priore (@toninjaa), Twitter post; Software Engineer for a FinTech Startup for 3+ years

"I have nothing but fantastic things to say about @LearnEnough courses. I am just about finished with the #javascript course. I must say, the videos are mandatory because @mhartl will play the novice and share in the joy of having something you wrote actually work!"

—Claudia Vizena

"I must say, this Learn Enough series is a masterpiece of education. Thank you for this incredible work!"

—Michael King

"I want to thank you for the amazing job you have done with the tutorials. They are likely the best tutorials I have ever read."

—Pedro Iatzky

This page intentionally left blank

Learn Enough Python to Be Dangerous

Learn Enough Python to Be Dangerous

Software Development, Flask Web Apps, and Beginning Data Science with Python

Michael Hartl

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Cover image: Alexey Boldin/Shutterstock
Figure 1.4: The Pallets Projects
Figures 1.6-1.8, 9.3: Amazon Web Services, Inc.
Figures 1.9, 1.10, 8.2: GitHub, Inc.
Figures 1.11, 1.12, 10.2, 10.3: Fly.io
Figure 2.9: Python Software Foundation
Figures 4.4, 4.5, 4.10, 8.8: Regex101
Figures 5.6, 9.5: Google LLC
Figures 9.4, 9.7: The Wikimedia Foundation
Figures 11.1-11.5, 11.23: Jupyter

Library of Congress Control Number: 2023935869

ISBN-13: 978-0-13-805095-5
ISBN-10: 0-13-805095-3

ScoutAutomatedPrintCode

Contents

Preface

Learn Enough Python to Be Dangerous teaches you to write practical and modern programs using the elegant and powerful Python programming language. You'll learn how to use Python for both general-purpose programming and for beginning web-application development. Although mastering Python can be a long journey, you don't have to learn everything to get started . . . you just have to learn enough to be *dangerous*.

You'll begin by exploring the core concepts of Python programming using a combination of the interactive Python interpreter and text files run at the command line. The result is a solid understanding of both *object-oriented programming* and *functional programming* in Python. You'll then build on this foundation to develop and publish a simple self-contained Python package. You'll then use this package in a simple dynamic web application built using the *Flask* web framework, which you'll also deploy to the live Web. As a result, *Learn Enough Python to Be Dangerous* is especially appropriate as a prerequisite to learning web development with Python.

In addition to teaching you specific skills, *Learn Enough Python to Be Dangerous* also helps you develop *technical sophistication*—the seemingly magical ability to solve practically any technical problem. Technical sophistication includes concrete skills like version control and coding, as well as fuzzier skills like Googling the error message and knowing when to just reboot the darn thing. Throughout *Learn Enough Python to Be Dangerous*, we'll have abundant opportunities to develop technical sophistication in the context of real-world examples.

Chapter by Chapter

In order to learn enough Python to be dangerous, we'll begin at the beginning with a series of simple "hello, world" programs using several different techniques (Chapter 1), including an introduction to the Python interpreter, an interactive command-line program for evaluating Python code. In line with the Learn Enough philosophy of always doing things "for real", even as early as Chapter 1 we'll deploy a (very simple) dynamic Python application to the live Web. This chapter also includes pointers to the latest setup and installation instructions via *Learn Enough Dev Environment to Be Dangerous* (https://www.learnenough.com/dev-environment), which is available for free online and as a free downloadable ebook.

After mastering "hello, world", we'll take a tour of some Python *objects*, including strings (Chapter 2), lists (Chapter 3), and other native objects like dates, dictionaries, and regular expressions (Chapter 4). Taken together, these chapters constitute a gentle introduction to *object-oriented programming* with Python.

In Chapter 5, we'll learn the basics of *functions*, an essential subject for virtually every programming language. We'll then apply this knowledge to an elegant and powerful style of coding known as *functional programming*, including an introduction to *comprehensions* (Chapter 6).

Having covered the basics of built-in Python objects, in Chapter 7 we'll learn how to make objects of our own. In particular, we'll define an object for a *phrase*, and then develop a method for determining whether or not the phrase is a *palindrome* (the same read forward and backward).

Our initial palindrome implementation will be rather rudimentary, but we'll extend it in Chapter 8 using a powerful technique called *test-driven development* (TDD). In the process, we'll learn more about testing generally, as well as how to create and publish a Python package.

In Chapter 9, we'll learn how to write nontrivial *shell scripts*, one of Python's biggest strengths. Examples include reading from both files and URLs, with a final example showing how to manipulate a downloaded file as if it were an HTML web page.

In Chapter 10, we'll develop our first full Python web application: a site for detecting palindromes. This will give us a chance to learn about *routes*, *layouts*, *embedded Python*, and *form handling*, together with a second application of TDD. As a capstone to our work, we'll deploy our palindrome detector to the live Web.

Finally, in Chapter 11, we'll get an introduction to Python tools used in the booming field of *data science*. Topics include numerical calculations with NumPy, data visualization with Matplotlib, data analysis with pandas, and machine learning with scikit-learn.

Additional Features

In addition to the main tutorial material, *Learn Enough Python to Be Dangerous* includes a large number of exercises to help you test your understanding and to extend the material in the main text. The exercises include frequent hints and often include the expected answers, with community solutions available by separate subscription at www.learnenough.com.

Final Thoughts

Learn Enough Python to Be Dangerous gives you a practical introduction to the fundamentals of Python, both as a general-purpose programming language and as a specialist language for web development and data science. After learning the techniques covered in this tutorial, and especially after developing your technical sophistication, you'll know everything you need to write shell scripts, publish Python packages, deploy dynamic web applications, and use key Python data-science tools. You'll also be ready for a huge variety of other resources, including books, blog posts, and online documentation.

Learn Enough Scholarships

Learn Enough is committed to making a technical education available to as wide a variety of people as possible. As part of this commitment, in 2016 we created the Learn Enough Scholarship program.[1] Scholarship recipients get free or deeply discounted access to the Learn Enough All Access subscription, which includes all of the Learn Enough online book content, embedded videos, exercises, and community exercise answers.

As noted in a 2019 RailsConf Lightning Talk,[2] the Learn Enough Scholarship application process is incredibly simple: Just fill out a confidential text area telling us a little about your situation. The scholarship criteria are generous and flexible—we understand that there are an enormous number of reasons for wanting a scholarship, from being a student, to being between jobs, to living in a country with an unfavorable

1. https://www.learnenough.com/scholarship
2. https://www.learnenough.com/scholarship-talk

exchange rate against the U.S. dollar. Chances are that, if you feel like you've got a good reason, we'll think so, too.

So far, Learn Enough has awarded more than 2,500 scholarships to aspiring developers around the country and around the world. To apply, visit the Learn Enough Scholarship page at www.learnenough.com/scholarship. Maybe the next scholarship recipient could be you!

Acknowledgments

Thanks to Paul Logston, Tom Repetti, and Ron Lee for their helpful comments on drafts of *Learn Enough Python to Be Dangerous*. Thanks also to Prof. Jetson Leder-Luis of Boston University and data scientist Amadeo Bellotti for their helpful feedback and assistance in preparing Chapter 11. Any errors that remain in the text are entirely the fault of these fine gentlemen.

As always, thanks to Debra Williams Cauley for shepherding the production process at Pearson.

About the Author

Michael Hartl (www.michaelhartl.com) is the creator of the *Ruby on Rails*™ *Tutorial* (www.railstutorial.org), one of the leading introductions to web development, and is cofounder and principal author at Learn Enough (www.learnenough.com). Previously, he was a physics instructor at the California Institute of Technology (Caltech), where he received a Lifetime Achievement Award for Excellence in Teaching. He is a graduate of Harvard College, has a Ph.D. in Physics from Caltech, and is an alumnus of the Y Combinator entrepreneur program.

This page intentionally left blank

CHAPTER 8

Testing and Test-Driven Development

Although rarely covered in introductory programming tutorials, *automated testing* is one of the most important subjects in modern software development. Accordingly, this chapter includes an introduction to testing in Python, including a first look at *test-driven development*, or TDD.

Test-driven development came up briefly in Section 6.5.1, which promised that we would use testing techniques to add an important capability to finding palindromes, namely, being able to detect complicated palindromes such as "A man, a plan, a canal—Panama!" (Figure 6.5) or "Madam, I'm Adam." (Figure 8.1[1]). This chapter fulfills that promise.

