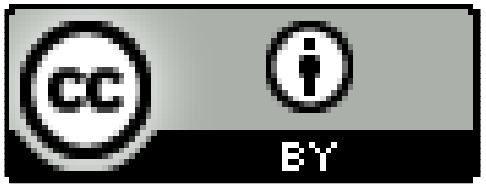
Testing

Introduction



Nobody enjoys testing

Nobody enjoys testing

So you can skip this lecture *if*

Nobody enjoys testing

So you can skip this lecture *if*

- your programs always work correctly, or

Nobody enjoys testing

So you can skip this lecture *if*

- your programs always work correctly, or
- you don't care if they're correct or not, so long as their output looks plausible, and

Nobody enjoys testing

So you can skip this lecture *if*

- your programs always work correctly, or
- you don't care if they're correct or not, so long as their output looks plausible, and
- you like being inefficient

Nobody enjoys testing

So you can skip this lecture *if*

- your programs always work correctly, or
- you don't care if they're correct or not, so long as their output looks plausible, and
- you like being inefficient

The more you invest in quality, the less total time it takes to build working software

Testing tells you:

Testing tells you:

- if the program is doing what it's supposed to

Testing tells you:

- if the program is doing what it's supposed to
- what the program actually *is* supposed to do

Testing tells you:

- if the program is doing what it's supposed to
- what the program actually *is* supposed to do

Tests are runnable specifications

Testing tells you:

- if the program is doing what it's supposed to
- what the program actually *is* supposed to do

Tests are runnable specifications

- Less likely to fall out of sync with the program than documentation

Quality is *not* just testing

Quality is *not* just testing

Trying to improve the quality of software by doing more testing is like trying to lose weight by weighing yourself more often.

– Steve McConnell

Quality is *not* just testing

Trying to improve the quality of software by doing more testing is like trying to lose weight by weighing yourself more often.

– Steve McConnell

Good tests localize problems to speed up debugging

Testing comparison of 7-digit phone numbers

Testing comparison of 7-digit phone numbers

10^7 possible numbers

Testing comparison of 7-digit phone numbers

10^7 possible numbers

$(10^7)^2$ possible pairs of numbers

Testing comparison of 7-digit phone numbers

10^7 possible numbers

$(10^7)^2$ possible pairs of numbers

At 10^6 million tests/sec, that's 155 days

Testing comparison of 7-digit phone numbers

10^7 possible numbers

$(10^7)^2$ possible pairs of numbers

At 10^6 million tests/sec, that's 155 days

...and then you start testing the next function

How do you know that your tests are correct?

"All" testing can do is show that

there *might* be a problem

"It might work in practice,

but it'll never work in theory."

If testing isn't easy, people won't do it

If testing isn't easy, people won't do it

Must be easy to:

If testing isn't easy, people won't do it

Must be easy to:

- add or change tests

If testing isn't easy, people won't do it

Must be easy to:

- add or change tests
- understand existing tests

If testing isn't easy, people won't do it

Must be easy to:

- add or change tests
- understand existing tests
- run tests

If testing isn't easy, people won't do it

Must be easy to:

- add or change tests
- understand existing tests
- run tests
- understand test results

If testing isn't easy, people won't do it

Must be easy to:

- add or change tests
- understand existing tests
- run tests
- understand test results

And test results must be reliable

If testing isn't easy, people won't do it

Must be easy to:

- add or change tests
- understand existing tests
- run tests
- understand test results

And test results must be reliable

- No false positives or false negatives

A *unit test* tests one component in a program

A *unit test* tests one component in a program

fixture

A *unit test* tests one component in a program

fixture

What the test is run on

A *unit test* tests one component in a program

fixture

action

A *unit test* tests one component in a program

fixture

action

What's done to the fixture

A *unit test* tests one component in a program

fixture

action

expected result

A *unit test* tests one component in a program

fixture

action

expected result

What *should* happen

A *unit test* tests one component in a program

fixture

action

expected result

---

actual result

A *unit test* tests one component in a program

fixture

action

expected result

---

actual result

What *actually* happened

A *unit test* tests one component in a program

fixture

action

expected result

---

actual result

report

A *unit test* tests one component in a program

fixture

action

expected result

---

actual result

report

Summary

Test `dna_starts_with`

Test `dna_starts_with`

`True` if second argument is a prefix of the first

Test `dna_starts_with`

`True` if second argument is a prefix of the first

`False` otherwise

Test `dna_starts_with`

`True` if second argument is a prefix of the first

`False` otherwise

```
dna_starts_with('actggt', 'act') => True
```

Test `dna_starts_with`

`True` if second argument is a prefix of the first

`False` otherwise

```
dna_starts_with('actggt', 'act') => True
```

```
dna_starts_with('actggt', 'ctg') => False
```

Test `dna_starts_with`

`True` if second argument is a prefix of the first

`False` otherwise

```
dna_starts_with('actggt', 'act') => True
```

```
dna_starts_with('actggt', 'agt') => False
```

Do this one from scratch to show ideas

Test `dna_starts_with`

`True` if second argument is a prefix of the first

`False` otherwise

```
dna_starts_with('actggt', 'act') => True
```

```
dna_starts_with('actggt', 'agt') => False
```

