

Hack110 Sign-Up Form!

When? Saturday, April 5th from 10 AM - 12 AM (Midnight)

Where? In Sitterson Lower Lobby

Who can join? Anyone in COMP 110! No prior experience required. Bring a partner or come as yourself (we'll have team-building activities if you want a partner)

Come for a fun day of coding, workshops and events (**food and CLE credit will be provided**):

- Choose between web development or game development track
- Go to various **workshops & events** such as: Navigating the CS Major, Resume workshop, ice cream station, and kahoot trivia and MORE!
- Link: Sign-Up Here! Or via the QR code 
- **Sign-Up form EXTENDED TO Monday, March 31st at 11:59 pm**
 - Spots are limited! So we'll prioritize interest!
 - If you have a partner, **ONLY ONE OF YOU** has to sign up - you will just enter your partner's info in the form.

Sign-Up Here!





CL21: Importing and Writing Automated Tests for Functions

Announcements

- Quiz 02 will be returned soon!
- Quiz 03 on Friday, March 28
 - Reminder to schedule with ARS or reach out to me if you have a University Excused Absence and need to take it at another time
- EX03 will be released on Wednesday

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 REPL examples of the **count_regs** function we wrote in the previous lecture. Which of these represent use cases and edge cases, respectively?

```
1 >>> count_regs("Orange", ["Wake", "Orange", "Orange", "Durham"])
2 2
3 >>> count_regs("Wake", ["Wake"])
4 1
5 >>> count_regs("Orange", ["Durham"])
6 0
7 >>> count_regs("Lee", [])
8 0
```

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
 - It's possible for your unit tests to be incorrect
- Rules of thumb:
 - Test ≥ 2 use cases and ≥ 1 edge case per function
 - When a function has if-else statements, or loops, write a test per branch/body

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: `lecture.cl20_module.py`
 - b. Example name of test module: `lecture.cl20_module_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 cl20_module import count_regs`
3. Next, add tests which are procedures whose names begin with `test_`
 - a. Example test name: `test_count_regs_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:

 # Other code can go here!

`assert <boolean expression>`

For reference:

Question 6: Function and Unit Test Writing Write a function definition that calculates the total number of people who are registered to vote in a specific county. It should take in a string representing the county of interest and a list of strings representing the counties in which a group of people are registered, and return the count of people who are registered in the county of interest.

The function should satisfy the following expectations:

- The function name is `count_regs`
- It has two parameters: a `str` representing the county of interest, and a `list[str]` representing the counties in which a group of people are registered
- The function returns an `int` of the total number of people who are registered in the county of interest
- The function should not modify the input list
- The function should have the docstring: Count number of people who are registered in the specified county.
- Explicitly type variables, parameters, and return types
- The following REPL examples demonstrate the expected functionality of your `count_regs` function:

```
1 >>> count_regs("Orange", ["Wake", "Orange", "Orange", "Durham"])
2
3 >>> count_regs("Wake", ["Wake"])
4 1
5 >>> count_regs("Orange", ["Durham"])
6 0
7 >>> count_regs("Lee", [])
8 0
```

```
def count_regs(coi: str, counties: list[str]) -> int:
    """Count number of people who are registered in the
    specified county."""
    idx: int = 0 # Current index in counties list
    total: int = 0 # Total occurrences of county of
    interest
    while idx < len(counties):
        if counties[idx] == coi:
            total += 1
        idx += 1
    return total
```