RSpec

GLOSSARY v0.1

1. gem install................1
2. init RSpec.................1
3. run options................1
4. spec_helper...............1
describe........................2
context.....................2
it..............................2
expect.......................2
one-liner.............2
8. examples..................3
Zombies..................3
StringCalculator.......3
9 config.......................3
Color output............3
Cheat Sheet.................4
The Basics...............4
Let...........................4
Subject Under Test. .4
Hooks......................5
Skipping..................5
Pending....................5
Alternate Syntax......6
Defining Methods....6
Shared Examples.....6
Shared Context........7

1. gem install

```
$ gem install rspec
```

OR

in Gemfile:

```
source “https://rubygems.org”
gem “rspec
```

```
$ bundle install --path .bundle
```

2. init RSpec

```
$ rspec –init
```

3. run options

```
bundle exec rspec --format documentation
```

4. spec_helper

All the files can be required in
`# spec/spec_helper.rb`

describe

The `describe` block is always used at the top to put specs in a context. It can accept either a **class name**, in which case the class needs to exist, or **any string** you'd like.

class methods are prefixed with a **dot**	(`".add"`)
instance methods	with a **dash** (`"#add"`)

context

`context` is technically the same as **`describe`**, but is used in different places, to aid reading

it

to describe a specific ***example***
it is RSpec's way to say "**test case**"
Generally, every example should be descriptive

expect

to define expected outcomes

one-liner

```
it {is_expected.to respond_to(:duration)}
```

```
it "responds to '#duration'" do
 expect(subject).to respond_to(:duration)
end
```

it is convenient when you can avoid duplication between a matcher and the string that documents the test example.

8. examples

Zombies

```
# spec/lib/zombie_spec.rb
```

```
require "spec_helper"

describe "A Zombie" do
  it "is named Ash"
end
```

```
$ rspec spec/lib/zombie_spec.rb
```

```
zombie.propriety.should  ==
'uoo'/false/>5
zombie.alive.should be_false
zombie.height.should_not == 5
tweet.status.length.should be >= 140
```

StringCalculator

Path

```
# spec/string_calculator_spec.rb
```

File

```
describe StringCalculator do
  describe ".add" do
    context "empty string" do
      it "returns zero" do
       expect(StringCalculator.add
       ("")).to eql(0)
      end
    end
  end
end
```

Run

```
$ bundle exec rspec
```

9 config

Color output

add **`--color`** to a **`.rspec`** file to the project dir root

or, place the **`.rspec`** file in your home directory to apply the settings for all projects

NYANSPEC install the gem **`nyan-cat-formatter`** and put the options in the **`.rspec`** file **`--format NyanCatFormatter`**

Cheat Sheet

The Basics

An example that uses common RSpec features.

```
RSpec.describe 'an array of animals' do
 let(:animal_array){ [:cat,:dog,:mule] }

  it 'has three animals' do
    expect(animal_array.size).to eq(3)
  end

  context 'mutation' do
    after
{expect(animal_array.size).not_to eq(3)}

    it 'can have animals added' do
      animal_array << :cow
      expect(animal_array).to
            eq([:cat, :dog, :mule, :cow])
    end

    it 'can have animals removed' do
      animal_array.pop
      expect(animal_array).to
                         eq([:cat, :dog])
    end
  end
end
```

Let

Variables that are recreated for every test.

```
RSpec.describe 'Uses of `let`' do
  let(:random_number) { rand }
  let(:lazy_creation_time) { Time.now }
  let!(:eager_creation_time)
{ Time.now }

  it 'memoizes values' do
    first = random_number
    second = random_number
    expect(first).to be(second)
  end

  it 'creates the value lazily' do
    start_of_test = Time.now
    expect(lazy_creation_time).to be >
                            start_of_test
  end

  it 'creates the value eagerly using
                              `let!`' do
    start_of_test = Time.now
    expect(eager_creation_time).to be <
                            start_of_test
  end
end
```

Subject Under Test

Convenience methods for accessing the subject under test, for more concise tests. Like `let`, the subject object is recreated for every test.

```
RSpec.describe Array do
  it 'provides methods based on the
            `RSpec.describe` argument' do
    # described_class = Array
    expect(described_class).to be(Array)
    # subject = described_class.new
    expect(subject).to eq(Array.new)
    # is_expected = expect(subject)
    is_expected.to eq(Array.new)
  end

 context 'explicitly defined subject' do
    # subject can be manually defined
    subject { [1,2,3] }
    it 'is not empty' do
      is_expected.not_to be_empty
    end
  end

  context 'can be named' do
  #you can provide a name, just like `let`
    subject(:bananas) { [4,5,6] }
    it 'can be called by name' do
     expect(bananas.first).to eq(4)
    end
  end
end
```

