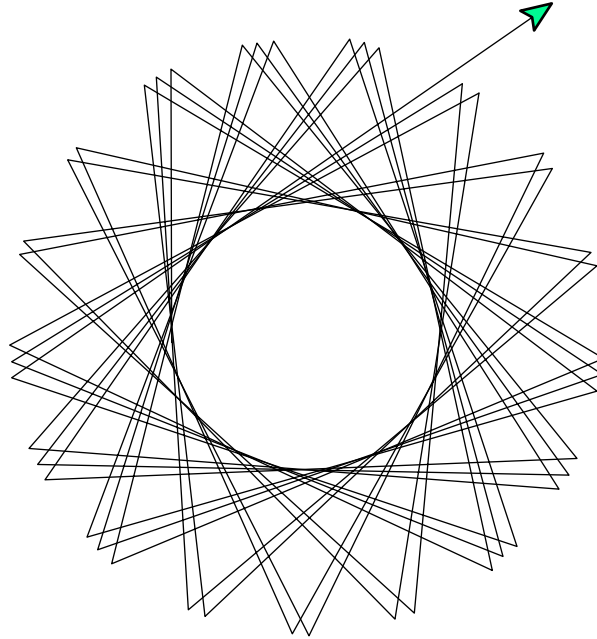# *Explorations in Computing*

Lab Manual for Linux

*RubyLabs Version 0.9.2*

John S. Conery

[2011-07-15 r451]

How to Use This Manual

This manual contains instructions for installing and using the software that accompanies the textbook *Explorations in Computing: An Introduction to Computer Science*. All the software is written in a language named Ruby. The Ruby system and the lab modules are open source software that can be freely downloaded from the Internet.

These instructions were written with a beginning student in mind. We do not assume you know anything about Ruby or any other programming language. We do assume, however, that you are a proficient computer user, that you are comfortable using a web browser such as Firefox, and that you know how to download a file and save it on your hard drive.

The manual is organized to allow you to do the minimal amount of work to install Ruby and the software used for lab projects, and then refer back to other sections later as you start to work on projects. Very few people will read the entire manual; in fact, students who are already familiar with the Terminal application and editing files with gedit might only have to follow the instructions in the first part of the manual.

Part I of the manual has sections on installing Ruby, a graphics library named Tk, and the lab software, which is called RubyLabs. If you are going to use a computer that has
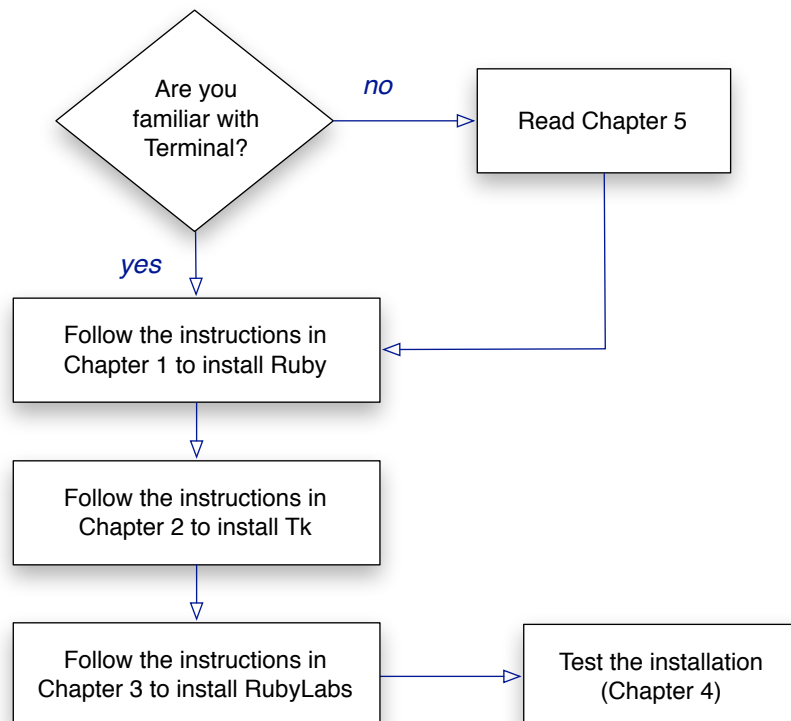
already been set up for you (*e.g.* a system in a campus computer lab) you can skip this part, otherwise carefully follow all the instructions in Chapters 1 to 4 to install the software on your own computer. Some of these steps require you to use Terminal, an application where you type commands on the keyboard to carry out some of the steps in the installation. If you have never used Terminal you should read Chapter 5 to learn how to type command lines in Linux before you start the process in Chapter 1.

Part II of the manual has general information and suggestions that will be useful as you work on projects:

- Chapter 5 explains how to use the Terminal application, which is the main piece of software that will be used in every project.

- Chapter 6 has recommendations for setting up folders for projects and explains the terminal commands that work with folders.

- Chapter 7 explains how to work with a text editor to read and/or modify files used in projects.

- Chapter 8, on running Interactive Ruby (IRB), includes a set of suggestions for techniques that will help you be more efficient as you work on lab projects.

- Chapter 9 has a small project that uses the topics from the previous chapters: create a project folder, add a file with a simple Ruby program, and then run the program from the command line.

- Chapter 10 explains how to get detailed information about all the RubyLabs software, including functions that are not described in the text, for students who want to go more deeply into the projects and explore further on their own.

Finally, Part III has technical information on installing Ruby, Tk, and RubyLabs. It is intended for systems administrators or other people with experience installing and configuring software, and has advice for how to address problems that some students might have with their installation.

Contents

Part I: Installing the Software

The chapters in this section of the manual explain how to install the software you will need to carry out the tutorial projects in the text.

At a minimum you need to install the Ruby programming language and a set of software modules called RubyLabs. The third piece of software, called Tk, is used in labs that draw images on the screen. If you do not install Tk you will still be able to complete the tutorials, but you will have to skip the steps that create or update the graphical displays.

1

Installing Ruby

To install software on Ubuntu and most other varieties of Linux you need to run a program named `apt-get`. Make sure your computer is attached to the Internet, then start the Terminal application. Type this command to tell your system to install Ruby:[1]

```
$ sudo apt-get install ruby irb ri gem
```

You will be asked to enter your password. After you type in your password, hit the return key, and the system will begin the installation.

If you're wondering what all these words mean, `sudo` tells Linux you are the systems administrator (the "su" is short for "superuser") and you have the authority to do this installation. The word `apt-get` is the name of the program that does the installation. The remaining words (`ruby`, `irb`, *etc.*) are the names of the programs you want to install. They are all pieces that are required to make a complete Ruby installation.

---

[1]Don't forget the convention described in Chapter 5 for showing terminal commands in this manual: the prompt (the dollar sign in this example) is shown in black, and the only letters you type are in blue. To run this command, simply type the letters in blue after the dollar sign and hit the return key.

2

Installing Tk

Tcl is a programming language that is widely used for creating graphical user interfaces. Tk (which stands for "tool kit") is the software library that creates windows and places buttons and other graphics on the screen.

