Lecture 9: Code and project management

PCL II, CL, UZH
April 27, 2016

Universität Zürich[UZH]

Outline

- Avoiding/dealing with bugs
  - Error/bug types
  - Code Testing/Unit testing
  - Debugging
- Complexity, profiling
- Source control

Errors

Universität Zürich[UZH]

- compile-time errors
  - Syntax errors
    - e.g. error in program code syntax
    - Mostly typos
- runtime errors
  - division by 0
  - file does not exist
- logical errors
  - the program does something unexpected
  - does not conform to the specifications
    - e.g. infinite loop, wrong output

Code Testing

Universität Zürich[UZH]

- Automatic
- (Fast)
- Tests functions in isolation
  - break down functionality
  - better understanding what is happening
- “White-box” testing/Structural testing
  - vs. “black-box” testing/functional testing
  - at unit, integration and system levels
- “Test-Driven Development”/”Code-Driven Testing”
  - implementation goals are defined through tests
  - short development cycles

Unit Testing

Universität Zürich[UZH]

→ **Check for expected functionality**

- Check a piece of code for correctness
  - Prove that bit of code does what you want (for all possible cases)

- Can check any manageable piece of code
  - Modules
  - Functions
  - Regular expressions
  - …

Unit Testing

Universität Zürich[UZH]

→ **Check for expected functionality**

- Define unit tests before/during writing code
- Provide a list of
  - (representative) input parameters
  - expected output values

- Python:
  - assert statement
  - doctest module
  - unittest module

Unit Testing

Universität Zürich[UZH]

```
def levenshtein(str1, str2):
    ...

if __name__ == "__main__":
    TESTS = [
        #string1, string2, expected lev. distance
        ("", "abccba", 6),
        ("abcd", "abce", 1),
        ("abcd", "abdc", 2),
        ("apple", "orange", 5)]

    #apply every test
    for (s1, s2, expectedVal) in TESTS:
        assert levenshtein(s1, s2) == expectedVal
```

Unit Testing

Universität Zürich[UZH]

```
def levenshtein(str1, str2):
    ...

if __name__ == "__main__":
    TESTS = [
        #string1, string2, expected lev. distance
        ("", "abccba", 6),
        ("abcd", "abce", 1),
        ("abcd", "abdc", 2),
        ("apple", "orange", 5)]

    #apply every test
    for (s1, s2, expectedVal) in TESTS:
        assert levenshtein(s1, s2) == expectedVal
```

Python's `doctest` module

- Looks for “Interactive” elements in docstring
- Checks that tests work exactly as shown
- Not very sophisticated, primarily for testing the documentations

Python's doctest module

Universität Zürich[UZH]

```
import doctest
class LevDist():
    def compute(self, a, b):
        '''
        >>> lev.compute("apple", "orange")
        3 #expected lev. distance
        >>> lev.compute("apple", "apple")
        0
         ...
         '''
        ......
if __name__ == "__main__":
    doctest.testmod(extraglobs={'lev': LevDist()})
```

Python's `unittest` module

- Suitable for more complex tasks
- Based on JUnit (framework for testing Java code)
- Test cases are defined outside of program code
  - Clean division between program/program documentation and tests
- Basic components to define:
  - single test cases
  - test contexts (or fixtures)
    - required preparations before the test and cleaning up after it

Python's `unittest` module


```
class LevDist():
    def compute(self, a, b):
        m = []
        ...
```

Python's `unittest` module


```
class LevDist():
    def compute(self, a, b):
        m = []
        ...

class TestLevDist(unittest.TestCase):
    def setUp(self):
        self.levDist = LevDist()

    def test_apples_oranges(self):
        actual = self.levDist.compute("apples", "oranges")
        expected = 5
        self.assertEqual(expected, actual)
```

