Programming 2: **Testing & Debugging** (Lecture 4)

Comp 111

Forman Christian University

The $460 Million Bug

Knight Capital Group, 2012

- A software bug caused the company to lose **$460 million** in just **45 minutes**
- An old piece of code was accidentally reactivated during deployment

Knight Capital Group, 2012

- A software bug caused the company to lose **$460 million** in just **45 minutes**
- An old piece of code was accidentally reactivated during deployment
- **No proper testing** caught the issue before it went live
- The company nearly went **bankrupt overnight**

Knight Capital Group, 2012

- A software bug caused the company to lose **$460 million** in just **45 minutes**
- An old piece of code was accidentally reactivated during deployment
- **No proper testing** caught the issue before it went live
- The company nearly went **bankrupt overnight**

"In the real world, untested code costs money, jobs, and sometimes lives."

Quick Poll

- How many of you have spent more than an hour hunting for a bug that turned out to be something **tiny**?

Quick Poll

- How many of you have spent more than an hour hunting for a bug that turned out to be something **tiny**?
- How do you currently find bugs in your code?
  - Print statements?
  - Staring at code?
  - Asking friends?

Today's Promise

By the end of today, you'll have two **professional superpowers**:

1. **Tests** that catch bugs *before* they embarrass you

Today's Promise

By the end of today, you'll have two **professional superpowers**:

1. **Tests** that catch bugs *before* they embarrass you
2. **Debugger skills** that find bugs in minutes, not hours

Why Testing Matters

The “I Fixed It!” Disaster

You write a helper function that works great:

```
1 def half(n):
2     return n / 2
```

The "I Fixed It!" Disaster

You write a helper function that works great:

```
1 def half(n):
2     return n / 2
```

Your friend uses it in their code:

```
1 def average(a, b):
2     total = a + b
3     return half(total)
4
5 print(average(3, 7))  # 5.0 ✓
```

Later, You Use It Too...

```
1 def split_evenly(items):
2     mid = half(len(items))
3     return items[:mid]
4
5 print(split_evenly([1,2,3,4]))
6 # TypeError: indices must be integers!
```

Later, You Use It Too...

```
1 def split_evenly(items):
2     mid = half(len(items))
3     return items[:mid]
4
5 print(split_evenly([1,2,3,4]))
6 # TypeError: indices must be integers!
```

So you "fix" your function:

```
1 def half(n):
2     return n // 2  # integer division
```

Later, You Use It Too...

```
1 def split_evenly(items):
2     mid = half(len(items))
3     return items[:mid]
4
5 print(split_evenly([1,2,3,4]))
6 # TypeError: indices must be integers!
```

So you "fix" your function:

```
1 def half(n):
2     return n // 2  # integer division
```

Your code works now! But wait...

You Broke Your Friend's Code!

```
print(average(3, 7))  # Now returns 5, not 5.0
print(average(3, 4))  # Returns 3, should be 3.5!
```

You Broke Your Friend's Code!

```
print(average(3, 7))  # Now returns 5, not 5.0
print(average(3, 4))  # Returns 3, should be 3.5!
```

You fixed YOUR bug, but broke your friend's code.

They won't notice until their program gives wrong answers!

What Are Unit Tests?

Unit tests are code that checks your code:

- Separate file (usually `test_*.py`)

```
helpers.py
    def half(n):  ...
    def average(a, b):  ...
```

```
test_helpers.py
    def test_half():  ...
    def test_average():  ...
```

What Are Unit Tests?

Unit tests are code that checks your code:

- Separate file (usually `test_*.py`)
- Each function tests one piece

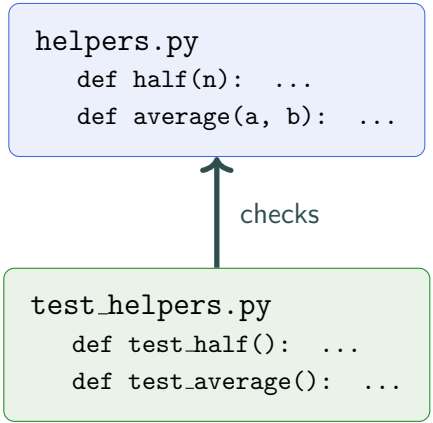
```
helpers.py
   def half(n):  ...
   def average(a, b):  ...
```

```
test_helpers.py
   def test_half():  ...
   def test_average():  ...
```

What Are Unit Tests?