The RubyLabs modules use Tk for projects that display graphics on the screen during an experiment. The version of Tcl and Tk we use comes from an organization called ActiveState.

Download the installer from

```
http://www.activestate.com/activetcl/downloads
```

Use the process you normally use to unpack the archive (*e.g.* on Ubuntu double click on the file to start the Archive Manager).

When the archive has been expanded you will have a folder with a name that starts with something like ActiveTcl8.5... Start the Terminal application, navigate to this folder, and type the following command:

```
$ sudo ./install.sh
```

You will be asked to type your password. After you enter your password, hit the return key, and the system will begin the installation. Whenever you are asked to enter some information, just use the default values that are displayed by the installer.

3

Installing the RubyLabs Gem

One reason Ruby has become so popular is that it is easy for developers to make Ruby software available for other people to use. Ruby software packages are called "gems," and they can be downloaded over the Internet from a "gem server." One of the applications included as part of the standard Ruby installation is a program named `gem` that connects to a gem server to download and install gems.

To set up the software you will use for the lab projects in *Explorations in Computing* you need to run `gem` to have it download and install the package that contains all the lab modules, and then run a configuration script that is part of the package. Both steps require you to type a command in terminal emulator (the Terminal application). Be very careful when you type the commands: command line programs are extremely fussy about spelling and punctuation, and you need to type everything exactly as it is shown here.

Make sure your computer is connected to the Internet, and then start Terminal. Type this command to tell `gem` to install the RubyLabs modules:

```
$ gem install --user-install rubylabs -n .
```

Note: make sure you include the space and period at the end of the line, following the `-n`. It may look like nothing is happening for a few seconds, but eventually you should see a message from the `gem` program that it is installing `rubylabs`.

If you see an error message, refer to the Troubleshooting section on the next page – you might have an out of date version of the `gem` program, and if so, you just need to update `gem` itself and then try this command again.

As soon as `gem` tells you the installation is complete, type this command to run the configuration script:

```
$ ruby lab-setup.rb
```

When the configuration script has finished setting up your lab environment it will print a message announcing it has succeeded.

Troubleshooting

Here are some of the things that might go wrong during the installation process:

- If you see an error message that says "Command not found" it means the first word on a line you typed was a command the system did not recognize. Did you type `gem` or `ruby` correctly? The Terminal application is fussy about capitalization, so make sure you type the command in all lower case letters.

- If you typed "gem" or "ruby" correctly, it means the Ruby system is not in your "path." See the section on setting up your execution path in the notes for systems administrators on page 33.

- If you get an error message from the `gem` program when you type the first command on the previous page, it might have a phrase like "requires RubyGems version later than 1.5.0." If you see this message , type the following command to update the `gem` system itself:

      $ sudo apt-get update

  Now go back and retry the command that installs RubyLabs.

- If you get an error message for the second command that says something like "ruby: No such file or directory" it means you typed the name of the configuration script incorrectly. Make sure there is a hyphen between "lab" and "setup" and that the name of the script ends with ".rb".

- If you typed the script name correctly, and you are still getting a "no such file" message, type this command:

      $ ls -l lab-setup.rb

  If the output is "No such file or directory" it means `gem` didn't put the configuration script in your home folder.

  Download a copy of the script from `http://www.cs.uoregon.edu/eic`, making sure it is saved in your home folder, and then run the `ruby` command again. If you still get error messages, read Chapter 6 to make sure you are typing the `ruby` command in the same folder where the script was saved.

4

Testing Your Installation

After you have installed Ruby, the Tk graphics library, and the RubyLabs Gem, your computer should be all set to run the experiments described in the textbook, but we recommend you run some quick tests to make sure everything is installed and working.

First, open your terminal emulator window, if it's not already running, and start Interactive Ruby. Simply type a single word to start running Ruby:

```
$ irb
```

When IRB starts, it prints its own prompt, which is a pair of greater-than signs. If you have any trouble starting Interactive Ruby, you can find more information about running IRB in Chapter 8 of this manual.

When you see the IRB prompt just type the word "hello" and hit the return key. Two things should happen: the RubyLabs module will print a friendly greeting in your terminal window, and a new graphics window should appear on your screen with another message from RubyLabs.

If you see both messages – one in your terminal window and the other in the new graphics window – then your installation is complete. To exit from IRB, type the word "quit" and hit the return key. The graphics window will be closed, and the prompt in the terminal window will change back to the regular system command prompt.

At this point you can either continue with some more tests, described in the next paragraph, or continue reading some of the other chapters in the next part of this manual. If you see any error messages, refer to the section on Troubleshooting on the next page.

Unit Tests

The RubyLabs software consists of a set of modules, one main module that has software used in several different experiments, plus one specialized module for each chapter of the book. Each module includes what software developers refer to as a "unit test." These tests run each of the Ruby programs in the module, making sure they produce the expected results. For example, a program described in Chapter 3 of the textbook generates a list of prime numbers. One of the unit tests for this module runs the program and compares the output to a known list of primes.

To run the unit tests, simply start your terminal emulator and type this command:

```
$ gem test --force rubylabs
```

Note there are two hyphens before the word "force."

When `gem` starts the tests for a module, it prints the module name. You should see a list of module names, things like RubyLabs, BitLab, SieveLab, and so on. Each time the `gem` system runs a test it will print a period, so you will see lots of periods on a line after a module name. If a test fails, `gem` will print an error message and halt the tests for that module.

Troubleshooting

If you see an error message that says "Command not found" when you type the `irb` command first make sure you typed the command exactly as it's shown, with all lower case letters. If you still see this message, it means the operating system did not find the IRB system. See the trouble-shooting section at the end of Chapter 3 for suggestions on how to fix this sort of error.

If you get an error from the `gem` program when you run unit tests it might be because you have an out of date version of the Gem manager system. The group developing the Ruby Gems software recently changed the way unit tests are administered. If you see a message saying something like "Unknown command test" it probably means you have an older version of `gem`. Type this command to update the `gem` system and then try the unit test command again:

```
$ sudo apt-get update
```

Part II: Using the Software

The chapters in this section of the manual contain suggestions and hints for using the software you installed when working on lab projects. There are sections on effectively using the terminal emulator, suggestions for editing text, and how best to set up your file system for Ruby projects.

All of the reading in this part of the manual is optional – if you successfully installed the software you can start right in on the tutorial projects in the book. But you should at least skim these sections so you know what material is here, and you can come back and read these chapters as you need them.

5

Using the Terminal Emulator

When you run software for the projects in *Explorations in Computing* you will be using a program named Interactive Ruby, or *IRB* for short. The word "interactive" is an old term in computer science, dating to the time when users typed commands on a teletype or video display terminal. When a person used these systems, they would type a command on the keyboard, and the output from the command would be displayed on the terminal. The alternative to interactive computing was known as "batch processing," where a user would submit a long-running job to the system and then come back some time later (often the following day) to pick up the results in the form of a printed report.

Modern computers have large video monitors and a variety of input devices, including mice, graphic tablets, and trackpads, in addition to keyboards. Computer users today are more familiar with applications that have a different type of interaction, known as a *graphical user interface*, or *GUI*. Instead of simply typing commands, users click on buttons, select items from menus, drag items from one place to another on the screen, type URLs into text boxes, or any one of a variety of other operations.

