6th Workshop on Collaborative Scientific Software Development and Management of Open Source Scientific Packages

Introduction to code testing

Alessandro Corbetta

Post-doctoral researcher – Department of Applied Physics
Eindhoven University of Technology, NL

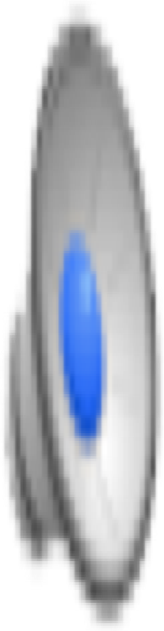
http://corbetta.phys.tue.nl

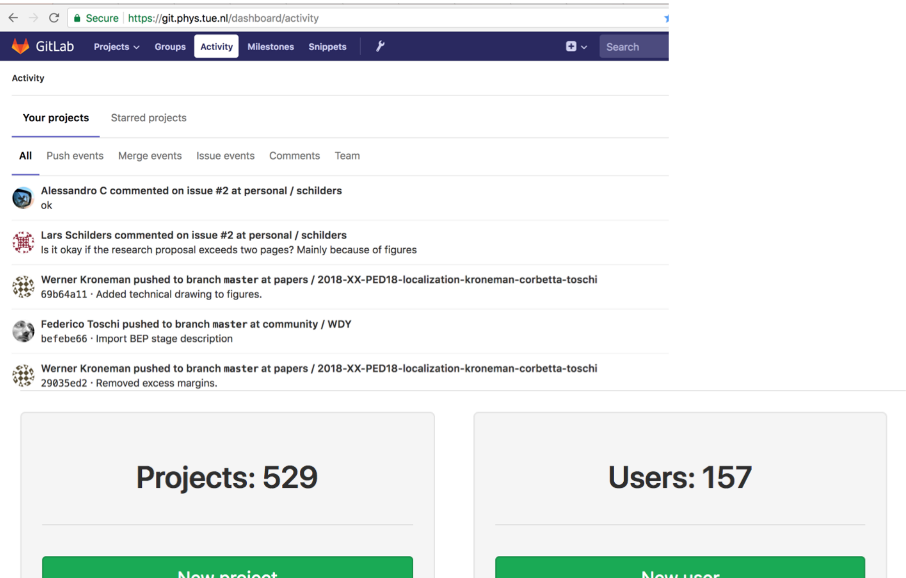
TU/e Technische Universiteit Eindhoven University of Technology

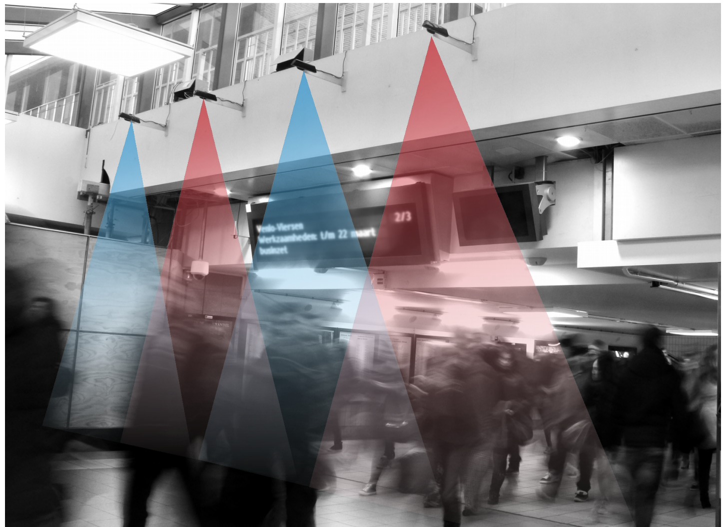
ICTP The Abdus Salam International Centre for Theoretical Physics

