

# Test, Debug, Profile

Nicola Chiapolini

Physik-Institut  
University of Zurich

2021-06-29

Based on a talk by Pietro Berkes



This work is licensed under the [Creative Commons Attribution-ShareAlike 3.0 License](https://creativecommons.org/licenses/by-sa/3.0/).

# Scientific Programming

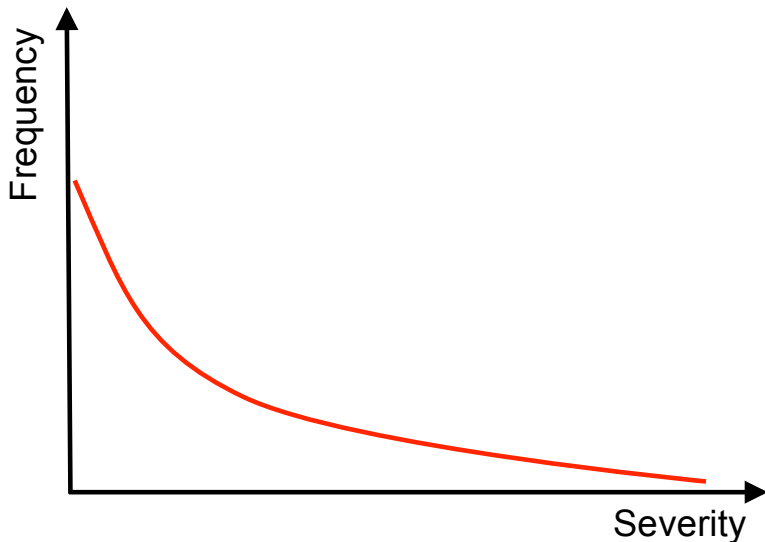
## Goal

- ▶ allow exploring many different approaches
- ▶ allow frequent changes and adjustments
- ▶ produce correct and reproducible results

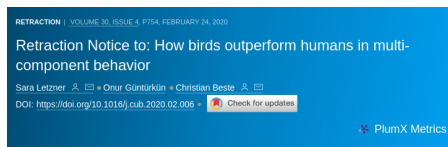
## Requirements

- ▶ bugs must be noticed
- ▶ code can be modify easily
- ▶ others can run code too
- ▶ scientist's time is used optimally

# Effect of Software Errors



# Effect of Software Errors: Retractions



(Current Biology 27, R996–R998; September 25, 2017)

In our Correspondence, we reported evidence leading us to conclude that pigeons are on par with humans when tested with a behavioral task that demands simultaneous processing resources; in particular, we claimed that pigeons show faster responses than humans when sub-tasks are separated with a short STOP–CHANGE delay of 300 ms—the “SCD 300” condition (time advantage of 200 ms). We have subsequently discovered, however, that the MATLAB script that was used for the analysis of reaction times in the pigeon paradigm was wrongly indexed.

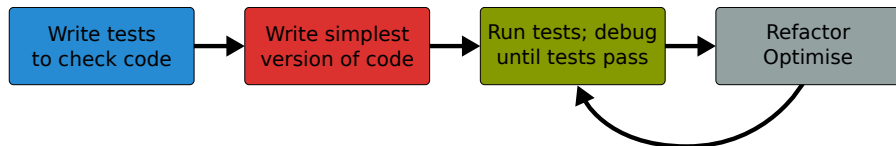
## Retraction Watch

### Error in one line of code sinks cancer study

without comments

Authors of a 2016 cancer paper have retracted it after finding an error in one line of code in the program used to calculate some of the results.

# Outline



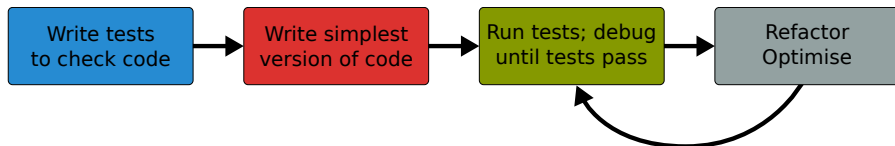
`unittest`  
`doctest`  
`coverage`

`pdb`

`timeit`  
`cProfile`  
`pstats`

- ▶ standard python tools
- ▶ ipython magic commands
- ▶ mostly command line

# Outline



`unittest`  
`doctest`  
`coverage`

`pdb`

`timeit`  
`cProfile`  
`pstats`

- ▶ standard python tools
- ▶ ipython magic commands
- ▶ mostly command line

# Testing

Something you do anyway.