As it turns out, learning how to write Python tests will also give us a chance to learn how to create (and publish!) a Python package, another exceptionally useful Python skill rarely covered in introductory tutorials.

Here's our strategy for testing the current palindrome code and extending it to more complicated phrases:

1. Set up our initial package (Section 8.1).
2. Write automated tests for the existing **`ispalindrome()`** functionality (Section 8.2).
3. Write a *failing* test for the enhanced palindrome detector (RED) (Section 8.3).

1. "The Temptation of Adam" by Tintoretto. Image courtesy of Album/Alamy Stock Photo.

Figure 8.1: The Garden of Eden had it all—even palindromes.

4. Write (possibly ugly) code to get the test *passing* (GREEN) (Section 8.4).
5. *Refactor* the code to make it prettier, while ensuring that the test suite stays GREEN (Section 8.5).

8.1 Package Setup

We saw as early as Section 1.5 that the Python ecosystem includes a large number of self-contained software packages. In this section, we'll create a package based on the palindrome detector developed in Chapter 7. As part of this, we'll set up the beginnings of a *test suite* to test our code.

Python packages have a standard structure that can be visualized as shown in Listing 8.1 (which contains both generic elements like **`pyproject.toml`** and non-generic elements like **`palindrome_YOUR_USERNAME_HERE`**). The structure includes some configuration files (discussed in just a moment) and two directories: a **`src`** (source) directory and a **`tests`** directory. The **`src`** directory in turn contains a directory for the palindrome package, which includes a special required file called **`__init__.py`** and the **`palindrome_YOUR_USERNAME_HERE`** module itself.[2] (It is possible to flatten the directory structure by eliminating the package directory, but the structure in Listing 8.1 is fairly standard and is designed to mirror the official Packaging Python Projects documentation.) The result of the structure in Listing 8.1 will be the ability to include the **`Phrase`** class developed in Chapter 7 using the code

```
from palindrome_mhartl.phrase import Phrase
```

Listing 8.1: File and directory structure for a sample Python package.

```
python_package_tutorial/
├── LICENSE
├── pyproject.toml
├── README.md
├── src/
│   └── palindrome_YOUR_USERNAME_HERE/
│       ├── __init__.py
│       └── phrase.py
└── tests/
    └── test_phrase.py
```

We can create the structure in Listing 8.1 by hand using a combination of **`mkdir`** and **`touch`**, as shown in Listing 8.2.

Listing 8.2: Setting up a Python package.

```
$ cd ~/repos    # Use ~/environment/repos on Cloud9
$ mkdir python_package_tutorial
$ cd python_package_tutorial
$ touch LICENSE pyproject.toml README.md
$ mkdir -p src/palindrome_YOUR_USERNAME_HERE
$ touch src/palindrome_YOUR_USERNAME_HERE/__init__.py
$ touch src/palindrome_YOUR_USERNAME_HERE/phrase.py
$ mkdir tests
$ touch tests/test_phrase.py
```

2. Technically, there are various distinctions between *packages* and *modules* in Python, but they are rarely important. See this Stack Overflow comment (https://stackoverflow.com/questions/7948494/whats-the-difference-between-a-python-module-and-a-python-package/49420164#49420164) for some of the minutiae on the subject.

At this point, we'll fill in a few of the files with more information, including the project configuration file **`pyproject.toml`** (Listing 8.3), a README file **`README.md`** (Listing 8.4), and a **`LICENSE`** file (Listing 8.5).[3] Some of these files are only templates, so you should replace things like **`<username>`** in **`pyproject.toml`** with your own username, the **`url`** field with the planned URL for your project, etc. (Being able to do things like this is an excellent application of technical sophistication.) To see a concrete example of the files in this section, see the GitHub repo (https://github.com/mhartl/python_package_tutorial) for my version of this package.

Listing 8.3: The project configuration for a Python package.
~/python_package_tutorial/project.toml

```
[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"

[project]
name = "example_package_YOUR_USERNAME_HERE"
version = "0.0.1"
authors = [
  { name="Example Author", email="author@example.com" },
]
description = "A small example package"
readme = "README.md"
requires-python = ">=3.7"
classifiers = [
    "Programming Language :: Python :: 3",
    "License :: OSI Approved :: MIT License",
    "Operating System :: OS Independent",
]

[project.urls]
"Homepage" = "https://github.com/pypa/sampleproject"
"Bug Tracker" = "https://github.com/pypa/sampleproject/issues"
```

3. Don't worry about the details of files like **`pyproject.toml`**; I don't understand them either. I just copied them from the documentation (Box 1.2).

Listing 8.4: A README file for the package.
~/python_package_tutorial/README.md

```
# Palindrome Package

This is a sample Python package for
[*Learn Enough Python to Be Dangerous*](https://www.learnenough.com/python)
by [Michael Hartl](https://www.michaelhartl.com/).
```

Listing 8.5: A license template for a Python package.
~/python_package_tutorial/LICENSE

```
Copyright (c) YYYY Your Name

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

With all that configuration done, we're now ready to configure the environment for development and testing. As in Section 1.3, we'll use **`venv`** for the virtual environment. We'll also be using **`pytest`** for testing, which we can install using **`pip`**. The resulting commands are shown in Listing 8.6.

Listing 8.6: Setting up the package environment (including testing).

```
$ deactivate    # just in case a virtual env is already active
$ python3 -m venv venv
$ source venv/bin/activate
(venv) $ pip install --upgrade pip
(venv) $ pip install pytest==7.1.3
```

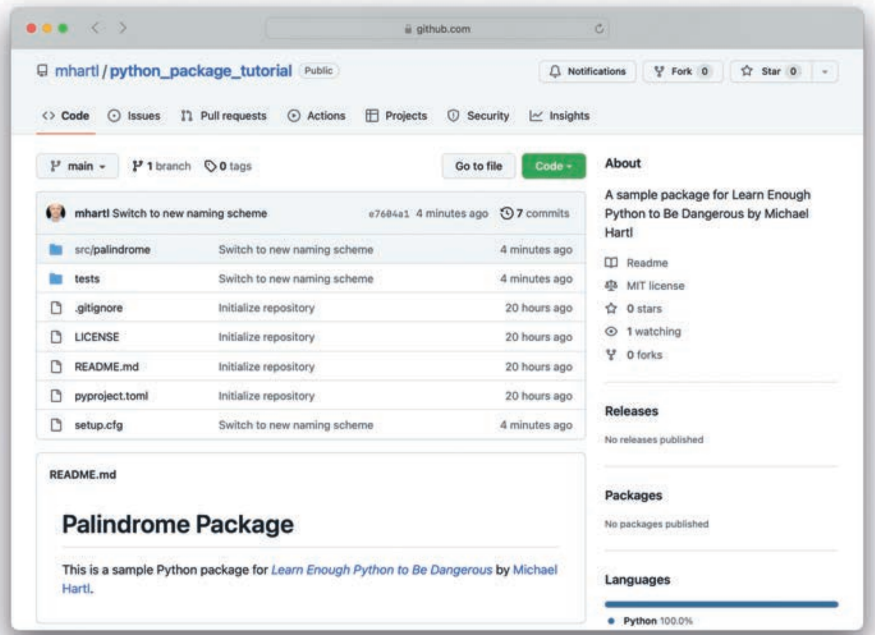
At this point, as in Section 1.5.1, it's a good idea to create a **`.gitignore`** file (Listing 8.7), put the project under version control with Git (Listing 8.8), and create a repository at GitHub (Figure 8.2). This last step will also give you URLs for the configuration file in Listing 8.3.

Listing 8.7: Ignoring certain files and directories.
`.gitignore`

```
venv/

*.pyc
__pycache__/

instance/

.pytest_cache/
.coverage
htmlcov/

dist/
build/
*.egg-info/

.DS_Store
```

Listing 8.8: Initializing the package repository.