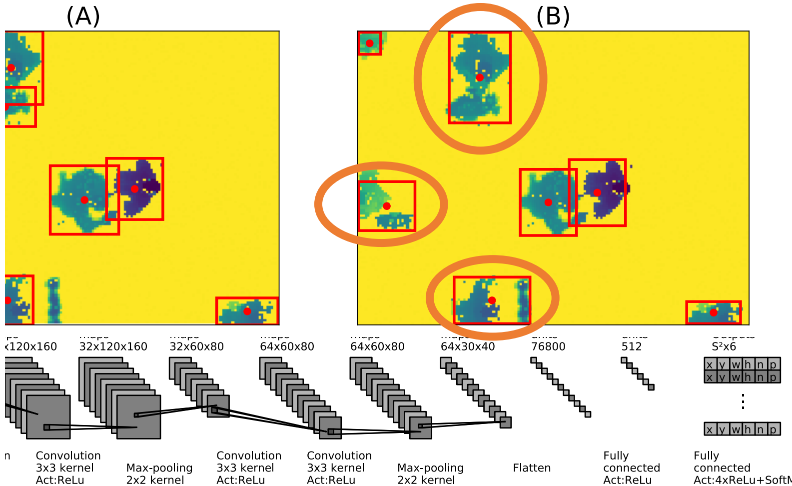
Alessandro Corbetta

- Post-doc researcher in Applied Physics (TU Eindhoven)
  - Study human crowds dynamics as fluid mechanics
  - Deep learning for computer vision & physics
  - PhD in Applied Mathematics, PhD in Structural Engineering
- Admin of a Git(Lab) server since 2014
  Supported/Designed most of project testing/CI









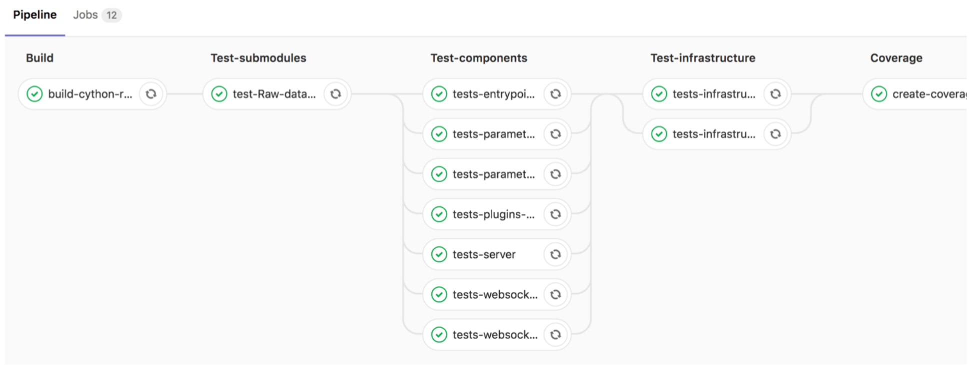
My Lectures

Software testing, versioning, quality control

Testing intro
(now)

Git and version control
(Saturday)

Testing suites/CI
/conversational development
(Monday)



This afternoon:

Floating point arithmetic
(exercises will connect with testing)

6th Workshop on Collaborative Scientific Software Development and Management of Open Source Scientific Packages

Introduction to code testing

Alessandro Corbetta

Post-doctoral researcher – Department of Applied Physics
Eindhoven University of Technology, NL

http://corbetta.phys.tue.nl

Technische Universiteit Eindhoven University of Technology

The Abdus Salam International Centre for Theoretical Physics

Testing: main question

1. Does my code **work**?

   (…broad question…)

Testing: main question

1. Does my code **work**?
   (…broad question…)

..as in..

2. Does my code have the **expected features** and **functionalities**?
   [e.g. “client” request; in science: ”do I match the analytic solution?”]

3. Does my code **still work**?
   [e.g. after a modification of myself, **collaborators**,…]

This lecture

- Testing: introduction to the concept
  - Contemporary testing as in the scientific method
  - Scales of testing
  - Contemporary vs. traditional testing
  - Unit testing heuristics/best practices
- Testing in python
  - Naïve approach
  - `nosetests`
  - More advanced nose (fixture options, coverage reports)

Contemporary testing anatomy: Key idea

We `ASSERT` that our software satisfies a given requirement

The test is passed if the assertion is satisfied; it fails otherwise

The validity of our assertions is checked through programs and scripts.

