How to make the most of unit testing

Greg Detre

greg@gregdetre.co.uk
@gregdetre

memrise

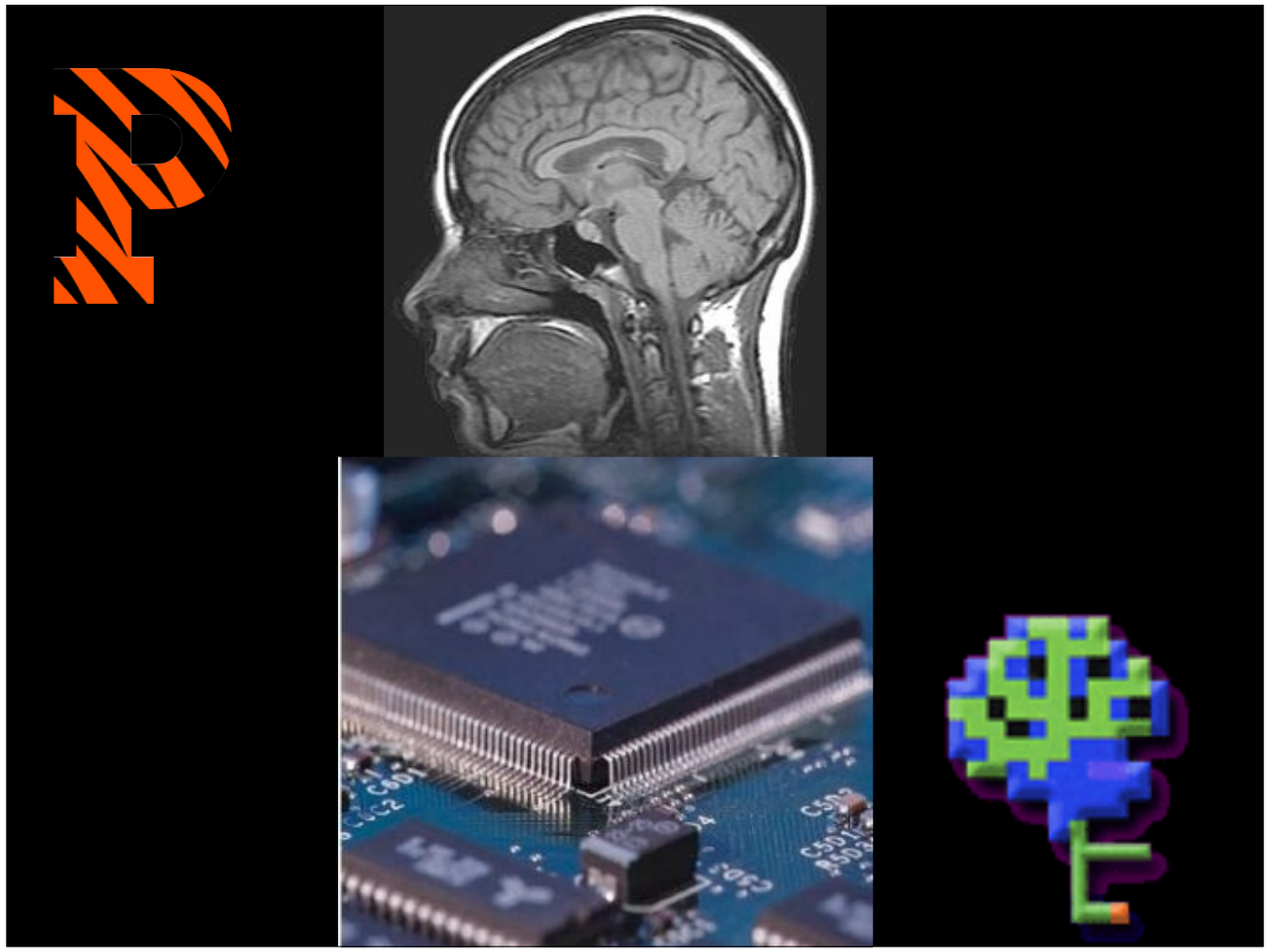
theguardian

The Tampa Bay Python Meetup Group
13th October, 2015

github.com/gregdetre/unit-testing-pres

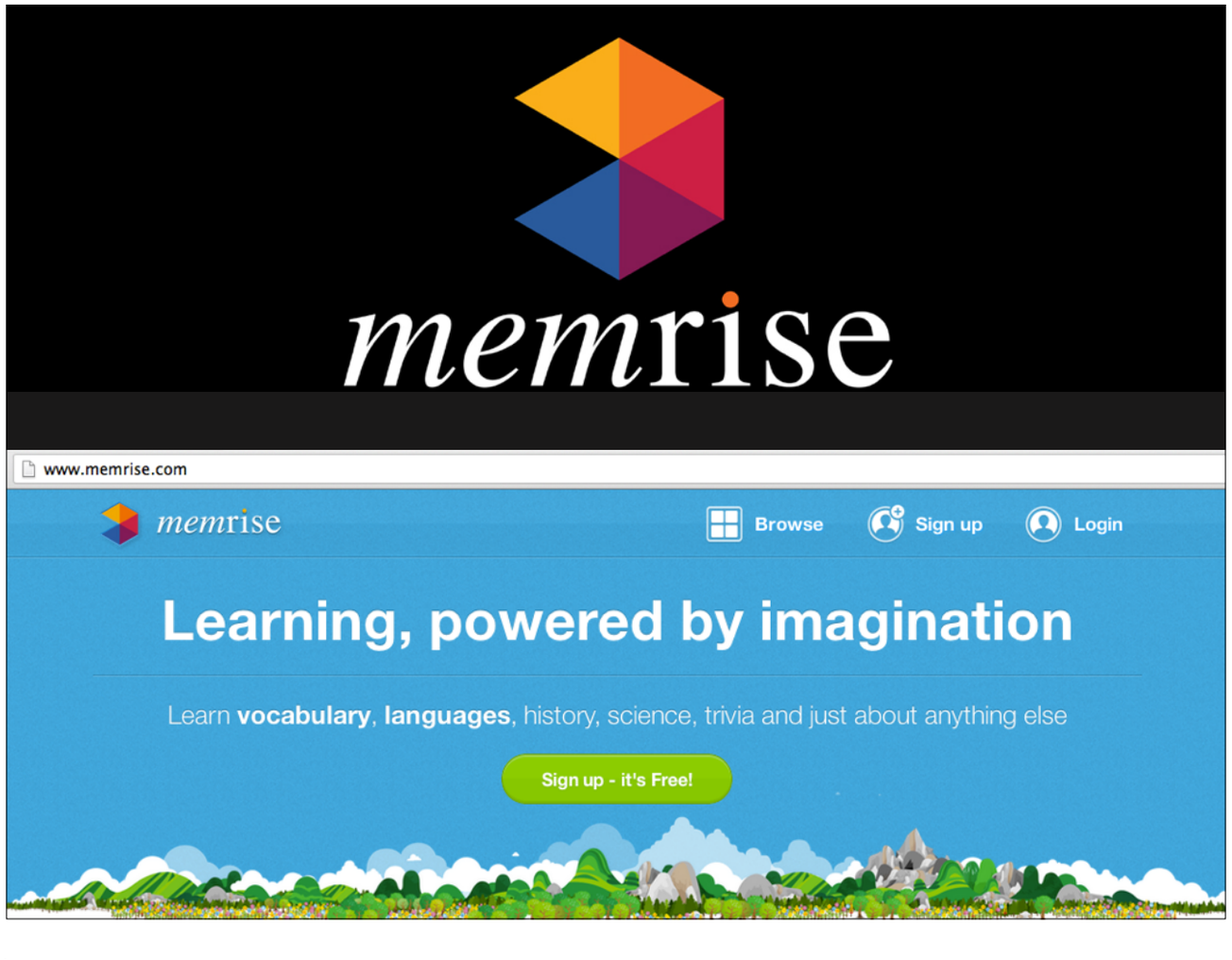
github.com/gregdetre/unit-testing-pres

ME



did my PhD work at Princeton in with Ken Norman in the Computational Memory Lab
I'm Greg Detre

has a PhD in the neuroscience of human memory and forgetting at Princeton
scan people's brains, including my own - it turned out to be smaller than I'd hoped



after grad school, I started a company called Memrise

Led the brand-new data science team at The Guardian

One more caveat - I haven't been working in the trenches as an engineer for the last year, and I'm very rusty on some things. I hope you'll help me along if I get stuck on something obvious!

WHAT IS UNIT TESTING?

What is unit testing?

What is unit testing?

Take the smallest piece of testable software in the application, isolate it from the remainder of the code, and determine whether it behaves exactly as you expect.

- msdn.microsoft.com/en-us/library/aa292197(v=vs.71).aspx

What is unit testing?

> *Take the smallest piece of testable software in the application, isolate it from the remainder of the code, and determine whether it behaves exactly as you expect.*
>
> - msdn.microsoft.com/en-us/library/aa292197(v=vs.71).aspx

e.g. if I call this function with input X, I expect to get output Y back

What is unit testing?

> *Take the smallest piece of testable software in the application, isolate it from the remainder of the code, and determine whether it behaves exactly as you expect.*
>
> - msdn.microsoft.com/en-us/library/aa292197(v=vs.71).aspx

e.g. if I call this function with input X, I expect to get output Y back

