




unittesting



Jason J. Watson
Zeuthen Data Science Seminar
April 16, 2024

Contents

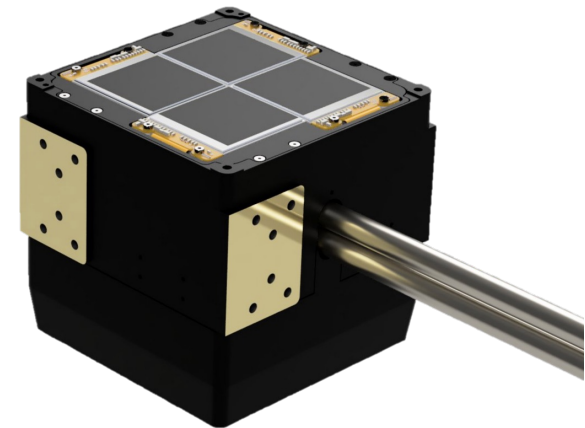
- What is unit testing? Why use it?
- `pytest` 
- Advanced approaches and features

Credits to Maximilian Nöthe

<https://indico.in2p3.fr/event/20306/contributions/96818/>

Who am I?

- Postdoc in the Gamma group (since Jan 2020) – background in astrophysics
- Software development with Python and C++
- Strong advocate for unit testing
- Current projects: writing slow control & image readout software for two cameras



Testing Software

- Anyone who writes software almost certainly tests their software
- The simplest possible test: manually running your script
- But this is error prone:
 - We can forget
 - Can often miss bugs (especially in edge cases)
 - Very tedious and time consuming
- Better solution: **automated testing**



Automated Testing

- Automated tests are developed as part of the codebase, and can be (re-)executed at many stages of the project development
- Tests for software can be divided into three categories:
 - **Unit test:** Ensure that the smallest components of the codebase behave as expected
 - **Integration test:** Ensure that the interaction between components (internal and external) behave as expected
 - **Performance test:** Ensure that the component meets requirements (e.g. speed)
- The lines between these three can often be blurred – what you call your tests is less important than writing them in the first place!

Simple Unit Test Example

```
1 def add(x: float, y: float) -> float:
2     return x + y
3
4
5 assert add(1, 1) == 2
6 assert add(-1, 1) == 0
7 assert add(2, 6) == 8
```

- This very simple example demonstrates the key components to a unit test:
 - We have isolated our smallest testable component: the `add` method
 - We have defined our test inputs
 - We have passed the input into our unit
 - We have `assert`ed the expected outputs
- This form of unit-testing is known as “example-based testing”
 - We will later explore its twin: “property-based testing”

Quick Note on Type Hinting

```
1 def add(x: float, y: float) -> float:
2     return x + y
3
4
5 assert add(1, 1) == 2
6 assert add(-1, 1) == 0
7 assert add(2, 6) == 8
```

- In the simple example, I use **type hinting**
- This is a feature of the Python standard library
- It helps to document the expected types of variables, arguments and return values
- Like unit tests, they are a good practice which catch bugs early (through IDE warnings)
- Pair nicely with unit tests, as they further encourage you to define the inputs and outputs of a unit
- But they are only **hints**, the types are not enforced at runtime... unless you use [mypy](#)

Why unit test?

- **Trust**
 - Unit tests are a guarantee to your users that the product works
 - Unit tests are a reassurance to yourself that the product works
- **Design**
 - Encourages good software practices, such as splitting our code into smaller, more-understandable parts
 - Forces us to more carefully consider a unit's purpose, its inputs, outputs, and its error conditions
- **Maintainability**
 - Aided by unit tests, we can “refactor with confidence”
 - We have a quick way to check compatibility with new language or dependency versions
- **Documentation**
 - Tests can serve as additional documentation for your project, providing usage examples

