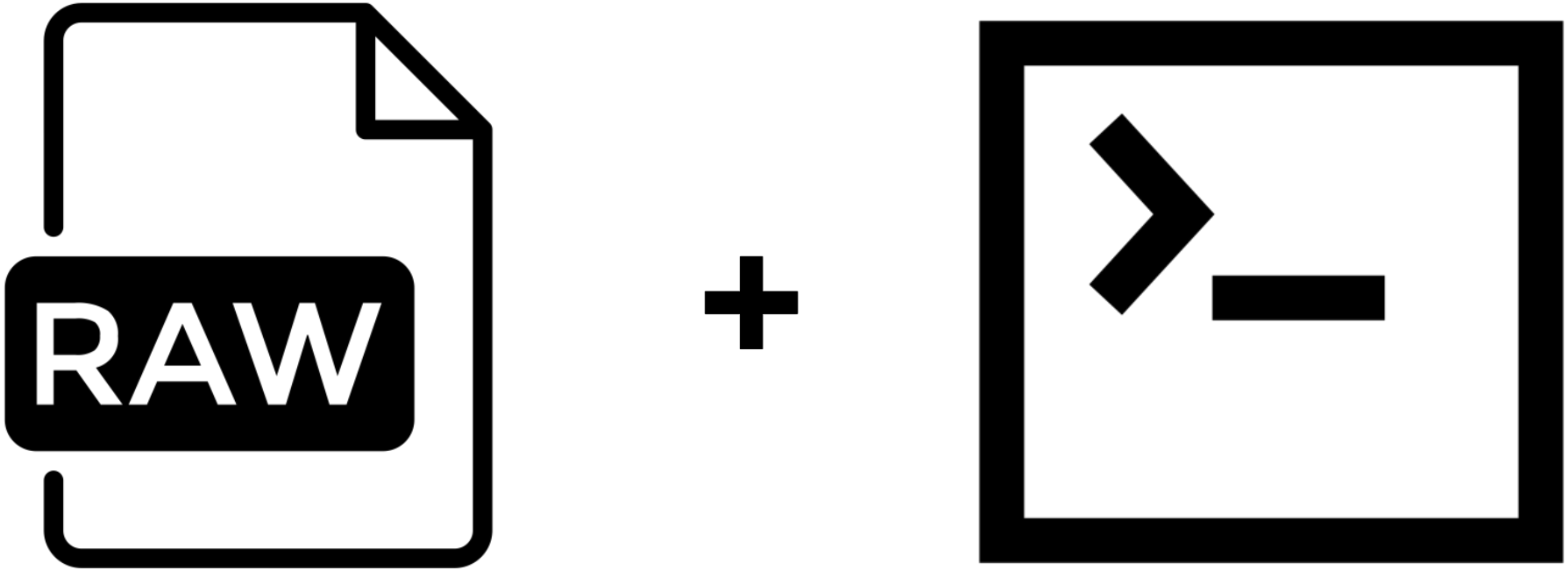
# Rethinking Software Testing for Data Science

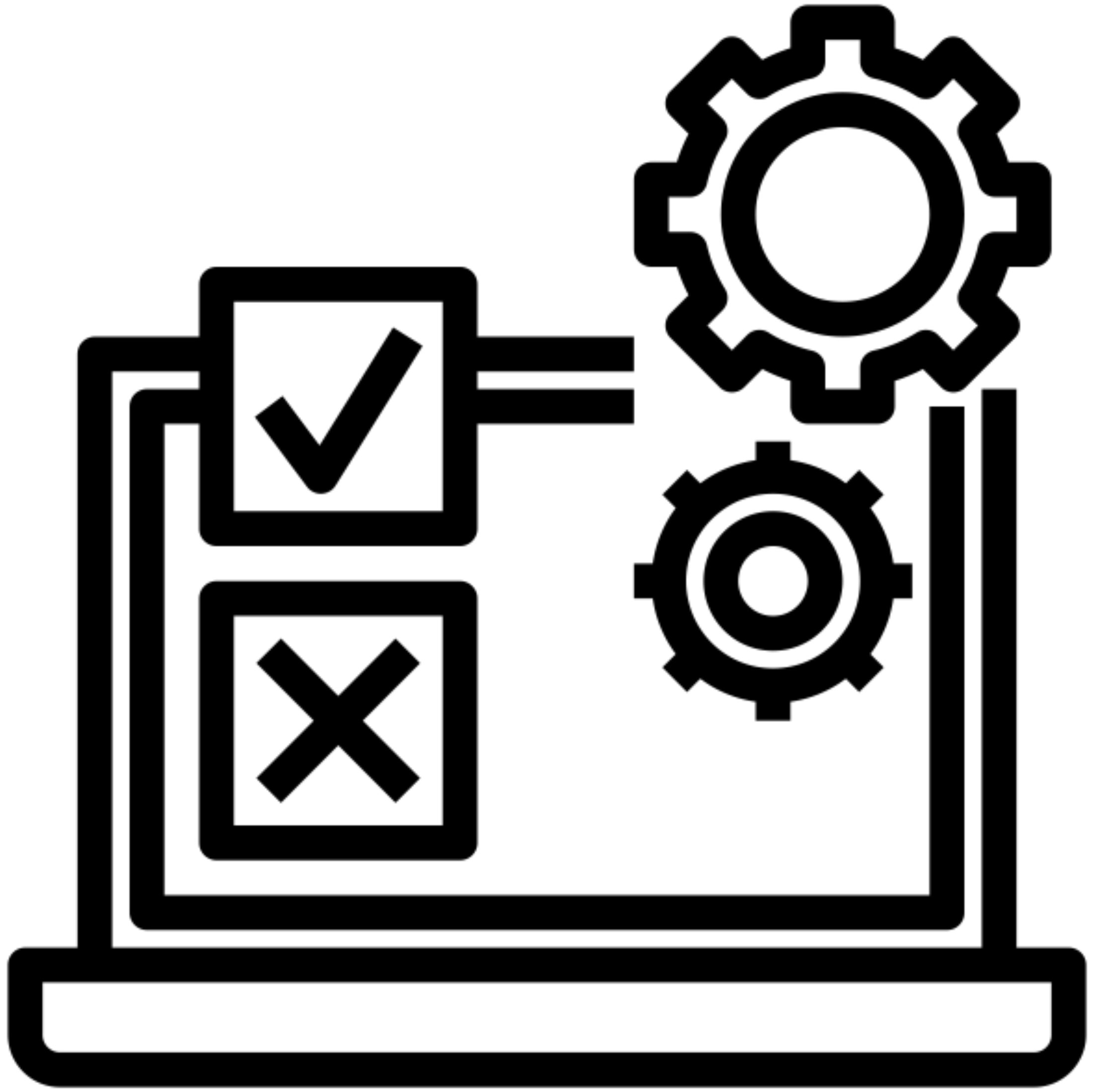
Eduardo Blancas (@edublancas)

