Unit testing assignment (Python)

CSC 305 – David H. Brown (Summer 2020)

Introduction

This document walks you through creating a few unit tests in Python. You will then write additional unit tests for features that have not yet been implemented. Your assignment submission must include:

1. Your enhanced `test_Palindrome.py` file with
    a. at least one additional, distinct test that expects a `ValueError` to be raised
    b. Three more tests that cover as-yet-untested "partitions" of the possible input space
2. A **screen capture** showing that you have run your unit tests – it's fine if they fail, but I have given you some hints in `Palindrome.py` if you want to take a stab at making them work.

What is a palindrome?

Courtesy of https://en.wikipedia.org/wiki/Palindrome, we know that…

A **palindrome** is a word, phrase, number, or other sequence of characters which reads the same backward or forward, such as *madam* or *kayak*. Sentence-length palindromes may be written when allowances are made for adjustments to capital letters, punctuation, and word dividers, such as "A man, a plan, a canal, Panama!", "Was it a car or a cat I saw?" or "No 'x' in Nixon"

API and Specification

To be able to write tests for a class, you need to know the API of the class and its expected behavior

- The class `Palindrome` will have a method `is_palindrome()` that takes a string as a parameter and returns `True` or `False`.
- Consider only alphanumeric characters (a-z, A-Z, 0-9), ignoring any punctuation and whitespace
- Disregard capitalization: e.g., "E" and "e" are the same
- Consider only the base letter, ignoring accents: e.g., "e" and "é" are the same
- Raise a `ValueError` exception if the string has no alphanumeric characters (https://docs.python.org/3/library/exceptions.html#ValueError)

Supplied code

Along with this document, you should find attached to the assignment a .zip containing code for `Palindrome.py` and `test_Palindrome.py`. In these instructions, I describe how to build the files yourself. You will learn more if you ignore the supplied files at first and recreate them on your own. But please switch to the provided test_Palindrome.py

Writing the code

Palindrome.py

Choose **File** > **New…** and create a new Python file named `Palindrome.py`.

I don't yet know how to code this algorithm, so I'll start with just a stub that honors the API contract (take a string, return a bool):

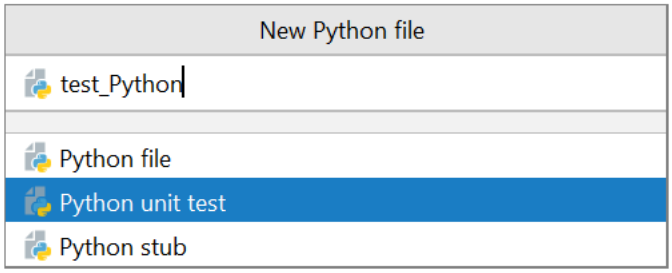
```
class Palindrome:
    def is_palindrome(self, text: str) -> bool:
        return False
```

(PyCharm correctly suggests that this method could be static (it doesn't use `self`), but that's okay.)

test_Palindrome.py

PyCharm's unittest template

When you ask PyCharm to create a new Python file (File > New…), it gives you three choices:



Type a name at the top – conventionally, test_ followed by the name of the thing you're testing – and arrow down or click to select "Python unit test." If you start the name of this file with "test," then the `unittest` module will discover it automatically. Press enter to create the file which PyCharm will fill with this generic starter code template:

```
import unittest


class MyTestCase(unittest.TestCase):
    def test_something(self):
        self.assertEqual(True, False)


if __name__ == '__main__':
    unittest.main()
```

The first line loads the `unittest` framework's module; it's built into Python. See https://docs.python.org/3/library/unittest.html for more on this. The `if` block lets us run this test as a script. These two items don't need changing. The `class` must inherit `(unittest.TestCase)`.

Customizing the unit test template with a setUp() method

After importing `unittest`, `from Palindrome import Palindrome.` The class definition should be renamed from `MyTestCase` to your class name plus "TestCase," i.e., `PalindromeTestCase`. Similarly, `test_something` should be renamed to identify more precisely exactly what is being tested in that method. Always begin test methods with `test_` so that `unittest` knows it's a test to be run instead of some other function. You can (and usually should) have multiple `test_` methods defined within a `TestCase` class.

