6. előadás

Tesztelés

Programozás (2) előadás

2022. Október 17.

Halász Gábor

Debreceni Egyetem

Általános tudnivalók

Ajánlott irodalom:

- Nyékyné G. Judit (szerk): Programozási nyelvek, Kiskapu, 2003.
- Juhász, István: Magas szintű programozási nyelvek 2, elektronikus egyetemi jegyzet, 2009
- Tarczali, Tünde: UML diagramok a gyakorlatban, Typotex Kiadó, 2011.
- Angster, Erzsébet: Objektumorientált tervezés és programozás: JAVA, 4KÖR Bt., 2002, ISBN: 9632165136
- Bird, S., Klein, E., Loper, E.: Natural Language Processing with Python, O'Reilly Media, 2009

Félév teljesítésének feltételei: jelenlét + 2 gyakorlati + 1 elméleti ZH

Érdemjegy: $1 < 60\% \leq 2 < 70\% \leq 3 < 80\% \leq 4 < 90\% \leq 5$

További részletek: `https://elearning.unideb.hu/`

Why Testing?

Why Testing?

- Earlier we talked about protecting our programs from bad input, from users
- Who is going to protect us, the programmers, from ourselves (from writing programs with mistakes)?
- The answer is that we, the programmers, must protect ourselves. We are responsible for testing our programs

“Good” programs

- As much as possible, programmers need to integrate testing into programming.
  - how do you know if your program is right without tests to show that it is?
- There are a number of programming paradigms that require writing tests first, before a single line of code is written (test- driven programming)

Kinds of error

Kinds of error

Syntax:	Errors of language use. Usually caught by Python
Semantics:	Syntax is correct, but we are asking Python to do something it cannot (divide by 0, convert ‘a’ to an integer, etc.)
Design:	Program doesn’t fulfill our expectations. No errors but wrong answers, incomplete results etc.

Testing Helps

- Testing can help us avoid both semantic and design errors
- In so doing we can avoid errors, or if we still encounter them recover more quickly as the code is being developed, rather than having to rewrite it after the fact.

design-test-program

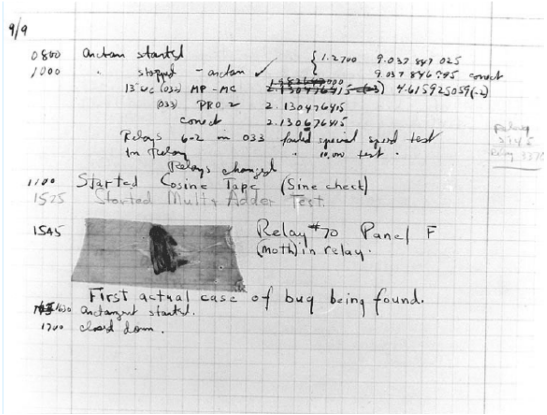
- A good programmer should:
  - design their program before they write it.
    - what functions, what classes, user interface etc.
  - write the test programs that the program should pass
  - then write the code
- proper design and testing makes writing code much easier, much clearer, less error prone

The term “bug”

- The term bug, at least as it applies to computing, is used to indicate an error in the program (of any kind)
- predates computing:
  - Edison used the term
  - Telegraphers and some of their specialized equipment
- Admiral Hopper had a real example!

> In 1946, when Hopper was released from active duty, she joined the Harvard faculty at the Computation Laboratory where she continued her work on the Mark II and Mark III. Operators traced an error in the Mark II to a moth trapped in a relay, coining the term *bug*. This bug was carefully removed and taped to the log book. Stemming from the first bug, today we call errors or glitch's [sic] in a program a *bug*.[1]

[1] http://ei.cs.vt.edu/~history/Hopper.Danis.html



```
9/9

0800   Arctan started                           { 1.2700   9.037 847 025
1000     "    stopped - arctan ✓                {          9.037 846 995 correct
         13"00 (033) MP - MC   ~~1.982147000~~
                               ~~2.130476415~~ (-3)   4.615925059(-2)
               (033) PRO 2     2.130476415
                     correct   2.130676415
         Relays 6-2 in 033 failed special speed test
         In relay          "  10,000 test .
                Relays changed
1100   Started Cosine Tape (Sine check)
1525   Started Mult + Adder Test.
1545                           Relay #70 Panel F
                               (moth) in relay.
       First actual case of bug being found.
1630   Arctangent started.
1700   closed down.
                                                          Relay
                                                          2145
                                                          Relay 3370
```

Poor choice of term

- The use of the term is disingenuous[1]
- The bug didn't just "creep in" while we were sleeping! It was an error we made.
- It is our fault (the programmer), not some external force. We are responsible.

