Testing 1, 2, 3. . .

Christopher R. Genovese

Department of Statistics & Data Science

Tue 09 Sep 2025
Session #5

Plan

Motivation

Plan

Plan

Plan

Plan

Announcements

- Office Hours: me Tuesday 4pm, Thea Thursday 2pm, and by appointment. I encourage you to ask questions.
- **Reading**:
  - Finish Previous Readings
- **Homework**:
  - **swag** assignment due today. Available on Canvas and github problem bank.
  - **migit** assignment Tasks #1–#6 due Tue 16 Sep. Available on github problem bank.
  - **migit** assignment Tasks #1–#10 due Tue 23 Sep. Available on github problem bank.

Goals for Today

Last time, we studied the idea of objects and Object-Oriented Programming (OOP), and introduced migit.

Today's goals:

1. Discuss how to using regular testing to enhance the correctness of our code, considering different approaches to testing and various frameworks in each language
2. Offer practical guidance on writing and using tests
3. Practice generating tests
4. Revisit the Fold Pattern

Plan

Complexity can have consequences!

Bugs will happen:

- 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),
- a 15-year-old fMRI analysis software bug that inflated significance levels (2015).

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.

Plan

Quick Testing Terminology

- **test** – a function that runs other code in a library or application codebase and checks its results for correctness
- **test suite** – a collection of tests on a related theme
- **unit** – (loosely) a piece of code with a small, well-defined purpose and scope.
- **assertion** – a claim about the state of a program or the result of a test
- **unit test** – an automated test of a unit, usually checking the result with several different inputs or configurations
- **property** – a logical invariant that a piece of code should satisfy
- **property test** – a test of a property that generates consistent inputs and upon failure, attempts to produce a near-minimal example that causes failure
- **fixture** – a context or setting that needs to be setup before tests are run (and usually torn down afterwards); these often create fakes/stubs/mocks that simulate entities our code should interact with in reality

Recommended Libraries

- Python
  - unit testing+: pytest
  - property testing: Hypothesis

- R
  - unit testing: testthat
  - property testing: hedgehog or quickcheck

- Language Agnostic Tools (usually for other kinds of testing)
  Cucumber, Robot Framework, and others.

Test Structure

In these (language-specific) frameworks, ***tests*** are functions flagged by a special name, e.g., `test_`.

Each test contains one or more ***assertions***. These can take different forms. For example, the `testthat` assertions are functions like `expect_equal` or `expect_error`.

The assertions take the result to be tested and (usually) the correct result and compare them in a suitable way.

Each test can have many assertions; the failure of any one causes the test to fail.

In practice, you can select which tests are run, though we usually just run them all.

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 that 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.

Unit Testing: Python

```python
from __future__ import annotations
import pytest
# ...

def test_kinds_factories():
    "Builtin kind factories"
    a = symbol('a')

    assert constant(1).values == {1}
    assert constant((2,)).values == {2}
    assert constant((2, 3)).values == {vec_tuple(2, 3)}

    assert either(0, 1).values == {0, 1}
    assert weights_of(either(0, 1, 2).weights) == pytest.approx([as_quant('2/3'),
                                                                 as_quant('1/3')])
    assert lmap(str, values_of(either(a, 2 * a, 2).weights)) == ['<a>', '<2 a>']

    # ...

    with pytest.raises(KindError):
        k0 >> me1
```

Unit Testing: R

```r
library(testthat)

source("foobar.R")

test_that("foo values are correct", {
    expect_equal(foo(4), 8)
    expect_equal(foo(2.2), 1.9)
})

test_that("bar has correct limits", {
    expect_lt(bar(4, c(1, 90), option = TRUE), 8)
})

test_that("bar throws an error on bad inputs", {
    expect_error(bar(-4, c(1, 10))) # test passes if bar calls stop()
                                    # or throws an error here
})
```

Property Testing (aka Generative Testing)

A powerful technique for automatically testing logical invariants without having to create many small examples by hand. A **property** is a claim made about a specified set of quantities; we specify the set through **generators** that produce values of particular types and shapes.