In general:

1 Test = piece of software that checks **1** `ASSERT`

Scientific method analogy

Scientific method:

We run experiments to invalidate
(or “prove/hinting correctness of”) an hypothesis.

A test is like an experiment

Functional testing paradigm:

Arrange → preparation of e.g. script that checks for feature

Act → run the test

`ASSERT` → if fails an issue is found

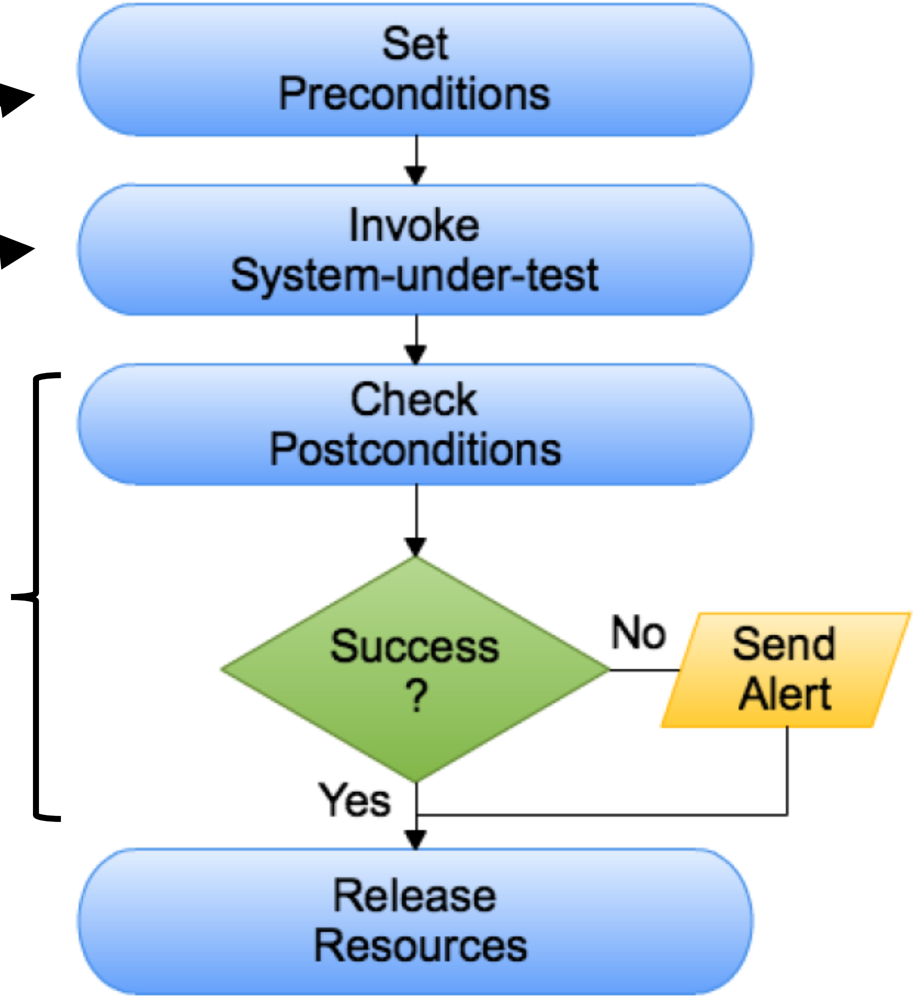
Scientific method analogy

Functional testing paradigm:

Arrange

Act

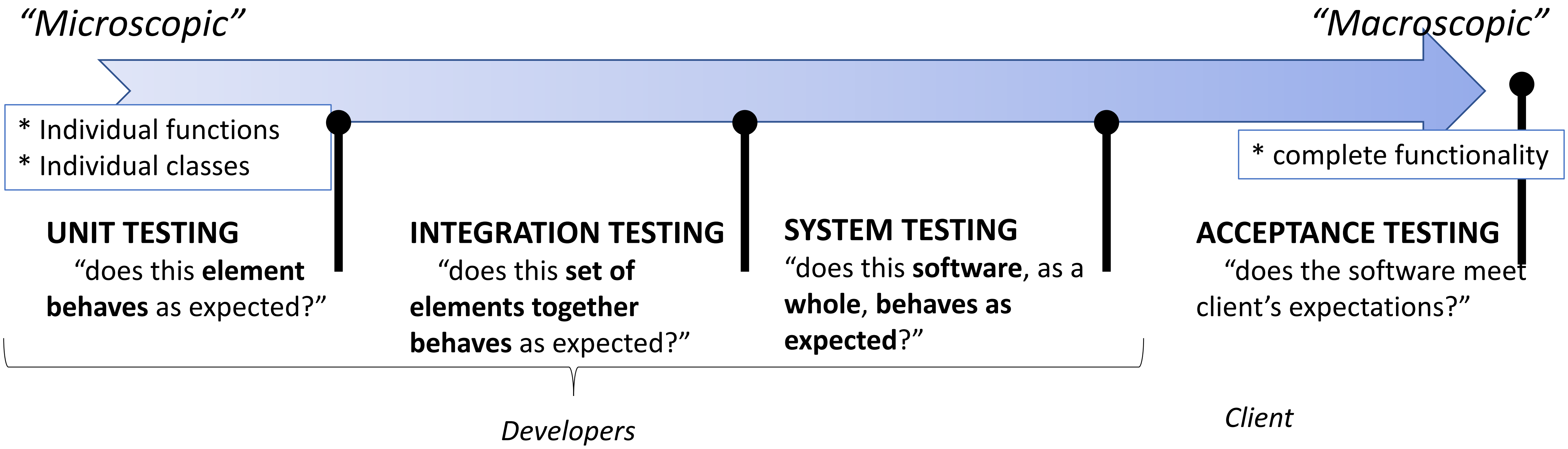
ASSERT



Scales of testing

Good modern software comes as a collection of <u>weakly</u> coupled modules operated together to deliver a result.

Testing follows these scales



In other words

- **Unit test:** if fails -> a piece of your code needs to be fixed.
- **Integration test:** if fails -> pieces of your application are not working *together* as expected.
- **System test:** if fails -> it tells that the application is not working as expected
- **Acceptance test:** if fails -> the application is not doing what the customer expects it to do.

- **Regression test:** when it fails, it tells you that the application no longer behaves the way it used to.

answered Oct 7 '11 at 3:44

Mathias
10.3k ● 6 ● 41 ● 88

”Contemporary” vs. “traditional” testing

Issues emerge with **usage** or In **production**

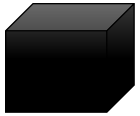
- **Traditional approach & pitfalls**
  - System tested as a whole
  - High complexity (how do I build a proper test?)
  - Hard to test individual components
  - Hard to find sources of errors
  - Testing done through print statements/debuggers/script

Issues emerge In **development**

- **Contemporary/Unit testing**
  - Lower complexity
  - Compliance to past requirements easily checked by module
  - To the limit: **Test Driven Development** (TDD – Exercise this afternoon)
    - As a requirement is identified, tests are written before the implementation

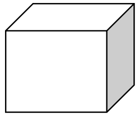
Black Box vs. White box testing

Black box



- Tests functionality **without** knowledge of internal structure
- Cost effective: High -> can be authored by non-developers (no biases though!)
- Efficacy: low -> relies on tester's luck about triggering all internals