You already test manually as you're writing code.

You already test manually as you're writing code.

But that's inefficient.

SETUP

Docs (including this presentation)

github.com/gregdetre/unit-testing-pres/

tell Greg your GitHub account name and I'll give you commit access or submit pull requests

follow the README.md *Setup* instructions

Google Hangout (so we can share your screen)

http://bit.ly/1GELH73

join in, share your screen, turn off speakers, mute

Python unit testing frameworks

unittest - standard library

nose - just like unittest, but nicer

Python unit testing frameworks

unittest - standard library
nose - just like unittest, but nicer

```
$ pip install nose2
$ nose2
```

plugins, e.g.

- colored output
- autodiscovery
- coverage
- debug on error

```
# in test_template0.py
def test_blah():
    assert True


$ nose2 test_template0
```

INTERACTIVE

Goal

to write a function that identifies all and only legitimate email addresses

for now, don't look online

```
def isitanemail(e):
    # 'a@b.com', 'x@@y' -> True
    # 'ab.com' -> False
    return '@' in e
```

team 1 = write a better implementation of `isitanemail()`

team 2 = write a set of tests for email addresses that *should* be allowed, e.g.

```
def test_legal_basic():
    assert isitanemail('ab@cd.com')
```

team 3 = write a set of tests for email addresses that *shouldn't* be allowed, e.g.

```
def test_illegal_no_tld():
    assert not isitanemail('ab@cd')
```

TIME’S UP!

Team 1 - send me your code

Team 1 - send me your code

Team 2 - let’s try running your tests

Team 1 - send me your code

Team 2 - let’s try running your tests

Team 3 - let’s try running your tests

Team 1 - send me your code

Team 2 - let’s try running your tests

Team 3 - let’s try running your tests

What did it feel like to write tests without code to run them against?