[1] not candid or sincere, typically by pretending that one knows less about something than one really does.

Edsger Dijkstra on bugs

“We could, for instance, begin with cleaning up our language by no longer calling a bug ‘a bug’ but by calling it an error. It is much more honest because it squarely puts the blame where it belongs, viz. , with the programmer who made the error ... While before, a program with only one bug used to be ‘almost correct,’ afterwards a program with an error is just ‘wrong.’ ”

Kinds of testing

Testing is hard!

- Before we go to far, it is important to note that testing is both a complicated area as well as a broad area (many aspects)
- We won't be able to cover such a big topic in a short time, but we can give an overview.
- We encourage you to look deeper!

static vs. dynamic (1)

- Also described as planning versus testing of the code written to meet the plans
  - planning means understanding the problem and its requirements before you write the code. Shouldn't change over time (though in real life it does), hence static.
  - testing of code means fixing what you wrote to implement the plan. The code is constantly updated, hence dynamic

post vs pre testing (2)

When to test? Before or after coding?

- post testing is the traditional way, code it then test to make sure it runs
- pre testing is a more modern approach. Write the tests before the code, then make sure code updates work, and add to the test.

Levels of testing

Unit testing.
: Test each individual unit (function, class, etc.)

Integration testing.
: Do two units work together as designed.

System testing.
: Does the overall system do what it is supposed to do.

Test What?

This is probably the hardest part. Test what?

- This is really a topic unto itself. There are many aspects of a program that we might like to test
- Knowing the options is part of the problem.

Test What (1)

Correctness.

Are the results produced correct? What counts as correct? Do we know all the cases?

Completeness.

Are all the potential cases covered: all user entries, all file formats, all data types, etc.

Test What (2)

Security.
: Is the program safe from being broken into by another user, through the Internet, etc.

Interface.
: Is the interface useable? Are its features complete? Does the user get what they expect when using the program?

Test What (3)

Load.
Does the program respond well if it is presented with a heavy load: many users, large data sets, heavy Internet traffic etc.

Resources.
Does the program use appropriate amounts of memory, CPU, network?

Test What (4)

Responsiveness.

Does the program respond in a reasonable amount of time. If there are time limits, does it meet those limits?

Correctness, the Pentium story

- Is it possible that you can prove that your program is correct? Some researchers are doing just that, proving correctness
- Pentium chip of 1993 had a small flaw in its division algorithm (1 in 1x10^9 divisions). Never found in testing, showed up in the field for people doing research

NBA efficiency example

Catching User Errors

- We know how to do this. Chapter 14, the discussion about exceptions, gives us the tool to write our program so we might protect it
- This gives us security, better usability and other factors.

```
def main ():
    file_name = input('NBA player file name:')

    try:
        nba_file = open(file_name)
    except IOError:
        print('File named {} not found'.format(file_name))
    else:
        nba_dict={}
        # check the first line
        line_str = nba_file.readline()
        if line_str[0:5]!='ilkid':
            print('Bad File Format, first line was:',line_str)
            raise IOError
        # process the rest of the lines
        for line_str in nba_file:
            calc_efficiency(line_str,nba_dict)
        results_list = find_most_efficient(nba_dict,20)
        print_results(results_list)
        nba_file.close()
```

Catch Developer Errors

- We need to protect ourselves from ourselves as well.
- Couple of ways we can check ourselves as we program
  - insert tests in the code for conditions which should never occur if the code is written correctly
  - add tests to the code to automatically check for test conditions provided by the programmer

The `assert` statement

- The `assert` statement has an associated Boolean clause. This clause is a condition that must always be `True` if the code is correct
- If the clause evaluates to `False`, then the program is halted.