Unit tests are code that checks your code:

- Separate file (usually `test_*.py`)
- Each function tests one piece
- **Run after every change**

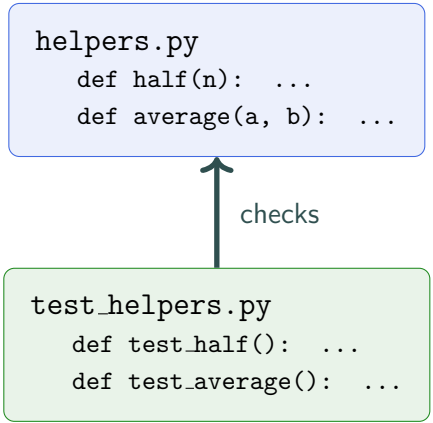


What Are Unit Tests?

Unit tests are code that checks your code:

- Separate file (usually `test_*.py`)
- Each function tests one piece
- **Run after every change**
- **Pass** = working
  Fail = broken

✓ **All pass = safe!**



Unit Tests: Your Safety Net

A **unit test** checks that a function works correctly:

```
1  def test_half():
2      assert half(10) == 5
3      assert half(7) == 3.5  # FAILS after the "fix"!
4
5  def test_average():
6      assert average(3, 7) == 5.0
7
8  def test_split():
9      assert split_evenly([1,2,3,4]) == [1,2]
```

Run Tests After Every Change

```
test_half()          ✗ FAIL ← oops!

test_average()       ✗ FAIL ← Uh oh!

test_split()         ✓ PASS
```

Run Tests After Every Change

```
test_half()          ✗ FAIL ← oops!

test_average()       ✗ FAIL ← Uh oh!

test_split()         ✓ PASS
```

The Golden Rule:

"If all tests pass before AND after your change, you haven't broken anything."

You Try

Write test cases for this function:

```
def is_palindrome(s):
    """ Returns True if s reads the same
        forwards and backwards (case-sensitive),
        False otherwise.
        Ignores spaces. """
```

Think about:

- Normal palindromes: `"racecar"`, `"madam"`
- Non-palindromes: `"hello"`, `"python"`
- Edge cases: `""`, `"a"`, `"ab"`, `"A man a plan a canal Panama"`

Write at least 4 test cases using `assert` statements!

Solution

```
def test_is_palindrome():
    # Basic palindromes
    assert is_palindrome("racecar") == True
    assert is_palindrome("madam") == True

    # Non-palindromes
    assert is_palindrome("hello") == False

    # Edge cases
    assert is_palindrome("") == True
    assert is_palindrome("a") == True
    assert is_palindrome("ab") == False

    # With spaces
    assert is_palindrome("a man a plan a canal panama") == True
```

Solution

```
1  def test_is_palindrome():
2      # Basic palindromes
3      assert is_palindrome("racecar") == True
4      assert is_palindrome("madam") == True
5
6      # Non-palindromes
7      assert is_palindrome("hello") == False
8
9      # Edge cases
10     assert is_palindrome("") == True
11     assert is_palindrome("a") == True
12     assert is_palindrome("ab") == False
13
14     # With spaces
15     assert is_palindrome("a man a plan a canal panama") == True
```

Did you think of the empty string case?

You Try: Test This Buggy Function

This function has a **subtle bug**. Write tests to catch it!

```
1 def find_min(numbers):
2     """Return the smallest number in list"""
3     min_val = 0
4     for num in numbers:
5         if num < min_val:
6             min_val = num
7     return min_val
```

Hint: Think about what happens when...

- All numbers are positive?
- All numbers are negative?
- The list has one element?

Solution: The Bug Revealed

```
1  def test_find_min():
2      # These pass (numbers below 0 exist)
3      assert find_min([-3, -1, -7]) == -7     ✓
4      assert find_min([2, -5, 3])   == -5     ✓
5
6      # These FAIL! (bug: min_val starts at 0)
7      assert find_min([5, 3, 8])    == 3      ✗ Returns 0!
8      assert find_min([42])         == 42     ✗ Returns 0!
```

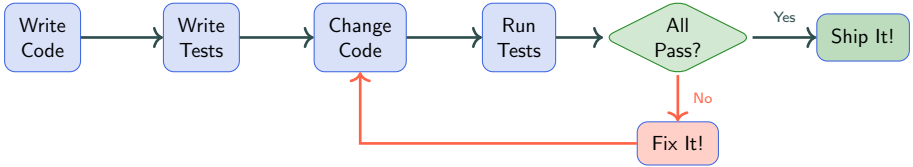
Solution: The Bug Revealed

```
1  def test_find_min():
2      # These pass (numbers below 0 exist)
3      assert find_min([-3, -1, -7]) == -7     ✓
4      assert find_min([2, -5, 3])   == -5     ✓
5
6      # These FAIL! (bug: min_val starts at 0)
7      assert find_min([5, 3, 8])    == 3      ✗ Returns 0!
8      assert find_min([42])         == 42     ✗ Returns 0!
```

Fix: `min_val = numbers[0]` instead of `min_val = 0`

Lesson: Good tests think about **initial values** and **edge cases**!

