Testing

Fixtures

Back to those fields in Saskatchewan...

Finding areas in photographs where fields overlap

Finding areas in photographs where fields overlap

Each photograph contains one or more rectangles

Finding areas in photographs where fields overlap

Each photograph contains one or more rectangles

So a photo is a collection (set? list?) of rectangles

Finding areas in photographs where fields overlap

Each photograph contains one or more rectangles

So a photo is a collection (set? list?) of rectangles

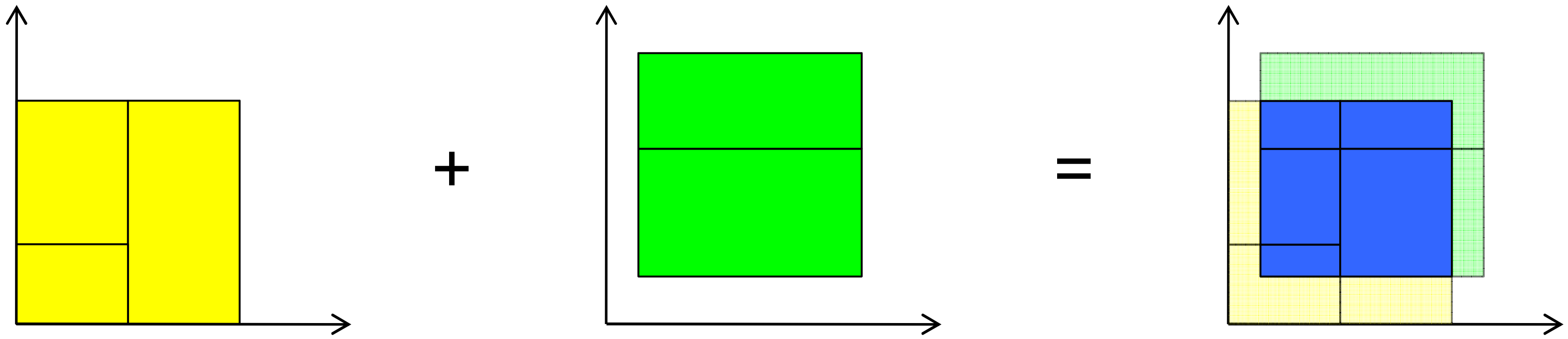
Want to find *all* overlaps

Finding areas in photographs where fields overlap

Each photograph contains one or more rectangles

So a photo is a collection (set? list?) of rectangles

Want to find *all* overlaps



Have tested `overlap_rect(rect_1, rect_2)`

Have tested `overlap_rect(rect_1, rect_2)`

Now want to test `overlap_photo(photo_1, photo_2)`

Have tested `overlap_rect(rect_1, rect_2)`

Now want to test `overlap_photo(photo_1, photo_2)`

Imagine its implementation is something like this

```
def overlap_photo(photo_1, photo_2):
  result = set()
  for rect_1 in photo_1:
    for rect_2 in photo_2:
      temp = overlap_rect(rect_1, rect_2)
      if temp is not None:
        result.add(temp)
  return result
```

Have tested `overlap_rect(rect_1, rect_2)`

Now want to test `overlap_photo(photo_1, photo_2)`

Imagine its implementation is something like this

```
def overlap_photo(photo_1, photo_2):
  result = set()
  for rect_1 in photo_1:
    for rect_2 in photo_2:
      temp = overlap_rect(rect_1, rect_2)
      if temp is not None:
        result.add(temp)
  return result
```

Compare all against all

Have tested `overlap_rect(rect_1, rect_2)`

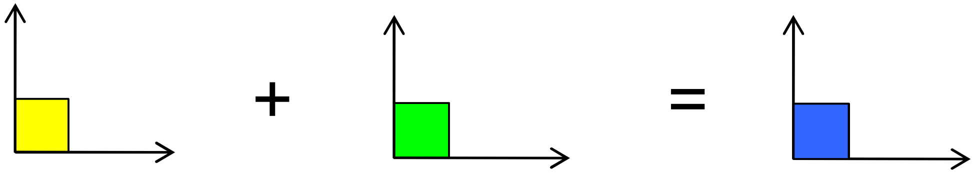
Now want to test `overlap_photo(photo_1, photo_2)`

