Quest for Quality:

Exploring Software Labyrinths with Testing Tools and Techniques

Introduction

Hi, I'm Dave Bunten!

With the **Software Engineering Team** at **CU Anschutz DBMI**.

Department of Biomedical Informatics
SCHOOL OF MEDICINE
UNIVERSITY OF COLORADO **ANSCHUTZ MEDICAL CAMPUS**

Visit our webpage for more info!
https://cu-dbmi.github.io/set-website/

Introduction

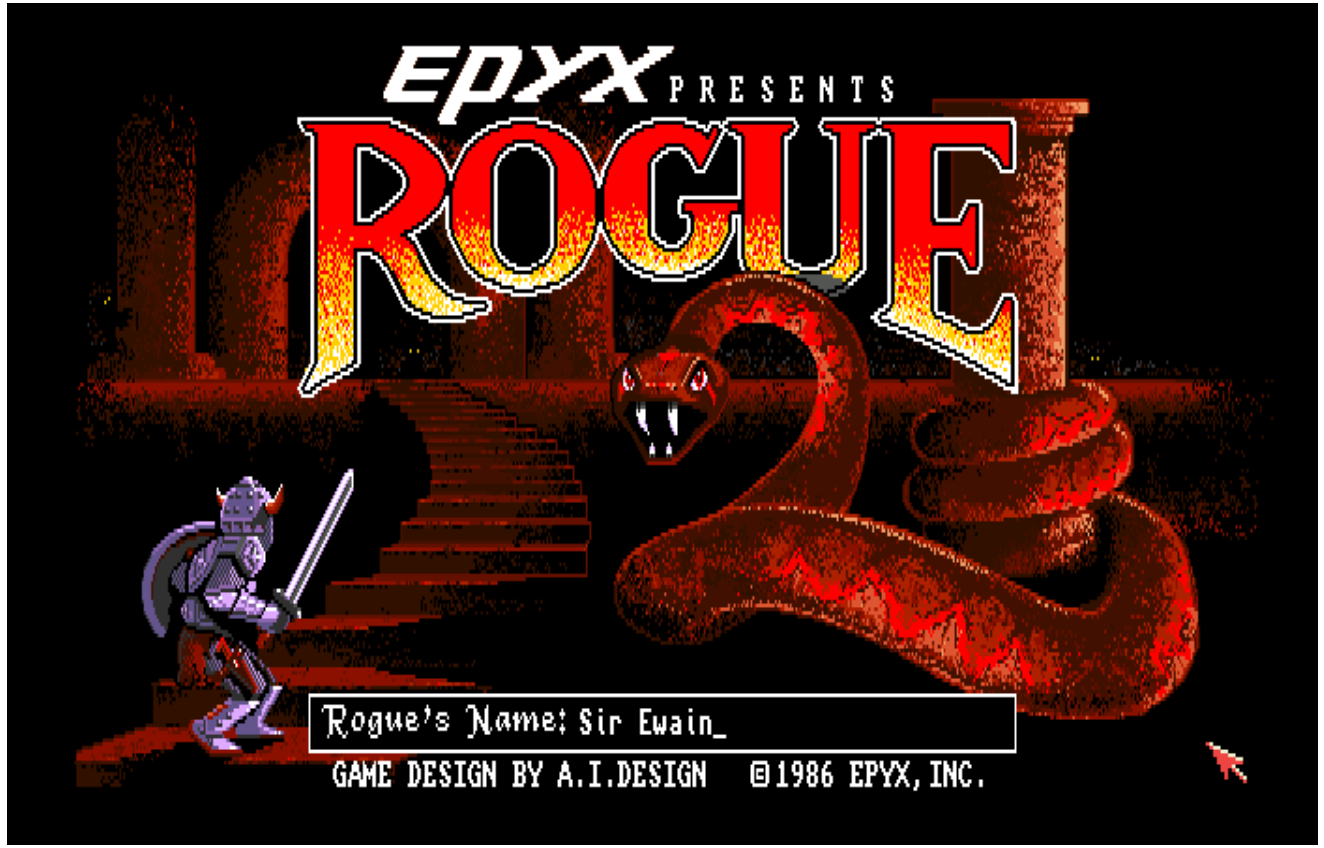
1. 🗺️ Software as an adventure
2. 😱 Dangers of the unknown
3. 🕯️ Lanterns for the dungeon

Software as an adventure

Roguelikes: procedurally generated RPG games with permadeath.

Image: Citrocipia, Wikimedia

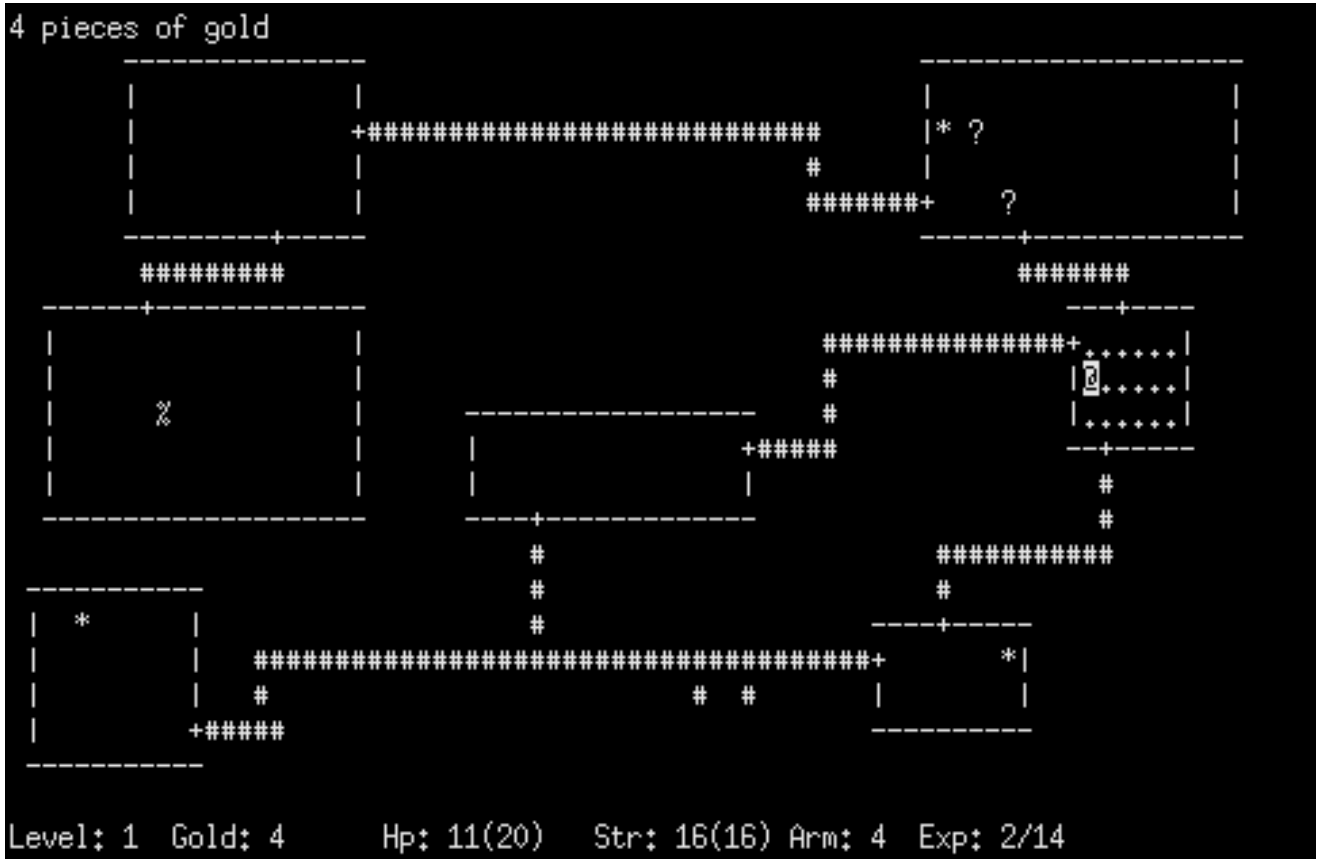
Software as an adventure



Software development can feel like a roguelike dungeon: full of surprises, risks, and hidden traps.