How Professionals Work



- Professional companies require tests for all code
- Tests run automatically on every change

Big Idea

Tests are like having a **robot friend**

who checks your work **every single time**

Debugging using Print Statements

Why Print Statements Aren't Enough

```
1  def find_max(nums):
2      print("nums:", nums)
3      result = nums[0]
4      print("start:", result)
5      for n in nums:
6          print("checking:", n)
7          if n > result:
8              result = n
9              print("new max:", result)
10     print("final:", result)
11     return result
```

Why Print Statements Aren't Enough

```
1  def find_max(nums):
2      print("nums:", nums)
3      result = nums[0]
4      print("start:", result)
5      for n in nums:
6          print("checking:", n)
7          if n > result:
8              result = n
9              print("new max:", result)
10     print("final:", result)
11     return result
```

5 lines of clutter!

Why Print Statements Aren't Enough

```
1  def find_max(nums):
2      print("nums:", nums)
3      result = nums[0]
4      print("start:", result)
5      for n in nums:
6          print("checking:", n)
7          if n > result:
8              result = n
9              print("new max:", result)
10     print("final:", result)
11     return result
```

5 lines of clutter!

Only 4 lines of logic!

Why Print Statements Aren't Enough

```
1  def find_max(nums):
2      print("nums:", nums)
3      result = nums[0]
4      print("start:", result)
5      for n in nums:
6          print("checking:", n)
7          if n > result:
8              result = n
9              print("new max:", result)
10     print("
11     return
```

5 lines of clutter!

More debug code than actual code!

And you still have to remove it all later...

Problems with Print Debugging

- ✗ Clutters your code
- ✗ Must remove all prints later
- ✗ Can't see things you forgot to print
- ✗ Can't pause and explore

There has to be a better way...

Introducing the Debugger

A debugger lets you:

- **Pause** your program at any line
- **See** every variable's value
- **Step** through code one line at a time
- **Watch** how values change

Introducing the Debugger

A debugger lets you:

- **Pause** your program at any line
- **See** every variable's value
- **Step** through code one line at a time
- **Watch** how values change

"It's like having a **pause button** *and* **X-ray vision** *for your code."*

Demo: A Buggy Function

```
1  def count_vowels(text):
2      vowels = "aeiou"
3      count = 0
4      for char in text:
5          if char in vowels:
6              count += 1
7      return count
8
9  result = count_vowels("HELLO")
10 print(result)  # Returns 0???
```

Demo: A Buggy Function

```
1  def count_vowels(text):
2      vowels = "aeiou"
3      count = 0
4      for char in text:
5          if char in vowels:
6              count += 1
7      return count
8
9  result = count_vowels("HELLO")
10 print(result)  # Returns 0???
```

Bug!

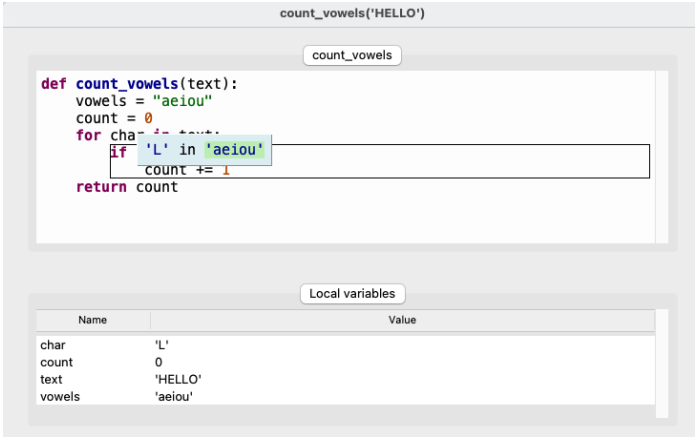
Demo: A Buggy Function

```
1  def count_vowels(text):
2      vowels = "aeiou"
3      count = 0
4      for char in text:
5          if char in vowels:
6              count += 1
7      return count
8
9  result = count_vowels("HELLO")
10 print(result)  # Returns 0???
```

