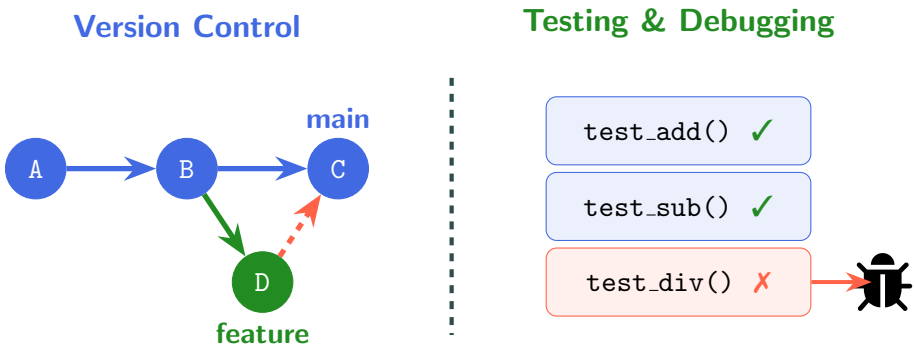
Programming 2: Lab 02

Version Control with Git & Testing/Debugging

A Walkthrough Tutorial



Comp 111 — Forman Christian College

Spring 2026

Estimated Time: ∼**3 hours**

How to Use This Lab

This lab is a **walkthrough tutorial**—it is designed to guide you through learning, not to test you. Work through each section at your own pace:

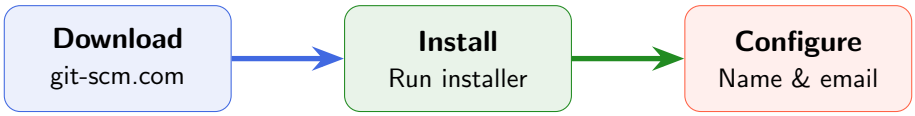
- **Read** the short explanations and study the diagrams.
- **Type** every code listing yourself (don't copy-paste!). Muscle memory matters.
- **Run** every example and check that your output matches.
- **Complete** the exercises. They are graded by difficulty:
  - ★ Easy — Immediate practice. Follow the pattern you just saw.
  - ★★ Medium — Requires some thinking. Combines concepts.
  - ★★★ Hard — Stretch goal. It's OK to need help!
- **Ask for help** whenever you're stuck for more than 5 minutes.

Before You Begin

This lab assumes you completed Lab 01. You should be comfortable opening the **Command Prompt**, navigating with `cd`, creating folders with `mkdir`, and running Python files with `python filename.py`. You also need **Git** installed and a free **GitHub** account—Section 1 will walk you through both.

1 Installing & Configuring Git

Before we can use **Git**, we need to install it and tell it who we are. Git attaches your name and email to every **commit** (save point) so that teammates know who made each change.



1.1 Installation Steps (Windows)

1. Go to https://git-scm.com and click **Download for Windows**.
2. Run the installer. Accept all defaults—just keep clicking **Next**.
3. When it finishes, open a **new** Command Prompt and verify:

```
git --version
```

You should see something like:

```
git version 2.44.0.windows.1
```

Must Open a New Terminal

If `git --version` says `'git' is not recognized`, close your terminal and open a **new** one. The installer updates your system PATH, but only new terminals pick it up.

1.2 Configuring Your Identity

Tell Git who you are (you only need to do this once):

```
git config --global user.name "Your Name"
git config --global user.email "your.email@example.com"
```

Verify your settings:

```
git config --list
```

```
user.name=Your Name
user.email=your.email@example.com
```

Use Your Real Email

Use the same email address you used (or will use) for your GitHub account. This links your local commits to your GitHub profile.

1.3 Creating a GitHub Account

If you haven't already:

1. Go to `https://github.com` and click **Sign Up**.
2. Choose a professional-looking username (future employers will see it!).
3. Verify your email address.

1.4 Exercises: Git Setup

1. ★ Easy Run `git --version` in your terminal. Write down the version number you see.
2. ★ Easy Run `git config --global user.name "Your Name"` with your actual name, then verify with `git config user.name`.
3. ★★ Medium Git has many configuration options. Run `git config --list` and count how many settings are shown. Can you spot `user.name` and `user.email` in the list?
4. ★★ Medium Research: what does `git config --global core.editor "notepad"` do? Run it and explain in one sentence.

2 Your First Repository

A **repository** (or **repo**) is simply a folder that Git is tracking. Once you **initialize** a repo, Git watches every change you make inside it and lets you save snapshots called **commits**.

Repository = Tracked Folder

Think of a repository as a folder with a built-in time machine. Every time you **commit**, Git takes a snapshot of all your files. You can travel back to any snapshot at any time.