Image: Blake Patterson, Flickr

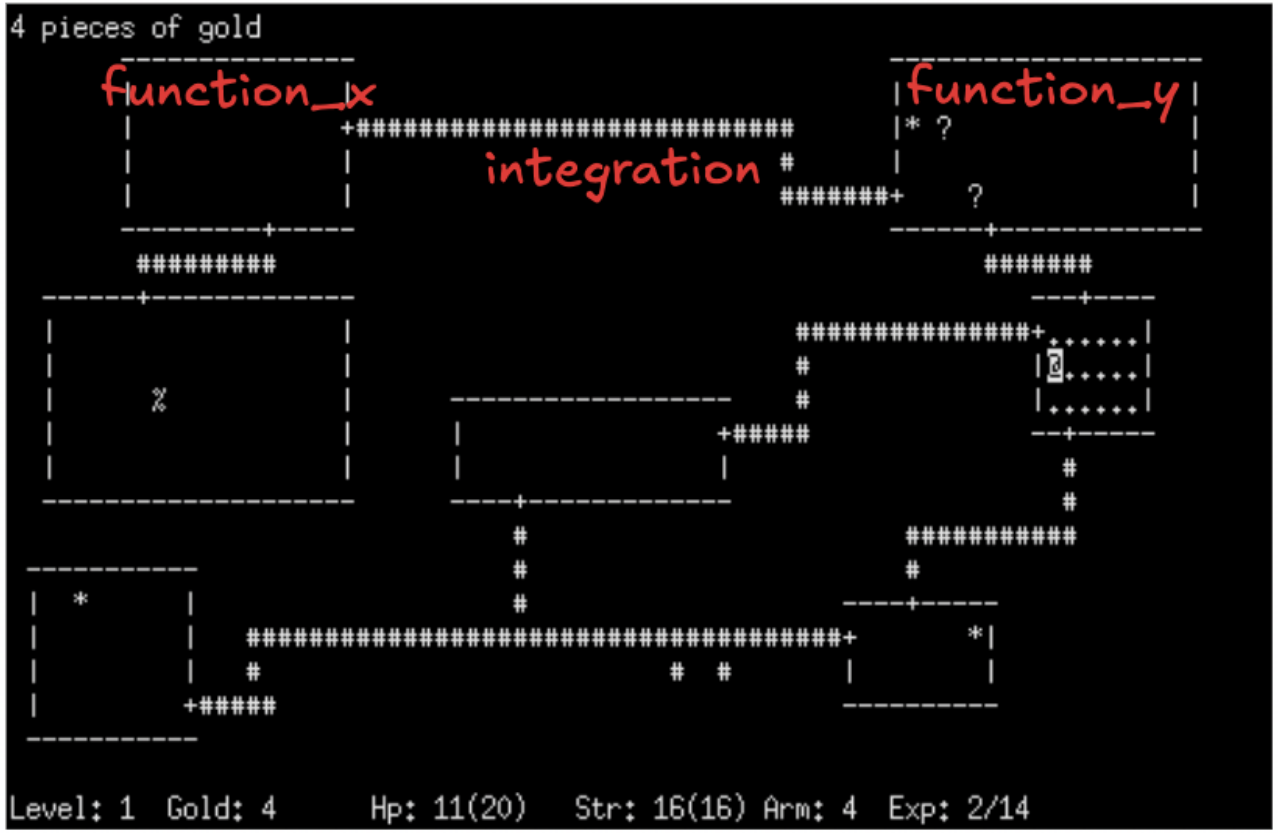
Software as an adventure



Each room we explore can include treasure or grave danger!

Image: Thedarkb, Wikimedia

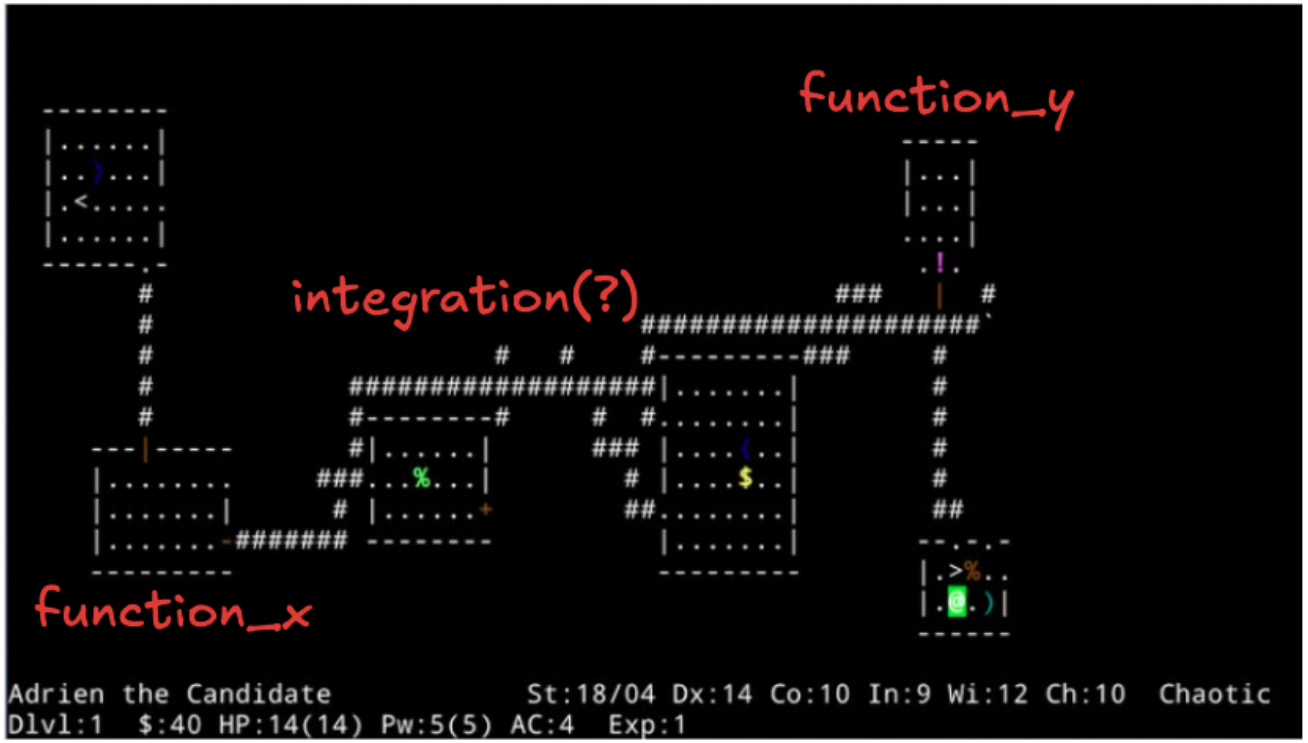
Software as an adventure



Each ~~room~~ software component we explore can include treasure or grave danger!

Image: Modified from Thedarkb, Wikimedia

Software as an adventure



The processing of each software component might change each time (especially with data or code changes).

Image: Modified from Linux Screenshots, Flickr

Dangers of the unknown

NEWS OF THE WEEK

A Scientist's Nightmare: Software Problem Leads to Five Retractions

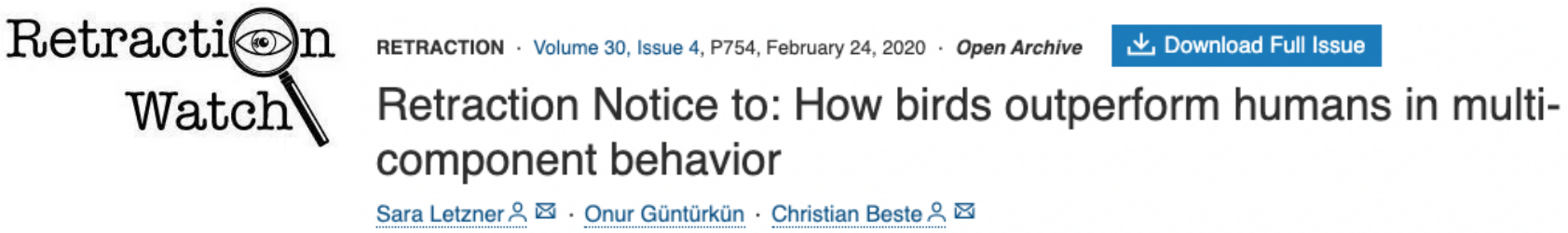
GREG MILLER Authors Info & Affiliations

SCIENCE · 22 Dec 2006 · Vol 314, Issue 5807 · pp. 1856-1857 · DOI: 10.1126/science.314.5807.1856

Adventures without armor sometimes lead to major challenges (source).

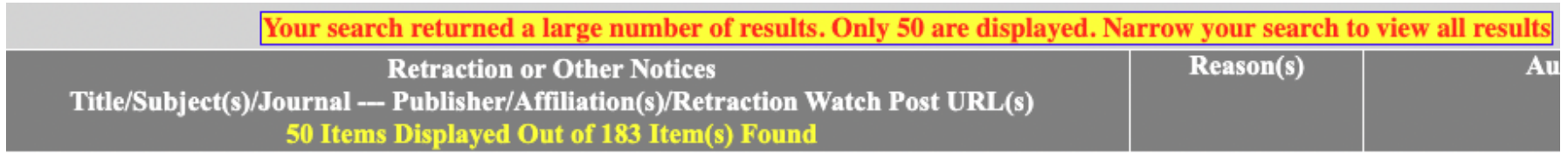
> "… a homemade data-analysis program had flipped two columns of data …"

Dangers of the unknown



> "… We have subsequently discovered, however, that the MATLAB script that was used for the analysis of reaction times in the pigeon paradigm was wrongly indexed. …" (source)

Dangers of the unknown

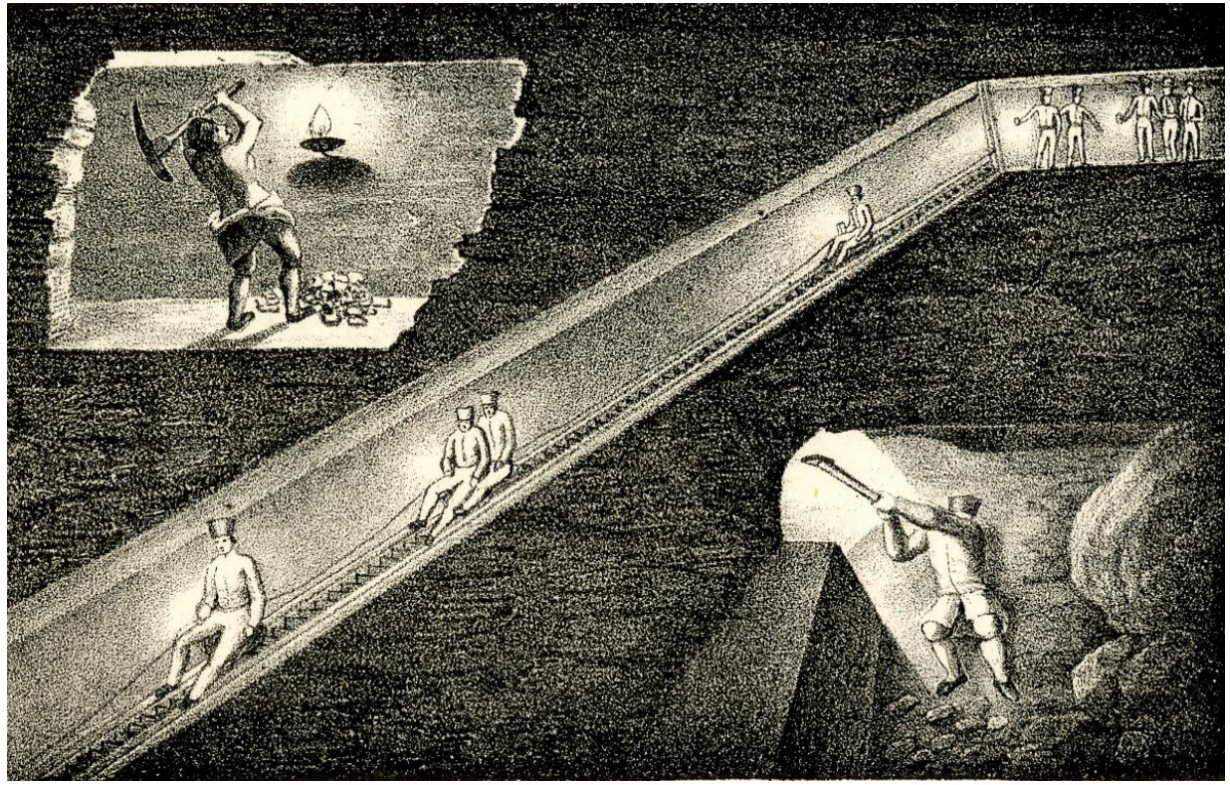

Your search returned a large number of results. Only 50 are displayed. Narrow your search to view all results

