Extracted from:

Intuitive Python

Productive Development for Projects that Last

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Pragmatic exPress

Intuitive Python

Productive Development for Projects that Last

David Muller

edited by Adaobi Obi Tulton

Intuitive Python

Productive Development for Projects that Last

David Muller

The Pragmatic Bookshelf

Raleigh, North Carolina

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Adaobi Obi Tulton
Copy Editor: Karen Galle
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

ISBN-13: 978-1-68050-823-9
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—June 2021

Detecting Problems Early

Unlike traditionally compiled languages, Python does not require source code to be compiled into machine code before it is run. Instead, Python accepts source code directly and executes it as is. This means that it is possible for you to, for example, run invalid Python source code that never stands a chance of executing or working.

To reduce this risk, many Python projects run static analysis tools to help them validate and verify source code before trying to run it. What are static analysis tools? Static analysis tools don't actually run your code, but read it and inspect it for issues that can be found just by browsing the source code itself. Kind of like a friend peering over your shoulder as you type and letting you know when you've made a mistake before you've even tried to run anything.

In this section we'll talk about two tools for finding and eliminating bugs in your programs ahead of time: flake8 and mypy.

Running flake8 to Find Errors

The flake8 static analysis tool detects a number of different errors in Python source code and flags them for you to fix.[22] In this section, we'll highlight a select few errors flake8 detects to help you get a sense of flake8's benefits.

How Do I Run flake8?

flake8 is a third-party package that you can install with Python's package manager pip.[23] If you're using this book's companion Docker image from Setting Up Your Environment and Using This Book's Companion Docker Image, on page ?, flake8 is already installed and ready to go—just type flake8 --help. If you're not using the companion image, we'll cover how to use pip in Running pip, on page ?. If you are already comfortable with pip and virtual environments, feel free to install flake8==3.8.4 and run it. Otherwise, it's okay to just follow along as we explore the capabilities of flake8.

Detecting Undefined Variables

flake8 detects, ahead of time, any Python source code that tries to access a variable that does not exist. For example, variable_does_not_exist.py contains a variable named oops that is never bound to a value:

22. https://flake8.pycqa.org/en/3.8.4/user/error-codes.html
23. https://pip.pypa.io/en/stable/

variable_does_not_exist.py

```python
a = 1

# `a + oops` will never work:
# Traceback (most recent call last):
#   File "variable_does_not_exist.py", line 8, in <module>
#     a + oops
# NameError: name 'oops' is not defined
a + oops
```

While variable_does_not_exist.py is valid syntactically, it will never run. a + oops will always raise an exception because the oops variable doesn't have a value.

flake8 is able to catch this error ahead of time. If you run flake8 against variable_does_not_exist.py by saying flake8 variable_does_not_exist.py, you'll see the following output:

```
variable_does_not_exist.py:8:5: F821 undefined name 'oops'
```

flake8 reports that it detects an error at line 8 column 5 of variable_does_not_exist.py. In particular, it detected that the oops variable is not defined (and so variable_does_not_exist.py will not be able to run successfully). Also included in the output is the code: F821. F821 is the code name flake8 uses to identify this error—allowing you to find all occurrences of a specific kind of error in your project.[24]

While the variable_does_not_exist.py file may seem a little trivial, problems like these tend to crop up relatively frequently especially as a Python codebase grows (or when developers decline to write tests for their code). Catching these errors ahead of time with flake8 spares you from bugs later.

Remove Wildcard Imports

flake8 detects when wildcard (*) imports are used and forbids them. For example, consider redefine_path.py which subtly and silently clobbers the value of the path variable:

redefine_path.py

```python
path = "/etc/hosts"

from os import *

print(path)
```

If you run python3 redefine_path.py you will receive output like the following:

```
<module 'posixpath' from '/usr/local/lib/python3.9/posixpath.py'>
```

24. https://flake8.pycqa.org/en/3.8.4/user/error-codes.html

Shouldn't the output have been /etc/hosts? Unfortunately, even though the path variable was initially bound to /etc/hosts, the path variable is clobbered to a new value when from os import * is run. This is because the os module itself includes a path module. So, when * is imported from os, the path module gets bound to the path variable and the /etc/hosts string effectively disappears without a trace. (Note that Python aliases os.path to an underlying module of posixpath if you are running on a POSIX system and ntpath if you are running on Windows).

flake8 forbids this variable clobbering situation from occurring at all. If you run flake8 redefine_path.py, one of the errors you'll see is:

```
redefine_path.py:3:1: F403 'from os import *' used; unable to detect
undefined names
```

flake8 detects an issue at line 3 column 1 saying that a wildcard (*) import is used, preventing flake8 from detecting undefined names. To resolve this error, you have to replace the * and import the exact names you are interested in using. (For example, from os import environ if you wanted to use environ to get and set operating system environment variables.)