The **two-step save** is the heart of Git: first you **stage** files (choose what to include), then you **commit** (actually save them). It's like putting items in a shopping cart and then checking out.

2.1 Essential Git Commands

Command	Short description	What it does
`git init`	Initialize	Creates a new repo in the current folder
`git status`	Check status	Shows which files are changed/staged/untracked
`git add file`	Stage a file	Puts the file in the staging area
`git add .`	Stage all	Stages every changed file
`git commit -m "msg"`	Commit	Saves a snapshot with a message
`git log`	View history	Lists all commits
`git log --oneline`	Short history	One commit per line
`git diff`	See changes	Shows what changed since last commit

2.2 Walkthrough: Creating and Committing

Create a project folder and initialize a repo:

```
mkdir git-lab
cd git-lab
git init
```

```
Initialized empty Git repository in C:/.../git-lab/.git/
```

Now create a Python file. Open your text editor and type the following code into a file called `greet.py`:

```python
# greet.py
def greet(name):
    return f"Hello, {name}! Welcome to Git."

print(greet("World"))
```

Save the file inside your `git-lab` folder, then check Git's status:

```
git status
```

```
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        greet.py

nothing added to commit but untracked files present
```

Git sees the file but isn't tracking it yet. Let's stage and commit:

```
git add greet.py
git commit -m "Add greeting function"
```

```
[main (root-commit) a1b2c3d] Add greeting function
 1 file changed, 5 insertions(+)
 create mode 100644 greet.py
```

Now let's verify our history:

```
git log --oneline
```

```
a1b2c3d Add greeting function
```

Congratulations—you just made your first commit!

Write Good Commit Messages

A good commit message says *what* you changed and *why*. Compare these:
Bad: `"fixed stuff"`, `"update"`, `"asdfgh"`
Good: `"Add greeting function"`, `"Fix off-by-one error in search"`, `"Add unit tests for calculator"`

2.3 Exercises: First Repository

1. ★ Easy Create a new file called `goodbye.py` that contains a function `goodbye(name)` which returns `"Goodbye, <name>!"`. Stage it with `git add goodbye.py` and commit with the message `"Add goodbye function"`.
2. ★ Easy Run `git log --oneline`. You should now see two commits. Write down both commit messages.
3. ★★ Medium Edit `greet.py`: change the return string to include an exclamation mark and a smiley face. *Before* staging, run `git diff` and observe the output. What do the `+` and `-` symbols mean?
4. ★★ Medium Stage and commit your edit from Exercise 3 with an appropriate message. Then run `git log --oneline` and confirm you now have three commits.
5. ★★ Medium Create two new files: `file_a.py` and `file_b.py`. Use `git add .` to stage both at once. Verify with `git status` that both are staged, then commit.
6. ★★★ Hard Run `git log` (without `--oneline`). Study the full output. Each commit shows a long hexadecimal string (the **commit hash**), the author, the date, and the message. Why do you think Git uses these hashes instead of simple numbers like 1, 2, 3?

3 Undoing Mistakes

One of Git's greatest powers is the ability to **undo** changes. There are two main tools for this: **git restore** and **git revert**. They solve different problems.

`git restore`

Undo *local* changes (before committing)

No new commit created
History unchanged

`git revert`

Undo a *committed* change (after committing)

Creates a new commit
History preserved

3.1 Walkthrough: git restore

Let's deliberately break a file and then rescue it:

```
cd git-lab
```

Open `greet.py` in your editor and replace the entire contents with:

```
# greet.py -- BROKEN!
print("I accidentally deleted everything!")
```

Save the file. Now check:

```
git status
```

```
modified:   greet.py
```

Don't panic! Restore the file to the last committed version:

```
git restore greet.py
```

Open `greet.py` again—it's back to normal!

Restore Discards Changes Permanently

`git restore` throws away your uncommitted edits. There is no "undo the undo." Make sure you really want to discard your changes before running it.

3.2 Walkthrough: git revert

What if you already *committed* a bad change? Let's simulate that:

```
echo "This line is a mistake" >> greet.py
git add greet.py
git commit -m "Add a mistaken line"
```

Now view your log:

```
git log --oneline
```

```
f7g8h9i Add a mistaken line
... (earlier commits)
```

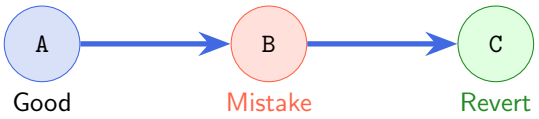
To undo this commit safely:

```
git revert f7g8h9i
```

Git opens your editor to write a revert message. Save and close. Now check:

```
git log --oneline
```

```
b2c3d4e Revert "Add a mistaken line"
f7g8h9i Add a mistaken line
... (earlier commits)
```



All three commits remain in history!