PyData Global 2020

# The reproducibility test



# Automated testing



# Outline

1. Introduction to software testing
2. Challenges for Data Science
3. Debugging strategy
4. Tools and resources

# Standard software testing procedure

1. Programmer makes small code changes
2. Save changes (usually via `git commit`)
3. Tests are executed
4. Changes marked as success or failure
5. If success, continues to edit. Otherwise, fixes errors until tests pass

# Benefits of automated testing

- Quickly be notified when things break
- Pinpoint errors to specific code changes
- Speeds up development cycle in the long run

For testing to be effective, results have to come back quickly.

# Challenges for Data Science

1. **Test cases**. Hard to come up with tests for functions that transform data.

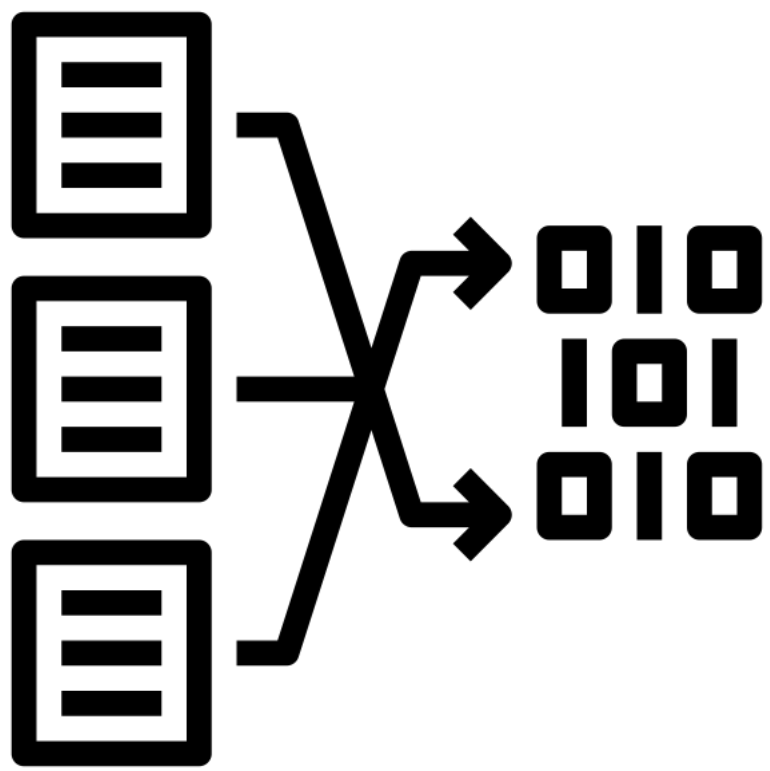
2. **Structure**. Tasks are not independent.

3. **Speed.** Data processing takes time.

4. **Changing data.** New data can break your code.

# Challenges for Data Science:

# Testing data transformations



# What can go wrong?

| user_id | timestamp | song_genre |
|---|---|---|
| 1 | 2020-01-01 | Rock |
| 1 | 2020-02-10 | Pop |
| 2 | 2020-01-15 | Jazz |
| 3 | **NA** | Jazz |
| 3 | 2020-07-28 | Rock |
| 4 | 2020-03-12 | **Unknown** |

# Testing data expectations

## Python (pandas)

```python
# Check all users have an ID
assert (not user_streams
        .user_id
        .isna().sum())

# Check no unknown genres
genres = (user_streams
          .song_genre.unique())
assert 'Unknown' not in genres
```

## SQL

```sql
SELECT NOT EXISTS(
    SELECT * FROM user_streams
    WHERE user_id IS NULL
)

SELECT NOT EXISTS(
    SELECT * FROM user_streams
    WHERE song_genre = 'UNKNOWN'
)
```

# (Unit) Testing code

- Data is expected but output is incorrect
- Break down your data transformations in small parts
- This way it's easier to test them and come up with test cases
- An effective test looks for concrete expected behavior of a single unit

```
@pytest.mark.parametrize('data',
[
    # complete case
    {'id': [1, 1, 1],
     'song_genre':
        ['Rock', 'Pop', 'Jazz']},
    # incomplete case
    {'id': [2, 2, 2],
     'song_genre':
        ['Rock', 'Pop', 'Rock']},
])
def test_count_song_genres(data):
    df = pd.DataFrame(data)
    out = count_song_genres(df)
    expected = ['Rock', 'Pop',
                'Jazz']
    assert out.columns == expected
```