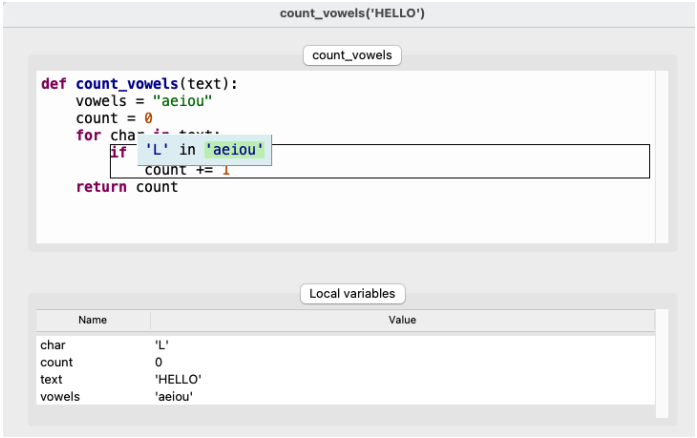
Bug!

Let's use the debugger to find out why...

Thonny Debugger Demo



Thonny Debugger Demo



Discovery: ‘H’ is not in "aeiou" because it’s **uppercase**!

Thonny Debugger Cheat Sheet

Action	Shortcut	What It Does
Start Debugging	`Ctrl+F5`	Begin debug mode
Step Over	`F6`	Execute line, move to next
Step Into	`F7`	Enter a function call
Resume	`F8`	Run until end/error
Step Back	`Ctrl+B`	Go back one step
Stop	`Ctrl+F2`	End debug session

Thonny Debugger Cheat Sheet

Action	Shortcut	What It Does
Start Debugging	`Ctrl+F5`	Begin debug mode
Step Over	`F6`	Execute line, move to next
Step Into	`F7`	Enter a function call
Resume	`F8`	Run until end/error
Step Back	`Ctrl+B`	Go back one step
Stop	`Ctrl+F2`	End debug session

Most important: F6 (Step Over) + watch the Variables panel!

Guided Practice: Find the Bug

```
1 def get_last_three(items):
2     start = len(items) - 3
3     end = len(items) - 1
4     return items[start:end]
5
6 nums = [1, 2, 3, 4, 5]
7 print(get_last_three(nums))
8 # Expected: [3, 4, 5]
9 # Actual:   [3, 4]
```

Guided Practice: Find the Bug

```
1  def get_last_three(items):
2      start = len(items) - 3
3      end = len(items) - 1
4      return items[start:end]
5
6  nums = [1, 2, 3, 4, 5]
7  print(get_last_three(nums))
8  # Expected: [3, 4, 5]
9  # Actual:   [3, 4]
```

Bug!

Debugging Step by Step

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

Debugging Step by Step

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

Variables

```
items = [1,2,3,4,5]
```

Debugging Step by Step

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

Variables

```
items = [1,2,3,4,5]

start = 2
```

Debugging Step by Step

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

Variables

```
items = [1,2,3,4,5]
start = 2
end = 4
```

Debugging Step by Step

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

Variables

```
items = [1,2,3,4,5]
start = 2
end = 4
```

Problem: `items[2:4]` only gets indices 2 and 3!

Fix: `end` should be `len(items)` *(or just omit it!)*

The Fix

```
1 def get_last_three(items):
2     start = len(items) - 3
3     return items[start:]  # Omit end index!
```

The Fix

```
1 def get_last_three(items):
2     start = len(items) - 3
3     return items[start:]  # Omit end index!
```

`items[2:]` → `[3, 4, 5]` ✓

Quick Break

"99 little bugs in the code,
99 little bugs...
Take one down, patch it around...
127 little bugs in the code."

Systematic Debugging

The Scientific Method for Bugs

Don't randomly change things! Use a process:

1. **Reproduce** — Make the bug happen again

The Scientific Method for Bugs

Don't randomly change things! Use a process:

1. **Reproduce** — Make the bug happen again
2. **Isolate** — Where exactly does it fail? (Use debugger!)

The Scientific Method for Bugs

Don't randomly change things! Use a process:

1. **Reproduce** — Make the bug happen again
2. **Isolate** — Where exactly does it fail? (Use debugger!)
3. **Hypothesize** — What do you think is wrong?

The Scientific Method for Bugs

Don't randomly change things! Use a process:

1. **Reproduce** — Make the bug happen again
2. **Isolate** — Where exactly does it fail? (Use debugger!)
3. **Hypothesize** — What do you think is wrong?
4. **Test** — Make ONE small change

The Scientific Method for Bugs

Don't randomly change things! Use a process:

1. **Reproduce** — Make the bug happen again
2. **Isolate** — Where exactly does it fail? (Use debugger!)
3. **Hypothesize** — What do you think is wrong?
4. **Test** — Make ONE small change
5. **Verify** — Does it work now? Did you break anything else?

