SIM Global Education

Topic 8
Cleaning and restructuring data (Test Driven Development)

Learning Outcomes

After completing this topic and the recommended reading, you should be able to:

- Implement data processing pipelines that have been broken into simple steps.
- Explain what data restructuring is and define data processing pipelines to get from one form to another.
- Explain what unit tests are and write unit tests that can test the steps in a data processing pipeline.

1. Test-driven Development

What is test-driven development?

- Test-driven development is a discipline that helps professional software developers ship clean, flexible code that works, on time.

1. Identify requirements
2. Tests are written
    - Objective: to fail the modules/codes
3. Codes are written
    - Objective: to pass the tests
4. Repeat
    - Objective: 2 minutes per cycle

Professionalism and Test-Driven Development

- Robert C. Martin [IEEE Software 24(3) 2007]

- The three laws of TDD
    - You may not write production code unless you've first written a failing unit test.
    - You may not write more of a unit test than is sufficient to fail.
    - You may not write more production code than is sufficient to make the failing unit test pass.

Benefits of Test-Driven Development

- Flexibility
- Documentation
- Minimal debugging
- Better design

- Removing external dependencies helps improve testability
- Reflective thinking promotes emergent design
- A well-factored design and good test coverage also help new design emerge

2. Types of Testing

- Interface
    - Input/Output
- Exercising data structures
    - Type mismatch?
- Boundary conditions
    - Array out-of-bound?
- Execution paths
    - Infinite loop? Conditions not reached?
- Error handling
    - Try…catch

3. Assertion

- An assertion is a Boolean expression placed at a specific point in a program which will always be evaluated "TRUE" unless there is a bug in the program.
- Check whether the program is in a desirable state.

A different kind

- Comments used by the programmer
- Documents of constraints
- Codes

Types of Assertion

- Run-time Assertions
  - Ensure programming running correctly
- Unit Tests
  - Ensure functionalities
- Compile-time Assertions
  - Ensure codes written correctly

Handling Failed Assertions

- Terminate the program
  - Not good
  - Prevents issues from occurring again, but other parts couldn't carry on
- Allow execution to continue unhindered
  - Not good

    - No idea on how the error might show up later again
- Print an error message
    - Not good
    - Printing the error have little context knowledge on the error
    - Automatic system might not have an output interface
- Throw an exception to back out of the erroneous code path
    - Good
    - Find ways to deal with the error
    - Does not stop the program

Benefits of Assertions

- Detect subtle errors that might otherwise go undetected
- Detect errors sooner after they occur than they might otherwise be detected
- Make a statement about the effects of the code that is guaranteed to be true

4. Unit Testing in Python

unittest

- import unittest
    - prefix functions with “test”
    - self.“functions”

- assertEqual(a,b)
    - a == b

- assertNotEqual(a,b)
    - a != b

- assertTrue(a)
    - a == True

- assertFalse(a)
    - a == False

- assertIs(a,b)
    - a IS b

- assertIsNot(a,b)
    - a IS NOT b

- assertIsNone(a)
    - a IS None

- assertIsNotNone(a)
    - a IS NOT None

- assertIn(a,b)
    - a IN b

- assertNotIn(a,b)
    - a NOT IN b

- assertIsInstance(a,b)
    - isInstance(a,b) == True
    - a IS INSTANCE of b

- assertNotIsInstance(a,b)
    - isInstance(a,b) == False
    - a IS NOT INSTANCE of b

5. Exercises

8.05 NumPy Practice

- Refers to “8.05 numpyPractice.html”

8.103 Cleaning more Data

- Refers to “8.103 cleaningData2.html”

8.109 Defensive Coding

- Refers to “8.109 defensive_coding.html”

Useful Resources

-
    - http://