The computational experiments described in this book will be based on a programming language named Ruby. We will use Ruby to create data objects and implement algorithms that manipulate those objects. This chapter is a “warm-up exercise” that introduces the Ruby language and its interactive programming environment. There will be a simple project to implement a program that converts temperature values from Fahrenheit to Celsius. The program is trivial – it can be written on a single line – but going through the steps required to implement it is a good introduction to Ruby. Once we get through the basics of how to set up and run experiments with Ruby we'll be ready to tackle the more challenging projects in later chapters.

2

The Ruby Workbench

One of the first important decisions to make when writing a new program is how people will use it. For a program that converts temperatures from Fahrenheit to Celsius a programmer must choose how a person will specify the temperature value to convert and how the program will display the result. There may also be many other decisions; for example, the program may be a general purpose program that could also convert from Celsius back to Fahrenheit or do other related functions, which means there must be a way for a person to select which operation the program should perform. Taken together, the methods used by a program to get input data and present results define the program’s user interface.

Most programs that run on laptops or small desktop machines have a graphical user interface, or GUI, that consists of a set of windows, menus, buttons, and other controls. A simple interface for a temperature conversion program is shown in Figure 2.1a. Here the user is expected to enter a temperature value in the box next to a label and then click a button. When the button is pressed the program runs the algorithm that does the conversion, and the resulting value is displayed in the window.

Creating a full-featured application with a graphical user interface is very difficult, and writing even the simplest program requires knowledge of GUI techniques and a significant amount of planning and attention to details. The simple program shown in Figure 2.1a was written with the help of a system named MacRuby, which takes care of most of the organization of windows and buttons for OS/X applications written in Ruby, but as simple as this program is it still requires a substantial amount of programming effort.
For word processors, graphic design programs, and many other applications the effort required to create a GUI is an investment that is repaid many times over since the final program is usually more intuitive and easier to use, but for our purposes spending time on a fancy interface would be a distraction from our main goal of understanding algorithms. What we need an “workbench” that lets us experiment and explore – we want to be able to write small programs, run them, modify them, run them again, and see the results of our changes, all with the goal of understanding the algorithm implemented by the program. For the temperature conversion program, we are mainly interested in the mathematical expression that relates degrees Fahrenheit to degrees Celsius, and we want to see what happens when we ask Ruby to evaluate that expression.

Using an interactive programming language is similar to using a calculator: we type an expression, then the computer evaluates it and prints the result. So, for example, to ask the computer to calculate the sum of eight and seven, we would type the string 8 + 7 and the system would print 15. If the system knows about trigonometric functions, we would type \( \cos(1.0) \) to ask it to evaluate the cosine of 1.0. And as we will see as we work through the project in this chapter, after we build an interactive program to do temperature conversions, we be able to type the expression \( \text{celsius}(80) \) and the system will tell us the Celsius equivalent of 80° F (Figure 2.1b).

Using the word “interactive” to describe this style of computing might seem confusing at first. After all, a person using an application with a GUI is certainly interacting with the program, so why use “interactive” to describe a language that can be used like a calculator? With an interactive language the emphasis is on the use of the programming language itself. Instead of interacting with buttons, menus, windows, and other things that form the interface to an application, we are typing expressions in a programming language and interacting with the program and its data. The term “interactive programming language” has a long and rich tradition in computer science. LISP, one of the very first programming languages, was implemented as an interactive system programming system in the early 1960s.

In order to use a language interactively we need a way to enter expressions and a way for the system to show us the result. The first interactive languages used the technology that was available at the time. Users typed expressions on a teletype, electric typewriter, or one of the early types of display terminals, and the computer printed results on the same device.
A Note About Displayed Text

The Ruby programs shown in this book will be displayed with what is known as a fixed-width or “typewriter” font. This font will be used for the names of programs, text you should type on your terminal emulator, and strings of letters printed by Ruby.

To help distinguish between text you type during a session with Interactive Ruby and the things Ruby prints in response the text you should type will be shown in slanted blue letters. For example, when the Fahrenheit to Celsius conversion program is working, you will be able to type an expression to ask Ruby to convert 80°F to Celsius. This is what the IRB session will look like:

```
>> celsius(80)
=> 26
```

The >> on the first line is the prompt from IRB. The string following that is the text you type to ask Ruby to run your temperature conversion program. The text on the second line is what Ruby prints as the output of your program.

