Testing

Andrew S. Fitz Gibbon

UW CSE 160

Winter 2022

Testing

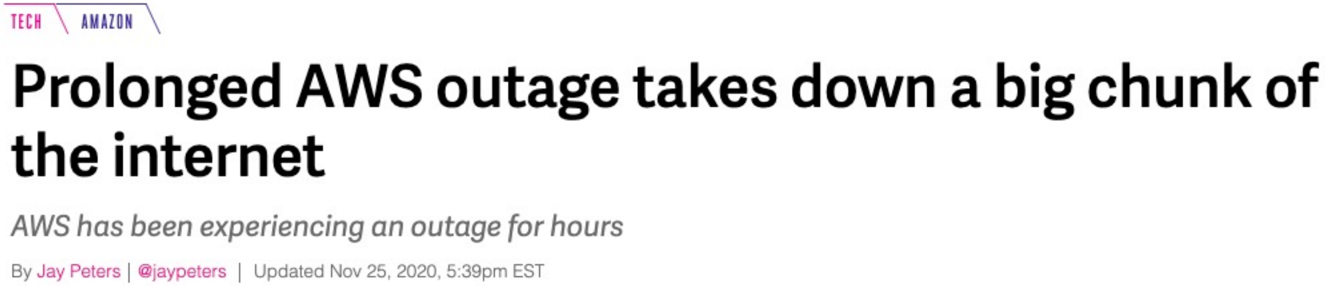
- Programming to analyze data is powerful
- It’s useless (or worse!) if the results are not correct
- **Correctness is far more important than speed**

Famous examples

- Ariane 5 rocket (1996)
  - fault in the software in the inertial navigation system (link)

- Therac-25 radiation therapy machine (1986/1987)
  - Fatal overdose due to software bugs and no external controls (link)

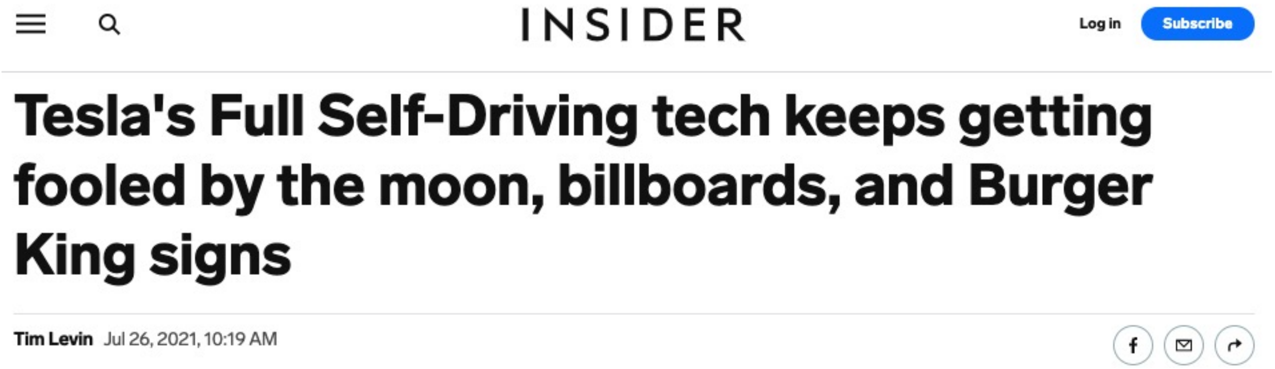
More examples


TECH AMAZON

Prolonged AWS outage takes down a big chunk of the internet

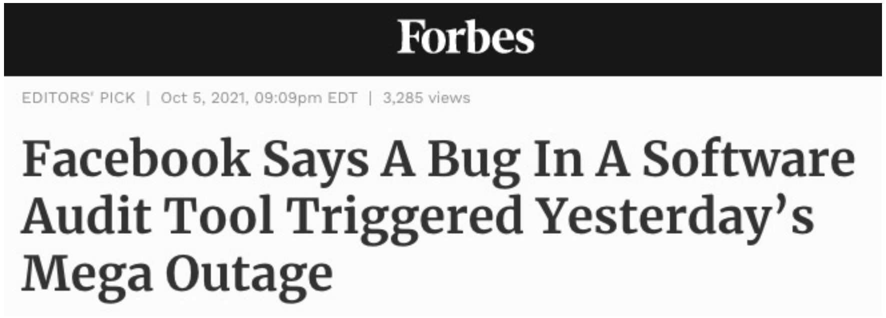
AWS has been experiencing an outage for hours

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



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



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 must 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

