

Testing

The big picture

- **A well-designed function, well-tested function**
 - code you can *trust* — it has been tested
- **Automated tests do the checking for you**
 - once written, can run automatically
 - no repeated manual effort
- **Testing turns expectations into evidence**
 - expected behaviour is checked against how the code actually works, every time, automatically
- **Confidence to change code without fear**
 - a good test suite is a safety net for every future code edits
- **The ‘nightly build’**
 - automation at scale: every test run every night
 - lets teams work on the same code without breaking it
 - ... or if it does break it is detected by tests



Overview

- **Test logic**
 - expected vs. actual behaviour
- **Assertions**
 - testing checks a claim/prediction
- **The trouble with manual testing**
 - from manual checks to `assert`
- **Unit testing with `pytest`**
 - proper testing



Test logic

Testing logic

- **Expected behaviour**

- what you believe correct output should be
- prediction, not something the computer tells you

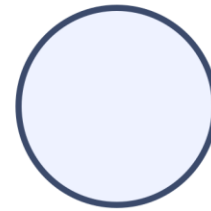
- **Actual behaviour**

- what the function returns *actually*
- observed result produced by computer
- independent of prediction

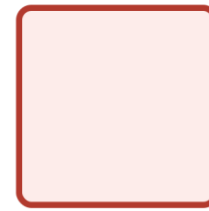
- **Comparison**

- when expected and actual don't match
Test Failed!
- when expected and actual match
Test Passed !

Expected



Actual



Expected



Actual



Assertions

- **A statement put forward as true**
 - an assertion is either true or false.
- **Different from proof**
 - assert a claim confidently
 - ...but still might be wrong
- **The assertion is the claim itself**
 - not the evidence



Assertion 1: *this is not a pizza!*

Assertion 2: *this is a pizza!*

Example: 'The doorbell rings'



Problem with manual tests

Manual inputs

- **Wastes time**

- tedious, repetitive, inconsistent process
- doesn't scale ...
- ... every time the code changes, all that time wasted is wasted all over again

- **Poor coverage**

- can cause programmers to miss edge cases such as zero or negative numbers
- especially once there are too many *combinations* of inputs.

```
1  def divide(a, b):
2      return a / b
3
4
5  def main():
6      a = int(input("Enter a: "))
7      b = int(input("Enter b: "))
8      print(divide(a, b))
9
a
```

```
1  def divide(a, b):
2      return a / b
3
4
5  def main():
6      while True:
7          a_input = input("Enter a (or 'q' to quit): ")
8          if a_input == "q":
9              break
10
11         a = int(a_input)
12         b = int(input("Enter b: "))
13         print(divide(a, b))
14
"
```

Automating testing

- **Big step up from manual testing**

- comparison and the pass/fail verdict happen automatically every time you run the file
- Simple report instead of you eyeballing each result yourself.

- **Each test repeats the same shape:**

- set up inputs, call the function, compare expected to actual, print pass or fail
- scaffolding identical

```
1 def sum_with_limit(numbers, limit):
2     """
3     Sum a list of numbers in a for loop.
4     If the sum exceeds 'limit', return an empty list.
5     Otherwise, return the total sum.
6     """
7     total = 0
8     for n in numbers:
9         total += n
10        if total > limit:
11            return [] # return empty list if limit exceeded
12    return total
13
14 def main():
15     # Test 1: well under the limit
16     numbs = [1, 2, 3]
17     lim = 10
18     expected = 6
19     actual = sum_with_limit(numbs, lim)
20     if expected == actual:
21         print(f"PASS: expected {expected}, got {actual}")
22     else:
23         print(f"FAIL: expected {expected}, got {actual}")
24
25     # Test 2: well over the limit
```

Automated testing: `assert`

- **Comparison & pass/fail judgment**
- **Handled by `assert`**
 - does pass/fail judgement automatically
- **Similar (but better) functionality**
 - assertion hands work to the computer

```
# Test 1: well under the limit
nums = [1, 2, 3]
lim = 10
expected = 6
actual = sum_with_limit(nums, lim)
if expected == actual:
    print(f"PASS: expected {expected}, got {actual}")
else:
    print(f"FAIL: expected {expected}, got {actual}")

# Test 2: well over the limit
nums = [5, 6, 7]
lim = 10
expected = []
actual = sum_with_limit(nums, lim)
if expected == actual:
    print(f"PASS: expected {expected}, got {actual}")
else:
    print(f"FAIL: expected {expected}, got {actual}")
```

too
many
lines
Of
source
code
increases
risk
of
error
creeping in

```
# Test 1: well under the limit
assert sum_with_limit([1, 2, 3], 10) == 6

# Test 2: well over the limit
assert sum_with_limit([5, 6, 7], 10) == []
```