Team 1 - send me your code

Team 2 - let’s try running your tests

Team 3 - let’s try running your tests

What did it feel like to write tests without code to run them against?

I’ll commit the new code & tests

Team 1 - send me your code

Team 2 - let’s try running your tests

Team 3 - let’s try running your tests

What did it feel like to write tests without code to run them against?

I’ll commit the new code & tests

Get all tests passing

Team 1 - send me your code

Team 2 - let's try running your tests

Team 3 - let's try running your tests

What did it feel like to write tests without code to run them against?

I'll commit the new code & tests

Get all tests passing

- Did it feel different coding when you have tests?

Next steps?

Define exceptions, then test & build them

Completely rewrite isitanemail() while making tests pass - open book

Modularize - check TLD, check illegal characters

Black vs white box

BENEFITS

Find more bugs earlier (and more cheaply)

*Shull et al (2002) estimate that non-severe defects take approximately **14 hours of debugging effort after release**, but **only 7.4 hours before release** .*

Find more bugs earlier (and more cheaply)

*Shull et al (2002) estimate that non-severe defects take approximately **14 hours of debugging effort after release**, but **only 7.4 hours before release** . … However, the multiplier becomes much, much larger for severe bugs: … severe bugs are **100 times more expensive to fix after shipping than they are to fix before shipping.***

https://kev.inburke.com/kevin/the-best-ways-to-find-bugs-in-your-code/

Pretty much every study [on test-driven development] showed an increased effort (15-60%) and an increase in quality (5-267%). The ratio of effort to quality is 1:2 - so [test driven development] seems to pay.

http://morenews.blogspot.com/2007/08/tdd-results-are-in.html

Though see ‘As well as testing’

Develop faster

Find new bugs when you introduce them:

- Let's say you make a change that introduces a subtle new bug. But you don't realize it at the time. A while later, that bug shows up. But you don't realize that it's been there for ages. So you inspect all the code that's been changed recently and waste a lot of time.
- If you had better tests, you might have noticed the bug when you made the change that introduced it.

Help realize if the error is elsewhere:

- Let's say you're investigating a bug. The error appears in one part of the code, so that's where you're looking. But actually, it turns out to be a problem that was caused upstream, but is only being noticed over here.
- If you had better tests, you might have noticed that your upstream code wasn't handling things correctly, or that those tests are now failing.

Develop faster

Manually testing as you go is slow and incomplete

Find new bugs when you introduce them:

- Let's say you make a change that introduces a subtle new bug. But you don't realize it at the time. A while later, that bug shows up. But you don't realize that it's been there for ages. So you inspect all the code that's been changed recently and waste a lot of time.
- If you had better tests, you might have noticed the bug when you made the change that introduced it.

Help realize if the error is elsewhere:

- Let's say you're investigating a bug. The error appears in one part of the code, so that's where you're looking. But actually, it turns out to be a problem that was caused upstream, but is only being noticed over here.
- If you had better tests, you might have noticed that your upstream code wasn't handling things correctly, or that those tests are now failing.

Develop faster

Manually testing as you go is slow and incomplete

Most development time is spent debugging. Unit tests take more time up front, but help you debug much faster

Find new bugs when you introduce them:

- Let's say you make a change that introduces a subtle new bug. But you don't realize it at the time. A while later, that bug shows up. But you don't realize that it's been there for ages. So you inspect all the code that's been changed recently and waste a lot of time.
- If you had better tests, you might have noticed the bug when you made the change that introduced it.

Help realize if the error is elsewhere:

- Let's say you're investigating a bug. The error appears in one part of the code, so that's where you're looking. But actually, it turns out to be a problem that was caused upstream, but is only being noticed over here.
- If you had better tests, you might have noticed that your upstream code wasn't handling things correctly, or that those tests are now failing.

Develop faster

Manually testing as you go is slow and incomplete

Most development time is spent debugging. Unit tests take more time up front, but help you debug much faster

Help you isolate what is and is not working.

Find new bugs when you introduce them:

- Let's say you make a change that introduces a subtle new bug. But you don't realize it at the time. A while later, that bug shows up. But you don't realize that it's been there for ages. So you inspect all the code that's been changed recently and waste a lot of time.
- If you had better tests, you might have noticed the bug when you made the change that introduced it.

Help realize if the error is elsewhere:

- Let's say you're investigating a bug. The error appears in one part of the code, so that's where you're looking. But actually, it turns out to be a problem that was caused upstream, but is only being noticed over here.
- If you had better tests, you might have noticed that your upstream code wasn't handling things correctly, or that those tests are now failing.

Develop faster

Manually testing as you go is slow and incomplete

Most development time is spent debugging. Unit tests take more time up front, but help you debug much faster

Help you isolate what is and is not working.

You make a change that fixes the issue you were thinking about, but your tests highlight that you've inadvertently created a problem elsewhere.

Find new bugs when you introduce them:

- Let's say you make a change that introduces a subtle new bug. But you don't realize it at the time. A while later, that bug shows up. But you don't realize that it's been there for ages. So you inspect all the code that's been changed recently and waste a lot of time.
- If you had better tests, you might have noticed the bug when you made the change that introduced it.

Help realize if the error is elsewhere:

- Let's say you're investigating a bug. The error appears in one part of the code, so that's where you're looking. But actually, it turns out to be a problem that was caused upstream, but is only being noticed over here.
- If you had better tests, you might have noticed that your upstream code wasn't handling things correctly, or that those tests are now failing.

