RUBY ON RAILS TUTORIAL

Table of Contents

Command Line Descriptions for beginners

Description	Command	Example
list contents	`ls`	`$ ls -l`
make directory	`mkdir <dirname>`	`$ mkdir environment`
change directory	`cd <dirname>`	`$ cd environment/`
cd one directory up		`$ cd ..`
cd to home directory		`$ cd ~` or just `$ cd`
cd to path incl. home dir		`$ cd ~/environment/`
move file (rename)	`mv <source> <target>`	`$ mv foo bar`
copy file	`cp <source> <target>`	`$ cp foo bar`
remove file	`rm <file>`	`$ rm foo`
remove empty directory	`rmdir <directory>`	`$ rmdir environment/`
remove nonempty directory	`rm -rf <directory>`	`$ rm -rf tmp/`
concatenate & display file contents	`cat <file>`	`$ cat ~/.ssh/id_rsa.pub`

Your First Rails App

Installing Rails

`gem install rails -v 5.1.6`

Use the most up-to-date version while installing rails

Creating Rails App

`rails new <App Name>`

This creates a list of directories that will work together to build the app itself

Default Rails Directory Structure

File/Directory	Purpose
`app/`	Core application (app) code, including models, views, controllers, and helpers
`app/assets`	Applications assets such as cascading style sheets (CSS), JavaScript files, and images
`bin/`	Binary executable files
`config/`	Application configuration
`db/`	Database files
`doc/`	Documentation for the application
`lib/`	Library modules
`lib/assets`	Library assets such as cascading style sheets (CSS), JavaScript files, and images
`log/`	Application log files
`public/`	Data accessible to the public (e.g., via web browsers), such as error pages
`bin/rails`	A program for generating code, opening console sessions, or starting a local server
`test/`	Application tests
`tmp/`	Temporary files
`vendor/`	Third-party code such as plugins and gems
`vendor/assets`	Third-party assets such as cascading style sheets (CSS), JavaScript files, and images
`README.md`	A brief description of the application
`Rakefile`	Utility tasks available via the `rake` command
`Gemfile`	Gem requirements for this app
`Gemfile.lock`	A list of gems used to ensure that all copies of the app use the same gem versions
`config.ru`	A configuration file for Rack middleware
`.gitignore`	Patterns for files that should be ignored by Git

Table 1.2: A summary of the default Rails directory structure.

Bundler

Bundler is used to install and include the gems needed by the app.

`cd <App Name>`

`bundle install`

This command is automatically called while creating a new Rails app but needs to be called again.

If you open the Gemfile within your app you will see the list of gems that are included:

- `gem 'sqlite3'`: if the version is not mentioned, it will include the latest one
- If you want a specific version of a gem: `gem 'uglifier', '>=1.3.0'`
- If you want a particular version group (anything from 4.0, but not 4.1 and onwards): `gem 'coffee-rails', '~> 4.0.0'`

When you run `bundle install` you might get a message saying you need to run `bundle update`. In that case, you need to run `bundle update`.

Run Rails Server

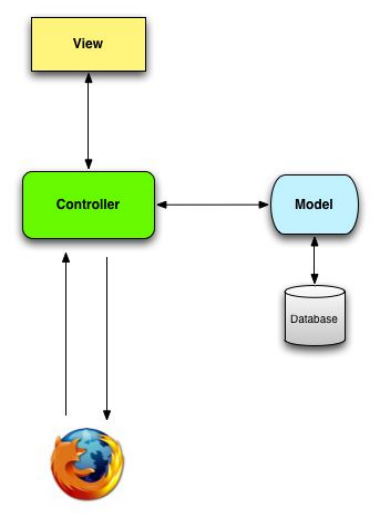
`cd <App Name>`

`rails server`

This command will start running the rails app. You can view your running app on your browser using the URL `http://localhost:3000` which will show the rails app that you created! Hooray!!

Rails Architecture: Model View Controller (MVC)

The `app/` directory consists of subdirectories `models`, `views`, and `controllers`. This creates a separation between the data in the application and the code used to display it.



Controller: when a browser sends a request, the web server passes it to a Rails *controller,* which is in charge of what to do next.

Model: The controller interacts with the model, which is a Ruby object that represents an element in the site (such as a user) and is in charge of communicating with the database.