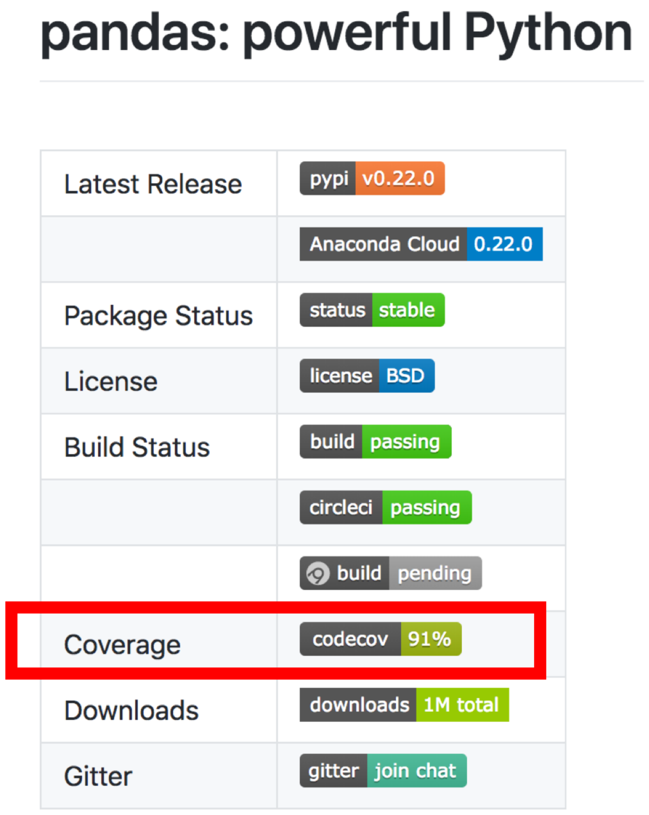
White box



- Test **with** knowledge of internals
- Cost effective: Low -> Must be written by developers
- Efficacy: high -> all internals can be triggered

White box testing "quality" metrics: coverage

- If parts of our code are not tested (i.e. not covered by a test) bugs have higher chance to reach production
- **Line coverage**
  - **Percentage of lines** that are covered by at least one test (an if condition might be unsatisfied in all tests, thus the if-true branch remains always untested).
- **Branch coverage**
  - 100% line coverage might still leave many branches (that grow combinatorically) unexplored. Branch coverage counts how many of all branches are "seen" by at least one test.



Unit testing heuristics 1

1. Create test when object design is complete

   In TDD write test when interface is defined

2. **Design components that are testable**

   Make life of a tester easy: e.g. allow swappable mocks

3. Testing time slows down development

   make quick tests (at run time)
   make tests that are no-brainer to run

4. Develop tests using effective number of testing cases

   Heads up: generally combinatorial explosion of inputs, cannot be matched by as many `ASSERT`

   Selecting relevant & (all) edge cases -> more practice than theory

Unit testing heuristics 2

5. If possible compare e.g. with analytic solution or even slower-but-working versions of the same algorithms

   (Model-based testing)

6. In computing: knowing about internal computing mechanisms to make relevant tests
   1. **respect computer arithmetic**
   2. **Avoid non-determinism/fix seed in testing**
7. **REM: a failed test means a bug is introduced, not the other way around!**

8. **Best practice: Every new bug -> new test**

   against future regressions (e.g. from rollbacks)

Unit testing in Python: naïve way

```
$ python tests.py

     import my_package

     def test_1():
          <preparation>
          assert condition_f_1

     def test_2():
          <preparation>
          assert condition_f_2


     if __name__ == '__main__':
          test_1()
          test_2()
```

Unit testing in Python: naïve way

```
$ python tests.py

    import my_package

    def test_1():
        <preparation>
        assert condition_f_1

    def test_2():
        <preparation>
        assert condition_f_2


    if __name__ == '__main__':
        test_1()
        test_2()
```

Issues:

- Not immediate
  - *cannot be run without thought*
- Not scalable
  - e.g. need to add calls under the if shield
- Output does not come as a simple report
- Running individual tests requires work
- …

Common issues!

Common solution:

using testing frameworks..

Unit testing in Python

- Python comes with packages helping unit testing
  - Note: Unit testing libraries exist for *any* programming language.
  - Same concepts as here apply (compiler might be needed)

- E.g. **`Nosetest`**, `PyTest`, `doctest`

Nosetest

- `nose` runs tests in files/directories under the `cwd`

  (*(?:\b|_)[Tt]est*)

  - whose names include "`test`" or "`Test`" at a word boundary
    - (like "test_this" or "functional_test" or "TestClass" but not "libtest").
  - Test output includes captured `stdout` output from failing tests, for easy debugging.