Retraction or Other Notices Title/Subject(s)/Journal — Publisher/Affiliation(s)/Retraction Watch Post URL(s) 50 Items Displayed Out of 183 Item(s) Found	Reason(s)	Au

Searching on the Retraction Watch Database using similar reasons for retraction yielded 183 items.

Consider the lost time and potential for ripple effects!

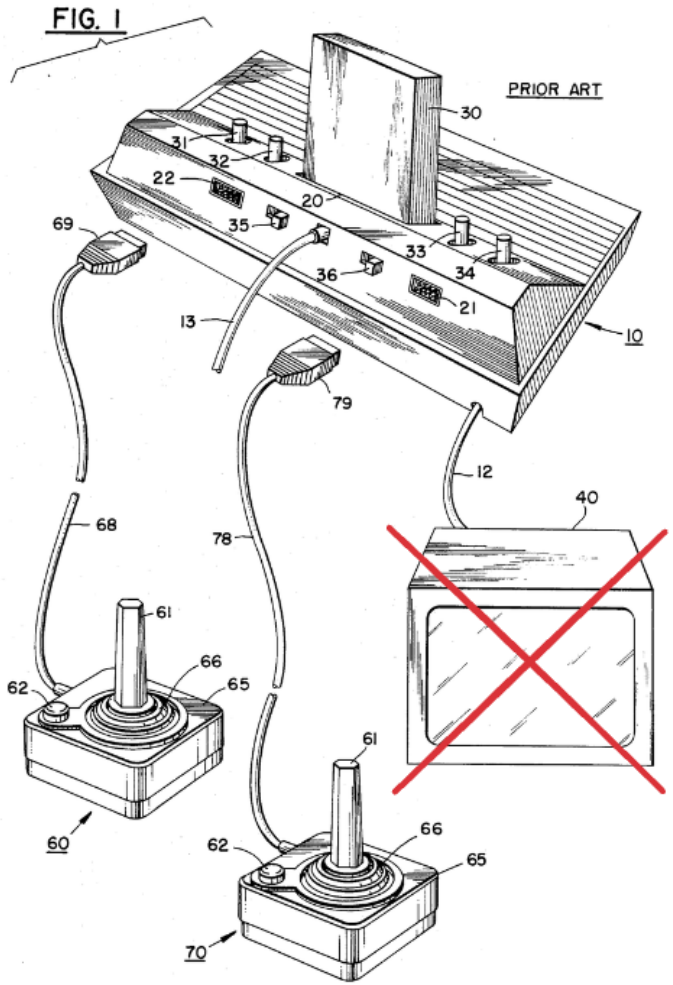
Lanterns for the dungeon



> "As we work to create light for others, we naturally light our own way." (Mary Anne Radmacher)

Image: Modified from the British Museum, Wikimedia

Lanterns for the dungeon



Creating software without tests can be like playing a video game with no screen.

Image: Modified from George C. Stone, Stuart E. Ross for Cbs Inc., Wikimedia

Lanterns for the dungeon

Software testing is a practice of ensuring software meets certain expectations.

We can create “lanterns” through tests to help make our software visible and prepare it for use by ourselves or others.

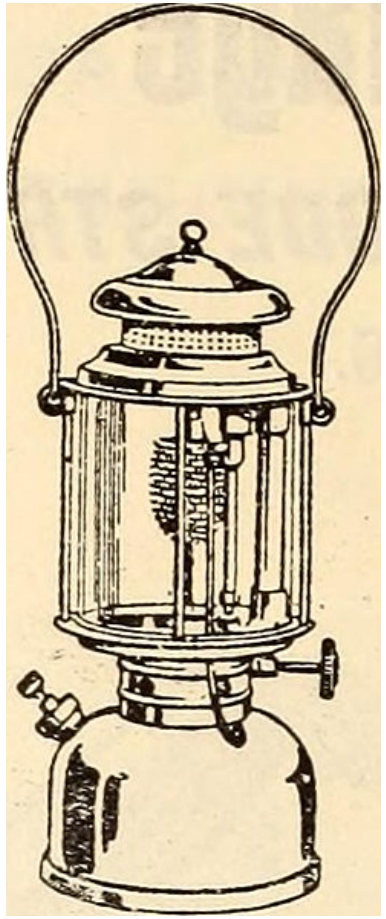
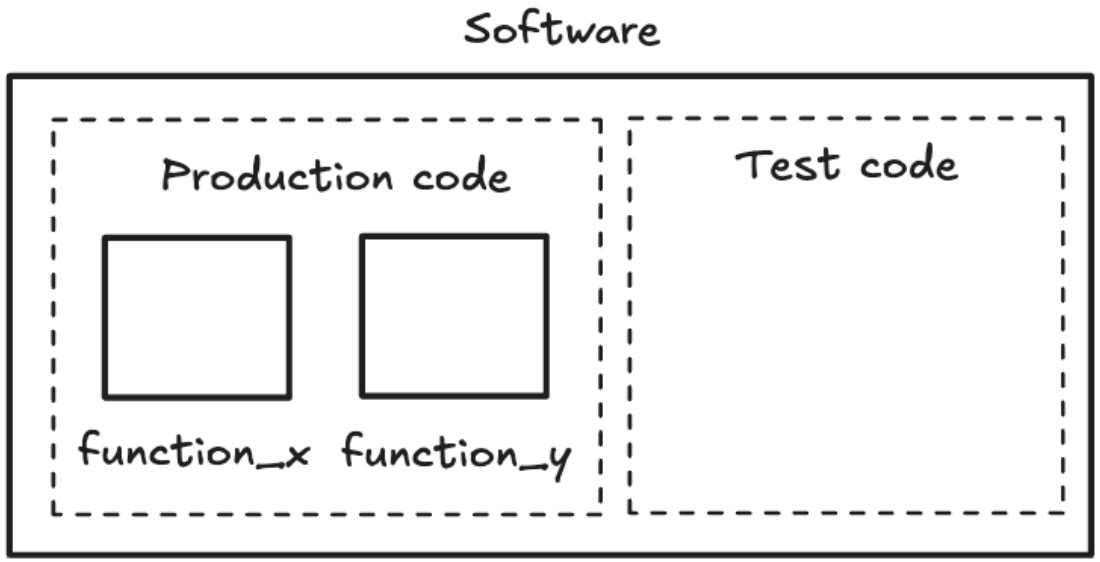


Image: Internet Archive Book Images, Wikimedia

Lanterns for the dungeon



Writing software tests involves distinguishing “**production**” from “**test**” code.

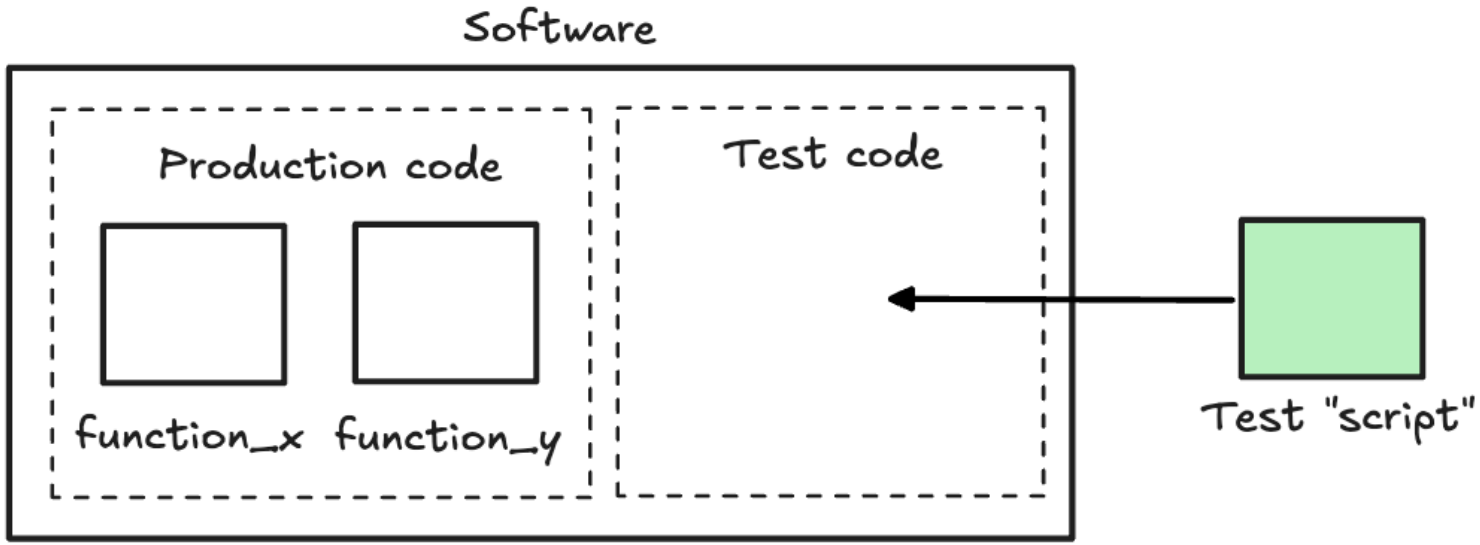
Sometimes we already have test code but it’s not implemented using a testing framework.