**Writing unit tests are a worthwhile investment.
They will save us more time than the time spent writing them.**

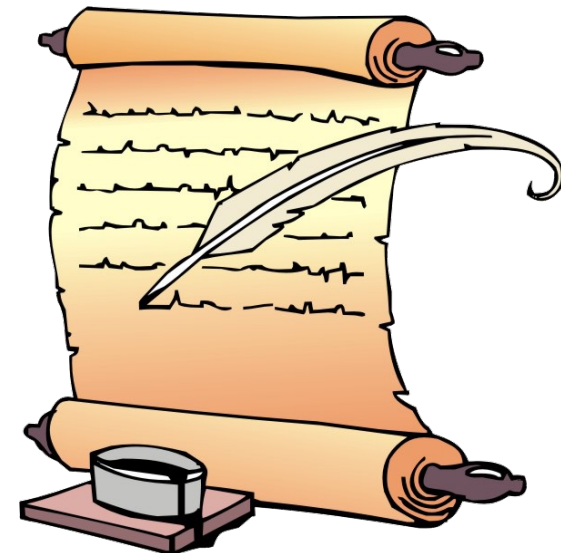
F.I.R.S.T Principles of Unit Testing

- **Fast:** should be quick to run, so you aren't discouraged from running them often
- **Independent:** should not depend on the state of the previous test
- **Repeatable:** avoid random input and conditions (use a seed) – can result in “flaky tests”
- **Self-validating:** each test should have a single boolean output of “pass” or “fail”
- **Thorough:** test for all use-cases (including error and edge conditions)
 - Don't just aim for 100% line coverage



Additional Ingredients of Good Unit Tests

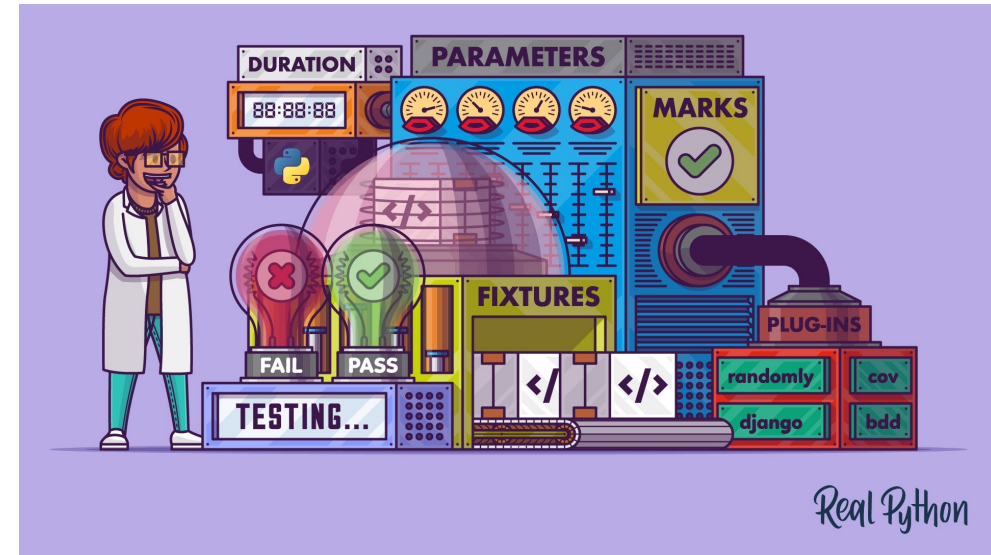
- **Existence!**
- **Simple:** split testing of different unit features between different tests
- **Readable:** the intention of the test needs to be clear, so it is helpful during future refactoring
- **Resilient:** changing the unit implementation shouldn't require major unit test changes
- **Distinct:** the unit test should not repeat the unit implementation
- **Unique:** each test should test a different aspect of the unit



pytest 

What is pytest?

- The most popular test framework for Python
- Test cases are automatically detected using patterns:
 - Modules matching either `test_*.py` or `*_test.py`
 - Functions named `test*`
 - Methods named `test*` of classes named `Test*`
- `assert` used inside the test
- Test fails if `assertion` is false or if an exception is raised
- Performs introspection of the `assertion` to report on failure reason



[Effective Python Testing With Pytest](#)

Our Example Unit

```
1 def fibonacci(n: int) -> int:  
2     if n <= 1:  
3         return n  
4     return fibonacci(n-1) + fibonacci(n-2)
```

First tests - FAIL

```
1 from fibonacci_v1 import fibonacci
2
3 def test_fibonacci():
4     assert fibonacci(0) == 1
5     assert fibonacci(1) == 1
6     assert fibonacci(4) == 3
7     assert fibonacci(5) == 5
8     assert fibonacci(11) == 89
```

```
(unittest) ~/Downloads/240416_unit_testing $ pytest 5.39G 1.63 11:44
===== test session starts =====
platform darwin -- Python 3.10.14, pytest-8.1.1, pluggy-1.4.0
rootdir: /Users/Jason/Downloads/240416_unit_testing
collected 1 item

test_fibonacci.py F [100%]

===== FAILURES =====
----- test_fibonacci -----

  def test_fibonacci():
>     assert fibonacci(0) == 1
E     assert 0 == 1
E     + where 0 = fibonacci(0)

test_fibonacci.py:4: AssertionError
===== short test summary info =====
FAILED test_fibonacci.py::test_fibonacci - assert 0 == 1
===== 1 failed in 0.05s =====
```

