Unit Testing
Statistics 650/750
Week 3 Tuesday

Alex Reinhart and Christopher Genovese

12 Sep 2017

Announcements

- We're soon going to be doing a lot more programming: if you have little previous programming experience, please talk to us ASAP
- Assignment categories added; see `problem-bank/README.org` for details
- Resources on text editors and programming languages added to `documents/Info`
- You should merge your assignments to the `master` branch *after* they're marked Approved
- Please use descriptive pull request titles – indicate which assignment they're for at least. Think of a PR as an email to a real person, asking them to review your code
- Turn on RStudio's R style diagnostics if you're using it
  - under Preferences/Options -> Code -> Diagnostics -> Provide R style diagnostics
- Use unit testing in future assignments
- Office hours!

Day	Time	Person	Location
Monday	2:30-3:30	Chris	BH 232E
Tuesday	4-5	Justin	BH 232K
Wednesday	1:30-2:30	Chris	BH 232E
Wednesday	2:30-3:30	Alex	BH 132Q
Thursday	3-4	Chris	BH 232E
Friday	2:30-3:30	Alex	BH 132A

And you can always email us or catch us after class for appointments.

A Tip for Windows Users

You can have Git use Notepad as its default editor for commit messages:

```
git config --global core.editor notepad
```

No more getting stuck in Vim!

Unit Testing

Does this work?

Complexity can have consequences!

Whether it is

- the crash of the Mars Climate Orbiter (1998),
- a failure of the national telephone network (1990),
- a deadly medical device (1985, 2000),
- a massive Northeastern blackout (2003),
- the Heartbleed, Goto Fail, Shellshock exploits (2012–2014),
- an early warning failure that almost caused World War III (1983), or
- a 15-year-old fMRI analysis software bug that inflated significance levels (2015),

bugs will happen. It is hard to know whether a piece of software is actually doing what it is supposed to do. It is easy to write a thousand lines of research code, then discover that your results have been wrong for months.

Discipline, design, and careful thought are all helpful in producing working software. But even more important is

effective testing

and that is the central topic for today.

A Common (Interactive) Workflow

1. Write a function.
2. Try some reasonable values at the REPL to check that it works.
3. If there are problems, maybe insert some print statements, and modify the function.
4. Repeat until things seem fine.

(REPL: Read-Eval-Print Loop, the console at which you type commands and view results, such as you might use in RStudio or IPython.)

A similar pattern is common for a compiled workflow.

This will leave you with a lot of bugs. Worse, you can't keep track of what cases you tested, so when you return to edit your code, you can't be sure your edits don't break some of those cases.

For example, suppose you're modeling some binomial data. One argument to your function is the number of successes in the trials. One day you decide it's more convenient to use the number of *failures*, so you change the function and the functions that call it. But you've forgotten about some other functions which call it – and you may never notice that they're now giving the wrong answers. If you do, it may take hours to track down the cause.

This sounds far-fetched, but variations on it happen *all the time.*

So how can we more systematically test our code for correctness?

One method is unit testing.

Unit Testing

A "unit" is a vaguely defined concept that is intended to represent a small, well-defined piece of code. A unit is usually a function, method, class, module, or small group of related classes.

A test is simply some code which calls the unit with some inputs and checks that its answer matches an expected output.

Unit testing consists of writing tests that are

- focused on a small, low-level piece of code (a unit)
- typically written by the programmer with standard tools
- fast to run (so can be run often, i.e. before every commit).