The key difference: `git revert` doesn't erase history—it creates a *new* commit that undoes the old one. This is safe to use even when working with teammates.

3.3 Exercises: Undoing Mistakes

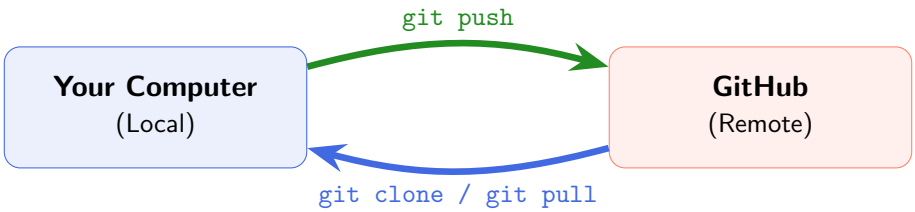
1. ★ Easy Edit `goodbye.py` to break it (delete a line or add garbage text). Then use `git restore goodbye.py` to fix it. Verify the file is back to normal.
2. ★ Easy Run `git status` after restoring. What does it say?
3. ★★ Medium Make a change to `greet.py`, stage it with `git add`, but do *not* commit. Run `git status`—the file is shown as staged. Now run `git restore --staged greet.py`. What happened? Run `git status` again to see.
4. ★★ Medium Make a deliberate bad commit (e.g., add the line `print("BUG")` to any file, stage, and commit). Then use `git revert` to undo it. Verify with `git log --oneline` that both the bad commit and the revert commit appear.
5. ★★★ Hard What happens if you try to `git restore` a file that has no changes? Try it and explain the result.
6. ★★★ Hard Explain in your own words: why is `git revert` safer than just deleting the bad code and committing again? Think about what happens when you're working with a teammate.

4 Working with GitHub

So far, your repository only exists on *your computer*. **GitHub** is a website that hosts repositories online, letting you back up your code, share it, and collaborate with others.

Git vs GitHub

Git is the tool on your computer. **GitHub** is the website where you store and share repositories online. They are related but separate—like video files vs. YouTube.



4.1 The GitHub-First Workflow

The most common way to start a project with GitHub:

Step	Command	What happens
1	(GitHub website)	Create a new repository on github.com
2	`git clone` *url*	Download the repo to your computer
3	(edit, add, commit)	Work locally as normal
4	`git push`	Send your commits to GitHub

4.2 Walkthrough: Clone, Edit, Push

Step 1. Create a repo on GitHub. Go to https://github.com, click the green **"New"** button (or the + icon). Name it `lab02-practice`. Check **"Add a README file"**, then click **"Create repository"**.

Step 2. Clone it to your computer. Copy the HTTPS URL from the green **"Code"** button on GitHub, then:

```
cd C:\Users\YourName\Desktop
git clone https://github.com/YourUsername/lab02-practice.git
cd lab02-practice
```

```
Cloning into 'lab02-practice'...
done.
```

Step 3. Create a file and commit. Create a file called `hello.py`:

```python
# hello.py
print("Hello from my first GitHub repo!")
```

Stage and commit:

```
git add hello.py
git commit -m "Add hello script"
```

Step 4. Push to GitHub:

```
git push
```

```
Enumerating objects: 4, done.
Writing objects: 100% (3/3), 289 bytes | 289.00 KiB/s, done.
```

```
To https://github.com/YourUsername/lab02-practice.git
   abc1234..def5678  main -> main
```

Now go to your GitHub repository page in your browser and refresh—you should see `hello.py`!

⚠ Authentication — GitHub Requires a Personal Access Token

GitHub **no longer accepts passwords** for Git operations over HTTPS. Instead, you must use a **Personal Access Token (PAT)**. When Git asks for your password, paste your token instead.

How to create a Personal Access Token:

1. Go to https://github.com/settings/tokens (or: GitHub → click your profile picture → **Settings** → **Developer settings** → **Personal access tokens** → **Tokens (classic)**).
2. Click **"Generate new token"** → **"Generate new token (classic)"**.
3. Give it a descriptive name (e.g., `comp111-lab`).
4. Set an expiration (e.g., 90 days).
5. Under **Scopes**, check the **repo** checkbox (this grants access to your repositories).
6. Click **"Generate token"** at the bottom.
7. **Copy the token immediately!** You will not be able to see it again.

When you push for the first time:

```
Username for 'https://github.com': YourUsername
Password for 'https://YourUsername@github.com': <paste your token
    here>
```

The token will look like `ghp_xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx`. Paste it at the password prompt (it will not show on screen—just paste and press Enter).