Even though modern systems are designed to run applications with graphical user interfaces, it is still possible to run software that has a command line interface. The idea is to run an application that acts like an old-fashioned video display terminal. This application, which is known as a *terminal emulator*, typically displays a single window. When the emulator is running, everything you type in the keyboard will show up in the window, as if you were typing on an old-fashioned terminal. If you type a command that runs a program, the terminal emulator will start that program, and everything the program generates as output will also be displayed in the terminal emulator window.

An example of the difference between an old-fashioned command line interface and a more modern graphical interface is shown in Figure 5.1, which shows two different programs that convert temperatures between Fahrenheit and Celsius. With the GUI application, a user would type a value in the box labeled Fahrenheit and then click the "convert" button to have the application calculate the equivalent temperature in Celsius and display it in a second box. With the command line interface, the user types a command in the terminal emulator window. The command tells the system to run a program that converts temperatures, and the output from the program is also displayed in the terminal window. In the figure the string "celsius(80)" is a command that was typed by the user; it tells the system to compute the Celsius equivalent of $80°$F. The number 26 on the next line was printed by the computer when the program produced its result.
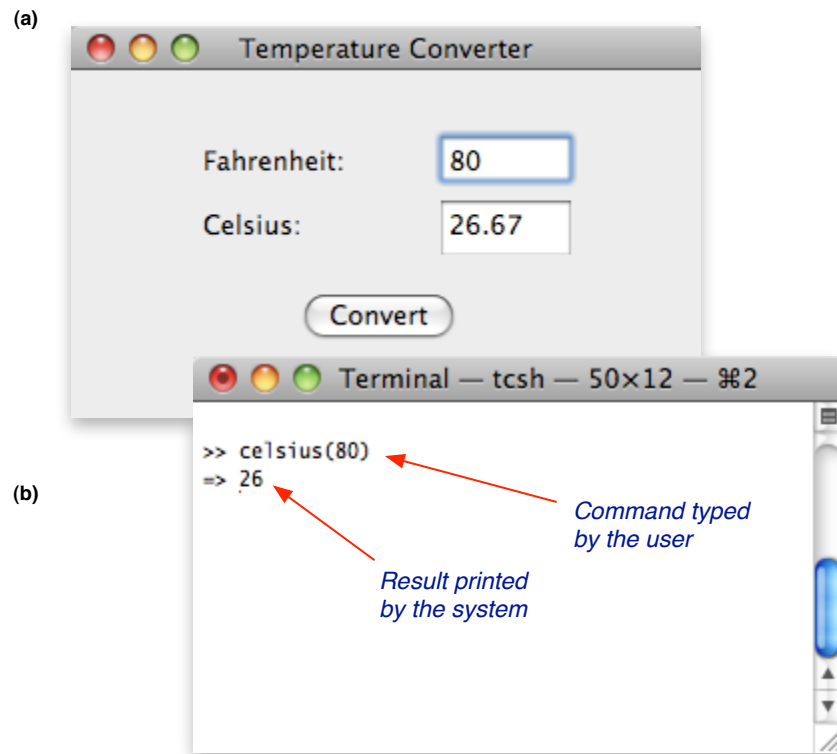
Figure 5.1: *(a) An application for converting temperatures, using a graphical user interface. (b) A terminal emulator running a program with a command line interface. These screenshots were taken on a Mac OS X system. There are also terminal emulators for Linux and Microsoft Windows XP systems.*

The terminal emulator in Linux is named, appropriately enough, Terminal. In Ubuntu you can find the program under the Application menu, as shown in Figure 5.2.

When the terminal emulator starts you should see a window that looks a lot like the one shown in Figure 5.3. Depending on how your system is set up, the program might print a message first. After the message, you will see a line that has what is known as a *prompt*, which is a special character the program prints when it is ready for you to type a command. The cursor will be displayed just to the right of the prompt. The prompt in this example includes the name of the computer the emulator is running on and the name of the user who is running the emulator, followed by the actual prompt character, which is a dollar sign. Again the string you see will be different, and may or may not have your computer name or user name; the main thing to look for is the last character on the line, which is the prompt character. Other common prompt characters are a percent sign or a "greater than" sign. Whenever you see this symbol in the terminal window, with the cursor next to it, the terminal is ready and waiting for you to type a command.

To see how the terminal emulator program works, type the letters `ls` (lower case L followed by lower case S) and hit the return key. This tells the system to run a program that will list the contents of your home folder. If you open a file viewer window you should see icons with these same names.
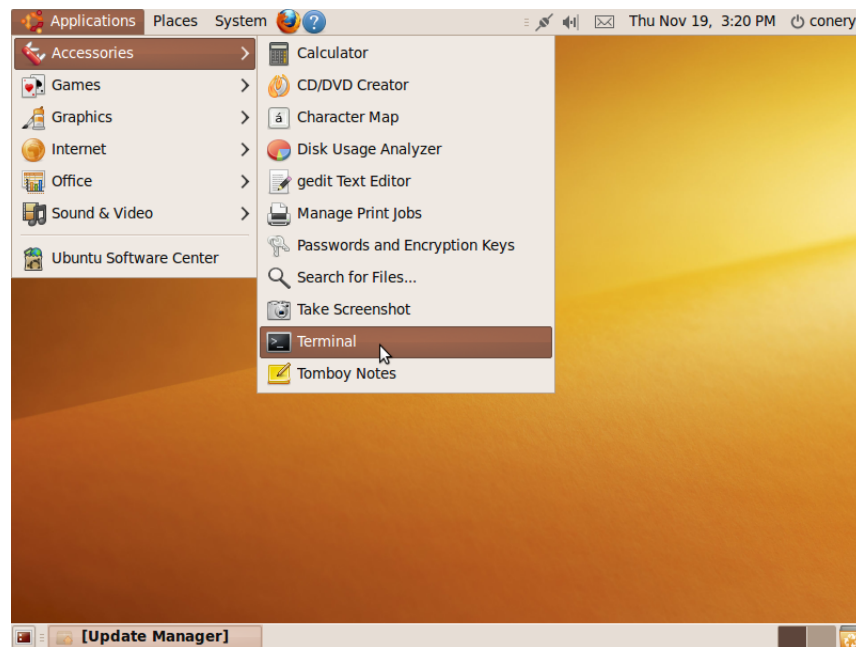
Figure 5.2: *In Ubuntu the Terminal application is one of the programs in the Applications menu.*

Conventions for Displaying Commands and Outputs

There will be several places in the rest of this manual, and throughout the textbook, where there will be an example of a command you should type, along with the output you should see. To make it easier to show these commands, we use the following convention:

- Characters you should type will be displayed in blue italics.