On a modern computer we run Ruby interactively with the help of a terminal emulator application. When the terminal emulator is active, everything you type on your keyboard (or cut and paste into the terminal window) is processed by the terminal emulator, and everything printed by the emulator is displayed in the window.

The system we use to run Ruby interactively is named, appropriately enough, Interactive Ruby, often abbreviated IRB. When you first start IRB, it will print a prompt to let you know it is ready for you to type an expression. IRB’s prompt is two greater-than signs. So when you first start the system, you should see this in your terminal window as soon as IRB is ready for you to type an expression:

```
>>
```

To have Ruby evaluate an expression, simply type the expression and hit the return key. Ruby will print => and the value of the expression. So to ask IRB to print the sum of 5 and 6, you just type `5 + 6` and IRB will print the result:

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

Ruby and other programming languages use an asterisk as the symbol for multiplication and a slash (/) for division:

```
>> 5 * 10
=> 50
>> 6 / 3
=> 2
```

When an expression has more than one operator, Ruby applies the operators according to their precedence. Since multiplication should be performed before addition, the result of `3 + 4 * 5` is 23:
The Ruby Workbench

>> 3 + 4 * 5
=> 23

If you want Ruby to evaluate the operations in a different order you can use parentheses:

>> (3 + 4) * 5
=> 35

When you are done using IRB just type quit and the program will exit:

>> quit

Tutorial Project

The first step in the tutorial project for this chapter is to make sure you can start the Interactive Ruby application. Ruby is open source software, which means it is freely available for anyone to download and install on their own computer. If you have a Mac OS/X system it comes with Ruby already installed; Windows and Linux versions are widely available via the Internet. If you need help installing or running IRB on your system you will find detailed instructions at the SoC web site or the Ruby project home page.

Once you have started IRB and see the >> prompt in your terminal emulator window you are ready to start in on the tutorial project.

1. Type a simple expression, e.g. 5 + 6 and hit the return key. Did you see the result you expected?

2. Try some simple expressions involving other operators:
   6 - 3
   3 * 7
   8 / 4

3. Try some expressions with and without parentheses:
   3 + 4 * 5
   (3 + 4) * 5
   8 - 4 / 2
   (8 - 4) / 2

4. Does Ruby care if you include spaces in the middle of your expressions? e.g. do you get the same result when you type 3+4 as you get when you type 3 + 4 with spaces before and after the plus sign?

5. What happens if you leave out an operand, e.g. if you type 3 + * 5 instead of 3 + 4 * 5?

6. Type an expression that mistakenly uses a symbol instead of a number, e.g. 3 + x. The error message you see has some unfamiliar terminology, but at this point you can glean some information: the message has the word “undefined” and the name “x” enclosed in quotes, so there’s a good chance Ruby was complaining about the x in that expression.

After completing these exercises you are ready to move on to the next section to learn more about numbers in Ruby. The experiments below reveal a little bit more about how Ruby evaluates expressions, and are worth doing if you want to gain some more experience with IRB so you can use it on your own projects.

7. In grade school you might have learned a mnemonic like “my dear aunt sally” (multiplication, division, addition, subtraction) to remember the precedence of arithmetic operators.
Ruby and other programming languages have a slightly different rule: multiplication and division have the same precedence, as do addition and subtraction. Expressions are evaluated left to right, first evaluating multiplications and division, then again to do additions and subtractions. Evaluate the following expressions with IRB to see how Ruby applies these precedence rules:

\[
\begin{align*}
6 & \div 3 \times 4 \\
8 & \times 3 \div 4 \\
5 & - 4 + 2
\end{align*}
\]

Would any of these expressions have a different value using the “dear aunt sally” rules?

8. Explain how Ruby will evaluate \(3 \times 4 + 5 \times 2\). Use IRB to evaluate the expression. Did you get the result you expected?

9. What happens if you accidently hit the return key before you are done entering a complete expression? Try typing \(3 +\) and then the return key. What do you see?

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 reason nothing showed up on your screen is that Ruby is just waiting for you to complete the expression on the next line. Note that the prompt has changed: instead of \(\gg\) you should see \(--\). This means Ruby is in the middle of reading an arithmetic expression. To complete the expression, just type \(5\) and hit the return key. Did Ruby evaluate \(3 + 5\)?

