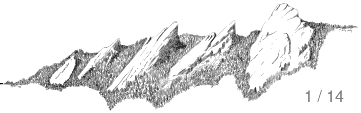
# Introduction to testing scientific codes with py.test

Dorota Jarecka

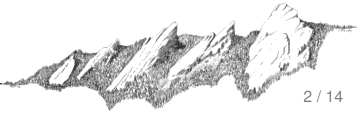
University of Warsaw
NCAR, MMM

2015 April 15th
UCAR SEA Software Engineering Conference 2015
Boulder, CO, USA

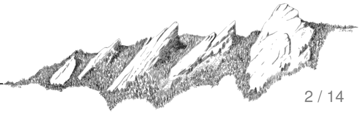


## talk outline

1. Introduction: testing and py.test
2. py.test - examples
3. py.test - options and layout



## talk outline



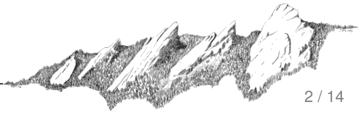
## software testing - motivation

**why testing software?**

## software testing - motivation

**why testing software?**

- mistakes happens and always will



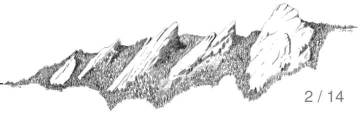
## software testing - motivation

**why testing software?**

- mistakes happens and always will

  $\rightsquigarrow$ guard against them

  $\rightsquigarrow$ raise your confidence during development



## software testing - motivation

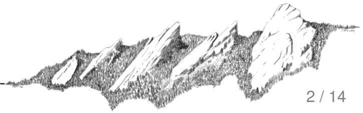
**why testing software?**

- mistakes happens and always will

  $\rightsquigarrow$ guard against them

  $\rightsquigarrow$ raise your confidence during development

- makes you think about desirable output



## software testing - motivation

**why testing software?**

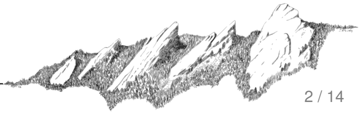
- mistakes happens and always will

  $\rightsquigarrow$ guard against them

  $\rightsquigarrow$ raise your confidence during development

- makes you think about desirable output

  $\rightsquigarrow$ helps you to write a better code



## software testing - motivation

**why testing software?**

- mistakes happens and always will

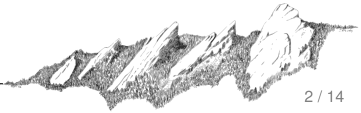
  ⇝ guard against them

  ⇝ raise your confidence during development

- makes you think about desirable output

  ⇝ helps you to write a better code

- improves readability of your code



## software testing - motivation

**why testing software?**

- mistakes happens and always will

  ⇝ guard against them

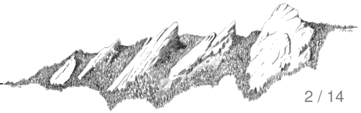
  ⇝ raise your confidence during development

- makes you think about desirable output

  ⇝ helps you to write a better code

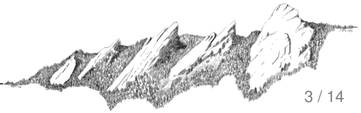
- improves readability of your code

  ⇝ helps to reuse your code



## software testing - types

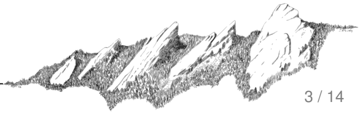
|  |  |
|---|---|
| unittest | testing isolated parts of the code |
| integration | checking if components cooperate |
| functional | checking if code works in an environment |



## software testing - types

| | |
|---|---|
| unittest | testing isolated parts of the code |
| integration | checking if components cooperate |
| functional | checking if code works in an environment |