Save Your Token

To avoid entering your token every time, you can tell Git to remember it:

```
git config --global credential.helper store
```

After running this command, Git will save your credentials the next time you enter them, so you won't have to paste the token again.

4.3 Exercises: Working with GitHub

1. ★ Easy Add a second file called `about.txt` containing your name and your favorite programming language. Stage, commit with a good message, and push to GitHub. Verify it appears on the website.
2. ★ Easy Run `git remote -v` in your cloned repo. What does it show? What does `origin` mean?

3. ★★ Medium Make three separate commits (edit a file, commit, repeat three times). Then push with a single `git push`. Check GitHub—do all three commits appear in the history?

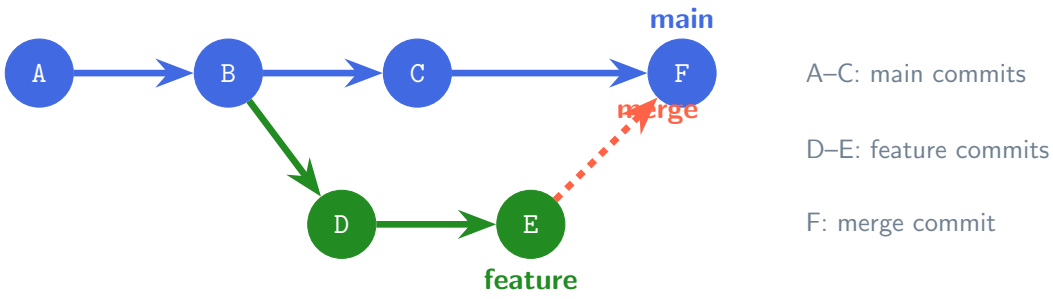
4. ★★ Medium On GitHub, click on your `hello.py` file, then click the pencil icon to edit it directly on the website. Add a comment line and commit the change on GitHub. Back in your terminal, run `git pull`. What happens?

5. ★★★ Hard Why can't you just email your code to a teammate instead of using GitHub? List at least three advantages of using GitHub over emailing files.

6. ★★★ Hard Research: what is a `.gitignore` file? Create one in your repo that tells Git to ignore all files ending in `.pyc`. Commit and push it.

5 Branches

A **branch** is a parallel version of your code. You can create a branch to experiment with a new feature without affecting your main code. If the experiment works, you **merge** it back. If it fails, you just delete the branch—no harm done.



5.1 Branch Commands Reference

Command	What it does
`git branch`	List all branches (current branch has a `*`)
`git checkout -b name`	Create a new branch and switch to it
`git checkout name`	Switch to an existing branch
`git merge name`	Merge the named branch into your current branch
`git branch -d name`	Delete a branch (after merging)

5.2 Walkthrough: Feature Branch

Let's add a feature using a branch (use your `lab02-practice` repo):

```
cd lab02-practice
git checkout -b add-farewell
```

```
Switched to a new branch 'add-farewell'
```

Create a new file `farewell.py`:

```python
# farewell.py
def farewell(name):
    return f"Goodbye, {name}! See you next time."

print(farewell("World"))
```

Stage and commit:

```
git add farewell.py
git commit -m "Add farewell function"
```

Now switch back to main:

```
git checkout main
```

Check your folder—`farewell.py` is *gone*! It only exists on the `add-farewell` branch. Now merge:

```
git merge add-farewell
```

```
Updating abc1234..def5678
Fast-forward
 farewell.py | 5 +++++
 1 file changed, 5 insertions(+)
 create mode 100644 farewell.py
```

Now `farewell.py` is on main! Finally, push to GitHub:

```
git push
```

Never experiment directly on main. Create a branch, do your work, then merge it back when it's ready.

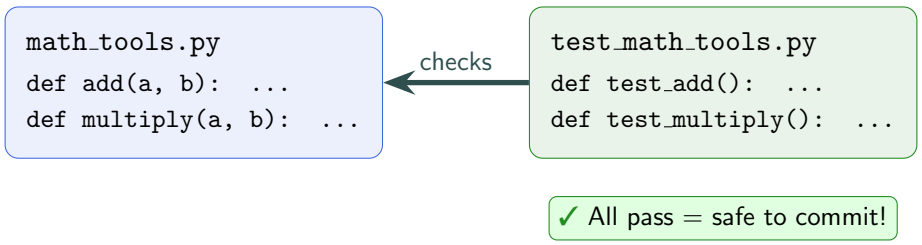
5.3 Exercises: Branches

1. ★ Easy Run `git branch` to see all branches. Which one has the * next to it?
2. ★ Easy Create a new branch called `add-joke`. Create a file `joke.py` that prints your favorite joke. Commit it on the branch.
3. ★★ Medium Switch back to `main`. Verify that `joke.py` is not visible. Then merge `add-joke` into `main` and verify the file appears.
4. ★★ Medium After merging, delete the `add-joke` branch with `git branch -d add-joke`. Run `git branch` to confirm it's gone. Is the `joke.py` file still on main?
5. ★★ Medium Push your updated main branch to GitHub. On the GitHub website, can you see all the commits from both branches in the history?
6. ★★★ Hard Create two branches from main: `feature-a` and `feature-b`. On `feature-a`, create `a.py`. On `feature-b`, create `b.py`. Merge both back into main (one at a time). Run `git log --oneline` and observe the history.