Hooks

Run arbitrary code before or after each test or context.

```
RSpec.describe 'Hooks' do
  order = []

 before(:all) { order << 'before(:all)'}
 before       { order << 'before' }
 after        { order << 'after' }
 after(:all)  { order << 'after(:all)';
                            puts order }

  around do |test|
    order << 'around, pre'
    test.call
    order << 'around, post'
  end

  it 'runs first test' do
    order << 'first test'
  end

  it 'runs second test' do
    order << 'second test'
  end
end
```

Execution order for the tests above:

1. before(:all)
2. around, pre
3. before
4. first test
5. after
6. around, post
7. around, pre
8. before
9. second test
10. after
11. around, post
12. after(:all)

Skipping

Temporarily prevent tests from being run.

```
RSpec.describe 'Ways to skip tests' do
  it 'is skipped because it has no body'

  skip 'uses `skip` instead of `it`' do
  end

  xit 'uses `xit` instead of `it`' do
  end

  it 'has `skip` in the body' do
    skip
  end

  xcontext 'uses `xcontext` to skip a
                     group of tests' do
    it 'wont run' do; end
    it 'wont run either' do; end
  end
end
```

Pending

Temporarily ignore failing tests.

```
RSpec.describe 'Ways to mark failing
                   tests as "pending"' do
  pending 'has a failing expectation' do
    expect(1).to eq(2)
  end

  it 'has `pending` in the body' do
    pending('reason goes here')
    expect(1).to eq(2)
  end

  pending 'tells you if a pending test
                     has been fixed' do
 # Pending tests are supposed to fail.
 # This test passes, so RSpec will give
 # an error saying that this pending
 # test has been fixed.
    expect(2).to eq(2)
  end
end
```

Alternate Syntax

RSpec has aliases for `describe` and `it`.
They function exactly the same, but the different wording might be clearer reading the tests.
These are the default aliases:

- Groups: `context`, `example_group`, `describe`
- Examples: `it`, `example`, `specify`

```
RSpec.describe 'Alternate syntax' do
  example_group 'alernative to
                            "context"' do
    example 'alternative to "it"' do
      expect(2).to eq(2)
    end
  end

  describe 'alternative to "context"' do
    specify 'alternative to "it"' do
      expect(2).to eq(2)
    end
  end
end
```

Defining Methods

Define arbitrary methods for use within your tests.

```
RSpec.describe 'Defining methods' do
 def my_helper_method(name)
  "Hello #{name}, you just got helped!"
  end

  it 'uses my_helper_method' do
    message = my_helper_method('Susan')
    expect(message).to eq('Hello Susan,
                   you just got helped!')
  end

 context 'within a context group' do
  it 'can still use my_helper_method' do
      message = my_helper_method('Tom')
      expect(message).to eq('Hello Tom,
                   you just got helped!')
    end
  end
end
```

Shared Examples

A reusable set of tests that can be included into other tests.

```
RSpec.shared_examples 'acts like non-nil
                              array' do
  it 'has a size' do
    expect(subject.size).to be > 0
  end

  it 'has has non-nil values for each
                              index' do
   subject.size.times do |index|
    expect(subject[index]).not_to be_nil
   end
  end
end

RSpec.describe 'A real array' do
  include_examples 'acts like non-nil
                              array' do
    subject { ['zero', 'one', 'two'] }
  end
end

RSpec.describe 'A hash with integer
                               keys' do
  include_examples 'acts like non-nil
                              array' do
    subject { Hash[0 => 'zero', 1 =>
                    'one', 2 => 'two'] }
  end
end
```

Shared Context

A reusable context setup (hooks, methods, and lets) that can be included into other tests.

```
RSpec.shared_context 'test timing' do
  around do |test|
    start_time = Time.now
    test.call
    end_time = Time.now
    puts "Test ran in #{end_time -
                     start_time} seconds"
  end
end

RSpec.describe 'big array' do
  include_context 'test timing'

  it 'has lots of elements' do
    big_array = (1..1_000_000).to_a
    expect(big_array.size).to
eq(1_000_000)
  end
end
```