

Name: \_\_\_\_\_ UCSD Email: \_\_\_\_\_  
Partner's Name: \_\_\_\_\_ Partner's UCSD Email: \_\_\_\_\_

# CSE 190: Working with Large Codebases Fall 2026 (Gerald Soosairaj)

## Lecture 6: Writing Tests for a Changed Feature

---

Thursday, October 15, 2026

**NOTE:** To get lecture credit, submit this handout on Gradescope. Write your answers by hand on the lines, then scan the pages. You can finish it after class. **Deadline: Thursday, October 22, 2026, 2:00pm (one week from today).**

## LAST CLASS

---

1. Running IDLE's tests with the Python you built
2. A failing test: is the code wrong or is the test out of date?
3. Looking for side effects, and reviewing your own diff

## TODAY

---

1. Activity: Test your Go To Line change from Assignment 1

## Announcements

---

## Before You Start

---

You need the setup from Assignment 1. **Use your pair's A1 repository** (`idlelib-[user1]-[user2]`) and the Python you built there. Today you work on your own A1 code, not on `cpython_lectures`.

1. **Go to your pair's repository and check out your A1 branch.**

```
git switch assignment1
```

2. **Make a new branch for today's work.** This keeps your A1 pull request untouched. **Do not push this branch and do not commit to assignment1.**

```
git switch -c lecture6-tests
```

3. **Check that your Go To Line change works.** Start IDLE from the top folder of your repository (`./python -m idlelib`), open a file, and try the Go To Line dialog. If your A1 code is not finished, finish the main parts first. Today's questions need working code.

**Reminder on AI tools (A1 and A2 rules apply today too):** Web chat tools (ChatGPT, Gemini) may be used only to look up documentation or to explain existing IDLE code. Do not use them to write tests, write or review your code, or answer these questions. Agentic and IDE-integrated AI tools are off. Write your answers

in your own words.

**Your code may differ from your classmates'.** You each wrote Go To Line in your own way, so your test results will differ too. That is fine. Answer for **your** code.

---

## 1. Activity: Test Your Go To Line Change

---

In Lecture 5 you ran the tests after changing the shell prompt, and some of them failed. Today you do the same for your Go To Line change from A1. First you check **which existing tests your change breaks**. Then you **write the new tests the feature needs**. A feature is not done until the tests say what it should do.

Run all commands from the **top folder** of your repository, with the Python you built. Use `./python -m unittest ....` On macOS use `./python.exe`. (This is different from Lecture 5. The built Python finds its own Lib folder, so you do not need to `cd` into it.)

### Part A. Find the tests for the feature

1. Which file in `Lib/idlelib/idle_test/` tests the Go To Line dialog? Which class in that file tests Go To Line **without opening a window**? (The file has a second Go To Line class that opens a real dialog. It may fail with your change. We will focus on GUI tests in a later lecture, so ignore it today.) Write what you searched for to find them.

---

---

---

2. Read the existing tests in that class. Fill in the table.

| Test name       | What does the test type or pass in? | What does it check? |
|-----------------|-------------------------------------|---------------------|
| test_bogus_goto |                                     |                     |
| test_bad_goto   |                                     |                     |
| test_good_goto  |                                     |                     |

3. The class does **not** open a window. It uses a small fake class made just for the test. What does the fake provide? Why might the authors prefer this to opening a real dialog? (Hint: look at the top of the file.)

---

---

---

### Part B. Predict and run

1. Before you run anything. **Which of the three existing tests do you think will fail with your change?**  
Say why for each one you pick.

---

---

---

2. Run that class with `-v`. Write the command you used.

---

3. Fill in the results.

| Test name       | ok, FAIL, or ERROR |
|-----------------|--------------------|
| test_bogus_goto |                    |
| test_bad_goto   |                    |
| test_good_goto  |                    |

**FAIL and ERROR are different.** A FAIL means a check in the test was false. An ERROR means the code raised an exception before the check was reached. Read a traceback **from the bottom up**: the last lines say what went wrong, and the lines above say where.