The benefits of unit testing are many, including

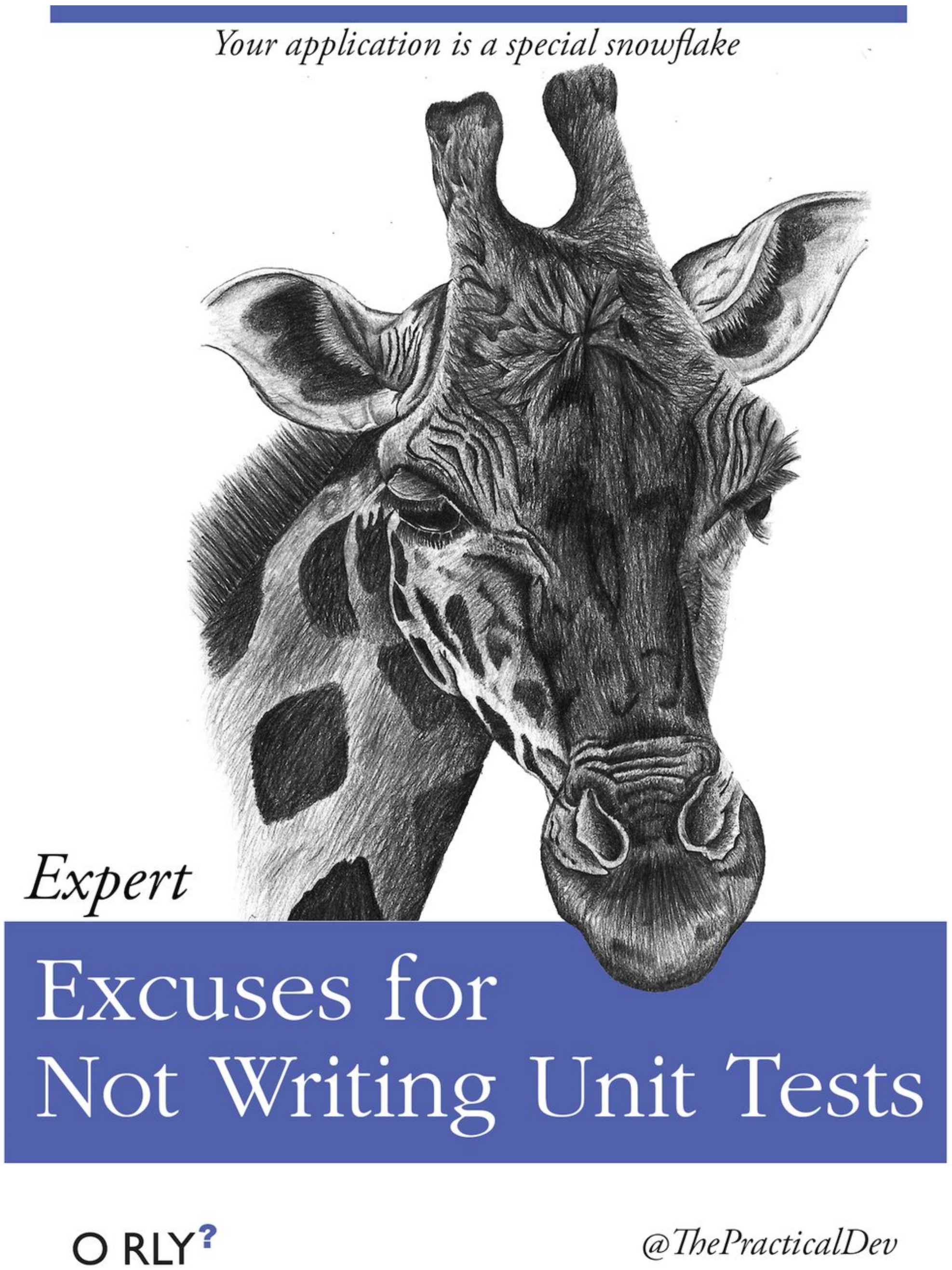
- Exposing problems early

- Making it easy to change (refactor) code without forgetting pieces or breaking things
- Simplifying integration of components
- Providing natural documentation of what the code should do
- Driving the design of new code.

Research has shown that unit testing costs you in time written for tests, but dramatically reduces the number of bugs later found in the software.

Personal experience has shown that nothing is quite as satisfying as writing a bunch of tests for some code, realizing you can rewrite the code in a much better way, and having it pass all the tests on the first try.

Nonetheless,


Your application is a special snowflake
Expert
Excuses for Not Writing Unit Tests
O RLY?
@ThePracticalDev

Components of a Unit Testing Framework

A test is a collection of one or more assertions executed in sequence, within a self-contained environment. The test fails if any of those assertions fail.