7. ★★★ Hard Research: what is a **merge conflict**? When might one happen? (Hint: think about what happens when two branches edit the *same line* of the *same file*.)

6 Unit Testing with Assert

Now we switch gears from version control to **testing**. A **unit test** is code that checks whether your code works correctly. As you learned in the lectures, tests are your safety net—they catch bugs before they cause real damage.



6.1 The `assert` Statement

Python's `assert` statement checks whether a condition is true. If it is, nothing happens. If it's false, Python raises an `AssertionError`:

```
assert 2 + 2 == 4        # Passes silently
assert 2 + 2 == 5        # AssertionError!
```

We use `assert` to write test functions:

```
def test_add():
    assert add(2, 3) == 5
    assert add(-1, 1) == 0
    assert add(0, 0) == 0
```

The Testing Pattern

Every test follows the same pattern:

1. Call the function you want to test.
2. Compare the result to the **expected** answer using `assert`.
3. If they match, the test passes. If not, you found a bug!

6.2 Walkthrough: Writing and Running Tests

Create a file called `math_tools.py` in your `lab02-practice` folder:

```
# math_tools.py

def add(a, b):
    return a + b

```

```
def multiply(a, b):
    return a * b

def is_even(n):
    return n % 2 == 0
```

Now create `test_math_tools.py` in the same folder:

```
# test_math_tools.py
from math_tools import add, multiply, is_even

def test_add():
    assert add(2, 3) == 5
    assert add(-1, 1) == 0
    assert add(0, 0) == 0
    print("test_add: ALL PASSED")

def test_multiply():
    assert multiply(3, 4) == 12
    assert multiply(-2, 5) == -10
    assert multiply(0, 100) == 0
    print("test_multiply: ALL PASSED")

def test_is_even():
    assert is_even(4) == True
    assert is_even(7) == False
    assert is_even(0) == True
    print("test_is_even: ALL PASSED")

# Run all tests
test_add()
test_multiply()
test_is_even()
print("--- All tests passed! ---")
```

Run the tests from the terminal:

```
python test_math_tools.py
```

```
test_add: ALL PASSED
test_multiply: ALL PASSED
test_is_even: ALL PASSED
--- All tests passed! ---
```

Now let's **commit** this using Git:

```
git add math_tools.py test_math_tools.py
git commit -m "Add math tools with unit tests"
git push
```

Test Before You Commit!

The professional workflow is: **write code → write tests → run tests → commit**. Only commit when all tests pass. This way, every commit in your history represents working code.

6.3 Thinking About Edge Cases

Good tests don't just test the "happy path." They also test **edge cases**—unusual or extreme inputs that might trip up your function:

Edge Case Type	Examples
Zero / empty	`0`, `""`, `[]`
Negative numbers	`-1`, `-100`
Very large values	`999999999`
Single element	A list with one item
Boundary values	First/last valid input, `range` endpoints
Type edge cases	Uppercase vs lowercase strings

6.4 Exercises: Unit Testing

1. ★ Easy Add a function `subtract(a, b)` to `math_tools.py` that returns `a - b`. Write a `test_subtract()` function with at least 3 assert statements. Run your tests.

2. ★ Easy Now introduce a *deliberate* bug into `subtract`: change `return a - b` to `return a + b`. Run your tests again. Which assert fails? Fix the bug and verify all tests pass.

3. ★★ Medium Write a function `max_of_three(a, b, c)` that returns the largest of three numbers. Write `test_max_of_three()` with at least 5 assert statements. Include a case where two numbers are equal and a case where all three are equal.

4. ★★ Medium Write a function `is_palindrome(s)` that returns `True` if the string `s` reads the same forwards and backwards (ignoring spaces, case-sensitive). Write at least 6 test cases including edge cases: empty string, single character, and strings with spaces.