Python's `unittest` module

Universität Zürich[UZH]

```
import unittest

class LevDist():
    def compute(self, a, b):
        m = []
        ...

class TestLevDist(unittest.TestCase):
    def setUp(self):
        self.levDist = LevDist()

    def test_apples_oranges(self):
        actual = self.levDist.compute("apples", "oranges")
        expected = 5
        self.assertEqual(expected, actual)

unittest.main()
```

Python's `unittest` module

Universität Zürich[UZH]

```
import unittest
import re

eMailRe = r'[a-z0-9]+@[a-z0-9]+\.[a-z0-9]+'

class TestEmailRegExp(unittest.TestCase):
    def test_match(self):
        self.assertTrue(re.match(eMailRe, 'john@smith.com'))

    def test_dont_match(self):
        self.assertFalse(re.match(eMailRe, 'this@'))

unittest.main()
```

Debugging

Universität Zürich[UZH]

- Try running *levdist.py* – it reports (some) numbers for inputs
- Try running *testlevdist.py* – both tests fail
- *levdist.py* does not report any run-time or compile-time errors => logical error

Q: How to find logical errors?

Debugging

Universität Zürich[UZH]

Finding errors in the program code:

- compiling and running the code in your head
  - go through the code
    - understand what is happening
    - understand what is going wrong
    - (use google)
- tracing
- step-wise execution

Tracing

Universität Zürich[UZH]

- add commands to
  - display internal variable values
  - report reached program steps

Tracing

Universität Zürich[UZH]

```
def average(numList):
    sum = 0

    for num in numList:
        sum = sum + num

    return sum / len(numList)


print average([1, 3, 7]) # should be 3.6666...
                         # is actually 3
```

Tracing

Universität Zürich[UZH]

```
def average(numList):
    sum = 0
    print "initial sum:", sum

    for num in numList:
        print "adding", num, "to total sum"
        sum = sum + num
        print "updated total sum:", sum

    total = len(numList)
    result = sum / total
    print "total sum:", sum, ", #items:", total,
        ", result:", result
    return result


print average([1, 3, 7])
```

Tracing

Universität Zürich[UZH]

```
def average(numList):
    sum = 0.0
    print "initial sum:", sum

    for num in numList:
        print "adding", num, "to total sum"
        sum = sum + num
        print "updated total sum:", sum

    total = len(numList)
    result = sum / total
    print "total sum:", sum, ", #items:", total,
        ", result:", result
    return result


print average([1, 3, 7])
```

Tracing

Universität Zürich[UZH]

- Python `trace` module
  - Can be used directly from command line
    - e.g. `python -m trace --trackcalls levdist.py`
  - Track statement execution:
    - `--trace`: display executed lines
    - `--listfuncs`: display executed function names
    - `--trackcalls`: display calling relationships

```
*** levdist.py ***

    levdist.<module> -> levdist.LevDist

    levdist.<module> -> levdist.LevDist.compute
```

Step-wise execution

Universität Zürich[UZH]

- run small pieces of code at a time
- execute one function or go inside them
- add breaks and skip to them directly
  - on line numbers
  - in the beginning of functions
  - also conditional breaks
- inspect any variables etc. at any step
  - advantage over tracing: no need to modify code and re-run the program

Step-wise execution

Universität Zürich[UZH]

- pdb - the python debugger
- Works on command-line
- usage: `python -m pdb myprogram.py`
- commands:
  - n(ext)
  - s(tep)
  - c(ontinue)
  - locals()
  - b(reak) line_nr
  - b(reak) function_name [, condition]

Step-wise execution

Universität Zürich[UZH]

- Other way: inside a GUI IDE
- GUI means to step, add breaks and/or conditions
- Possible IDEs:
  - https://wiki.python.org/moin/PythonDebuggingTools#IDEs_with_Debug_Capabilities
  - E.g. PyCharm, PyDev, etc.

