

Testing software, and building Python packages

Data Sciences Institute
University of Toronto

Simeon Wong

Course objective

How to write robust software in a team that we, our colleagues, and the public can trust and use with confidence.

Course overview

- ✓ 1. Version control with Git
- ✓ 2. Command-line Python & Configuration
- ✓ 3. Raising errors
- 4. Testing software**
- 5. Application programming interfaces
- 6. Modules and packages

Today's learning outcomes

- I can explain the importance of testing code
- I can write unit tests and integration tests using `pytest`
- I can run `pytest` from the command line and calculate code coverage

Testing software

Why are tests an integral part of coding?

A large-scale study on research code quality and execution

Ana Trisovic,¹ Matthew K. Lau,² Thomas Pasquier,³ and Mercè Crosas¹

► Author information ► Article notes ► Copyright and License information ► PMC Disclaimer

Associated Data

► Data Citations

► Data Availability Statement

Abstract

This article presents a study on the replication datasets at the Harvard group of scientists and published to and reproducibility. For this study, we define ten questions to address aspects impacting research reproducibility and reuse. First, we retrieve and analyze more than 2000 replication datasets with over 9000 unique R files published from 2010 to 2020. Second, we execute the code in a clean runtime environment to assess its ease of reuse. Common coding errors were identified, and some of them were solved with automatic code cleaning to aid code execution. We find that 74% of R files failed to complete without error in the initial execution, while 56% failed when code cleaning was applied, showing that many errors can be prevented with good coding practices. We also analyze the replication datasets from journals' collections and discuss the impact of the journal policy strictness on the code re-execution rate. Finally, based on our results, we propose a set of recommendations for code dissemination aimed at researchers, journals, and repositories.

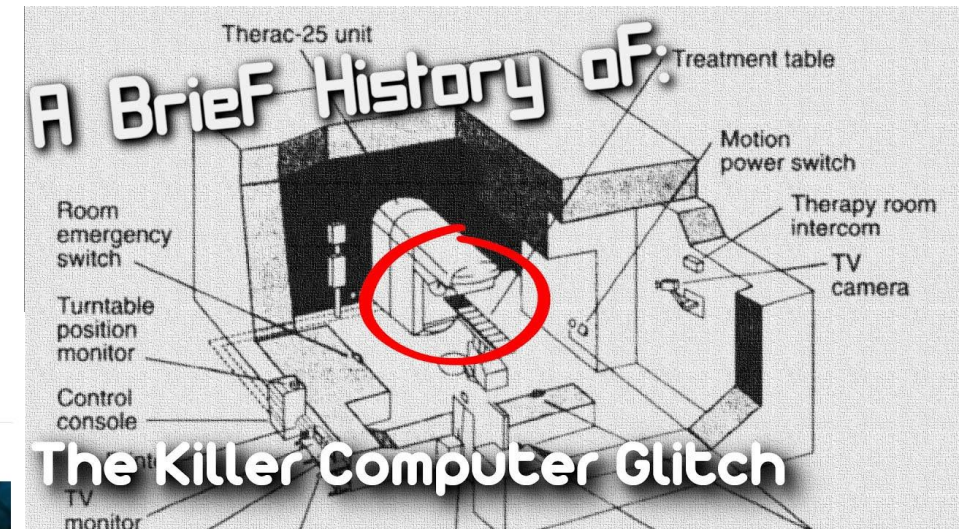
Subject terms: Research data, Software, Information technology

Introduction

Knight Capital Says Trading Glitch Cost It \$440 Million

BY NATHANIEL POPPER AUGUST 2, 2012 9:07 AM 356

Runaway Trades Spread Turmoil Across Wall St.



Plainly Difficult

The New York Times

Boeing Starliner Flight's Flaws Show 'Fundamental Problem,' NASA Says

A software glitch that could have destroyed the capsule was fixed in orbit, during an uncrewed December test flight that had already gone awry.

