PRG – PROGRAMMING ESSENTIALS

Lecture 8 – Testing, Unit Tests, Exceptions

Milan Nemy

Czech Technical University in Prague,
Faculty of Electrical Engineering, Dept. of Cybernetics

https://beat.ciirc.cvut.cz/people/milan-nemy/

milan.nemy@cvut.cz

TESTING

- Including **automated tests** proves invaluable if the project becomes larger or if we have to return to it to make a small change after a long absence
- Tests serve as a form of **documentation** – by reading through test cases we can get an idea of the expected behavior
- Test driven approach – **writing tests first**, thereby creating a **specification** for what the program is supposed to do, and filling in the actual program code **afterwards**

SOURCE: courtesy of Petr Posik BE5b33PR 2016/2017

SELECTING TEST CASES

- Two major approaches: **black-box** or **glass-box** testing
- In **black-box** testing treat tested function like an opaque "black box" – only think about what the function is supposed to do
  (*strategies*: **equivalence testing**, **boundary value analysis**)
- In **glass-box** testing choose test cases by analyzing the code inside our function
  (*strategies*: **path coverage**, **statement coverage**)

SOURCE http://python-textbok.readthedocs.io/en/1.0/Packaging_and_Testing.html#testing

TESTING

Example: `sum_digits()`

Specifications: In module `tools.py`, create function `sum_digits(string)` which return the sum of all digits in `string`.

Solution: We create the required module as follows:

```
%%writefile tools.py
def sum_digits(string):
    """Return the sum of all digits in the string"""
    sum = 0
    for ch in string:
        if ch in '012346789':
            sum += int(ch)
    return sum
```

```
Writing tools.py
```

Are we finished? How do we test the code?

SOURCE: courtesy of Petr Posik BE5b33PR 2016/2017

TESTING

Option 1: Try to use it in Python shell

```
>>> from tools import sum_digits
>>> sum_digits('1, 2, 3, dee, dah, dee')
```

6

- We have tested a single test case.
- We have to manually check the correctness of the result.
- What if we want to run the test again?

SOURCE: courtesy of Petr Posik BE5b33PR 2016/2017

TESTING

Option 2: Including the test code directly in the module

The code previously written on Python console can be stored directly with the module (or in some other module).

```
%%writefile tools2.py
def sum_digits(string):
    """Return the sum of all digits in the string"""
    sum = 0
    for ch in string:
        if ch in '012346789':
            sum += int(ch)
    return sum

if __name__ == "__main__":
    # All the code below is executed only when the file is run as a script.
    print(sum_digits('1, 2, 3, dee, dah, dee'))
```

SOURCE: courtesy of Petr Posik BE5b33PR 2016/2017

TESTING

```
import tools2    # "Nothing" happens when we import the module (desired), ...
```

```
%run tools2.py     # ... but the testing code is executed when we run the module!
```

```
6
```

- We still test a single test case only.
- We still have to manually check the correctness of the result.
- **But we can run the test easilly. As many times as we want!**

SOURCE: courtesy of Petr Posik BE5b33PR 2016/2017

TESTING

Option 3: Check the correctness of the result automatically

Instead of mere printing out the result, we can check its correctness!

```
%%writefile tools3.py
def sum_digits(string):
    """Return the sum of all digits in the string"""
    sum = 0
    for ch in string:
        if ch in '012346789':
            sum += int(ch)
    return sum

if __name__ == "__main__":
    observed = sum_digits('1, 2, 3, dee, dah, dee')
    expected = 6
    if observed == expected:
        print('.')
    else:
        print('Test failed.')
        print('- Expected:', str(expected))
        print('- But got: ', str(observed))
```

SOURCE: courtesy of Petr Posik BE5b33PR 2016/2017

TESTING

```
%run tools3.py
```

.

- We still test a single test case only.
- **But we do not have to manually check the correctness of the result, we can immediately see if the test passed or not.**
- **And we can run the test easilly. As many times as we want!**

SOURCE: courtesy of Petr Posik BE5b33PR 2016/2017

TESTING

Our own module for testing!