View: After(for dynamic sites) or before (for static sites) invoking the model, the controller renders the view, which is a template that gets rendered to HTML and sends it back to the browser.

Exercise One: Hello World!

Change default Rails page using controller action

The methods in *controllers* are known as *actions*. Controller actions are used to render whatever will be displayed on the Rails page. When you initially create an app, the `application_controller.rb` is the only controller that will be present in the `app/controllers`.

To add 'hello world!' text to your *root* Rails page (the first page that is displayed on the app), add the controller action `hello` to the `application_controller.rb`

```
class ApplicationController < ActionController::Base
  protect_from_forgery with: :exception

  def hello
    render html: "hola mundo¡"
  end

end
```

This defines an action that returns a string which will be rendered as a text on the Rails page. Now we need to tell Rails to use `hello` instead of the default page. To do this, edit the router which determines where to send requests that come in from the browser. We will be changing the *root route (the default landing page displayed by localhost:3000)*, `config/routes.rb` to have the following code:

```
Rails.application.routes.draw do
    root 'application#goodbye'
end
```

This code includes instruction on how to define the root route by directing the `application` controller to the `hello` action so that when the website is displayed the 'Hola Mundo¡ text will be displayed'

Version Control with Git

Version control systems allow us to track changes to our project's code, and collaborate easily. We will be using Git Version control and Github (repository hosting and sharing site) to deploy the app to an online website.

Git System Setup:

1. `git config --global user.name "Your Name"`
2. `git config --global user.email <your email>`

Repository Setup:

1. `git init` `#initialize empty Git repository`

2. `git add -A` `#add all the project files to the repository except '.gitignore' files`

Note: You can use `git status` to check if all of the files had been added

3. `git commit -m "Initialize repository"` `#tell Git we want to keep the changes by committing them to the repository`
    - You can use `git log` to see the list of commit messages.
    - You can use commits to reset the code back to your previous version of the app by 'Force Overwriting' the current changes using `git checkout -f`
4. Create a new Github Repository on your Github account. Copy the URL of the remote repository
5. `git remote add origin <remote repo URL>` `#sets the new remote`
6. `git remote -v` `#verifies the new remote`
7. `git push -u origin master` `#pushes changes in the local repo to remote repo specified as the origin`

Branches in Git:

- `git checkout -b <branch-name>` `#switched to a new branch`
- `git branch` `#lists all the local branches and '*' identifies the current branch`

Merging branches:

- `git checkout master` `#switched to branch 'master'`
- `git merge <branch-name>` `#merge branch-name to current branch`
- If you want to delete an unwanted branch use `git branch -d <branch-name>`

Deploying Rails App using Heroku

Heroku is a hosted platform built specifically for deploying Rails and other web applications. Sign up to Heroku and Install Heroku on the command line if you haven't already (you can check if Heroku is already installed on Terminal using `heroku --version`). Heroku uses the PostgreSQL database, and so we need to add the `pg` gem in the production environment. We also need to change the version of `sqlite` to 1.3.13 since SQLite Database is not yet supported by Heroku.

- Add `gem 'sqlite3', '1.3.13'` to your Gemfile underneath `group :development, :test do`
- Add the pg gem to the Gemfile

```
group :production do
    gem 'pg', '0.20.0'
end
```

- Bundling without production to avoid local installation of `pg` gem:

`bundle install --without production`

- Login to Heroku using `heroku login`
- Add your SSH key using `heroku keys:add`
- Create a place on the Heroku servers for the app to live using `heroku create`
- Push the master branch up to Heroku using Git: `git push heroku master`

And that's it! If you want to view the deployed Heroku app, type `heroku open` on Command Line and it should open the app in a browser for you.

There you have it; you have successfully created a 'hello world' web app on Rails, saved it on Github using Git and deployed your app online using Heroku. Nice Job!

Deploying from various branches to same app

Suppose you have two branches, but you want to deploy the code for the app to the same Heroku App. Then you could add, commit, and push all of the code to Github, and then call the following function: `git push heroku <branch_name>:master`

Heroku Commands

Check out *Heroku Command Line* to learn more commands that can be used to work on your Heroku Apps. You can also use `heroku help` to see a list of Heroku commands.

One example of a useful Heroku command is `rename;` it can be used to rename the application, which will allow you to customize your Heroku app URL.
`heroku rename <new-app-name>`. Now if you open the app again using `heroku open`, it will show a new URL for your Heroku app. Yay!