Lanterns for the dungeon

Programming languages often include built-in **testing frameworks** (for example, `unittest` in Python).

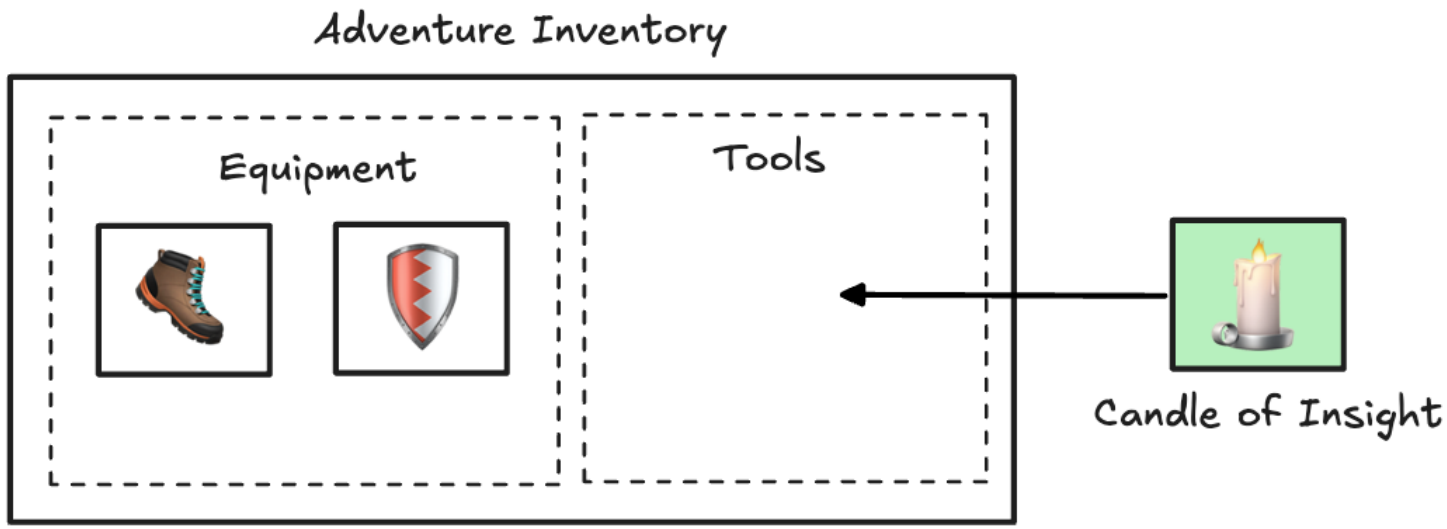
They also might include external packages which can help test in a specific way (for example, `pytest` in Python, or `testthat` in R).

Lanterns for the dungeon



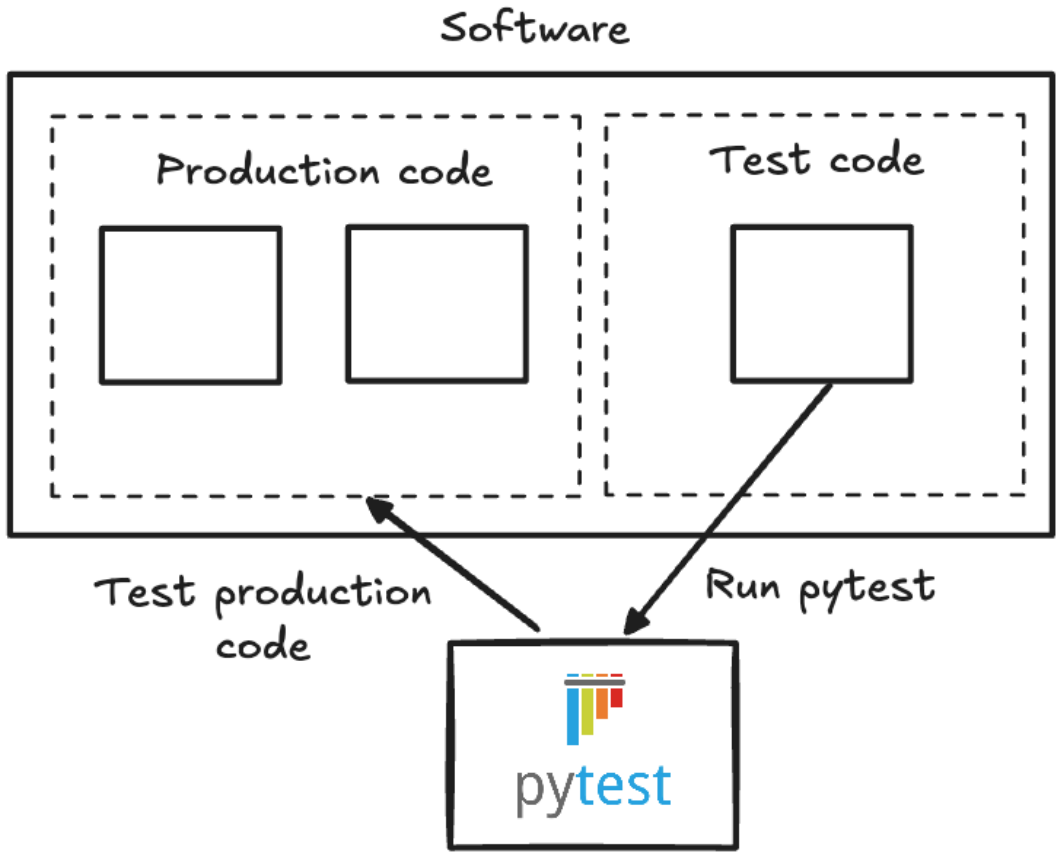
It’s important to create or move testing scripts to testing frameworks which are versioned with production code.

Lanterns for the dungeon



Think of this like you would moving useful tools into your inventory as part of your software adventure.

Lanterns for the dungeon



We'll use `pytest` for the following examples. It is installed as an external package (e.g. `pip install pytest`).

Lanterns for the dungeon

```
example_project
├── pyproject.toml
├── src
│   └── package_name
│       └── package_module.py
└── tests
    └── test_package_module.py
```

Python development often involves using the **`src` directory** for production code and the **`test` directory** for test code.

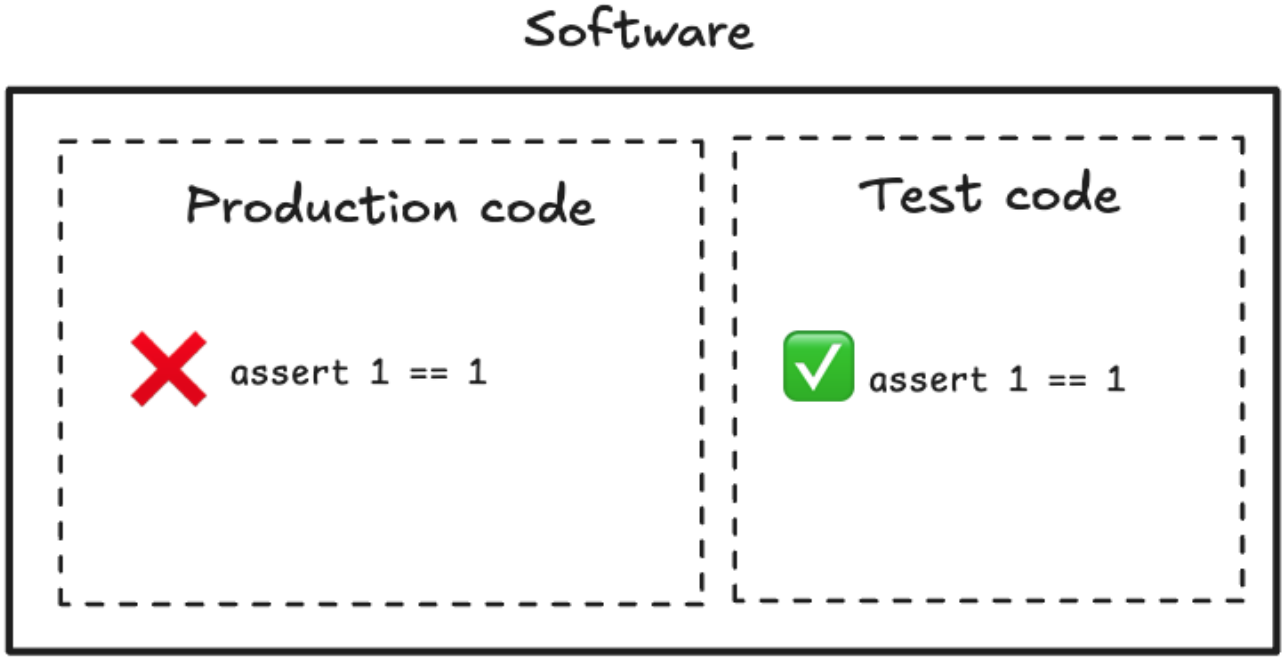
`pytest` by default looks for tests under the `tests` dir and for modules which contain a `test_` prefix.