- ▶ run code and see if it crashes
- ▶ check if output makes sense
- ▶ run code with trivial input
- ▶ ...

# Formal Testing

- ▶ important part of modern software development
- ▶ unittest and integration tests
- ▶ tests written in parallel with code
- ▶ tests run frequently/automatically
- ▶ generate reports and statistics

```
[...]
```

```
replace predefined histogram ... ok
```

```
add a legend; change line color of last histogram to red ... ok
```

```
put title and axis labels ... ok
```

```
-----  
Ran 18 tests in 5.118s
```

```
OK
```

```
GoodBye!
```



# Benefits

- ▶ only way to trust your code
- ▶ faster development
  - ▶ know where your bugs are
  - ▶ fixing bugs will not (re)introduce others
  - ▶ change code with out worrying about consistency
- ▶ encourages better code
- ▶ provides example/documentation

```
FAIL: test_result (test_fibonacci.FiboTest)
test 7th fibonacci number
```

```
-----
Traceback (most recent call last):
```

```
  File "test_fibonacci.py", line 18, in test_result
    self.assertEqual(result, expect)
```

```
AssertionError: 21 != 13
```

# An Example

```
def remove(thelist, entry):  
    """ remove entry object from list """  
    for idx, item in enumerate(thelist):  
        if entry is item:  
            del thelist[idx]  
            break  
    else:  
        raise ValueError("Entry not in the list")
```

Assume we find this code in an old library of ours.

# An Example

```
def remove(thelist, entry):  
    """ remove entry object from list """  
    thelist.remove(entry)
```

We prefer to keep it simple! Everything fine, right?

# An Example

```
def remove(thelist, entry):  
    """ remove entry object from list """  
    thelist.remove(entry)
```

```
ERROR: test_remove_array (__main__.RemoveTest)
```

```
-----  
Traceback (most recent call last):
```

```
  File "list_tests.py", line 19, in test_remove_array  
    lrm.remove(l, x)
```

```
  File ".../examples/list_removal.py", line 3, in remove  
    thelist.remove(entry)
```

```
ValueError: The truth value of an array with more than one  
element is ambiguous. Use a.any() or a.all()
```

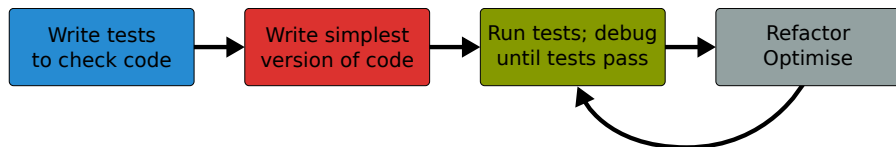
# Start Testing

At the beginning, testing feels weird:

1. It's obvious that this code works
2. The tests are longer than the code
3. The test code is a duplicate of the real code

→ it might take a while to get used to testing,  
but it will pay off quiet rapidly.

# Outline



`unittest`  
`doctest`  
`coverage`

`pdb`

`timeit`  
`cProfile`  
`pstats`

- ▶ standard python tools
- ▶ ipython magic commands
- ▶ mostly command line

# unittest

- ▶ library for unittests
- ▶ part of standard python
- ▶ at the level of other modern tools

## Alternatives

- ▶ `pytest`

# Anatomy of a TestCase

```
import unittest

class DemoTests(unittest.TestCase):

    def test_boolean(self):
        """ tests start with 'test' """
        self.assertTrue(True)
        self.assertFalse(False)

    def test_add(self):
        """ docstring can be printed """
        self.assertEqual(2+1, 3)

if __name__ == "__main__":
    """ execute all tests in module """
    unittest.main()
```



# Summary on Anatomy

## Test Cases

- ▶ are subclass of `unittest.TestCase`
- ▶ group test units

## Test Units

- ▶ methods, whose names **start** with `test`
- ▶ should cover **one** aspect
- ▶ check behaviour with "assertions"
- ▶ rise exception if assertion fails