The process of checking the correctness of a result may be extracted to a function that will

- allow us to write tests using only a little code,
- be part of a module that can be reused in many projects.

Let's create module `testing` with function `test_equal()` which shall have 3 parameters:

- the `observed` and `expected` values, and
- an optional `name` of the test.

The function shall print

- `"."` if the test passes, or
- an informative message about the failure, if the test fails.

SOURCE: courtesy of Petr Posik BE5b33PR 2016/2017

TESTING

```
%%writefile testing.py
import sys

def quote(name):
    if name:
        name = "'" + name + "' "
    return name

def test_equal(observed, expected, name=''):
    """Compare the observed and expected results"""
    if observed == expected:
        print('.', end='')
    else:
        linenum = sys._getframe(1).f_lineno   # Get the caller's line number.
        print("\nTest {}at line {} FAILED:".format(quote(name), linenum))
        print("- Expected:", str(expected))
        print("- But got: ", str(observed))
```

SOURCE: courtesy of Petr Posik BE5b33PR 2016/2017

TESTING

With the help of our `testing` module, we can rewrite the `tools` module as follows:

```
%%writefile tools4.py
from testing import test_equal

def sum_digits(string):
    """Return the sum of all digits in the string"""
    sum = 0
    for ch in string:
        if ch in '012346789':
            sum += int(ch)
    return sum

if __name__ == "__main__":
    test_equal(sum_digits('1, 2, 3, dee, dah, dee'), 6, 'Test 1')
```

SOURCE: courtesy of Petr Posik BE5b33PR 2016/2017

TESTING

```
%run tools4.py
```

.

- We still test a single test case only.
- **But we do not have to manually check the correctness of the result, we can immediately see if the test passed or failed.**
- **And we do not need to write much code to test a single case!**
- **And we can run the tests easilly. As many times as we want!**

SOURCE: courtesy of Petr Posik BE5b33PR 2016/2017

TESTING

Adding more tests

When we have more test cases, we can add them either

- to the `if __name__=="__main__"` section of the main file, or
- to a separate testing module.

Let's create a separate testing module.

```
%%writefile test_tools.py
from testing import test_equal
from tools4 import *

def test_sum_digits():
    test_equal(sum_digits(''), 0, 'Test empty string')
    test_equal(sum_digits('0'), 0, 'Test 0')
    test_equal(sum_digits('1'), 1, 'Test 1')
    test_equal(sum_digits('2'), 2, 'Test 2')
    test_equal(sum_digits('3'), 3, 'Test 3')
    test_equal(sum_digits('4'), 4, 'Test 4')
    test_equal(sum_digits('5'), 5, 'Test 5')
    test_equal(sum_digits('6'), 6, 'Test 6')
    test_equal(sum_digits('7'), 7, 'Test 7')
    test_equal(sum_digits('8'), 8, 'Test 8')
    test_equal(sum_digits('9'), 9, 'Test 9')
    test_equal(sum_digits('1, 2, 3, dee, dah, dee'), 6, 'Non trivial test')

# Run the test suite
test_sum_digits()
```

SOURCE: courtesy of Petr Posik BE5b33PR 2016/2017

TESTING

```
%run test_tools.py
```

```
......
Test 'Test 5' at line 11 FAILED:
- Expected: 5
- But got:  0
.....
```

Ha! We have an error in our code! Can you find it?

With the help of a testing framework:

- We can easilly build comprehensive test suites.
- We do not have to manually check the correctness of the result, we can immediately see if the test passed or failed.
- We do not need to write much code to test a single case!
- We can run the test suite easilly. As many times as we want.

Other testing frameworks

Our module `testing` is not an original idea. Python has several popular testing frameworks, e.g. modules

- `doctest` and
- `unittest`.