Testing software

Why are tests an integral part of coding?

- We should **always consider** how the code needs to be tested when writing it
 - Impact and risk (code authority) vs. cost of testing
 - When using `rm *.txt`, it's easy to double-check for typos and run `ls *.txt` first
 - It's probably not worth writing an entire testing program with SOPs
 - Using `ls *.txt` is very low risk, so might decide to run first and check that way
 - Which parts of the code need more focus?
 - What are edge/unexpected cases that the code might need to handle?
 - Testing the code's ability to gracefully and accurately handle errors

Testing paradigms

Testing paradigms

Testing as part of coding process

Two common paradigms for testing

1. Test-driven development
2. Checking-driven development

Testing paradigms

Test-driven development

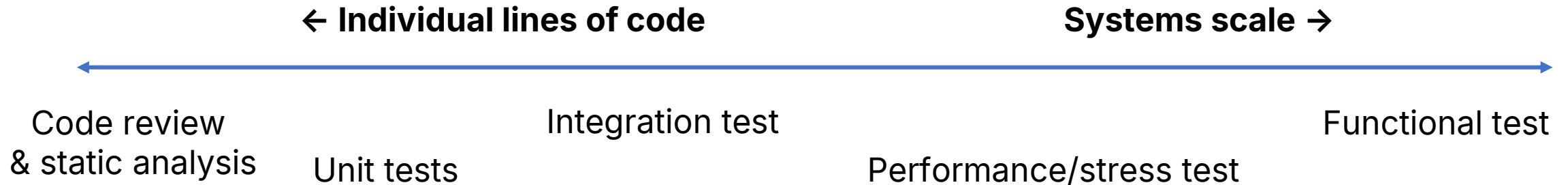
- Rather than writing code and then writing tests, we write tests first and then write just enough code to make them pass
- Advocates claim that it leads to better code because:
 - Writing tests clarifies what the code is supposed to do.
 - It eliminates confirmation bias.
 - If someone has just written a function, they are predisposed to want it to be right, so they will bias their tests towards proving that it is correct instead of trying to uncover errors.
 - Writing tests first ensures that they get written.

Checking-driven development

- Writing just a few lines of code and testing it before moving on rather than writing several pages of code and then spending hours on testing
- For example: every time we add a step to our pipeline
 - Look at its output
 - Write a test or check of some kind to the pipeline
 - Ensure that what we are checking remains true if it were run on other data or if the pipeline evolves

Testing paradigms

Some types of software testing



Other types of testing:

- Security tests
- Usability tests
- Acceptance tests

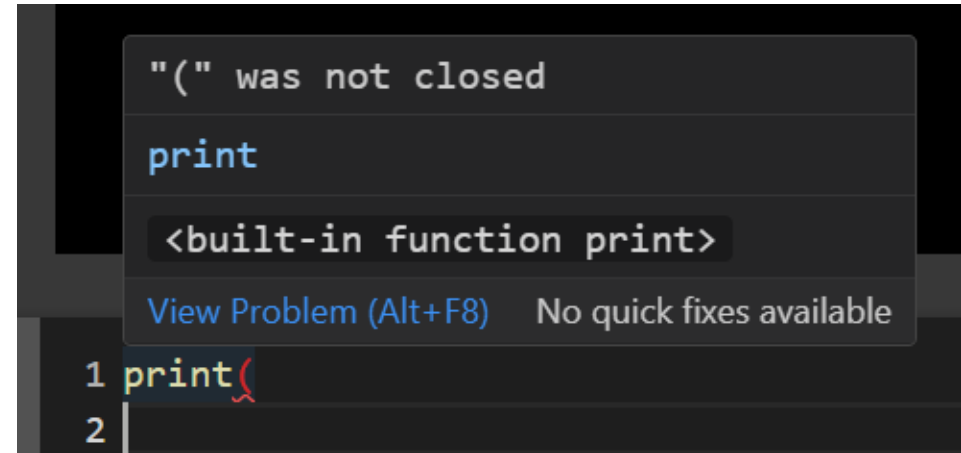
Tests get more expensive when
hardware / systems / humans are
in-the-loop