# Running Tests

Option 1 execute all test units in all test cases of this file

```
if __name__ == "__main__":  
    unittest.main(verbosity=1)  
  
python3 test_module.py
```

Option 2 Execute all tests in one file

```
python3 -m unittest [-v] test_module
```

Option 3 Discover all tests in all submodules

```
python3 -m unittest discover [-v]
```

# TestCase.assertSomething

## ► check boolean value

```
assertTrue('Hi'.islower())           # fail
assertFalse('Hi'.islower())          # pass
```

## ► check equality

```
assertEqual(2+1, 3)                   # pass
""" assertEqual can compare all sorts of objects """
assertEqual([2]+[1], [2, 1])          # pass
```

## ► check numbers are close

```
from math import sqrt, pi
assertAlmostEqual(sqrt(2), 1.414, places=3) # pass
""" values are rounded, not truncated """
assertAlmostEqual(pi, 3.141, 3)           # fail
assertAlmostEqual(pi, 3.142, 3)           # pass
```

# TestCase.assertRaises

- ▶ most convenient with context managers

```
with self.assertRaises(ErrorType):  
    do_something()  
    do_some_more()
```

- ▶ Important: use most specific exception class

```
bad_file = "inexistent"  
with self.assertRaises(FileNotFoundError):    # raises NameError  
    open(bad_fil, 'r')  
  
with self.assertRaises(Exception):  
    open(bad_fil, 'r')                        # pass
```

# TestCase.assertMoreThings

```
assertGreater(a, b)
```

```
assertLess(a, b)
```

```
assertRegex(text, regexp)
```

```
assertIn(value, sequence)
```

```
assertIsNone(value)
```

```
assertIsInstance(my_object, class)
```

```
assertCountEqual(actual, expected)
```

complete list at

<https://docs.python.org/3/library/unittest.html>

# TestCase.assertNotSomething

Most of the `assert` methods have a `Not` version

```
assertEqual  
assertNotEqual
```

```
assertAlmostEqual  
assertNotAlmostEqual
```

```
assertIsNone  
assertIsNotNone
```

# Testing with numpy

numpy arrays have to be compared elementwise

```
class SpecialCases(unittest.TestCase):  
    def test_numpy(self):  
        a = numpy.array([1, 2])  
        b = numpy.array([1, 2])  
        self.assertEqual(a, b)
```

```
=====
```

```
ERROR: test_numpy (__main__.SpecialCases)
```

```
-----
```

```
Traceback (most recent call last):
```

```
[..]
```

```
ValueError: The truth value of an array with more than one  
element is ambiguous. Use a.any() or a.all()
```

# numpy.testing

- ▶ defines appropriate function

```
numpy.testing.assert_array_equal(x, y)  
numpy.testing.assert_array_almost_equal(x, y, decimal=6)
```

- ▶ use numpy functions for more complex tests

```
numpy.all(x)           # True if all elements of x are true  
numpy.any(x)           # True if any of the elements of x is true  
numpy.allclose(x, y)   # True if element-wise close
```

## Example

```
""" test that all elements of x are between 0 and 1 """  
assertTrue(all(logical_and(x > 0.0, x < 1.0))
```



# Strategies for Testing

- ▶ What does a good test look like?
- ▶ What should I test?
- ▶ What is special for scientific code?

# What does a good test look like?

**Given** put system in right state

- ▶ create objects, initialise parameters, ...
- ▶ define expected result

**When** action(s) of the test

- ▶ one or two lines of code

**Then** compare result with expectation

- ▶ set of assertions

# What does a good test look like? – Example

```
import unittest

class LowerTestCase(unittest.TestCase):

    def test_lower(self):
        # given
        string = 'HeLlO wOrld'
        expected = 'hello world'

        # when
        result = string.lower()

        # then
        self.assertEqual(result, expected)
```

# What should I test?

- ▶ simple, general case

```
string = 'HeLlO wOrld'
```

- ▶ corner cases

```
string = ''  
string = 'hello'  
string = '1+2=3'
```

often involves design decisions

- ▶ any exception you raise explicitly
- ▶ any special behaviour you rely on

# Reduce Overhead 1: Loops

```
import unittest

class LowerTestCase(unittest.TestCase):

    def test_lower(self):
        # given
        # Each test case is a tuple (input, expected)
        test_cases = [('HeLlO wOrld', 'hello world'),
                      ('hi', 'hi'),
                      ('123 ([?', '123 ([?'),
                      ('', '')]
        for string, expected in test_cases:
            # run several subtests
            # when
            output = string.lower()
            # then
            self.assertEqual(output, expected)
```

