Testing, Mocking and Profiling

Unit Testing

Structure of Test Cases

- Derive from unittest.TestCase
- Call setUp before each test
- Call tearDown after each test
- Execute tests by calling unittest.main()
- Test names begin with ‘test’

Some Test Functions

- assertEqual()
  - check for an expected result
- assert_()
  - verify a condition
- assertRaises()
  - verify that an expected exception gets raised

DocTest

- DocTest module searches for text that look like interactive Python sessions
    - Executes those sessions to verify that they work exactly as shown
- Tests are embedded in code
    - easy to define tests
    - can cause clutter

Unittest

- Derive from unittest.TestCase
- setUp call before each test
- tearDown called after each test
- execute tests by calling unittest.main()
- test names must begin with ‘test’

Components of unittest

- test fixture
  - Code needed to prepare for tests
  - Any clean-up actions
- test case
  - unit of testing (implemented as a method of test class)
- test suite
  - a collection of test cases (aggregate tests to be executed together)
- test runner
  - orchestrates the execution of tests
  - provides output about the success or failure of tests

PyTest

- PyTest runs as a separate program
  - pip install -U pytest
  - inspects code for methods beginning with test_ and runs them
- To run tests:
  - pytest -v myExample.py

Differences

- pytest
  - assert something
  - assert a==b
  - assert a<=b
- unittest
  - assertTrue(something)
  - assertEqual(a,b)
  - assertLessEqual(a,b)

Nose

- Nose is not a test methodology, but it simplifies calling other tests
- Nose will search out and run tests defined as
    - functions
    - methods of a class
    - unit tests

Mocking

- When writing unit tests the unit is typically the function, module or class being tested
- If the unit has dependencies you can end up implicitly testing them as well
- We can replace the dependency with a substitute known as a mock
- The test class will
  - Create an instance of the class under test
  - Create mock versions of any dependencies
  - Set the mock versions as properties of the class under test
  - Exercise the class under test
  - Verify that the mock was used correctly by the class under test

MagicMock Assertions

- MagicMock provides a number of assertions to check your mock was interacted with in the correct way
  - assert_called
  - assert_called_once
  - assert_called_with(args)
  - assert_called_once_with(args)
  - assert_not_called()

Patching

- The Python mocking library also has a patch() method which allows you to replace all occurrences of a particular type with a patched version

Profiling

cProfile

- Helps profile code characteristics
  - ncalls
    - The number of times a line/function is called throughout the execution of our program
  - tottime
    - The total time that the line or function took to execute
  - percall
    - The total time divided by the number of calls
  - cumtime
    - The cumulative time spent executing this line or function
  - percall
    - The quotient of cumtime divided by the number of primitive calls

Invoking cProfile

- Already part of Python
- python -m cProfile appendDeque.py

line_profiler and Kernprof

- Tool for line-by-line analysis of how long programs take to execute
  - Instead of manually wrapping timeit time calculations around every line of our code
- pip install line_profiler
  - https://github.com/rkern/line_profiler

Memory Profiling

- pip install -U memory_profiler
- python -m memory_profiler profile_function.py

Memory profile graphs

- The mprof tool takes a series of memory usage samples at a 0.1-second intervals, and then plots this usage into a series of .dat files
- Can then be analysed with matplotlib to show the memory usage of code over a period of time