All these tests are part of a comprehensive quality strategy

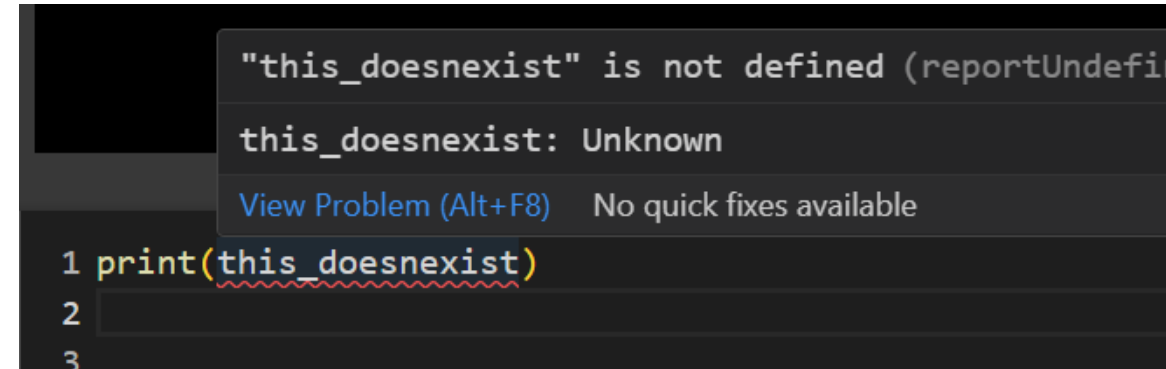
Static analysis

Static analysis

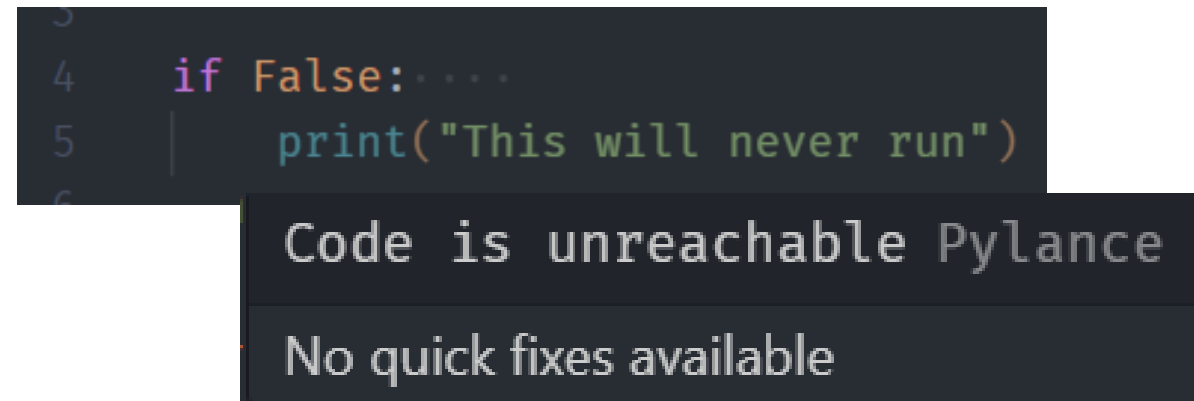
- Like spell check, but for code
- Catches some common mistakes
 - Use before declare
 - Variable type issues
 - Syntax errors
- Common analysis engines:
 - Python: pylance, pyflake
 - Bash: shellcheck



```
"("(" was not closed
print
<built-in function print>
View Problem (Alt+F8) No quick fixes available
1 print(
2 |
```



```
"this_doesnexist" is not defined (reportUndefinedVariable)
this_doesnexist: Unknown
View Problem (Alt+F8) No quick fixes available
1 print(this_doesnexist)
2 |
3 |
```



```
4 if False:
5 |     print("This will never run")
6 |
Code is unreachable Pylance
No quick fixes available
```