# A recipe for writing data transformations

1. Write docstring

2. What am I expecting about the data? And add data tests.

3. What scenarios should my code cover? And add unit tests.

4. Code your function and run tests until they all pass

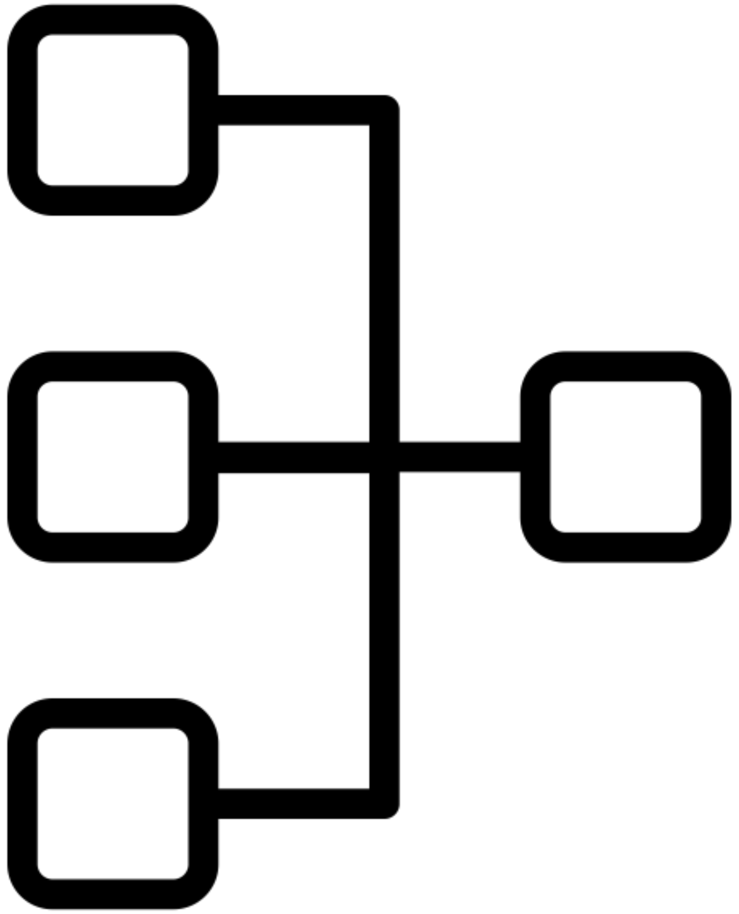
```python
def count_song_genres(df):
    """
    Counts song genres per user,
    one column per action
    (Rock, Pop or Jazz)
    """
    # code to count song genres...


def clean_song_genres(df):
    """
    Prepares user's song genres
    data for training
    """
    counts = count_song_genres(df)

    # more transformations...
```

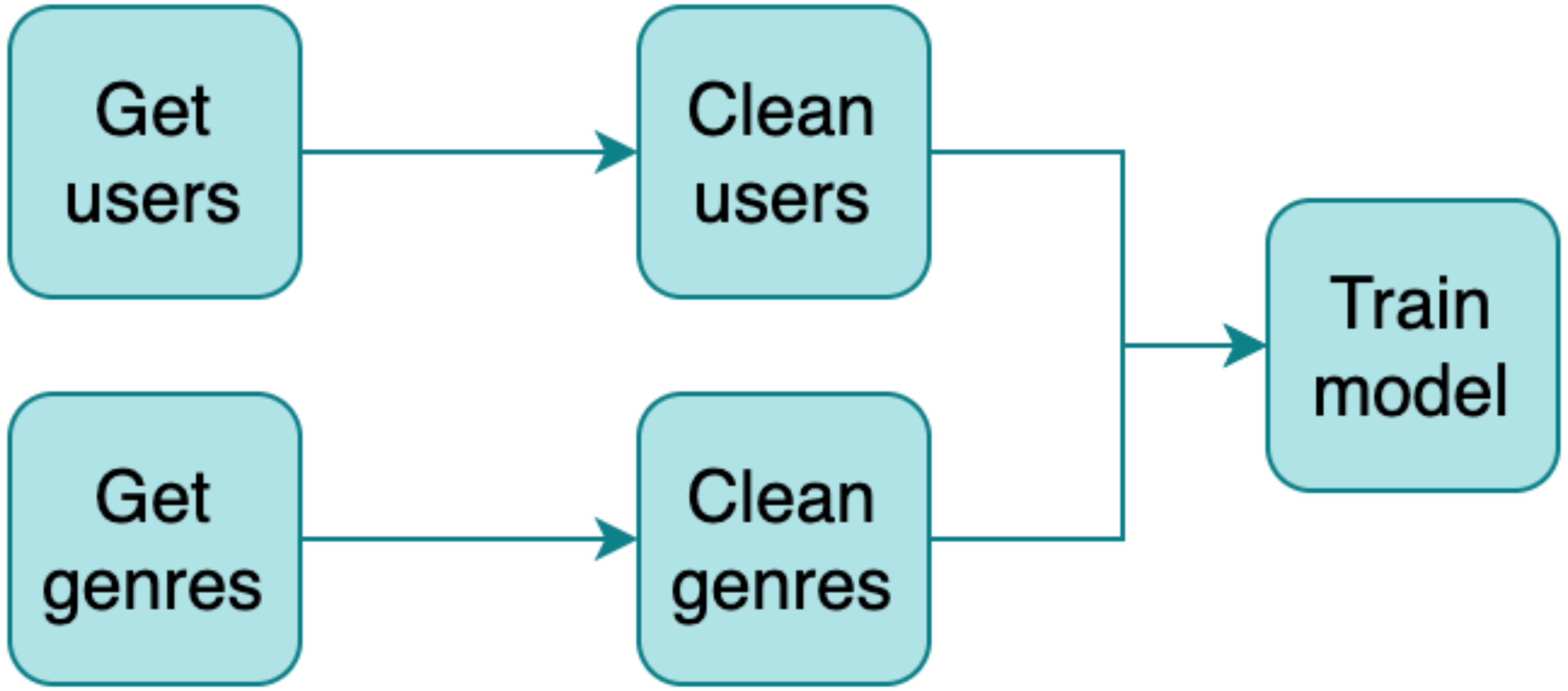
# Challenges for Data Science:

# Data pipeline structure



# Data transformations depend on each other

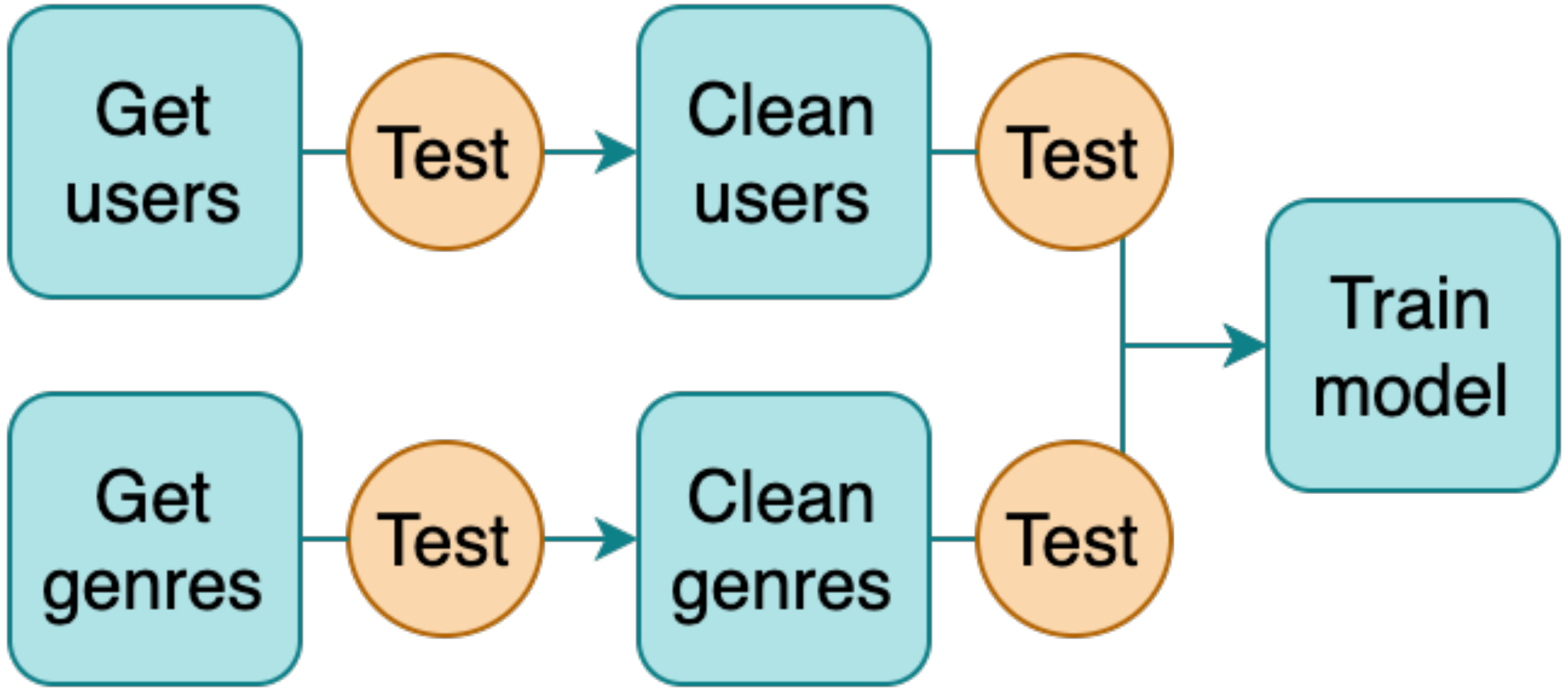
- Data processing comes in steps
- Errors propagate to any downstream task
- Testing procedure has to account for this



# From data testing to integration testing

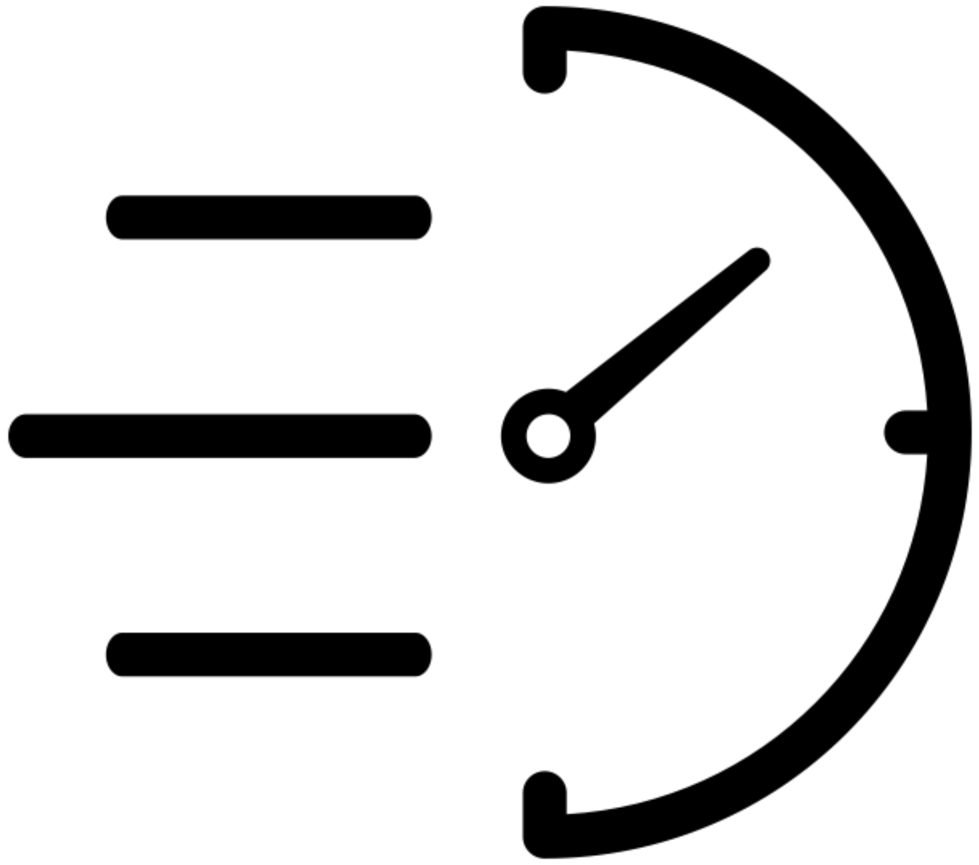
```python
def clean_song_genres():
    # code for cleaning
    # song genres...
    pass

def test_song_genres():
    # data tests here...
    pass
```