Imagine its implementation is something like this

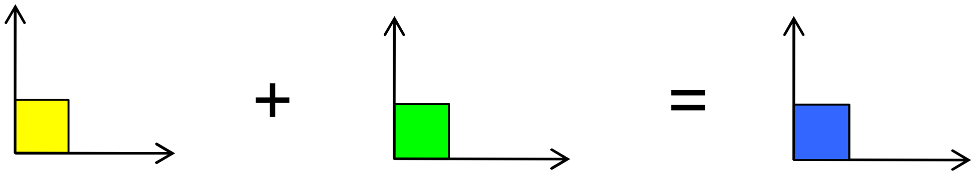
```
def overlap_photo(photo_1, photo_2):
  result = set()
  for rect_1 in photo_1:
    for rect_2 in photo_2:
      temp = overlap_rect(rect_1, rect_2)
      if temp is not None:
        result.add(temp)
  return result
```

Save every non-empty overlap

First test

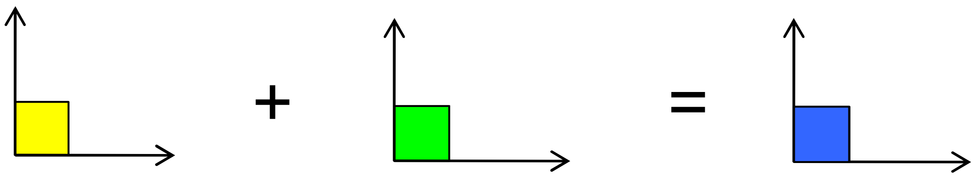


First test



```
def test_unit_with_unit():
  unit = ((0, 0), (1, 1))
  photo_1 = { unit }
  photo_2 = { unit }
  result = overlap_photo(photo_1, photo_2)
  assert result == { unit }
```

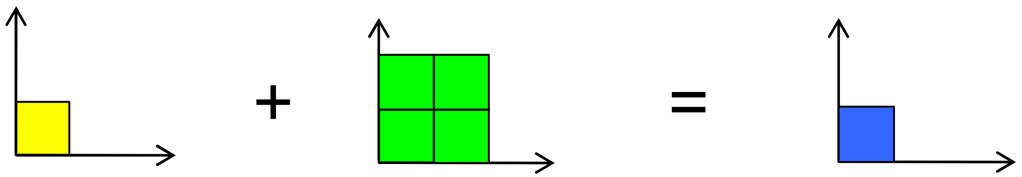
First test



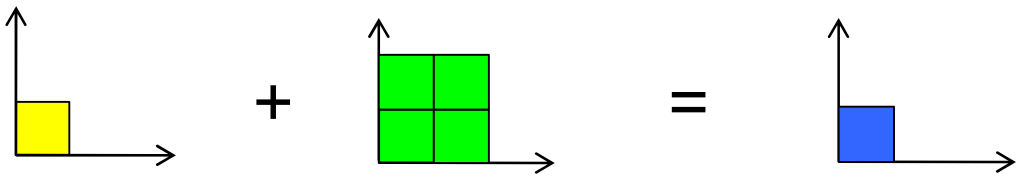
```
def test_unit_with_unit():
  unit = ((0, 0), (1, 1))
  photo_1 = { unit }
  photo_2 = { unit }
  result = overlap_photo(photo_1, photo_2)
  assert result == { unit }
```