First tests - SUCCESS

```
1 from fibonacci_v1 import fibonacci
2
3 def test_fibonacci():
4     assert fibonacci(0) == 0
5     assert fibonacci(1) == 1
6     assert fibonacci(4) == 3
7     assert fibonacci(5) == 5
8     assert fibonacci(11) == 89
```

```
(unittest) ~/Downloads/240416_unit_testing $ pytest 5.2G 1.60 11:44
===== test session starts =====
platform darwin -- Python 3.10.14, pytest-8.1.1, pluggy-1.4.0
rootdir: /Users/Jason/Downloads/240416_unit_testing
collected 1 item
test_fibonacci.py . [100%]
===== 1 passed in 0.01s =====
```

Useful Arguments

- Select individual tests

```
pytest test_fibonacci.py::test_fibonacci
```

- Run matching tests

```
pytest -k "fib"
```

- Re-run test which failed on last run

```
pytest --last-failed
```

- Run tests until N failures

```
pytest --maxfail=2
```

- Enter interactive “pdb” debugger on test failure (or IPython debugger)

```
pytest --pdb --pdbcls=IPython.terminal.debugger:TerminalPdb
```

- Show print or logging output during tests

```
pytest -s --log-cli-level=DEBUG
```


Testing Exceptions

```
1  def fibonacci(n: int) -> int:
2      if n < 0:
3          raise ValueError("Positive integer required as argument")
4      if n <= 1:
5          return n
6      return fibonacci(n-1) + fibonacci(n-2)
7
8  import pytest
9
10 def test_fibonacci_positive():
11     with pytest.raises(ValueError):
12         fibonacci(-1)
```

Floating Point Approximations

```
1 import pytest
2
3 def test_float():
4     assert 0.1 + 0.2 != 0.3
5     assert 0.1 + 0.2 == pytest.approx(0.3)
```

<https://0.30000000000000004.com/>

Fixtures

- Re-use data or resources between tests
- Define different scopes for fixtures: session, module, class or function (default)

```
1 import pytest
2
3 @pytest.fixture
4 def data() -> list[int]:
5     return [1, 2, 3]
6
7 def multiply_inplace(d: list[int], x: float):
8     for i in range(len(d)):
9         d[i] *= x
10
11 def test_x2(data: list[int]):
12     multiply_inplace(data, 2)
13     assert data == [2, 4, 6]
14
15 def test_x3(data: list[int]):
16     multiply_inplace(data, 3)
17     assert data == [3, 6, 9]
```

Pytest-Provided Fixtures: capsys

- <https://docs.pytest.org/en/6.2.x/fixture.html>

```
1 import pytest
2
3 def test_capsys(capsys: pytest.CaptureFixture):
4     print("this is a test")
5     captured = capsys.readouterr()
6     assert captured.out == "this is a test\n"
```

Pytest-Provided Fixtures: tmp_path

- <https://docs.pytest.org/en/6.2.x/fixture.html>

```
1 import pytest
2 from pathlib import Path
3
4 @pytest.fixture
5 def abc_path(tmp_path: Path) -> Path:
6     path = tmp_path / "abc.txt"
7     with open(path, mode="w") as file:
8         file.write("abc")
9     return path
10
11 def test_abc(abc_path: Path):
12     with open(abc_path, mode="r") as file:
13         assert file.read() == "abc"
```

Fixture Clean-up

- `yield` a fixture to perform additional clean-up of the fixture after the test is finished with it

```
1 class DatabaseConnection:
2     def __init__(self, address: str):
3         self._address = address
4         self._active = False
5         self._db = {"entry0": 0}
6
7     def open(self):
8         self._active = True
9
10    def close(self):
11        self._active = False
12
13    def get(self, key: str) -> int:
14        if not self._active:
15            raise ConnectionError()
16        return self._db[key]
```

```
18 import pytest
19
20 @pytest.fixture
21 def database() -> DatabaseConnection:
22     db = DatabaseConnection("127.0.0.1:12345")
23     db.open()
24     yield db
25     db.close()
26
27
28 def test_db(database: DatabaseConnection):
29     assert database.get("entry0") == 0
```