Challenges for Data Science:

Speed

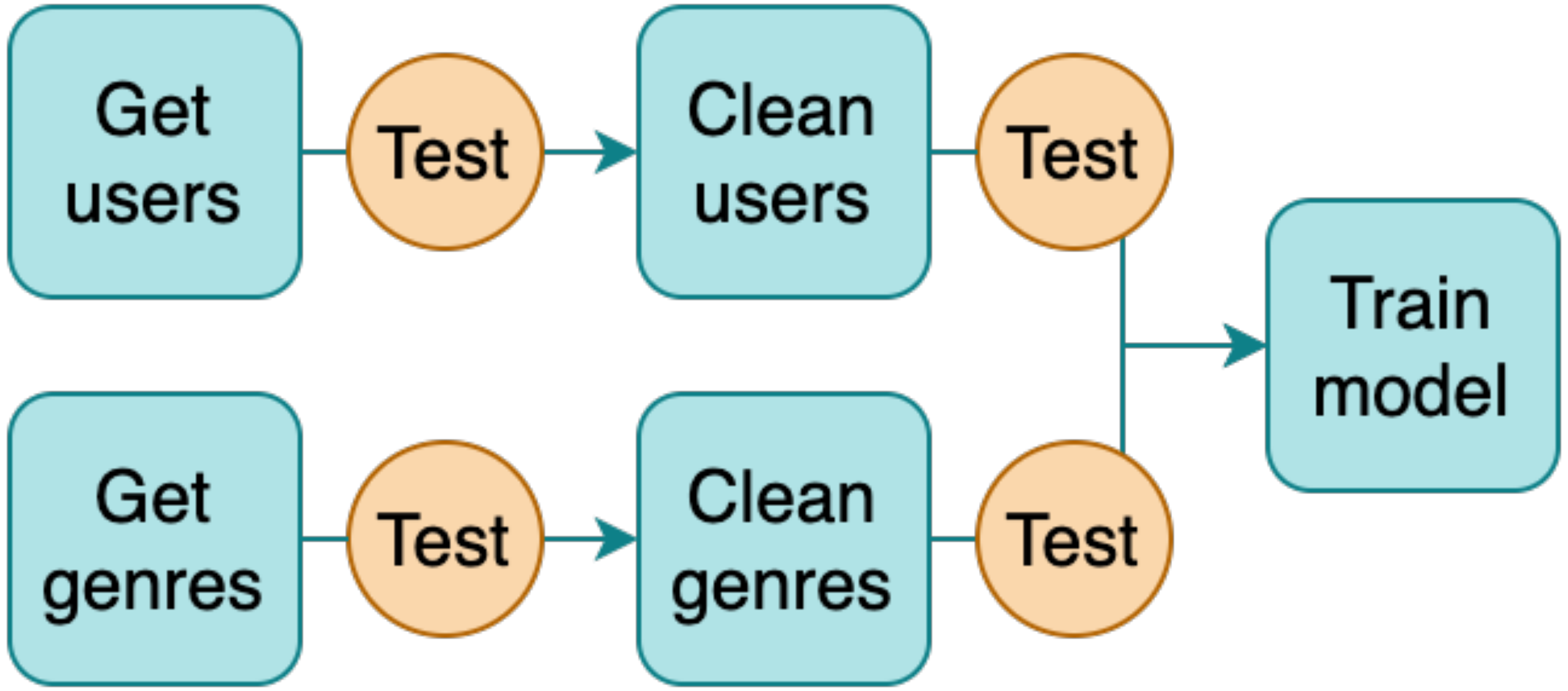


# Integration testing with a sample

```yaml
# pipeline.yaml (Ploomber)
tasks:
    - source: get_users.py
      # integration test
      # defined in tests.py
      on_finish: tests.get_users
      params:
        sample: true

    - source: get_song_genres.py
      on_finish: tests.clean_song_genres

    # continues for each task...
```