Exercise Two: Toy App

Use Scaffold Generators(aka MAGIC 🧙) to rapidly generate an app

Scaffolding is the fastest way to generate models without actually understanding how MVC architecture works

Setting up a new Toy App

- `rails new toy_app`
- `cd toy_app/`

- Update the Gemfile the same way you did in *Hello World App* by changing the SQLite version to 1.3.13 and adding PostgreSQL gem to production. You should also `bundle install --without production`

```
group :development, :test do
  gem 'sqlite3', '1.3.13'
  gem 'byebug',  '9.0.6', platform: :mri
end

group :development do
  gem 'web-console',           '3.5.1'
  gem 'listen',                '3.1.5'
  gem 'spring',                '2.0.2'
  gem 'spring-watcher-listen', '2.0.1'
end

group :production do
  gem 'pg', '0.20.0'
end
```

- `git init`
- `git add -A`
- `git commit -"initialize repository"`
- `git remote add origin <remote repo URL>`
- `git push -u origin --all`

Deploy the Toy App

Deploy the toy_app with "hello world" just like in exercise one.

- In `app/controllers/application_controller.rb`, add following action:
  ```
  def hello
      render html: "Hello, world!"
  end
  ```
- In `config/routes.rb`, add following redirection of root controller to action:
  `root 'application#hello'`
- Commit changes and push to Heroku:
  ```
  git commit -am "Add Hello"
  heroku create
  git push heroku master
  ```
- Rename app (if you want): `heroku rename <new-app-name>`
- Open newly deployed app using `heroku open`

user model Design

We'll create a *user* model (a Ruby object) which will interact with the Postgres database. Users of the toy app will have identifiers such as `id` (integer), `name` (string), and `email` (string).

Users	
id	integer
name	string
email	string

The table on the right represents the *user* model which corresponds to a *table* in a database, and the identifiers are the columns in the database.

`microposts` model design

To create a microposts model, we'll need the following identifiers: `id` (integer), `content` (text), and `user_id` (integer). `user_id` is used to associate each micropost with a user (owner of the post).

microposts	
id	integer
content	text
user_id	integer

Create a `user` model

We will be using scaffolding to generate the *user* model (since learning scaffolding was one of the goals of this exercise). At the end of the *user* model scaffolding command, we will add the identifiers and its types to generate columns in the database.

```
rails generate scaffold User name:string email:string
```

If you want to replace an existing User model, use `--force` to forcefully replace the existing model with a new one.

To proceed, we need to *migrate* the database using `rails db:migrate`. This will update the database with our new *user* data model. For versions before Rails 5, use `rake`, which is a 'Ruby Make'. Hence to update the database use `bundle exec rake db:migrate`.

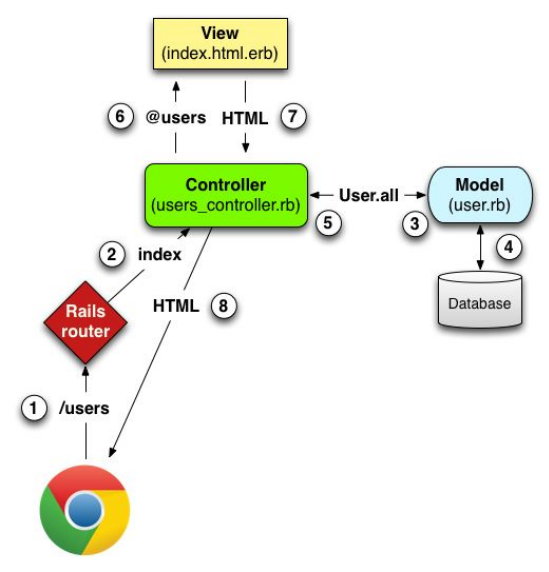
If you run `rails server`, and then open the URL *http://localhost:3000/users*, you can see the new user model generated by scaffold command, where you will be able to create a new user, and edit/ destroy existing user.

- When you open the URL *http://localhost:3000/users*, you are on the `index` action within the `users_controller.rb` controller;
- */new* directs you to *new* controller action,
- */edit* directs you to *edit* controller action, and so on.

- Here is a table of URL→Action→Purpose:

URL	Action	Purpose
/users	`index`	page to list all users
/users/1	`show`	page to show user with id `1`
/users/new	`new`	page to make a new user
/users/1/edit	`edit`	page to edit user with id `1`

How does MVC allows `/users` to be viewed?