- Anything printed by the computer, including prompts and the output from running a command, will be shown in black.
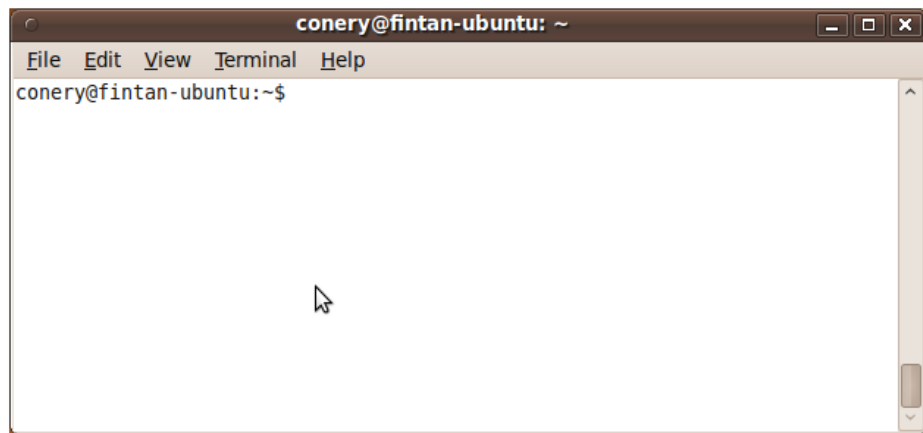

Figure 5.3: *The initial display of the terminal emulator window.*

An example is shown below. The `date` command tells the system to print the current date and time. On the system where this command was run, the prompt printed by the terminal emulator is a dollar sign. Because the text before the prompt varies so much from system to system we will only show the last character in the prompt. In this example, the command typed by the user is shown on the first line, and the lines printed below the command are the output from the command:

```
$ date
Fri Jul 15 11:18:33 PDT 2011
```

It would be a good idea to try this command now in your terminal emulator to verify it does what you expect.

Command Line Editing

To see what happens when you enter a command that has an error in it, type the `date` command again, but this time leave out the "a" and just type `dte` and hit the return key. You will see an error message.

For a short command like this you could simply retype the command, but for longer commands the terminal program allows us to go back and edit the line to fix the error. If you hit the up-arrow key (usually found on the lower right of the keyboard) the terminal will show you the command you just typed. Now you can use the left and right arrows to move the cursor forward and backward. Use the left-arrow key to move the cursor back two spaces and type the letter you left out the first time. The system will insert the letter into the command. Now you can hit the return key again, and the system will execute the new command.

This ability to go back and fix lines you typed previously is known as *command line editing*. It works when you are typing system commands, like the ones shown above, and it will also work in Interactive Ruby, when you are type Ruby expressions. A summary of the kinds of things you can do to edit command lines is given in Table 5.1.

You might want to try some more experiments now, to learn what the command line editing keys do. Here are some more commands to use in your tests:

`ls`	print the names of the files and folders in your home directory
`pwd`	print the name of your home directory
`man ls`	print the "manual page" (help page) for the `ls` command

The main thing to remember is that command line editing makes it possible to type a new command by editing something you typed earlier. After you try these editing operations a few times in lab projects, when you start entering more complex statements, they will become second nature.

Key	*Operation*
↑	Move to the previous command.
↓	Move to the next command.
←	Move the cursor left.
→	Move the cursor right.
^A	Move the cursor to the front of the line (hold down the control key when typing A).
^E	Move the cursor to the end of the line (hold down the control key when typing E).
delete	Erase the character to the left of the cursor.
return	Execute the current command line (the cursor does not need to be at the end of the line).

Table 5.1: *Keystrokes used in command line editing.*

Control Characters

One final note about typing command lines. The *control key* (on the bottom row of the keyboard on the left and right side) is like a shift key. When you see the notation ^X in the textbook or in this manual it means "press the control key while typing the letter X." Two of the editing commands in Table 5.1 are defined by control key combinations. Another place to use the control key is to type ^C to stop a program. For example, if you make a mistake and type a line that starts a program or an interactive Ruby command that seems to be running forever try typing ^C to interrupt the program and get back to the prompt.

6

Getting Organized

When you first set up your system you were given a "home folder" similar to the one shown in Figure 6.1. As you started using the system you may have saved files in this folder or left them on your desktop. After a while you probably realized that your system was starting to get cluttered and it was getting harder and harder to find things. If you haven't done so already, it's time to get organized.

What we suggest for doing the experiments in *Explorations in Computing* is to use the file browser to make a new folder, and name it something like "EiC". If you already have a scheme for organizing things, *e.g.* if you make a new folder for each of your classes and put them in a folder named "Classes," you can put this new folder in the one you set up for this course. Then, each time you start a new tutorial project in the *Explorations in Computing* text, make a new sub-folder for the project (Figure 6.2). For reasons we'll explain later below, it will be best if you choose a simple one-word name for each project folder.

Changing the Working Directory

Often programs you run by typing a command in a terminal emulator window will read data from a file or generate output data to store in a new file. A natural question is, if your files are organized and saved inside different folders created for separate projects, how can you tell a command line program where the files are? How do you tell it which folder to look in to find an input file or to write an output file?

The answer is that command line interfaces designate one folder at a time to be the "current" folder. By default, when a program looks for an input file to read, it will look in the current folder, and when it generates an output file the file is saved in the current folder. An older term for folder is "directory," and we usually refer to the "current folder" as the "working directory."

When you first start the terminal emulator the working directory will be your home folder.

When you are working on lab projects, you are going to want to make one of the project folders your working directory. For example, when you are working on the "Sieve of Eratosthenes" project, you want to have Ruby read and write files that are in the Sieve folder you put inside your top level EiC folder. The way to do this is to type a command named `cd` (short for "change directory") which tells the system to use a different directory as the current working directory.
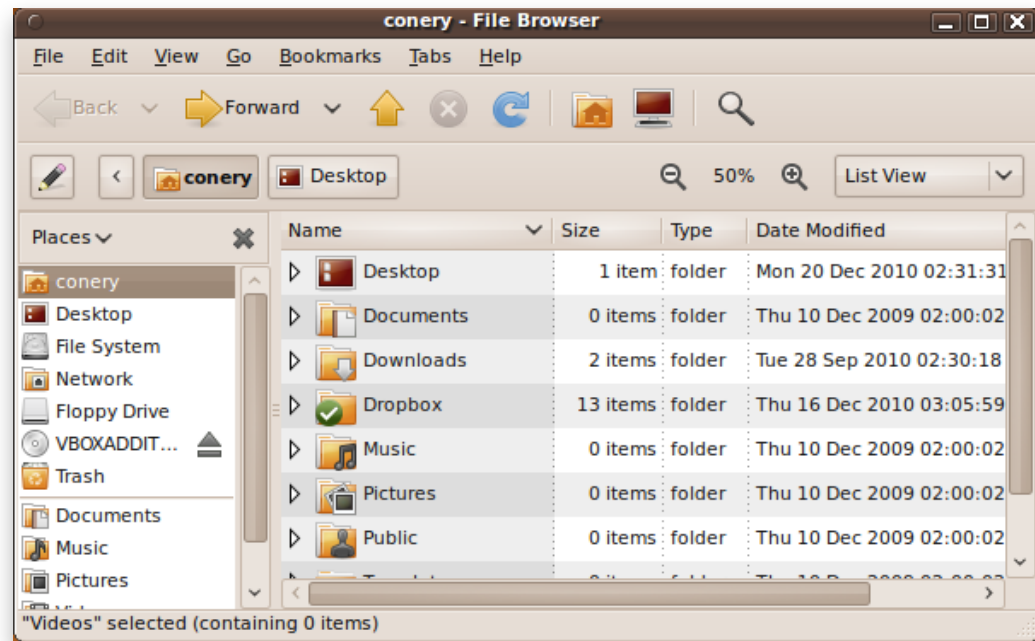
Figure 6.1: *The initial home folder for a new user account.*

Assuming your EiC folder is a top level folder inside your home directory, you can make the Sieve folder your working directory by typing two `cd` commands:

```
$ cd EiC
$ cd Sieve
```

Note that Terminal program in Linux is very fussy about capitalization, so make sure the folder names you type in these commands have the same combination of upper and lower case letters you used when you made the folder.