```
$ git init
$ git add -A
$ git commit -m "Initialize repository"
```



mhartl / python_package_tutorial Public

1 branch 0 tags

mhartl Switch to new naming scheme e7684a1 4 minutes ago 7 commits

src/palindrome	Switch to new naming scheme	4 minutes ago
tests	Switch to new naming scheme	4 minutes ago
.gitignore	Initialize repository	20 hours ago
LICENSE	Initialize repository	20 hours ago
README.md	Initialize repository	20 hours ago
pyproject.toml	Initialize repository	20 hours ago
setup.cfg	Switch to new naming scheme	4 minutes ago

README.md

Palindrome Package

This is a sample Python package for *Learn Enough Python to Be Dangerous* by Michael Hartl.

About

A sample package for Learn Enough Python to Be Dangerous by Michael Hartl

Readme
MIT license
0 stars
1 watching
0 forks

Releases

No releases published

Packages

No packages published

Languages

Python 100.0%

Figure 8.2: The package repository and README at GitHub.

8.1.1 Exercise

1. If you haven't already, update Listing 8.3 with the right package name and fill the `url` and `Bug Tracker` fields with the corresponding GitHub URLs (the tracker URL is just the base URL plus `/issues`). Likewise, update the license template in Listing 8.5 with your name and the current year. Commit and push your changes up to GitHub.

8.2 Initial Test Coverage

Now that we've set up our basic package structure, we're ready to get started testing. Because the necessary `pytest` package has already been installed (Listing 8.6), we can actually run the (nonexistent) tests immediately:

```
(venv) $ pytest
============================= test session starts ==============================
platform darwin -- Python 3.10.6, pytest-7.1.3, pluggy-1.0.0
rootdir: /Users/mhartl/repos/python_package_tutorial
collected 0 items

============================ no tests ran in 0.00s =============================
```

Exact details will differ (and will be omitted in future examples for that reason), but your results should be similar.

Now let's write a minimal failing test and then get it to pass. Because we've already created a **tests** directory with the test file **test_phrase.py** (Listing 8.2), we can begin by adding the code shown in Listing 8.9.

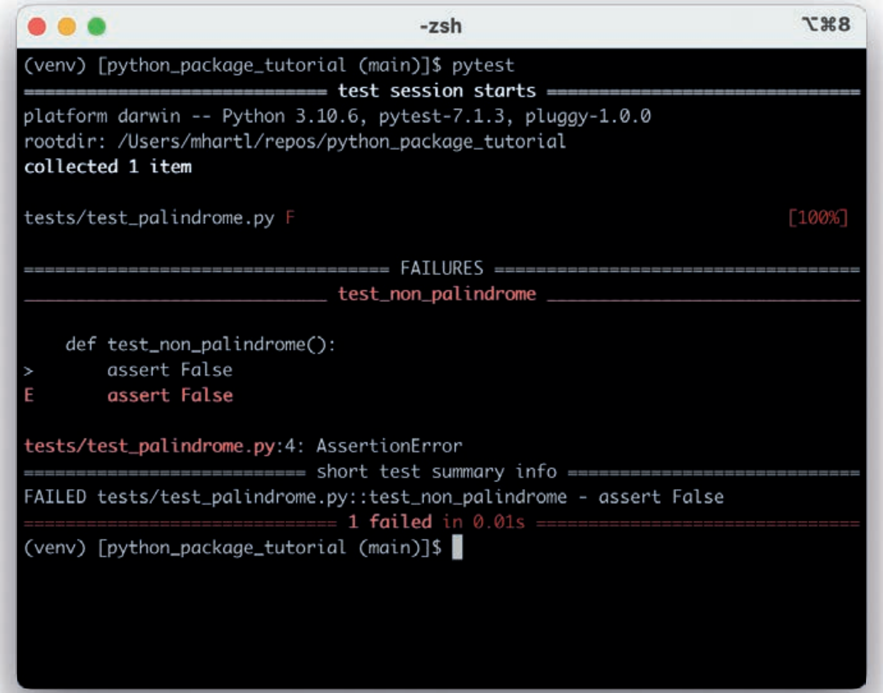
Listing 8.9: The initial test suite. RED
test/test_phrase.py

```
def test_initial_example():
    assert False
```

Listing 8.9 defines a function containing one *assertion*, which asserts that something has a boolean value of **True**, in which case the assertion passes, and fails otherwise. Because Listing 8.9 literally asserts that **False** is **True**, it fails by design:

Listing 8.10: RED

```
(venv) $ pytest
============================= test session starts ==============================
collected 1 item

tests/palindrome_test.py F                                              [100%]

=================================== FAILURES ===================================
_____________________________ test_non_palindrome ______________________________

    def test_non_palindrome():
>       assert False
E       assert False

tests/palindrome_test.py:4: AssertionError
=========================== short test summary info ============================
FAILED tests/palindrome_test.py::test_non_palindrome - assert False
============================== 1 failed in 0.01s ===============================
```



```
(venv) [python_package_tutorial (main)]$ pytest
============================= test session starts ==============================
platform darwin -- Python 3.10.6, pytest-7.1.3, pluggy-1.0.0
rootdir: /Users/mhartl/repos/python_package_tutorial
collected 1 item

tests/test_palindrome.py F                                               [100%]

=================================== FAILURES ===================================
______________________________ test_non_palindrome _____________________________

    def test_non_palindrome():
>       assert False
E       assert False

tests/test_palindrome.py:4: AssertionError
=========================== short test summary info ============================
FAILED tests/test_palindrome.py::test_non_palindrome - assert False
============================== 1 failed in 0.01s ===============================
(venv) [python_package_tutorial (main)]$
```

Figure 8.3: The RED state of the initial test suite.

By itself, this test isn't useful, but it demonstrates the concept, and we'll add a useful test in just a moment.

Many systems, including mine, display failing tests in the color red, as shown in Figure 8.3. Because of this, a failing test (or collection of tests, known as a *test suite*) is often referred to as being RED. To help us keep track of our progress, the captions of code listings corresponding to a failing test suite are labeled RED, as seen in Listing 8.9 and Listing 8.10.

To get from a failing to a passing state, we can change `False` to `True` in Listing 8.9, yielding the code in Listing 8.11.

Listing 8.11: A passing test suite. GREEN
test/test_phrase.py

```
def test_initial_example():
    assert True
```

As expected, this test passes:

Listing 8.12: GREEN

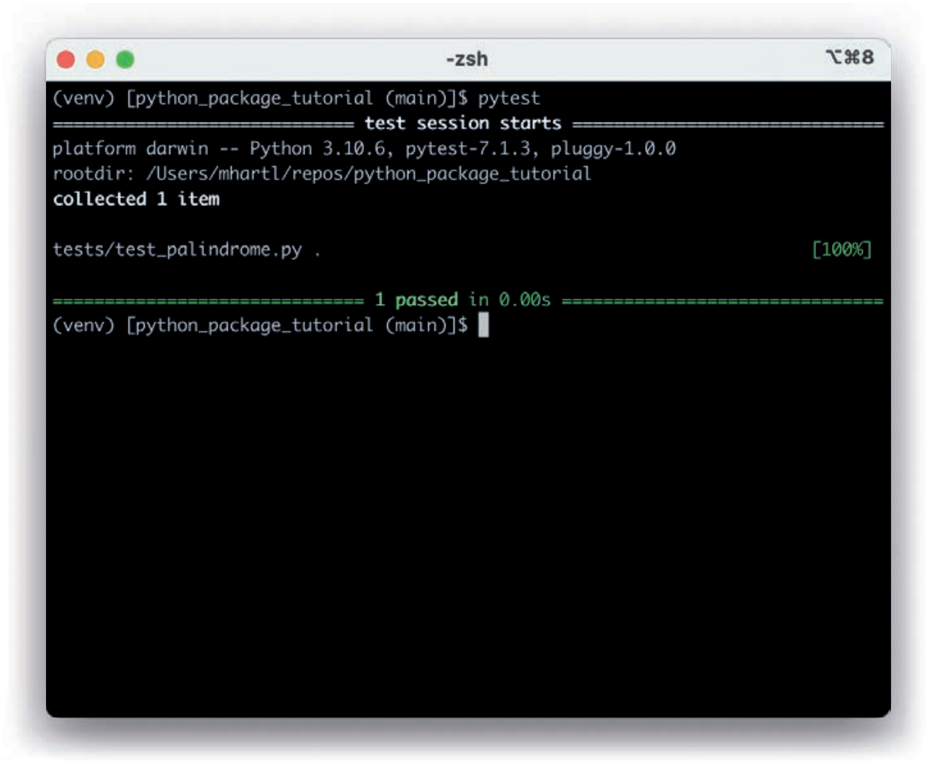
```
(venv) $ pytest
============================= test session starts ==============================
collected 1 item

tests/test_phrase.py .                                                   [100%]

