Testing

Unit Testing

Copyright © Software Carpentry 2010
This work is licensed under the Creative Commons Attribution License
See http://software-carpentry.org/license.html for more information.
Studying impact of climate change on agriculture
Studying impact of climate change on agriculture

Have aerial photos of farms from 1980–83
Studying impact of climate change on agriculture
Have aerial photos of farms from 1980–83
Want to compare with photos from 2007–present
Studying impact of climate change on agriculture

Have aerial photos of farms from 1980–83

Want to compare with photos from 2007–present

First step is to find regions where fields overlap
Luckily, these fields are in Saskatchewan...
Luckily, these fields are in Saskatchewan...

...where fields are rectangles
A student has written a function that finds the overlap between two rectangles.
A student has written a function that finds the overlap between two rectangles

We want to test it before using it
A student has written a function that finds the overlap between two rectangles. We want to test it before using it. We're also planning to try to speed it up...
A student has written a function that finds the overlap between two rectangles. We want to test it before using it. We're also planning to try to speed it up... and want tests to make sure we don't break it.
A student has written a function that finds the overlap between two rectangles.
We want to test it before using it.
We're also planning to try to speed it up...
...and want tests to make sure we don't break it.

Use Python's Nose library.
Each test is a function
Each test is a function

- Whose name begins with test_
Each test is a function

- Whose name begins with `test_`

Group related tests in files
Each test is a function
- Whose name begins with test_
Group related tests in files
- Whose names begin with test_
Each test is a function
  – Whose name begins with `test_`
Group related tests in files
  – Whose names begin with `test_`

Run the command `nose tests`
Each test is a function
- Whose name begins with test_
Group related tests in files
- Whose names begin with test_
Run the command nosetests
- Which automatically search the current directory and sub-directories for tests
Simple example: testing dna_starts_with
Simple example: testing `dna_starts_with`

```python
def test_starts_with_itself():
    dna = 'actgt'
    assert dna_starts_with(dna, dna)

def test_starts_with_single_base_pair():
    assert dna_starts_with('actg', 'a')

def does_not_start_with_single_base_pair():
    assert not dna_starts_with('ttct', 'a')
```
Simple example: testing `dna_starts_with`

```python
def test_starts_with_itself():
    dna = 'actgt'
    assert dna_starts_with(dna, dna)

def test_starts_with_single_base_pair():
    assert dna_starts_with('actg', 'a')

def does_not_start_with_single_base_pair():
    assert not dna_starts_with('ttct', 'a')
```

Give tests meaningful names
Simple example: testing `dna_starts_with`

```python
def test_starts_with_itself():
    dna = 'actgt'
    assert dna_starts_with(dna, dna)

def test_starts_with_single_base_pair():
    assert dna_starts_with('actg', 'a')

def does_not_start_with_single_base_pair():
    assert not dna_starts_with('ttct', 'a')
```

Use `assert` to check results.
Simple example: testing `dna_starts_with`

```python
def test_starts_with_itself():
    dna = 'actgt'
    assert dna_starts_with(dna, dna)

def test_starts_with_single_base_pair():
    assert dna_starts_with('actg', 'a')

def does_not_start_with_single_base_pair():
    assert not dna_starts_with('ttct', 'a')
```

Use variables for fixtures to prevent typing mistakes.
Simple example: testing `dna_starts_with`

def test_starts_with_itself():
    dna = 'actgt'
    assert dna_starts_with(dna, dna)

def test_starts_with_single_base_pair():
    assert dna_starts_with('actg', 'a')

def does_not_start_with_single_base_pair():
    assert not dna_starts_with('ttct', 'a')

Test lots of cases
"Test lots of cases"
"Test lots of cases"

How many?
"Test lots of cases"

How many?
"Test lots of cases"

How many?

How to choose cost-effective tests?
"Test lots of cases"

How many?

How to choose cost-effective tests?

If we test `dna_starts_with('atc', 'a')`, we're unlikely to learn much from testing `dna_starts_with('ttc', 't')`
"Test lots of cases"

How to choose cost-effective tests?

If we test `dna_starts_with('atc', 'a')`, we're unlikely to learn much from testing `dna_starts_with('ttc', 't')`

So choose tests that are as different from each other as possible
"Test lots of cases"

How many?

How to choose cost-effective tests?