1. The browser issues a request for the /users URL
2. Rails routes /users to the index action in the Users controller: The `config/routes.rb` in the app routes the /users request to users controller (`resources :users`). Then the root of users controller directs *users* request to *index* action.
3. The `index` action asks the User model (`app/controllers/users_controller.rb`) to retrieve all users (with the help of `User.all`).
4. The User model pulls all the users from the database.
5. The User model returns the list of users to the controller by placing them in the variable `@users`.
6. The controller captures the users in the `@users` variable, which is passed to the index view. Variables that start with `@` sign, called instance variables, are automatically available in the views. Therefore `@users` will have a line for each identifier of the model.
7. The view uses embedded Ruby to render the page as HTML.
8. The controller passes the HTML back to the browser.

Problems with Scaffolding

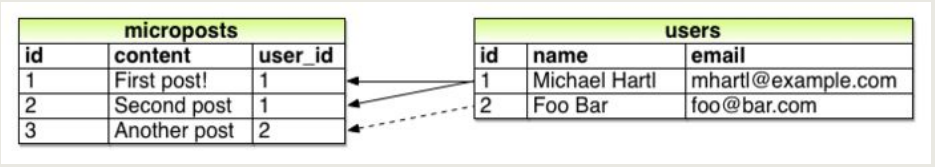
- **No data validation:** blank or invalid data is accepted
- **No authentication:** scaffolding doesn't have logging in/out features
- **No tests:** No tests for data validation/ authentication
- **No consistent site style or layout** (such as navigation)
- **No real understanding** of how the model is generated & what the code means

Create *a micropost* model

```
rails generate scaffold Micropost content:text user_id:integer
Rails db:migrate
```

In the `app/models/microposts.rb`, add a content length validation using the following code: `validates :content, length: { maximum: 140 }`

To allow *associations* between multiple microposts and a user add `has_many :microposts` to `app/models/user.rb`. You also need to specify that each micropost *belongs to* a particular user by adding `belongs_to :user` to `app/models/microposts.rb`.

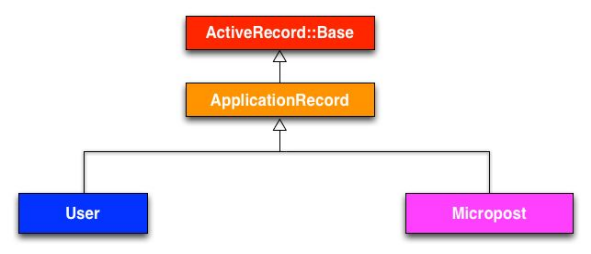


microposts		
id	content	user_id
1	First post!	1
2	Second post	1
3	Another post	2

users		
id	name	email
1	Michael Hartl	mhartl@example.com
2	Foo Bar	foo@bar.com

- To see the effect of *associations* type `rails console` at the command line. This will open up the console for the app in the command line. Then retrieve the first user from the database typing `first_user = User.first`. Later, retrieve the micropost of the first user by typing `first_user.microposts`, which will display details of all of the microposts of the user with id 1.
- To see the effects of *belongs_to* type `micropost = first_user.microposts.first` and then type `micropost.user`. This will return the details of the user to which the micropost belongs to.
- To exit out of the console, type `exit` (duh).

Inheritance Hierarchies



In both User model and Micropost model, we'll see an inheritance specified to `ApplicationRecord`, which in turn inherits `ActiveRecord::Base` (which provides the ability to the user and micropost models to communicate with the database). Similarly, for both UsersController and MicropostsController, there is an inheritance of `ApplicationRecord < ActiveRecord::Base`

Deploying Toy App with new models and designs

- First, we need to add, commit and push all of the changes that we have made to the remote repository. *This has to be done before pushing anything to Heroku.*
  ```
  git status
  git add -A
  git commit -m "Finish toy app"
  git push
  ```

- Once the remote repository is updated, deploy the Heroku app by pushing all of the code to Heroku app:
  `git push heroku` (this assumes that Heroku app is created already. Otherwise run `heroku run` and then `git push heroku master`)

- Now that all of the code is available to the Heroku app, migrate the Heroku database to create the connection between the models and the database:
  `heroku run rails db:migrate`

Exercise Three: Mostly Static Pages

Making static pages is a highly instructive exercise, rich in implications. We will also learn how to do automated testing on these pages.