============================== 1 passed in 0.00s ===============================
```

Because many systems display passing tests using the color green (Figure 8.4), a passing test suite is often referred to as GREEN. As with RED test suites, the captions of code listings corresponding to passing tests will be labeled GREEN (as seen in Listing 8.11 and Listing 8.12).

In addition to asserting that true things are `True`, it is often convenient to assert that false things are *not* `False`, which we can accomplish using `not` (Section 2.4.1), as shown in Listing 8.13.

Listing 8.13: A different way to pass. GREEN
test/test_phrase.py

```
def test_initial_example():
    assert not False
```



```
(venv) [python_package_tutorial (main)]$ pytest
============================== test session starts ==============================
platform darwin -- Python 3.10.6, pytest-7.1.3, pluggy-1.0.0
rootdir: /Users/mhartl/repos/python_package_tutorial
collected 1 item

tests/test_palindrome.py .                                                 [100%]

=============================== 1 passed in 0.00s ===============================
(venv) [python_package_tutorial (main)]$
```

Figure 8.4: A GREEN test suite.

As before, this test is GREEN:

Listing 8.14: GREEN

```
(venv) $ pytest
============================= test session starts ==============================
collected 1 item

tests/test_phrase.py .                                                   [100%]

============================== 1 passed in 0.00s ===============================
```

8.2.1 A Useful Passing Test

Having learned the basic mechanics of GREEN and RED tests, we're now ready to write our first useful test. Because we mainly want to test the **`Phrase`** class, our first step is to fill in **`phrase.py`** with the source code for defining phrases. We'll start with just **`Phrase`** itself (without **`TranslatedPhrase`**), as shown in Listing 8.15. Note that for brevity we've also omitted the iterator code from Section 5.3.

Listing 8.15: Defining **`Phrase`** in a package.
~/src/palindrome/phrase.py

```
class Phrase:
    """A class to represent phrases."""

    def __init__(self, content):
        self.content = content

    def processed_content(self):
        """Process the content for palindrome testing."""
        return self.content.lower()

    def ispalindrome(self):
        """Return True for a palindrome, False otherwise."""
        return self.processed_content() == reverse(self.processed_content())

def reverse(string):
    """Reverse a string."""
    return "".join(reversed(string))
```

At this point, we're ready to try importing **`Phrase`** into our test file. With the package structure as in Listing 8.1, the **`Phrase`** class should be importable from the **`palindrome`** package, which in turn should be available using **`palindrome.-phrase`**.[4] The result appears in Listing 8.16, which also replaces the example test from Listing 8.13.

4. You wouldn't necessarily have been able to guess this; it's just the way Python packages work based on the directory structure shown in Listing 8.1 (i.e., the **`phrase.py`** file is in a directory called **`palindrome`**).

Listing 8.16: Importing the **`palindrome`** package. RED
test/test_phrase.py

```
from palindrome_mhartl.phrase import Phrase
```

Unfortunately, the test suite doesn't pass even though there's no longer even a test that could fail:

Listing 8.17: RED

```
(venv) $ pytest
============================= test session starts ==============================
collected 0 items / 1 error

==================================== ERRORS ====================================
___________________ ERROR collecting tests/test_phrase.py ____________________
ImportError while importing test module
'/Users/mhartl/repos/python_package_tutorial/tests/test_phrase.py'.
Hint: make sure your test modules/packages have valid Python names.
Traceback:
lib/python3.10/importlib/__init__.py:126: in import_module
    return _bootstrap._gcd_import(name[level:], package, level)
tests/test_phrase.py:1: in <module>
    from palindrome_mhartl.phrase import Phrase
E   ImportError: cannot import name 'Phrase' from 'palindrome.palindrome'
(/Users/mhartl/repos/python_package_tutorial/src/palindrome/phrase.py)
=========================== short test summary info ============================
ERROR tests/test_phrase.py
!!!!!!!!!!!!!!!!!!!! Interrupted: 1 error during collection !!!!!!!!!!!!!!!!!!!!
=============================== 1 error in 0.03s ===============================
```

The issue is that our package needs to be installed in the local environment in order to perform the **`import`** in Listing 8.16. Because it hasn't been installed yet, the test suite is in an error state. Although this is technically not the same as a failing state, an error state is still often called RED.

To fix the error, we need to install the **`palindrome`** package locally, which we can do using the command shown in Listing 8.18.

Listing 8.18: Installing an editable package locally.

```
(venv) $ pip install -e .
```

As you can learn from running **`pip install --help`** (or by viewing the **`pytest`** documentation), the **`-e`** option installs the package in **e**ditable mode, so it will update automatically when we edit the files. The location of the installation is in the current directory, as indicated by the **`.`** (dot).

At this point, the test suite should be, if not quite GREEN, at least no longer RED:

```
(venv) $ pytest
============================= test session starts ==============================
collected 0 items

============================ no tests ran in 0.00s =============================
```

Now we're ready to start making some tests to check that the code in Listing 8.15 is actually working. We'll start with a negative case, checking that a non-palindrome is correctly categorized as such:

```
def test_non_palindrome():
    assert not Phrase("apple").ispalindrome()
```

Here we've used **`assert`** to assert that **`"apple"`** should *not* be a palindrome (Figure 8.5[5]).

In similar fashion, we can test a literal palindrome (one that's literally the same forward and backward) with another **`assert`**:

```
def test_literal_palindrome():
    assert Phrase("racecar").ispalindrome()
```

Combining the code from the above discussion gives us the code shown in Listing 8.19.

Listing 8.19: An actually useful test suite.
`test/test_phrase.py`

```
from palindrome_mhartl.phrase import Phrase


def test_non_palindrome():
    assert not Phrase("apple").ispalindrome()
```

5. Image courtesy of Glayan/Shutterstock.

Figure 8.5: The word “apple”: not a palindrome.

```
def test_literal_palindrome():
    assert Phrase("racecar").ispalindrome()
```

Now for the real test (so to speak):

Listing 8.20: GREEN

```
(venv) $ pytest
============================= test session starts ==============================
platform darwin -- Python 3.10.6, pytest-7.1.3, pluggy-1.0.0
rootdir: /Users/mhartl/repos/python_package_tutorial
collected 2 items

tests/test_phrase.py ..                                                   [100%]

============================== 2 passed in 0.00s ===============================
```

The tests are now GREEN, indicating that they are in a passing state. That means our code is working!

8.2.2 Pending Tests

Before moving on, we'll add a couple of *pending* tests, which are placeholders/ reminders for tests we want to write. The way to write a pending test is to use the **`skip()`** function, which we can include directly from the **`pytest`** package, as shown in Listing 8.21.

Listing 8.21: Adding two pending tests. YELLOW
`test/test_phrase.py`

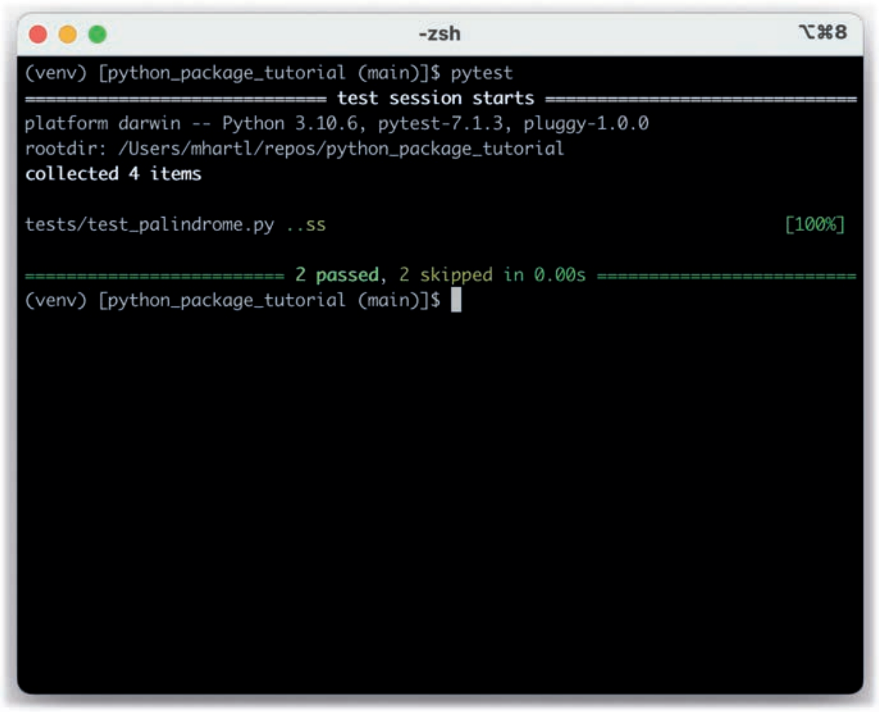
```
from pytest import skip