2 lines
simple

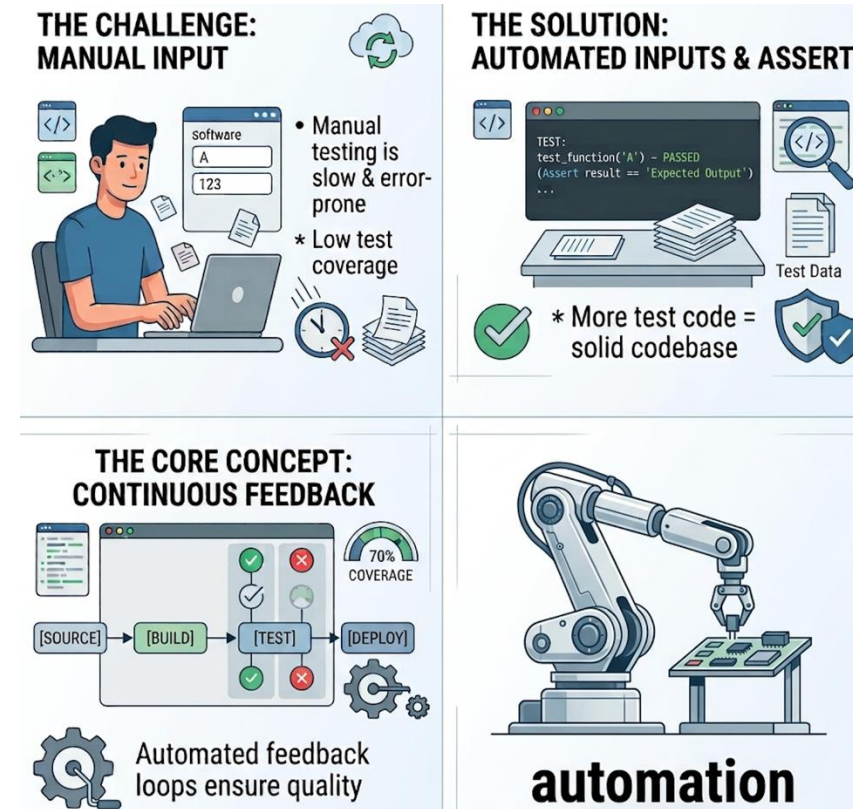
Assertions in programs

- **Doesn't branch like a normal condition**
 - reacts when assertion = false
- **If the claim is false, that's serious**
 - program stops: `AssertionError`
- **Failure **halts** program**
 - points to **exact line**
- **Otherwise**
 - All tests passed!

```
1 def sum_with_limit(numbers, limit):
2     """
3     Sum a list of numbers in a for loop.
4     If the sum exceeds 'limit', return an empty list.
5     Otherwise, return the total sum.
6     """
7     total = 0
8     for n in numbers:
9         total += n
10        if total > limit:
11            return [] # return empty list if limit exceeded
12    return total
13
14
15 def test_sum_with_limit():
16     # Test 1: well under the limit
17     assert sum_with_limit([1, 2, 3], 10) == 6
18
19     # Test 2: well over the limit
20     assert sum_with_limit([5, 6, 7], 10) == []
21
22     # Test 3: boundary case
23     assert sum_with_limit([4, 6], 10) == 10
24
25     print("All tests passed!")
26
```

DevOps

- **Aims to shorten the development lifecycle**
 - test automation makes that possible by removing human delays from the verification process
- **Enabling Continuous Integration (CI)**
 - automatically test every code change the moment it is pushed
- **Automated testing rapid**
 - actionable feedback to developers.
 - confirming if new code breaks existing **now**
- **Shift-Left Culture**
 - move automated tests into early development stages
- **Quality**
 - verified continuously