Initialize and deploy your sample app

- `rails new toy_app`
- `cd toy_app/`

```
group :test do
  gem 'rails-controller-testing', '1.0.2'
  gem 'minitest',                 '5.10.3'
  gem 'minitest-reporters',       '1.1.14'
  gem 'guard',                    '2.14.1'
  gem 'guard-minitest',           '2.4.6'
end

group :production do
 gem 'pg', '0.18.4'
 gem 'fog', '1.42'
end
```

- Update the Gemfile the same way you did in *Toy World App* by changing the SQLite version to 1.3.13 and adding PostgreSQL gem to production. In addition, add a test group as well for your automation testing.
- You should also `bundle install --without production`
- `git init`
- `git add -A`
- `git commit -"initialize repository"`
- `git remote add origin <remote repo URL>`
- `git push -u origin --all`
- Deploy the app in Heroku just the way you did in Toy App.
- If you want to deploy an app from multiple branches, check out this section.

Develop a custom static page using `rails generate`

We will be developing a static page as the first step towards making a dynamic page.

- Create a new branch for the static page version of the sample app
  `git checkout -b static-pages`
- Create a controller *StaticPages* for different pages you want to build. We will be building two pages (*home* and *help*) to start off with.
  `rails generate controller StaticPages home help`
- Now that the StaticPage controllers are built, you can customize your app by changing the text and style of `home.html.erb` and `help.html.erb` within `app/views/static_pages` (just like you do in normal HTML pages).
- Add, commit everything from new branch to Github.
  `git add -A`
  `git commit -m "Adding static page controller"`
- Since it is the first time we are pushing from the *static-pages* branch, we have to specify that we are upstreaming (`-u`) from `static-pages` branch to `origin`.

Note: `rails generate controller` automatically updates the `routes.rb` file by adding the rules for the static_pages. `get 'static_pages/home'` maps requests for

URL/static_pahes/home to the home action in the StaticPages controller. The `get` command is arranging to respond to a GET request (which means "get a page").

Develop a custom static page manually

Suppose you want to create a static page called *'about'* manually. In that case, you'll have to do the following things:

1. In the `config/routes.rb` file, add `get 'static_pages/about'`.
2. In the `app/controllers/static_pages_controllers.rb`, add a new `about` action
3. Create a new template file (`about.html.erb`) in the `app/views/static_pages` directory by typing the following command: `touch app/views/static_pages/about.html.erb`.
4. Make changes to the about.html.erb file as you wish.

So what have you done?

Once you run the server using `rails server`, visit `URL/static_pages/home`. There you will see a page that says "StaticPages#Home". So what does this mean? This means that you created `StaticPagesController`, which is a Ruby Class that inherits its behavior from `ApplicationController`. When visiting `URL/static_pages/home`, Rails look in the Static Pages controller and executes the code in `home` action and renders the view. Since home action is empty, Rails simply just renders the view which is what you see on the browser. When Rails render the view, it will generate a file called `home.html.erb` which is what that is displayed on the browser.

Undoing things

- If you want to get rid of a controller: `rails destroy controller StaticPages home help`
- If you want to get rid of a model: `rails destroy model User`
- If you want to undo a database migration (db:migrate): `rails db:rollback`
    - If you want to rollback to first version: `rails db:migrate VERSION=0`

Rails Shortcuts

Full command	Shortcut
`$ rails server`	`$ rails s`
`$ rails console`	`$ rails c`
`$ rails generate`	`$ rails g`
`$ rails test`	`$ rails t`
`$ bundle install`	`$ bundle`

Automated Testing

Automated testing allows us to check if the code that you wrote is working the way it should work. If we create tests as we develop a new feature of function, it will save us a lot of time from checking if each component of the app works.

When you generated the *StaticPages* controller, Rails automatically generated a test file in the `test/controllers` directory, where there are individual tests generated for both *home* and *help* controller actions. Each test simply gets a URL and verifies the result is a success.

To run the tests, type `rails test`, which will respond with the result of the tests.