Asserts

```
# we can use the assert statements to determine
# the truthiness (or lack thereof) of values.
assert 1 == 1
# returns True
assert 1 == 0
# returns False
```

`assert` statements are a common part of writing tests in Python. We can use assert to determine the truthiness of certain output.

Asserts



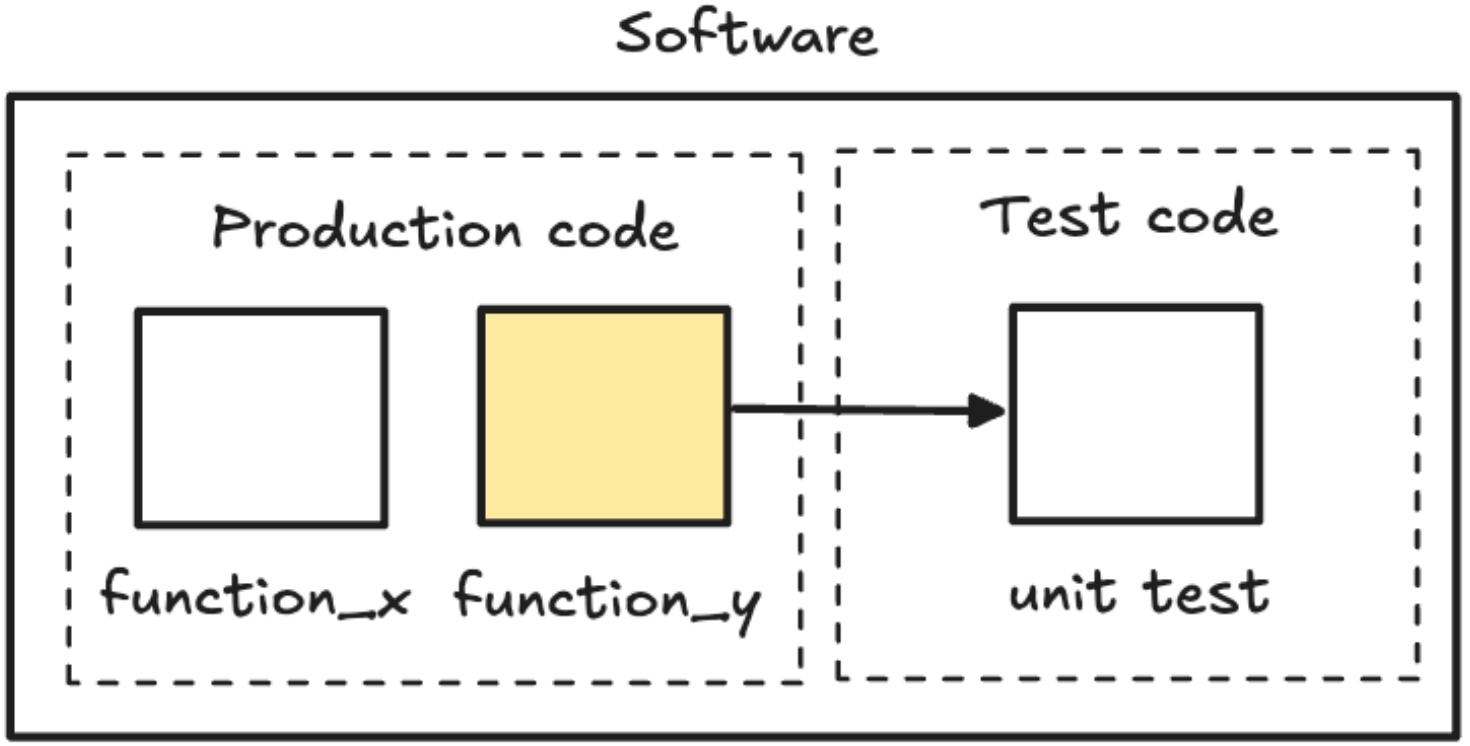
It's highly recommended to avoid using `assert` in production code. `assert` can be disabled during processing, for example, using `python -O ...`. This is another reason to keep our production and testing code distinct.

Unit, integration, and system tests

As we develop our test code it's useful to classify different kinds of tests. Three test types that might be useful to think about:

- **Unit tests**
- **Integration tests**
- **System / end-to-end tests**

Unit, integration, and system tests



Unit tests: validate small, isolated parts of your code, like individual functions or methods.

Unit, integration, and system tests

Production code

```
1  def is_even(number: int) -> bool:
2      # check that a number is even
3      return number % 2 == 0
```

Test code

```
1  def test_is_even():
2      # assert that 2 is an even number
3      assert is_even(2)
```

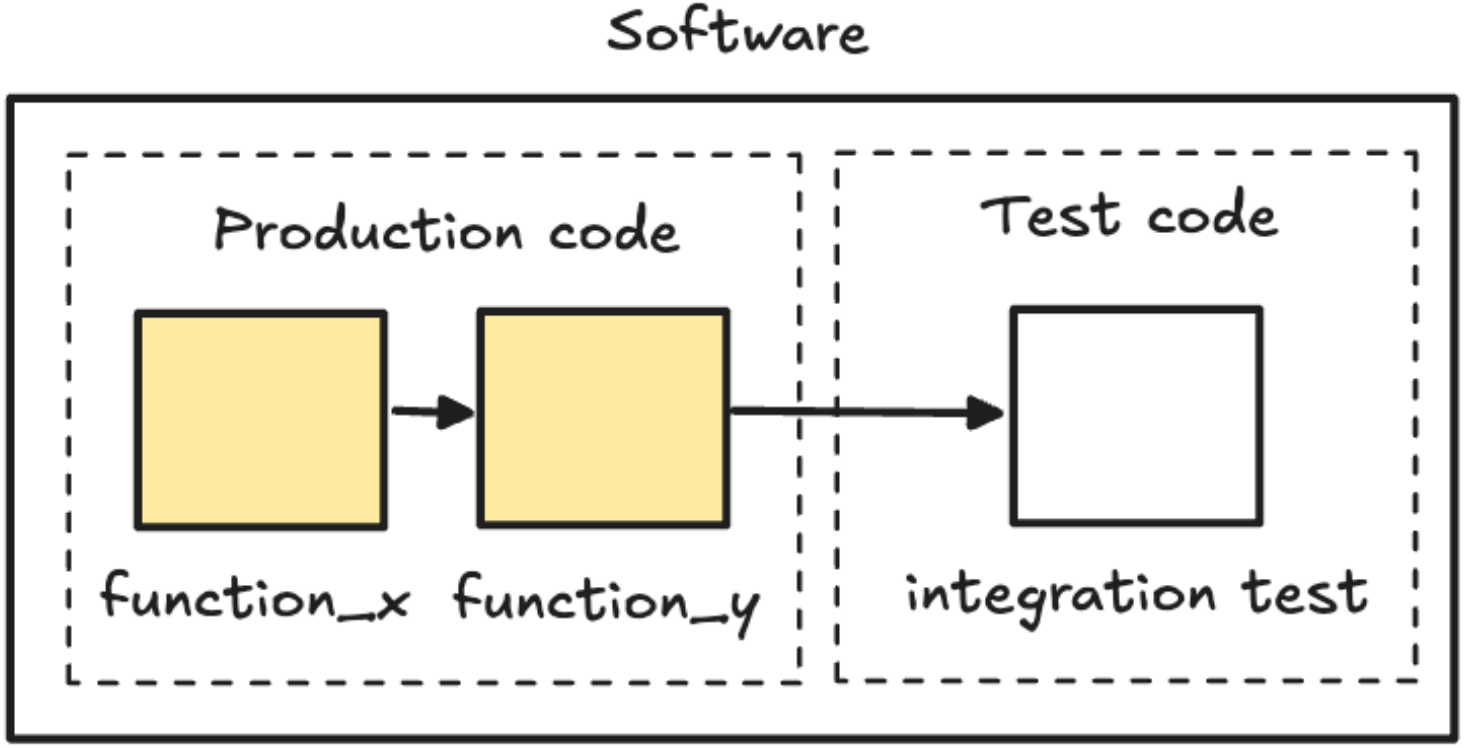
Assume we have production code function `is_even()`.
We can test that code using a similar function (`pytest` conventions).

Unit, integration, and system testsn

```
example_project
├── pyproject.toml
├── src
│   └── package_name
│       └── package_module.py # is_even() could go here
└── tests
    └── test_package_module.py # test_is_even() could go here
```

Reminder: these functions could be organized into the directory structure in their relevant locations.

Unit, integration, and system tests



Integration tests: integration tests ensure software components work together. This is especially important in scientific software where different models, algorithms, and data structures interact.