$\rightsquigarrow$ We will concentrate on unittests

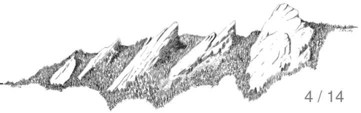


## unit tests and assert statement

## unit tests and assert statement

A simple example - testing division by two

```
def div(a):
    return a/2
```

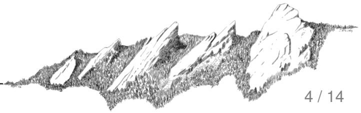


## unit tests and assert statement

```
def div(a):
    return a/2
```

- the simplest assert statement

  ```
  assert div(5) == 2.5
  ```



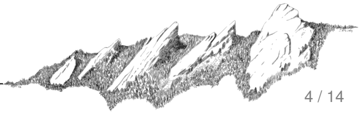
## unit tests and assert statement

```python
def div(a):
    return a/2
```

- the simplest assert statement

  ```python
  assert div(5) == 2.5
  ```

```
Traceback (most recent call last):
  File "test_asserts.py", line 13, in <module>
    assert div(5) == 2.5
AssertionError
```

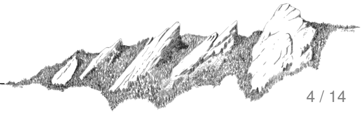


## unit tests and assert statement

```python
def div(a):
    return a/2
```

- the assert statement with message

```python
assert div(5) == 2.5, "div returns wrong values"
```



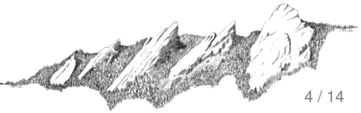
## unit tests and assert statement

```python
def div(a):
    return a/2
```

- the assert statement with message

  ```python
  assert div(5) == 2.5, "div returns wrong values"
  ```

```
Traceback (most recent call last):
  File "test_asserts.py", line 17, in <module>
    assert div(5) == 2.5, "div function returns wrong number"
AssertionError: div function returns wrong number
```

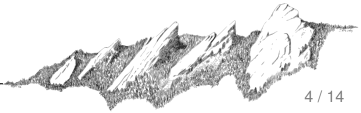


## unit tests and assert statement

```
def div(a):
    return a/2
```

- using Unittest built-in library

```
import unittest

class TestDiv(unittest.TestCase):
    def test_div5(self):
        self.assertEqual( div(5), 2.5)

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



## unit tests and assert statement

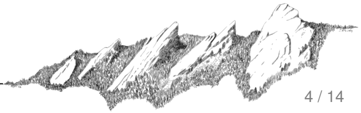
```
F
======================================================================
FAIL: test_div5 (__main__.TestDiv)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_asserts.py", line 25, in test_div5
    self.assertEqual( div(5), 2.5)
AssertionError: 2 != 2.5

----------------------------------------------------------------------
Ran 1 test in 0.002s

FAILED (failures=1)
```

## unit tests and assert statement

```
def div(a):
    return a/2
```

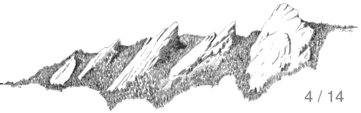
- using py.test

```
def test_div():
    assert div(5) == 2.5
```



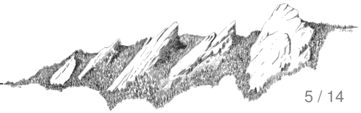
## unit tests and assert statement

```
================================= test session starts =========================
platform darwin -- Python 2.7.5 -- py-1.4.23 -- pytest-2.6.1
collected 1 items

test_asserts.py F

======================================= FAILURES ==============================
_______________________________________ test_div ______________________________

    def test_div():
>       assert div(5) == 2.5
E       assert 2 == 2.5
E        +  where 2 = div(5)