The ability to break complex expressions into smaller lines is not something we will need to use very often in this book, but it’s worth knowing in case you ever find yourself in the middle of an IRB session and the prompt has changed from \(\gg\) to \(--\). If you get into this situation by mistake, type a number to complete the expression, or if that doesn’t work, type \(^C\) to interrupt IRB and get it back to the normal state (in computer manuals the notation \(^C\) is shorthand for “hold down the control key while hitting the C key”).

### 2.2 Numbers

All of the expressions in the previous section involved integers, i.e. whole numbers with no fractional parts. We can write expressions that use real numbers as well. The general rule is that if you want Ruby to treat a number as a real number include a decimal point when you write the number. Thus to distinguish between the integer \(5\) and the real number \(5.0\) you have to type \(5.0\) in expressions you enter into IRB.

To be more precise, in a programming language like Ruby \(5.0\) is a floating point number, not a real number. Real numbers are bothersome things like \(1/3\) and \(\sqrt{2}\) that have an infinite number of digits. Because numbers are stored in a finite amount of space inside a computer, we have to use an approximate value. Floating point numbers are approximations of real numbers, usually accurate to around 12 digits.

The distinction between integers and floating point numbers can be important. Consider what happens if we ask Ruby to divide \(10\) by \(3\). If we have Ruby do the operation using integers this is what we get:

\[
\begin{align*}
\gg & \ 10 \div 3 \\
\Rightarrow & \ 3
\end{align*}
\]

Ruby printed \(3\) because the two operands are integers, and the result is the largest integer
Floating Point Numbers

The term “floating point” comes from the fact that the internal representation of these numbers is based on scientific notation. The number 141.42 is stored internally as something like $1.4142 \times 10^2$.

This format makes it easier to represent very large numbers ($1.4142 \times 10^{23}$) or very small numbers ($1.1412 \times 10^{-10}$) by letting the decimal point “float” to the left or right.

below the actual result of 3.3333\ldots We can do the same calculation using floating point numbers:

```
>> 10.0 / 3.0
=> 3.33333333333333
```

Note that Ruby does not round off to the nearest integer, it truncates, as shown in this example:

```
>> 5.0 / 3.0
=> 1.66666666666667
>> 5 / 3
=> 1
```

The equation you learned in school for converting Fahrenheit to Celsius is often written as $C = (F - 32) \times \frac{5}{9}$. We can use Ruby to help us convert 80°F to Celsius:

```
>> (80 - 32) * 5 / 9
=> 26
```

Note that all the numbers in this example are integers, and that Ruby prints an integer result. A more accurate result (which you would get if you used a temperature conversion program on the Internet or if you retype the expression using floating point values) is 26.67°C.

It may seem at this point like Ruby is being rather “pedantic” in the way it forces us to write 80.0 instead of 80 if we want a more accurate conversion. Why can’t it simply recognize that in an expression like $(80 - 32) \times \frac{5}{9}$ we want the result to be 26.67 and not simply 26? The answer is that in general Ruby cannot simply figure out what we are looking for and give us the result in a format we might prefer. There may be other situations where the integer value is in fact the correct value, even if it is truncated and not rounded. We will see many examples of algorithms throughout the book that rely on the fact that operands and results are all integers. So the bottom line is that we need to be aware of the fact that algorithms rely on two basic types of numbers, integers and floating point, and be careful to specify the correct type of operands when we enter expressions.
Tutorial Project

10. Use IRB to evaluate a simple expression such as 3 * 5 where both operands are integers.
11. Repeat the previous expression, but replace the integer operands with floating point numbers. Can you see the difference between the outputs for these two expressions?
12. Use Ruby to convert the following temperature values to Celsius: 100°F, 90°F, 70°F, 212°F, 32°F.
13. The formula for converting from Celsius to Fahrenheit is \( F = C \times \frac{9}{5} + 32 \). Use this formula to convert the following temperatures to Fahrenheit: 0°C, 10°C, 20°C, 30°C, 100°C.

You now have enough experience with numbers in Ruby to complete the tutorial project for this chapter. The remaining problems, which explore expressions involving numbers in a little more depth, are recommended if you will be doing any of the more advanced projects on your own.

The first question: What happens if you try to mix integers and floating point numbers in an expression?