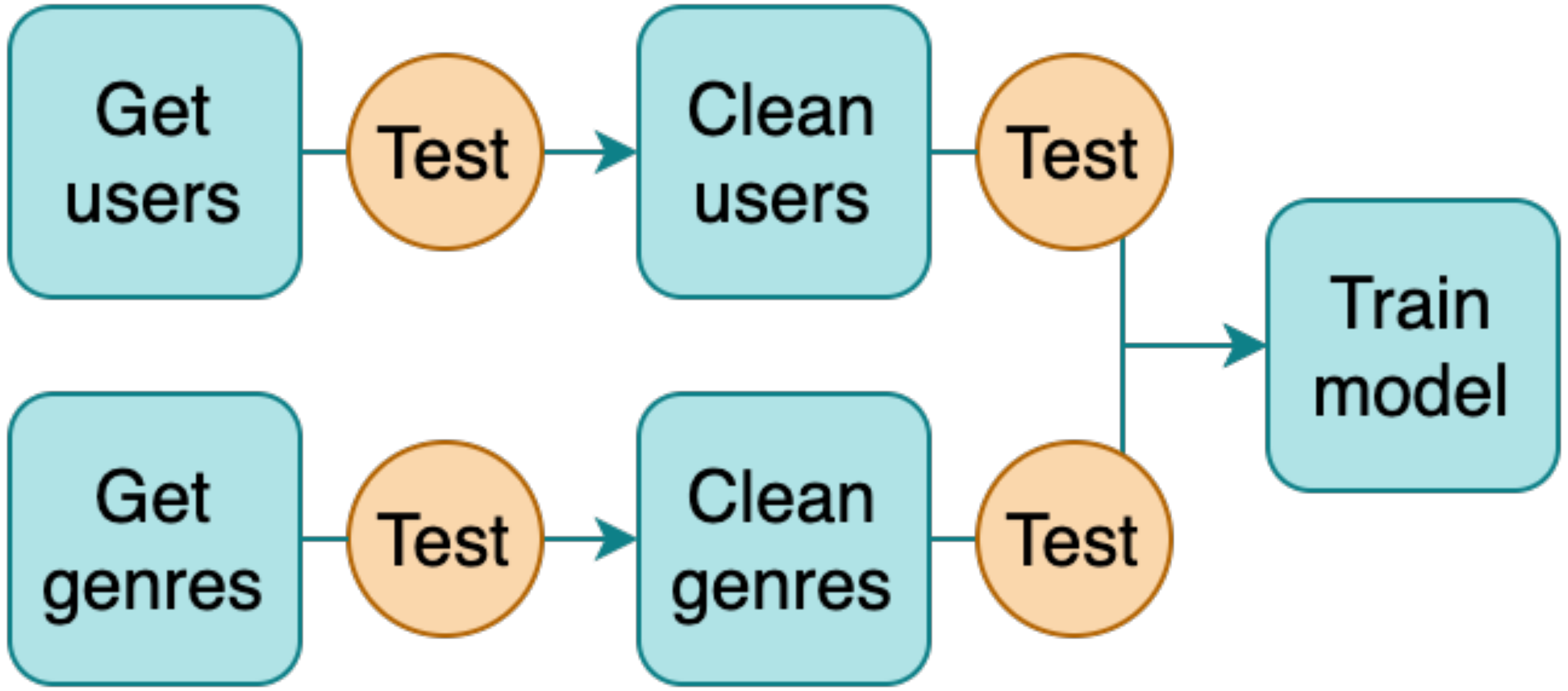



Ploomber was presented at JupyterCon 2020, talk available on Youtube.

**Challenge: Speed**

# Other options

- Incremental builds
  - Only run outdated tasks
- Task parallel execution
  - Run pipeline branches in parallel



Challenges for Data Science:

Data changes

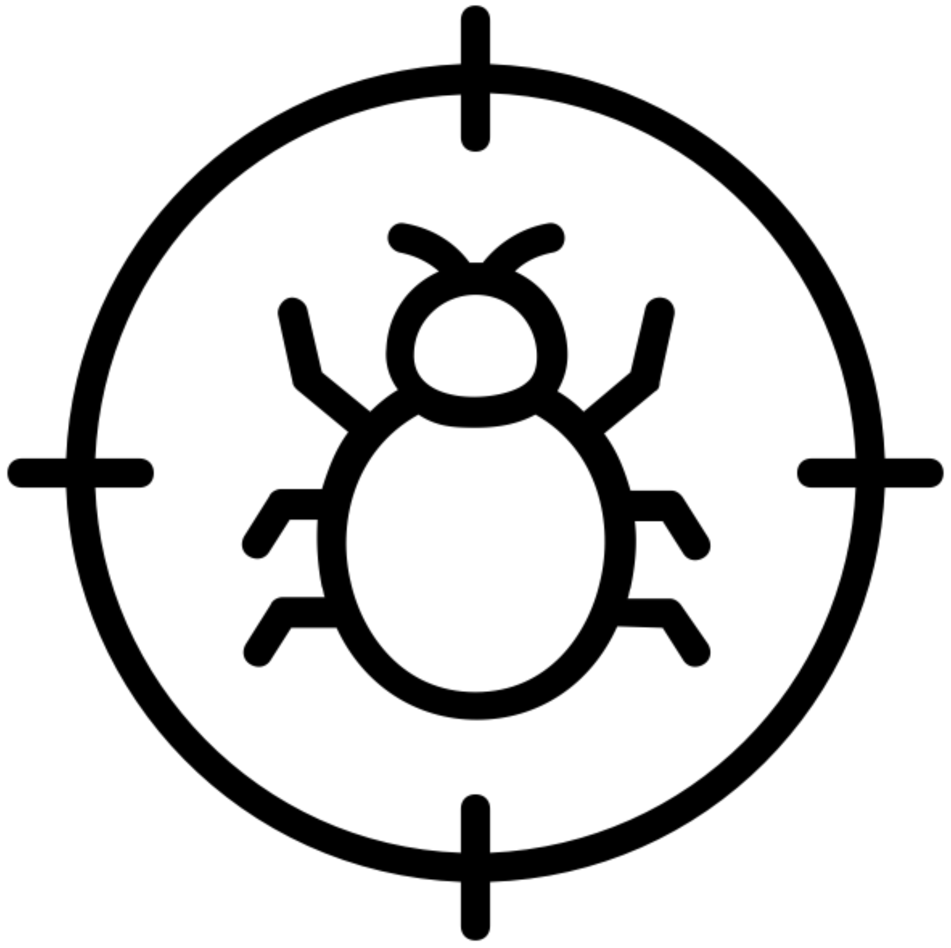
# Data changes during development

- Unrealistic to cover all possible input cases
- Focus on data tests
- Trying our new things vs testing current code tradeoff
- Code reusability