test_asserts.py:22: AssertionError
=============================== 1 failed in 0.05 seconds ======================
```



## py.test

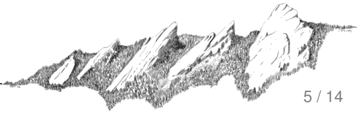
- it's easy to get started
- straightforward asserting with the assert statement
- helpful traceback and failing assertion reporting
- automatic test discovery



## py.test

basic invocation

- *$ py.test*
- *$ python -m pytest*

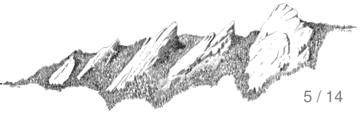


## py.test

basic invocation

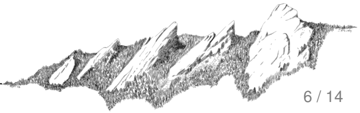
- *$ py.test*
- *$ python -m pytest*

`http://pytest.org/latest/index.html`

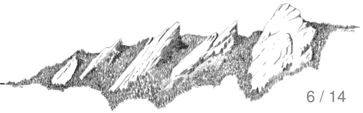


## talk outline

1. Introduction: testing and py.test
2. py.test - examples
3. py.test - options and layout



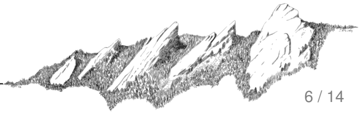
## talk outline



## more about assert statements

- output of a function

```
import math
def test_math_pow():
    assert math.pow(3, 2) == 9
```



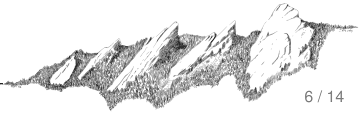
## more about assert statements

- output of a function

```
import math
def test_math_pow():
    assert math.pow(3, 2) == 9
```

- element in a list

```
states = ["CO", "CA", "FL"]
def test_el_list():
    assert "CA" in states
```



## more about assert statements

- output of a function

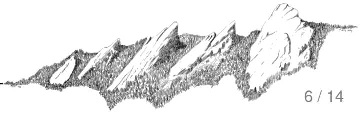
```
import math
def test_math_pow():
    assert math.pow(3, 2) == 9
```

- element in a list

```
states = ["CO", "CA", "FL"]
def test_el_list():
    assert "CA" in states
```

- type of an object

```
import numpy as np
a = np.array([[2,5,12],[4,1,7]])
def test_array_type():
    assert a.dtype == 'float64'
```



## more about assert statements

- output of a function

```
import math
def test_math_pow():
    assert math.pow(3, 2) == 9
```

- element in a list

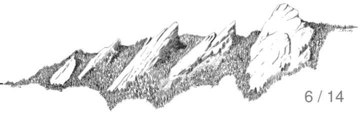
```
states = ["CO", "CA", "FL"]
def test_el_list():
    assert "CA" in states
```

- type of an object

```
import numpy as np
a = np.array([[2,5,12],[4,1,7]])
def test_array_type():
    assert a.dtype == 'float64'
```

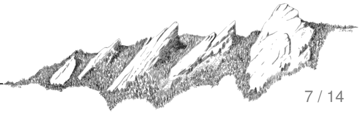
- continuity in memory

```
b = a[1:]
def test_array_Fcont():
    assert b.flags['F_CONTIGUOUS']
```



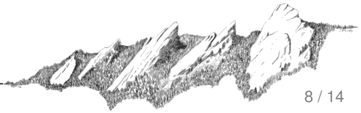
## parametrization of arguments

```
@pytest.mark.parametrize('base,exponent,expected', [
        ( 3, 2, 9),
        (10, 0, 1),
        ])
def test_math_pow(base, exponent, expected):
    assert math.pow(base, exponent) == expected
```



## fixtures

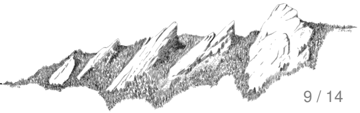
```python
@pytest.fixture(params=[
    (3, 2, 9),
    (10, 0, 1),
    ])
def data(request):
    return request.param

def test_math_pow(data):
    base, exponent, expected = data
    assert math.pow(base, exponent) == expected

def test_your_pow(data):
    base, exponent, expected = data
    assert your_pow_int(base, exponent) == expected
```



## atmospheric example - asserts

Potential temperature

$$\theta = T \left(\frac{p_0}{p}\right)^{(R_d/cp)}$$

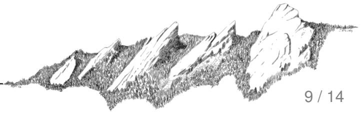


## atmospheric example - asserts

Potential temperature

$$\theta = T\left(\frac{p_0}{p}\right)^{(R_d/cp)}$$