It is possible to combine both commands into a single command by typing `cd` and then a *path* that lists several folders. A path is just a sequence of folder names separated by slash characters. The command that moves from your home directory all the way to the Sieve directory in a single step is:

```
$ cd EiC/Sieve
```

If you're working on a project, and you see an instruction that says "navigate to your project directory," it is a short-hand notation for "type a `cd` command with a path that takes you to your project directory."

An example that shows how to navigate from your home directory to the Sieve project directory is shown in Figure 6.3. The first command asks the system to list the contents of the home folder, and the second uses a path to go to the Sieve folder inside the EiC folder.
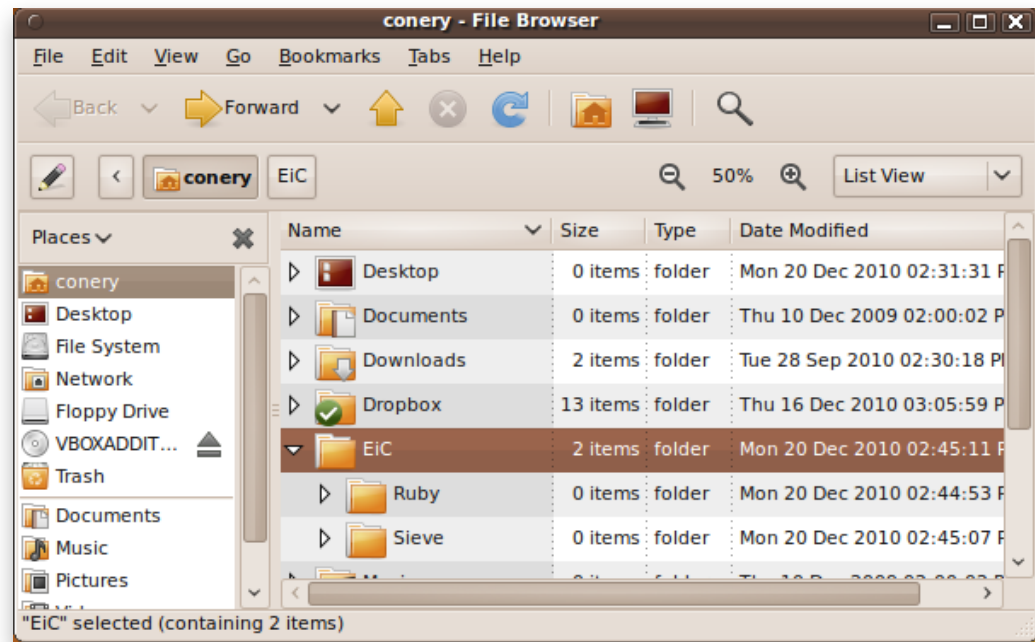
Figure 6.2: *A new folder named "EiC" and two sub-folders for the projects in Chapter 2 (Introduction to Ruby) and Chapter 3 (The Sieve of Eratosthenes).*

Figure 6.3: *A screen shot from a session with the terminal emulator, showing how to navigate to the directory set up for the "sieve" project.*

Folders with Spaces in Their Names

It is possible to put spaces in the names of folders, but if you do you have to be careful when you type the folder name in a command. For example, suppose you decide you want to name the top level folder something like "CS 101" instead of "EiC". Here is an example of what happens if you try to use the cd command to make CIS 101 your current directory:

```
$ cd CS 101
cd: Too many arguments.
```

This error message means the system thought CS and 101 were two separate names.

There is a way to combine the two words into a single long name. When you type this long name, put a backslash in front of the space. The correct command to make this folder the current folder would be

```
$ cd CS\ 101
```

With a little practice you will learn to read these long names without the backslashes getting in your way. If you find the idea of backslashes in names confusing, just make sure all your folder names are single words.

The Parent Directory

Two periods in a row in a path stand for "the parent folder." For example, suppose you are in the directory for the Sieve project, and you want to start working on the Iteration project. You need to type commands that move up to the parent folder and then back down to the Iteration folder:

```
$ cd ..
$ cd Iteration
```

It's also possible to combine these two commands into a single command by making a path that connects the two folders:

```
$ cd ../Iteration
```

♦ File Name Completion

The topic of this section is an advanced feature found in most terminal emulators. You can safely skip this section when you first read about Terminal, but if you practice typing a few commands using the feature described here you will quickly learn how useful it is and how it can save you a lot of time (and mistakes).

When you have long folder names (whether or not they have spaces in them) there is a time-saving shortcut: you can have the system type most of the name for you. Here's an example of how this works. Suppose you have a folder named "MyLatestRubyProject". To make this folder your current working directory, you can type cd, a space, and then the first few letters of the folder name. Then all you need to do is hit the tab key and the system will type the remaining letters in the name.

If you hit the tab key and nothing happens it means you didn't type enough of the name to make it unique. For example, if you also have a folder named "MyOtherProjects" and you

only type "My" after the `cd` the system can't tell which folder you want to go to. In that case, just keep typing letters until the name is no longer ambiguous, and then when you hit the tab the system will complete the name for you.

You can use this feature when you type a long path involving several folders. For example, if the path you want to type is `EiC/Sieve` you can type `cd`, a space, the letter "E", and the tab key. If no other folders start with "E" the system will type the "iC", and then you can type the slash, the letter "S", and the tab again.

7

Editing Text

There are two main types of applications you can use to create and edit Ruby programs.

The first type of application is a *word processor* like Open Office. The advantage of using a word processor is that you probably already have one of these applications on your system and are familiar with how it works, so you won't need to learn any new software. The problem, however, is that word processors are set up to make structured documents like term papers and journal articles. A structured document has section headers, text formatting information like font size and color, and they often have things like figures and tables. All this extra information is stored in the file along with the document itself.

Programs written in a language like Ruby do not have section headers or any other extra information. They are simple pieces of text, stored in what are known as *plain text* files. A word processor can open a plain text file, to show you what's in the file, but if you try to make any changes and save the result the application will want to turn it into a structured document. If you want to make changes to a file, or if you want to create a new Ruby program or data file from scratch, you have to make sure you tell your application to save the file in a plain text format. There may also be a few unexpected problems. For example, some applications don't recognize files with names ending in `.rb` (the standard file name extension for Ruby programs) as text files so you won't even be able to open the files in the first place.