SOURCE: courtesy of Petr Posik BE5b33PR 2016/2017

TESTING

Testing the code using `doctest`

- Create the habit to include examples of the functions' usage in their docstrings (see below).
- Module `doctest` allows you to easilly execute the examples from the docstrings:

```
%%writefile modulewithdoctests.py
def average(x,y):
    """Return the average of 2 numbers.

    >>> average(10,20)
    15.0
    >>> average(1.5, 2.0)
    1.75
    """
    return (x + y) / 2

if __name__ == "__main__":
    import doctest
    doctest.testmod(verbose=True)
```

```
Writing modulewithdoctests.py
```

SOURCE: courtesy of Petr Posik BE5b33PR 2016/2017

TESTING

Then, if you run the module, the tests are executed automatically and compared with their expected results:

```
%run modulewithdoctests.py
```

```
Trying:
    average(10,20)
Expecting:
    15.0
ok
Trying:
    average(1.5, 2.0)
Expecting:
    1.75
ok
1 items had no tests:
    __main__
1 items passed all tests:
   2 tests in __main__.average
2 tests in 2 items.
2 passed and 0 failed.
Test passed.
```

SOURCE: courtesy of Petr Posik BE5b33PR 2016/2017

TESTING

Summary

- Testing your own code is **extremely important!**
- You should learn several ways how to test your code.
- Using a **testing framework**, from simple ones (like our `testing`) to comprehensive ones (like `unittest`), gives you an considerable **advantage**!
- Testing frameworks like `unittest` are common to many other languages. If you learn it for one languaga, you will profit from it also in the other languages.

SOURCE: courtesy of Petr Posik BE5b33PR 2016/2017

UNITTESTS

```
ourprog/
    ourprog/
        __init__.py
        db.py
        gui.py
        rules.py
        test/
            __init__.py
            test_db.py
            test_gui.py
            test_rules.py
    setup.py
```

- Advanced framework for testing – python **unittest** module
- All tests in a file hierarchy **separated** from main the program (directory *test/*)
- Create a **test module for each program module** and put them all in a **separate test directory**

SOURCE http://python-textbok.readthedocs.io/en/1.0/Packaging_and_Testing.html#testing

UNITTESTS

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

```
# these commands will try to find all our tests
python -m unittest
python -m unittest discover

# but we can be more specific
python -m unittest ourprog.test.test_rules
python -m unittest ourprog.test.test_rules.TestPerson
python -m unittest ourprog.test.test_rules.TestPerson.test_fullname

# we can also turn on verbose output with -v
python -m unittest -v test_rules
```

- *Many ways of running tests!* (test automation frameworks)
- Run all the tests from a single file by adding **unittest.main()** at the bottom of *test_rules.py* and **execute as a script**
- Execute the unittest module on the **commandline** and use it to import and run some or all of our tests

SOURCE http://python-textbok.readthedocs.io/en/1.0/Packaging_and_Testing.html#testing

UNITTESTS

```
def suite():
    suite = unittest.TestSuite()
    suite.addTest(TestPerson)
    return suite
```

- The **unittest** package allows to **group some or all of our tests into suites**
- This way many related tests can be executed at once
- **EXAMPLE**:
  One way to add all the tests from the **TestPerson** class to a suite is to add for example **suite()** function to the **test_rules.py** file

SOURCE http://python-textbok.readthedocs.io/en/1.0/Packaging_and_Testing.html#testing

UNITTESTS – DOCUMENTATION

The `unittest` module provides a rich set of tools for constructing and running tests. This section demonstrates that a small subset of the tools suffice to meet the needs of most users.

Here is a short script to test three string methods:

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)

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

SOURCE https://docs.python.org/3.7/library/unittest.html

UNITTESTS – DOCUMENTATION

A testcase is created by subclassing `unittest.TestCase`. The three individual tests are defined with methods whose names start with the letters `test`. This naming convention informs the test runner about which methods represent tests.

The crux of each test is a call to `assertEqual()` to check for an expected result; `assertTrue()` or `assertFalse()` to verify a condition; or `assertRaises()` to verify that a specific exception gets raised. These methods are used instead of the `assert` statement so the test runner can accumulate all test results and produce a report.