Do this one from scratch to show ideas

Then introduce a library that can take care of the repetitive bits

```
# Test directly
assert dna_starts_with('a', 'a')
assert dna_starts_with('at', 'a')
assert dna_starts_with('at', 'at')
assert not dna_starts_with('at', 't')
```

```
# Test directly
assert dna_starts_with('a', 'a')
assert dna_starts_with('at', 'a')
assert dna_starts_with('at', 'at')
assert not dna_starts_with('at', 't')
```

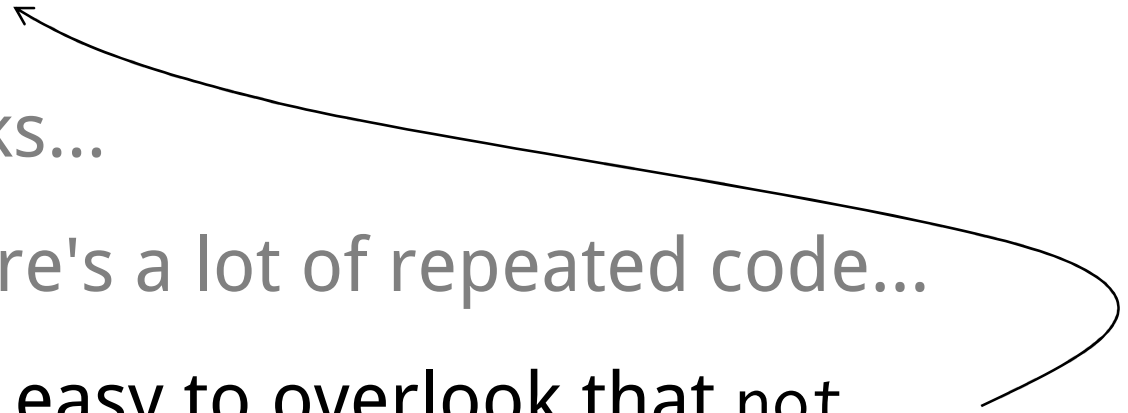
This works...

```
# Test directly
assert dna_starts_with('a', 'a')
assert dna_starts_with('at', 'a')
assert dna_starts_with('at', 'at')
assert not dna_starts_with('at', 't')
```

This works...

...but there's a lot of repeated code...

```
# Test directly
assert dna_starts_with('a', 'a')
assert dna_starts_with('at', 'a')
assert dna_starts_with('at', 'at')
assert not dna_starts_with('at', 't')
```



This works...

...but there's a lot of repeated code...

...and it's easy to overlook that `not`...

```
# Test directly
assert dna_starts_with('a', 'a')
assert dna_starts_with('at', 'a')
assert dna_starts_with('at', 'at')
assert not dna_starts_with('at', 't')
```

This works...

...but there's a lot of repeated code...

...and it's easy to overlook that `not`...

...and it only tests up to the first failure

```
# Tests in table
# Sequence   Prefix   Expected
Tests = [
  ['a',      'a',     True],
  ['at',     'a',     True],
  ['at',     'at',    True],
  ['at,      't',     False]
]
```

```
# Tests in table
# Sequence   Prefix   Expected
Tests = [
  ['a',      'a',     True],
  ['at',     'a',     True],
  ['at',     'at',    True],
  ['at,      't',     False]
]
```

Easy to read

```
# Tests in table
# Sequence   Prefix   Expected
Tests = [
  ['a',      'a',     True],
  ['at',     'a',     True],
  ['at',     'at',    True],
  ['at,      't',     False]
]
```

Easy to read

Easy to add new tests

```
# Run and report
passes = 0
for (seq, prefix, expected) in Tests:
  if dna_starts_with(seq, prefix) == expected:
    passes += 1
print '%d/%d tests passed' % (passes, len(Tests))
```

```
# Run and report
passes = 0
for (seq, prefix, expected) in Tests:
  if dna_starts_with(seq, prefix) == expected:
    passes += 1
print '%d/%d tests passed' % (passes, len(Tests))
```

No runnable code is copied when adding tests

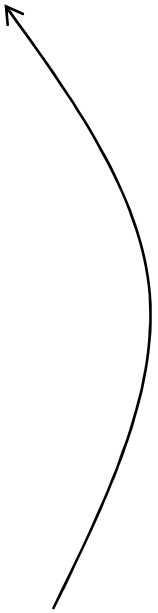
```
# Run and report
passes = 0
for (seq, prefix, expected) in Tests:
  if dna_starts_with(seq, prefix) == expected:
    passes += 1
print '%d/%d tests passed' % (passes, len(Tests))
```

No runnable code is copied when adding tests

But when tests fail, we don't know which ones

```
# Run and report
passes = 0
for (i, (seq, prefix, expected)) in enumerate(Tests):
  if dna_starts_with(seq, prefix) == expected:
    passes += 1
  else:
    print 'test %d failed' % i
print '%d/%d tests passed' % (passes, len(Tests))
```