Instead of using a word processor it is better to use a simple *text editor*. These applications don't try to do any fancy formatting, and often include features that are included specifically to help edit programs. For example, most text editors will balance parentheses, to make sure there is a closing parenthesis for every opening parenthesis, and many will display letters enclosed in quotes in a different color so they stand out and it's easier to find the ending quote.

It may sound like an extra hassle to learn a new application, but text editors are very simple programs, and if you know how to use a word processor it is very easy to learn how to use the text editor. If you are planning on editing the programs used in the labs, or writing your own new programs, the time and effort spent learning how to use a text editor is probably a good investment that will pay dividends later.

There are many sophisticated text editors available for Linux systems. For the lab projects in the book, an application named `gedit`, which comes with Ubuntu and other Linux systems, is probably the best choice. You can find it in the Applications menu, under Accessories.

8

Suggestions for Running IRB

The projects in the textbook all use a part of the Ruby system known as Interactive Ruby (IRB). As you work on a project, you will see a set of Ruby expressions. To complete a project, you need to type these expressions into IRB to see how Ruby carries out certain operations. For example, in the project that explores an algorithm that generates a list of prime numbers, you will enter an expression that asks Ruby to create a list of primes between 2 and 100. As the program runs, you will be able to see the list as it is being created.

To run IRB, simply start your terminal emulator (the Terminal application) and then just type the word `irb`.
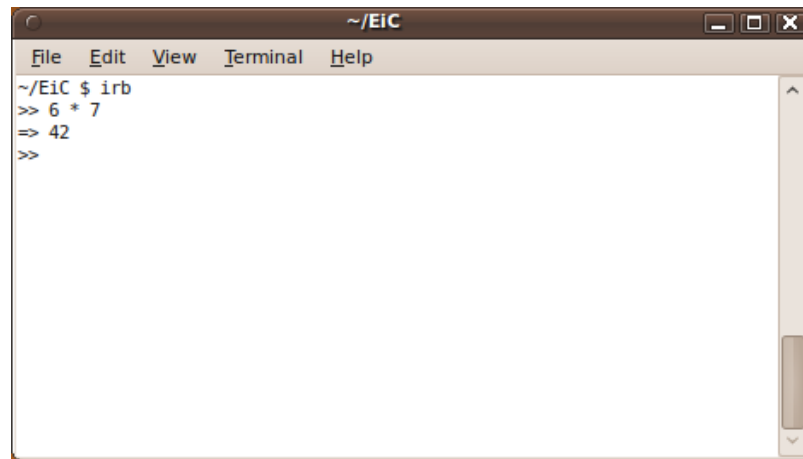
Using IRB is very much like using the terminal emulator itself. The system will print a prompt to let you know it is ready for you to enter an expression. You type in an expression, and at the end hit the return key. Ruby will evaluate the expression and print the result. An example of what the terminal window will look like while it is running IRB is shown in Figure 8.1.

One of the things the installation script does is set up your IRB configuration so the prompts look like the ones in the textbook. The standard prompt is two "greater than" signs in a row. After you enter an expression, Ruby will print the characters => to let you know it is printing the value of the expression. The convention used in the book is that things you

---

Starting a New Tutorial Project

At the beginning of each chapter in the main textbook you will see instructions that tell you to start IRB. Whenever you see a phrase like "start a new IRB session in your project directory" you should:

- launch the Terminal application, if it's not already running

- type a `cd` command to go to your project directory

- start Interactive Ruby by typing `irb`

Figure 8.1: *A snapshot of a terminal emulator window during a session with IRB. The first line shows the user typed* `irb` *after the prompt from the operating system. When IRB is running, it prints its own prompt, which is two greater-than signs. The second line shows that after the user saw the prompt from IRB they entered an arithmetic expression and hit the return key. The third line shows the result printed by Ruby, and the fourth line shows Ruby was ready for the user to enter another expression.*

are supposed to type as part of a tutorial project are shown in blue slanted letters, to make them stand out. Everything printed by Ruby (including it's prompt) will be shown in normal black characters.

Here is a sample expression to enter into your IRB session to make sure the RubyLabs software was installed properly:

```
>> hello
=> "Hello, you have successfully installed RubyLabs"
```

The fact that >> is shown in black means it was printed by IRB (those are the prompt characters). You are supposed to type the characters in blue, the word `hello`, and hit the return key. Make sure you type the string exactly as it is shown, in all lower case letters. If the lab software was installed and configured properly, Ruby will print that friendly little message.

As another example, to see how Ruby multiplies two numbers type this expression:

```
>> 6 * 7
=> 42
```

Again, you just type the letters in blue italics, and Ruby will print the => and the answer, 42.

Often a tutorial will have instructions to enter a slightly different version of an expression you just typed. For example, the project might say "go back an enter the previous expression, but change the 7 to 8." To do this you have several choices. For a very simple expression like this you can just retype the entire expression, but this time type an 8 instead of a 7. For a more complicated expression you can cut-and-paste: use the mouse to select the first part of the previous expression, paste it in the terminal window next to the IRB prompt, and then continue typing the rest of the new expression.

However, the best way to enter a new expression that is very similar to a previous one is to use *command line editing*, which was explained previously in Chapter 5 of this manual. If you hit the up-arrow key, the system will display the previous expression. Now you can just hit the delete key to erase the 7, type an 8, and then hit the return key. Ruby will evaluate the new expression and display the result. Any of the command line editing keys shown in Table 5.1 can be used in an IRB session to edit a Ruby expression you typed earlier.

If you hit the return key before you are done entering a complete expression Ruby will print a different prompt. If you see the three characters `--+` on your terminal it means Ruby has read part of an expression and is waiting for you to type the rest. For example, suppose you want Ruby to evaluate $5 + 6$ but you hit the return key by mistake after you hit the plus key. Ruby will print this new prompt, and at this point you can type the rest of the expression (the number 6) and the return key. This is what the terminal will show:

```
>> 5 +
--+ 6
=> 11
```

What happened in the previous problem was that Ruby thought you hit the return key on purpose. Since expressions can become very complicated, Ruby lets us break them into several lines, and typing the return key is like inserting a space into the middle of an expression. The ability to break complex expressions into smaller lines is not something you will need to use very often in the projects, but it's worth knowing in case you ever find yourself in the middle of an IRB session and the prompt has changed from `>>` to `--+`. If you get into this situation by mistake, type something to complete the expression, or if that doesn't work, type ^C to interrupt IRB and get it back to the normal state (recall that the notation ^C is shorthand for "hold down the control key while hitting the C key").

9

Hello, World

A common exercise for a first program after installing a new programming language is to write a "hello, world" program. As a program it's trivial—all it does is print a message and halt—but typing the program into a text editor, saving it, and then running it is a good way to work out all the kinks in setting up a new environment.

1. Use the file browser to make a new folder named Hello. This folder should be a subfolder inside your top level EiC folder.

