Testing

CSE 160 Summer 2026

Questions?

Sli.do #cse160

Announcements

- **Written Check-In 7 due tonight at 11:59 PM**
- **Homework Assignment 5 due Monday, August 17 at 11:59PM**
- **Programming Practice 6 due Sunday, August 9 at 11:59PM**
- **Resubmission Cycle 3 due Sunday, August 9th at 11:59PM**
  - Homeworks 1-3 eligible
  - Fill out the Google Form!
- **Final Exam information and Practice Finals** now available on course website!
  - Final Exam on Friday, August 21 at 9:40am in HRC 155

Today's Roadmap

1. How important is testing?
2. Testing programs vs. functions
3. Writing Test Suites
4. Testing Approaches
5. Testing Guidelines
6. Assertions for Debugging

Jupyter Hub

Why should we test our code?

- Programming to analyze data is powerful
- It's useless (or worse!) if the results are not correct
- Correctness is far more important than speed

Famous Examples of Software Failures

- Ariane 5 rocket (1996)
  - fault in the software in the inertial navigation system
- Therac-25 radiation therapy machine (1986/1987)
  - Fatal overdose due to software bugs and no external controls
- List of Software Failures

More Recent Software Failures

- Mars Rovers (1999)
  - Two systems were sharing information in different units
- Airplanes have a lot of software…
  - Airbus (2015) & Boeing (2019)
- WannaCry ransomware attack (2017)
  - Exploited a bug in software to demanding ransom payments
- CrowdStrike (2024)
  - “Routine” update by CrowdStrike crashed Windows devices: $5 Billion cost
- Tesla Autopilot Crashes (2016-present)

TECH | AMAZON

Prolonged AWS outage takes down a big chunk of the internet

AWS has been experiencing an outage for hours

By Jay Peters | @jaypeters | Updated Nov 25, 2020, 5:39pm EST

INSIDER

Log in Subscribe

Tesla's Full Self-Driving tech keeps getting fooled by the moon, billboards, and Burger King signs

Tim Levin Jul 26, 2021, 10:19 AM

Forbes

EDITORS' PICK | Oct 5, 2021, 09:09pm EDT | 3,285 views

Facebook Says A Bug In A Software Audit Tool Triggered Yesterday's Mega Outage

Testing does not prove correctness

"Program testing can be used to show the presence of bugs, but never to show their absence!"

- Edsger Dijkstra

- Testing can only increase our confidence in program correctness.
- Exhaustive testing (e.g. testing all possible inputs) is generally not possible
- Instead we have to be smart about testing

Testing ≠ debugging

Testing: determining whether your program is correct

Doesn't say where or how your program is incorrect

Debugging: locating the specific defect in your program, and fixing it

Today's Roadmap

1. ~~How important is testing?~~
2. Testing programs vs. functions
3. Writing Test Suites
4. Testing Approaches
5. Testing Guidelines
6. Assertions for Debugging

Jupyter Hub

Testing your program

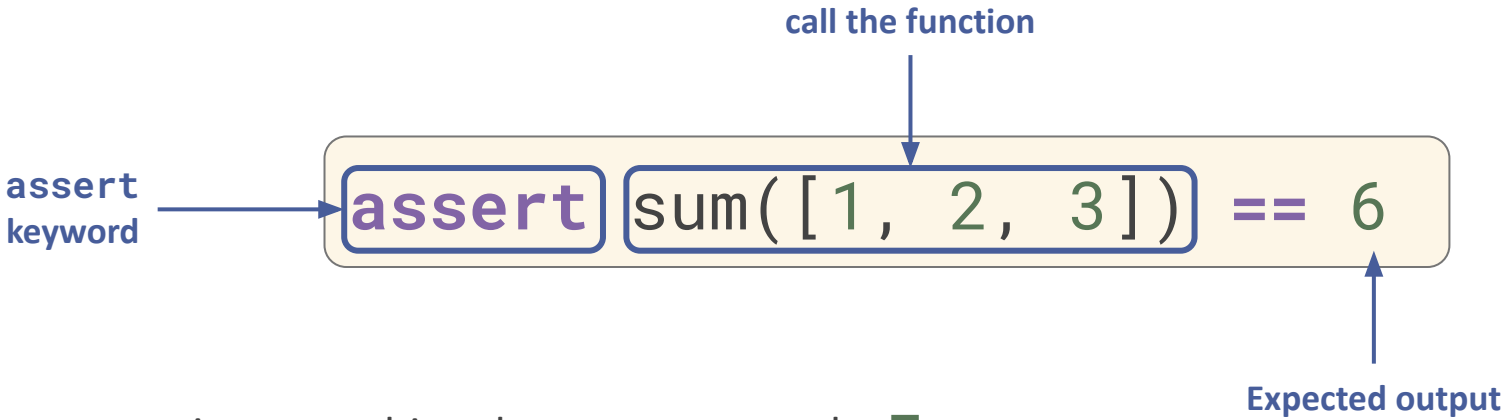
- How do you know your **program** is right?
  - Compare its output to a correct output
- How do you know a correct output?
  - Real data is big
  - You wrote a computer program because it is not convenient to compute it by hand
- Use small inputs so you can compute the expected output by hand
  - We did this in HW2, HW3, HW4 & HW5 with small data sets