Generators may be combined to form new generators that are more specific to our needs.

Property Testing (aka Generative Testing)

```
(def sort-idempotent-prop
  (for-all [v (gen/vector gen/int)]  ;for integer vectors v
    (= (sort v) (sort (sort v)))))   ;  sorted & twice sorted v are equal

(quick-check 100 sort-idempotent-prop)
;; => {:result true,
;;     :pass? true,
;;     :num-tests 100,
;;     :time-elapsed-ms 28,
;;     :seed 1528580707376}
```

Property Testing (aka Generative Testing)

```
from hypothesis import given
from hypothesis.strategies import text

@given(text())
def test_decode_inverts_encode(s):
    assert decode(encode(s)) == s
```

Property Testing (aka Generative Testing)

One of the most useful features of Property/Generative testing is that on failures, the generators will /search/ for a (roughly) minimal/simple example that fails, to make it easier to identify the problem.

```
(def sorted-first-less-than-last-prop
  (for-all [v (gen/not-empty (gen/vector gen/int))]
    (let [s (sort v)]
      (< (first s) (last s)))))

(quick-check 100 sorted-first-less-than-last-prop)
;; => {:num-tests 5,  :seed 1528580863556,
;;       :fail [[-3]],
;;       :failed-after-ms 1,
;;       :result false,
;;       :result-data nil,
;;       :failing-size 4,
;;       :pass? false,
;;       :shrunk
;;       {:total-nodes-visited 5, :depth 2,
;;        :pass? false, :result false, :result-data nil,
;;        :time-shrinking-ms 1,
;;        :smallest [[0]]}}
```

Property Testing: Generators/Strategies

```python
from hypothesis.strategies import characters, text
from hypothesis import given
from unicodedata import category

names = text(
    characters(max_codepoint=1000, blacklist_categories=('Cc', 'Cs')),
    min_size=1).map(lambda s: s.strip()).filter(lambda s: len(s) > 0)

@given(names)
def test_names_match_our_requirements(name):
    assert len(name) > 0
    assert name == name.strip()
    for c in name:
        assert 1 <= ord(c) <= 1000
        assert category(c) not in ('Cc', 'Cs')
```

Property Testing: Generators/Strategies

```python
from hypothesis import given
from hypothesis.extra.datetime import datetimes

project_date = datetimes(timezones=('UTC',), min_year=2000, max_year=2100)


@given(project_date)
def test_dates_are_in_the_right_range(date):
    assert 2000 <= date.year <= 2100
    assert date.tzinfo._tzname == 'UTC'
```

Property Testing: Generators/Strategies

```r
library(testthat)
library(hedgehog)  # or library(quickcheck)

test_that( "Reverse of reverse is identity",
  forall( gen.c( gen.element(1:100) ), function(xs) expect_equal(rev(rev(xs)), xs))
)

test_that( "a is less than b + 1",
    forall(list(a = gen.element(1:100), b = gen.unif(1,100, shrink.median = F))
  , function(a, b) expect_lt( a, b + 1 ))
```

Property Testing: Generators/Strategies

```r
library(testthat)
library(hedgehog)  # or library(quickcheck)

test_that( "Reversed of concatenation is flipped concatenation of reversed",
  forall( list( as = gen.c( gen.element(1:100) )
              , bs = gen.c( gen.element(1:100) )
              ),
          function(as,bs) expect_equal ( rev(c(as, bs)), c(rev(bs), rev(as)))
  )
)

gen.df.of <- function ( n )
  gen.with (
    list( as = gen.c(of = n, gen.element(1:10) )
        , bs = gen.c(of = n, gen.element(10:20) )
        ),
    as.data.frame
  )

test_that( "Number of rows is 5",
  forall( gen.df.of(5), function(df) expect_equal(nrow(df), 5))
)
```