14. Ask Ruby to evaluate 3.0 * 5 and 3 * 5.0. Do you get an error message? If not, is the result an integer or a floating point value?
15. Try a few more expressions that mix integers and floating point, e.g. (80 - 32) * 5 / 9 or (80 - 32) * 5.0 / 9.

What Ruby did for these cases turns out to be a general rule: if Ruby is asked to apply an operation where one of the operands is an integer value, the integer is “promoted” to floating point and then the operation is applied, generating a floating point result.

The next set of questions explore what can happen if we rearrange the order of evaluation in the temperature conversion equation.

16. One way of describing the method for converting Fahrenheit to Celsius is “subtract 32 and multiply by \( \frac{5}{9} \).” If we follow this prescription exactly, the equation for converting 80°F is (80 - 32) * (5 / 9). What happens if you type that expression into IRB? Can you explain what went wrong?
17. Whenever you find yourself in a situation where Ruby is not evaluating a complicated expression the way you think it should, the best way to figure out what is happening is to break the complicated expression into smaller parts. The two parts of this expression are the subexpressions inside parentheses. What does Ruby produce for 80 - 32? How about 5 / 9? Does this help explain how Ruby evaluated (80 - 32) * (5 / 9)?
18. We can fix this problem by using floating point values instead of integers. What is the result of 5.0 / 9.0? What is (80 - 32) * (5.0 / 9.0)?

2.3 Variables

Suppose you need to calculate the area of a countertop like the one shown in Figure 2.2(a). The counter is a square with one corner missing, and you know that the sides of the missing triangular piece are half as big as the edge of the square. Figure 2.2(b) shows one strategy for computing the area: since we know the formulas for the area of a square and the area of a right triangle it’s easy to compute the size of the countertop by subtracting the area of the missing triangle from the area of the square. The formula for the area is \( x^2 - (x/2)^2 / 2 \) where \( x \) is the length of one edge of the square.

© 2008 John S. Conery. Please do not distribute without permission.
Now suppose you measure one of the edges of the square and find it is 109 cm. If you have your laptop open on the counter you can use IRB to calculate the area:

```ruby
>> 109 * 109 - (((109 / 2) * (109 / 2)) / 2
=> 10423
```

The number 109 appears in several places in this expression. We can simplify the expression a bit by using a variable name instead of the number. Creating a variable in Ruby is simple: just pick a name for the variable, and write the name on the left side of what is known as an assignment statement. A good choice for the name of our variable is `x` since that's the symbol that was used in the equation above to derive the formula for the area. So in IRB we just type

```ruby
>> x = 109
=> 109
>> x * x - (((x / 2) * (x / 2)) / 2
=> 10423
```

We can actually simplify the expression a bit further. Ruby has an exponentiation operator, written as `**` (two asterisks adjacent to each other). So to ask for the value of $x^2$ we simply write `x**2`:

```ruby
>> x**2 - (((x / 2)**2) / 2
=> 10423
```

Although this is a trivial example, it illustrates an important idea: it is almost always preferable to use the simplest form of an expression. We often make mistakes when typing numbers – it would be very easy to type 190 instead of 109 – and when a number is repeated four times in an equation we are four times as likely to make that mistake. We will see examples of complex situations that can be simplified in other projects throughout the book,
and the general advice is to put a little thought into simplifying equations and expressions. In many cases the simplifications will derive from insights we gain after experimenting with different versions or early prototypes.

Note that it is also possible to make a mistake when typing the name of a variable. The x key is right next to the z key and you might type ztt2 when you mean xtt2. But here Ruby gives a little bit of help – if there is no variable named z you would get an error message saying z is undefined.

There is a second advantage to using the expression that contains a variable name. Suppose, after computing the area the first time, you double check your measurements and find the edge is actually 107 cm. All you have to do to recompute the area is change the value of the variable and then re-evaluate the expression:

```ruby
>> x = 107
=> 107
>> x**2 - ((x / 2)**2) / 2
=> 10045
```

The first expression above updates the value of x – the old value is thrown away and the new value replaces it. The second expression is identical to the one typed earlier. To re-evaluate the expression you could either cut and paste the previous expression, or, if IRB on your system allows it, you might also be able to use keyboard keys to back up to a previous expression to evaluate it again.