That's not too bad

Second test

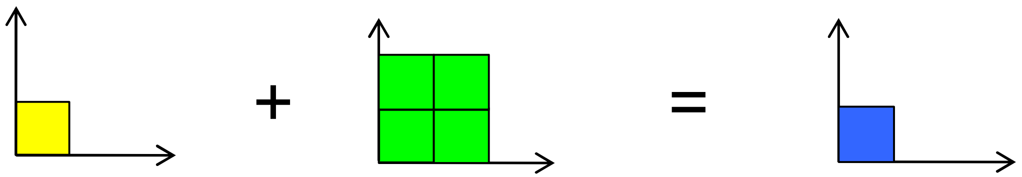


Second test



```
def test_unit_with_checkerboard():
  photo_1 = { ((0, 0), (1, 1)) }
  photo_2 = { ((0, 0), (1, 1)), ((1, 0), (2, 1)),
              ((0, 1), (1, 2)), ((1, 1), (2, 2)) }
  result = overlap_photo(photo_1, photo_2)
  assert result == { ((0, 0), (1, 1)) }
```

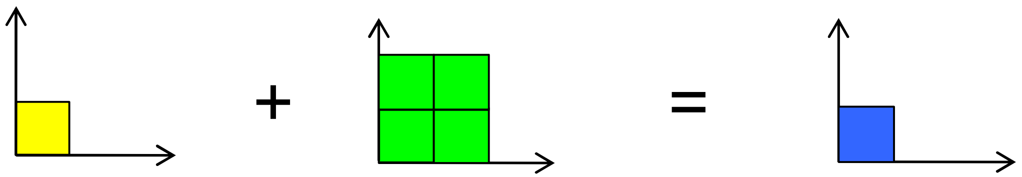
Second test



```
def test_unit_with_checkerboard():
  photo_1 = { ((0, 0), (1, 1)) }
  photo_2 = { ((0, 0), (1, 1)), ((1, 0), (2, 1)),
                   ((0, 1), (1, 2)), ((1, 1), (2, 2)) }
  result = overlap_photo(photo_1, photo_2)
  assert result == { ((0, 0), (1, 1)) }
```

That's hard to read

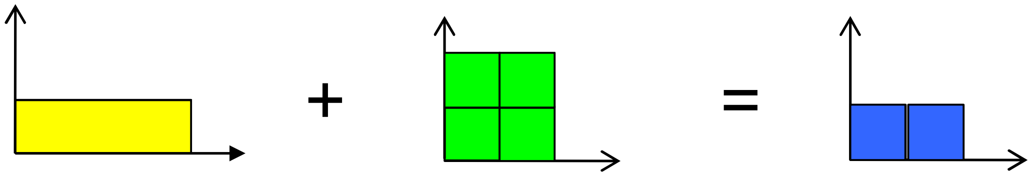
Second test



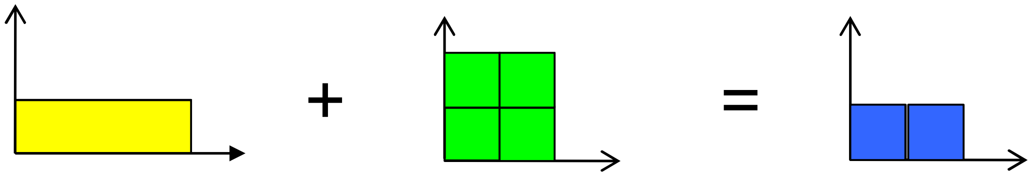
```
def test_unit_with_checkerboard():
  unit = ((0, 0), (1, 1))
  photo_1 = { unit }
  photo_2 = { ((0, 0), (1, 1)), ((1, 0), (2, 1)),
              ((0, 1), (1, 2)), ((1, 1), (2, 2)) }
  result = overlap_photo(photo_1, photo_2)
  assert result == { unit }
```

Using `unit` instead of `((0, 0), (1, 1))` doesn't really help much

Third test

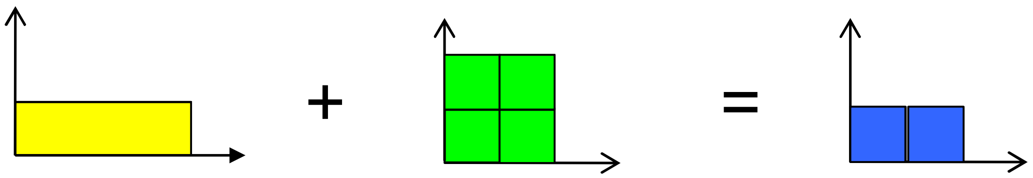


Third test



```
def test_unit_checkerboard_with_short_and_wide():
  photo_1 = { ((0, 0), (3, 1)) }
  photo_2 = { ((0, 0), (1, 1)), ((1, 0), (2, 1)),
              ((0, 1), (1, 2)), ((1, 1), (2, 2)) }
  result = overlap_photo(photo_1, photo_2)
  assert result == { ((0, 0), (1, 1)), ((1, 0), (2, 1)) }
```