2. Start your text editor and tell it to create a new file.

3. Enter a single line that has the following text, exactly as it is shown here (if you're reading this lab manual on-line you might be able to copy this text and paste it into your new document):

   ```
   puts "hello, world!"
   ```

   This line is the Ruby command to print the letters between quotes (`puts` is short-hand for "print string").

4. Save the file in your Hello folder, giving the file the name `hello.rb`. The `.rb` at the end is a convention for naming files that contain Ruby programs.

5. Start your terminal emulator and enter a `cd` command that navigates to the Hello folder.

6. Type this command in the terminal emulator window:

   ```
   $ ruby hello.rb
   ```

7. If you see the string "hello, world!" in your terminal window (Figure 9.2) then everything is working as expected.
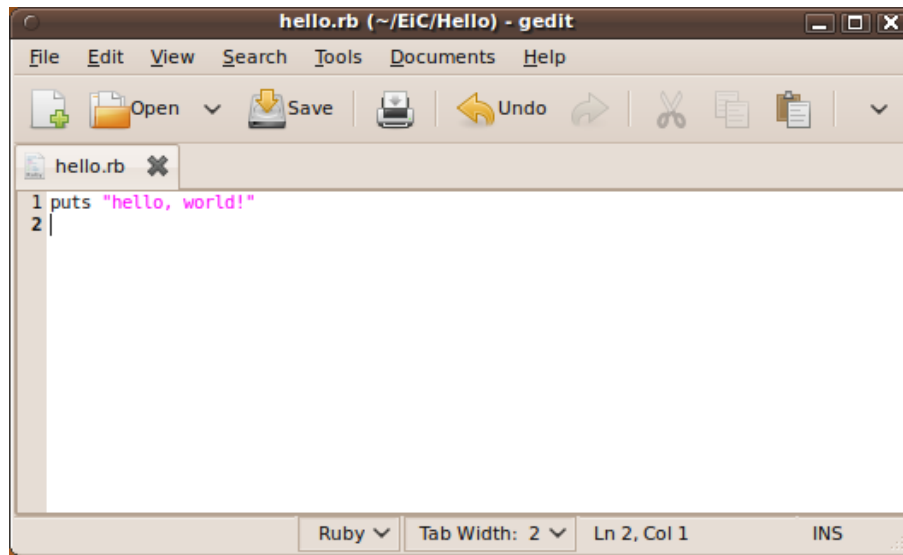
Figure 9.1: *The "hello, world" program in a* `gedit` *window. Save the file in the Hello folder inside your EiC folder.*
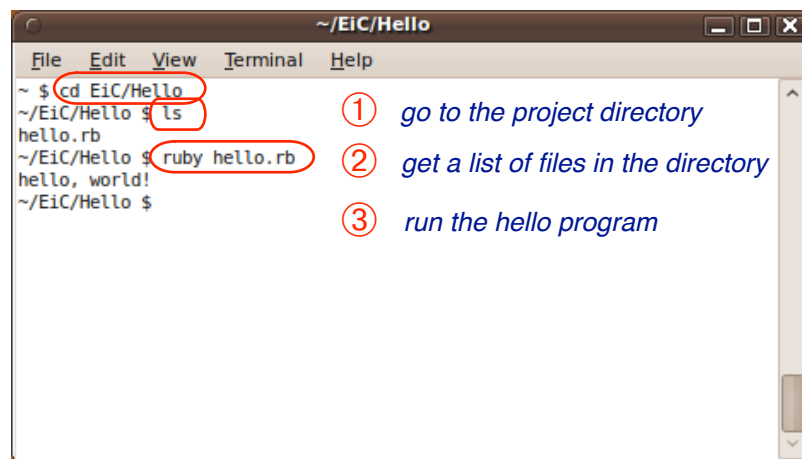

Figure 9.2: *Running the "hello, world" program.*

Troubleshooting

If, instead of seeing the string "hello, world!", you got an error message, here are some things to look for:

- If the error says something like

      Command not found

  then the error came from the Linux system and it means it didn't think `ruby` was a valid command. Make sure you type `ruby` with all lower case letters.

- If the error is something like

      ruby: No such file or directory -- hello.rb (LoadError)

  then it means Ruby is running (that's good news, at least) but it can't find your `hello.rb` file. Make sure you type the name `hello.rb` exactly as it's shown. If the name is correct, next check to see you are in the right directory. For example, if the file is in a project folder named `Hello`, but you typed `cd Sieve` then you are in the wrong folder.

  Type the command `pwd` (it stands for "print working directory"). Is this the directory where you saved `hello.rb`?

  If you're in the wrong directory use the `cd` command to navigate to the right place and re-type the command that runs the Ruby program.

- If you got an error message that starts off with something like

      hello.rb:1: ...

  it means the Ruby was running, and it found your program and started it, but there is something wrong with the program itself. The string `hello.rb:1` means Ruby found an error on line 1 of the file named `hello.rb`. Check to make sure `puts` is spelled correctly, and that both quote marks are in the file.

10

RubyLabs Documentation

The RubyLabs gem, like most Ruby software, comes with a substantial amount of documentation. For most projects, everything you need to know about the module used for the project is described in the textbook. However, there may be situations where you want to learn more about the various options that can be passed as arguments to a method, or you may want to learn about methods that are included in a module but not described in the text. You may also be interested in looking at the actual source code for a method.

All of this information is included with the RubyLabs documentation. In the Ruby world, this documentation is in a format called `rdoc`, and it is automatically generated by software that scans the Ruby module and extracts the documentation.

The easiest way to see the `rdoc` documentation is through the same `gem` program that you used to install the software in the first place. Start by opening a new terminal emulator window. If you already have a window open and are using it for IRB, open a second window. In this new window, type the following command:

```
$ gem server
```

If the command works properly, it will print a line that looks like this on your terminal:

```
Server started at http://0.0.0.0:8808
```

Note that there is no terminal prompt following this line – that means the `gem` command is still running.

What that output means is that the `gem` program has started a web server running on your computer. If you tell your web browser to connect to this server, it will show you the `rdoc` documentation for all the gems currently installed on your system. To connect to this server, simply start a web browser, and then enter the following URL:

```
http://localhost:8808/
```