We'll discuss the dangers of wildcard imports in more detail in Dodging Wildcard Variable Shadowing, on page ?, but for now its enough to know that flake8 helps you detect and remove these before they manifest as dangerous bugs.

Prevent Duplicated Names

flake8 detects when you define a dictionary that repeats the same key multiple times with different values. Consider, for example, duplicate_dict_keys.py that defines two different values for the "pi" key in my_dict:

duplicate_dict_keys.py

```python
my_dict = {"pi": 3.14, "pi": "apple"}

# is this `3.14 * 5` or `"apple" * 5` ??
print(my_dict["pi"] * 5)
```

The existing code is—at best—ambiguous. Should my_dict["pi"] return 3.14, or should it return "apple"? At the end of the day, a dictionary data structure only allows the "pi" key to appear once, so only one of 3.14 and "apple" will actually stick and win out as the value.

If you run flake8 duplicate_dict_keys.py, flake8 will catch and suggest you fix the key duplication ahead of time:

```
duplicate_dict_keys.py:1:12: F601 dictionary key 'pi' repeated with
different values
duplicate_dict_keys.py:1:24: F601 dictionary key 'pi' repeated with
```

```
different values
```

flake8 indicates the two locations on line 1 that assign the 'pi' key to a different value and suggests you address the duplication. Again, while the duplicate_dict_keys.py example may feel a little contrived because it is so small, this check becomes especially helpful when your codebase defines many dictionary literals—especially ones with dozens, hundreds, or even thousands of keys.

Prevent Duplicated Names Part 2: Tests

flake8 uses a similar duplicate detection scheme that can help you catch issues with tests as well.

After introducing flake8 to a codebase at a new job, I was simultaneously relieved and horrified when flake8 detected some tests that were being implicitly skipped. Consider the following simplified example of a unittest TestCase that tests math operations:

duplicate_tests.py

```python
import unittest


class TestMath(unittest.TestCase):
    def test_add_1(self):
        self.assertEqual(1 + 1, 2)

    def test_add_1(self):
        self.assertEqual(1 - 1, 0)
```

duplicate_tests.py defines a TestCase class named TestMath that—supposedly—tests that both 1 + 1 = 2 and 1 - 1 = 0. Unfortunately, if you actually run this test file with the unittest standard library module, you'll see that only one test runs:

```
python3 -m unittest duplicate_tests.py
```

```
❮ .
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

The test output indicates that it only Ran 1 test despite our code defining two tests. Similar to the duplicate dictionary key example shown earlier, only one method of a given name can ultimately bind to a class object. In this case there are two methods named test_add_1 that each try to bind to the TestMath class. However, since only one of the test_add_1 methods can get bound to the class, only one test actually runs.

When a large project accumulates many tests, and developers have accidentally written tests that have the same name, it's easy to miss this kind of problem. It's dangerous when code that developers thought was covered with tests actually isn't.

flake8 helps you avoid skipping tests by flagging the duplicated names. If you run flake8 duplicate_tests.py, you'll see an error message like this:

```
duplicate_tests.py:8:5: F811 redefinition of unused 'test_add_1' from line 5
```

flake8 notes that test_add_1 has been redefined on line 8. If you fix this issue by, for example, changing the name of the test on line 8 to test_subtract_1, flake8 will stop complaining and two tests will run:

duplicate_tests_fixed.py

```python
Line 1 import unittest
     2
     3
     4 class TestMath(unittest.TestCase):
     5     def test_add_1(self):
     6         self.assertEqual(1 + 1, 2)
     7
     8     def test_subtract_1(self):
     9         self.assertEqual(1 - 1, 0)
```

After renaming the second test_add_1 to test_subtract_1, using the unittest module to execute the tests results in both tests actually executing:

```
python3 -m unittest duplicate_tests_fixed.py
```

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Running Additional Checks with flake8-bugbear

There a number of available add-ons that you can use to augment flake8. I highly recommend one add-on in particular: flake-bugbear. flake8-bugbear[25] allows flake8 to detect a few more classes of errors and mistakes. In particular, flake8-bugbear automatically flags instances of the mutable default argument trap and helps you eliminate them from your codebase. The mutable default argument trap is covered in Binding Early: Problems with Default Arguments, on page ?. Don't worry too much about the mutable default argument trap now—just know that flake8-bugbear will help you automatically eliminate it.

25. https://pypi.org/project/flake8-bugbear/

Any Python project should use flake8 and flake8-bugbear to prevent errors in its code. I highly recommend adding flake8 and flake8-bugbear to your development flow (for example, in your continuous integration server) to prevent committing code with bugs that flake8 can spare you from. (Be sure to run flake8 with its --select=F option if you want to ignore flake8's style suggestions and only enable its error detection.[26])

In the next section we'll introduce another tool that belongs in your Python development flow: mypy.

26. https://flake8.pycqa.org/en/3.8.4/user/violations.html#selecting-violations-with-flake8