from palindrome_mhartl.phrase import Phrase


def test_non_palindrome():
    assert not Phrase("apple").ispalindrome()

def test_literal_palindrome():
    assert Phrase("racecar").ispalindrome()

def test_mixed_case_palindrome():
    skip()

def test_palindrome_with_punctuation():
    skip()
```

We can see the result of Listing 8.21 by rerunning the test suite:

Listing 8.22: YELLOW

```
(venv) $ pytest
============================= test session starts ==============================
collected 4 items

tests/test_phrase.py ..ss                                                [100%]

========================= 2 passed, 2 skipped in 0.00s =========================
```

Note how the test runner displays the letter **`s`** for each of the two "skips". Sometimes people speak of a test suite with pending tests as being YELLOW, in analogy with the



```
(venv) [python_package_tutorial (main)]$ pytest
============================== test session starts ===============================
platform darwin -- Python 3.10.6, pytest-7.1.3, pluggy-1.0.0
rootdir: /Users/mhartl/repos/python_package_tutorial
collected 4 items

tests/test_palindrome.py ..ss                                             [100%]

========================= 2 passed, 2 skipped in 0.00s ==========================
(venv) [python_package_tutorial (main)]$ 
```

Figure 8.6: A YELLOW (pending) test suite.

red-yellow-green color scheme of traffic lights (Figure 8.6), although it's also common to refer to any non-RED test suite as GREEN.

Filling in the test for a mixed-case palindrome is left as an exercise (with a solution shown in Listing 8.25), while filling in the second pending test and getting it to pass is the subject of Section 8.3 and Section 8.4.

8.2.3 Exercises

1. By filling in the code in Listing 8.23, add a test for a mixed-case palindrome like "RaceCar". Is the test suite still GREEN (or YELLOW)?

2. In order to make 100% sure that the tests are testing what we *think* they're testing, it's a good practice to get to a failing state (RED) by intentionally *breaking* the tests. Change the application code to break each of the existing tests in turn, and then confirm that they are GREEN again once the original code has been restored. An example of code that breaks the test in the previous exercise (but not the other tests) appears in Listing 8.24. (One advantage of writing the tests *first* is that this RED–GREEN cycle happens automatically.)

Listing 8.23: Adding a test for a mixed-case palindrome.
test/test_phrase.py

```
from pytest import skip

from palindrome_mhartl.phrase import Phrase


def test_non_palindrome():
    assert not Phrase("apple").ispalindrome()

def test_literal_palindrome():
    assert Phrase("racecar").ispalindrome()

def test_mixed_case_palindrome():
    FILL_IN

def test_palindrome_with_punctuation():
    skip()
```

Listing 8.24: Intentionally breaking a test. RED
src/palindrome/phrase.py

```
class Phrase:
    """A class to represent phrases."""

    def __init__(self, content):
        self.content = content

    def processed_content(self):
        """Process the content for palindrome testing."""
        return self.content#.lower()
    def ispalindrome(self):
        """Return True for a palindrome, False otherwise."""
        return self.processed_content() == reverse(self.processed_content())

def reverse(string):
    """Reverse a string."""
    return "".join(reversed(string))
```

8.3 Red

In this section, we'll take the important first step toward being able to detect more complex palindromes like "Madam, I'm Adam." and "A man, a plan, a canal—Panama!". Unlike the previous strings we've encountered, these phrases—which contain both spaces and punctuation—aren't strictly palindromes in a literal sense, even if we ignore capitalization. Instead of testing the strings as they are, we have to figure out a way to select only the letters, and then see if the resulting letters are the same forward and backward.

The code to do this is fairly tricky, but the tests for it are simple. This is one of the situations where test-driven development particularly shines (Box 8.1). We can write our simple tests, thereby getting to RED, and then write the application code any way we want to get to GREEN (Section 8.4). At that point, with the tests protecting us against undiscovered errors, we can change the application code with confidence (Section 8.5).

Box 8.1: When to test

When deciding when and how to test, it's helpful to understand *why* to test. In my view, writing automated tests has three main benefits:

1. Tests protect against *regressions*, where a functioning feature stops working for some reason.
2. Tests allow code to be *refactored* (i.e., changing its form without changing its function) with greater confidence.
3. Tests act as a *client* for the application code, thereby helping determine its design and its interface with other parts of the system.

Although none of the above benefits *require* that tests be written first, there are many circumstances where test-driven development (TDD) is a valuable tool to have in your kit. Deciding when and how to test depends in part on how comfortable you are writing tests; many developers find that, as they get better at writing tests, they are more inclined to write them first. It also depends on how difficult the test is relative to the application code, how precisely the desired features are known, and how likely the feature is to break in the future.

In this context, it's helpful to have a set of guidelines on when we should test first (or test at all). Here are some suggestions based on my own experience:

- When a test is especially short or simple compared to the application code it tests, lean toward writing the test first.
- When the desired behavior isn't yet crystal clear, lean toward writing the application code first, then write a test to codify the result.
- Whenever a bug is found, write a test to reproduce it and protect against regressions, then write the application code to fix it.
- Write tests before refactoring code, focusing on testing error-prone code that's especially likely to break.

We'll start by writing a test for a palindrome with punctuation, which just parallels the tests from Listing 8.19:

```
def test_palindrome_with_punctuation():
    assert palindrome.ispalindrome("Madam, I'm Adam.")
```

The updated test suite appears in Listing 8.25, which also includes the solution to a couple of exercises in Listing 8.23 (Figure 8.7[6]).

Listing 8.25: Adding a test for a punctuated palindrome. RED
test/test_phrase.py

```
from pytest import skip

from palindrome_mhartl.phrase import Phrase


def test_non_palindrome():
    assert not Phrase("apple").ispalindrome()
```

6. Image courtesy of Msyaraafiq/Shutterstock.

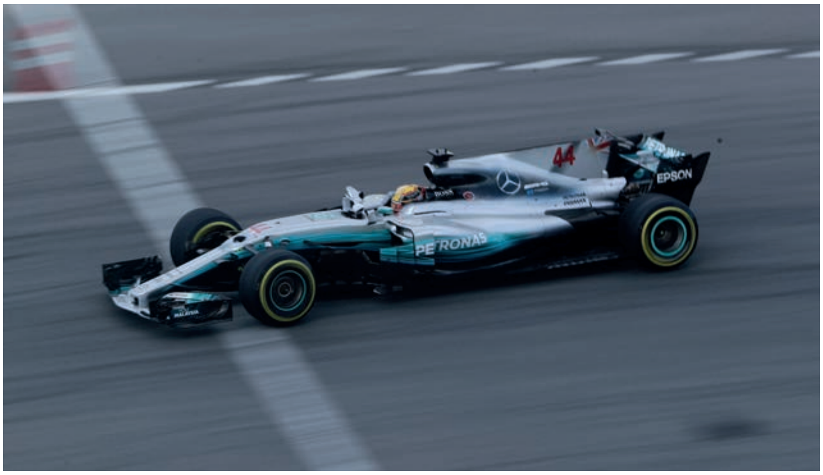


Figure 8.7: "RaceCar" is still a palindrome (ignoring case).