- Returns a report of pass/failures with different possible levels of verbosity
- Can return coverage report

Nosetest - installation

- via pip or easy_install

```
$ easy_install nose
$ pip install nose
```

Test successful installation:

```
acorbe@Alessandros-MacBook-Pro ~  $ pwd
/Users/acorbe
acorbe@Alessandros-MacBook-Pro ~  $ nosetests

----------------------------------------------------------------------
Ran 0 tests in 0.004s

OK
acorbe@Alessandros-MacBook-Pro ~  $
```

Call nosetests in an empty folder.
The program terminates successfully with no test run.

Nosetest - examples

- Testing my_sum.py (basics)

```python
from __future__ import print_function
import sys

def sum_foo(a,b):
    return a + b

def converter(x):
    return float(x)

def main():
    a = converter(sys.argv[1])
    b = converter(sys.argv[2])
    ret = sum_foo(a,b)
    print(ret)
    return ret


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

1 — acorbe-gpu@crowdflow2: ~/workspace/ML-for-turbulence/1D/goy_working/GOY-m

```
acorbe@Alessandros-MacBook-Pro 1  [master] $ python my_sum.py 3 4
7.0
acorbe@Alessandros-MacBook-Pro 1  [master] $
```

Nosetest - examples

- Testing my_sum.py (basics)

```
from __future__ import print_function
import sys

def sum_foo(a,b):
    return a + b

def converter(x):
    return float(x)

def main():
    a = converter(sys.argv[1])
    b = converter(sys.argv[2])
    ret = sum_foo(a,b)
    print(ret)
    return ret


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

1 — acorbe-gpu@crowdflow2: ~/workspace/ML-for-turbulenc

```
acorbe@Alessandros-MacBook-Pro 1  [master] $ tree
.
├── my_sum.py
├── my_sum.pyc
└── tests
    ├── __init__.py
    ├── __init__.pyc
    ├── tests.py
    └── tests.pyc

1 directory, 6 files
acorbe@Alessandros-MacBook-Pro 1  [master] $
```

Nosetest - examples

- Testing my_sum.py (basics)

```python
from __future__ import print_function
import sys

def sum_foo(a,b):
    return a + b

def converter(x):
    return float(x)

def main():
    a = converter(sys.argv[1])
    b = converter(sys.argv[2])
    ret = sum_foo(a,b)
    print(ret)
    return ret


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

tests/test.py

```python
import my_sum

def test_sum_function():
    "tests that the summation operation is performed correctly"
    a = 2
    b = 3
    assert my_sum.sum_foo(a,b) == a + b

def test_conversion():
    "tests that the command-line number converter performs properly"
    a = '6'
    assert my_sum.converter(a) == 6

def test_invalid_conversion():
    "tests that the command-line number converter performs properly"
    a = 'a'
    assert my_sum.converter(a) == 6
```

Nosetest - examples

- Testing my_sum.py (basics)

```
from __future__ import print_function
import sys

def sum_foo(a,b):
    return a + b

def converter(x):
    return float(x)

def main():
    a = converter(sys.argv[1])
    b = converter(sys.argv[2])
    ret = sum_foo(a,b)
    print(ret)
    return ret


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

tests/test.py

```
import my_sum

def test_sum_function():
    "tests that the summation operation is performed correctly"
    a = 2
    b = 3
    assert my_s

def test_conversion():
    "tests that the command-line number converter performs properly"
    a = '6'
    assert my_sum.converter(a) == 6

def test_invalid_conversion():
    "tests that the command-line number converter performs properly"
    a = 'a'
    assert my_sum.converter(a) == 6
```

IMPOSSIBLE TEST (just to see a failure)

`Nosetest` - examples

- Testing `my_sum.py` (basics)

`$ nosetests -v`

```python
import my_sum

def test_sum_function():
    "tests that the summation operation is p
    a = 2
    b = 3
    assert my_sum.sum_foo(a,b) == a + b

def test_conversion():
    "tests that the command-line number conv
    a = '6'
    assert my_sum.converter(a) == 6

def test_invalid_conversion():
    "tests that the command-line number conv
    a = 'a'
    assert my_sum.converter(a) == 6
```