Unit, integration, and system tests

Production code

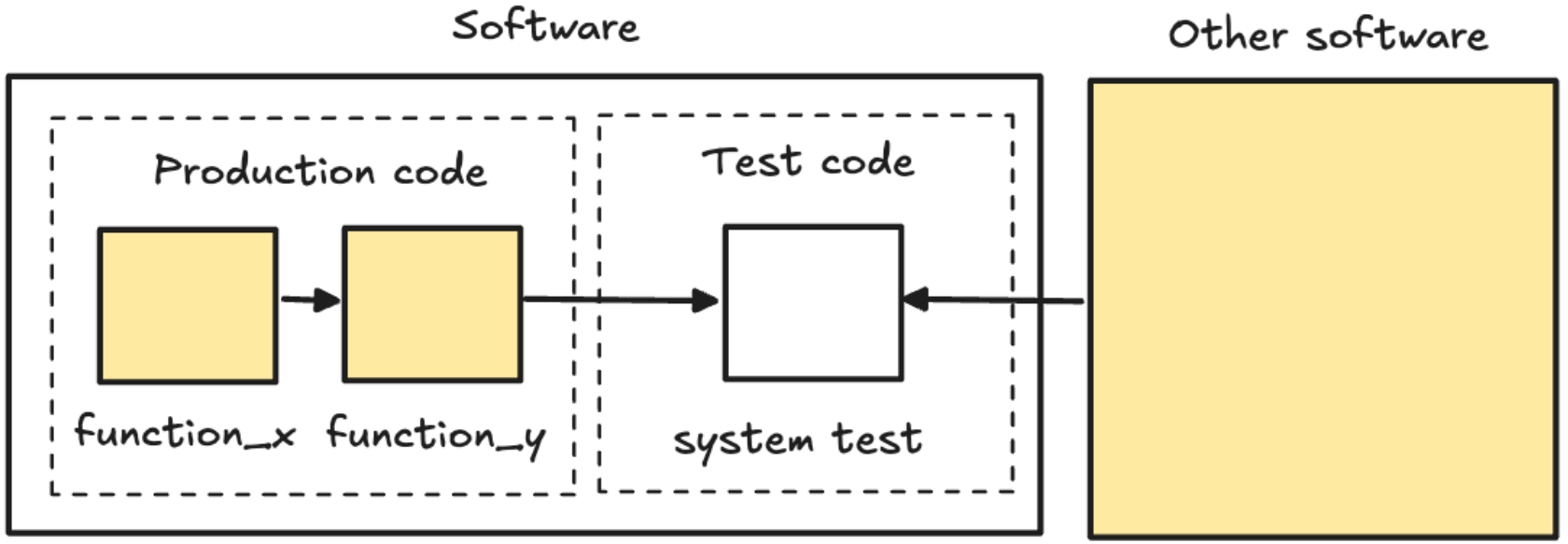
```
1  def is_even(number: int) -> bool:
2      # check that a number is even
3      return number % 2 == 0
4
5  def is_odd(number: int) -> bool:
6      # check that a number is odd
7      return not is_even(number=number)
```

Test code

```
1  def test_is_odd():
2      # assert there's no inconsistency
3      assert is_even(2) != is_odd(2)
4      # assert that 3 is an odd number
5      assert is_odd(3)
```

Assume we have another function `is_odd()` that leverages `is_even()`. We can build an integration test which involves the use of both functions.

Unit, integration, and system tests



System / end-to-end tests: These check the software from a user's perspective. For scientific software, this often means running entire workflows or simulations to make sure that everything from data input to output runs smoothly.

Unit, integration, and system tests

Production code

```
1  def is_even(number: int) -> bool:
2      # check that a number is even
3      return number % 2 == 0
4
5  def is_odd(number: int) -> bool:
6      # check that a number is odd
7      return not is_even(number=number)
```

Test code

```
1  import other_pkg
2
3  def test_is_even_with_external():
4      # assert we get the expected
5      # result when using is_even
6      # with other_pkg
7      assert other_pkg(is_even(2))
```

We could build a test with an external package as a system / end-to-end test. This might simulate how someone experiences implementing the work.

Unit, integration, and system tests

```
# run pytest by using it through a CLI
$ pytest

# a report like the following will display
============== test session starts ===============
platform darwin -- Python 3.11.9, pytest-8.3.3,
pluggy-1.5.0
rootdir: /example_project
configfile: pyproject.toml
collected 1 item

tests/test_package_module.py .                [100%]

=============== 1 passed in 0.00s ================
```

We can run pytest by using it as a CLI. It shows a report of passing or failing tests.

Unit, integration, and system tests

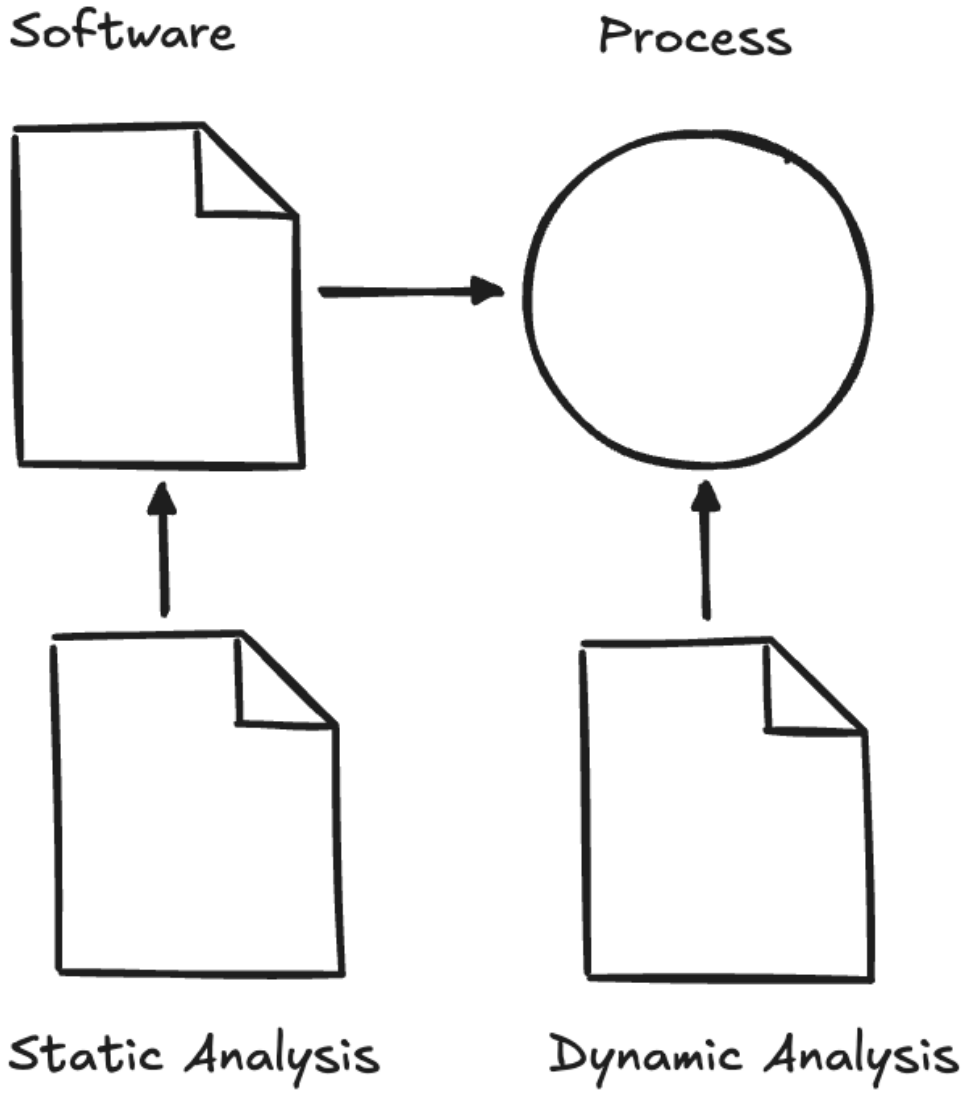
“This looks complicated. Is there a faster way to add tests?”

Tests like the ones mentioned earlier are a best practice and highly recommended. Just the same, there are other tools which can help address different aspects of your software (relatively quickly).

Static and dynamic analysis

So far, we've talked about **dynamic analysis** through tests (how software acts when it is processed).

We can also use various **static analysis** (how the software alone looks without being processed).



Static and dynamic analysis

Static analysis tests are akin to enhancement items or “buffs” in video games. They are great learning tools and provide a chance to improve software with best practices.

Linters

Linters are **static analysis** tools to help address specific practices **before software is processed**. They can be a great way to ensure minimum levels of code quality without writing your own tests.

Image: Isarra, Wikimedia

Linters

`ruff` includes a collection of **linters** and **formatters** (for consistent formatting of your code) for Python which can be added to perform **automatic tests on your software**.

It builds upon existing work from projects like `pylint`, `pyflakes`, `flake8`, and others.

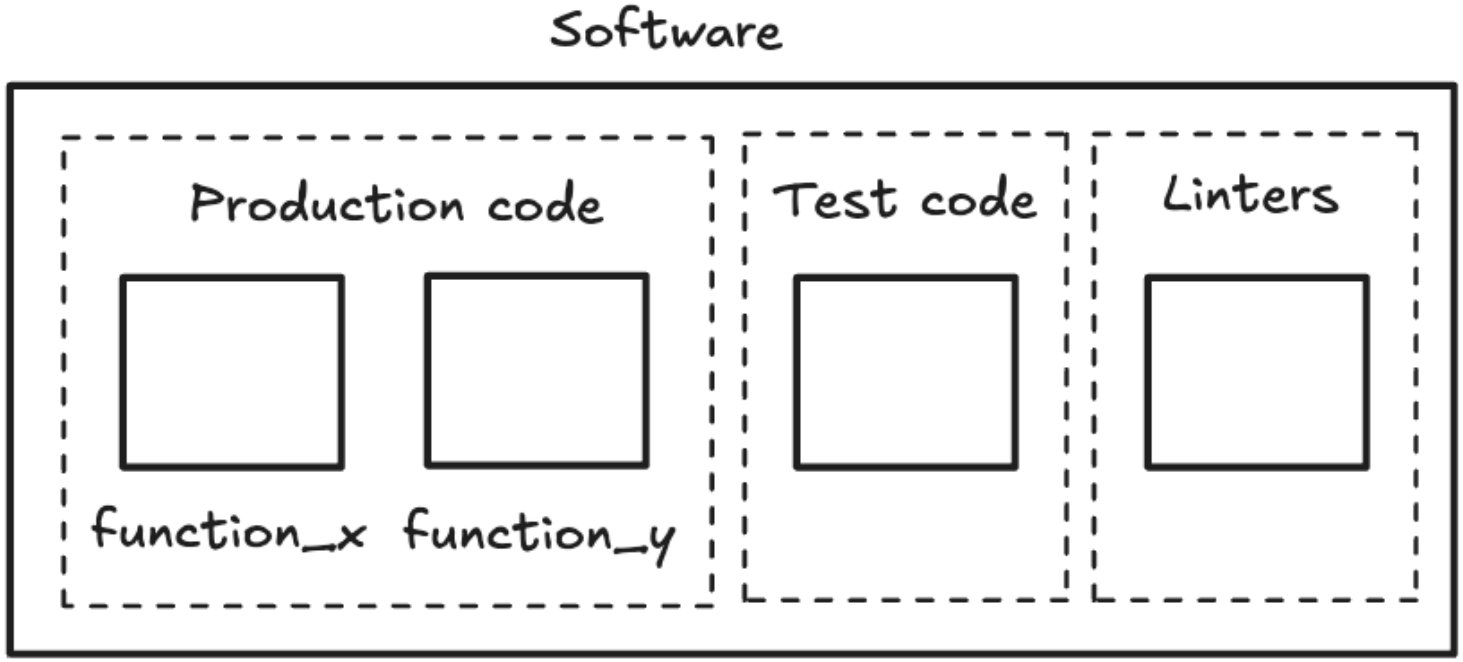
Linters

```
# referenced from: https://docs.astral.sh/ruff/linter/
$ ruff check                  # Lint all files in the current directory.
$ ruff check --fix            # Lint all files and fix any fixable errors.
$ ruff check --watch          # Lint all files and re-lint on change.
$ ruff check path/to/code/    # Lint all files in `path/to/code`.
```

`ruff` can be `pip` installed (e.g. `pip install ruff`) and used from the command line to check, fix, or watch your software files.