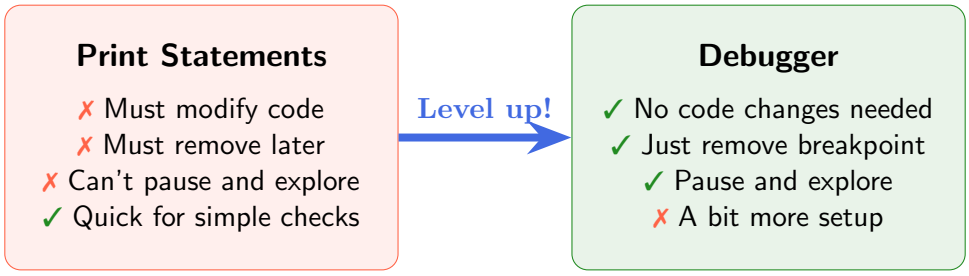
5. ★★★ Hard Write a function `find_min(numbers)` that returns the smallest number in a list. Be careful with your initial value! Write tests that would *catch* the common bug of starting `min_val = 0` (hint: test with a list of all positive numbers).

6. ★★★ Hard Write a function `remove_duplicates(items)` that returns a new list with duplicates removed, preserving the original order. Write at least 6 tests including: empty list, no duplicates, all duplicates, mixed types (e.g., `[1, "1"]`).

7. ★★★ Hard After completing Exercises 1–6, stage all your changes and commit with the message `"Add more functions and comprehensive tests"`. Push to GitHub. You've just followed the professional workflow!

7 Debugging with Thonny

When your tests fail, you need to find the bug. While `print()` statements work in a pinch, the **debugger** is a far more powerful tool. It lets you pause your program at any line, inspect every variable, and step through code one line at a time.



7.1 Thonny Debugger Cheat Sheet

Action	Shortcut	What it does
Start Debug	`Ctrl+F5`	Begin debug mode
Step Over	`F6`	Execute current line, move to next
Step Into	`F7`	Enter a function call to see inside it
Resume	`F8`	Run until end or next breakpoint
Stop	`Ctrl+F2`	End debug session

7.2 Walkthrough: Finding a Bug with the Debugger

Open Thonny and create a file called `buggy.py`:

```
# buggy.py
def count_vowels(text):
    vowels = "aeiou"
    count = 0
    for char in text:
        if char in vowels:
            count += 1
    return count

result = count_vowels("HELLO")
print(f"Vowel count: {result}")
```

Run it normally first:

```
python buggy.py
```

```
Vowel count: 0
```

That's wrong! "HELLO" has 2 vowels (E and O). Let's debug:

Step 1. In Thonny, press **Ctrl+F5** to start the debugger.

Step 2. Press **F7** (Step Into) repeatedly to enter the `count_vowels` function.

Step 3. Watch the **Variables panel** on the right. When `char` is `"H"`, the condition `char in vowels` is `False`.

Step 4. Continue stepping. When `char` is `"E"`, the condition is *still* `False`!

Step 5. Aha! The variable `vowels` contains only *lowercase* letters: `"aeiou"`. But `"E"` is uppercase.

The fix: convert the text to lowercase before checking:

```
def count_vowels(text):
    vowels = "aeiou"
    count = 0
    for char in text.lower():    # Fix: convert to lowercase
        if char in vowels:
            count += 1
    return count
```

```
Vowel count: 2
```

7.3 Breakpoints

A **breakpoint** tells the debugger: "pause here." Instead of stepping through every single line from the beginning, you can jump straight to the code you're suspicious of.

In Thonny, click in the **gray margin** (gutter) next to a line number. A red dot appears—that's your breakpoint. When you run in debug mode, the program will run at full speed until it hits that line, then pause.

When to Use Breakpoints

Set a breakpoint right **before** the line you think might be wrong. When the program pauses, check the Variables panel—are the values what you expected?

7.4 Exercises: Debugging

1. ★ Easy Open `buggy.py` in Thonny. Set a breakpoint on the `if char in vowels` line. Start debugging (Ctrl+F5). Press F8 (Resume) to jump to the breakpoint. What does the Variables panel show?

2. ★ Easy Step through the loop with F6 a few times. Watch how `char` changes with each iteration. Write down the value of `char` and `count` for the first three iterations.

3. ★★ Medium Create a file `bug_hunt1.py` with this code:

```
def get_last_three(items):
    start = len(items) - 3
    end = len(items) - 1
    return items[start:end]

```

```
print(get_last_three([10, 20, 30, 40, 50]))
# Expected: [30, 40, 50]
# Actual:   [30, 40]
```

Use the debugger to find and fix the bug. (Hint: check the values of `start` and `end`.)

4. ★★ Medium Create `bug_hunt2.py`:

```
def make_uppercase(text):
    text.upper()
    return text

print(make_uppercase("hello"))
# Expected: "HELLO"
# Actual: "hello"
```

Use the debugger. What does the Variables panel show right after `text.upper()` runs? What went wrong?