4. Pick one test that did not pass. Write the last line of its traceback, and the line of **your** code (file and line number) where it went wrong.

---

---

---

5. Did your prediction match? If not, what did you miss?

---

---

## Part C. Decide and fix

1. For each test that did not pass: is your **code** wrong, or is the **test** out of date? Argue it in two or three sentences. The question to ask is: *what did the test assume that is no longer true?*

---

---

- 
- 
2. Update the old tests so they pass **and still check what they checked before**. Do not delete a test. Do not skip a test. Do not change an expected value unless the feature itself changed it. Then run the class again. Write the numbers from the last lines.

- 
3. Run `git diff` on the test file. In your own words, what did you change in the test file, and why was that the right place to change it?
- 
- 

## Part D. Decide what needs testing

You cannot test every input. You pick the ones where bugs are likely: **the edges**. Suppose the file in the editor has **10 lines**. Fill in the table for your feature. Use the A1 requirements.

| Typed in the box | What should happen? (the line you go to, or the message) | Which existing test checks it? (name, or "none") |
|------------------|----------------------------------------------------------|--------------------------------------------------|
| 1                |                                                          |                                                  |
| 10               |                                                          |                                                  |
| 11               |                                                          |                                                  |
| -1               |                                                          |                                                  |
| -10              |                                                          |                                                  |
| -11              |                                                          |                                                  |
| 0                |                                                          |                                                  |
| abc              |                                                          |                                                  |
| (empty box)      |                                                          |                                                  |

1. The rows 10, 11, -10, and -11 are next to the edges of the range. Why do bugs like to hide there?
- 
- 

2. Pick the two rows you think are most **likely** to hide a bug in code like yours. Say why.
- 
-

## Part E. Write the new tests

Add tests to the same test file. **Follow the conventions around you:** put each test in the class you found in Part A, name it `test_` plus what it checks, and copy the setup style of the tests next to it. Each test checks **one idea**. Check the result **and**, for errors, the message text. (The error text may have a prefix such as `ERROR:`. A check that the message **contains** the required text is enough.)

Write:

- **At least five** new tests. Choose them from your Part D table. Cover at least: the last line, one number past the end, a negative number in range, and a number too negative.

1. List the names of your new tests, and in one phrase say what each checks.

---

---

---

---

---

---

2. Run the class. Write the numbers from the last lines. If a new test failed on the first run, was your **test** wrong or your **code** wrong? How did you decide?

---

---

---

---

3. Commit your tests on the `lecture6-tests` branch (`git add` the test file, then `git commit`). Write your commit message here. Remember: say **why**, not only what.

---

---

---

---

## Part F. Can your tests catch a bug?

A test that cannot fail is not a test. Check yours by **breaking your code on purpose**. Your tests are committed, so you can undo each change with `git restore <file>`.

1. **Break 1:** In your code, change one comparison at an edge of the range so it also accepts the edge value (for example, turn “greater than” into “greater than or equal to”). Run the Go To Line tests. Which test failed? If **none** failed, which test is missing? Undo the change.

---

---

---

2. **Break 2:** In your code, change how a negative number is turned into a line number so the answer is off by one. Run the Go To Line tests. Which tests failed? Undo the change.

---

---

---

3. If any break was **not** caught, write the extra test you added and run it. Run the Go To Line tests one last time with your code restored. Write the numbers.

---

---

---

Remember to submit on Gradescope by **2:00pm on Thursday, October 22, 2026**.

“Program testing can be used to show the presence of bugs, but never to show their absence!” Edsger W. Dijkstra, *Notes on Structured Programming*

*Acknowledgment: This handout was co-created by Gerald Soosairaj and Claude Sonnet 5.5. Gerald chose the example and wrote the content and structure. Claude helped with brainstorming and formatting the handout in markdown, based on his inputs.*