more `assert`

- If `assert` is triggered, it raises the builtin `AssertionError`
- `assert` can take a string, separated from the clause by a comma, to print out with the error

```
def calc_efficiency (line_str, the_dict):
    # asserts on the parameters
    assert isinstance(the_dict,dict),\
           'bad parameter, expected a dictionary, got {}'.format(the_dict)
    assert isinstance(line_str,str) and line_str != '', \
           'bad parameter, expected string, got {}'.format(line_str)

    line_str = line_str.strip()
    fields_list_list = line_str.split(',')
    first_name = fields_list_list[1]
    last_name = fields_list_list[2]

    # mapping fields_list in a line to their particular variable.
    # league is a str, everything else an int
    leag,gp,mins,pts,oreb,dreb,reb,asts,stl,blk,to,pf,fga,fgm,fta,ftm,tpa,
tpm = \
     fields_list_list[3],int(fields_list[4]),int(fields_list[5]),int
(fields_list[6]),\
     int(fields_list[7]),int(fields_list[8]),int(fields_list[9]),int
(fields_list[10]),\
     int(fields_list[11]),int(fields_list[12]),int(fields_list[13]),\
     int(fields_list[14]),int(fields_list[15]),int(fields_list[16]),\
     int(fields_list[17]),int(fields_list[18]), int(fields_list[19]),\
     int(fields_list[20])

    # gp can't be 0
    assert gp!= 0, '{} {} has no games played'.format(first_name, last_name)
```

assert is for Programmers

- when `calc_efficiency` is called, it is not called by a user, nor does that function require user input
- the assumption is that what is passed to the function is already correct, as the programmer is making the invocation
- check to make sure we, the programmer, don't make a mistake.

Inserting Tests

doctest

- `doctest` is a module provided by Python for module testing
- lets you describe (and then run) good usage examples for your code
- when code is changed, you can run those tests.
- also helps the reader/user to see what the designer's intention is

doctest modifies the docstring

- in the docstring at the top of the module, you can place special tests
- each test is preceeded by a `>>>`
- just after the test is the exact result of having run that test
- each test is run and the output compared. If no problems, then the module ‘passes’ and no output is produced

testmod

- once written, you must call the `doctest.testmod()` function
- it looks in the file and examines the docstring for `>>>`
- runs each test and compares the answer
- if you run with a `-v` command line, all the tests will be compared
- otherwise, no output

can’t be too long

- cannot have the the docstring be too long as it becomes too much for the user
- however, if you are developing it is a convenient thing to run tests after each change
- short tests can then be used in conjunction with system tests

```
def main (file_name):
    '''

   >>> main('')
    File named  not found
   >>> main('x')
    Traceback (most recent call last):
        ...
        raise IOError('bad file format.')
    IOError: bad file format.
   >>> main('player_career.csv')
    The top 10 players in efficency are
    ******************
              Wilt Chamberlain : 41.50
                  Bill Russell : 31.71
              Oscar Robertson : 31.61
                    Bob Pettit : 31.11
           Kareem Abdul-jabbar : 30.93
                    Larry Bird : 29.77
                  Elgin Baylor : 29.74
                Michael Jordan : 29.19
                 Magic Johnson : 29.10
               Charles Barkley : 28.16
    '''

    try:
        nba_file = open(file_name)
    except IOError:
```

```
>python nbaEfficiency.py -v
Trying:
    main('')
```

```
Expecting:
    File named  not found
ok
Trying:
    main('x')
Expecting:
    Traceback (most recent call last):
        ...
    IOError: bad file format, line was: this is a bad file
ok
Trying:
    main('player_career.csv')
Expecting:
    The top 10 players in efficency are
    ********************
             Wilt Chamberlain : 41.50
                 Bill Russell : 31.71
              Oscar Robertson : 31.61
                   Bob Pettit : 31.11
          Kareem Abdul-jabbar : 30.93
                   Larry Bird : 29.77
                 Elgin Baylor : 29.74
               Michael Jordan : 29.19
                Magic Johnson : 29.10
              Charles Barkley : 28.16
ok
4 items had no tests:
    __main__
    __main__.calcEfficiency
    __main__.findMostEfficient
    __main__.printResults
1 items passed all tests:
   3 tests in __main__.main
3 tests in 5 items.
3 passed and 0 failed.
Test passed.
```

unittest module

- system testing is done using the `unittest` module
- unlike `doctest`, this is a separate file that describes in detail how to perform the overall test
- probably more useful, since it is not embedded in the sourcefile itself.