Develop faster

Manually testing as you go is slow and incomplete

Most development time is spent debugging. Unit tests take more time up front, but help you debug much faster

- Help you isolate what is and is not working.
- You make a change that fixes the issue you were thinking about, but your tests highlight that you've inadvertently created a problem elsewhere.
- Your tests pause the debugger exactly where the problem is.

Find new bugs when you introduce them:

- Let's say you make a change that introduces a subtle new bug. But you don't realize it at the time. A while later, that bug shows up. But you don't realize that it's been there for ages. So you inspect all the code that's been changed recently and waste a lot of time.
- If you had better tests, you might have noticed the bug when you made the change that introduced it.

Help realize if the error is elsewhere:

- Let's say you're investigating a bug. The error appears in one part of the code, so that's where you're looking. But actually, it turns out to be a problem that was caused upstream, but is only being noticed over here.
- If you had better tests, you might have noticed that your upstream code wasn't handling things correctly, or that those tests are now failing.

Develop faster

Manually testing as you go is slow and incomplete

Most development time is spent debugging. Unit tests take more time up front, but help you debug much faster

Help you isolate what is and is not working.

You make a change that fixes the issue you were thinking about, but your tests highlight that you've inadvertently created a problem elsewhere.

Your tests pause the debugger exactly where the problem is.

Find new bugs when you introduce them (when it's easy to fix), not months later

Find new bugs when you introduce them:

- Let's say you make a change that introduces a subtle new bug. But you don't realize it at the time. A while later, that bug shows up. But you don't realize that it's been there for ages. So you inspect all the code that's been changed recently and waste a lot of time.
- If you had better tests, you might have noticed the bug when you made the change that introduced it.

Help realize if the error is elsewhere:

- Let's say you're investigating a bug. The error appears in one part of the code, so that's where you're looking. But actually, it turns out to be a problem that was caused upstream, but is only being noticed over here.
- If you had better tests, you might have noticed that your upstream code wasn't handling things correctly, or that those tests are now failing.

Develop faster

Manually testing as you go is slow and incomplete

Most development time is spent debugging. Unit tests take more time up front, but help you debug much faster

Help you isolate what is and is not working.

You make a change that fixes the issue you were thinking about, but your tests highlight that you've inadvertently created a problem elsewhere.

Your tests pause the debugger exactly where the problem is.

Find new bugs when you introduce them (when it's easy to fix), not months later

Help realize if the error is elsewhere from where it manifests

Find new bugs when you introduce them:

- Let's say you make a change that introduces a subtle new bug. But you don't realize it at the time. A while later, that bug shows up. But you don't realize that it's been there for ages. So you inspect all the code that's been changed recently and waste a lot of time.
- If you had better tests, you might have noticed the bug when you made the change that introduced it.

Help realize if the error is elsewhere:

- Let's say you're investigating a bug. The error appears in one part of the code, so that's where you're looking. But actually, it turns out to be a problem that was caused upstream, but is only being noticed over here.
- If you had better tests, you might have noticed that your upstream code wasn't handling things correctly, or that those tests are now failing.

Develop faster

Manually testing as you go is slow and incomplete

Most development time is spent debugging. Unit tests take more time up front, but help you debug much faster

Help you isolate what is and is not working.

You make a change that fixes the issue you were thinking about, but your tests highlight that you've inadvertently created a problem elsewhere.

Your tests pause the debugger exactly where the problem is.

Find new bugs when you introduce them (when it's easy to fix), not months later

Help realize if the error is elsewhere from where it manifests

Easy to change/refactor code confidently

Find new bugs when you introduce them:

- Let's say you make a change that introduces a subtle new bug. But you don't realize it at the time. A while later, that bug shows up. But you don't realize that it's been there for ages. So you inspect all the code that's been changed recently and waste a lot of time.
- If you had better tests, you might have noticed the bug when you made the change that introduced it.

Help realize if the error is elsewhere:

- Let's say you're investigating a bug. The error appears in one part of the code, so that's where you're looking. But actually, it turns out to be a problem that was caused upstream, but is only being noticed over here.
- If you had better tests, you might have noticed that your upstream code wasn't handling things correctly, or that those tests are now failing.

Develop faster

Manually testing as you go is slow and incomplete

Most development time is spent debugging. Unit tests take more time up front, but help you debug much faster

Help you isolate what is and is not working.

You make a change that fixes the issue you were thinking about, but your tests highlight that you've inadvertently created a problem elsewhere.

Your tests pause the debugger exactly where the problem is.

Find new bugs when you introduce them (when it's easy to fix), not months later

Help realize if the error is elsewhere from where it manifests

Easy to change/refactor code confidently

You know when you're done

Find new bugs when you introduce them:

- Let's say you make a change that introduces a subtle new bug. But you don't realize it at the time. A while later, that bug shows up. But you don't realize that it's been there for ages. So you inspect all the code that's been changed recently and waste a lot of time.
- If you had better tests, you might have noticed the bug when you made the change that introduced it.

Help realize if the error is elsewhere:

- Let's say you're investigating a bug. The error appears in one part of the code, so that's where you're looking. But actually, it turns out to be a problem that was caused upstream, but is only being noticed over here.
- If you had better tests, you might have noticed that your upstream code wasn't handling things correctly, or that those tests are now failing.

Write better code

- Dependencies - if you have to load in 5 other things and setup multiple arguments that aren't strictly required for a function to do its job, it'll be a lot more work to set up tests for it. So instead, you'll find yourself minimizing the argument and dependencies to make each function do one simple thing well and require as few arguments as possible.