Note: `fix` will attempt to automatically fix certain failing checks where possible.

Linters



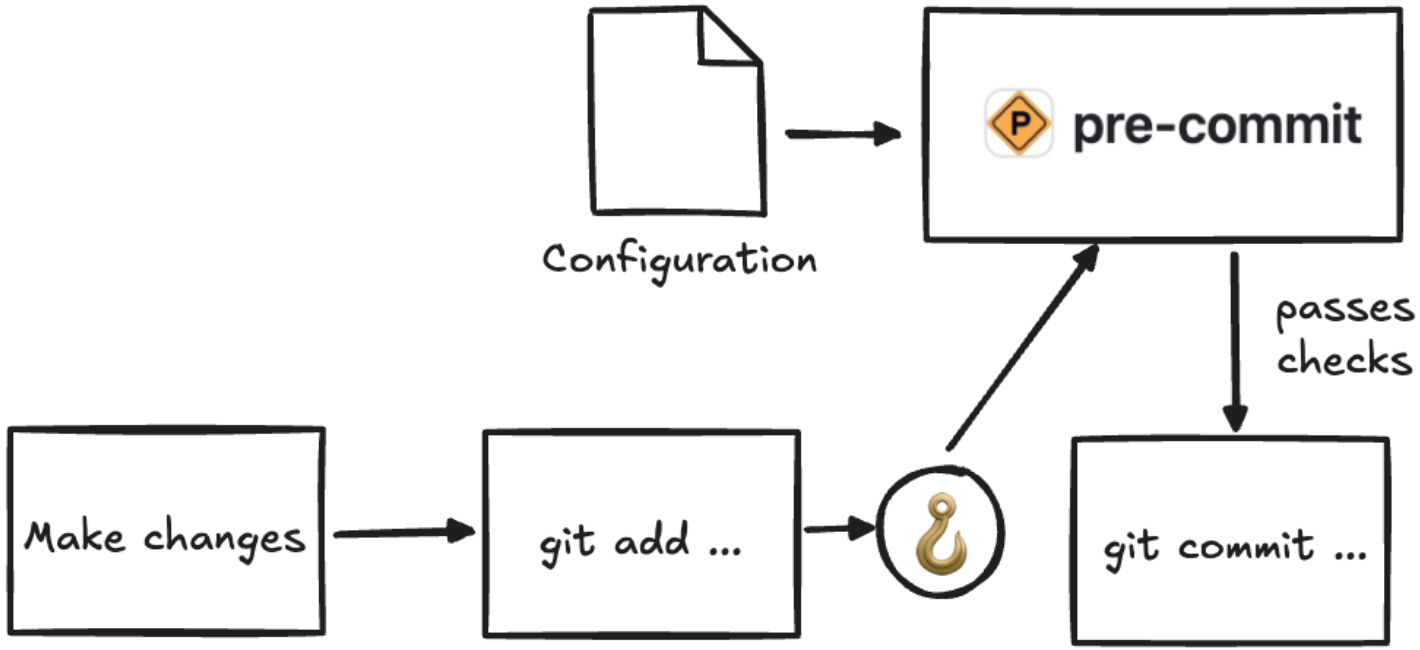
Linters often get organized outside of the formal tests.

Linters

```
example_project
├── .pre-commit-config.yaml # linters can be defined here
├── pyproject.toml
├── src
│   └── package_name
│       └── package_module.py
└── tests
    └── test_package_module.py
```

Linters can be defined using something called `pre-commit`.

Linters



`pre-commit` is designed to be used as a git hook. It can also be used as a standalone
(for ex. `pre-commit run check-name`).

Linters

```yaml
# See https://pre-commit.com for more information
# See https://pre-commit.com/hooks.html for more hooks
default_language_version:
  python: python3.11
repos:
  - repo: https://github.com/astral-sh/ruff-pre-commit
    rev: "v0.6.8"
    hooks:
      - id: ruff
```

This is an example `pre-commit-config.yaml` which will add `ruff` as a check.

Linters

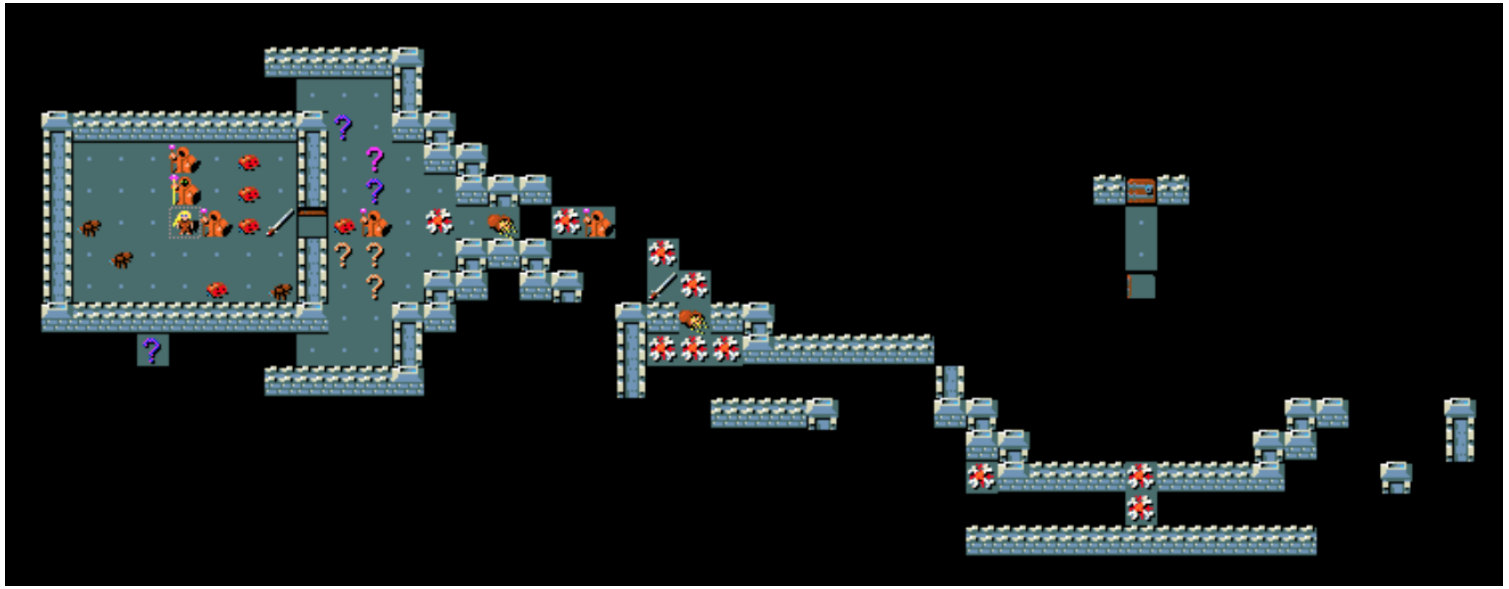
```
# we can run pre-commit manually or as a git hook
$ pre-commit run

# a report like the following will display
ruff.....................................................................Fa
- hook id: ruff
- exit code: 1

src/package_name/module.py:5:8: F401 [*] `os` imported but unused
  |
3 | """
4 |
5 | import os
  |        ^^ F401
6 |
7 | def is_even(number: int) -> bool:
  |
  = help: Remove unused import: `os`
```

We can run `pre-commit` through the CLI. It will show a report of passing or failing checks like the above.