5. ★★★ Hard Create `bug_hunt3.py`:

```
def remove_negatives(numbers):
    for num in numbers:
        if num < 0:
            numbers.remove(num)
    return numbers

print(remove_negatives([1, -2, -3, 4, -5]))
# Expected: [1, 4]
# Actual: [1, -3, 4]
```

Use the debugger to step through the loop. Why is `-3` being skipped? Fix the function. (Hint: build a new list instead of modifying the old one.)

6. ★★★ Hard Create `bug_hunt4.py`:

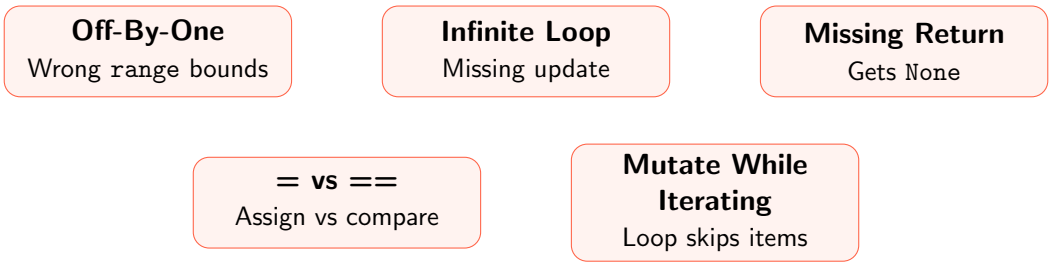
```
def sum_evens(numbers):
    total = 0
    for i in range(len(numbers)):
        if i % 2 == 0:
            total += numbers[i]
    return total

print(sum_evens([1, 2, 3, 4, 5, 6]))
# Expected: 12 (2 + 4 + 6)
# Actual: 9
```

Use the debugger. What is the function actually checking—the *index* or the *value*? Fix it so it sums even *values*, not values at even *indices*.

8 Common Bug Patterns

Experienced developers don't just randomly poke at code—they **recognize patterns**. Here are the most common bugs beginners encounter. Learning to spot them will save you hours.



8.1 Bug Pattern Reference

Symptom	Likely Bug	Fix
Wrong number of items	Off-by-one error	Check `range()` start/end; remember slices exclude the end
Program never stops	Infinite loop	Make sure the loop variable is updated
Function returns `None`	Missing `return`	Add a `return` statement
`SyntaxError` on `if` line	`=` instead of `==`	Use `==` for comparison
Loop skips elements	Mutating while iterating	Build a new list instead
String doesn't change	Strings are immutable	Assign the result: `s = s.upper()`

8.2 The Scientific Method for Debugging

When you encounter a bug, don't randomly change code! Follow this systematic process:

1. **Reproduce:** Make the bug happen again. Write a test case that triggers it.
2. **Isolate:** Narrow down *where* the bug is. Use the debugger!
3. **Hypothesize:** Form a theory about what's wrong.
4. **Test:** Make *one* small change to test your theory.
5. **Verify:** Run all your tests—did the fix work? Did it break anything else?

Tests catch bugs. The debugger finds them. Git lets you undo them. Together, they form your professional safety net.

8.3 Exercises: Bug Patterns

1. ★ Easy Look at this code and identify the bug *without* running it:

```python
def square(x):
    result = x * x
    # return result
```

What will `print(square(5))` output? What kind of bug is this?

2. ★ Easy Look at this code and identify the bug:

```python
def countdown(n):
    while n > 0:
        print(n)
    print("Blastoff!")
```

What will happen when you call `countdown(3)`? What kind of bug is this?

3. ★★ Medium Write a test function `test_first_n()` for this buggy function. Make sure at least one of your tests catches the bug:

```python
def first_n(items, n):
    result = []
    for i in range(1, n):
        result.append(items[i])
    return result
```

4. ★★ Medium This function is supposed to calculate the total price after tax. Find the bug and fix it:

```python
def total_with_tax(price, tax_rate):
    tax = price * tax_rate
    total = price + tax
    # Meant to return total
    price + tax

print(total_with_tax(100, 0.15))
# Expected: 115.0
# Actual: None
```

5. ★★★ Hard This function should reverse a list in place. It has a subtle bug—write tests to catch it, then use the debugger to find it, then fix it:

```python
def reverse_list(items):
    for i in range(len(items)):
        j = len(items) - 1 - i
        items[i], items[j] = items[j], items[i]
    return items
```

6. ★★★ Hard Apply the Scientific Method for Debugging to Exercise 5: describe each of the five steps (Reproduce, Isolate, Hypothesize, Test, Verify) as you find and fix the bug. Write your answers as comments in your code.

