

Automated Testing

Software testing

Why testing?

Why automated testing?

Types of tests

Testing for data science



Why testing?

Gain confidence the software meets requirements

Why automated testing?

Manual testing is unreliable and expensive

Remove the fear of changes by getting immediate feedback (“Regression Testing”)

Provides clear documentation with code examples

Types of tests



Unit Test

Test a single functionality in isolation

e.g. `sum([1, 3]) == 4`

Usually the unit test boundary is a **function**

Small and fast to execute

No complicated dependencies

Integration Test

Test interaction between modules that have been unit tested

E.g: returning metrics from a file (requires io, parsing and processing)

Runs slower in comparison to unit test

Set up dependencies (e.g. database, specific http service)

System Test

Check the application as a whole meets functional requirements

E.g: click on a button and the document is saved

Slow to run and requires all dependencies for the application

Automated testing for data science

- **Functional requirements**

- Range of inputs (preconditions)
- Range of outputs (postconditions)
- Known inputs/outputs

- **Robustness**

- Overfitting/Underfitting test: how well does it perform to new data?
- Benchmark: how long does it take to train and / or predict?

Unit Testing Python Code

Overview of `unittest`

Running `unittest`

Given/When/Then and Assertions

Exceptions, Errors and Failures

Python Project structure

Overview of unittest (Demo)

Live demo: `courses.py`

```
python -m unittest test_courses.py
```

Given / When / Then

A framework to structure your unit tests

Given: pre-conditions

When: the behaviour that is being tested

Then: the post-conditions

Assertions



`assertEqual`

`assertFalse`

`assertNotEqual`

`assertRaises`

`assertTrue`

`assertAlmostEqual`

`assertFalse`

`assertListEqual`

See: <https://docs.python.org/3/library/unittest.html#unittest.TestCase>



CAMBRIDGE SPARK

assertEqual is your friend

Prefer `assertEqual` as it gives you a meaningful diagnostic:

```
self.assertTrue(value == 42)
```

```
AssertionError: False is not true
```

vs.

```
self.assertEqual(value, 42)
```

```
AssertionError: 42 != 43
```

What about notebooks?

Live demo: `NotebookTestDemo.ipynb`

Exceptions, Errors and Failures



Things will go wrong!

Exceptions

Python language feature to indicate problematic outcome

```
raise ValueError("Element not found")
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
ValueError: Element not found
```

Errors: unittest terminology

An unexpected exception is thrown. E.g. `TypeError`, `ValueError`

Appears at any point

Suggests test or code is broken

`TypeError: must be str, not NoneType`

Failures: unittest terminology

An assertion failed

Suggests implementation code is broken

```
self.assertEqual(value, expected)
```

```
AssertionError: 42 != 43
```

Project structure

Option 1: simple

`your_module.py` and `test_your_module.py` in the same directory

Option 2: separate folders

`/src/your_module.py`

`/tests/test_your_module.py`

Make sure that `your_module` is available in `sys.path` for the tests to run

How to write good tests?

Treat tests like any other code

Example of Bad Test

```
def test1(self):  
    x = PredictiveModel(5)  
    x.predict([1, 0, 1])  
    self.assertTrue(x.r == 10)
```

Example of Better Test

```
def test_model_predicts_correctly_with_standard_input(self):  
    model = PredictiveModel(weights=5)  
  
    input_features = [1, 0, 1]  
  
    expected_prediction = 10  
  
    result = model.predict(input_features)  
  
    self.assertEqual(result, expected_prediction)
```


Testing best practices

Verbose naming is better

Test Behaviour not implementation

Magic number anti-pattern

Don't repeat yourself

Use assertions that provide enhanced diagnostics

Exercise (15 min)

Write unit tests for `text_formatter.py`

```
python text_formatter.py
```

```
python -m unittest test_text_formatter.py
```

Think about what are the edge cases?

Fix the current implementation if necessary

Is there any missing implementation based on the documentation?

Add tests to validate the requirements



Testing Numbers: Quiz

```
>>> 0.1 + 0.2 == 0.3
```

Testing Numbers: Quiz

```
>>> 0.1 + 0.2 == 0.3
```

False

Testing Numbers: Quiz

```
>>> 0.1 + 0.2 == 0.3
```

False

```
>>> 0.1 + 0.1 == 0.2
```

Testing Numbers: Quiz

```
>>> 0.1 + 0.2 == 0.3
```

False

```
>>> 0.1 + 0.1 == 0.2
```

True

What Every Computer Scientist Should Know About Floating-Point Arithmetic

http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html



CAMBRIDGE SPARK

Error Tolerance

Absolute error

Are these two values within a specified bound of each-other?

$$e = |a - b|$$

Relative error

Are these two values within x% of each-other?

$$e = |a - b| / |a|$$

Use a Tolerance Delta in Python

```
unittest.TestCase.assertAlmostEqual(a, b)
```

```
numpy.isclose(a, b, rtol=1e-05, atol=1e-08)
```

```
math.isclose(a, b, rel_tol=1e-09, abs_tol=0.0)
```


Additional interesting Python testing topics

- **Pyhamcrest:** popular unit testing assertions library coming from the Java world
 - A bit outdated now (last commit 3 years ago)
 - Alternative by Google: <https://github.com/google/pytruth>
- **Mocking:** replace objects dependencies with controlled behaviour to isolate tests
 - <https://docs.python.org/3/library/unittest.mock.html>
- **Hypothesis:** generates tests based on expected “properties” rather than values
 - <https://hypothesis.works/articles/intro/>

Takeaways

- Testing helps build confidence that you meet the requirements
- Automated testing removes the fear of introducing changes
- Unittest provides a library of assertions to validate the expected output of functions
- Treat tests like regular code: make them readable and maintainable
- Be careful with floating points
- Numpy.testing provide testing assertions to work with arrays and floating points

PyHamcrest

- Lets you write more declarative and readable tests
- Provides extensive list of matchers for strings, lists, dictionaries, objects
- Useful diagnostics

<https://pypi.python.org/pypi/PyHamcrest>

PyHamcrest

```
from hamcrest import *
```

```
assert_that(result, ends_with(".py")  
                 matcher)
```

Common matchers

```
assert_that(result, close_to(value, delta))
```

```
assert_that(output, equal_to(expected))
```

```
assert_that(output, contains_string(expected))
```

```
assert_that(output, equal_to_ignoring_case(expected))
```

```
assert_that(list, has_item(item))
```

PyHamcrest example

```
python -m unittest test_recommender.py
```

Exercise (15min)

Refactor `unittest` assertions to use `pyhamcrest` matchers

```
python -m unittest test_hamcrest.py
```