# Reduce Overhead 1: Subtests

```
import unittest

class LowerTestCase(unittest.TestCase):

    def test_lower(self):
        # given
        # Each test case is a tuple (input, expected)
        test_cases = [('HeLlO wOrld', 'hello world'),
                      ('hi', 'hi'),
                      ('123 ([?', '123 ([?'),
                      ('', '')]
        for string, expected in test_cases:
            with self.subTest(config = string):
                # when
                output = string.lower()
                # then
                self.assertEqual(output, expected)
```

## Reduce Overhead 2: Fixtures

- ▶ allow to use same setup/cleanup for several tests
- ▶ useful to
  - ▶ create data set at runtime
  - ▶ load data from file or database
  - ▶ create mock objects
- ▶ available for test case as well as test unit

```
class FixtureTestCase(unittest.TestCase):  
  
    @classmethod  
    def setUpClass(cls):          # called at start of TestCase  
  
    def setUp(self):              # called before each test  
  
    def tearDown(self):           # called at end of each test
```

# What is special for scientific code?

often deterministic test cases very limited/impossible

## Numerical Fuzzing

- ▶ generate random input (print random seed)
- ▶ still need to know what to expect

## Know What You Expect

- ▶ use inverse function
- ▶ generate data from model
- ▶ add noise to known solutions
- ▶ test general routine with specific ones
- ▶ test optimised algorithm with brute-force approach



# Automated Fuzzing: Hypothesis (not in standard library)

`hypothesis` generates test inputs according to given properties.

```
import unittest, numpy
from hypothesis import given, strategies as st

class SumTestCase(unittest.TestCase):

    @given(st.lists(st.integers(), min_size=2, max_size=2))
    def test_sum(self, vals):
        self.assertEqual(vals[0]+vals[1], numpy.sum(vals))
```

## Why?

- ▶ cover large search-space (default 100 inputs)
- ▶ good for finding edge cases
- ▶ less manual work

# Test Driven Development (TDD)

## Tests First

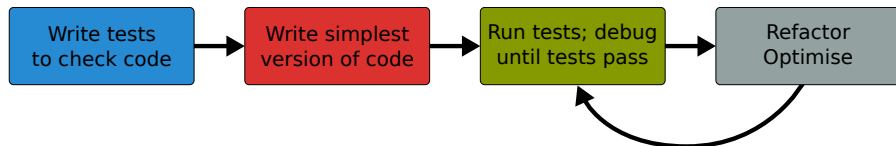
- ▶ choose next feature
- ▶ write test(s) for feature
- ▶ write simplest code

## Benefits

- ▶ forced to think about design before coding
- ▶ code is decoupled and easier to maintain
- ▶ you will notice bugs

# DEMO

# Outline



`unittest`  
`doctest`  
`coverage`

`pdb`

`timeit`  
`cProfile`  
`pstats`

- ▶ standard python tools
- ▶ ipython magic commands
- ▶ mostly command line

# doctest

- ▶ poor man's unittest
- ▶ ensure docstrings are up-to-date

```
def add(a,b):  
    """ add two numbers
```

*Example*

-----

```
>>> add(40,2)  
42
```

```
"""
```

```
return a+b
```

```
python3 -m doctest [-v] my_doctest.py
```

Trying:

```
    add(40,2)
```

Expecting:

```
    42
```

ok

1 items had no tests:

```
    my_doctest
```

1 items passed all tests:

```
    1 tests in my_doctest.add
```

1 tests in 2 items.

1 passed and 0 failed.

Test passed.