  2 key ideas:
  - divide and conquer
  - the scientific method

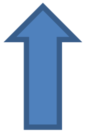
Different types of tests

- There are a lot of different types of tests…
  - Unit tests
  - Component tests
  - Integration tests
  - Performance tests
  - Security tests
  - …
- We will discuss unit testing- testing the output of individual functions/class/module is correct

How to write a test

- An example test for **sum**:

```
assert sum([1, 2, 3]) == 6
```



Call the function

How to write a test

- An example test for **`sum`**:

```
assert sum([1, 2, 3]) == 6
```

Input (sometimes called "test data")

- Input should be simple, easy to calculate the expected output by hand

How to write a test

- An example test for **`sum`**:

```
assert sum([1, 2, 3]) == 6
```

Expected output

How to write a test

- An example test for **`sum`**:

```
assert sum([1, 2, 3]) == 6
```

Ask Python to do
the check for us

- **`assert True`** does nothing
- **`assert False`** crashes the program
  - and prints a message

How to write a test

- An example test for **`sqrt`**:

```
assert sqrt(2) == 1.41421356237…
```

- Is this a proper way to test this function?

How to write a test

- An example test for **sqrt**:

```
~~assert sqrt(2) == 1.41421356237…~~

assert math.abs(sqrt(2) - 1.414) < 0.001
```

- Be careful about floating point comparison!

How to write a good test suite

- Test suite: a collection of test cases used to test a program
- Property:
  - Good coverage of input space
  - Good coverage of code execution (not always know beforehand)
  - Address boundary cases

Example (input space coverage)

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

What are the possible categories of values **a** can take?

a > 0, a < 0, or a = 0

Example (code coverage)

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

What are the possible paths to go through this function?

Example (code coverage)

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

assert abs(5) == 5
```

Example (code coverage)

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

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

Example (code coverage)

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

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

Example (code coverage)

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

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

Still 100% code coverage, but **`abs(1)`** won't produce the right output!

Example (boundary cases)

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

What are the possible boundary cases to test?

```
assert abs(0) == 0
```

Coming up with good test cases

- Think about and test “corner cases”
  - Numbers:
    - int vs. float values (remember not to test for equality with floats)
    - Zero
    - Negative values
  - Lists:
    - Empty list
    - Lists containing duplicate values (including all the same value)
    - Lists in ascending order/descending order
    - Mix of types in list (if specification does not rule out)

How to write a good test suite

- Test suite: a collection of test cases used to test a program
- Property:
  - Good coverage of input space
  - Good coverage of code execution (not always know beforehand)
  - Address boundary cases

Another example (discussion)

```
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.
    """
```

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

Testing approaches

- Regression testing
  - Whenever you found 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
  - Do NOT remove tests- protect 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 test cases

- At the **top level**: is run every time you load your program

```
def hypotenuse(a, b):
  … body of hypotenuse …
assert hypotenuse(3, 4) == 5
assert hypotenuse(5, 12) == 13
```

- In a **test function**: is run when you invoke the function

```
def hypotenuse(a, b):
  … body of hypotenuse …
def test_hypotenuse():
  assert hypotenuse(3, 4) == 5
  assert hypotenuse(5, 12) == 13
# test_hypotenuse()
```

What not to test

- Input types not described in the specification

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

Example of unnecessary tests:

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

What not to test

- Function behaviors 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 roots(1, 0, -1) == [-1, 1]
```

The **specification** did not imply that this should be the *order* these two roots are returned.

What not to test

- Use the output of your function as the expected output
- A common **mistake**:
  1. Write the function
  2. Make up test **inputs**
  3. Run the function
  4. Use the result as the expected output – BAD!!
- You didn’t write a full test: only half of a test!
  - Created the tests inputs, but not the expected output, so does not guarantee correctness

It's HARD to write good tests!

- Requires:
  - Good understanding of specification and function behavior with different input
  - Overcoming confirmation bias (especially if you have already written the code)
    - Adopt an adversarial mindset

Assertions are not just for test cases

- Use assertions throughout your code
- Documents what you think is true about your algorithm
  - E.g., `assert 0 <= index < len(mylist)`
- Let 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)

Conclusion

- Testing doesn't prove correctness, only increase confidence
- Writing a good test suite is hard, but can use heuristics including:
  - Good coverage of input space
  - Good coverage of code execution (not always know beforehand)
  - Address boundary cases
- Good tests help with debugging

Next step ☺

- Try adding more tests for your homework!
  - Only after you make sure you know what the function behavior should be, of course…
- Add more tests for your final!
  - Our provided tests won't cover all cases- up to you to read the specification carefully and cover all grounds!