```
def pot_temp(T, p):
    return T * (p0/p) ** (Rd/cp)
```



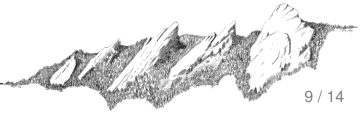
# atmospheric example - asserts

Potential temperature

$$\theta = T \left( \frac{p_0}{p} \right)^{(R_d/cp)}$$

```
def pot_temp(T, p):
    return T * (p0/p) ** (Rd/cp)
```

```
@pytest.mark.parametrize("arg, expected",
                         [({"p": 1.e5, "T": 300}, 300),
                          ({"p": 8.e4, "T": 283}, 301.6)])
def test_expected_output_pottemp(arg, expected, eps=0.05):
    assert (pot_temp(**arg) - expected) < eps
```



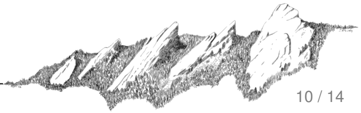
## dealing with tests that can not succeeds

- skipping test

  ```
  def test_function():
  ```

- expecting tests to fail

  ```
  @pytest.mark.xfail
  def test_function():
  ```



## dealing with tests that can not succeeds

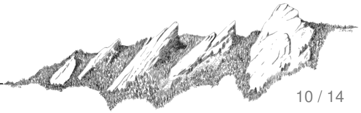
- skipping test

```
def test_function():
```

- expecting tests to fail

```
@pytest.mark.xfail
def test_function():
```

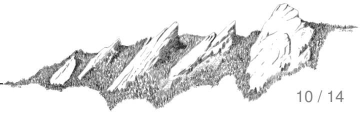
  - forcing to run xfail tests
    *$ pytest –runxfail*



## dealing with tests that can not succeeds

```
@pytest.mark.skipif("qv" not in pot_temp.func_code.co_varnames,
                    reason="a function doesn’t depend on qv")
@pytest.mark.parametrize("arg, expected",
                         [({"p": 1.e5, "T": 300, "qv": 0}, 300.0),
                          ({"p": 8.e4, "T": 283, "qv": 0}, 301.6)])
def test_expected_output_pottemp_qv(arg, expected, eps=0.05):
    assert (pot_temp(**arg) - expected)< eps


def pot_temp(T, p):
    return T * (p0/p) ** (Rd/cp)
```

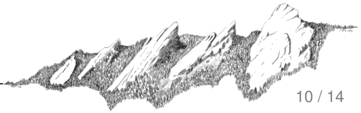
$\rightsquigarrow$ test will be skipped



## dealing with tests that can not succeeds

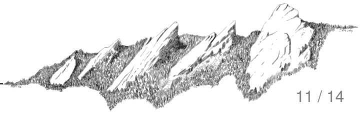
```
@pytest.mark.skipif("qv" not in pot_temp.func_code.co_varnames,
                    reason="a function doesn’t depend on qv")
@pytest.mark.parametrize("arg, expected",
                         [({"p": 1.e5, "T": 300, "qv": 0}, 300.0),
                          ({"p": 8.e4, "T": 283, "qv": 0}, 301.6)])
def test_expected_output_pottemp_qv(arg, expected, eps=0.05):
    assert (pot_temp(**arg) - expected)< eps


def pot_temp(T, p, qv):
    return T * (1+0.61*qv) * (p0/p) ** (Rd/cp)
```

$\rightsquigarrow$ test will be run



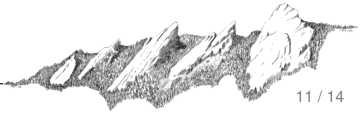
## testing exceptions