You should see a page that lists all your gems, including `rubylabs` (you may also see some others that were installed automatically when you installed `rubylabs`). To see the documentation, click the link that says `rdoc` next to the current version of `rubylabs`. Now you should see a page that looks like the one in Figure 10.1. The page has three panels in the top section, and a documentation viewer in the bottom section. To start with, you will be
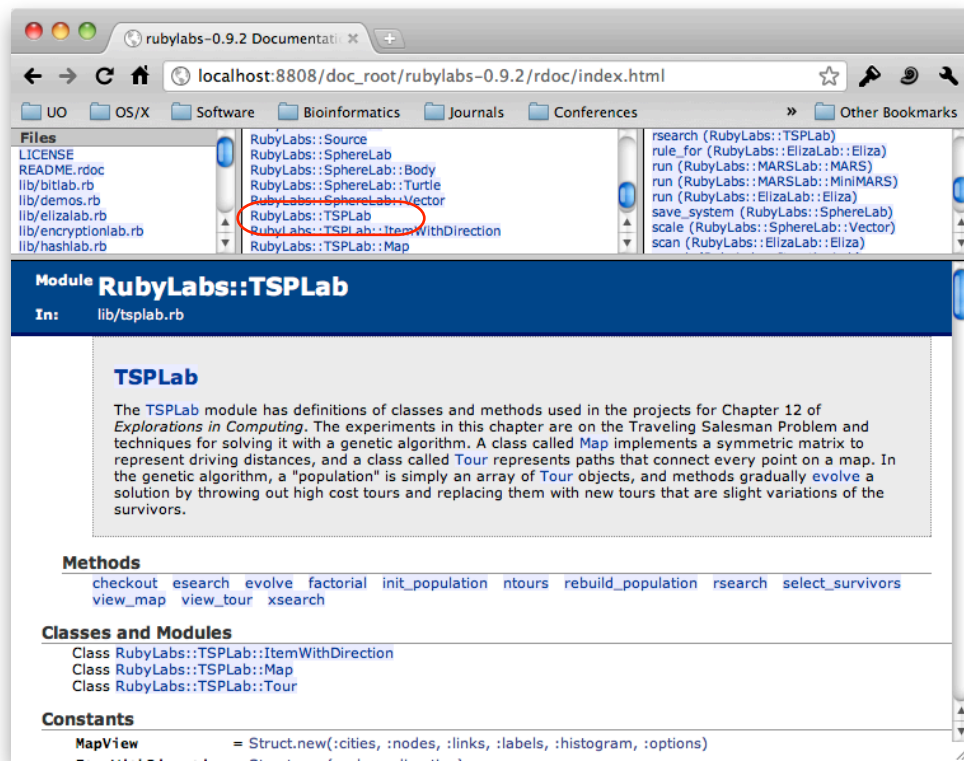
Figure 10.1: *The `rdoc` page for the RubyLabs modules.*

using the center panel in the top section, which has a list of all the modules in the RubyLabs gem.

Here is an example of how to use this documentation. Suppose you are working on the Traveling Salesman Project in Chapter 12. The lab module for this chapter is named `TSPLab`. In the documentation, all the modules have `RubyLabs::` at the front of the name. If you scroll down in the module list (center panel) you will find `RubyLabs::TSPLab` near the bottom (it's circled in red in Figure 10.1). Click on the module name, and the documentation for this module will show up in the bottom panel.

When the web page shows the documentation for a module, you will see all the methods that are defined in the module. For example, in the `TSPLab` documentation you will see a description of `esearch`, the method that does a full evolutionary search for a solution to the traveling salesman problem. Here you can read about the various options you can pass to the method, and see examples of how the method is called and the expected results.

If you would rather jump straight to the documentation for a particular method, you can use the rightmost panel in the top section. For example, if you want to look up the documentation for the `sieve` method from Chapter 3, simply scroll through the right panel until you see `sieve` and then click on the name.

If you want to see the complete Ruby code for any method, simply click on the method name in the bottom panel. A new window will pop up showing you the Ruby code.

When you are done looking at the documentation, you can go back to the terminal window where you typed the `gem server` command. Click in this window to make it the active window, and then type ^C to halt the server. You should now see the normal system prompt in the terminal window, meaning the server has finished and the system is ready for you to type a new command in this window.

Important Reminder: You can only connect to the `rdoc` web page when the `gem` program is acting as a web server. If you enter the URL and get a "server not found" error, start a new terminal window, enter the `gem server` command, and reload the page.

Part III: Information for Systems Administrators

This section of the manual has detailed information about how the RubyLabs modules are installed and what they expect from the host operating system. Most students should not have to read anything in this part of the lab manual. These chapters were written for systems administrators and others with experience installing and configuring software who either want to install the software in an environment where it can be shared by several computers or who are helping students who are having problems installing the software on their own system.

Execution Path

The installer should have put the Ruby executables (`irb`, `gem`, `rake`, *etc.*) in a standard place that is already in the student's path. Some places where Ruby could be hiding are:

- `/usr/local/ruby/bin` (where Apple puts the version preinstalled with Mac OS X)

- `/opt/local/bin` (where `dports` puts it)

- `/usr/bin` (where `apt-get` puts it on Ubuntu)

The Configuration Script (`lab-setup.rb`)

The configuration script named `lab-setup.rb` is distributed as part of the `rubylabs` gem. When students run this script, it initializes a file named `.irbrc` in their home directory. The file can contain arbitrary Ruby code which is executed at the start of each IRB session.

The setup script will create a new `.irbrc` if one does not already exist. It then adds commands to change various IRB settings. If the student already had a `.irbrc` the setup script checks to see if any of these things were already configured, and if so they are not changed.

The items configured by the setup script are:

- Set the default prompt so it matches the prompt found in the book (and in most other books and online documentation).

- Tell IRB the session will be using one or more gems.

- Automatically load the `rubylabs` gem at the start of each session.

One thing that can go wrong during this process is that the setup script was not saved in the student's home directory when the gem was installed. Most likely that means the student didn't include the `-n .` at the end of the install command. You can download the script from the book web site, or get it out of the gem install directory. To find out where the gem is installed, type

```
$ gem which rubylabs
```

You will see a path ending with `.../lib/rubylabs.rb`. The setup script is in a `bin` directory next to the `lib` directory.

Students will be able to complete all the projects if they do not run the setup script. They will just see a different set of IRB prompts, and will have to remember to type `require 'rubylabs'` and `require 'rubygems'` at the start of each session.

The RubyLabs Canvas

BitLab, SphereLab, and other modules that draw pictures on a canvas use Tk to create the graphics window and draw glyphs. All of the communication is one-way: students type Ruby expressions in their IRB session, which is running in a terminal window, and Ruby sends Tk commands down a pipe to a separate process, which parses the commands and figures out what to draw. None of the labs (currently) includes any widgets that trigger actions, and none of the Tk code tries to return any values to the IRB session. Any operations that appear to be based on getting information from a graphic object (*e.g.* getting the current location or orientation of the robot explorer in SphereLab) are simply getting that data from a proxy object kept in the IRB session, and if any operation causes the object to move, the graphics commands are sent to the Tk process.

To make sure the communication between IRB and Tk is working, the `hello` method that students call to make sure the installation is complete (see Section 4, page 10) will ask Tk to open a graphics window to display a hello message.

One thing that can go wrong with this architecture is that IRB can't find Tk, so students get an error message when they call a RubyLabs method that initializes the canvas. The lab module assumes `wish` (Tk's windowing shell) is in the user's path.

We initially chose Tk for these operations because Tk is part of the standard Ruby library and we expected it to be preinstalled on all three major platforms. Unfortunately, even though the Tk library is still part of Ruby, the Tk binaries are not being distributed as part of the version installed by the latest One-Click Installer for Windows, and Apple is not including it as part of the Ruby installation with Mac OS X 10.5 and later.

Our workaround is to have students install a third party version of the Tk system. Although this means an extra step for students when they first install the system, it is working well on all three platforms, and is consistent across all platforms (important when a class has students using different operating systems).