Common Bug: Off-By-One

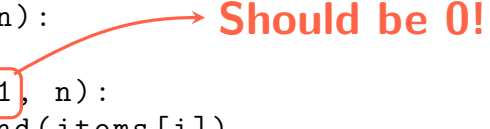
One of the most frequent bugs ever:

```
1  def first_n(items, n):
2      result = []
3      for i in range(1, n):
4          result.append(items[i])
5      return result
6
7  print(first_n([10,20,30,40], 3))
8  # Expected: [10, 20, 30]
9  # Actual:   [20, 30]
```

Common Bug: Off-By-One

One of the most frequent bugs ever:

```
1  def first_n(items, n):
2      result = []
3      for i in range(1, n):
4          result.append(items[i])
5      return result
6
7  print(first_n([10,20,30,40], 3))
8  # Expected: [10, 20, 30]
9  # Actual:   [20, 30]
```



Debugging Off-By-One

Step through the loop:

Debugging Off-By-One

Step through the loop:

```
i=1:  append(items[1])
→ append(20)
```

Debugging Off-By-One

Step through the loop:

`i=1:` `append(items[1])`
→ append(20)

`i=2:` `append(items[2])`
→ append(30)

Debugging Off-By-One

Step through the loop:

`i=1:` `append(items[1])`
→ append(20)

`i=2:` `append(items[2])`
→ append(30)

`i=3:` `loop ends`

Debugging Off-By-One

Step through the loop:

`i=1:` `append(items[1])`
→ append(20)

`i=2:` `append(items[2])`
→ append(30)

`i=3:` `loop ends`

Problem:

`i` starts at 1, not 0!

Fix:

`range(0, n)`
or just `range(n)`

Common Bug Patterns

The Bug Zoo

Every beginner makes these mistakes. Let's learn to **recognize** them:

1. **Infinite Loop** — program never stops

The Bug Zoo

Every beginner makes these mistakes. Let's learn to **recognize** them:

1. **Infinite Loop** — program never stops
2. **Forgetting to Return** — function gives `None`

The Bug Zoo

Every beginner makes these mistakes. Let's learn to **recognize** them:

1. **Infinite Loop** — program never stops
2. **Forgetting to Return** — function gives `None`
3. **Mutating While Iterating** — loop skips items

The Bug Zoo

Every beginner makes these mistakes. Let's learn to **recognize** them:

1. **Infinite Loop** — program never stops
2. **Forgetting to Return** — function gives `None`
3. **Mutating While Iterating** — loop skips items
4. **= vs ==** — assignment vs comparison

The Bug Zoo

Every beginner makes these mistakes. Let's learn to **recognize** them:

1. **Infinite Loop** — program never stops
2. **Forgetting to Return** — function gives `None`
3. **Mutating While Iterating** — loop skips items
4. **= vs ==** — assignment vs comparison

Once you know these patterns, you'll spot them in seconds!

Bug #1: Infinite Loop

```
1 def countdown(n):
2     i = n
3     while i > 0:
4         print(i)
5         # oops... forgot i -= 1
6     print("Blastoff!")
7
8 countdown(3)  # Prints 3 forever!
```

Bug #1: Infinite Loop

```
1  def countdown(n):
2      i = n
3      while i > 0:
4          print(i)
5          # oops... forgot i -= 1
6      print("Blastoff!")
7
8  countdown(3)  # Prints 3 forever!
```

Missing!

Bug #1: Infinite Loop

```
1  def countdown(n):
2      i = n
3      while i > 0:
4          print(i)
5          # oops... forgot i -= 1
6      print("Blastoff!")
7
8  countdown(3)   # Prints 3 forever!
```

Missing!

Debugger tip: Step through the loop — watch `i` **never change**

Bug #2: Forgetting to Return

```
1 def square(x):
2     result = x * x
3     # forgot return!
4
5 answer = square(5)
6 print(answer)        # None ???
7 print(answer * 2)    # TypeError!
```

Bug #2: Forgetting to Return

```
1 def square(x):
2     result = x * x
3     # forgot return!
4
5 answer = square(5)
6 print(answer)        # None ???
7 print(answer * 2)    # TypeError!
```

Should be: `return result`

Bug #2: Forgetting to Return

```
1  def square(x):
2      result = x * x
3      # forgot return!
4
5  answer = square(5)
6  print(answer)          # None ???
7  print(answer * 2)      # TypeError!
```

Should be: `return result`

Debugger tip:

Variables panel shows `result = 25`

but the function returns `None` because there's no `return` statement!