# Preparing for deployment

- A unambiguous input schema definition
- Heavily invest in unit testing
- Good error messages
- Logging to help debugging
- Batch vs live API

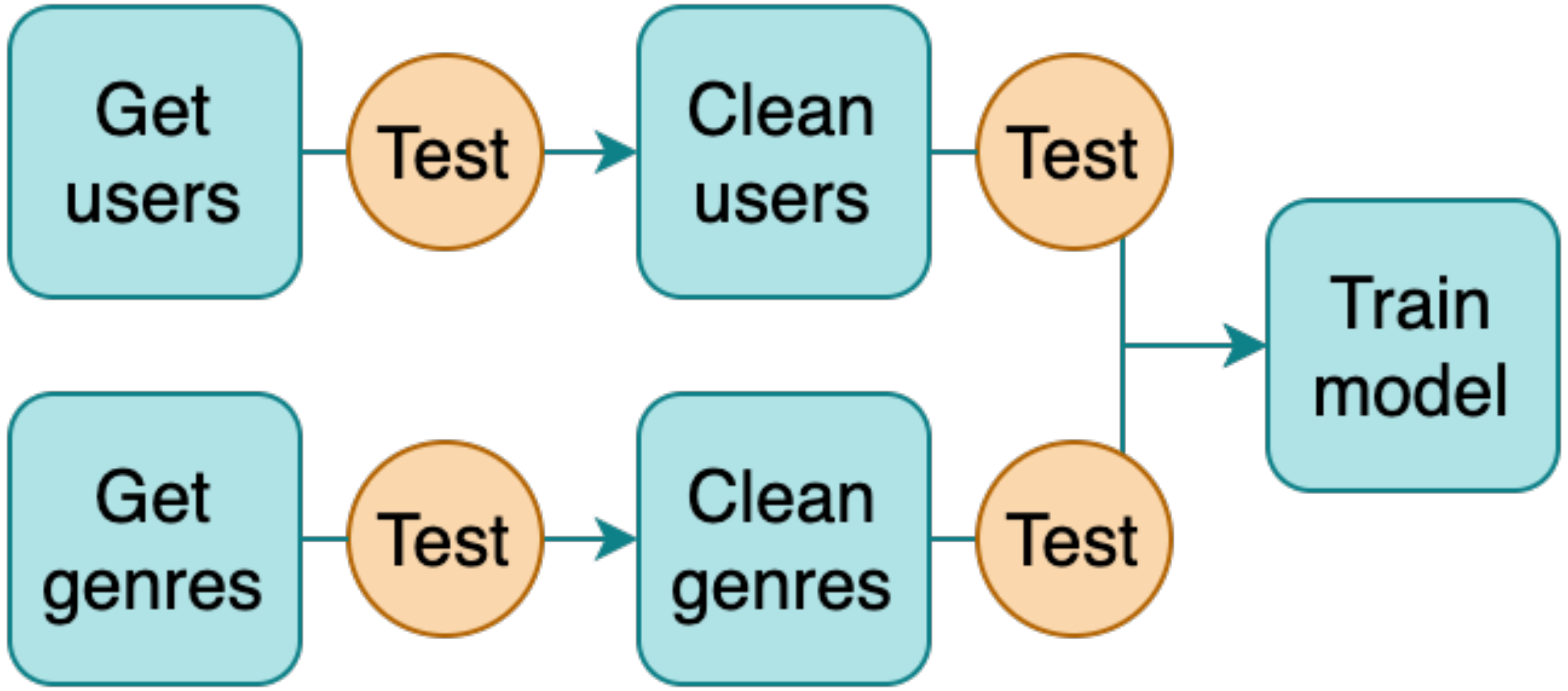
# 3. Debugging strategy



# Fixing crashes

1. Unexpected data (data test crash)
    - Relax data expectations
    - Or leave some some data out
2. Unexpected data (task crash)
    - Add data test (go back to 1)
3. Incorrect code (task crash)
    - Add unit test, see it fail, fix