In most projects that use IRB you will find yourself creating several variables. In order to keep everything straight it’s important to choose meaningful names. Fortunately, Ruby and other programming languages allow us to use more than a single letter in a name, so we could have chosen edge or even square_side for our variable. The rules for naming variables in Ruby are:

- Names must start with a letter.
- The remainder of the name can have a mix of upper and lower case, digits, or an underscore. Examples are squareSide, square_side, or sq123.
- Case is important: a and A are two different names in Ruby.
- Variable names must begin with a lower case letter. We will see names that begin with upper case letters later in the book, but the rule for now is when you are choosing a name for a variable choose one that begins with a lower case letter.

So far all the expressions in this chapter have been arithmetic expressions based on numeric values. Ruby also lets us write expressions that use strings as values. Strings can be used to represent names, addresses, product descriptions, and a wide variety of other useful pieces of data. To write a string in Ruby, simply enclose the characters you want to include in the string in double quotes:

```ruby
>> s = "hello"
=> "hello"
```

We won’t be using strings in the temperature conversion project, so we will put off the discussion of what we can do with strings until we need them in other projects. But introducing strings here will give us a better idea of what variables are and how Ruby evaluates expressions that contain variable names.
A useful way to visualize how Ruby manages variables is to imagine a large storage area, initially empty. When you define a variable, Ruby allocates space for the value and then associates the variable's name with this value. Figure 2.3(a) shows what the storage area might look like during the IRB session after defining the variables $x$ and $s$ in the previous examples.

The values of the variables are shown inside “clouds” to emphasize that how a value is represented inside the computer is, at this point, not that important. Usually all we need to know is that we've asked Ruby to create two values, one a number and the other a string, and to attach names to these values. How Ruby chooses to represent these values inside the computer’s memory is not important for this project, so we show the values as abstract ideas inside clouds.

Ruby is one of a group of programming languages known as *object-oriented languages*. One of the distinguishing features of this kind of language is that variables are just names attached to *objects* (the clouds shown in Figure 2.3). Programmers often refer to the memory that holds data items as object storage, or in a abbreviated form, the *object store*.

We saw earlier that the value of a variable can change – we updated the value of the variable $x$ by associating it with a new number. It turns out we can also change the type of value associated with a variable. Ruby doesn’t complain if, after defining $s$ to be a string, we then type something like

$$s = 9 \times 5$$

Ruby is perfectly happy to throw away the string object that used to be the value of $s$ and create a new object representing the number 45 and associate $s$ with this object (Figure 2.3(b)).

There are several important consequences of this flexible and dynamic behavior concerning variable names, and we will come back to this topic in future projects. For now the important concept to learn is that variables are simply names of objects, and when we use a variable in an expression Ruby looks up the value of the variable and uses that value in its calculations.
2.4 Methods

Tutorial Project

19. Start IRB, and type two expressions that define variables x and y:
   >> x = 6
   => 6
   >> y = 5
   => 5

20. Try out a few expressions using these variables:
   >> x + 3
   >> x * y
   >> (x + 3) * y
   Did you get what you expected?

21. Change the value of x:
   >> x = 2
   => 2
   Go back and re-evaluate the previous three expressions. Did the values of these expressions change after you changed the value of x?

22. Next define a variable with a string value:
   >> s = "hello"
   => "hello"

23. What do you think will happen if you try to add 3 to s?
   >> s + 3

24. You should have gotten a rather cryptic looking error message for that last expression. The word TypeError is a hint that Ruby did not know how to add a string and a number, since they are two different types of things. Assign s a new, numeric, value:
   >> s = 7
   => 7
   Now ask Ruby to evaluate s + 3 again. What happened this time?

From these simple experiments it’s tempting to conclude that Ruby has rules forbidding the mixing of types, since it didn’t allow us to add a number to a string. But if you try to “multiply” a string by a number (e.g. ask Ruby to evaluate "hello" * 3) you might be surprised by the result. The rules governing how Ruby handles expressions like this are in fact fairly straightforward, but we’ll postpone that topic until we have a project that mixes types of objects.

2.4 Methods

Most calculators have buttons for trigonometric functions, logarithms, and maybe a few other functions. We can use these functions in Ruby, too, but there a slight twist: before we use any of these functions we have to tell Ruby we want to use its “math library” by typing the following:

>> include Math

In this expression the word Math is the name of a module. A module is a collection of several related functions. Ruby organizes functions into modules because a name like log might be used in several different contexts: it might be the name of a mathematical function, or it might be the name of a function that writes error messages to a log file. To keep all these
names straight Ruby organizes them into modules and requires us to load the necessary
module before we invoke one of these functions.

