UNIT TESTING II (PYTEST)

REMINDERS:

- Midterm 1 is coming up: Weds, Oct 7 in class
  - Exam Review session in class on Monday
  - Remember to shcedule accomodated exams with Weingarten ASAP (if you still can). We can't do it for you
  - Midterm Clobber Policy: If you don't do as well as you want, you can get a grade improvement by doing better on the final exam.
- No class Friday (it's break)
- Personality Quiz Presentations next week (after break)

Previously, we've thought about how to come up with test cases to see if our code is running as we expect

```
# returns the most common letter in the string.
# In case of a tie, returns the first of the most common letters.
# Return None if string is empty.
def most_common_letter(input_string: str) -> str:
```

INPUT	EXPECTED OUTPUT	ACTUAL OUTPUT	PASSES?
"Harry"	"R"	"R"	True
"Olivia"	"I"	"O"	False
"Hannah"	"H"	"N"	False

Based on these tests, what does this function seem to actually do? **[L1]**

Let's look at one way we can do this...

```
print(f"When I give input 'Harry',
      I expect 'R', and I get '{most_common_letters('Harry')}'")
print(f"When I give input 'Olivia',
      I expect 'I', and I get '{most_common_letters('Olivia')}'")
```

🖨️👇

```
"When I give input 'Harry', I expect 'R', and I get 'R'"
"When I give input 'Olivia', I expect 'I', and I get 'O'"
```

BUT IT'S ANNOYING TO PRINT EACH ANSWER LIKE THIS, HOW CAN IT BE EASIER?

BUT IT'S ANNOYING TO PRINT EACH ANSWER LIKE THIS, HOW CAN IT BE EASIER?

> **WE CAN USE A TESTING FRAMEWORK** 🎉🎉🎉

(in this class we will use **pytest**)

WHAT DOES THAT MEAN?

We can write each of our tests as *functions*, which keeps them clearly named and organized, then we can run them all at once with a simple command! It looks like this:

```python
import pytest

def most_common_letter():

def test_name_with_adjacent_duplicate():
  test_input = "Harry"
  expected = "R"
  actual = most_common_letter(test_input)
  assert expected == actual
```

```
taliem:~/whatever$ pytest my_file.py
========================= test session starts ==========================
platform linux -- Python 3.11.2, pytest-9.1.1, pluggy-1.6.0
collected 1 item

my_file.py .                                                      [100%]
========================== 1 passed in 0.01s ===========================
```

LET'S BREAK THIS DOWN

```
import pytest # <--- pytest is the name of the framework we're using

def most_common_letter():

def test_name_with_adjacent_duplicate():
  test_input = "Harry"
  expected = "R"
  actual = most_common_letter(test_input)
  assert expected == actual
```

LET'S BREAK THIS DOWN

```
import pytest # <--- pytest is the name of the framework we're using

def most_common_letter(): # <-- The function we're testing

def test_name_with_adjacent_duplicate():
  test_input = "Harry"
  expected = "R"
  actual = most_common_letter(test_input)
  assert expected == actual
```

LET'S BREAK THIS DOWN

```
import pytest # <--- pytest is the name of the framework we're using

def most_common_letter(): # <-- The function we're testing

def test_name_with_adjacent_duplicate(): # <-- A single test
  # ^^ must include the word "test" at the beginning or end of name
  test_input = "Harry"
  expected = "R"
  actual = most_common_letter(test_input)
  assert expected == actual
```

LET'S BREAK THIS DOWN

```
def test_name_with_adjacent_duplicate(): # <-- A single test
  # ^^ must include the word "test" at the beginning or end of name
  test_input = "Harry" # <-- input we're testing
  expected = "R"       # <-- expected output
  actual = most_common_letter(test_input) # <-- actual output
  assert expected == actual
```

Just like before 👇

INPUT	EXPECTED OUTPUT	ACTUAL OUTPUT	PASSES?
"Harry"	"R"	"R"	True

LET'S BREAK THIS DOWN

```
def test_name_with_adjacent_duplicate():
  test_input = "Harry"
  expected = "R"
  actual = most_common_letter(test_input)
  assert expected == actual # <-- What is going on here???
```

ASSERT STATEMENTS

Assert is what we use to say "I claim this to be true, and if it isn't then error out and tell me"

To use it, we first write `assert` and then some *boolean expression*

So in our example...

```
def test_name_with_adjacent_duplicate():
  test_input = "Harry"
  expected = "R"
  actual = most_common_letter(test_input)
  assert expected == actual
  # ^ check if most_common_letter(test_input) == "R" and error if not
```

LET'S GET SOME ASSERTION PRACTICE [C1]

For each of the following assertions, will they pass (be true) or fail (be false)

ASSERTION	PASSES?
assert 3 > 5	
assert "l" in "talie"	
assert int("5") == 5	
assert 5 == "5"	

