Unit Testing: doctest and unittest (PyUnit)

30 Nov 2010
CMPT140
Dr. Sean Ho
Trinity Western University

Quiz 6 (5min, 10pts)

- What two properties are needed to guarantee that a recursive function will stop? **[2]**
- Write a recursive Python function to calculate the factorial (no error checking needed). **[3]**
- Write an iterative function to do the same. **[3]**
- Assume appleInv is a dictionary where the keys are strings ('Fuji', 'Red Delicious', etc.) and the values are positive integers. **[2]**
  - You want to extract the value corresponding to the key 'Gala', but you're not sure if the key is in the dictionary. How do you do this?

Quiz 6: answers #1-2

- What two properties are needed to guarantee that a recursive function will stop? **[2]**
  - Base case (start/stop point)
  - Inductive step (progress on each recursion)
- Write a recursive Python function to calculate the factorial (no error checking needed). **[3]**

```
def factorial( n ):
    if n <= 1:
        return 1
    return n * factorial( n – 1 )
```

Quiz 6: answers #3-4

- Write an iterative function to do the same. [3]

```
def factorial( n ):
    result = 1
    for i in range(2, n+1):
        result *= i
    return result
```

- Assume appleInv is a dictionary where the keys are strings ('Fuji', 'Red Delicious', etc.) and the values are positive integers. [2]

```
appleInv.get( 'Gala', 0 )
```

WADES and testing

- Testing is part of “Scrutinize”
- Ensure our code does what it claims to do
  - Code ↔ Specifications
- Ensure our code does what the client wants
  - Code ↔ Requirements
- How do we know our code is correct?
  - Prove it mathematically (often not feasible)
  - Or demonstrate it with many test cases
- Regression testing:
  if it passes all the tests, it is “correct”

Test-driven development

- Instead of putting off testing until the end,
- Write your test cases before you code!
  - Once you have the design/specification (e.g., pre/post-conditions of a method),
  - Write comprehensive test cases first
  - Then code and test along the way
  - Your code is correct if it passes all the tests
- Unit testing: each class/method in isolation
- Integration testing:
  do all components work correctly together?

Test frameworks in Python

- Python has two built-in libraries to help you with running tests (plus more 3rd-party libraries):
- doctest: simple framework, easy to use
  - "Test script" goes in docstring
  - Small function at bottom of file runs tests
- unittest (PyUnit): more sophisticated
  - Test cases organized into test suites
  - Test cases may share common test fixtures

doctest: simple test library

- In each docstring, include a test script:

```
"""Calculate factorial.
>>> factorial(5)
24
"""
```

See factorialtest.py

- Command/response exactly as in IDLE:
  - Prefix commands with '>>>'
  - Put expected output on line by itself
  - If you expect an exception, write the expected "red text" for the exception!
- More details in Python doctest library docs

doctest: running tests

- To run doctest, put this function at the end:

```
if __name__ == “__main__”:
    import doctest
    doctest.testmod()
```

- The if statement prevents the tests from being run if another program is importing the module
- doctest.testmod() scans through all docstrings in the file and runs all test cases it can find
- If all tests succeed, output is silent
- If any tests fail, a report is output to screen
- You can also run: python -m doctest myfile.py

doctest: test narratives

- doctest searches your docstrings for text resembling a test script (e.g., '>>>')
- You can also put your test scripts in a separate file, interspersed amongst your documentation
  - As though the whole file were a docstring
- A test narrative is a document written for humans (e.g., user manual) where test cases are interleaved with the narrative
- See factorialtest.txt for an example
- Run: `python -m doctest factorialtest.txt`
- Or in code: `doctest.testfile(“factorialtest.txt”)`

unittest (PyUnit)

- doctest is quick and easy to use, but limited
- The unittest module provides more flexibility:
  - Organize test cases into suites
    (in separate classes and even separate files)
  - Fixtures: common setup / tear-down
    for all test cases in a suite
- Uses standard methodology from Java (JUnit)
- Test suites are classes (inherit from TestCase)
- Test cases are methods within a suite
  - Prefix method name with “test_”

unittest: creating a test suite

See factorialunit.py

- Usually, put the test suites in separate files from the modules you are testing
- Import unittest and the module you want to test
- Create test suites as subclasses of TestCase:

```
class FactorialTests( unittest.TestCase ):
```

- Fixtures: define setUp() / tearDown() methods to be run before/after each test (separately)
- Test cases: define test_() methods
- Use self.assert() methods to check results:

```
self.assertEqual( factorial(6), 720 )
```

unittest: running tests

- Put the following at the end of the file of tests:

```
if __name__ == '__main__':
    unittest.main()
```

- Run from the command line:

```
python myunittests.py
```

- Outputs results and any failures
- Can also run from IDLE, but it will try to SystemExit after tests are finished