After loading the Math module, we can use a function by typing an expression that con-
tains the name of the function followed by its arguments. So to compute the natural loga-
rithm of 10, for example, we would type this:

\[ \log(10) \]
\[ \Rightarrow 2.30258509299405 \]

In Ruby a function is known as a *method*. Other languages have their own terminology;
what Ruby calls a method is variously named a subroutine, function, or procedure in these
other languages. Whatever the terminology, they are all ways of implementing a very im-
portant idea: the ability to attach a name to a complex operation, and to use the name in
an expression whenever we want to perform the operation.

Here is some more terminology associated with methods:

- To ask Ruby to carry out the operations associated with a method we *call* the method.
  When Ruby evaluates the expression shown in the example above it calls the method
  named `log`.

- When a method is done it *returns* a value. The number 2.3\ldots printed in the example
  is the value returned by the `log` method.

- Values used by a method are called *parameters*, and we talk about *passing* parameters
to the method. The `log` method has one parameter, but in general methods can have
any number of parameters (including zero).

When calling a method the standard notation is to write parameters in parentheses fol-
lowing the method name. If the method takes more than one parameter the parameter
values are separated by commas. For example, Ruby’s `Math` module has a method named
`hypot` that takes two parameters, \( x \) and \( y \), and returns the hypotenuse of a right triangle
with side lengths \( x \) and \( y \) (i.e. it calculates \( \sqrt{x^2 + y^2} \)). To calculate the hypotenuse of a
triangle with sides of length 5 and 12 the call is

\[ \text{hypot}(5, 12) \]
\[ \Rightarrow 13.0 \]

The parentheses can be left out if the resulting expression is not ambiguous. Ruby un-
derstands what we want if the call to `hypot` is written without parentheses around the
parameters:

\[ \text{hypot} \ 5, \ 12 \]
\[ \Rightarrow 13.0 \]

Often the choice of whether or not to use parentheses is up to you. Most of the examples
in this book will use parentheses, mainly because most expressions are easier to understand
when the parentheses are included. However, many examples of Ruby code you’ll see in
other books or sample programs you find on the Internet will often leave out parentheses,
so to help you get used to this notation the expressions in this book will follow the most
common idiom.

This next examples illustrate one of the most important ideas to learn about method
calls: one can place a method call anywhere in an expression, and when Ruby evaluates
the expression, the value returned by the method is “plugged in” to the expression:
In the first example Ruby calls the `sqrt` method, which returns the number 5.0. Then that number is used to evaluate $3 \times 5.0$ and the result becomes the value of the variable `n`. In the second example, Ruby calls methods that implement two different trigonometric functions. The values returned by these methods are plugged in to the original expression; Ruby squares each number, adds the squares, and prints the result.

These two examples reinforce an idea introduced in the previous section. There we saw that when an expression contains a variable name Ruby looked up the value of the variable and plugged it into the expression in place of the variable. The same thing is going on here: when an expression contains a call to a method, Ruby invokes the method, and when the method returns a value Ruby plugs that value into the expression. We will see many more examples of expressions that contain method calls in the rest of this chapter and throughout the book. If you ever find yourself confused by what looks like a complex expression, break it down into its constituent parts, try to understand what the value of each part is, and then ask how Ruby would plug those values into the original expression.

### Exercises

25. Start by defining some variables to use in the expressions:
   ```ruby
   >> pi = 3.14159
   => 3.14159
   >> x = pi/4
   => 0.7853975
   >> y = sqrt(2)
   => 1.414213562373
   ```

26. Use IRB to evaluate the following expressions, using methods from the `Math` module where appropriate.
   - $\sin x$
   - $\cos x$
   - $\tan x$
   - $\sin x / \cos x$
   - $y^2$
   - $\log y$
   - $\log 10$
   - $\log 1$
   - $\log 0.01$
   - $\log 0.00001$
   - $\log 0$

The following expressions are more complicated; give them a try if you want an additional challenge, or think you might want to start using IRB as your calculator.