1 — acorbe-gpu@crowdflow2: ~/workspace/ML-for-turbulence/1D/goy_working/GOY-model/prepared_chunks_04.17 — -bash — 110×36

```
acorbe@Alessandros-MacBook-Pro 1  [master] $ ls
my_sum.py       my_sum.pyc      tests
acorbe@Alessandros-MacBook-Pro 1  [master] $ nosetests -v
tests that the summation operation is performed correctly ... ok
tests that the command-line number converter performs properly ... ok
tests that the command-line number converter performs properly ... ERROR

======================================================================
ERROR: tests that the command-line number converter performs properly
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/acorbe/anaconda/lib/python2.7/site-packages/nose/case.py", line 197, in runTest
    self.test(*self.arg)
  File "/Users/acorbe/workspace/2018-corbetta-collaborative-computing-lectures/introduction_to_low_level_testi
ng/examples/1/tests/tests.py", line 17, in test_invalid_conversion
    assert my_sum.converter(a) == 6
  File "/Users/acorbe/workspace/2018-corbetta-collaborative-computing-lectures/introduction_to_low_level_testi
ng/examples/1/my_sum.py", line 8, in converter
    return float(x)
ValueError: could not convert string to float: a

----------------------------------------------------------------------
Ran 3 tests in 0.004s

FAILED (errors=1)
acorbe@Alessandros-MacBook-Pro 1  [master] $
```

`Nosetest` - examples

- Previous tests: **unit tests**
- One scale up: **integration** (here also *system test*)
- We fake command line parameters, we expect the sum
  - Issue: we need to hijack sys.argv parameters
  - **Fixture:** we temporarily change the sys.argv state.
- **Fixture** (in general)
  - pre-test setup + post-test teardown
  - Turn module, class, package to favorable **fake** state

Nosetest - examples

- Fixture example: naïve implementation.

```
def test():
        setup_test()
        try:
                do_test()
                make_test_assertions()
        finally:
                cleanup_after_test()
```

A lot of tests with similar structure

Common issue -> nose helps

Nosetest - examples

- System-level testing

```
acorbe@Alessandros-MacBook-Pro 2  [master] $ tree
.
├── my_sum.py
├── my_sum.pyc
└── tests
    ├── __init__.py
    ├── __init__.pyc
    ├── tests.py
    ├── tests.pyc
    ├── tests_aggregate.py
    └── tests_aggregate.pyc

1 directory, 8 files
```

```python
from __future__ import print_function
import sys

def sum_foo(a,b):
    return a + b

def converter(x):
    return float(x)

def main():
    a = converter(sys.argv[1])
    b = converter(sys.argv[2])
    ret = sum_foo(a,b)
    print(ret)
    return ret


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

```python
from nose.tools import with_setup
import sys
import my_sum


argv_copy = []
def setup_function():
    argv_copy[:] = [x for x in sys.argv]

def teardown_function():
    sys.argv[:] = [x for x in argv_copy]


@with_setup(setup_function, teardown_function)
def test_input():
    sys.argv = ["my_sum", "7", "14"]
    assert abs(my_sum.main() - 21) < (5 * sys.float_info.epsilon)
```

`Nosetest` - examples

- System-level testing

```python
from __future__ import print_function
import sys

def sum_foo(a,b):
    return a + b

def converter(x):
    return float(x)

def main():
    a = converter(sys.argv[1])
    b = converter(sys.argv[2])
    ret = sum_foo(a,b)
    print(ret)
    return ret


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

`$ nosetests -v`

Runs ALL tests

```
2 — acorbe-gpu@crowdflow2: ~/workspace/ML-for-turbulence/1D/goy_working/GO
acorbe@Alessandros-MacBook-Pro 2  [master] $ nosetests -v
tests that the summation operation is performed correctly ... ok
tests that the command-line number converter performs properly ... ok
system level: hijacks argv and checks correct output ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.005s

OK
acorbe@Alessandros-MacBook-Pro 2  [master] $
```