```python
def test_literal_palindrome():
    assert Phrase("racecar").ispalindrome()

def test_mixed_case_palindrome():
    assert Phrase("RaceCar").ispalindrome()

def test_palindrome_with_punctuation():
    assert Phrase("Madam, I'm Adam.").ispalindrome()
```

As required, the test suite is now RED (output somewhat streamlined):

Listing 8.26: RED

```
(venv) $ pytest
============================= test session starts ==============================
collected 4 items

tests/test_phrase.py ...F                                             [100%]

================================== FAILURES ===================================
_______________________ test_palindrome_with_punctuation _______________________

    def test_palindrome_with_punctuation():
>       assert Phrase("Madam, I'm Adam.").ispalindrome()
E       assert False
```

```
tests/test_phrase.py:14: AssertionError
=========================== short test summary info ============================
FAILED tests/test_phrase.py::test_palindrome_with_punctuation - assert False
========================= 1 failed, 3 passed in 0.01s ==========================
```

At this point, we can start thinking about how to write the application code and get to GREEN. Our strategy will be to write a **`letters()`** method that returns only the letters in the content string. In other words, the code

```
Phrase("Madam, I'm Adam.").letters()
```

should evaluate to this:

```
"MadamImAdam"
```

Getting to that state will allow us to use our current palindrome detector to determine whether the original phrase is a palindrome or not.

Having made this specification, we can now write a simple test for **`letters()`** by asserting that the result is as indicated:

```
assert Phrase("Madam, I'm Adam.").letters() == "MadamImAdam"
```

The new test appears with the others in Listing 8.27.

Listing 8.27: Adding a test for the **`letters()`** method. RED
test/test_phrase.py

```
from pytest import skip

from palindrome_mhartl.phrase import Phrase


def test_non_palindrome():
    assert not Phrase("apple").ispalindrome()

def test_literal_palindrome():
    assert Phrase("racecar").ispalindrome()

def test_mixed_case_palindrome():
    assert Phrase("RaceCar").ispalindrome()
```

```python
def test_palindrome_with_punctuation():
    assert Phrase("Madam, I'm Adam.").ispalindrome()

def test_letters():
    assert Phrase("Madam, I'm Adam.").letters() == "MadamImAdam"
```

Meanwhile, although we aren't yet ready to define a working `letters()` method, we can add a *stub*: a method that doesn't work, but at least exists. For simplicity, we'll simply return nothing (using the special `pass` keyword), as shown in Listing 8.28.

Listing 8.28: A stub for the `letters()` method. RED
src/palindrome/phrase.py

```python
class Phrase:
    """A class to represent phrases."""

    def __init__(self, content):
        self.content = content

    def ispalindrome(self):
        """Return True for a palindrome, False otherwise."""
        return self.processed_content() == reverse(self.processed_content())

    def processed_content(self):
        """Return content for palindrome testing."""
        return self.content.lower()

    def letters(self):
        """Return the letters in the content."""
        pass

def reverse(string):
    """Reverse a string."""
    return "".join(reversed(string))
```

The new test for `letters()` is RED as expected (which also shows that the `pass` in Listing 8.28 just returns `None`):

Listing 8.29: RED

```
(venv) $ pytest
============================= test session starts ==============================
collected 5 items

tests/test_phrase.py ...FF                                               [100%]

=================================== FAILURES ===================================
________________________ test_palindrome_with_punctuation ______________________

    def test_palindrome_with_punctuation():
>       assert Phrase("Madam, I'm Adam.").ispalindrome()
E       assert False

tests/test_phrase.py:14: AssertionError
_________________________________ test_letters _________________________________

    def test_letters():
>       assert Phrase("Madam, I'm Adam.").letters() == "MadamImAdam"
E       assert None == 'MadamImAdam'
tests/test_phrase.py:17: AssertionError
=========================== short test summary info ============================
FAILED tests/test_phrase.py::test_palindrome_with_punctuation - assert False
FAILED tests/test_phrase.py::test_letters - assert None == 'MadamImAdam'
========================= 2 failed, 3 passed in 0.01s ==========================
```

With our two RED tests capturing the desired behavior, we're now ready to move on to the application code and try getting it to GREEN.

8.3.1 Exercise

1. Confirm that commenting out the `letters()` stub in Listing 8.28 yields a failing state rather than an error state. (This behavior is relatively unusual, with many other languages distinguishing between a non-working method and one that's missing altogether. In Python, though, the result is the same failing state in either case.)

8.4 Green

Now that we have RED tests to capture the enhanced behavior of our palindrome detector, it's time to make them GREEN. Part of the philosophy of TDD is to get them passing without worrying too much at first about the quality of the implementation. Once the test suite is GREEN, we can polish it up without introducing regressions (Box 8.1).

The main challenge is implementing `letters()`, which returns a string of the letters (but not any other characters) making up the `content` of the `Phrase`. In other words, we need to select the characters that match a certain pattern. This sounds like a job for regular expressions (Section 4.3).

At times like these, using an online regex matcher with a regex reference like the one shown in Figure 4.5 is an excellent idea. Indeed, sometimes they make things a little *too* easy, such as when the reference has the exact regex you need (Figure 8.8).

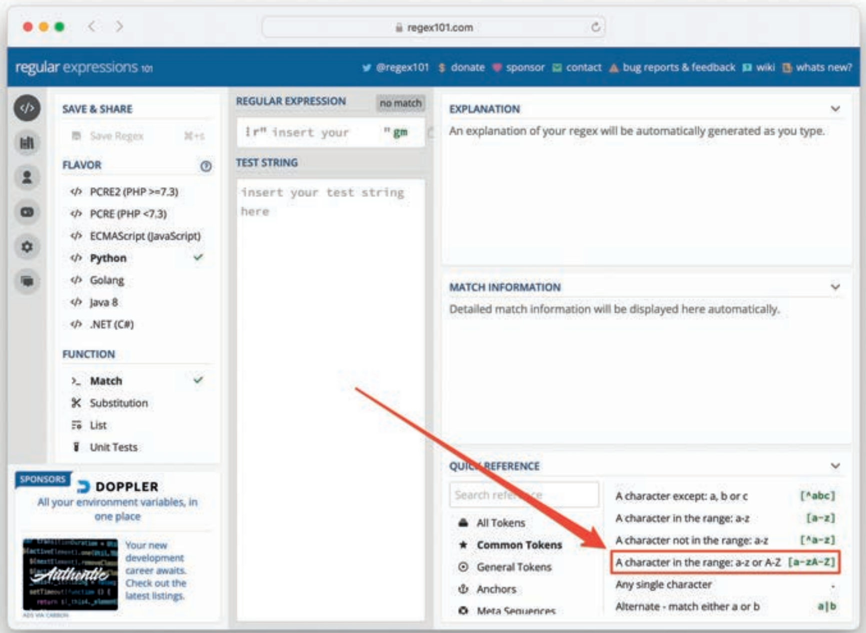

regex101.com
regular expressions 101
@regex101
donate
sponsor
contact
bug reports & feedback
wiki
whats new?
SAVE & SHARE
Save Regex
FLAVOR
PCRE2 (PHP >=7.3)
PCRE (PHP <7.3)
ECMAScript (JavaScript)
Python
Golang
Java 8
.NET (C#)
FUNCTION
Match
Substitution
List
Unit Tests
SPONSORS
DOPPLER
All your environment variables, in one place
Your new development career awaits. Check out the latest listings.
REGULAR EXPRESSION
no match
r" insert your
" gm
TEST STRING
insert your test string here
EXPLANATION
An explanation of your regex will be automatically generated as you type.
MATCH INFORMATION
Detailed match information will be displayed here automatically.
QUICK REFERENCE
Search reference
All Tokens
Common Tokens
General Tokens
Anchors
Meta Sequences
A character except: a, b or c [^abc]
A character in the range: a-z [a-z]
A character not in the range: a-z [^a-z]
A character in the range: a-z or A-Z [a-zA-Z]
Any single character .
Alternate - match either a or b a|b

Figure 8.8: The exact regex we need.

Let's test it in the console to make sure it satisfies our criteria (using the **`re.search()`** method introduced in Section 4.3):[7]