Parameterized Test Input

- We can reduce repetition in the original example test using `pytest.mark.parametrize`

```
1 from fibonacci_v1 import fibonacci
2 import pytest
3
4 @pytest.mark.parametrize("n, expected", [
5     (0, 0),
6     (1, 1),
7     (4, 3),
8     (5, 5),
9     (11, 89),
10 ])
11 def test_fibonacci(n: int, expected: int):
12     assert fibonacci(n) == expected
```

Skipping Tests

- Skip tests where the dependencies/requirements are not met

```
1 import pytest
2 import sys
3
4 def test_awesome_dependency():
5     awesome = pytest.importorskip("awesome")
6     assert awesome.is_awesome()
7
8 @pytest.mark.skipif(sys.platform != 'win32', reason="windows only")
9 def test_windows():
10     assert os.path.exists('C:\\')
```


Expected Failures

- Mark tests which are expected to fail with
- E.g. features not yet implemented, known (but not yet fixed) bugs

```
1 import pytest
2
3 @pytest.mark.xfail
4 def test_xfail():
5     unimplemented_feature()
```

Advanced Usage

Example Database

- Lets say we have a database which our application uses
- We want to test our unit which retrieves the full name of a member from the database

```
(unittest) ~/Downloads/240416_unit_testing $ sqlite3 avengers.db
SQLite version 3.39.5 2022-10-14 20:58:05
Enter ".help" for usage hints.
sqlite> SELECT * from members;
0|Tony|Stark|M|1971-03-28
1|Peter|Parker|M|1982-05-12
2|Jessica|Jones|F|1990-07-01
```

```
1 class Avengers:
2     def get_member_name(self, member_id: int) -> str:
3         first_name = pass # Get from database
4         last_name = pass # Get from database
5         return first_name + " " + last_name
6
7 def test_avengers_get_member_name():
8     avengers = Avengers()
9     assert avengers.get_member_name(1) == "Peter Parker"
```

Naïve Attempt: High Dependence

- Our first attempt at an implementation & unit test might directly use the database
- Problems:
 - Requires the database to be available
 - Strong coupling to the database technology (sqlite)

```
1 import sqlite3
2
3 class Avengers:
4     def __init__(self):
5         self.connection = sqlite3.connect("avengers.db")
6         self.cursor = self.connection.cursor()
7
8     def get_member_name(self, member_id: int) -> str:
9         cmd = f"SELECT * FROM members WHERE member_number = {member_id};"
10        self.cursor.execute(cmd)
11        result = self.cursor.fetchall()[0]
12        return result[1] + " " + result[2]
13
14 def test_avengers_get_member_name():
15     avengers = Avengers()
16     assert avengers.get_member_name(1) == "Peter Parker"
```

Dependency Injection Pattern

- Coupling can be reduced by isolating the database into its own class ("separation of concerns")
- This class is then "injected" as a dependency for our Avengers class
- The database connection can now be reused in multiple places
- Problems remain: needs database to be available and only compatible with sqlite

```
1 import sqlite3
2
3 class SQLiteDatabase:
4     def __init__(self):
5         self.connection = sqlite3.connect("avengers.db")
6         self.cursor = self.connection.cursor()
7
8     def get_field(self, member_id: int, field: int) -> str:
9         cmd = f"SELECT * FROM members WHERE member_number = {member_id};"
10        self.cursor.execute(cmd)
11        result = self.cursor.fetchall()[0][field]
12        return result
13
14 class Avengers:
15     def __init__(self, db: SQLiteDatabase):
16         self.db = db
17
18     def get_member_name(self, member_id: int) -> str:
19         first_name = self.db.get_field(member_id, 1)
20         last_name = self.db.get_field(member_id, 2)
21         return first_name + " " + last_name
```

```
23 import pytest
24
25 @pytest.fixture
26 def database() -> SQLiteDatabase:
27     return SQLiteDatabase()
28
29 def test_avengers_get_member_name(database: SQLiteDatabase):
30     avengers = Avengers(database)
31     assert avengers.get_member_name(1) == "Peter Parker"
```