- Interface - if you write the tests early in the development, you'll be using the interface before you've written the implementation. If it feels ugly or the abstractions are wrong, you'll be able to tell early, while it's still easy to change it.

- Entrances & complexity - If you know that you're going to have to write tests for each entrance and each fork in the logic, there's a pressure to minimise those. Tests make you conscious of the costs of each entrance and increase in cyclomatic complexity.

We are concrete thinkers. We find it easier to reason about examples than abstractions.

Write better code

Writing tests encourages you to:

- Dependencies - if you have to load in 5 other things and setup multiple arguments that aren't strictly required for a function to do its job, it'll be a lot more work to set up tests for it. So instead, you'll find yourself minimizing the argument and dependencies to make each function do one simple thing well and require as few arguments as possible.

- Interface - if you write the tests early in the development, you'll be using the interface before you've written the implementation. If it feels ugly or the abstractions are wrong, you'll be able to tell early, while it's still easy to change it.

- Entrances & complexity - If you know that you're going to have to write tests for each entrance and each fork in the logic, there's a pressure to minimise those. Tests make you conscious of the costs of each entrance and increase in cyclomatic complexity.

We are concrete thinkers. We find it easier to reason about examples than abstractions.

Write better code

Writing tests encourages you to:

- Reduce dependencies

- Dependencies - if you have to load in 5 other things and setup multiple arguments that aren't strictly required for a function to do its job, it'll be a lot more work to set up tests for it. So instead, you'll find yourself minimizing the argument and dependencies to make each function do one simple thing well and require as few arguments as possible.

- Interface - if you write the tests early in the development, you'll be using the interface before you've written the implementation. If it feels ugly or the abstractions are wrong, you'll be able to tell early, while it's still easy to change it.

- Entrances & complexity - If you know that you're going to have to write tests for each entrance and each fork in the logic, there's a pressure to minimise those. Tests make you conscious of the costs of each entrance and increase in cyclomatic complexity.

We are concrete thinkers. We find it easier to reason about examples than abstractions.

Write better code

Writing tests encourages you to:

- Reduce dependencies
- Think about the interface

- Dependencies - if you have to load in 5 other things and setup multiple arguments that aren't strictly required for a function to do its job, it'll be a lot more work to set up tests for it. So instead, you'll find yourself minimizing the argument and dependencies to make each function do one simple thing well and require as few arguments as possible.

- Interface - if you write the tests early in the development, you'll be using the interface before you've written the implementation. If it feels ugly or the abstractions are wrong, you'll be able to tell early, while it's still easy to change it.

- Entrances & complexity - If you know that you're going to have to write tests for each entrance and each fork in the logic, there's a pressure to minimise those. Tests make you conscious of the costs of each entrance and increase in cyclomatic complexity.

We are concrete thinkers. We find it easier to reason about examples than abstractions.

Write better code

Writing tests encourages you to:

- Reduce dependencies
- Think about the interface
- Minimise entrances and complexity

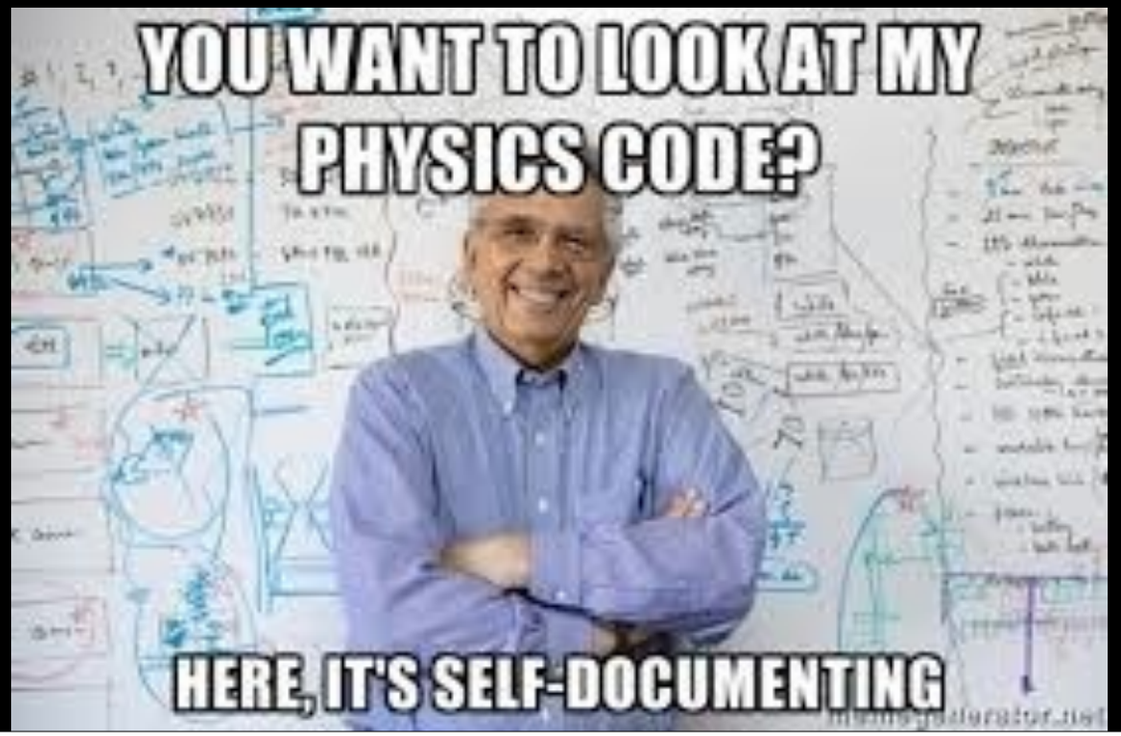
- Dependencies - if you have to load in 5 other things and setup multiple arguments that aren't strictly required for a function to do its job, it'll be a lot more work to set up tests for it. So instead, you'll find yourself minimizing the argument and dependencies to make each function do one simple thing well and require as few arguments as possible.