Testing parts of your program

- Often called “unit testing”
- Testing that the output of individual functions is correct.
- One way: using `assert` statements
- We have also been doing this!
  - In Homeworks
  - In Programming Practices

Writing a test using an `assert` statement

- An example test for the `sum` function.
- We expect the function to return 6 when we pass in the list [1, 2, 3].



We are asserting something that we expect to be **`True`**

- If the assertion is **True**, then nothing will happen
- If the assertion is **False**, then the program halts and prints a message

Aside: Don't use == for comparing floats!

An example test in restaurants.py:

```
assert avg == 4.06
```

Don't do this! Do not test for equality with floats!

See `float_equality.py` on JupyterHub

Aside: Use `math.isclose()` for comparing floats!

An example test restaurants.py:

```
assert avg == 4.06
```

Don't do this! Do not test for equality with floats!

Instead:

```
assert math.isclose(avg, 4.06)
```

See `float_equality.py` on JupyterHub. Don't forget to `import math`.

How to write a good Test Suite

Test Suite: a collection of test cases used to test a program or function

A good Test Suite includes:

- Good coverage of the **input space**
  - What are the possible inputs to the function?
- Good coverage of **code**
  - What are the possible paths through the code?
  - Note: we do not always know what the code is before writing tests.
- Addresses **boundary cases**
  - What are cases at the edges of the input space

Example (input space coverage)

```
def abs(x):
    """
    Takes in an integer x and returns the absolute value of
    that integer.
    """
  if x > 0:
      return x
  else:
      return -x
```

What are the possible *categories* of values **x** can be?

Example (code coverage)

```
def abs(x):
    """
    Takes in an integer x and returns the absolute value of
    that integer.
    """
  if x > 0:
      return x
  else:
      return -x
```

What are the possible *paths through the code* in this function?

Example (code coverage): `assert abs(5) == 5`

```
def abs(x):
    """
    Takes in an integer x and returns the absolute value of
    that integer.
    """
    if x > 0:
        return x
    else:
        return -x
```

What are the possible *paths through the code* in this function?

Example (code coverage): `assert abs(-2) == 2`

```
def abs(x):
    """
    Takes in an integer x and returns the absolute value of
    that integer.
    """
    if x > 0:
        return x
    else:
        return -x
```

What are the possible *paths through the code* in this function?

Example (code coverage): Test Suite

```
def abs(x):
    """
    Takes in an integer x and returns the absolute value of
    that integer.
    """
   if x > 0:
       return x
   else:
       return -x
```

```
assert abs(5) == 5
assert abs(-2) == 2
```

Example (code coverage): Can still miss bugs

```
def abs(x):
    """
    Takes in an integer x and returns the absolute value of
    that integer.
    """
   if x > 1:
       return x
   else:
       return -x
```

```
assert abs(5) == 5
assert abs(-2) == 2
```

Example (boundary cases): `assert abs(0) == 0`

```
def abs(x):
    """
    Takes in an integer x and returns the absolute value of
    that integer.
    """
    if x > 0:
        return x
    else:
        return -x
```

```
assert abs(5) == 5
assert abs(-2) == 2
```

Another Example (Discuss!)

```
def find_max(lst):
    """
    Takes in a list of integers lst and returns the maximum
    value in the list. If the list is empty, return None.
    """
```

Come up with at least 4 assert statements including input and output that tests `find_max()`

Coming up with good test cases

Numbers:

- int vs. float values (Note: do not to test for equality with floats)
- Zero
- Negative values

Lists:

- Empty list, lists of different lengths
- Lists containing duplicate values (or including all the same value)
- Lists in ascending order/descending order
- Mix of types in list (if the function specification does not rule this out)

Today's Roadmap

1. ~~How important is testing?~~
2. ~~Testing programs vs. functions~~
3. ~~Writing Test Suites~~
4. Testing Approaches
5. Testing Guidelines
6. Assertions for Debugging

Jupyter Hub

Testing Approaches

- **Black box testing:** Choose test data ***without*** looking at the implementation, just test behavior mentioned in the specification (or doc-string)
- **Glass box** (white box, clear box) **testing:** Choose test data ***with*** knowledge of the implementation. Test that all paths through your code are exercised and correct. Examples:
  - If statement with several elifs, make sure your test cases will execute all branches
  - For loop, test if it is executed never, once, >1, max times

Don't Remove Tests!