```
$ source venv/bin/activate
(venv) $ python3
>>> import re
>>> re.search(r"[a-zA-Z]", "M")
<re.Match object; span=(0, 1), match='M'>
>>> bool(re.search(r"[a-zA-Z]", "M"))
True
>>> bool(re.search(r"[a-zA-Z]", "d"))
True
>>> bool(re.search(r"[a-zA-Z]", ","))
False
```

Lookin' good!

We're now in a position to build up an array of characters that matches upper- or lowercase letters. The most straightforward way to do this is with the **`for`** loop method we first saw in Section 2.6. We'll start with an array for the letters, and then iterate through the **`content`** string, pushing each character onto the array (Section 3.4.3) if it matches the letter regex:

```
# Works but not Pythonic
the_letters = []
for character in self.content:
    if re.search(r"[a-zA-Z]", character):
        the_letters.append(character)
```

At this point, **`the_letters`** is an array of letters, which can be **`join`**ed to form a string of the letters in the original string:

```
"".join(the_letters)
```

Putting everything together gives the **`letters()`** method in Listing 8.30 (with a highlight added to indicate the beginning of the new method).

7. Note that this won't work for non-ASCII characters. If you need to match words containing such characters, the Google search python unicode letter regular expression might be helpful. Thanks to reader Paul Gemperle for pointing out this issue.

Listing 8.30: A working `letters()` method (but with full suite still RED).
`src/palindrome/phrase.py`

```
import re

class Phrase:
    """A class to represent phrases."""

    def __init__(self, content):
        self.content = content

    def ispalindrome(self):
        """Return True for a palindrome, False otherwise."""
        return self.processed_content() == reverse(self.processed_content())

    def processed_content(self):
        """Return content for palindrome testing."""
        return self.content.lower()

    def letters(self):
        """Return the letters in the content."""
        the_letters = []
        for character in self.content:
            if re.search(r"[a-zA-Z]", character):
                the_letters.append(character)
        return "".join(the_letters)

def reverse(string):
    """Reverse a string."""
    return "".join(reversed(string))
```

Although the full test suite is still RED, our **`letters()`** test should now be GREEN, as indicated by the number of failing tests changing from 2 to 1:

Listing 8.31: RED

```
(venv) $ pytest
============================= test session starts ==============================
platform darwin -- Python 3.10.6, pytest-7.1.3, pluggy-1.0.0
rootdir: /Users/mhartl/repos/python_package_tutorial
collected 5 items

tests/test_phrase.py ...F.                                               [100%]

=================================== FAILURES ===================================
________________________ test_palindrome_with_punctuation ________________________

    def test_palindrome_with_punctuation():
>       assert Phrase("Madam, I'm Adam.").ispalindrome()
E       assert False

tests/test_phrase.py:14: AssertionError
=========================== short test summary info ============================
FAILED tests/test_phrase.py::test_palindrome_with_punctuation - assert False
========================= 1 failed, 4 passed in 0.01s ==========================
```

We can get the final RED test to pass by replacing **`self.content`** with **`self.letters()`** in the **`processed_content()`** method. The result appears in Listing 8.32.

Listing 8.32: A working **`ispalindrome()`** method. GREEN
src/palindrome/phrase.py

```python
import re


class Phrase:
    """A class to represent phrases."""

    def __init__(self, content):
        self.content = content

    def ispalindrome(self):
        """Return True for a palindrome, False otherwise."""
        return self.processed_content() == reverse(self.processed_content())

    def processed_content(self):
        """Return content for palindrome testing."""
        return self.letters().lower()

    def letters(self):
        """Return the letters in the content."""
        the_letters = []
        for character in self.content:
            if re.search(r"[a-zA-Z]", character):
                the_letters.append(character)
        return "".join(the_letters)

def reverse(string):
    """Reverse a string."""
    return "".join(reversed(string))
```

Figure 8.9: Our detector finally understands Adam's palindromic nature.

The result of Listing 8.32 is a GREEN test suite (Figure 8.9[8]):

Listing 8.33: GREEN

```
(venv) $ pytest
============================= test session starts ==============================
collected 5 items

tests/test_phrase.py .....                                               [100%]

============================== 5 passed in 0.00s ===============================
```

It may not be the prettiest code in the world, but this GREEN test suite means our code is working!

8. Image courtesy of Album/Alamy Stock Photo.

8.4.1 Exercise

1. Using the same code shown in Listing 8.16, import the **Phrase** class into the Python REPL and confirm directly that **ispalindrome()** can successfully detect palindromes of the form "Madam, I'm Adam."

8.5 Refactor

Although the code in Listing 8.32 is now working, as evidenced by our GREEN test suite, it relies on a rather cumbersome **for** loop that appends to a list rather than creating it all at once. In this section, we'll *refactor* our code, which is the process of changing the form of code without changing its function.

By running our test suite after any significant changes, we'll catch any regressions quickly, thereby giving us confidence that the final form of the refactored code is still correct. Throughout this section, I suggest making changes incrementally and running the test suite after each change to confirm that the suite is still GREEN.

Per Chapter 6, a more Pythonic way of creating a list of the sort in Listing 8.32 is to use a list comprehension. In particular, the loop in Listing 8.32 bears a strong resemblance to the **imperative_singles()** function from Listing 6.4:

```
states = ["Kansas", "Nebraska", "North Dakota", "South Dakota"]
.
.
.
# singles: Imperative version
def imperative_singles(states):
    singles = []
    for state in states:
        if len(state.split()) == 1:
            singles.append(state)
    return singles
```

As we saw in Listing 6.5, this can be replaced using a list comprehension with a condition:

```
# singles: Functional version
def functional_singles(states):
    return [state for state in states if len(state.split()) == 1]
```

Let’s drop into the REPL to see how to do the same thing in the present case:

```
>>> content = "Madam, I'm Adam."
>>> [c for c in content]
['M', 'a', 'd', 'a', 'm', ',', ' ', 'I', "'", 'm', ' ', 'A', 'd', 'a', 'm', '.']
>>> [c for c in content if re.search(r"[a-zA-Z]", c)]
['M', 'a', 'd', 'a', 'm', 'I', 'm', 'A', 'd', 'a', 'm']
>>> "".join([c for c in content if re.search(r"[a-zA-Z]", c)])
'MadamImAdam'
```

We see here how combining a list comprehension with a condition and a **`join()`** lets us replicate the current functionality of **`letters()`**. In fact, inside the argument to **`join()`** we can omit the square brackets and use a generator comprehension (Section 6.4) instead:

```
>>> "".join(c for c in content if re.search(r"[a-zA-Z]", c))
'MadamImAdam'
```

This leads to the updated method shown in Listing 8.34. As is so often the case with comprehension solutions, we have been able to condense the imperative solution down to a single line.

Listing 8.34: Refactoring **`letters()`** down to a single line. GREEN
`src/palindrome/phrase.py`

```python
import re


class Phrase:
    """A class to represent phrases."""

    def __init__(self, content):
        self.content = content

    def ispalindrome(self):
        """Return True for a palindrome, False otherwise."""
        return self.processed_content() == reverse(self.processed_content())

    def processed_content(self):
        """Return content for palindrome testing."""
        return self.letters().lower()

    def letters(self):
        """Return the letters in the content."""
        return "".join(c for c in self.content if re.search(r"[a-zA-Z]", c))

def reverse(string):
    """Reverse a string."""
    return "".join(reversed(string))
```

As noted in Chapter 6, functional programs are harder to build up incrementally, which is one reason why it's so nice to have a test suite to check that our changes had their intended effect (that is, no effect at all):

Listing 8.35: GREEN

```
(venv) $ pytest
============================= test session starts ==============================
collected 5 items

tests/test_phrase.py .....                                               [100%]

============================== 5 passed in 0.01s ===============================
```

Huzzah! Our test suite still passes, so our new one-line **`letters()`** method works.

This is a major improvement, but in fact there's one more refactoring that represents a great example of the power of Python. Recall from Section 4.3 that regular expressions have a **`findall()`** method that lets us select regex-matching characters directly from a string:

```
>>> re.findall(r"[a-zA-Z]", content)
['M', 'a', 'd', 'a', 'm', 'I', 'm', 'A', 'd', 'a', 'm']
>>> "".join(re.findall(r"[a-zA-Z]", content))
'MadamImAdam'
```

By using **`findall()`** with the same regex we've been using throughout this section and then joining on the empty string, we can simplify the application code even further by eliminating the list comprehension, as shown in Listing 8.36.

Listing 8.36: Using `re.findall`. GREEN

src/palindrome/phrase.py