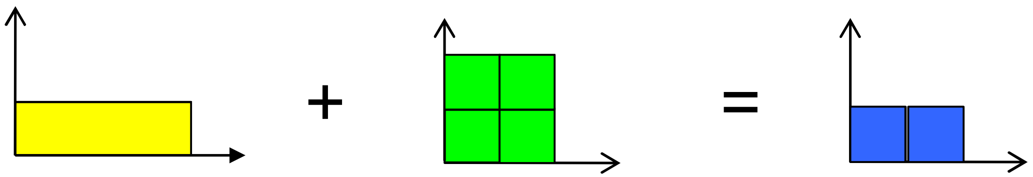
Third test



```
def test_unit_checkerboard_with_short_and_wide():
  photo_1 = { ((0, 0), (3, 1)) }
  photo_2 = { ((0, 0), (1, 1)), ((1, 0), (2, 1)),
              ((0, 1), (1, 2)), ((1, 1), (2, 2)) }
  result = overlap_photo(photo_1, photo_2)
  assert result == { ((0, 0), (1, 1)), ((1, 0), (2, 1)) }
```

Also hard to read

Third test



```
def test_unit_checkerboard_with_short_and_wide():
  photo_1 = { ((0, 0), (3, 1)) }
  photo_2 = { ((0, 0), (1, 1)), ((1, 0), (2, 1)),
              ((0, 1), (1, 2)), ((1, 1), (2, 2)) }
  result = overlap_photo(photo_1, photo_2)
  assert result == { ((0, 0), (1, 1)), ((1, 0), (2, 1)) }
```

Also hard to read

And a new problem: too much duplicated code

Solution: create fixtures outside specific tests

Nose

Solution: create fixtures outside specific tests

(Reminder: the *fixture* is the thing the test is run on)

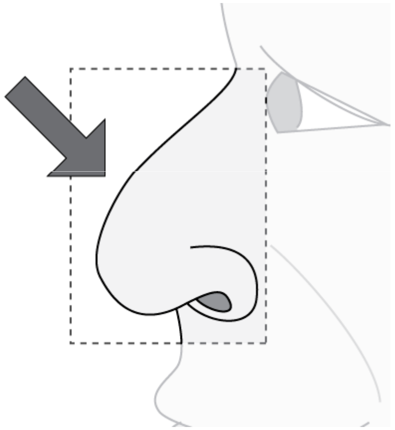
Nose

Solution: create fixtures outside specific tests

(Reminder: the *fixture* is the thing the test is run on)

If a module contains a function called `setup`,

Nose runs that before it runs any of the tests



Nose

Here's how it works

```
import sys

def setup():
  print >> sys.stderr, 'setup'

def test_1():
  print >> sys.stderr, 'test 1'

def test_2():
  print >> sys.stderr, 'test 2'
```

Here's how it works

```
import sys

def setup():
  print >> sys.stderr, 'setup'

def test_1():
  print >> sys.stderr, 'test 1'

def test_2():
  print >> sys.stderr, 'test 2'
```

Would actually create fixtures

Here's how it works

```
import sys

def setup():
  print >> sys.stderr, 'setup'

def test_1():
  print >> sys.stderr, 'test 1'

def test_2():
  print >> sys.stderr, 'test 2'
```

Would actually run tests

Here's how it works

```
import sys

def setup():
  print >> sys.stderr, 'setup'

def test_1():
  print >> sys.stderr, 'test 1'

def test_2():
  print >> sys.stderr, 'test 2'
```

➔

```
setup
test 1
.test 2
.
--------------------------------
Ran 2 tests in 0.001s

OK
```