What to Test

Tests should pass for correct functions and not for incorrect functions. Obvious? Consider:

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

# Fine test but passes for both of these:
add <- function(a, b) {
    return(4)
}
add <- function(a, b) {
    return(a * b)
}
```

We need to test more thoroughly.

What to Test

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; and
- 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.

Whenever you encounter a bug, record it as a test!

Tips and Practices

- 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.
- All tests can now be run with a single command (e.g. using `testthat`'s `test_dir` function or Python's `pytest` module)
- Run tests **often**. It is common to set up a *hook* that runs your tests before each commit, and perhaps rejects the commit if the tests fail.
- Every time you check your code, such as at the repl or with an example run, *make a test out of it*. Every time you encounter a bug or other failure, make a test out of it. Every example you put in your documentation can produce a test.
- There is a wide variety of built-in assertions in common testing libraries, including for instance asserting that a piece of code throws an error.
  There may also be third party libraries that add additional assertions and tools; these can be included as a "dev dependency" without affecting your users.
  And you can always write your own if you need something special repeatedly.
- It is valuable when possible to write some tests *before* you implement a function. This can help you understand (and even document) what the function needs to do, including edge cases.
- Make tests *replicable*: If a test involves random data, what do you do when the test fails? You need some way to know what random values it used so you can figure out why the test fails.

Plan

Rapid-Fire Testing 1

Scenario. 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].

As we've seen, 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 repository problem for that! Try the `max-sub-sum` exercise.)

Test ideas? Consider properties!

Rapid-Fire Testing 2

Scenario. 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.) An example:

```
foo <- half_space_of(c(2, 2))
foo(c(1, 1)) == TRUE
```

Test ideas?

Plan

Activity: Writing Tests for Migit Task #2
`GitRepository` Object

In the file system, a repository is represented by the presence of a `.git` directory at the project root that contains particular files. You can see what these are by using `git init` in a temporary directory and looking at the `.git` it creates.

For the moment, all we need from the `GitRepository` object are methods for finding the project root from an arbitrary path and for testing if a path is inside a (mi)git repository. We will also need methods for checking the validity of a repository (through the presence of specific files in the `.git` directory) and for creating a repository if requested. It should signal an error at an attempt to create a repository in an existing project or to read a repository with incomplete data.

To this end, create class (formal or informal) for a `GitRepository`. What data do you need? What methods?

Although you have not yet implemented this class, write some appropriate tests.

Plan

The “Fold” Pattern

Here is a common computational pattern expressed in TL2:

```
1       accumulator = starting_value
2       for item in items
3           accumulator = update(accumulator, item)
```

This is called a **fold**.

We successively update an “accumulator” from a starting value for each item in a collection, returning the final value of the accumulator.

- Line 1 initializes the accumulator.
- Line 2 iterates over each item in the collection.
- Lines 3 updates the accumulator with the given item.
  The function `update` takes an accumulator and an item and returns a *new accumulator*. It is often called the folding function, reducing function, or update rule.

This yields the final value of the `accumulator`, which can be an arbitrary data structure.

(Renaming `accumulator` to `state` reveals another view of this pattern.)

The "Fold" Pattern

Here is a common computational pattern expressed in TL2:

```
    accumulator = starting_value
    for item in items
        accumulator = update(accumulator, item)
```

This is called a **fold**.

We successively update an "accumulator" from a starting value for each item in a collection, returning the final value of the accumulator.

A fold is a general sequence *consumer*.

Here is a fold expressed as a *function* in TL1:

```
foldLeft : (a -> i -> a) -> a -> List i -> a
foldLeft _ start [] = start
foldLeft update start (x :: xs) = foldLeft update (update start x) xs
```

Ensure that you see the equivalence of these two formulations.

Some Useful Fold Variants

The "Scan" pattern: A fold that also collects all intermediate results is called a **scan**.

