Testing Best Practices in Ruby

(fka Ruby Testing Best Practices)

- Why test?
- Best practices
- Tips and tricks
- Appendix

Why test?

- Design
  - If the test is hard to write, the design is probably wrong
- Ensure branching logic behaves appropriately
- Exercise boundary conditions
- Refactoring
- Onboard new team members
- Documentation never out of date
- Like, 10,000 other reasons, probably

Why test first?

- Decreases coupling
- Rhythm
- Tighter feedback loop
- "What do I really want to test?"
- Easier to engage a third-party when questions arise
- Identify complexity beforehand by switching into "test" mode

Kyle's List of Ruby Testing Best Practices Culled from Two Years' Worth of Reading, Writing, and Refactoring Tests Running the Gamut from Awful to Inspiring

Name Tests Descriptively

BAD

```
test "add_friend" do
  # What do I write here? I have no idea where to start.
end
```

GOOD

```
test "add_friend raises an error when given a private user" do
  user = User.new
  private_user = User.new(:private => true)

  assert_raises{ user.add_friend(private_user) }
end
```

One Assertion per Test

BAD

```
test "eat_sandwich works entirely as expected" do
    human = Human.new
    sandwich = Sandwich.new

    human.eat_sandwich sandwich
    assert_true human.full?
    assert_true sandwich.eaten?
end
```

One Assertion per Test

GOOD

```
test "eat_sandwich marks the given sandwich as eaten" do
    sandwich = Sandwich.new
    Human.new.eat_sandwich sandwich

    assert_true sandwich.eaten?
end

test "eat_sandwich fills the human in question" do
    human = Human.new
    human.eat_sandwich Sandwich.new

    assert_true human.full?
end
```

Declare Your Assumptions

BAD

```
test "popularity is 100 if person loves Lady Gaga" do
    assert_equal 100, Person.new.popularity
end
```

GOOD

```
test "defaults to loving Lady Gaga " do
    assert_equal ['Lady Gaga'], Person.new.loves
end

test "popularity is 100 if person loves Lady Gaga" do
    assert_equal 100,
        Person.new(:loves => ['Lady Gaga']).popularity
end
```

Don't Loop

BAD

```
test "every possible combination of invalid passwords fails" do
    ["1", "abcdef", "weinergateisnotachriswingatejoke"].each do |p|
        assert_true User.new(:password => p).invalid_password?
    end
end
```

Don't Loop

GOOD

```
test "is invalid if shorter than 6 characters" do
    assert_true User.new(:password => "1").invalid_password?
end

test "is invalid if exactly 6 characters long" do
    assert_true User.new(:password => "abcdef").invalid_password?
end

test "is invalid if explicitly restricted" do
    restricted = "weinergateisnotachriswingatejoke"
    RestrictedPassword.create! restricted
    assert_true User.new(:password => restricted).invalid_password?
end
```

Use Factories to Simplify

BAD

```
test "peon is set to gather_lumber when assigned to a tree" do
    # peon validates all of these fields
    # but we only care about the last one
    peon = Peon.create! :position => [0,0,0],
        :sounds => ["Zug zug", "We need lumber"],
        :skin => "green.png",
        :damage => 10,
        :armor => 5,
        :hilarity => 17.2,
        :can_harvest => [Tree, Mine]

    peon.assign_to Tree.new

    assert_equal "gather_lumber", peon.state
end
```

Use Factories to Simplify

GOOD

```
test "peon is set to gather_lumber when assigned to a tree" do
    # we don't actually care about most of the validated fields,
    # we just want a peon that can harvest trees
    peon = Factory(:peon, :can_harvest => [Tree])
    peon.assign_to Tree.new

    assert_equal "gather_lumber", peon.state
end
```

Don't over-define Cucumber steps

BAD
Given that I am a level 15 Guardian of the Wastes and I have a Rock-it Launcher
When I see a radscorp
Then I kill it in one shot

GOOD
Given that I am a Fallout player
And I am level 15
And I am a Guardian of the Wastes
And I have a Rock-it Launcher
When I see a radscorp
Then I kill it in one shot

Avoid Setup/Teardown Methods

- You lose isolation/encapsulation
- More import in tests to be explicit than DRY
- ...But they're okay in some situations
  - Refinery CMS needs 1 User object before anything else
  - Module testing (coming up)

Fixtures are evil

- Introduce a lot of dependencies
- People use the same fixture for different purposes
- Changing a fixture results in hundreds of broken tests
  - Fixtures are okay for testing bulk imports, I guess.

Use factories instead!

Don't write complex rake tasks

They're impossible to test.

Instead, create a facilitator object and have your rake task delegate to that.

```
task :prepare_alan_rickman_for_his_party do
    PartyPreparer::AlanRickman.remember_turtle_joke!
end
```

Avoid Rails callback hooks

(i.e. before_save, before_create, etc)

You'll never be able to create an object again for testing without stubbing out every one of these.

BAD

```
class Baby
    before_create :parental_sex
    before_save :baptize
end
```

GOOD

```
Baby.create_through_intercourse
Baby.save_with_baptism
```

Keep your test suite fast

One of the purposes of testing is to shorten feedback cycle time.

If your tests start getting out of control, you've already moved on to a new feature by the time you get feedback on the first.

How to fix?

- Profile
- Distribute
- Upgrade

Tricks - Testing Modules Explicitly

```
module TigerBlood
    def hashtag; '#winning'; end
    def dna_quality; DNA::Adonis; end
end

class TigerBloodTest
    def setup
        @helper = Class.new{ include TigerBlood }.new
    end

    test "hashtag is '#winning'" do
        assert_equal '#winning', @helper.hashtag
    end

    test "dna_quality is Adonis" do
        assert_equal DNA::Adonis, @helper.dna_quality
    end
end
```

Tricks - Extend Your Test Suite

```
def assert_true(expression)
  assert_equal true, expression
end
```

```
# terrible code that needs refactoring
def when_constant(constant_string, new_value)
    class_or_module = nil
    constant_name = nil

    class_or_module, constant_name = if constant_string =~ /::/
        values = constant_string.split("::")
        [values[0..-2].join.constantize, values[-1]]
    else
        [Object, constant_string]
    end

    old_value = class_or_module.const_get constant_name
    class_or_module.send(:remove_const, constant_name)
    class_or_module.const_set constant_name, new_value
    yield
    class_or_module.send(:remove_const, constant_name)
    class_or_module.const_set constant_name, new_value
end
```

Higher-Level Ideas

- Write a test for every (sufficiently complex) bug
- Test the most important things first
- Tests are not a replacement for good design

tl;dr

- Be explicit
- Be concise
- Don't rely on coincidences
- Make everything testable