Dependency Inversion Principle

- Instead of depending on the SQL-specific database class, we can depend on a generic interface which implements the necessary methods: "Dependency inversion"
- This can be communicated in Python with `typing.Protocol`
- Both our previous `SQLDatabase` and the new `MockDatabase` implement the necessary interface defined by `GenericDatabase`
- We use `MockDatabase` to test our unit with better isolation from other units (i.e. the database)

```
1 from typing import Protocol
2
3 class GenericDatabase(Protocol):
4     def get_field(self, member_id: int, field: int) -> str:
5         ...
6
7 class Avengers:
8     def __init__(self, db: GenericDatabase):
9         self.db = db
10
11     def get_member_name(self, member_id: int) -> str:
12         first_name = self.db.get_field(member_id, 1)
13         last_name = self.db.get_field(member_id, 2)
14         return first_name + " " + last_name
```

```
18 class MockDatabase:
19     def __init__(self):
20         self._data = {1: ("1", "Peter", "Parker", "M", "1982-05-12")}
21
22     def get_field(self, member_id: int, field: int) -> str:
23         return self._data[member_id][field]
24
25 @pytest.fixture
26 def database() -> GenericDatabase:
27     return MockDatabase()
28
29 def test_avengers_get_member_name(database: GenericDatabase):
30     avengers = Avengers(database)
31     assert avengers.get_member_name(1) == "Peter Parker"
```

Seamless Integration Testing

- Leveraging advanced features of pytest, we can re-use the same test in the integration testing for situations where the SQL database is available

conftest.py

```
1 def pytest_addoption(parser):  
2     parser.addoption("--sql", action="store_true", help="Run tests against SQL database.")
```

```
37 def pytest_generate_tests(metafunc):  
38     if "dut_type" in metafunc.fixturenames:  
39         values = ["SQL", "MOCK"]  
40         metafunc.parametrize("dut_type", values)  
41  
42 @pytest.fixture  
43 def sql_available(pytestconfig) -> bool:  
44     return pytestconfig.getoption("sql", False)  
45  
46 @pytest.fixture  
47 def database(sql_available: bool, dut_type: str) -> GenericDatabase:  
48     if dut_type == "SQL":  
49         if not sql_available:  
50             pytest.skip("SQL not available")  
51         return SQLiteDatabase()  
52     else:  
53         return MockDatabase()
```

Seamless Integration Testing

- Both tests cases are ran
- If the --sql argument is not used, the test case requiring the database is skipped

```
(unittest) ~/Downloads/240416_unit_testing $ pytest -v dependency_v4.py 5.46G 2.56 13:04
===== test session starts =====
platform darwin -- Python 3.10.14, pytest-8.1.1, pluggy-1.4.0 -- /usr/local/Caskroom/miniforge3/bin/python
cachedir: .pytest_cache
rootdir: /Users/Jason/Downloads/240416_unit_testing
collected 2 items

dependency_v4.py::test_avengers_get_member_name[SQL] SKIPPED (SQL not available) [ 50%]
dependency_v4.py::test_avengers_get_member_name[MOCK] PASSED [100%]

===== 1 passed, 1 skipped in 0.01s =====
(unittest) ~/Downloads/240416_unit_testing $ pytest -v --sql dependency_v4.py
===== test session starts =====
platform darwin -- Python 3.10.14, pytest-8.1.1, pluggy-1.4.0 -- /usr/local/Caskroom/miniforge3/bin/python
cachedir: .pytest_cache
rootdir: /Users/Jason/Downloads/240416_unit_testing
collected 2 items

dependency_v4.py::test_avengers_get_member_name[SQL] PASSED [ 50%]
dependency_v4.py::test_avengers_get_member_name[MOCK] PASSED [100%]

===== 2 passed in 0.01s =====
```


Other Topics...

- `conftest.py`
- Mocking with `unittest.patch`
- Other Python unit testing frameworks (`unittest`, `hypothesis`)
- Unit testing in other languages
 - Yes, compiled languages (C++) also need unit tests
 - Yes, firmware (e.g. VHDL) also need unit tests
- Doctest-ing
- Test Driven Development

Summary

- Unit testing brings reassurance to your code
- It is worth the investment to write unit tests
 - time spent writing them now will save time debugging your code in the future
- Perfect is the enemy of good – better to write mediocre unit tests than have none at all
- Don't write unit tests for others, write them for yourself (or future you)!
 - useful also as documentation and can remind you how the unit is used
- Pytest is a very rich test framework (the richest I've seen so far)
 - but is also very simple to get started with