Unit tests

Unit tests

- A **unit test** checks of the correctness of a single unit of software.
 - What constitutes a “unit” is subjective, but typically it means the behavior of a single function in one situation.
- A unit test will typically have:
 - a **fixture**, which is the thing being tested (e.g., an array of numbers);
 - an **actual result**, which is what the code produces when given the fixture; and
 - an **expected result** that the actual result is compared to
- Easy to automate!

Unit tests

The pytest framework

- **pytest** is a testing framework that simplifies the creation, organization, and execution of tests.
 - Tests are put in files whose names begin with `test_`.
 - Each test is a function whose name also begins with `test_`.
 - These functions use `assert` to check results.
- To add more tests, we simply write more `test_` functions in this `py` file!

Unit tests

The pytest framework

- The pytest library can be used from the command-line.
- When we run it with no options, it searches for all files in the working directory whose names match the pattern `test_*.py`

```
$ pytest
```

```
===== test session starts =====  
platform darwin -- Python 3.7.6, pytest-6.2.0, py-1.10.0,  
pluggy-0.13.1  
rootdir: /Users/amira  
collected 1 item  
  
bin/test_zipfs.py . [100%]  
  
===== 1 passed in 0.02s =====
```

Unit tests

\$> Interactive live coding

- Write function to calculate the area of a square:

```
calc_area_square(side_length:float) -> float
```

- Write a unit test for this function

Unit tests

\$> Interactive live coding

```
import math
```

```
def calc_area_square(side_length: float) -> float:  
    area = side_length ** 2  
    return area
```

```
def test_calc_area_square():  
    assert calc_area_square(1) == 1  
    assert calc_area_square(0) == 0  
    assert calc_area_square(2) == 4  
    assert calc_area_square(3) == 9
```

```
~/codedir$ pytest
```

The pytest framework

- pytest includes functions to write tests for special circumstances:
 - approx – for floating point calculations where some tolerance is expected
 - raises – for checking if an error is raised given erroneous input

Unit tests

\$> Interactive live coding

- Write function to calculate the area of a circle:

```
calc_area_circle(radius:float) -> float
```

- Write a unit test for this function

Unit tests

\$> Interactive live coding

```
import math
from pytest import approx

def calc_area_circle(radii):
    areas = math.pi * radii**2
    return areas

def test_calc_area_circle():
    inputs = [2, 4, 10]
    exp_output = [12.5664, 50.2655, 314.159]

    for i, e in zip(inputs, exp_output):
        actual_output = calc_area_circle(i)
        assert actual_output == approx(e, rel=1e-3, abs=1e-3)
```

```
~/codedir$ pytest
```


\$> Interactive live coding

- Write function to calculate the area of a circle:

```
calc_area_circle(radius:float) -> float
```

- Add a test of whether the input is a valid number
- Add a test of whether the input is positive
- Write a unit test for these errors

Unit tests

\$> Interactive live coding

```
import math
from pytest import approx, raises

def calc_area_circle(radii):
    if not isinstance(radii, float):
        raise TypeError("The radius must be a number")
    if radii < 0:
        raise ValueError("The radius cannot be negative")
    areas = math.pi * radii**2
    return areas

def test_calc_area_circle_raises():
    with raises(TypeError):
        calc_area_circle("a")
    with raises(ValueError):
        calc_area_circle(-1)
```

```
~/codedir$ pytest
```

Integration testing

Integration testing

- Integration testing is a test that checks whether the parts of a system work properly **when put together**.
- Integration tests are structured the same way as unit tests:
 - a **fixture** is used to produce an **actual result** that is compared against the **expected result**.
 - However, creating the fixture and running the code can be considerably more complicated