Runs easily SPECIFIC tests

`$ nosetests –v tests.tests_aggregate`

```
acorbe@Alessandros-MacBook-Pro 2  [master] $ nosetests tests.tests_aggregate -v
system level: hijacks argv and checks correct output ... ok

----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
acorbe@Alessandros-MacBook-Pro 2  [master] $
```

NOTE: in general no strict equality checks on float ops

- System-level testing

```
from __future__ import print_function
import sys

def sum_foo(a,b):
    return a + b

def converter(x):
    return float(x)

def main():
    a = converter(sys.argv[1])
    b = converter(sys.argv[2])
    ret = sum_foo(a,b)
    print(ret)
    return ret


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

```
from nose.tools import with_setup
import sys
import my_sum


argv_copy = []
def setup_function():
    argv_copy[:] = [x for x in sys.argv]

def teardown_function():
    sys.argv[:] = [x for x in argv_copy]


@with_setup(setup_function, teardown_function)
def test_input():
    sys.argv = ["my_sum", "7", "14"]
    assert abs(my_sum.main() - 21) < (5 * sys.float_info.epsilon)
```

Topic of this afternoon lecture

`Nosetest` – (more) advanced usage (selected topics)

- nose supports fixtures (setup and teardown methods) at the package, module, class, and test level.
  - **Module**: *`setup_module()`* and *`teardown_module()` are called as the package is imported*
  - **Package**: `__init__.py` should contain *`setup_package()`* *,* *`teardown_package()`. After setup the test in the first module start*
  - **test Class**
    - Inherits from [unittest.TestCase]() or name matching regex
    - Methods in the class that match testMatch are discovered
    - a test case is constructed to **run each method with a fresh instance** of the test class.

`Nosetest` – coverage report

`--with-coverage` argument

```
2 — acorbe-gpu@crowdflow2: ~/workspace/ML-for-turbulence/1D/goy_working/GOY-model/pr
acorbe@Alessandros-MacBook-Pro 2  [master] $ nosetests -v --with-coverage -v
nose.config: INFO: Ignoring files matching ['^\\.', '^_', '^setup\\.py$']
tests that the summation operation is performed correctly ... ok
tests that the command-line number converter performs properly ... ok
system level: hijacks argv and checks correct output ... ok

Name          Stmts   Miss  Cover
---------------------------------
my_sum.py        14      1    93%
----------------------------------------------------------------------
Ran 3 tests in 0.006s

OK
acorbe@Alessandros-MacBook-Pro 2  [master] $
```

Just 93%

What happened???

1 line is missing. Which one?

Nosetest – hotline coverage report

--with-coverage --cover-html

```
2 — acorbe-gpu@crowdflow2: ~/workspace/ML-for-turbulence/1D/goy_working/GOY-model/prepared_chur
acorbe@Alessandros-MacBook-Pro 2  [master] $ nosetests -v --with-coverage --cover-html
tests that the summation operation is performed correctly ... ok
tests that the command-line number converter performs properly ... ok
system level: hijacks argv and checks correct output ... ok

Name         Stmts   Miss  Cover
--------------------------------
my_sum.py       14      1    93%
----------------------------------------------------------------------
Ran 3 tests in 0.008s

OK
acorbe@Alessandros-MacBook-Pro 2  [master] $
```

The folder "cover" with html description is generated

Nosetest – coverage report

file:///Users/acorbe/workspace/2018-corbetta-collaborative-computing-le

Coverage report: 93%

Module ↓	*statements*	*missing*	*excluded*	*coverage*
my_sum.py	14	1	0	93%
Total	**14**	**1**	**0**	**93%**

coverage.py v4.5.1, created at 2018-04-26 17:50

file:///Users/acorbe/workspace/2018-corbetta-collaborative-compu

Coverage for **my_sum.py** : 93%

14 statements | 13 run | 1 missing | 0 excluded

```
from __future__ import print_function
import sys

def sum_foo(a,b):
    return a + b

def converter(x):
    return float(x)

def main():
    a = converter(sys.argv[1])
    b = converter(sys.argv[2])
    ret = sum_foo(a,b)
    print(ret)
    return ret


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

« index *coverage.py v4.5.1, created at 2018-04-26 17:50*