CL23: More on Writing Automated Tests for Functions

Announcements

- **LS12 – Unit Tests** due today at 11:59pm
- **Quiz 02** this Friday!
  - Have a University-Approved Absence? Email me to schedule an alternate time to take it!
  - Practice problems on the course site
    - Questions? Visit office hours/tutoring!
  - Hybrid review session today from 6-7pm in Sitterson Hall (SN) 014 and online
    - Recording will be posted to the site

Review: Writing a Function that uses a list, set, and dict!

Together, we'll write a function named **`bin_len`** that "bins" a list of strings into a dictionary, where the key is an **`int`** length of a given string and the associated values are a **`set`** of strings of the key's length found in the original list.

For example:

`bin_len(["the", "quick", "fox"])` returns `{3: {"the", "fox"}, 5: {"quick"}}`

`bin_len(["the", "the", "fox"])` returns `{3: {"the", "fox"}}`

Big idea: We can write functions that validate the correctness of other functions!

In software, this concept is called ***testing.***

Testing at a *function-level* is generally called *unit* testing in industry (a *unit* of functionality)

A. Helps you confirm correctness during development
B. Helps you avoid accidentally breaking things that were previously working (regressions)

The strategy:

1. Implement the "skeleton" of the function you are working on
   (function name, parameters, return type, and some dummy (wrong/naive!) return value)
2. Think of examples use cases of the function and what you expect it to return in each case
3. Write a test function that makes the call(s) and compares expected return value with actual
4. Once you have a failing test case running, go correctly implement the function's body
5. Repeat steps #3 and #4 until your function meets specifications

This gives you a framework for knowing your code is behaving as you expect

Review: Steps to set up a *pytest* Test Module

To test the function definitions of a module:

1. Create a sibling module (a different file) with the same name, but ending in `_test`
   a. Example name of definitions module: `dictionary.py`
   b. Example name of test module: `dictionary_test.py`
   c. This convention is common to `pytest`
2. In the test module, import the function definitions you'd like to test
   a. Example: `from cl.cl22.dictionary import bin_len`
3. Next, add tests which are procedures whose names begin with `test_`
   a. Example test name: `test_bin_len_empty`
4. To run the test(s), you have two options:
   a. In a new terminal: `python -m pytest <path/to/testfile.py>`
   b. Use the Python Extension in VSCode's Testing Pane (the beaker icon)

We wrote the skeleton for the **bin_len** function:

```
def bin_len(words: list[str]) -> dict[int, set[str]]:
    """Sort the elements of a list into a dict based on their lengths."""
    result: dict[int, set[str]] = {}
    return result
```

Review: Writing a unit test

Test file names: end with _test.py
Test function names: begin with test_

```
def test_name() -> None:
    # Other code can go here!
    assert <boolean expression>
```

Last Lecture: Testing For Correct Return Values

- Checking that your function returns what you expect it to for a given function call

```
def test_bin_len_empty() -> None:
    """Test return value when input list is empty."""
    assert bin_len([]) == {}
```

Testing For Correct Behavior

- Checking that your function does what you want it to do rather than just checking what it returns.
- This can be useful for functions that *mutate* their input.

Example in VSCode…

Testing For Correct Errors for Invalid Inputs

```
def test_<rest_of_name>():
    with pytest.raises(<Error Type>):
        function_name(<invalid argument>)
```

Example in VSCode…

One version of the solution:

```
def bin_len(words: list[str]) -> dict[int, set[str]]:
    """Sort the elements of a list into a dict based on their lengths."""
    result: dict[int, set[str]] = {}
    for w in words:
        word_len: int = len(w)
        if word_len in result:
            result[word_len].add(w)
        else:
            result[word_len] = {w}
    return result
```