Here's how it works

```
import sys

def setup():
  print >> sys.stderr, 'setup'

def test_1():
  print >> sys.stderr, 'test 1'

def test_2():
  print >> sys.stderr, 'test 2'
```

➔

```
setup
test 1
.test 2
.
----------------------------------
Ran 2 tests in 0.001s

OK
```

This is Nose's usual output

Here's how it works

```
import sys

def setup():
  print >> sys.stderr, 'setup'

def test_1():
  print >> sys.stderr, 'test 1'

def test_2():
  print >> sys.stderr, 'test 2'
```

➔

```
..
--------------------------------
Ran 2 tests in 0.001s

OK
```

Would look like this without our print statements

Here's how it works

```
import sys

def setup():
  print >> sys.stderr, 'setup'

def test_1():
  print >> sys.stderr, 'test 1'

def test_2():
  print >> sys.stderr, 'test 2'
```

➔

```
setup
test 1
.test 2
.
--------------------------------
Ran 2 tests in 0.001s

OK
```

Nose runs setup

once at the start

Here's how it works

```
import sys

def setup():
  print >> sys.stderr, 'setup'

def test_1():
  print >> sys.stderr, 'test 1'

def test_2():
  print >> sys.stderr, 'test 2'
```

➔

```
setup
test 1
.test 2
.
----------------------------------
Ran 2 tests in 0.001s

OK
```

Then runs tests

(in any order)

Create fixtures for testing photo overlap

Create fixtures for testing photo overlap

```
Photos = {}

def setup():
  Photos['unit'] = { ((0, 0), (1, 1)) }
  Photos['checkerboard'] = { ((0, 0), (1, 1)),
                             ((1, 0), (2, 1)),
                             ((0, 1), (1, 2)),
                             ((1, 1), (2, 2)) }
   Photos['short_and_wide'] = { ((0, 0), (3, 1)) }
```

Create fixtures for testing photo overlap

```
Photos = {}
```

Store fixtures in a global variable

so they're visible in every test

```
def setup():
  Photos['unit'] = { ((0, 0), (1, 1)) }
  Photos['checkerboard'] = { ((0, 0), (1, 1)),
                             ((1, 0), (2, 1)),
                             ((0, 1), (1, 2)),
                             ((1, 1), (2, 2)) }
   Photos['short_and_wide'] = { ((0, 0), (3, 1)) }
```

Create fixtures for testing photo overlap

```
Photos = {}

def setup():
  Photos['unit'] = { ((0, 0), (1, 1)) }
  Photos['checkerboard'] = { ((0, 0), (1, 1)),
                             ((1, 0), (2, 1)),
                             ((0, 1), (1, 2)),
                             ((1, 1), (2, 2)) }
   Photos['short_and_wide'] = { ((0, 0), (3, 1)) }
```

Create fixtures once before tests are run

Then use fixtures in tests

Then use fixtures in tests

```
def test_unit_with_unit():
  temp = overlap_rect(Photos['unit'], Photos['unit'])
  assert temp == Photos['unit']
```

Then use fixtures in tests

```
def test_unit_with_unit():
  temp = overlap_rect(Photos['unit'], Photos['unit'])
  assert temp == Photos['unit']

def test_checkerboard_with_short_and_wide():
  temp = overlap_rect(Photos['checkerboard'],
                      Photos['short_and_wide'])
  assert temp == { ((0, 0), (1, 1)), ((1, 0), (2, 1)) }
```

Could create one global variable per fixture

Could create one global variable per fixture

```
Unit = None
Short_And_Wide = None

def setup():
  Unit = { ((0, 0), (1, 1)) }
  Short_And_Wide = { ((0, 0), (3, 1)) }
```

Could create one global variable per fixture

```
Unit = None
Short_And_Wide = None

def setup():
  Unit = { ((0, 0), (1, 1)) }
  Short_And_Wide = { ((0, 0), (3, 1)) }
```

A matter of taste and style

Don't actually need `setup` in this case

```
Unit = { ((0, 0), (1, 1)) }
Short_And_Wide = { ((0, 0), (3, 1)) }
```

Don't actually need `setup` in this case

```
Unit = { ((0, 0), (1, 1)) }
Short_And_Wide = { ((0, 0), (3, 1)) }
```

But this doesn't generalize

What if tests modify fixtures?