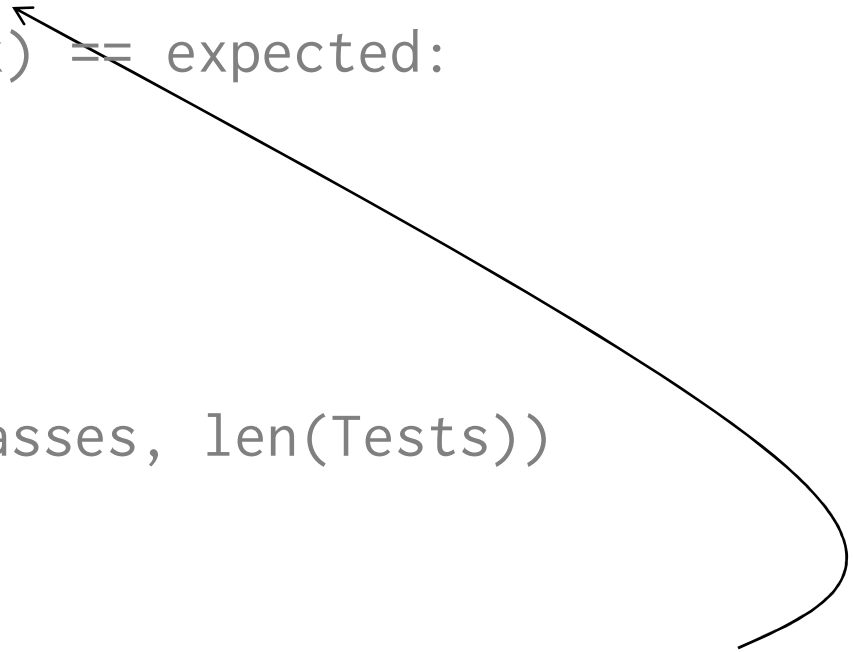
```
# Run and report
passes = 0
for (i, (seq, prefix, expected)) in enumerate(Tests):
  if dna_starts_with(seq, prefix) == expected:
    passes += 1
  else:
    print 'test %d failed' % i
print '%d/%d tests passed' % (passes, len(Tests))
```



Produces (index, element)

for each element of list

```
# Run and report
passes = 0
for (i, (seq, prefix, expected)) in enumerate(Tests):
  if dna_starts_with(seq, prefix) == expected:
    passes += 1
  else:
    print 'test %d failed' % i
print '%d/%d tests passed' % (passes, len(Tests))
```



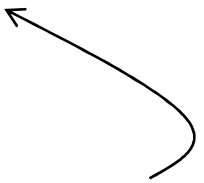
Decompose into variables

by matching structure

```
# Run and report
passes = 0
for (i, (seq, prefix, expected)) in enumerate(Tests):
  if dna_starts_with(seq, prefix) == expected:
    passes += 1
  else:
    print 'test %d failed' % i
print '%d/%d tests passed' % (passes, len(Tests))
```

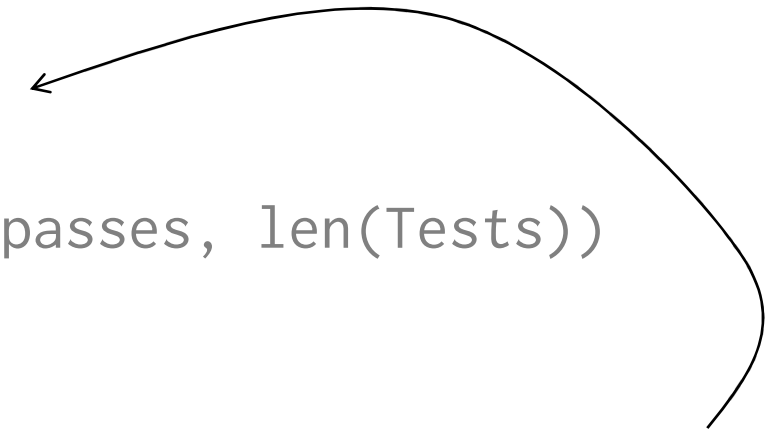
Test passes as before

```
# Run and report
passes = 0
for (i, (seq, prefix, expected)) in enumerate(Tests):
  if dna_starts_with(seq, prefix) == expected:
    passes += 1
  else:
    print 'test %d failed' % i
print '%d/%d tests passed' % (passes, len(Tests))
```



Summarize results that don't need attention

```
# Run and report
passes = 0
for (i, (seq, prefix, expected)) in enumerate(Tests):
  if dna_starts_with(seq, prefix) == expected:
    passes += 1
  else:
    print 'test %d failed' % i
print '%d/%d tests passed' % (passes, len(Tests))
```



Report each result that needs attention separately

This pattern is used for testing over and over

This pattern is used for testing over and over

Many libraries to support it in many languages

This pattern is used for testing over and over

Many libraries to support it in many languages

We'll look at one that comes with Python

This pattern is used for testing over and over

Many libraries to support it in many languages

We'll look at one that comes with Python

But first, we'll look at how to handle errors

in programs systematically

created by

Greg Wilson

July 2010