more complicated, more features

- Basically, inherit from `TestCase`:
- every method that begins with ‘test’ is run
- `setUp` is run before every case
- `tearDown` is run after every case
- `assertEqual` checks for equal result
- `assert_()` to check a condition
- `assertRaises` to check for a raised exception

```
import unittest
# Our code to be tested
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height
    def get_area(self):
        return self.width * self.height
    def set_width(self, width):
        self.width = width
    def set_height(self, height):
        self.height = height
# The test based on unittest module
class TestGetAreaRectangle(unittest.TestCase):
    def runTest(self):
        rectangle = Rectangle(2, 3)
        self.assertEqual(rectangle.get_area(), 6,\
                                   "incorrect area")
# run the test
unittest.main()
```

rectangle.py:

```
# Our code to be tested
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height
    def get_area(self):
        return self.width * self.height
    def set_width(self, width):
        self.width = width
    def set_height(self, height):
        self.height = height
```

test_rectangle.py:

```
import unittest
import rectangle as rect
# The test based on unittest module
class TestGetAreaRectangle(unittest.TestCase):
    def runTest(self):
        rectangle = rect.Rectangle(2, 3)
        self.assertEqual(rectangle.get_area(), 6,\
                                               "incorrect area")
# run the test
unittest.main()
```

output:

```
.
------------------------------------
Ran 1 test in 0.003s
OK
```

```
import unittest
import rectangle as rect
# The test based on unittest module
class TestGetAreaRectangleWithSetUp(unittest.TestCase):
    def setUp(self):
        self.rectangle = Rectangle(0, 0)
    def test_normal_case(self):
        self.rectangle.set_width(2)
        self.rectangle.set_height(3)
        self.assertEqual(self.rectangle.get_area(),\
                                   6, "incorrect area")
    def test_negative_case(self):
        """expect -1 as output to denote error
                   when looking at negative area"""
        self.rectangle.set_width(-1)
        self.rectangle.set_height(2)
        self.assertEqual(self.rectangle.get_area(),\
                   -1, "incorrect negative output")
# run the test
unittest.main()
```

output:

```
F.
======================================================================
FAIL: test_negative_case (__main__.TestGetAreaRectangle)
expect -1 as output to denote error when looking at negative area
----------------------------------------------------------------------
Traceback (most recent call last):
File "<ipython-input-96-59b1047bb08a>", line 9, in test_negative_case
self.assertEqual(rectangle.get_area(), -1, "incorrect negative output")
AssertionError:  -2 != -1 :  incorrect negative output
----------------------------------------------------------------------
Ran 2 tests in 0.003s
FAILED (failures=1)
```

We can even check whether a **particular exception** was thrown during execution:

```
def sting_divide(a, b):
    return int(a) / int(b)



          ...

def test_assert_raises(self):
    """using assertRaises to detect if an expected error is raised
                                    when running a particular block of code"""
    with self.assertRaises(ZeroDivisionError):
        sting_divide('1', '0')
```

more complicated, more features

- Basically, inherit from `TestCase`:
- every method that begins with ‘test’ is run
- `setUp` is run before every case
- `tearDown` is run after every case
- `assertEqual` checks for equal result
- `assert_()` to check a condition
- `assertRaises` to check for a raised exception

Reminder, rules so far

1. Think before you program!
2. A program is a human-readable essay on problem solving that also happens to execute on a computer.
3. The best way to improve your programming and problem solving skills is to practice!
4. A foolish consistency is the hobgoblin of little minds
5. Test your code, often and thoroughly
6. If it was hard to write, it is probably hard to read. Add a comment.
7. All input is evil, unless proven otherwise.
8. A function should do one thing.
9. Make sure your class does the right thing.