- Interface - if you write the tests early in the development, you'll be using the interface before you've written the implementation. If it feels ugly or the abstractions are wrong, you'll be able to tell early, while it's still easy to change it.

- Entrances & complexity - If you know that you're going to have to write tests for each entrance and each fork in the logic, there's a pressure to minimise those. Tests make you conscious of the costs of each entrance and increase in cyclomatic complexity.

We are concrete thinkers. We find it easier to reason about examples than abstractions.

Self-documenting



Benefits of testing

Helps you structure your code – if it's easy to test, it'll be easy to understand and refactor

Easier to read than code, how-to guide and expected behavior

Run your unit tests every time you deploy. Otherwise you might break something that used to work, and not realize it

Confidence – there may be bugs. But there are no easy bugs. No new bugs. If you have good tests, you might even be somewhat confident there are no bugs.

Benefits of testing

Find more bugs earlier and more cheaply

Helps you structure your code – if it's easy to test, it'll be easy to understand and refactor

Easier to read than code, how-to guide and expected behavior

Run your unit tests every time you deploy. Otherwise you might break something that used to work, and not realize it

Confidence – there may be bugs. But there are no easy bugs. No new bugs. If you have good tests, you might even be somewhat confident there are no bugs.

Benefits of testing

Find more bugs earlier and more cheaply

Develop faster

Helps you structure your code – if it's easy to test, it'll be easy to understand and refactor

Easier to read than code, how-to guide and expected behavior

Run your unit tests every time you deploy. Otherwise you might break something that used to work, and not realize it

Confidence – there may be bugs. But there are no easy bugs. No new bugs. If you have good tests, you might even be somewhat confident there are no bugs.

Benefits of testing

Find more bugs earlier and more cheaply

Develop faster

Write better code

Helps you structure your code – if it's easy to test, it'll be easy to understand and refactor

Easier to read than code, how-to guide and expected behavior

Run your unit tests every time you deploy. Otherwise you might break something that used to work, and not realize it

Confidence – there may be bugs. But there are no easy bugs. No new bugs. If you have good tests, you might even be somewhat confident there are no bugs.

Benefits of testing

Find more bugs earlier and more cheaply

Develop faster

Write better code

Self-documenting

Helps you structure your code – if it's easy to test, it'll be easy to understand and refactor

Easier to read than code, how-to guide and expected behavior

Run your unit tests every time you deploy. Otherwise you might break something that used to work, and not realize it

Confidence – there may be bugs. But there are no easy bugs. No new bugs. If you have good tests, you might even be somewhat confident there are no bugs.

Benefits of testing

Find more bugs earlier and more cheaply

Develop faster

Write better code

Self-documenting

Guard against new bugs in old code

Helps you structure your code – if it's easy to test, it'll be easy to understand and refactor

Easier to read than code, how-to guide and expected behavior

Run your unit tests every time you deploy. Otherwise you might break something that used to work, and not realize it

Confidence – there may be bugs. But there are no easy bugs. No new bugs. If you have good tests, you might even be somewhat confident there are no bugs.

Benefits of testing

Find more bugs earlier and more cheaply

Develop faster

Write better code

Self-documenting

Guard against new bugs in old code

Integrate with others (API, in a team)

Helps you structure your code – if it's easy to test, it'll be easy to understand and refactor

Easier to read than code, how-to guide and expected behavior

Run your unit tests every time you deploy. Otherwise you might break something that used to work, and not realize it

Confidence – there may be bugs. But there are no easy bugs. No new bugs. If you have good tests, you might even be somewhat confident there are no bugs.

Benefits of testing

Find more bugs earlier and more cheaply

Develop faster

Write better code

Self-documenting

Guard against new bugs in old code

Integrate with others (API, in a team)

More predictable progress

Helps you structure your code – if it's easy to test, it'll be easy to understand and refactor

Easier to read than code, how-to guide and expected behavior

Run your unit tests every time you deploy. Otherwise you might break something that used to work, and not realize it

Confidence – there may be bugs. But there are no easy bugs. No new bugs. If you have good tests, you might even be somewhat confident there are no bugs.

Benefits of testing

Find more bugs earlier and more cheaply

Develop faster

Write better code

Self-documenting

Guard against new bugs in old code

Integrate with others (API, in a team)

More predictable progress

Feel confident you've done a good job

Helps you structure your code - if it's easy to test, it'll be easy to understand and refactor

Easier to read than code, how-to guide and expected behavior

Run your unit tests every time you deploy. Otherwise you might break something that used to work, and not realize it

Confidence - there may be bugs. But there are no easy bugs. No new bugs. If you have good tests, you might even be somewhat confident there are no bugs.

WHAT MAKES A GOOD UNIT TEST?

What makes a bad
unit test?

see http://artofunittesting.com/definition-of-a-unit-test/

```
def test_isitanemail_ints:
     for num in range(1, 1000000000):
          # e.g. ‘123@example.com’
          e = ‘%i@example.com’ % num
          assert isitanemail(e)
```

This isn’t going to be fast to run. If it’s not fast, you won’t want to run it often.

