Learning Outcomes: Throughout this lab, you will work to:

- develop software using unit testing and test-driven development (TDD)
- understand the importance of regression testing
- appreciate that software development is *rarely* a linear process
- appreciate that TDD can help you to write better software from the start
- gain experience using pytest and pytest fixtures
- understand how Python decorators can be used effectively to extend/enhance an existing function without modifying the original function

---

Overview: In this lab, you will use pytest to conduct test-driven development (TDD), while writing the method implementations for a class that represents a `String` class.

Assignment:

1. **Download:** Starter code for both `String.py` and `test_String.py` (see below) are provided on Lyceum.

   Note that `String.py` already has all stubs for your class implemented, using Google docstring style for documentation, and also include Python type hints (see, e.g., https://realpython.com/lessons/type-hinting). Implementations of the bodies of these methods should be fairly straightforward and concise — consistent with the focus of this lab on TDD.

   More discussion on implementations within `test_String.py` will be provided below.

2. **Setup:** Start by setting up an appropriate directory structure for pytest so that you can separate your development code from your testing code. Below is one such structure:

   ```
   .
   |-- code_base
   |   |-- String.py
   |   |-- __init__.py
   |-- tests
       |-- __init__.py
       |-- test_String.py
   ```

3. **Read:** Read carefully through each of the methods and docstring documentation in `String.py`, and make sure you understand what each method should accomplish. Consistent with TDD, do not write the code for these methods yet. Plan to return back to reading the docstring documentation as you perform TDD later.

4. **Read some more:** Now read through the `test_String.py` testing file, to be used with pytest. I have already provided you with a number of things in this file, each discussed briefly below, but with many more details in comments in the code (so read them!).

   (a) `class print_test`: I have provided a class that will act like a function to streamline helpful printing of tests within your pytest unit-test functions. To use this "function", you simply call `print_test`, passing it a string version of the test you are conducting, the actual result of the test, and the expected result of the test. Then `print_test` should do the right thing to make for an easy-to-read output when you use `pytest -s tests`.

   Below is an example pytest unit-test call to `print_test` and the corresponding output that results when you run pytest with the `-s` flag. (You'll see a few examples similar to these unit-test calls already in `test_String.py`.)

```
# calls to print_test inside your pytest unit-test functions:
result = String(empty_string)._chars   # argument is pytest fixture
print_test('String("")._chars', result = result, expected = [])

...

result = String(random_string) == String(different_random_string)
print_test(f'String("random_string") == String("different_random_string")',
           result = result, expected = False)



# resulting pytest output:
Test 0: String("")._chars
     Result:   []
     Expected: []

...

Test 10: String("<t<v_M=0$9:ElAh") == String("]KCw@I-i")
      # indices: 012345678901234              01234567
      Result:   False
      Expected: False
```

(b) pytest fixtures: I have provided a few examples of well-documented pytest fixtures. Recall from class that fixtures are used to perform setup for data (and steps, as appropriate) that will get used throughout your subsequent test functions. Read through these fixtures and their comments carefully. While you likely will want to define more, I have provided you already with the following fixtures:

- `empty_string`: returns an empty string
- `empty_list`: returns an empty list
- `characters`: returns a string of characters used in creating random string
- `random_string`: returns a randomly-generated character string — note that the `characters` fixture is passed (automatically for you — the beauty of decorators) as an argument
- `different_random_string`: returns a different randomly-generated character string (e.g., for use in later testing the `__eq__` method)

Feel free to modify or remove any of these — they are your fixtures.

(c) pytest unit-test functions: I have also provided a few examples of well-documented pytest unit-test functions. Make sure to read carefully through these so you can see how I am recommending that you structure your unit tests in pytest.

- Recall that pytest requires each unit-test function to begin with `test_`.
- Also notice how previously defined fixtures are passed (automatically for you, by pytest) as arguments. Even though the fixture looks like a function that you should call, pytest doesn't work that way — just using the name of the fixture will evaluate to whatever value the fixture returns.
- I have given you a few different examples of how you can easily construct your unit tests, including calls to the `print_test` "function". Just set up your actual result, the expected result, call `print_test`, and then use the assert required by pytest. Read the comments and examples provided!
- I have also given you a couple of examples of how, in pytest, you handle tests that will result in exceptions being raised (e.g., trying to access any character in an empty string). This is done by using `with pytest.raises(`*ExceptionType*`)` and some corresponding information that provides you. Again, read the comments and examples provided!

5. **Conduct TDD:** OK, you should now be at the point where you understand the requirements of the `String` class, use of the provided `print_test` and fixtures, and how to nicely set up your unit tests. You can now begin actually writing some code, using TDD. Recall the process:

- Write one unit-test function in `test_String.py`. **Your** unit-test function should have meaningful documentation. Use the provided examples as guides.
- Show that the test fails, by running `pytest -s tests`.
- Write just enough code in `String.py` to make that test pass.
- Show that the test now passes.
- Refactor your code in `test_String.py` and re-test.
- Repeat.

Implement all of the methods inside the `String` class using this approach. While there is no "perfect" number of tests, you definitely will need at least dozens (that's multiple of a dozen).

Have fun, and enjoying thinking carefully about tests, and knowing that you're developing skills that will be useful well beyond DCS 229 and beyond Bates!

Submitting: When you're ready to submit, start by running all of your tests one last time to make sure that you didn't inadvertently change anything last minute. Then drop your well-documented `String.py` and `test_String.py` implementations into Lyceum. Only one submission per group is required.

➡ ➡ Make sure all group member's names appear in comments at the top of `String.py` and `test_String.py`. ⬅ ⬅

Randomly-selected groups: Below are the randomly-selected groups for this assignment. Because the number of students is neither a multiple of two nor three, occassionally you will need to work in pairs rather than triples.

For those who are working in pairs for this assignment, you are not required to implement `__add__` or `substring`. Of course you can choose to do so (and I recommend it, if you have the time) for your own experience.

Decide among your group how you would like to communicate and work together, and how you will split up the work. **Please make sure that every team member is contributing meaningfully to the work.**

- Abdullah, Nate, Chrissy
- Sam, Jesper, Amanda
- Kona, Dieter, Ken
- Mia, Matteo, Jesse
- Adrian, Joseph, Garrett
- Biruk, John, Clem
- Max, Anh, Thomas
- Nathan, Liza
- Lucas, Edgar