Bug #3: Mutating While Iterating

```
1  def remove_negatives(numbers):
2      for num in numbers:
3          if num < 0:
4              numbers.remove(num)  # changes list!
5      return numbers
6
7  print(remove_negatives([1, -2, -3, 4, -5]))
8  # Expected: [1, 4]
9  # Actual:   [1, -3, 4]
```

Bug #3: Mutating While Iterating

```
1  def remove_negatives(numbers):
2      for num in numbers:
3          if num < 0:
4              numbers.remove(num)  # changes list!
5      return numbers
6
7  print(remove_negatives([1, -2, -3, 4, -5]))
8  # Expected: [1, 4]
9  # Actual:   [1, -3, 4]
```

-3 was skipped!

Bug #3: Mutating While Iterating

```
1  def remove_negatives(numbers):
2      for num in numbers:
3          if num < 0:
4              numbers.remove(num)  # changes list!
5      return numbers
6
7  print(remove_negatives([1, -2, -3, 4, -5]))
8  # Expected: [1, 4]
9  # Actual:   [1, -3, 4]
```

-3 was skipped!

Why? Removing an item **shifts all indices** — the loop skips the next element!

Fix: Build a **new list** instead: `[n for n in numbers if n >= 0]`

Bug #4: = vs ==

The obvious one:

```
1 x = 5
2 if x = 10:   # SyntaxError!
3     print("ten")
```

Bug #4: = vs ==

The obvious one:

```
x = 5
if x = 10:   # SyntaxError!
    print("ten")
```

The sneaky one:

```
def total(numbers):
    result = 0
    for num in numbers:
        result = num    # =
            not +=
    return result
```

Bug #4: = vs ==

The obvious one:

```
1 x = 5
2 if x = 10:   # SyntaxError!
3     print("ten")
```

Symbol	Meaning
=	Assign a value
==	Compare values
is	Same object?

The sneaky one:

```
1 def total(numbers):
2     result = 0
3     for num in numbers:
4         result = num    # =
              not +=
5     return result
```

Bug Pattern Cheat Sheet

Symptom	Likely Bug	Fix
Program never stops	Infinite loop	Update loop variable
Function returns `None`	Missing `return`	Add `return` statement
Loop skips elements	Mutating while iterating	Build new list
`SyntaxError` on `if`	= instead of ==	Use == for comparison

Bug Pattern Cheat Sheet

Symptom	Likely Bug	Fix
Program never stops	Infinite loop	Update loop variable
Function returns `None`	Missing `return`	Add `return` statement
Loop skips elements	Mutating while iterating	Build new list
`SyntaxError` on `if`	= instead of ==	Use == for comparison

Know the patterns → Spot bugs faster!

Breakpoints

What is a Breakpoint?

- A **breakpoint** is a marker that tells your program:

  "STOP here and let me look around"

What is a Breakpoint?

- A **breakpoint** is a marker that tells your program:

  "STOP here and let me look around"

- When the program hits a breakpoint, it **pauses**

What is a Breakpoint?

- A **breakpoint** is a marker that tells your program:

  "STOP here and let me look around"

- When the program hits a breakpoint, it **pauses**
- You can then:
  - Inspect variable values
  - Step through code line by line
  - See the call stack

What is a Breakpoint?

- A **breakpoint** is a marker that tells your program:

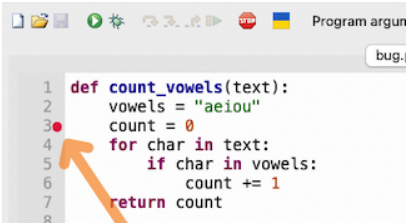
  "STOP here and let me look around"

- When the program hits a breakpoint, it **pauses**
- You can then:
  - Inspect variable values
  - Step through code line by line
  - See the call stack

Like pressing pause on a movie to examine a frame!

Setting a Breakpoint in Thonny

1. Click in the **gutter** (left margin) next to a line number
2. A **red dot** appears
3. Press **green bug** to run in debug mode
4. Program runs until it hits the breakpoint

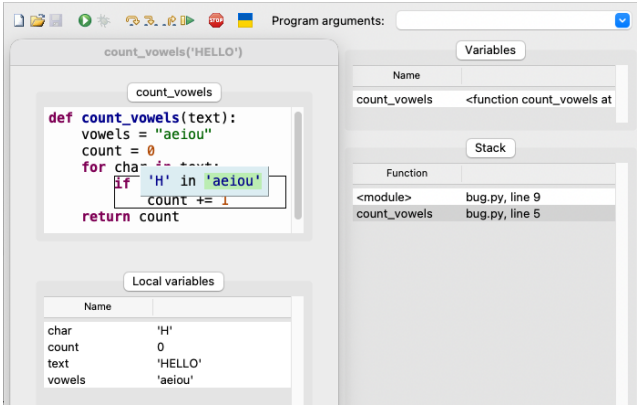


The Debug Panel

When paused, you can see:

- **Variables**
  Local and global values
- **Stack**
  Which functions called which
- **Heap** (optional)
  Visualize objects in memory

View → Variables / Stack / Heap



Live Debugging Example

When to Use the Debugger?