Note: For funsies, if you want to add colors to the results of the test (Red and Green), add the following gem to the `test/test_helper.rb` file

```
require "minitest/reporters"
Minitest::Reporters.use!
```

Automated testing using Guard

The Guard gem monitors the file system and if there are any changes to the files, it will automatically run the test for you! This makes integration and regression testing very easy. If you followed the initialize app section of this exercise, then you have already added the guard gem within the test group of your Gemfile.

Now you have to initialize the guard by calling `bundle exec guard init`. This creates a Guardfile, which determines what tests to be run when the files have changed. Replace the code in the new Guardfile with the code in the following website.

To prevent conflicts between Spring and Git when using Guard, you should add the `spring/` directory to the `.gitignore` file used by Git to determine what to ignore when adding files or directories to the repository. To do that, open .gitignore file and add the following line of code to the end of the file: `/spring/*.pid`

To run guard, open a new terminal, and run `bundle exec guard`. This will wait for any changes in the file system to be saved at the directory in which guard is running. If there are any changes saved in the file system, the tests will automatically run and report the results on the new terminal. Handy Dandy!

Create your own test

Suppose you want to check if the Home page has the <title> tag as “Home | Sample App”. You can do that by adding the following *assertion* to the “*should get home*” test:

`assert_select “title”, “Home | Sample App”`

If you want to avoid any repetition while testing, you can create a setup def where you can save variables that will be used multiple times during the test. For example you can create a variable `@base_title` and assign it the value “Sample App”. Then you can use that variable in the assertion with the help of `#{}`.

```ruby
def setup
  @base_title = "Sample App"
end
```

```ruby
assert_select "title", "Home | #{@base_title}"
```

Embedding Ruby in HTML page

Suppose you want to change the title of each page using Ruby instead of manually changing the name. In that case, you can embed the Ruby code to your HTML page in the following manner:

```
<% provide(:title, "Home") %>
<!DOCTYPE html>
<html>
  <head>
    <title><%= yield(:title) %> | Ruby on Rails Tutorial Sample App</title>
```

`<% … %>` executes the code.
`<% … %>` executes the code and *inserts* the result into the template.
`provide` function associates `“Home”` with `:title`.
`yield` function inserts the the value of to the <title> tag

Exercise Four: Focusing on 'Ruby' on Rails

In this chapter, we will be focusing on the important elements in Ruby that are important to Rails.

Create a new branch on your sample app

- `git checkout -b rails-flavoured-ruby`

Helpers

Helpers are functions (`def`) that you can create to be used in the views; You can view, edit and create helpers in `app/helpers`.

Creating a custom Helper

Suppose you want to have custom titles for your pages; to do that you can create a custom helper called '`full_title`', which returns a base title if no page is defined and adds a title along with a vertical bar and a base title, if one is defined. Here is an example of a full_title helper:

```
def full_title(page_title = '')
    base_title = "Ruby on Rails Tutorial Sample App"
    if page_title.empty?
        base_title
    else
        page_title + " | " + base_title
    end
end
```

String and Methods

You can experiment with Ruby strings and methods using the rails console
`rails console`
Comments: starts with `#` and extend to the end of line

Strings:

- Empty string: `""`
- Non-empty string: `"foo"`
- Concatenate string: `"foo" + "bar" => "foobar"`
- String interpolation:
  `>> name = Tom => "Tom"`
  `>> "#{name} Hartl" => "Tom Hartl"`
- *Printing:*
  `puts "foo"` (*prints* prints string and return *nil*. It also prints a new line)
  `Print "foo"` (prints string without a new line)
- Single Quote Strings: works the same way as double quotes, but doesn't allow interpolation unless it has a backslash (`\`) before special characters/ character combinations

`"#{foo} bar"` is the same as `'\#{foo} bar'`
`"\n"` is the same as `'\\n'`

Objects: everything in Ruby is an object (even *nil*). Objects responds to messages such as `.length` and `.empty?`. The `?` at the end of `.empty?` means the method returns a boolean

Boolean:

```
>> x = "foo"
=> "foo"
>> y = ""
=> ""
>> puts "Both strings are empty" if x.empty? && y.empty?
=> nil
>> puts "One of the strings is empty" if x.empty? || y.empty?
"One of the strings is empty"
=> nil
>> puts "x is not empty" if !x.empty?
"x is not empty"
=> nil
```

```
>> string = "foobar"
>> puts "The string '#{string}' is nonempty." unless string.empty?
The string 'foobar' is nonempty.
=> nil
```