Each test should focus on a single function or feature and should be well-named so that a failed test lets you know precisely where to look in your code for the problem:

```r
test_that("Conway's rules are correct", {
    # conway_rules(num_neighbors, alive?)
    expect_true(conway_rules(3, FALSE))
    expect_false(conway_rules(4, FALSE))
    expect_true(conway_rules(2, TRUE))
    ...
})
```

(This uses the `testthat` package for R, available from CRAN.)

A *test suite* is a collection of related tests in a common context. A test suite an make provisions to prepare the environment for each test and clean up after (load a big dataset, connect to a database…). A test suite might also make available "test doubles" (think stunt double) that mimic or partially implement other features in the code (e.g., database access) to enable the unit to be tested as independently as possible.

Test suites are run and the results reported, particularly failures, in a easy to parse and economical style. For example, Python's `unittest` can report like this:

```
$ python test/trees_test.py -v

test_crime_counts (__main__.DataTreeTest)
Ensure Ks are consistent with num_points. ... ok
test_indices_sorted (__main__.DataTreeTest)
Ensure all node indices are sorted in increasing order. ... ok
test_no_bbox_overlap (__main__.DataTreeTest)
Check that child bounding boxes do not overlap. ... ok
test_node_counts (__main__.DataTreeTest)
Ensure that each node's point count is accurate. ... ok
test_oversized_leaf (__main__.DataTreeTest)
Don't recurse infinitely on duplicate points. ... ok
test_split_parity (__main__.DataTreeTest)
Check that each tree level has the right split axis. ... ok
test_trange_contained (__main__.DataTreeTest)
Check that child tranges are contained in parent tranges. ... ok
```

```
test_no_bbox_overlap (__main__.QueryTreeTest)
Check that child bounding boxes do not overlap. ... ok
test_node_counts (__main__.QueryTreeTest)
Ensure that each node's point count is accurate. ... ok
test_oversized_leaf (__main__.QueryTreeTest)
Don't recurse infinitely on duplicate points. ... ok
test_split_parity (__main__.QueryTreeTest)
Check that each tree level has the right split axis. ... ok
test_trange_contained (__main__.QueryTreeTest)
Check that child tranges are contained in parent tranges. ... ok

----------------------------------------------------------------------
Ran 12 tests in 23.932s

OK
```

Test packages often make it easy to run an entire suite all at once. R's `testthat` package provides a simple function to run all tests in a directory:

```
test_dir("tests/")
```

This prints out a summary of results. You could automated this with a simple script, say `test-all.R`, that runs all of your tests at once.

Wait, What Do I Test?

So what should be included in tests?

A core principle: tests should be passed by a correct function, but not by an incorrect function.

That seems obvious, but it's deeper than it looks. You want your tests to stress your functions: try them in multiple ways to make them break. Test corner cases. After all, we could write this test:

```
test_that("Addition is commutative", {
    expect_equal(add(1, 3), add(3, 1))
})
```

which is a perfectly good test, but is also passed by these two functions:

```r
add <- function(a, b) {
    return(4)
}
```

```r
add <- function(a, b) {
    return(a * b)
}
```

So this test is easily passed by incorrect functions! We need to test more thoroughly. Always try to test:

- several specific inputs for which you know the correct answer
- "edge" cases, like a list of size zero or size eleventy billion
- special cases that the function must handle, but which you might forget about months from now
- error cases that should throw an error instead of returning an invalid answer
- previous bugs you've fixed, so those bugs never return.

Try to cover all branches of your function: that is, if your function has several different things it can do depending on the inputs, test inputs for each of these different things.

Test Exercises

Scenario 1. Find the maximum sum of a subsequence

Function name: `max_sub_sum(arr)`

Write a function that takes as input a vector of n numbers and returns the maximum sum found in any *contiguous* subvector of the input. (We take the sum of an *empty* subvector to be zero.)

For example, in the vector [1, -4, 4, 2, -2, 5], the maximum sum is 9, for the subvector [4, 2, -2, 5].