Profiling

Universität Zürich[UZH]

→ **Check how fast the code runs**

Counting hapax legomena

Universität Zürich[UZH]

```
hapaxCount = 0

for token in tokenList:
    tokenCount = 0

    for token2 in tokenList:
        if token2 == token:
            tokenCount += 1

    if tokenCount == 1:
        hapaxCount += 1
```

Counting hapax legomena

Universität Zürich[UZH]

```
for token in tokenList:
    tokenCount = 0

    for token2 in tokenList:
        if token2 == token:
            tokenCount += 1

    if tokenCount == 1:
        hapaxCount += 1
```

- python hapax.py data.txt 300: 0.186s
- python hapax.py data.txt 400: 0.316s
- python hapax.py data.txt 500: 0.545s
- python hapax.py data.txt 600: 0.816s

Counting hapax legomena

Universität Zürich[UZH]

```
frequencies = defaultdict(int)

for token in tokenList:
    frequencies[token] += 1

for token in frequencies:
    if frequencies[token] == 1:
        hapaxCount += 1
```

Counting hapax legomena

Universität Zürich[UZH]

```
frequencies = defaultdict(int)

for token in tokenList:
    frequencies[token] += 1

for token in frequencies:
    if frequencies[token] == 1:
        hapaxCount += 1
```

- python fhapax.py data.txt 1000: 0.033s
- python fhapax.py data.txt 2000: 0.039s
- python fhapax.py data.txt 4000: 0.049s
- python fhapax.py data.txt 8000: 0.077s

Profiling

Universität Zürich[UZH]

→ **Monitoring and "debugging" complexity**

- insert time measurements into the code
  - `time.clock()` / `time.time()` / `time.localtime()` / `time.gmtime()`
- report time or intervals for different parts of code
- understand why the code is running slow
- compare different approaches/methods
- Similar to tracing (but different purpose)

timeit module

Universität Zürich[UZH]

```
import timeit
code = """for token in tokenList:
    tokenCount = 0
    ..."""

timeit.timeit(code, number=100)
```

- run code a given number of times
- report the average time of execution

cProfile

Universität Zürich[UZH]

- measures time per every code line and function
- reports that in a strange but eventually useful way
- e.g.

```
python -m cProfile hapax.py data.txt 400
```

- see http://docs.python.org/library/profile.html

- Tools for better visualisation:
  - runsnakerun
  - pyprof2calltree

cProfile

python -m cProfile hapax.py data.txt 400

```
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.005    0.005 hapax.py:10(loadData)
     3625    0.001    0.000    0.321    0.000 hapax.py:19(countToken)
        1    0.001    0.001    0.327    0.327 hapax.py:29(main)
    [...]
```

python -m cProfile fhapax.py data.txt 400

```
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.004    0.004 fhapax.py:10(loadData)
        1    0.001    0.001    0.005    0.005 fhapax.py:19(main)
    [...]
```

cProfile: runsnakerun

Source control

- allows code and project management
- tracks all changes to a document/project
- developers can work concurrently

- **git**: free and open source distributed version control system
- **gitHub**: web-based git repository
- **bitbucket**
- **gitlab**

Source control

- *git* main commands:
  - `clone`
  - `pull`
  - `add`
  - `commit`
  - `push`
  - `status`
  - `branch`

- `gitk`
  - `git-GUI`

Source control - useful links

- Atlassian git tutorial
  - https://www.atlassian.com/git/tutorials/
- git - Der einfache Einstieg (in German)
  - *https://rogerdudler.github.io/git-guide/index.de.html*
- Tutorialspoint tutorial
  - *http://www.tutorialspoint.com/git/*
- Plus: videotutorials, codeacademy etc.

Conclusions

Universität Zürich[UZH]

- Making sure the code does what you expect
  - testing/unit tests
- Finding bugs
  - tracing / step-wise execution -- debugging
- Finding inefficient solutions
  - time measurements / profiling

Questions?

Lecture 9: Code and project management

PCL II, CL, UZH
April 22, 2015

Universität Zürich[UZH]