- ✅ Your code runs but gives **wrong output**
- ✅ You don't understand **why** something happens
- ✅ A loop or recursion behaves **unexpectedly**
- ✅ You want to see **variable values** at a specific point

When to Use the Debugger?

- ✅ Your code runs but gives **wrong output**
- ✅ You don't understand **why** something happens
- ✅ A loop or recursion behaves **unexpectedly**
- ✅ You want to see **variable values** at a specific point

Debugger = X-ray vision for your code!

Print vs Debugger

Print Statements	Debugger
Must modify code	No code changes

Print vs Debugger

Print Statements
Must modify code
Must remove later

Debugger
No code changes
Just remove breakpoint

Print vs Debugger

Print Statements
Must modify code
Must remove later
Can't pause execution

Debugger
No code changes
Just remove breakpoint
Pause anytime

Print vs Debugger

Print Statements
Must modify code
Must remove later
Can't pause execution
Quick for simple checks

Debugger
No code changes
Just remove breakpoint
Pause anytime
More setup needed

Print vs Debugger

Print Statements
Must modify code
Must remove later
Can't pause execution
Quick for simple checks

Debugger
No code changes
Just remove breakpoint
Pause anytime
More setup needed

Use both! But learn the debugger — it's powerful.

You Try

This function should return the sum of all even numbers in a list:

```
1  def sum_evens(numbers):
2      total = 0
3      for i in range(len(numbers)):
4          if i % 2 == 0:
5              total += numbers[i]
6      return total
7
8  print(sum_evens([1, 2, 3, 4, 5, 6]))
9  # Expected: 12 (2 + 4 + 6)
10 # Actual:   9
```

Bug!

Use Thonny's debugger to find and fix the bug!

Bug Hunt: Off-By-One

```
1  def count_evens(numbers):
2      count = 0
3      for i in range(1, len(numbers)):
4          if numbers[i] % 2 == 0:
5              count += 1
6      return count
7
8  print(count_evens([2, 5, 8, 3]))
9  # Expected: 2
10 # Actual:   1
```

Bug Hunt: Off-By-One

```
1  def count_evens(numbers):
2      count = 0
3      for i in range(1, len(numbers)):
4          if numbers[i] % 2 == 0:
5              count += 1
6      return count
7
8  print(count_evens([2, 5, 8, 3]))
9  # Expected: 2
10 # Actual:   1
```

Skips index 0!

Bug Hunt: Off-By-One

```
1  def count_evens(numbers):
2      count = 0
3      for i in range(1, len(numbers)):
4          if numbers[i] % 2 == 0:
5              count += 1
6      return count
7
8  print(count_evens([2, 5, 8, 3]))
9  # Expected: 2
10 # Actual:   1
```

Skips index 0!

Fix: range(0, len(numbers)) or range(len(numbers))

Bug Hunt: Accumulator Gone Wrong

```
1  def calculate(numbers):
2      result = 0
3      for num in numbers:
4          result = num
5      return result
6
7  print(calculate([1, 2, 3, 4, 5]))
8  # Expected: 15
9  # Actual:   5
```

Bug Hunt: Accumulator Gone Wrong

```
1  def calculate(numbers):
2      result = 0
3      for num in numbers:
4          result = num
5      return result
6
7  print(calculate([1, 2, 3, 4, 5]))
8  # Expected: 15
9  # Actual:   5
```

Overwrites instead of adding!

Bug Hunt: Accumulator Gone Wrong

```
1  def calculate(numbers):
2      result = 0
3      for num in numbers:
4          result = num
5      return result
6
7  print(calculate([1, 2, 3, 4, 5]))
8  # Expected: 15
9  # Actual:   5
```

Overwrites instead of adding!

Fix: `result += num` (not `result = num`)

Bug Hunt: String Surprise

```
1 def make_uppercase(text):
2     text.upper()
3     return text
4
5 print(make_uppercase("hello"))
6 # Expected: "HELLO"
7 # Actual:   "hello"
```

Bug Hunt: String Surprise

```
1 def make_uppercase(text):
2     text.upper()
3     return text
4
5 print(make_uppercase("hello"))
6 # Expected: "HELLO"
7 # Actual:   "hello"
```

Result thrown away!