```python
def strs2ints(str_list):
    ints = []
    for i, str in enumerate(str_list):
        try:
            value = int(str)
        except ValueError:
            raise ValueError(
                'Element {} is not an integer: {!r}'.format(i, str))
        ints.append(value)
    return ints
```



## testing exceptions

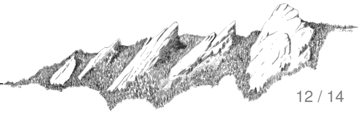
```
def strs2ints(str_list):
    ints = []
    for i, str in enumerate(str_list):
        try:
            value = int(str)
        except ValueError:
            raise ValueError(
                ’Element {} is not an integer: {!r}’.format(i, str))
        ints.append(value)
    return ints


def test_strs2ints_basic():
    with pytest.raises(ValueError):
        strs2ints([’12’, ’-20’, ’abc’, ’5’])
```



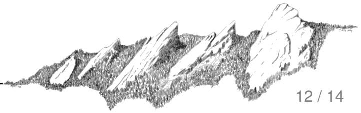
## testing exceptions

```
def strs2ints(str_list):
    ints = []
    for i, str in enumerate(str_list):
        try:
            value = int(str)
        except ValueError:
            raise ValueError(
                ’Element {} is not an integer: {!r}’.format(i, str))
        ints.append(value)
    return ints


def test_strs2ints_basic():
    with pytest.raises(ValueError):
        strs2ints([’12’, ’-20’, ’abc’, ’5’])


def test_strs2ints_advanced():
    with pytest.raises(ValueError) as exc_info:
        strs2ints([’12’, ’-20’, ’abc’, ’5’])
    assert str(exc_info.value) == "Element 2 is not an integer: ’abc’"
```

## talk outline

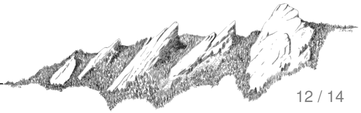


## talk outline



## pytest options

- additional information
  *$ py.test –version* - shows where pytest was imported from
  *$ py.test -h — –help* - show help on command line

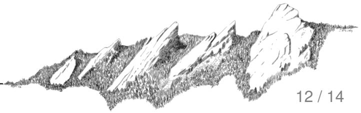


## pytest options

- additional information
  *$ py.test –version* - shows where pytest was imported from
  *$ py.test -h — –help* - show help on command line
- stopping after failures
  *$ py.test -x* - stops after first failure
  *$ py.test –maxfail=2* - stops after two failures

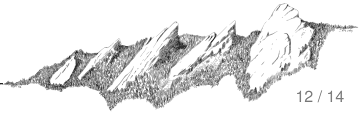
## pytest options

- additional information
  *$ py.test –version* - shows where pytest was imported from
  *$ py.test -h — –help* - show help on command line
- stopping after failures
  *$ py.test -x* - stops after first failure
  *$ py.test –maxfail=2* - stops after two failures
- using PDB (Python Debugger)
  *$ py.test –pdb* - invokes PDB on every failure
  *$ py.test -x –pdb* - drops to PDB on first failure and stops



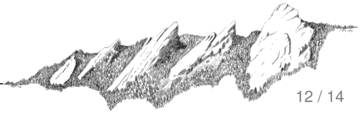
## pytest options

- additional information
  *$ py.test –version* - shows where pytest was imported from
  *$ py.test -h — –help* - show help on command line
- stopping after failures
  *$ py.test -x* - stops after first failure
  *$ py.test –maxfail=2* - stops after two failures
- using PDB (Python Debugger)
  *$ py.test –pdb* - invokes PDB on every failure
  *$ py.test -x –pdb* - drops to PDB on first failure and stops
- additional information about skips and xfails
  *$ py.test -rxs* - shows extra info on skips and xfails



## pytest options

- additional information
  *$ py.test –version* - shows where pytest was imported from
  *$ py.test -h — –help* - show help on command line
- stopping after failures
  *$ py.test -x* - stops after first failure
  *$ py.test –maxfail=2* - stops after two failures
- using PDB (Python Debugger)
  *$ py.test –pdb* - invokes PDB on every failure
  *$ py.test -x –pdb* - drops to PDB on first failure and stops
- additional information about skips and xfails
  *$ py.test -rxs* - shows extra info on skips and xfails
- profiling test execution duration
  *$ py.test –durations=10* - returns a list of the slowest tests



## pytest layouts

```
mypkg/
    __init__.py
    appmodule.py
    ...
    test/
        test_app.py
        ...
```

## further reading

https://pytest.org/latest/index.html
https://pytest.org/latest/talks.html
https://pytest.org/latest/talks.html