Regression testing

- Whenever you find a bug (not from an existing test)
  - Add a **new** test case with the input that exposes the bug and the expected output to the test suite
  - Verify that the test suite fails
  - Fix the bug
  - Verify the fix worked
- Do NOT remove tests - protects against reintroducing the same bug later

When to write tests

- Two possibilities:
  - Write code first, then write tests
  - Write tests first, then write code
- It's best to write tests first
  - If you write the code first, you remember the implementation while writing the tests (confirmation bias!)
  - You are likely to make the same mistakes that you made in the implementation (e.g. assuming that negative values would never be present in a list of numbers)
- If you write the tests first, you will think more about the functionality than about a particular implementation
  - You might notice some aspect of behavior that you would have made a mistake about, some special case of input that you would have forgotten to handle

Where to write tests

- At the top level: tests are run every time you load your program

```
def hypotenuse(a, b):
  … body of hypotenuse …

assert hypotenuse(3, 4) == 5
assert hypotenuse(5, 12) == 13
```

As in HW2, programming practices

- In a test function: tests are run when you invoke the test function

```
def hypotenuse(a, b):
  … body of hypotenuse …

def test_hypotenuse():
  assert hypotenuse(3, 4) == 5
  assert hypotenuse(5, 12) == 13

test_hypotenuse()
```

As in HW3, HW4, HW5

What Not to Test: **Input types** not described in the specification

```
def abs(x):
    """
    Takes in an integer x and returns the absolute value of
    that integer.
    """
```

Examples of inputs that are unnecessary to test:

```
abs(0.01)
abs('hi')
abs([])
```

What Not to Test: **Behavior** not described in the specification

```
def roots(a, b, c):
    """
    Returns a list of the two roots of ax**2 + bx + c = 0.
    """
```

What is wrong with this test?

```
assert sorted(roots(1, 0, -1)) == [-1, 1]
```

Come up with a Test Suite!

```
def isBigger(x, y):
    """
    Assumes x and y are ints.
    Returns True if x is greater than y, False otherwise.
    """
```

Summary

- Testing doesn't prove correctness, only increases confidence in program correctness
- Writing a good test suite is hard, but can use heuristics including:
  - Good coverage of **input space**
  - Good coverage of **code execution** (not always known beforehand)
  - Address **boundary cases**
- Write tests before you write the code!
- Good tests help with debugging

Today's Roadmap

1. ~~How important is testing?~~
2. ~~Testing programs vs. functions~~
3. ~~Writing Test Suites~~
4. ~~Testing Approaches~~
5. ~~Testing Guidelines~~
6. Assertions for Debugging

Jupyter Hub

Assertions are not just for test cases

- You an use assertions throughout your code
- Documents what you think is true about your algorithm or data structure
  - E.g., `assert 0 <= index < len(my_list)`
- Lets you know immediately when something goes wrong
- The longer between a code mistake and the programmer noticing, the harder it is to debug

Assertions make debugging easier

- Common, but unfortunate, course of events:
  - Code contains a mistake (incorrect assumption or algorithm)
  - Intermediate value (e.g., in local variable, or result of a function call) is incorrect
  - That value is used in other computations, or copied into other variables
  - Eventually, the user notices that the overall program produces a wrong result
  - Where is the mistake in the program? It could be anywhere.
- Suppose you had 10 assertions evenly distributed in your code
  - When one fails, you can localize the mistake to 1/10 of your code (the part between the last assertion that passes and the first one that fails)

Where to write assertions

- **Function entry**: are arguments of expected type/size/value/shape?
  - Place blame on the caller before the function fails
- **Function exit**: is result correct?
- Places with tricky or interesting code
- Assertions are ordinary statements; e.g., can appear within a loop:

```
for n in my_numbers:
    assert type(n) == int or type(n) == float
```

Where not to write assertions

- Don’t clutter the code (Same rule as for comments)
- Don’t write assertions that are certain to succeed
  - The existence of an assertion tells a programmer that it might possibly fail

- Don’t need to write an assertion if the following code would fail informatively:

```
a = 5
assert a == 5  # Not needed!
```

- Write assertions where they may be useful for debugging

```
assert type(name) == str
print("Hello, " + name)  # If this fails the exception will be useful
```