```
    accumulator = starting_value
    collected = [accumulator]
    for item in items
        accumulator = update(accumulator, item)
        collected.append(accumulator)
```

This yields both the final values of `accumulator` and `collected`.

A scan is a sequence *transformer* that maintains the lengths of all partial sequences.

We can write the function `scanLeft` to implement this version of the Scan pattern. (`scanRight` does a scan for a fold from the right.)

```
scanLeft : (a -> i -> a) -> a -> List i -> List a
```

Example: Cumulative sums are computed with `scanLeft (+) 0 items`.

Some Useful Fold Variants

Left vs. Right Folds. The previous fold and scan operate from the left, taking the items in order and updating the starting value of the accumulator successively.

In contrast, a right fold starts from the right end, which has some advantages in certain circumstances.

```
foldRight : (i -> a -> a) -> a -> List i -> i
foldRight _ start [] = start
foldRight update start (x :: xs) = update x (foldr update star
```

Note that the update functions given to `foldLeft` and `foldRight` have their argument orders swapped, with the accumulator parameter on the left or right, respectively.

Similarly, we can define an analogous `scanRight`.

Some Useful Fold Variants

Parallel Folds. Depending on the nature of the updates, folds need not be processed in either order.

In many cases, it is possible to do them in parallel, which allows both for improved efficiency and distributed computation over large collections.

This is a key part of the MapReduce idea.

The “Unfold” Pattern

Dual to a fold is an **unfold**, which generates a sequence from a `seed` and an unfolding function `ufn`.

Here is a simple TL2 implementation of the idea:

```
    collected = []
    while True:
        match ufn(seed):
            case None:
                break

            case Some((new_seed, value)):
                seed = new_seed
                collected.append(value)
```

As long as the unfolding function returns a non-trivial value, we update the seed and collect the value that was produced. This is a *left* unfold; if we used `collected.prepend` instead, we would get a right unfold.

The “Unfold” Pattern

Dual to a fold is an **unfold**, which generates a sequence from a `seed` and an unfolding function `ufn`.

Here’s a simple TL1 implementation using Lists:

```
unfoldLeft : (s -> Maybe (s, v)) -> s -> List v
unfoldLeft ufn seed = loop (ufn seed) |> reverse
  where
    loop : Maybe (s, v) -> List v
    loop None = Sequence.Empty
    loop (Some (seed', value)) = value :: loop (ufn seed')
```

We could define an analogous `unfoldRight` as well by excluding the `reverse` step.

The “Unfold” Pattern

Dual to a fold is an **unfold**, which generates a sequence from a seed and an unfolding function ufn.

Here’s another implementation with a more flexible sequence type:

```
unfoldLeft : (s -> Maybe (s, v)) -> s -> Sequence v
unfoldLeft ufn seed = loop empty (ufn seed)
  where
    loop : Sequence v -> Maybe (s, v) -> Sequence v
    loop collected None = collected
    loop collected (Some (seed', value)) =
        loop (append collected value) (ufn seed')
```

The “Fold” Pattern: Brief Exercises

1. Define a fold that computes the frequencies of items in the sequence, assuming the items are comparable. (R users can take the items in the sequence to be strings for simplicity.)
2. Define a fold that computes the five largest elements in a numeric sequence.
3. Define a fold that returns an element of a given index in the sequence.
4. Define a scan that computes an exponentially-weighted moving average of the values in a numeric sequence.
5. Define an unfold that computes the sequence of approximations by Newton's method to the root of a smooth univariate function, stopping when the change (absolutely or relatively, your choice) is sufficiently small.
6. Define an unfold that builds a `RoseTree`:

   ```
   data RoseTree : Type -> Type where
     Node : a -> List (RoseTree a) -> RoseTree a
   ```

   or for short

   ```
   data RoseTree a = Node a (List (RoseTree a))
   ```

   This unfold can have signature

   ```
   unfold : (b -> (a, List b)) -> b -> RoseTree a
   ```

THE END