Integration testing

Example

- Testing a function that reads a list of radii from a text file and calculates the corresponding areas:

```
def test_load_and_calculate():  
    ##### Generate test case #####  
    # save list of radii  
    np.array([5.2, 1.1, 9.3, 11.4, 19.2]).savetxt('radii.txt')  
  
    ##### Run function #####  
    # load list of radii  
    radii_list = load_from_file('radii.txt')  
  
    # calculate area  
    area_list = calc_area_circle(radii_list)  
  
    ##### Test result #####  
    assert area_list == approx([84.95, 3.8, 271.7, 408.3, 1158.1])
```

Regression tests

Regression tests

- When we don't know the answer, we can use regression tests to compare today's answer with a previous one.
- This doesn't guarantee that the answer is right
 - if the original answer is wrong, we could carry that mistake forward indefinitely
 - draw attention to any changes (or "**regressions**")
- Especially useful for large projects:
 - When dependencies are updated
 - When code is re-used in multiple places for different contexts

Test coverage

Test coverage

- How much of our code do we have unit and integration tests for?
- We can instruct `pytest` to get the percentage of lines of code that have been tested.

```
$ colab>> pytest --cov=mymodule
===== test session starts =====
platform linux -- Python 3.10.12, pytest-7.4.3, pluggy-1.3.0
rootdir: /content
plugins: cov-4.1.0, anyio-3.7.1
collected 1 item

test_mymodule.py . [100%]

----- coverage: platform linux, python 3.10.12-final-0 -----
Name           Stmts   Miss  Cover
-----
mymodule.py      4      1    75%
-----
TOTAL             4      1    75%

===== 1 passed in 0.03s =====
```

Continuous integration

Continuous integration

- Automatically run all tests whenever a change is made
 - Visibility into which changes resulted in errors
 - Integration tests help catch errors affecting other parts of the code, or errors resulting from operating system or system interactions
- GitHub Actions is a built-in way to do CI
 - Usually configured to run on every single push
 - Can run pretty much any command (e.g. pytest) on the code

Continuous integration

GitHub Actions

- Defined in a YAML file placed in `.github/workflows`

```
name: Run unit tests
on: [push]
jobs:
  my_pytest:
    runs-on: ubuntu-latest
    steps:
      - ...
```

Name of your workflow

When should this run?

What should happen when it runs?

Base system to run this action on

What steps?

Continuous integration

\$> Interactive live coding

- Create a GitHub repository for our circle code with unit tests
- Try `pytest` locally
- Write a GitHub Actions file to run `pytest` on push

Continuous integration

\$> Interactive live coding

```
name: My pytest workflow
on: [push]
jobs:
  run_pytest:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: '3.x'

      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install -r requirements.txt

      - name: Test with pytest
        run: |
          pip install pytest pytest-cov
          pytest --junitxml=junit/test-results.xml

      - name: Test Report
        uses: pmeier/pytest-results-action@main
        if: always() # run this step even if previous step failed
        with:
          path: junit/*-results.xml # Path to test results
```


Today's learning outcomes

- I can explain the importance of testing code
- I can write unit tests and integration tests using `pytest`
- I can run `pytest` from the command line and calculate code coverage

Course overview

- ✓ 1. Version control with Git
- ✓ 2. Command-line Python & Configuration
- ✓ 3. Raising errors
- 4. Testing software**
- 5. Application programming interfaces
- 6. Modules and packages

Course objective

How to write robust software in a team that we, our colleagues, and the public can trust and use with confidence.

Homework

- Refactor one section of your Homework 2 script into a function
- Write a unit test for that function
- Optionally, try running that unit test automatically with GitHub Actions
- Ungraded, submit before next class for feedback.

References

- Research Software Engineering with Python by Damien Irving, Kate Hertweck, Luke Johnston, Joel Ostblom, Charlotte Wickham, and Greg Wilson (<https://merely-useful.tech/py-rse/config.html>)