The `setUp()` and `tearDown()` methods allow you to define instructions that will be executed before and after each test method. They are covered in more detail in the section Organizing test code.

The final block shows a simple way to run the tests. `unittest.main()` provides a command-line interface to the test script. When run from the command line, the above script produces an output that looks like this:

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

SOURCE https://docs.python.org/3.7/library/unittest.html

UNITTESTS – DOCUMENTATION

Passing the `-v` option to your test script will instruct `unittest.main()` to enable a higher level of verbosity, and produce the following output:

```
test_isupper (__main__.TestStringMethods) ... ok
test_split (__main__.TestStringMethods) ... ok
test_upper (__main__.TestStringMethods) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

The above examples show the most commonly used `unittest` features which are sufficient to meet many everyday testing needs. The remainder of the documentation explores the full feature set from first principles.

- Verbosity for the tests can be defined using **-v**

SOURCE https://docs.python.org/3.7/library/unittest.html

UNITTESTS – DOCUMENTATION

The unittest module can be used from the command line to run tests from modules, classes or even individual test methods:

```
python -m unittest test_module1 test_module2
python -m unittest test_module.TestClass
python -m unittest test_module.TestClass.test_method
```

You can pass in a list with any combination of module names, and fully qualified class or method names.

Test modules can be specified by file path as well:

```
python -m unittest tests/test_something.py
```

- Unit tests can be executed for specified **modules**, **classes**, or **methods**; path to a python file can be used as well

SOURCE https://docs.python.org/3.7/library/unittest.html

EXCEPTIONS

- Whenever **runtime error** occurs, it creates an *exception object*
- The **program stops running** at this point and Python prints out the *traceback*, which ends with an **error message** describing the exception that occurred

- **Exception** – An error that occurs at runtime
- **Handle an exception** – To prevent an exception from causing our program to *crash*, by wrapping the block of code in a try ... except construct
- **Raise** – To create a deliberate exception by using the raise statement

SOURCE http://openbookproject.net/thinkcs/python/english3e/exceptions.html

EXCEPTIONS

```
>>> print(55/0)
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

```
>>> a = []
>>> print(a[5])
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
IndexError: list index out of range
```

```
>>> tup = ("a", "b", "d", "d")
>>> tup[2] = "c"
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

- The **error message** on the last line has two parts:
  - the **type of error** before the colon,
  - and **specifics** about the error after the colon

SOURCE http://openbookproject.net/thinkcs/python/english3e/exceptions.html

EXCEPTIONS

```
1  filename = input("Enter a file name: ")
2  try:
3      f = open(filename, "r")
4  except:
5      print("There is no file named", filename)
```

- **TASK:** To execute an operation that might **cause an exception** but **does not stop** the program
- **SOLUTION**: Handle the exception using the **try-except** statement to “wrap” a region of code
- **EXAMPLE**: *Prompt the user for the name of a file and then try to open it. If the file does not exist, we do not want the program to crash*

SOURCE http://openbookproject.net/thinkcs/python/english3e/exceptions.html

EXCEPTIONS

```
1  def exists(filename):
2      try:
3          f = open(filename)
4          f.close()
5          return True
6      except:
7          return False
```

- The **try statement** has three separate clauses, or parts, introduced by the keywords **try … except … else … finally**

- The **except**, **else** or the **finally** clauses can be omitted
- The try statement executes and **monitors the statements** in the first block and *If no exceptions occur, it skips the block under the except clause*

- If any *exception occurs, it executes the statements in the except clause* and then continues

SOURCE http://openbookproject.net/thinkcs/python/english3e/exceptions.html

EXCEPTIONS