```
import re


class Phrase:
    """A class to represent phrases."""

    def __init__(self, content):
        self.content = content

    def ispalindrome(self):
        """Return True for a palindrome, False otherwise."""
        return self.processed_content() == reverse(self.processed_content())

    def processed_content(self):
        """Return content for palindrome testing."""
        return self.letters().lower()

    def letters(self):
        """Return the letters in the content."""
        return "".join(re.findall(r"[a-zA-Z]", self.content))

def reverse(string):
    """Reverse a string."""
    return "".join(reversed(string))
```

One more run of the test suite confirms that everything is still copacetic (Figure 8.10[9]):

Listing 8.37: GREEN

```
(venv) $ pytest
============================= test session starts ==============================
collected 5 items

tests/test_phrase.py .....                                              [100%]

============================== 5 passed in 0.01s ===============================
```

9. Image courtesy of Album/Alamy Stock Photo.

Figure 8.10: Still a palindrome after all our work.

8.5.1 Publishing the Python Package

As a final step, and in line with our philosophy of shipping (Box 1.5), in this final section we'll publish our **palindrome** package to the Python Package Index, also known as PyPI.

Unusually among programming languages, Python actually has a dedicated test package index called TestPyPI, which means we can publish (and use) our test package without uploading to a real package index. Before proceeding, you'll need to register an account at TestPyPI and verify your email address.

Once you've set up your account, you'll be ready to build and publish your package. To do this, we'll be using the **build** and **twine** packages, which you should install at this time:

```
(venv) $ pip install build==0.8.0
(venv) $ pip install twine==4.0.1
```

The first step is to build the package as follows:

```
(venv) $ python3 -m build
```

This uses the information in **`pyproject.toml`** (Listing 8.3) to create a **`dist`** ("distribution") directory with files based on the name and version number of your package. For example, on my system the **`dist`** directory looks like this:

```
(venv) $ ls dist
palindrome\_mhartl-0.0.1.tar.gz
palindrome_mhartl-0.0.1-py3-none-any.whl
```

These are a tarball and wheel file, respectively, but the truth is that you don't need to know anything about these files specifically; all you need to know is that the **`build`** step is necessary to publish a package to TestPyPI. (Being comfortable with ignoring these sorts of details is a good sign of technical sophistication.)

Actually publishing the package involves using the **`twine`** command, which looks like this (and is just copied from the TestPyPI documentation):[10]

```
(venv) $ twine upload --repository testpypi dist/*
```

(For future uploads, you may need to remove older versions of your package using **`rm`** because TestPyPI doesn't let you reuse filenames.)

At this point, your package is published and you can test it by installing it on your local system. Because we already have an editable and testable version of the package in our main venv (Listing 8.18), it's a good idea to spin up a new venv in a temp directory:

```
$ cd
$ mkdir -p tmp/test_palindrome
$ cd tmp/test_palindrome
$ python3 -m venv venv
$ source venv/bin/activate
(venv) $
```

10. At this point, you will be prompted either for a username and password or for an API key. For the latter, see the TestPyPI page on tokens for more information.

Now you can install your package by using the `--index-url` option to tell `pip` to use the test index instead of the real one:

```
(venv) $ pip install <package> --index-url https://test.pypi.org/simple/
```

For example, I can install my version of the test package, which is called `palindrome_mhartl`, as follows:[11]

```
(venv) $ pip install palindrome_mhartl --index-url https://test.pypi.org/simple/
```

To test the installation, you can load the package in the REPL:

```
(venv) $ python3
>>> from palindrome_mhartl.phrase import Phrase
>>> Phrase("Madam, I'm Adam.").ispalindrome()
True
```

It works! (If it doesn't work for you—which is a real possibility since so many things can go wrong—the only recourse is to use your technical sophistication to resolve the discrepancy.)

For a general Python package, you can continue adding features and making new releases. All you need to do is increment the version number in `pyproject.toml` to reflect the changes you've made. For more guidance on how to increment the versions, I suggest learning a bit about the rules of so-called *semantic versioning*, or *semver* (Box 8.2).

Box 8.2: Semver

You might have noticed in this section that we've used the version number 0.1.0 for our new package. The leading zero indicates that our package is at an early stage, often called "beta" (or even "alpha" for very early-stage projects).

11. The `_mhartl` part comes from the `name` setting in `pyproject.toml`, which for me is `palindrome_-mhartl`. If you install my version of the package, you may notice that the version number is higher than 0.0.1, which is due to the aforementioned issue regarding package-name reuse. Because I've made quite a few changes in the course of developing this tutorial, I've incremented the version number (`version` in `pyproject.toml`) several times.

We can indicate updates by incrementing the middle number in the version, e.g., from 0.1.0 to 0.2.0, 0.3.0, etc. Bugfixes are represented by incrementing the rightmost number, as in 0.2.1, 0.2.2, etc., and a mature version (suitable for use by others, and which may not be backward-compatible with prior versions) is indicated by version 1.0.0.

After reaching version 1.0.0, further changes follow this same general pattern: 1.0.1 would represent minor changes (a "patch release"), 1.1.0 would represent new (but backward-compatible) features (a "minor release"), and 2.0.0 would represent major or backward-incompatible changes (a "major release").

These numbering conventions are known as *semantic versioning*, or *semver* for short. For more information, see semver.org.

Finally, if you ever go on to develop a package that isn't just a test like the one in this chapter, you can publish it to the real Python Package Index (PyPI). Although there is ample PyPI documentation, there is little doubt in such a case that you will also have ample opportunity to apply your technical sophistication.

8.5.2 Exercises

1. Let's generalize our palindrome detector by adding the capability to detect integer palindromes like **`12321`**. By filling in **`FILL_IN`** in Listing 8.38, write tests for integer non-palindromes and palindromes. Get both tests to GREEN using the code in Listing 8.39, which adds a call to **`str`** to ensure the content is a string and includes **`\d`** in the regex to match digits as well as letters. (Note that we have updated the name of the **`letters()`** method accordingly.)
2. Bump the version number in **`pyproject.toml`**, commit and push your changes, build your package with **`build`**, and upload it with **`twine`**. In your temp directory, upgrade your package using the command in Listing 8.40 and confirm in the REPL that integer-palindrome detection is working. *Note*: The backslash **`\`** in Listing 8.40 is a *continuation character* and should be typed literally, but the right angle bracket **`>`** should be added by your shell program automatically and should not be typed.

Listing 8.38: Testing integer palindromes. RED
tests/test_phrase.py

```python
from pytest import skip

from palindrome_mhartl.phrase import Phrase


def test_non_palindrome():
    assert not Phrase("apple").ispalindrome()

def test_literal_palindrome():
    assert Phrase("racecar").ispalindrome()

def test_mixed_case_palindrome():
    assert Phrase("RaceCar").ispalindrome()

def test_palindrome_with_punctuation():
    assert Phrase("Madam, I'm Adam.").ispalindrome()

def test_letters_and_digits():
    assert Phrase("Madam, I'm Adam.").letters_and_digits() == "MadamImAdam"

def test_integer_non_palindrome():
    FILL_IN Phrase(12345).ispalindrome()

def test_integer_palindrome():
    FILL_IN Phrase(12321).ispalindrome()
```

Listing 8.39: Adding detection of integer palindromes. GREEN
src/palindrome/phrase.py

```python
import re


class Phrase:
    """A class to represent phrases."""

    def __init__(self, content):
        self.content = str(content)

    def ispalindrome(self):
        """Return True for a palindrome, False otherwise."""
        return self.processed_content() == reverse(self.processed_content())
    def processed_content(self):
        """Return content for palindrome testing."""
        return self.letters_and_digits().lower()

    def letters_and_digits(self):
        """Return the letters and digits in the content."""
        return "".join(re.findall(r"[a-zA-Z]", self.content))

def reverse(string):
    """Reverse a string."""
    return "".join(reversed(string))
```

Listing 8.40: Upgrading the test package.

```
(venv) $ pip install --upgrade your-package \
> --index-url https://test.pypi.org/simple/
```

Index