



CL22: Importing and Writing  
Automated Tests for Functions

# Announcements

- **EX05 – Dictionary Utility Functions** due today (3/9) 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 Wednesday evening (time and location TBA on the site today)

# Code-along: 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"}}
```

Before we write any code: what concepts from the course will we need to use?

# Test-driven function-writing

Before writing a function, it's helpful to focus on concrete examples of *how the function should behave as if it were already implemented*.

Key questions to ask:

1. What are some usual arguments and expected return values?
  - a. These are the *use cases* or *expected cases*
2. What are some valid, but unusual arguments and expected return values?
  - a. These are your *edge cases*
  - b. Example: empty inputs, incorrect inputs

Below are the examples of the `bin_len` function. Which of these represent use cases and edge cases, respectively?

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

```
bin_len([]) returns {}
```

```
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

# Testing is no substitute for critical thinking...

- Passing your own tests does not guarantee your function is correct!
  - Your tests must validate a useful range of cases
    - “Will my function behave correctly for every possible input(s)?”
  - It’s possible for your unit tests to be incorrect (!)
- Rules of thumb:
  - Test  $\geq 2$  use cases and  $\geq 1$  edge case per function
  - When a function has if-else statements, or loops, write a test per branch/body

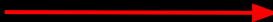
Let's write the skeleton for the `bin_len` function!

```
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
```

Let's write a test function for the `bin_len` function!

# 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) 



# Syntax: Writing a unit test

Test file names: end with `_test.py`

Test function names: begin with `test_`

def `test_name()` -> None:

```
    assert <boolean expression>
```

# Syntax: 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>`

*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
```

## Next Lecture: Testing For Desired 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...

Syntax: Expecting an error

```
>>> with pytest.raises(ZeroDivisionError) :  
...     1/0
```