27. $\sin x^2$
    - $\sin^2 x$
    - $\sin^2 x + \cos^2 x$
2.5 The **celsius** Method

Imagine you are working on a project that requires you to do a lot of temperature conversions – maybe you are translating recipes from an American cookbook for European readers and need to change oven settings from Fahrenheit into Celsius. If you want to use Ruby to do the conversions it won’t take long before you get tired of typing a series of expressions that are just slightly different from one another:

```
>> (350 - 32) * 5 / 9
=> 176
>> (425 - 32) * 5 / 9
=> 218
>> (400 - 32) * 5 / 9
=> 204
...
```

Ruby can help make this task a little simpler by letting us define our own method. The definition only requires three lines of text:

```ruby
def celsius(f)
    return (f - 32.0) * 5.0 / 9.0
end
```

The first line has the word `def` – this is what alerts Ruby to the fact that we want to define a new method. Following `def` are the name of the method we’re defining and a list of the parameter names. The last line simply has the word `end` to indicate the end of the definition.

The parameter is the name of a variable that we can use in the expressions in the body of the method. In this example we’re telling Ruby we want to use a variable named `f`. When the method is called, the value passed as a parameter in the call is stored in the variable. For example, when we type `celsius(50)`, Ruby will assign `f` the value 50 before it evaluates the expression in the body.

In between the lines with `def` and `end` we can put as many Ruby expressions as we need. One of these expressions will become the value that should be returned as the value of the method call. The convention in Ruby is to precede the expression that creates the return value by the word `return`. For this simple example there is just one expression in the body of the method, but we’ll see more complex examples later.

After the new method is defined, we can call it just like we call any other method in Ruby:

```
>> celsius(212)
=> 100
>> hot = celsius(95)
=> 35
>> hot - 10
=> 25
```

The last two lines illustrate that our new method works like other methods in Ruby – we can pass it a value, it performs a calculation, and it returns a result that we can use like any other value, in this case to define a new variable named `hot`.

Besides showing how to define a method in Ruby, this example introduces some additional terminology. The words `def`, `return`, and `end` have a special meaning to Ruby. Words
2.5 The \texttt{celsius} Method

like these are known as \textit{keywords} or \textit{reserved words} in a programming language. They are reserved because we are not allowed to use them as the names of variables or new methods.

It is possible to define new methods interactively. For the simple example you can type the entire definition into IRB on a single line, or use three lines as shown above. Normally, however, people who write their own methods use a text editor to create a file that contains the method definition and then tell Ruby to read the definition from the file. The main advantage to this approach is that if you want to change the definition it’s much easier to edit the file and re-read it than it is to retype the entire definition in IRB.

To tell IRB to read and evaluate the text in a file use a method named \texttt{load}. If we put the definition of the example method in a file named \texttt{temps.rb} then this is what we would type in IRB:

\begin{verbatim}
>> load "temps.rb"
=> true
\end{verbatim}

Note that the argument to \texttt{load} is a string – the name of a file surrounded by double quotes – and that the result of calling the \texttt{load} method is \texttt{true} if IRB successfully reads and evaluates each expression in the file. The .\texttt{rb} at the end of the file name is a common way to indicate that the file is a plain text file containing a Ruby program, similar to the convention that text processing documents have names ending in .\texttt{doc} or PDF files have names ending with .\texttt{pdf}. Ruby doesn’t care whether or not the file name ends in .\texttt{rb}, but it’s a good idea to follow this convention so you can quickly tell which of your files contain Ruby programs.

\textbf{Tutorial Project}

28. Type the following three lines into IRB to create a new method named \texttt{celsius}:

\begin{verbatim}
>> def celsius(f)
   return (f - 32) * 5 / 9
end
\end{verbatim}

You can either type these as three separate lines, as shown above, hitting the return key at the end of each line, or you can type them all together on one line.

29. Try calling your new method:

\begin{verbatim}
>> celsius(50)
=> 10
>> celsius(95)
=> 35
\end{verbatim}

30. Exit from IRB (type \texttt{quit}) and then start a new IRB session. In this new session all the definitions you created previously will be gone. Verify this by calling the \texttt{celsius} method:

\begin{verbatim}
>> celsius(95)
NoMethodError: undefined method `celsius' for main:Object
\end{verbatim}

31. Create a new file named \texttt{temps.rb} on your computer and type the three lines that define the \texttt{celsius} method into the file. Save the file\footnote{You can use a document formatting program such as Microsoft Word or Open Office, but if you do make sure you use the “save text only” option when you save the file. The book web site has a discussion and recommendations for other text editors that work well with Ruby and other programming languages.} in your project directory.