```
>>> while True:
...     try:
...         x = int(input("Please enter a number: "))
...         break
...     except ValueError:
...         print("Oops!  That was no valid number.  Try again...")
...
```

The `try` statement works as follows.

- First, the *try clause* (the statement(s) between the `try` and `except` keywords) is executed.
- If no exception occurs, the *except clause* is skipped and execution of the `try` statement is finished.
- If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the `except` keyword, the except clause is executed, and then execution continues after the `try` statement.
- If an exception occurs which does not match the exception named in the except clause, it is passed on to outer `try` statements; if no handler is found, it is an *unhandled exception* and execution stops with a message as shown above.

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

SOURCE https://docs.python.org/3/tutorial/errors.html

EXCEPTIONS

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print("division by zero!")
...     else:
...         print("result is", result)
...     finally:
...         print("executing finally clause")
...
>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

- The **finally** clause is *always executed before leaving the try statement* whether the exception has occurred or not
- When an exception is *not handled by corresponding except* clause (or is raised in the *except* or the *else* clause), it is re-raised after the finally (see the example for division of strings)

SOURCE https://docs.python.org/3/tutorial/errors.html

EXCEPTIONS

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except OSError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

The use of the `else` clause is better than adding additional code to the `try` clause because it avoids accidentally catching an exception that wasn't raised by the code being protected by the `try` ... `except` statement.

- Optional **else** clause that must follow all **except** clauses
- Useful for code that must be executed if the **try** clause *does not raise* an exception

SOURCE https://docs.python.org/3/tutorial/errors.html

EXCEPTIONS

```
1  def get_age():
2      age = int(input("Please enter your age: "))
3      if age < 0:
4          # Create a new instance of an exception
5          my_error = ValueError("{0} is not a valid age".format(age))
6          raise my_error
7      return age
```

```
>>> get_age()
Please enter your age: 42
42
>>> get_age()
Please enter your age: -2
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
  File "learn_exceptions.py", line 4, in get_age
    raise ValueError("{0} is not a valid age".format(age))
ValueError: -2 is not a valid age
```

- If the program detects an error condition, an exception can be **raised manually**.
- **EXAMPLE**: take input from the user and check that the number is non-negative

SOURCE http://openbookproject.net/thinkcs/python/english3e/exceptions.html

EXCEPTIONS

- **Line 5** creates an **exception object**, the **ValueError object**, that encapsulates specific information about the error

- **EXAMPLE**: Assume that in this case function A called B which called C which called D which called get_age()*:*

  - *The raise statement on line 6 carries this object out as a kind of "return value", and immediately exits from get_age() to its caller D*
  - *Then D again exits to its caller C, and C exits to B and so on, each returning the exception object to their caller, until it encounters a try ... except that can handle the exception*

SOURCE http://openbookproject.net/thinkcs/python/english3e/exceptions.html

EXCEPTIONS

```
1  raise ValueError("{0} is not a valid age".format(age))
```

- It is often the case that lines 5 and 6 (**creating the exception** object, then **raising the exception**) are combined into a single statement
- Those are **two different and independent things**, so it makes sense to keep the two steps separate

- **ERRORS** – multiple except clauses to handle different kinds of exceptions https://docs.python.org/3/tutorial/errors.html

SOURCE http://openbookproject.net/thinkcs/python/english3e/exceptions.html

REFERENCES

This lecture re-uses selected parts of the OPEN BOOK PROJECT
Learning with Python 3 (RLE)
http://openbookproject.net/thinkcs/python/english3e/index.html
available under **GNU Free Documentation License Version 1.3**)

- Version date: October 2012
- by Peter Wentworth, Jeffrey Elkner, Allen B. Downey, and Chris Meyers (based on 2nd edition by Jeffrey Elkner, Allen B. Downey, and Chris Meyers)
- Source repository is at https://code.launchpad.net/~thinkcspy-rle-team/thinkcspy/thinkcspy3-rle
- For offline use, download a zip file of the html or a pdf version from http://www.ict.ru.ac.za/Resources/cspw/thinkcspy3/

This lecture re-uses selected parts of the PYTHON TEXTBOOK
Object-Oriented Programming in Python
http://python-textbok.readthedocs.io/en/1.0/Packaging_and_Testing.html#testing
(released under CC BY-SA 4.0 licence Revision 8e685e710775)