What if tests modify fixtures?

Example: `photo_crop(photo, rect)` removes all rectangles in photo that are completely outside the given cropping window

What if tests modify fixtures?

Example: `photo_crop(photo, rect)` removes all rectangles in photo that are completely outside the given cropping window

This means it isn't safe to re-use fixtures

What if tests modify fixtures?

Example: `photo_crop(photo, rect)` removes all rectangles in photo that are completely outside the given cropping window

This means it isn't safe to re-use fixtures

So re-create fixtures for each test

Use a *decorator* for per-test setup

Use a *decorator* for per-test setup

```
import sys
from nose import with_setup

def setup_each():
  print >> sys.stderr, ’setup each’

@with_setup(setup_each)
def test_1():
  print >> sys.stderr, ’test 1’

@with_setup(setup_each):
def test_2():
  print >> sys.stderr, ’test 2’
```

Use a *decorator* for per-test setup

```
import sys
from nose import with_setup

def setup_each():
  print >> sys.stderr, ’setup each’

@with_setup(setup_each)
def test_1():
  print >> sys.stderr, ’test 1’

@with_setup(setup_each):
def test_2():
  print >> sys.stderr, ’test 2’
```

Import the decorator from the Nose library

Use a *decorator* for per-test setup

```
import sys
from nose import with_setup

def setup_each():
  print >> sys.stderr, ’setup each’

@with_setup(setup_each)
def test_1():
  print >> sys.stderr, ’test 1’

@with_setup(setup_each):
def test_2():
  print >> sys.stderr, ’test 2’
```

Import the decorator from the Nose library

(It's actually just a function that behaves a specific way)

Use a *decorator* for per-test setup

```
import sys
from nose import with_setup

def setup_each():
  print >> sys.stderr, 'setup each'

@with_setup(setup_each)
def test_1():
  print >> sys.stderr, 'test 1'

@with_setup(setup_each)
def test_2():
  print >> sys.stderr, 'test 2'
```

Use @decorator(args)

to apply it to a function

Use a *decorator* for per-test setup

```
import sys
from nose import with_setup

def setup_each():
  print >> sys.stderr, ’setup each’

@with_setup(setup_each)
def test_1():
  print >> sys.stderr, ’test 1’

@with_setup(setup_each)
def test_2():
  print >> sys.stderr, ’test 2’
```

Use @decorator(args) to apply it to a function

Tells Nose to run setup_each before running the test

Use a *decorator* for per-test setup

```
import sys
from nose import with_setup

def setup_each():
  print >> sys.stderr, ’setup each’

@with_setup(setup_each)
def test_1():
  print >> sys.stderr, ’test 1’

@with_setup(setup_each)
def test_2():
  print >> sys.stderr, ’test 2’
```

→

```
setup each
test 1
.setup each
test 2
.
--------------------------------
Ran 2 tests in 0.001s

OK
```

Use a *decorator* for per-test setup

```
import sys
from nose import with_setup

def setup_each():
  print >> sys.stderr, ’setup each’

@with_setup(setup_each)
def test_1():
  print >> sys.stderr, ’test 1’

@with_setup(setup_each)
def test_2():
  print >> sys.stderr, ’test 2’
```

➔

```
setup each
test 1
.setup each
test 2
.
--------------------------------
Ran 2 tests in 0.001s

OK
```

Standard Nose output

Use a *decorator* for per-test setup

```
import sys
from nose import with_setup

def setup_each():
  print >> sys.stderr, 'setup each'

@with_setup(setup_each)
def test_1():
  print >> sys.stderr, 'test 1'

@with_setup(setup_each)
def test_2():
  print >> sys.stderr, 'test 2'
```

➔

```
setup each
test 1
.setup each
test 2
.
--------------------------------
Ran 2 tests in 0.001s

OK
```