There's a clever algorithm for doing this fast, with an interesting history related to our department. But that's not important right now. How do we test it?

(If you want to implement it – there's a homework problem for that!)

Test ideas?

1. All negative numbers should be empty subvector, zero sum

2. Only one positive value
3. Max sum at the end of the vector
4. Empty vector
5. Throws an error when `arr` is non numeric
6. Decimal/floating points
7. Vector of both positive and negative numbers, check answer
8. Huge number or huge vector
9. Only positive numbers == just sum the vector

- Some assertions to test

```r
test_that("max_sub_sum works on select examples", {
    expect_equal(max_sub_sum(c(1, 2, 3, 4)), 10)

    expect_equal(max_sub_sum(c(-1, -2, -3, -4)), 0)

    expect_equal(max_sub_sum(c(-1, 1, -1, 1, -1, 1)), 1)

    expect_equal(max_sub_sum(c(31, -41, 59, 26, -53, 58, 97, -23, 84)), 187)
})

test_that("max_sub_sum on the empty vector", {
    expect_equal(max_sub_sum(c()), 0)
})
```

- Can we do more?

  That wasn't very hard to test. But are there more thorough tests to run? Can you imagine randomly generating data and testing *properties* of the results?

  An example to start you off: the `max_sub_sum` of any vector of nonnegative numbers must just be equal to the sum of the whole list.

```racket
;; Max sub sum of nonnegative list must be sum of the whole list
(define max-sub-sum-positive
  (property ([l (arbitrary-list arbitrary-real)])
            (let ([nonneg (map abs l)])
              (= (max-sub-sum nonneg) (apply + nonneg)))))

(quickcheck max-sub-sum-positive)

;; Max sub sum of negative list must be zero
(define max-sub-sum-negative
  (property ([l (arbitrary-list arbitrary-natural)])
            (= (max-sub-sum (map - l)) 0)))

(quickcheck max-sub-sum-negative)

;; Max sub sum of list must be equal to max sub sum of its reverse
(define max-sub-sum-reverse
  (property ([l (arbitrary-list arbitrary-real)])
            (approx=? (max-sub-sum l)
                      (max-sub-sum (reverse l)))))

(quickcheck max-sub-sum-reverse)
```

This is an example of *generative testing*, which we'll return to in a moment.

Scenario 2. Create a half-space function for a given vector

Function name: `half_space_of(point)`

Given a vector in Euclidean space, return a boolean **function** that tests whether a *new* point is in the positive half space of the original vector. (The vector defines a perpendicular plane through the origin which splits the space in two: the positive half space and the negative half space.)

Test ideas?

1. Original point should return true
2. Check points on the perpendicular plane – or the origin
3. Multiplying the *test* point by a positive number should still give the same answer – and the opposite answer for a negative number
4. Rotate the test vector by a small amount
5. Check that it throws an error if original point is 0
6. Try 2D space as well as 50D

7. Inner product of test point and original point

- The tests

   1. Handle error when point is the zero vector
   2. For any vector, it should be in its own half space, and so should all positive multiples
   3. Negative multiple of the vector should not be in its half space
   4. Just pick a bunch of arbitrary points (opportunity)

   Write the corresponding tests:

```r
test_that("half_space_of handles invalid input", {
    expect_error(half_space_of(c(0, 0, 0)), "zero vector doesn't define a half space")
})

test_that("multiple of point is in its own half space", {
    case_count <- 1000

    for ( test in 1:case_count ) {
        point <- c(rnorm(1), rnorm(1), rnorm(1))
        half_space <- half_space_of(point)

        multiple <- rgamma(1, 1, 1/10)

        expect_true(half_space(multiple * point))

        expect_false(half_space(-multiple * point))
    }
})

test_that("half_space_of on random inputs", {
    case_count <- 1000
        spread <- 10
    half_space <- half_space_of(c(0, 0, 1))

    # vectors with positive z coordinate
    for ( test in 1:case_count ) {
        case <- c(rnorm(1,0,spread), rnorm(1,0,spread), rgamma(1,1,1/spread))
        expect_true(half_space(case))
    }

    # vectors with negative z coordinate
    for ( test in 1:case_count ) {
        case <- c(rnorm(1,0,spread), rnorm(1,0,spread), -rgamma(1,1,1/spread))
        expect_false(half_space(case))
    }

    # vectors with 0 z coordinate
    for ( test in 1:case_count ) {
        case <- c(rnorm(1,0,spread), rnorm(1,0,spread), 0.0)
        expect_false(half_space(case))
    }
}
```