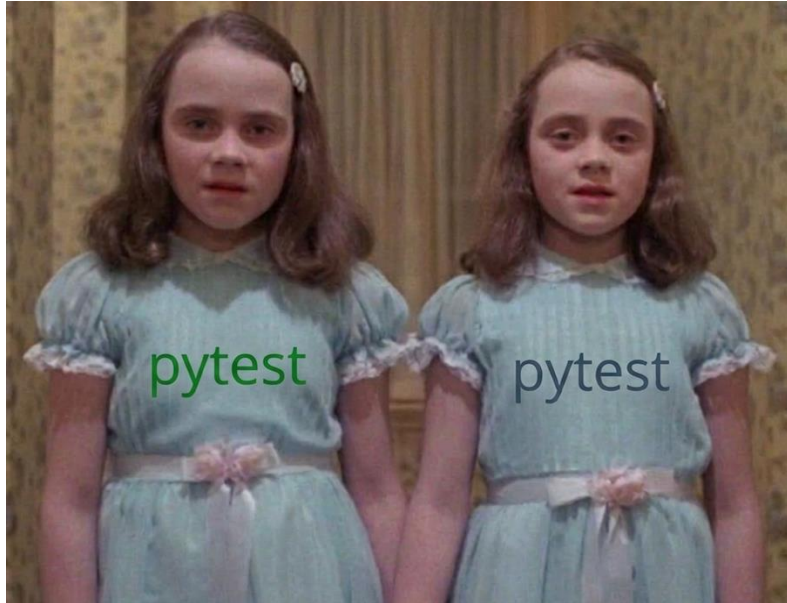
Unit Testing

Install `pytest` first!

```
[rholden@Richards-MacBook-Pro code % pip3 install pytest
Collecting pytest
  Downloading pytest-9.1.1-py3-none-any.whl.metadata (7.6 kB)
Collecting iniconfig>=1.0.1 (from pytest)
  Downloading iniconfig-2.3.0-py3-none-any.whl.metadata (2.5 kB)
Collecting packaging>=22 (from pytest)
  Using cached packaging-26.3-py3-none-any.whl.metadata (3.5 kB)
Collecting pluggy<2,>=1.5 (from pytest)
  Downloading pluggy-1.6.0-py3-none-any.whl.metadata (4.8 kB)
Collecting pygments>=2.7.2 (from pytest)
  Downloading pygments-2.21.0-py3-none-any.whl.metadata (2.5 kB)
Downloading pytest-9.1.1-py3-none-any.whl (386 kB)
Downloading pluggy-1.6.0-py3-none-any.whl (20 kB)
Downloading iniconfig-2.3.0-py3-none-any.whl (7.5 kB)
Using cached packaging-26.3-py3-none-any.whl (129 kB)
Downloading pygments-2.21.0-py3-none-any.whl (1.3 MB)
_____ 1.3/1.3 MB 20.9 MB/s 0:00:00
Installing collected packages: pygments, pluggy, packaging, iniconfig, pytest
Successfully installed iniconfig-2.3.0 packaging-26.3 pluggy-1.6.0 pygments-2.21.0 pytest-9.1.1
```

pytest is really *two* things

- pytest dualism



*"I am a Terminal
Command to run
tests with"*

*"I am a Python
Library to write
Python tests with"*

- Two pytests to remember



"TWO!! 'pytest' AND 'pytest' HUH!!"
"FINE!!"
"MOVE ON TO THE NEXT SLIDE!!!"

The `pytest` library

- **Third-party library to automate the entire testing process**
 - runs every test function & reports results
- **Test file**
 - collection of functions
 - name prefixed with `test_`
 - each test: one (or more) `assert`
 - imported code targeted
- **Just tests, no:**
 - `main()`
 - `if __name__ == "__main__":`
 - ... this should all be in the target file

```
1  from hello import hello
2
3
4  def test_default():
5      ... assert hello() == "hello, world"
6
7
8  def test_argument():
9      ... assert hello("hamish") == "hello, hamish"
10
11
12 def test_loop():
13     ... for name in ["isla", "callum", "morag"]:
14         ... assert hello(name) == f"hello, {name}"
15
16
17 def test_argument_again():
18     ... # Deliberate fail
19     ... assert hello("hamish") == "hello, richard"
--
```