If we test `dna_starts_with('atc', 'a')`, we're unlikely to learn much from testing `dna_starts_with('ttc', 't')`

So choose tests that are as different from each other as possible

Look for *boundary cases*
Apply this to overlapping rectangles
Apply this to overlapping rectangles

A "normal" case
Apply this to overlapping rectangles

A "normal" case

What else would be useful?
Apply this to overlapping rectangles

A "normal" case

What else would be useful?
Testing

Unit Testing
Testing

Unit Testing
Testing

Unit Testing
Testing

Unit Testing
Tests help us define what "correct" actually means.
Turn this into code
def test_touch_on_corner():
    one = ((0, 0), (1, 1))
    two = ((1, 1), (2, 2))
    assert overlap(one, two) == None
def test_touch_on_corner():
    one = ((0, 0), (1, 1))
    two = ((1, 1), (2, 2))
    assert overlap(one, two) == None

An unambiguous, runnable answer to our question about touching on corners
def test_unit_with_itself():
    unit = ((0, 0), (1, 1))
    assert overlap(unit, unit) == unit
def test_unit_with_itself():
    unit = ((0, 0), (1, 1))
    assert overlap(unit, unit) == unit

    # Wasn't actually in the set of test cases we came up with earlier
def test_partial_overlap():
    red = ((0, 3), (2, 5))
    blue = ((1, 0), (2, 4))
    assert overlap(red, blue) == ((1, 3), (2, 4))
def test_partial_overlap():
    red = ((0, 3), (2, 5))
    blue = ((1, 0), (2, 4))
    assert overlap(red, blue) == ((1, 3), (2, 4))

This test actually turned up a bug
def overlap(red, blue):
    '''Return overlap between two rectangles, or None.'''

    ((red_lo_x, red_lo_y), (red_hi_x, red_hi_y)) = red
    ((blue_lo_x, blue_lo_y), (blue_hi_x, blue_hi_y)) = blue

    if (red_lo_x >= blue_hi_x) or (red_hi_x <= blue_lo_x) or 
        (red_lo_y >= blue_hi_y) or (red_hi_y <= blue_lo_y):
        return None

    lo_x = max(red_lo_x, blue_lo_x)
    lo_y = max(red_lo_y, blue_lo_y)
    hi_x = min(red_hi_x, blue_hi_x)
    hi_y = min(red_hi_y, blue_hi_y)

    return ((lo_x, lo_y), (hi_x, hi_y))
def overlap(red, blue):
    '''Return overlap between two rectangles, or None.'''

    ((red_lo_x, red_lo_y), (red_hi_x, red_hi_y)) = red
    ((blue_lo_x, blue_lo_y), (blue_hi_x, blue_hi_y)) = blue

    if (red_lo_x >= blue_hi_x) or (red_hi_x <= blue_lo_x) or 
       (red_lo_y >= blue_hi_y) or (red_hi_y <= blue_lo_y):
        return None

    lo_x = max(red_lo_x, blue_lo_x)
    lo_y = max(red_lo_y, blue_lo_y)
    hi_x = min(red_hi_x, blue_hi_x)
    hi_y = min(red_hi_y, blue_hi_y)

    return ((lo_x, lo_y), (hi_x, hi_y))
def overlap(red, blue):
    '''Return overlap between two rectangles, or None.'''

    ((red_lo_x, red_lo_y), (red_hi_x, red_hi_y)) = red
    ((blue_lo_x, blue_lo_y), (blue_hi_x, blue_hi_y)) = blue

    if (red_lo_x >= blue_hi_x) or (red_hi_x <= blue_lo_x) or 
        (red_lo_y >= blue_hi_y) or (red_hi_y <= blue_lo_y):
        return None

    lo_x = max(red_lo_x, blue_lo_x)
    lo_y = max(red_lo_y, blue_lo_y)
    hi_x = min(red_hi_x, blue_hi_x)
    hi_y = min(red_hi_y, blue_hi_y)

    return ((lo_x, lo_y), (hi_x, hi_y))
You should spend your time choosing test cases and defining their answers
You should spend your time choosing test cases and defining their answers.

Nose (and its kin) are there to handle everything that you *shouldn't* re-think each time.
You should spend your time choosing test cases and defining their answers.
Nose (and its kin) are there to handle everything that you *shouldn't* re-think each time.

"The tool shapes the hand"