Now write code to make those tests pass:

```r
library(assertthat)
library(testthat)

#' Create a function that tests if a point belongs to a half-space
#'
#' The halfspace is the orthogonal complement of the vector \code{direction}.
#'
#' @param direction  a numeric vector of dimension n greater than 1
#' @return a boolean function testing if an n-vector belongs to
#'          the \emph{positive} half-space determined by \code{direction}.
#'
half_space_of <- function(direction) {
    assert_that(length(direction) > 1)
    assert_that(is.numeric(direction))
    assert_that(crossprod(direction) > 0)
    force(direction)

    test_fn <- function(query_point) {
        return ( query_point %*% direction > 0.0 )
    }
    return ( test_fn )
}
```

Should we be limited by specified points? What if our imagined scenarios miss something important?

Side question: why does `half_space_of` return a **function**?

Scenario 3. What's the closest pair of points?

Function name: `closest_pair(points)`

Given a set of points in the 2D plane, find the pair of points which are the closest together, out of all possible pairs.

Later we will learn a good algorithm, using dynamic programming, to solve this without comparing all possible pairs. For now, let's think of tests.

Question: are there *properties* of `closest_pair` that can be tested? Could we generate random data and test that these properties hold?

Ideas?

1. After finding closest pair in dataset, pick out some other points – they shouldn't be closer
2. Scale every point by the same value, closest pair should be the same
3. Translate all points by the same value, closest pair should be the same
4. Rotate all points by same amount, closest pair should be the same
5. If function works in different dimensions, try 1D data
6. Find the closest pair, then add a point closer to one of them, and run again
7. What if more than one pair has the same, closest, distance?
8. If a point has a duplicate, those two should be the closest pair
9. What should it return if all points are identical?
10. Permute the points: ordering of points should not change result

- Some assertions to test
    1. Try a few simple examples
    2. If we remove a point at random from `points`, the minimum distance can only increase
    3. There must not be a point between the two closest, or in a circle of that radius centered on either point
    4. Points returned must be in the set provided

5. If the set has duplicates, we must return the pair (or, the set should be unique, so throw an error)
6. Any other point must be farther away
7. If we add a point halfway between the pair, it must be in the new closest pair

```racket
(require rackunit)

(check-equal? (closest-pair (list '(0 0) '(1 1) '(-1 -1) '(0 0)))
              (min-pair 0 '(0 0) '(0 0)))

(check-equal? (closest-pair (list '(0 0) '(0 17) '(0 235) '(0 1)))
              (min-pair 1 '(0 0) '(0 1)))

(check-equal? (closest-pair (list '(0 0) '(0 17) '(0 1)))
              (min-pair 1 '(0 0) '(0 1)))

(check-equal? (closest-pair (list '(0 0) '(0 17) '(0 235) '(0 1) '(21 32)))
              (min-pair 1 '(0 0) '(0 1)))

(check-exn exn:fail? (lambda () (closest-pair '())))
```

- Generative tests

  A framework called QuickCheck, originally invented for Haskell, makes it simple to define properties of functions which must hold with *randomly generated inputs*. In Racket, for example:

```racket
(require quickcheck rackunit/quickcheck)

(define drop-points-geq
  (property
   ([points (arbitrary-list (arbitrary-tuple arbitrary-real arbitrary-real))])

   (if (> (length points) 2)
       (<= (min-pair-dist (closest-pair points))
           (min-pair-dist (closest-pair (rest points))))
       #t)))

(check-property drop-points-geq)
```

This idea – defining *properties* that hold for arbitrary data, instead of defining specific test cases – is called *generative testing.* It's particularly useful for problems like this one, where you can see clear mathematical properties that must hold for the solution.