LET'S GET SOME ASSERTION PRACTICE

For each of the following assertions, will they pass (be true) or fail (be false)

ASSERTION	PASSES?
assert 3 > 5	False
assert "l" in "talie"	True
assert int("5") == 5	True
assert 5 == "5"	False

WORKING TOWARDS WRITING TESTS [L2]

First, let's come up with some reasonable test cases (just like last class)

INPUT	EXPECTED OUTPUT
example 1 here	
example 2 here	
...	

```
# This function removes the first vowel of a given string
# Returns the original string if it contains no vowels
def remove_first_vowel(a : str) -> str:
```

WORKING TOWARDS WRITING TESTS [C2]

Remember, for a test in pytest:

- Define a function whose name starts with "test_" or ends in "_test"
  - Its name should describe what case you are testing for

Choose one of the test cases you wrote before, and write a reasonable function signature for it at the top of **C2**. Your function should take no inputs.

WORKING TOWARDS WRITING TESTS [C2]

Remember, for a test in pytest:

- Define a function whose name starts with "test_" or ends in "_test"
- Include an *input value*, a an *expected output*, and the *actual output* of the function you are testing

Add three variables to your function, an input, an expected output, and one to store the actual output

WORKING TOWARDS WRITING TESTS [C2]

Remember, for a test in pytest:

- Define a function whose name starts with "test_" or ends in "_test"
- Include an *input value*, a an *expected output*, and the *actual output* of the function you are testing
- Make sure you have an **assertion** at the end of your test, to say "this is what I expect to be true"

Add your assert statement of what you are checking to be true

WORKING TOWARDS WRITING TESTS [C2]

Finally, make sure to `import pytest`

You can call your tests from the terminal with `pytest name_of_file.py`

Example output...

```
========================= test session starts ==========================
collected 3 items
test_example.py ..F                                                  [100%]
================================ FAILURES ================================
_____________________________ test_second_letter _____________________________
    def test_second_letter():
>       assert remove_first_vowel("bad") == "bd"
E       assert "bad" == "bd"

test_example.py:11: AssertionError

======================== short test summary info =========================
FAILED test_example.py::test_second_letter - assert "bad" == "bd"
====================== 1 failed, 2 passed in 0.13s =======================
```

LET'S CHAT JUST A BIT MORE ABOUT ASSERTIONS

WHAT HAPPENS IN THESE SCENARIOS?

For each of the following assertions, will they pass (be true) or fail (be false)

ASSERTION	PASSES?
assert	
assert "hello"	
assert 3 = 3	

WHAT HAPPENS IN THESE SCENARIOS?

For each of the following assertions, will they pass (be true) or fail (be false)

ASSERTION	PASSES?
assert	True
assert "hello"	True
assert 3 = 3	error

SOME WEIRD ASSERTION STUFF

`assert __` == pass

- If you do not assert anything (if you just put assert with nothing after it), it will default to passing. You need to give it an expression for it to check!

SOME WEIRD ASSERTION STUFF

`assert __` == pass

- If you do not assert anything (if you just put assert with nothing after it), it will default to passing. You need to give it an expression for it to check!

`assert expected` == ??

- If you just pass a value directly into the assert, without a boolean expression to evaluate, then your value will be cast to a boolean. This does some weird stuff in Python -- 0, "", [] are false, anything with content is true

SOME WEIRD ASSERTION STUFF

`assert __` == pass

- If you do not assert anything (if you just put assert with nothing after it), it will default to passing. You need to give it an expression for it to check!

SOME WEIRD ASSERTION STUFF

`assert __` == pass

- If you do not assert anything (if you just put assert with nothing after it), it will default to passing. You need to give it an expression for it to check!

`assert expected` == ??

- If you just pass a value directly into the assert, without a boolean expression to evaluate, then your value will be cast to a boolean. This does some weird stuff in Python -- 0, "", [] are false, anything with content is true

errors in your test

- If your test, or the assertion statement, errors out, then the test will fail without reaching the assert

LAST BIT OF PRACTICE [C3]

Here is a proposed function. Write 3 unit tests for it using pytest syntax.

```
# Replaces the middle character of a string msg with a given char
# If msg is an even number of letters, chooses the smaller index
# If msg is empty, return None
def replace_middle_character(msg: str, char : str) -> str:
```

Remember: write *function signature*, determine *input*, *expected output*, *actual output*, and then include an *assertion statement*

WHY USE PYTEST INSTEAD OF JUST PRINTING?

If we were able to get similar effects by just printing? Why use pytest?

- Helps keep the code organized: Which stuff is for testing and which stuff is part of the actual program
- Helps keep the tests organized: What is each test checking for? Why did I include it?
- The test can check the output for us, so we only have to keep in mind what the "right" answer is once -- when we write the test