It’s not buying you anything. Just test a few representative samples. Be concrete, simple, readable, make the point of the test clear.

After all, if ‘123@example.com’ passes, probably so will ‘456@example.com’…

```
# test_raw_input.py
def test_isitanemail_ask():
     e = ‘x@example.com’
     print ‘Returned for %s:’ % e, isitanemail(e)
     assert raw_input(‘Ok?’) == ‘y’
```

This isn’t going to be fast to run. If it’s not fast, you won’t want to run it often.

It’s not buying you anything. Just test a few representative samples. Be concrete, simple, readable, make the point of the test clear.

After all, if ‘123@example.com’ passes, probably so will ‘456@example.com’…

```
from random import sample

def randstr():
     # e.g. ‘a1+’, ‘c3b’, ‘!.!’
     return ‘’.join(sample(‘abc123#+!.’, 3))

def test_isitanemail_rand():
     e = ‘%s@%s.%s’ % (randstr(), \
                       randstr(), \
                       randstr())
     assert isitanemail(e)
```

The problem with this is that it might fail some of the time… it’s not consistent.

However, this kind of ‘fuzz testing’ is occasionally useful, for making sure that your function is robust. For instance, if you had a big dataset of real-world email addresses, perhaps you could pipe in a random sample each time…

```
from blah.dns import validate_dns
from your.database import query_user_by_email

def isitanemail(e):
     if ‘@‘ not in e:
          return False
     if not validate_dns(e):
          return False
     if not query_user_by_email(e):
          return False
```

If you don’t have internet access, this test isn’t going to run (because it won’t be able to validate against the DNS).

This requires you to have your database set up as well, along with a bunch of populated data (how is that specified). Sometimes, prepopulating your database is part of the test, but maybe not here.

It’s going to be slow.

These are conceptually separate problems. Validating the form of the address is a distinct problem from validating whether it’s a real address or whether it’s an address in your database, and they should be separate functions, with separate tests. This is what the ‘unit’ refers to. Remember our initial definition: “Take the smallest piece of testable software in the application, isolate it from the remainder of the code, and determine whether it behaves exactly as you expect.”

An ideal unit test

Fully automated
Focus on a single, minimal ‘unit’
Readable, concrete
Independent
    from other tests/full system (see ‘mocks’)
Fast
Consistent
Comprehensive

see http://artofunittesting.com/definition-of-a-unit-test/

Tips

Tips

Concrete and easy to understand

- DAMP not DRY -"Descriptive and Meaningful Phrases”
- Implement the test a different way from the original function

Tips

- Concrete and easy to understand
  - DAMP not DRY -"Descriptive and Meaningful Phrases”
  - Implement the test a different way from the original function
- Write your tests early in the development cycle

Tips

- Concrete and easy to understand
  - DAMP not DRY -"Descriptive and Meaningful Phrases”
  - Implement the test a different way from the original function
- Write your tests early in the development cycle
- Keep them up to date, and always passing

Tips

- Concrete and easy to understand
  - DAMP not DRY -"Descriptive and Meaningful Phrases”
  - Implement the test a different way from the original function
- Write your tests early in the development cycle
- Keep them up to date, and always passing
- Run tests with every deploy

Tips

- Concrete and easy to understand
  - DAMP not DRY -"Descriptive and Meaningful Phrases”
  - Implement the test a different way from the original function
- Write your tests early in the development cycle
- Keep them up to date, and always passing
- Run tests with every deploy
- Make sure your test fails before it passes

Tips

- Concrete and easy to understand
  - DAMP not DRY -"Descriptive and Meaningful Phrases”
  - Implement the test a different way from the original function
- Write your tests early in the development cycle
- Keep them up to date, and always passing
- Run tests with every deploy
- Make sure your test fails before it passes
- Test a representative sample

TEST-DRIVEN DEVELOPMENT

Test-driven development

Think about your interface.

Write stub functions.

Write tests against those stub functions. They'll all fail.

Slowly fill out the stubs until your tests pass.

You're done!

CHOOSE YOUR OWN ADVENTURE

Other bug-finding techniques (code review, QA etc)

Data, algorithm and analysis testing

Testing performance

When is it hard to unit test?