Every one of our tests is going to need access to a fresh `Palindrome` object. A shared instance of Palindrome might change its state (maybe we want to be able to configure whether it ignores accents?) and it is very important that we don't allow one test to affect another. Any code that you would write at the beginning of each test should be moved to a `setUp()` method. The unittest module takes care of calling `setUp()` before each test. (It would also call `tearDown()` after each test, if you needed to do anything then.)

Our `setUp()` will create the new Palindrome and store it in a "private" class variable `_palindrome` so it's accessible to each tests. The IDE gives me a little help once I type `def se…` and accept the autocomplete. (You could also write a `tearDown()` method if you need to do something after each test.) Let's look at the code so far…

```
import unittest

from Palindrome import Palindrome
```

```
class PalindromeTestCase(unittest.TestCase):
    _palindrome: Palindrome

    def setUp(self) -> None:
        self._palindrome = Palindrome()

    def test_lowercase_word_is_palindrome(self):
        self.assertEqual(True, False)


if __name__ == '__main__':
    unittest.main()
```

Our first tests

Assert something

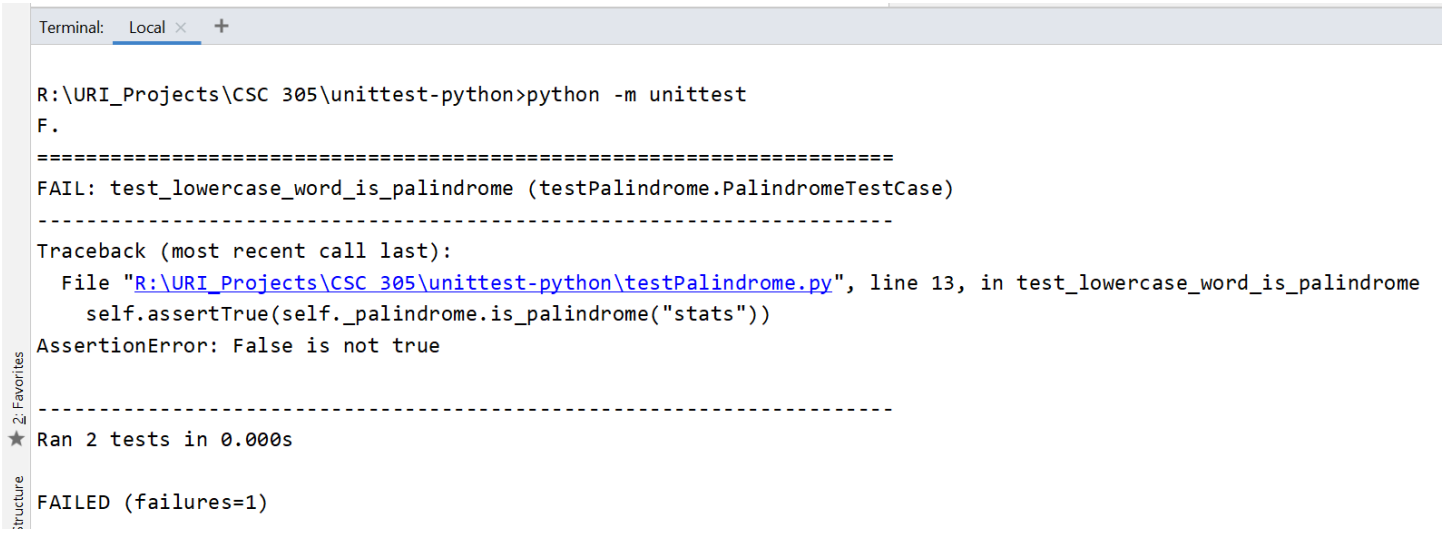
As you might infer from the method name, we want to test whether Palindrome can accurately identify a lowercase word that is a palindrome. The unittest.TestCase provides a couple dozen different assertions we can use to check our results. See https://docs.python.org/3/library/unittest.html#assert-methods for the list and explanations. For example, while we could assert that the result of calling `is_palindrome` equals `True`, it's clearer to `assertTrue`:

```
def test_lowercase_word_is_palindrome(self):
    self.assertTrue(self._palindrome.is_palindrome("stats"))

def test_lowercase_word_not_palindrome(self):
    self.assertFalse(self._palindrome.is_palindrome("python"))
```