32. Tell Ruby to load the definition of \texttt{celsius} from the file you just created:

\begin{verbatim}
>> load "temps.rb"
=> true
\end{verbatim}
If you get an error message instead of the value `true` make sure the file is in your project directory.

33. When you have successfully loaded the new definition you should be able to call it:
   ```ruby
   >> celsius(95)
   => 35
   ```

   In later projects we will be building up a method in stages, starting with a very simple version and then adding expressions when we are convinced the simpler version is working. Ruby supports this sort of “incremental development” by letting us update a method definition in a file and then, when we type the load expression again, overwriting the old definition with the new one.

34. Go back to your text editing program and modify the method so it uses floating point numbers. Change 32 to 32.0, 9 to 9.0, and 5 to 5.0. Save the file, go back to IRB, and load the new version:
   ```ruby
   >> load "temps.rb"
   => true
   ```

35. Test your new version:
   ```ruby
   >> celsius(100)
   => 37.7777777777778
   ```

   If the result is a floating point value (as shown above) then you successfully updated the version of the method used by IRB.

   The remaining exercises are optional, but they will help you gain a deeper understanding of the ideas illustrated in this section.

36. Did you notice that in the call to the new version of `celsius` the parameter was an integer, but the result was printed as a floating point number? Can you explain why? (Hint: what did you discover by doing one of the earlier optional exercises with mixed integer and floating point expressions?)

37. What do you suppose would happen if you pass a string value to `celsius`, e.g. if you ask Ruby to evaluate `celsius("hello")`?

38. Edit your file by adding a second method definition just below the first one. Your new method should have the name `fahrenheit`, and the body of the method should have a Ruby expression that implements the equation \( F = C \times \frac{9}{5} + 32 \). Reload the file and test your method by calling it with several different values.

## 2.6 Chapter Review

The main goal for this chapter was to introduce the Ruby programming language and the Interactive Ruby (IRB) environment that will be used for “computational experiments” in this book. The projects described in the remaining chapters will all have more complex experiments that will explore more interesting algorithms, but the project in this chapter was simply used to illustrate how IRB works by using it to implement a trivial program that converts temperatures from Fahrenheit to Celsius.

Numbers are the simplest type of data in Ruby. We saw that Ruby works with two types of numbers, integers and floating point numbers (which are finite approximations of real numbers). Expressions with numbers are based on the four basic operations in arithmetic.
For historical reasons (dating to the time when programs were created with keypunch machines that had a limited number of characters) Ruby uses an asterisk as the symbol for multiplication and a forward slash for division. We will see other arithmetic operators later when there are projects that need them.

Variables are symbolic names that can be used in expressions. Variables are created when we use them in assignment statements, which includes the name of a variable, an equal sign, and an expression that defines the value for the variable. After we define a variable, we can use its name as an operand in an expression. We can also change the value of a variable at any time simply by reusing it another assignment statement.

In addition to the standard arithmetic operations, Ruby knows how to perform more complicated calculations, such as finding the square root of a number. These more complex operations are known as “methods” in Ruby and other object-oriented programming languages. When Ruby sees the name of a method in an expression, it “calls” the method, which activates an algorithm. The algorithm produces a value that is then plugged in, or “returned,” to the original expression.

Finally, we saw that we can enclose a set of Ruby statements between the words `def` and `end` in order to define our own methods. The method that converts temperature values from Fahrenheit to Celsius requires only three lines of text:

```ruby
def celsius(f)
  return (f - 32) * 5 / 9
end
```

An example that uses this method is

```ruby
>> t = celsius(90)
=> 32
```

In this call to `celsius`, Ruby creates a variable named `f`, assigns it the value 90, evaluates the resulting expression, and returns the value of 32, which is then saved as the value of the variable `t`.

This example emphasizes the most important idea introduced in this chapter: Ruby is essentially just a system that carries out algebraic operations, continually rewriting equations until they are in the simplest form possible. To evaluate an expression, Ruby makes sure it has all the values it needs. If it sees the name of a variable, it looks up the current value of the variable and substitutes that value in the expression. When it sees the name of a method, it calls the method and then substitutes the value returned by the method into the original equation. In later chapters we will encounter several examples of Ruby expressions that, at first, may seem very complex, but they can all be understood if we remember that Ruby will evaluate these expressions by calling methods and substituting values as they are needed.