Nose ran `setup_each`

before `test_1`

Use a *decorator* for per-test setup

```
import sys
from nose import with_setup

def setup_each():
  print >> sys.stderr, 'setup each'

@with_setup(setup_each)
def test_1():
  print >> sys.stderr, 'test 1'

@with_setup(setup_each)
def test_2():
  print >> sys.stderr, 'test 2'
```

➔

```
setup each
test 1
.setup each
test 2
.
----------------------------------
Ran 2 tests in 0.001s

OK
```

And then again

before `test_2`

```
from nose import with_setup

checkerboard = None
unit = None
whole_map = None

@with_setup(create_fixtures)
def test_crop_unit():
  photo_crop(checkerboard, unit)
  assert checkerboard == unit

@with_setup(create_fixtures)
def test_crop_keep_everything():
  original = photo_copy(checkerboard)
  photo_crop(checkerboard, whole_map)
  assert checkerboard == original
```

```
from nose import with_setup

checkerboard = None
unit = None
whole_map = None

@with_setup(create_fixtures)
def test_crop_unit():
  photo_crop(checkerboard, unit)
  assert checkerboard == unit

@with_setup(create_fixtures)
def test_crop_keep_everything():
  original = photo_copy(checkerboard)
  photo_crop(checkerboard, whole_map)
  assert checkerboard == original
```

`create_fixtures`

– Create first copy of checkerboard

```
from nose import with_setup

checkerboard = None
unit = None
whole_map = None

@with_setup(create_fixtures)
def test_crop_unit():
  photo_crop(checkerboard, unit)
  assert checkerboard == unit

@with_setup(create_fixtures)
def test_crop_keep_everything():
  original = photo_copy(checkerboard)
  photo_crop(checkerboard, whole_map)
  assert checkerboard == original
```

`create_fixtures`

– Create first copy of checkerboard

`test_crop_unit`

– Modify checkerboard

```
from nose import with_setup

checkerboard = None
unit = None
whole_map = None

@with_setup(create_fixtures)
def test_crop_unit():
  photo_crop(checkerboard, unit)
  assert checkerboard == unit

@with_setup(create_fixtures)
def test_crop_keep_everything():
  original = photo_copy(checkerboard)
  photo_crop(checkerboard, whole_map)
  assert checkerboard == original
```

`create_fixtures`

– Create first copy of checkerboard

`test_crop_unit`

– Modify checkerboard

`create_fixtures`

– Creates fresh copy of checkerboard

```
from nose import with_setup

checkerboard = None
unit = None
whole_map = None

@with_setup(create_fixtures)
def test_crop_unit():
  photo_crop(checkerboard, unit)
  assert checkerboard == unit

@with_setup(create_fixtures)
def test_crop_keep_everything():
  original = photo_copy(checkerboard)
  photo_crop(checkerboard, whole_map)
  assert checkerboard == original
```

`create_fixtures`

– Create first copy of checkerboard

`test_crop_unit`

– Modify checkerboard

`create_fixtures`

– Creates fresh copy of checkerboard

`test_copy_keep_everything`

– Modify checkerboard again

Re-running setup wastes a few microseconds of the computer's time

Re-running setup wastes a few microseconds of the computer's time

That is much less valuable than any of yours

Decorators aren't magic

Decorators aren't magic

But they are tricky

Decorators aren't magic

But they are tricky

You don't have to understand how they work

Decorators aren't magic

But they are tricky

You don't have to understand how they work

Just as you don't have to understand how Nose finds test in files or files that contain tests

Decorators aren't magic

But they are tricky

You don't have to understand how they work

Just as you don't have to understand how Nose finds test in files or files that contain tests

As long as you know:

Decorators aren't magic

But they are tricky

You don't have to understand how they work

Just as you don't have to understand how Nose

finds test in files or files that contain tests

As long as you know:

– What @with_setup does

Decorators aren't magic

But they are tricky

You don't have to understand how they work

Just as you don't have to understand how Nose finds test in files or files that contain tests

As long as you know:

– What @with_setup does

– When and why to use it

software carpentry

created by

Greg Wilson

August 2010