Important: Fix in in the right place



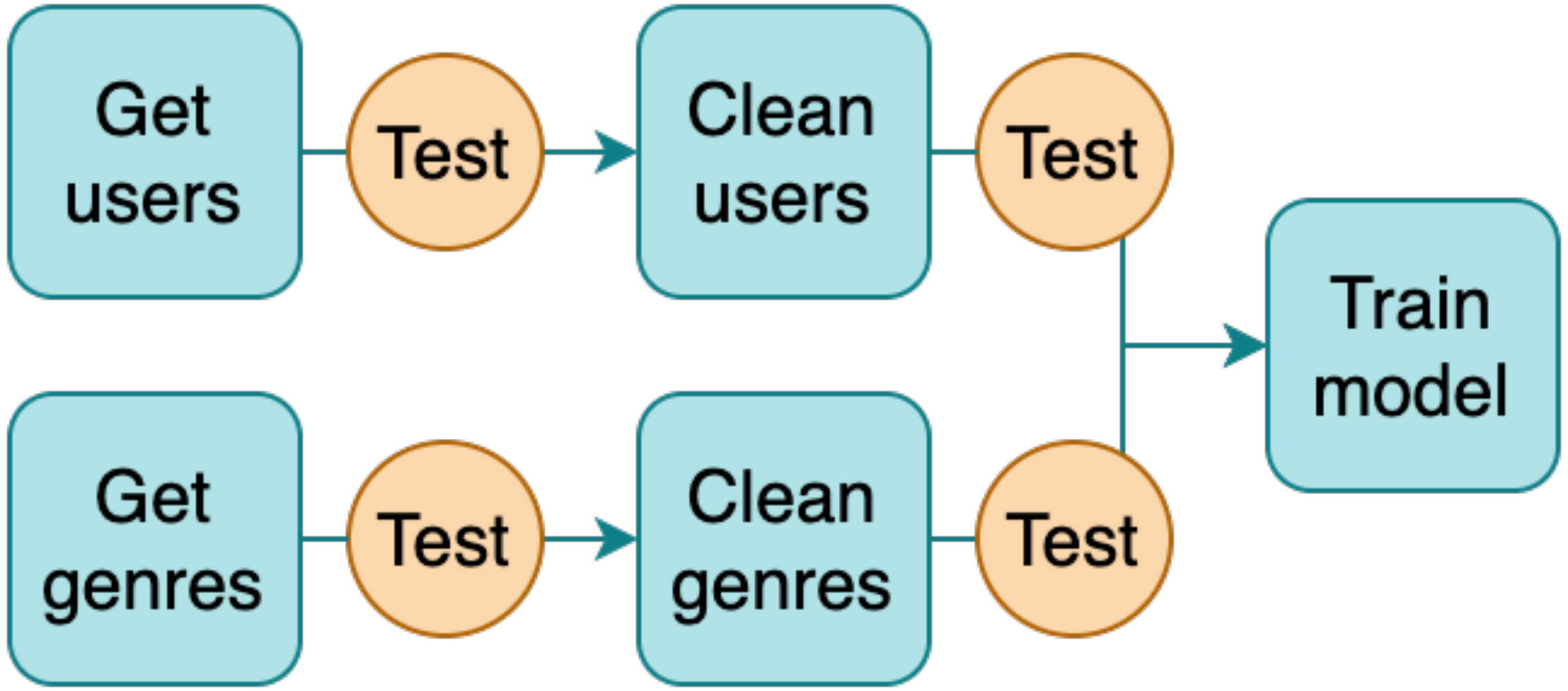
# Fixing silent bugs

Indicates a missing unit or a data test.

1. Work backwards to find the root cause

2. Add missing unit/data test

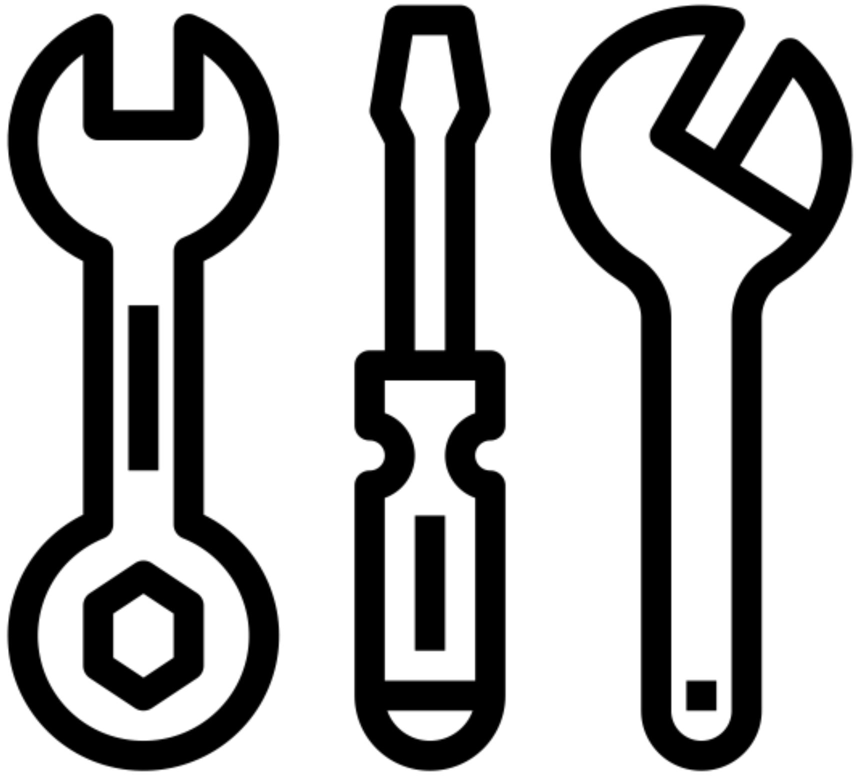
3. Apply the logic from "fixing crashes"



# Summary

- Testing makes reproducibility practical
- Unit tests check concrete expected behavior
- Integration tests check I/O boundaries
- Speed up integration tests with sampling
- Be prepared for data changes
- Write tests before you code
- Fix bugs in the right place

# 4. Tools and resources



# Tools

github.com/

- Pipeline development (+ integration testing):
  - ploomber/ploomber
- Running unit tests:
  - pytest-dev/pytest
- Creating virtual env when running tests:
  - theacodes/nox

# Resources

- Slides: blancas.io/talks/pydata-20.pdf
- Questions/Feedback? Twitter: @edublancas
- Blog post: ploomber.io/posts/ci4ds
- Code example: github.com/ploomber/projects (`ml-intermediate` folder)

**Images by Ilham Fitrotul Hayat, Jugalbandi, Template, Becris, Vadim Solomakhin, ProSymbols, Rockicon, Yoyon Pujiyono and Vichanon Chaimsuk from the Noun Project**

Thanks for watching!