It takes some careful thinking to decide on a set of properties which must hold for the correct function but which do not hold for any other incorrect function.

Tests in Practice

Tests are commonly kept in separate source files from the rest of your code. In a long-running project, you may have a `test/` folder containing test code for each piece of your project, plus any data files or other bits needed for the tests.

For example, in my current project, I have:

```
$ ls test/

background_test.py  covariance_test.py  kde_test.py         spatial_test.py
bayes_test.nb       crimedata_test.py   likelihood_test.nb  test-data.h5
bayes_test.py       em_test.nb          likelihood_test.py  trees_test.py
bbox_test.py        em_test.py          __pycache__/
bg-test.h5          emtools_test.py     rect_bg_1.png
cd-dump.h5          geometry_test.py    regress-fit.h5
```

and I can run all the tests with a single command. You should choose a similar structure: separate files with tests, plus a script or shell command that runs all the tests (e.g. using `testthat`'s `test_dir` function or Python's `unittest` module).

The goal is to make tests easy to run, so you run them *often.* Run your tests before you commit new code, after you make any interesting changes, and whenever you fix bugs (remember to add a test for the bug!). You should always be confident that your code is well tested.

Some homework assignments will ask for a "driver script" which runs all your tests. This can just be a separate source file with all your tests, which run when you run the file. (Run your tests before you submit your homework!)

Test-Driven Development

Unit testing can be the basis of a software design approach.

Test Driven Development (TDD) uses a short development cycle for each new feature or component:

1. Write tests that specify the component's desired behavior. The tests will initially fail as the component *does not yet exist.*

2. Create the minimal implementation that passes the test.
3. Refactor the code to meet design standards, running the tests with each change to ensure correctness.

Why work this way?

- Writing the tests may help you realize what arguments the function must take, what other data it needs, and what kinds of errors it needs to handle.
- The tests define a specific plan for what the function must do.
- You will catch bugs at the beginning instead of at the end (or never).
- Testing is part of design, instead of a lame afterthought you dread doing.

Try it. We will expect to see tests with your homework anyway, so you might as well write the tests first!

More Types of Testing

- Regression Testing
  - seeks to uncover new bugs introduced after changes
  - often done by re-running old tests and comparing with passing results
  - good for testing entire analyses to be sure nothing changes unexpectedly
- Integration Testing
  - test how system components fit together and interact
- Acceptance Testing
  - does the system meet its overall specifications?
  - blackbox system tests
- Top-down Testing
  - specifies test requirements in terms of constants and metaconstants
  - allows you to write functions in terms of other functions not yet written

  Top-down test, where `account-number` is implemented in terms of `digit-parcels` and `digit`, which are not yet implemented:

```
(unfinished digit-parcels digit)

(fact "an account number is constructed from character parcels"
  (account-number ..parcel..) => "01"
  (provided
    (digit-parcels ..parcel..) => [..0-parcel.. ..1-parcel..]
    (digit ..0-parcel..) => 0
    (digit ..1-parcel..) => 1))
```

- Others as well (functional, usability, . . .)

Test Frameworks

For whatever language you use, there is likely already a unit testing framework which makes testing easy to do. No excuses!

R

See the Testing chapter from Hadley Wickham's *Advanced R* book for examples using `testthat`.

- testthat (recommended, friendly and easy)
- RUnit (standard xUnit style)
- quickcheck (generative, abandoned?)
- checkr (newer generative framework)

Python

- unittest (builtin, nice)
- py.test (more ergonomic)

Java

- JUnit (the original)

Clojure

- clojure.test (built in)
- midje (excellent!)
- test.check (generative)

JavaScript

- Karma (cf. Protractor)
- Buster.js
- QUnit
- Mocha
- Jasmine (client side, BDD, no DOM)

C++

- Boost.Test
- CppUnit
- Catch
- lest (>= C++11)

Haskell

- HUnit
- QuickCheck (generative)

Behavior-Driven Development Frameworks

- Cucumber