Interpreting `pytest` output

- Green dot ●

- test passed

- Red letter F

- test failed

- prints detailed traceback

- exactly which `assert` failed

- the **expected** vs. **actual** result

```
● $ pytest test_hello.py
===== test session starts =====
platform darwin -- Python 3.14.7, pytest-9.1.1, pluggy-1.6.0
rootdir: /Users/rholden/Downloads/code
collected 3 items

test_hello.py ... [100%]

===== 3 passed in 0.00s =====
○ $
```

```
○ $ pytest test_hello.py
===== test session starts =====
platform darwin -- Python 3.14.7, pytest-9.1.1, pluggy-1.6.0
rootdir: /Users/rholden/Downloads/code
collected 4 items

test_hello.py ...F [100%]

===== FAILURES =====
_____ test_argument_again _____

    def test_argument_again():
        # Deliberate fail
        > assert hello("hamish") == "hello, richard"
E       AssertionError: assert 'hello, hamish' == 'hello, richard'
E
E       - hello, richard
E       + hello, hamish

test_hello.py:19: AssertionError

===== short test summary info =====
FAILED test_hello.py::test_argument_again - AssertionError: assert 'hello, hamish' == 'hello, richard'
===== 1 failed, 3 passed in 0.02s =====
```

Good practice

- **Several small tests e.g.**
 - `test_positive`, `test_negative`,
 - `test_zero`, `test_identity`,
 - `test_str`
 - ...not all in one test function
- **Descriptive, verbose naming**
 - good!
 - quicker recognition of problems
- **pytest**
 - runs every test function independently
 - if one fails, the others still run
 - no 'bail' on assert
 - Compare this to a loop-based test
- **Information rich**
 - learn more about what is broken
 - more efficient

```
1 import pytest
2 from demo_maths import pow
3
4
5 def test_positive():
6     for i in range(1, 10):
7         assert pow(i, 2) == i ** 2
8
9
10 def test_negative():
11     for i in range(-10, 0):
12         assert pow(i, 2) == i ** 2
13
14
15 def test_zero():
16     assert pow(0, 2) == 0
17
18
19 def test_identity():
20     assert pow(1000, 0) == 1
21     assert pow(2, 0) == 1
22
23
24 def test_str():
25     with pytest.raises(TypeError):
26         pow("nothing", 2)
```

Exceptions testing

- **Some inputs shouldn't just fail quietly**
 - verifying that a function correctly raises an error for incorrect input types
- **pytest.raises**
 - checks that expected exception is raised
- **Example**
 - `pow("nothing", 2)` raises a `TypeError`
 - rather than assuming error break your code

```
1  import pytest
2  from demo_maths import pow
3
4
5  > def test_positive(): ...
6
7
8
9
10 > def test_negative(): ...
11
12
13
14
15 > def test_zero(): ...
16
17
18
19 > def test_identity(): ...
20
21
22
23
24 def test_str():
25     ... with pytest.raises(TypeError):
26         ... pow("nothing", 2)
27
```

Run numerous test files

- Type `pytest` in working directory
 - Run all `test_` files in there and get feedback
- **Dedicated test directory**
 - separate to product code in large project
 - An empty `__init__.py` file tells Python to treat that folder as a package
 - Running `pytest` on the whole folder finds and runs every test file automatically
 - no need to name each test file individually

```
project/
├── demo_maths.py
├── hello.py
└── tests/
    ├── __init__.py
    ├── test_demo_maths.py
    └── test_hello.py

$ pytest tests/
```

Conclusions

- **Keeping tests small and simple**
 - so you do not have to write "tests for your tests".
- **Testing replaces assumptions with evidence**
 - comparing expected vs. actual turns a guess about your code into a checked fact
- **assert removes the manual work of checking**
 - one line does all the work. Lovely!
- **pytest automates discovery, execution, and reporting**
 - finds every `test_` function, runs it, reports **pass**/**fail**. Simple!
- **Well-designed tests**
 - small, descriptive, independent test functions
 - isolate failures for ease of identification
 - richer diagnostics
- **Scaleability**
 - from a single assert to testing every change automatically, with no one watching
 - Continuous Integration pipeline / 'DevOps'

Thanks!