Testing

Today starts as a Paper + Pencil or Tablet + Pencil day... please keep laptops stowed away!

COMP110 - CL11

2024/02/29

Announcements

- EX03 - Wordle
  - Due Tomorrow at 3/1 at 11:59pm

- EX04 - List Utility Functions
  - Out by Sunday 3/3, Part 1 Due by 4/8, Part 2 Due by 4/19

- Quiz 2 - Thursday 3/7

Warm-up

Trace a Memory Diagram

```python
def swap(t: list[int], i: int, h: int) -> list[int]:
    """Returns a copy of t with indices i and h swapped"""
    assert i < len(t) and h < len(t)
    copy: list[int] = t.copy()
    temp: int = copy[i]
    copy[i] = copy[h]
    copy[h] = temp


values: list[int] = [10, 20, 30]
result: list[int] = swap(values, 0, 2)

print(values)
print(result)
```

Common, Useful Methods of the List Class

Feature	Method Name	Example
Append item to a List	append	a_list.append(item) - item is type T of with a_list: list[T]
Remove item from a List	pop	a_list.pop(index) - index is an integer
Copy a List	copy	a_list.copy()
Count Items in a List	count	a_list.count(item) - returns number of occurrences of `item`
Reverse a List	reverse	a_list.reverse() - mutates a_list, reversing its elements
Find Index of a Value	index	a_list.index(item) - returns the index of `item` in list, ValueError otherwise
Clear Items of a List	clear	a_list.clear() - clears all items from list, mutating the list

Test-driven Function Writing

- Before you implement a function, 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?
   - These are your *use cases* or *expected cases*.
2. What are some valid but unusual arguments and expected return values?
   - These are your *edge cases*.

Using Wishful Thinking

Big Idea: Functions can validate the correctness 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
   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

- Rules of Thumb:
  - Test >= 2 use cases and >= 1 edge case per function
  - When a function has if-else statements, or loops, write one test per branch/body

Setting up a *pytest* Test Module

To test the definitions of a module, first create a sibling module with the same name, but ending in _test
- Example name of definitions module: lecture.cl11_module
- Example name of tests module: lecture.cl11_module_test
- This convention is common to pytest

Then, In the test module, import the definitions you'd like to test

Next, add tests which are procedures whose names begin with test_
Example test name: test_total_empty

To run the test(s), two options:
In a new terminal: pytest [package_folder/python_module_test.py]
Use the Python Extension in VSCode's Tests Pane