Run the tests

There are several ways to run the unit tests. A good IDE will provide some help (right-click the window/file and choose Run), but it's pretty easy from the terminal. The template code included an `if __name__ == '__main__':` block to run the file as a script, so we can enter the command `python test_Palindrome.py` to run just this script and test only Palindrome. Usually, we will type `python -m unittest` to let the unittest module run its discovery function and find our tests for us:



Look at that: 50% passing and we haven't even begun to work on the algorithm!

Test-driven development

Now that we have a failing test case, we have a reason to modify our code. This is the essence of test-driven development: Write tests until one doesn't pass and then improve your code, extending it to handle scenarios covered by your new tests, until all your tests again pass.

A simple algorithm to check for a palindrome is to see whether a string is the same as the reverse of that string. Python doesn't include a str.reverse() method, but you can get that effect with the slicing operator, `[<start>:<end>:<step>]`. A string is a list of characters, so just slice in reverse through the entire string:

```
class Palindrome:
    def is_palindrome(self, text: str) -> bool:
        return text[::-1] == text
```

Now our tests pass! Let's write some more…

Checking for exceptions

Exceptions are an important part of your code's public API. Raising (or "throwing" in Java) an exception is a reliable way of signaling that something has gone funny. Avoid the temptation to return special values (like None in Python or null in Java) that would have to be checked for by the caller and could be missed.

You can assert that an exception is raised with the `assertRaises` method. With this, the test passes if the specified exception is raised and fails otherwise (no exception, other exception). There are two ways to use it. The first one takes at least two parameters: the expected exception, the name of a callable function – no parentheses, you don't actually call the function here – and then any parameters to pass to that function. So, we could write:

```
def test_empty_string_raises_ValueError(self):
    self.assertRaises(ValueError, self._palindrome.is_palindrome, "")
```

That's a little cumbersome and the use of `is_palindrome` doesn't look like how we'd normally use the method, so Python 3.1 added the ability to use `assertRaises` as a context manager instead. This (slightly?) more readable version accomplishes pretty much the same thing:

```
def test_empty_string_raises_ValueError_context(self):
    with self.assertRaises(ValueError):
        self._palindrome.is_palindrome("")
```

Here's some code to make those tests pass:

```
class Palindrome:
    def is_palindrome(self, text: str) -> bool:
        if len(text) == 0:
            raise ValueError
        return text[::-1] == text
```

(`ValueError` is built-in and doesn't need importing.)

How to select test cases

Unit testing requires you to identify good test cases. You don't want to write lots of tests that all exercise the same bit of code, but you need to make sure you test all of your code. The textbook describes a couple of approaches.

Partitioning the test space

One approach that can work here is to "partition" the possible inputs into sets with similar characteristics that would probably cause the same bits of code to run, but the different sets would each trigger different paths through the code or have different results. Then you pick only one example from each set to cover all those cases.

Guidelines

Another approach is to use your experience think of weird values that might trick an algorithm. Think about the problem in general, not just your implementation of the algorithm. As appropriate to the particular code, you should test both minimum and maximum values and a value in the middle, values such as 0, 1, 0.000000000001, the empty string, None, edge cases (is a single letter a palindrome?), and that sort of thing.

Assignment

At this point, you should be able to complete the assignment and write additional test cases. Remember to submit both your code and a screen capture showing that you ran your unit tests.

Advanced, optional additional information

Alternatives to unittest

Unittest is convenient, being built into Python, but it's not the only option. An even more popular alternative, `pytest` can run most unittest TestCases, too. Find it at https://docs.pytest.org/

Automating the automation

It is already great to be able to run so many tests with a single command. This is so simple that running that command can itself be automated.

One of the easiest ways to do this is to set up an action on GitHub to run whenever a branch is pushed. Take a look at https://help.github.com/en/actions/language-and-framework-guides/using-python-with-github-actions -- there are similar workflows for other languages, too.