Bug Hunt: String Surprise

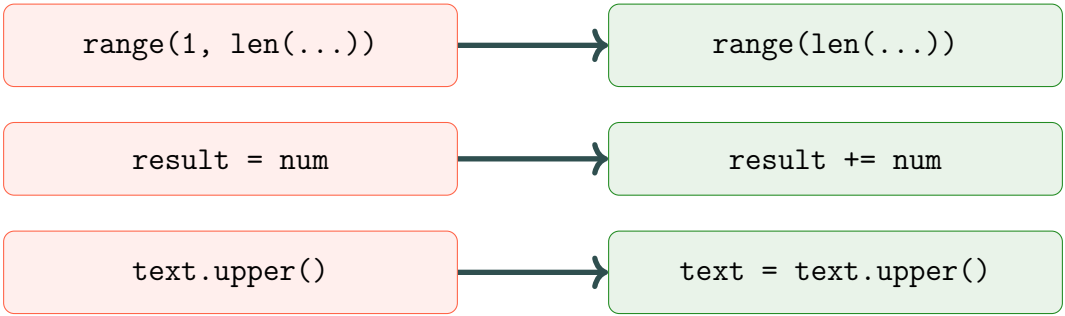
```
1 def make_uppercase(text):
2     text.upper()
3     return text
4
5 print(make_uppercase("hello"))
6 # Expected: "HELLO"
7 # Actual:   "hello"
```

Result thrown away!

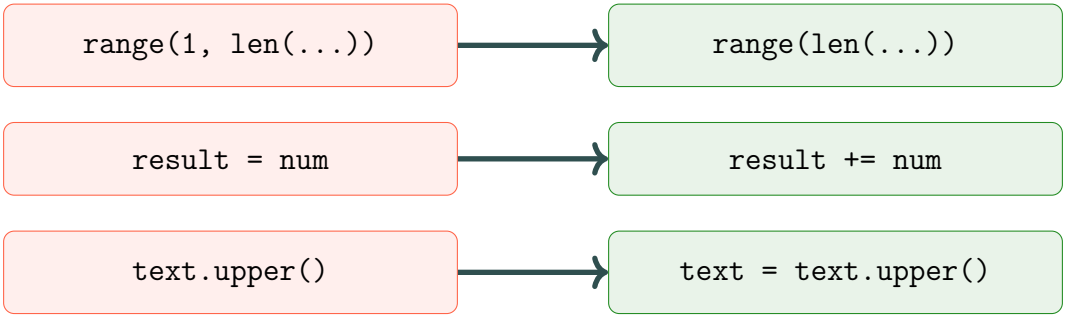
Why? Strings are **immutable** — `.upper()` returns a **new** string!

Fix: `return text.upper()` or `text = text.upper()`

Bug Hunt Recap



Bug Hunt Recap



"The debugger shows you exactly where **expectations diverge from reality**.*"*

Summary

Key Takeaways

1. Tests catch "fixes" that break other code

- Write tests for your functions
- Run all tests after every change

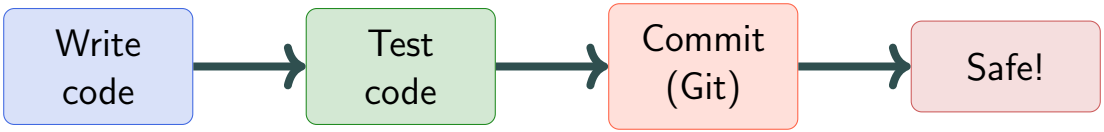
2. The debugger is your X-ray vision

- F6 to step through code
- Watch the Variables panel

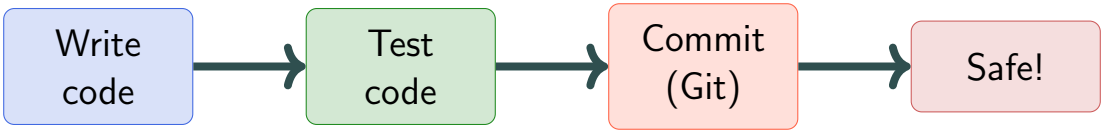
3. Debug systematically

- Reproduce → Isolate → Hypothesize → Test → Verify

Git + Testing = Superpower



Git + Testing = Superpower



If something breaks later, **you can go back!**

(That's what we learned in Lecture 1)

Questions?

Next time: **Recursion Foundations**

Problems that solve themselves!