# Code Coverage

- ▶ it's easy to leave part untested
  - ▶ features activated by keyword
  - ▶ code to handle exception
- ▶ coverage tools track the lines executed

## coverage.py

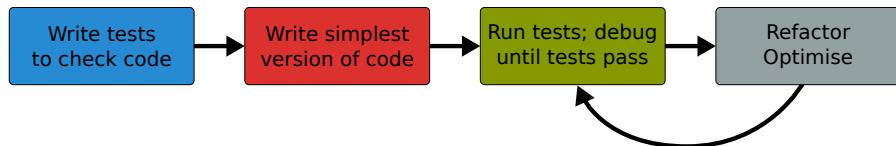
- ▶ python script
- ▶ produces text and HTML reports

```
python3 -m coverage run test_file.py
python3 -m coverage report [-m] [--omit="/usr*"]
```

- ▶ not in standard library  
get from <https://coverage.readthedocs.io/en/latest/>

# DEMO

# Outline



`unittest`  
`doctest`  
`coverage`

`pdb`

`timeit`  
`cProfile`  
`pstats`

- ▶ standard python tools
- ▶ ipython magic commands
- ▶ mostly command line



# Debugging

- ▶ use tests to avoid bugs and limit „search space”
- ▶ avoid `print` statements
- ▶ use debugger

## pdb – the Python debugger

- ▶ command line based
- ▶ opens an interactive shell
- ▶ allows to
  - ▶ stop execution anywhere in your code
  - ▶ execute code step by step
  - ▶ examine and change variables
  - ▶ examine call stack

# Entering pdb

- ▶ enter at start of file

```
python3 -m pdb myscript.py
```

- ▶ enter at statement/function

```
import pdb  
# your code here  
pdb.run(expression_string)
```

- ▶ enter at point in code

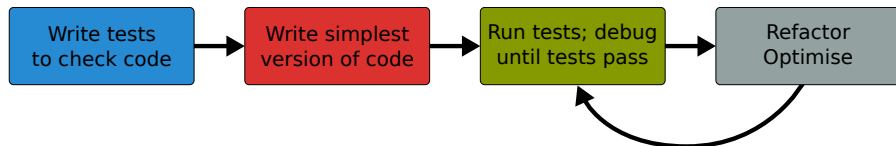
```
# some code here  
# the debugger starts here  
import pdb; pdb.set_trace()  
# rest of the code
```

- ▶ from ipython

```
%pdb      # enter pdb on exception  
%debug    # enter pdb after exception
```

# DEMO

# Outline



`unittest`  
`doctest`  
`coverage`

`pdb`

`timeit`  
`cProfile`  
`pstats`

- ▶ standard python tools
- ▶ ipython magic commands
- ▶ mostly command line

# Optimising

1. don't rush into optimisation
2. identify time-consuming parts of code
3. only optimise those parts
4. keep running tests
5. stop as soon as possible

# Optimising

1. don't rush into optimisation
2. identify time-consuming parts of code
3. only optimise those parts
4. keep running tests
5. stop as soon as possible

# timeit

- ▶ precise timing for function/expression
- ▶ test different versions of a code block
- ▶ easiest with ipython's magic command

## `a**2` or `pow(a,2)`?

```
In [1]: a = 43563
```

```
In [2]: %timeit pow(a,2)
```

```
10000000 loops, best of 5: 268 ns per loop
```

```
In [3]: %timeit a**2
```

```
10000000 loops, best of 5: 209 ns per loop
```

# cProfile & Pstats

**Profiling** identify where most time is spent

**cProfile** standard python module for profiling

**pstats** tool to look at profiling data

## ► run cProfile

```
python3 -m cProfile [-s cumtime] myscript.py  
python3 -m cProfile [-o myscript.prof] myscript.py
```

## ► analyse output from shell

```
python3 -m pstats myscript.prof
```

```
stats      # print statistics  
sort       # change sort order  
callers    # print callers  
callees    # print callees
```



# Non-Standard Tools

- ▶ **pyprof2calltree** and **kcachegrind**: open cProfile output in GUI

```
python3 -m cProfile -o myscript.prof myscript.py
. ~/school_venv/bin/activate # on school laptops activate venv
pyprof2calltree -i myscript.prof -k
```

- ▶ **pprofile**: line-granularity profiler

```
pprofile3 myscript.py

pprofile3 -f callgrind -o myscript.prof myscript.py
kcachegrind myscript.prof
```

# DEMO

# Final Thoughts

- ▶ testing, debugging and profiling can help you a lot
- ▶ using the right tools makes life a lot easier
- ▶ python comes with good tools included
- ▶ it's as easy as it gets – there are no excuses