More interactive

Nose plugins…

AS WELL AS TESTING

Combine software testing with other techniques

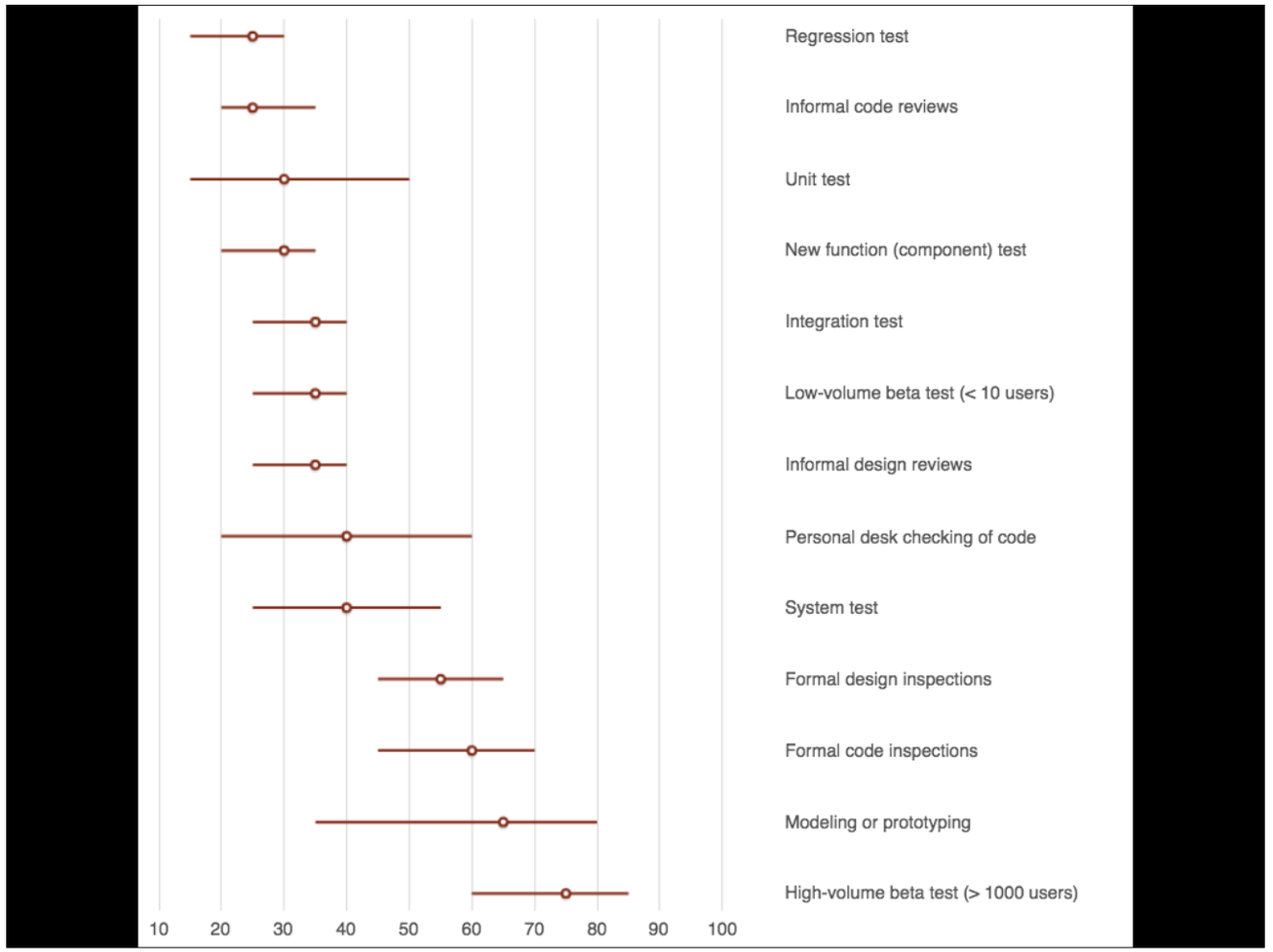
Automated testing finds a certain proportion (<40%) and type of bugs

If you want to ship high quality code, you should invest in more than one of formal code review, design inspection, testing, and quality assurance.

… according to Basili and Selby (1987), code reading detected 80 percent more faults per hour than testing…

https://kev.inburke.com/kevin/the-best-ways-to-find-bugs-in-your-code/

Consider code reviews, QA, beta testing, pair programming, and dog-fooding your own product.


Regression test
Informal code reviews
Unit test
New function (component) test
Integration test
Low-volume beta test (< 10 users)
Informal design reviews
Personal desk checking of code
System test
Formal design inspections
Formal code inspections
Modeling or prototyping
High-volume beta test (> 1000 users)
10
20
30
40
50
60
70
80
90
100

Different kinds of automated testing

unit vs integration tests

DATA & ALGORITHMS

How do you eat an elephant?

Validate on small data, build up

- Define your metric
- Run it on small data (quick, while prototyping)
- Show that you get better as you add more data

how do you eat an elephant? one bite at a time. start small, with a tiny subset of your data. that way, the algorithm runs quickly while you're prototyping

Fake data

Generate data that looks exactly the way you expect

Can be hard to do, but often helps you think things through

Confirm that the output looks as it should

Useful for orienting audience in presentations

Nonsense/scrambled data

Set a trap. Feed your algorithm nonsense data. It had better tell you the results aren't significant!

Easy: shuffle labels or feed in random numbers as data

This. Will Save. Your. Bacon.

e.g. guard against peeking

compare against ground truth if you have one

e.g. previous implementation, simpler version of the algorithm, doing it once by hand on real data

Defensive coding

Pepper your code with asserts and sanity checks

- e.g. confirm the dimensions, range of values, type of values

Fail immediately if things are wrong

- that way you'll notice early on in time and near to the cause of the problem
- rather than 2 weeks later and in a downstream part of the analysis

WHAT
ABOUT…?

What about…

- If the interface is changing very fast?
- If most of the work is being done by an external library?
- If the hard part is in the integration, not the pieces?
- If it requires a lot of infrastructure to be in place?
- If I’m in a really big hurry?

Which of these is the odd one out?

PERFORMANCE

Will this function/ algorithm/query scale?

measure

    time1 = running on some small N

    time2 = running on 100N

    coefficient = float(time2) / time1

assert(coefficient < 200)

maybe cast to floats as part of the assert

THE END

APPENDIX

Resources

https://docs.python.org/2/library/unittest.html
http://nose2.readthedocs.org/en/latest/index.html

Mark Pilgrim's (free) Dive Into Python chapters 13 and 14 on unit testing

http://www.diveintopython.net/unit_testing/index.html#roman.intro

http://nedbatchelder.com/text/test0.html

http://kev.inburke.com/kevin/the-best-ways-to-find-bugs-in-your-code/

back to Benefits