Understanding the shape of the labyrinth



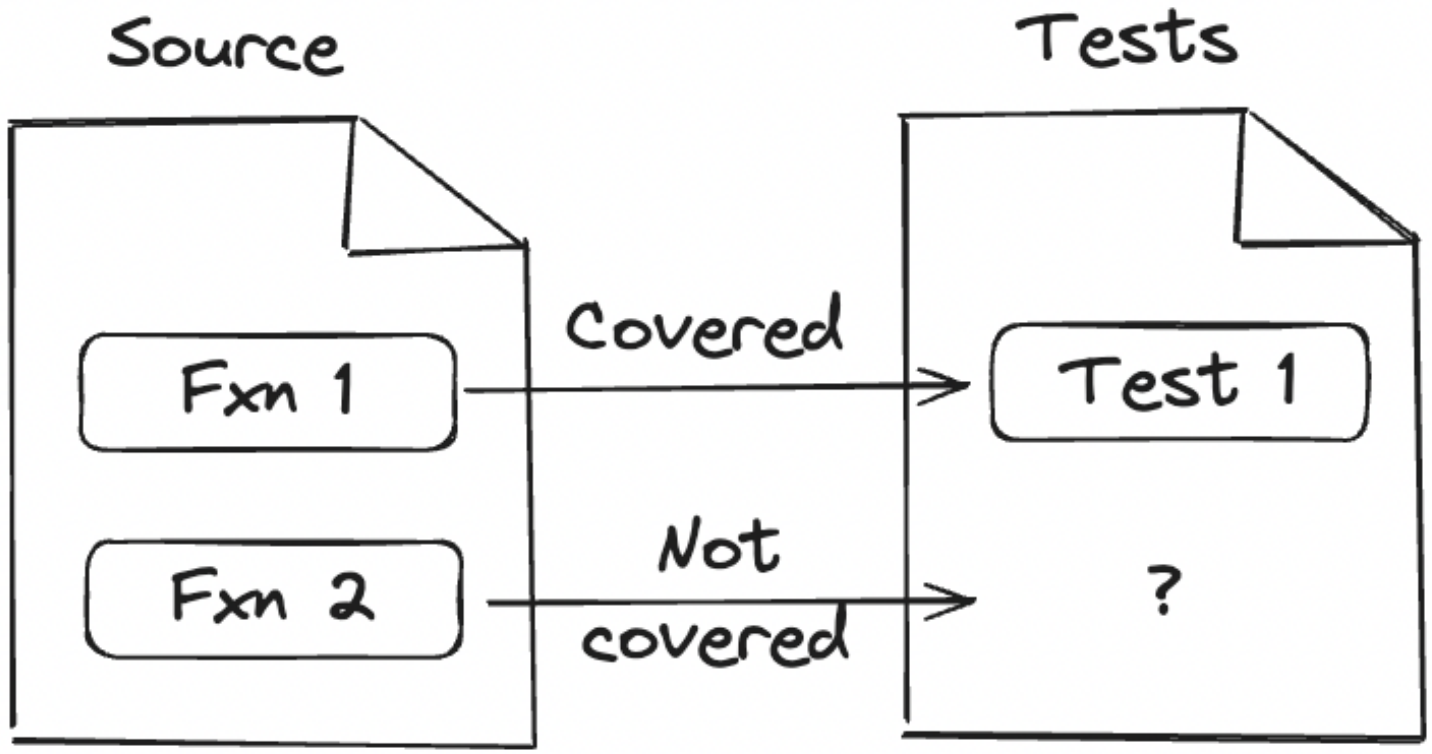
Okay great, so we've started to illuminate the labyrinth!

We start to discover there are more bugs (and edge cases) than we thought. 😱

How can we understand just how big the labyrinth might be?

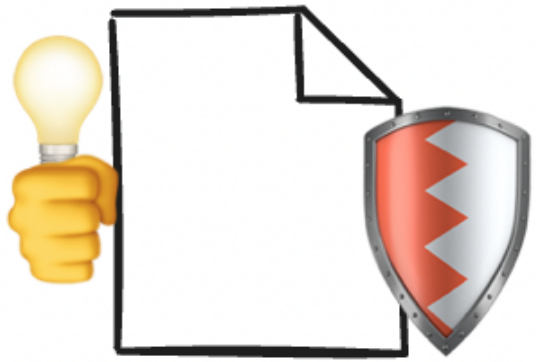
Image: modified from Mielas, Wikimedia

Understanding the shape of the labyrinth



Code coverage is a measure of how much of your production code is executed as part of your test code.

Understanding the shape of the labyrinth



- **Line coverage:** tracks the execution of lines of code
- **Statement coverage:** is similar but considers whether each statement (regardless of the number of lines).
- **Function coverage:** checks whether each function
- **Branch coverage:** helps verify whether every possible path through control structures (like if-else blocks).

Understanding the shape of the labyrinth

```
# example line
print("A line which could be tested for coverage.")

# example statement (potentially multi-line)
statement = [
    1,
    2,
    3,
]

# example function
def example():
    return "An example function."

# example branching
if 1 in statement:
    return True
else:
    return False
```

The above are examples of each type of coverage.

Understanding the shape of the labyrinth

`coverage.py` is a powerful tool for measuring code coverage in Python. It can be installed into your environment using, for example, `pip install coverage`.

Afterwards, you can run your tests with coverage, and generate reports in various formats.

Understanding the shape of the labyrinth

Production code

```
1 # module.py
2
3 def covered_test():
4     return "This is covered."
5
6 def uncovered_test():
7     return "No coverage here."
```

Test code

```
1 # test_module.py
2
3 from module import covered_test
4
5 def test_one_function():
6     result = covered_test()
7     assert result == "This is covered."
```

Consider the following example production and test code.
One function is tested and one is not.

Understanding the shape of the labyrinth

```
# first we process test coverage
$ coverage run -m pytest

# then we show the reported output of
# processed test coverage
$ coverage report
Name             Stmts   Miss  Cover
----------------------------------
module.py            4      1    75%
test_module.py       3      0   100%
----------------------------------
TOTAL                7      1    86%
```

We can run `coverage.py` on the code by using the CLI. Afterwards, we create a report (note there are many different options, including HTML).

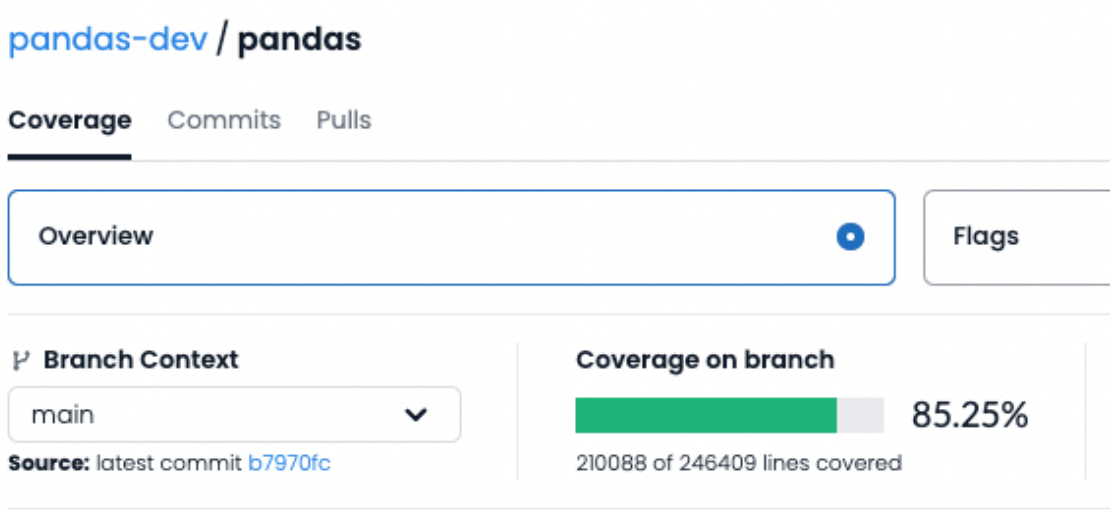
Note: `coverage.py` implements **statement coverage** checking by default.

Understanding the shape of the labyrinth

In addition to the `coverage.py` there are also code coverage platforms (often through software as a service (SaaS) companies).

- Codecov focuses on code coverage.
- Codacy includes code coverage in addition to other features.
- Code Climate includes code coverage in addition to other features.

Understanding the shape of the labyrinth

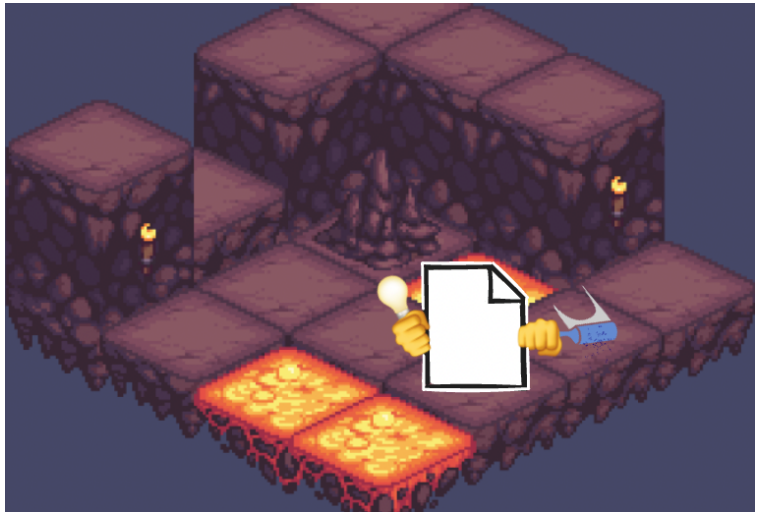


Just like a map of the labyrinth, we don't have to have 100% coverage to know the shape of the code.

- Generally 80-90% code coverage is considered pretty good!

Image: Codecov, Pandas

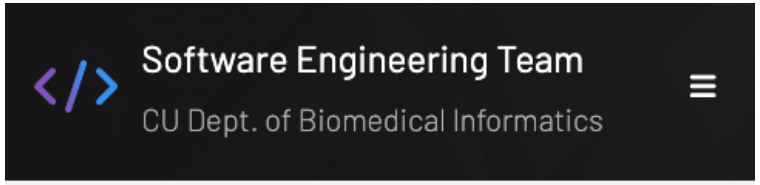
Additional thoughts



- Code coverage tools often don’t include linting.
- We can think about linting coverage similarly.
- Strive to provide coverage with both tests and linting!

Image: modified from fellfeline, DeviantArt

Check out the DBMI SET blog for more...



- GitHub Actions: find out how to automate your testing!
- Linting: a deeper dive into linting.
- Code Coverage: more on code coverage.
- Testing: additional content on testing.

Thank you!

Thank you for attending! Questions / comments?

Please don’t hesitate to reach out!

- </> CU Anschutz DBMI SET Team
- @d33bs