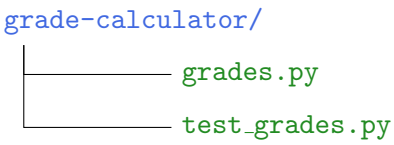
9 Putting It All Together: The Grade Calculator

Capstone Challenge

This section combines **everything** you've learned: Git, testing, and debugging. Take it one step at a time. If you get stuck, re-read the relevant section above.

You will build a small **Grade Calculator** project that converts numeric scores to letter grades. You will write the code, write tests, use Git to track your progress, and debug a deliberate bug.

Project structure:



Follow these steps:

Step 1. Create the repo. Create a new repo on GitHub called `grade-calculator` (with a README). Clone it to your computer and `cd` into it.

```
git clone https://github.com/YourUsername/grade-calculator.
    git
cd grade-calculator
```

Step 2. Create a branch. Never work directly on main for a feature:

```
git checkout -b add-grade-logic
```

Step 3. Write the code. Create `grades.py` and type the following:

```python
# grades.py

def letter_grade(score):
    """Convert a numeric score (0-100) to a letter grade.
       """
    if score >= 90:
        return "A"
    elif score >= 80:
        return "B"
    elif score >= 70:
        return "C"
    elif score >= 60:
        return "D"
    else:
        return "F"

def class_average(scores):
```

```
    """Return the average of a list of scores."""
    total = 0
    for score in scores:
        total = score           # <-- BUG! (deliberate)
    return total / len(scores)

def highest_score(scores):
    """Return the highest score in the list."""
    best = 0                    # <-- BUG! (deliberate)
    for score in scores:
        if score > best:
            best = score
    return best
```

Step 4. Commit the initial code:

```
git add grades.py
git commit -m "Add grade functions (with known bugs to fix)"
```

Step 5. Write the tests. Create `test_grades.py`:

```
# test_grades.py
from grades import letter_grade, class_average,
   highest_score

def test_letter_grade():
    assert letter_grade(95) == "A"
    assert letter_grade(90) == "A"
    assert letter_grade(85) == "B"
    assert letter_grade(80) == "B"
    assert letter_grade(75) == "C"
    assert letter_grade(65) == "D"
    assert letter_grade(50) == "F"
    assert letter_grade(0)  == "F"
    print("test_letter_grade: ALL PASSED")

def test_class_average():
    assert class_average([100, 80, 60]) == 80.0
    assert class_average([90, 90, 90]) == 90.0
    assert class_average([50]) == 50.0
    print("test_class_average: ALL PASSED")

def test_highest_score():
    assert highest_score([70, 85, 90, 60]) == 90
    assert highest_score([50]) == 50
    assert highest_score([-10, -20, -5]) == -5
    print("test_highest_score: ALL PASSED")

# Run all tests
test_letter_grade()
test_class_average()
test_highest_score()
```

```
31 print("--- All tests passed! ---")
```

Step 6. Run the tests:

```
python test_grades.py
```

You should see `test_letter_grade` pass, but the other tests will **fail** with an `AssertionError`. Good—the tests caught the bugs!

Step 7. Debug and fix. Open `grades.py` in Thonny. Use the debugger to step through `class_average` and `highest_score`. Find and fix both bugs:

- In `class_average`: change `total = score` to `total += score`.
- In `highest_score`: change `best = 0` to `best = scores[0]`.

Step 8. Run the tests again:

```
python test_grades.py
```

```
test_letter_grade: ALL PASSED
test_class_average: ALL PASSED
test_highest_score: ALL PASSED
--- All tests passed! ---
```

Step 9. Commit, merge, and push:

```
git add grades.py test_grades.py
git commit -m "Fix bugs in class_average and highest_score"
git checkout main
git merge add-grade-logic
git push
```

The Full Professional Cycle

You just completed the professional development cycle: create a branch → write code → write tests → find and fix bugs with the debugger → commit working code → merge to main → push to GitHub.

Congratulations!

Skills Unlocked!

Version Control with Git

- ✓ Install and configure Git
- ✓ Create repositories (`git init`)
- ✓ Stage and commit (`add`, `commit`)
- ✓ View history (`log`, `diff`)
- ✓ Undo mistakes (`restore`, `revert`)
- ✓ Work with GitHub (`clone`, `push`)
- ✓ Create and merge branches

Testing & Debugging

- ✓ Write unit tests with `assert`
- ✓ Test edge cases
- ✓ Use Thonny's debugger
- ✓ Set and use breakpoints
- ✓ Recognize common bug patterns
- ✓ Debug systematically
- ✓ Follow the professional workflow

Coming Up Next: **Recursion Foundations** — functions that call themselves to solve problems!