

Chapter 2

Ruby Lesson #1: Data Structures

Chapter Topics

- Ruby numbers, strings, and arrays
- Storing data with variables and constants
- Object-oriented design with the Ruby programming language

To access SketchUp in code, you need to speak its language: Ruby. The goal of this chapter is to present the rudiments of Ruby programming—just enough to get you comfortable with the language and its basic data structures. The presentation starts with a discussion of numbers and text, and proceeds to variables, constants, and arrays.

At each step, plenty of examples are provided so that you can enter and execute commands on your own. I recommend that you not only work alongside the text but also experiment with different commands. This will enhance your understanding of the Ruby programming language and make you more comfortable with the overall coding environment.

The last part of the chapter deals with object-oriented programming in Ruby, and discusses objects, classes, and methods. Once these topics are clear, you'll be ready for Chapter 3, which builds on this foundation to present the basics of SketchUp modeling. This is really where the fun starts, but you have to learn to walk before you can run.

2.1 The Ruby Console Window

If you haven't already, open the Window menu in SketchUp and select the Ruby Console option. The Ruby Console allows you to enter and execute commands one at a time. Later chapters will explain how to create scripts that store multiple Ruby statements, but for now we'll examine Ruby commands individually.

The Ruby Console is simple to use: enter commands in the text box and press Enter. The results will be displayed in the console that makes up the upper portion of the dialog. To see how this works, enter the following command:

```
2 + 2
```

Now press Enter. In the console, SketchUp displays the command and its output: 4. This is shown in Figure 2.1.

`2 + 2` is a valid Ruby command, as are similar arithmetic expressions. Numeric expressions are the simplest way to start learning Ruby, and the next section will explain them in greater detail.

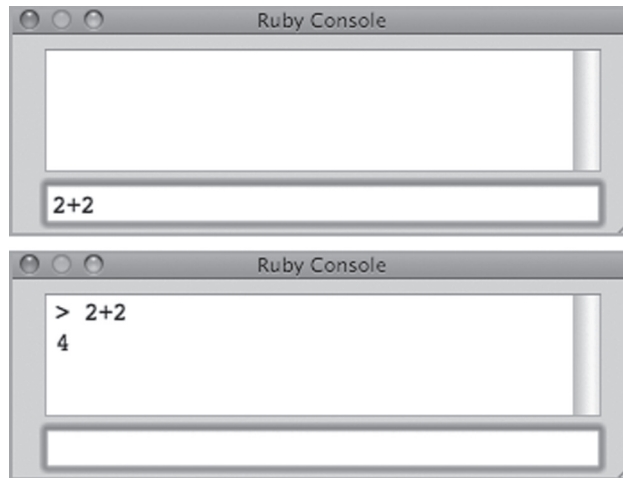


Figure 2.1: Ruby Console Input and Output

2.2 Numbers and Numeric Operators

When you create SketchUp models in Ruby, one of the most common tasks involves defining points that demarcate lines and surfaces. Each point is composed of three numeric coordinates, so it's fundamentally important to understand how Ruby handles numbers. This section discusses number formats, operators, and the order of operations.

Integers and Floating-Point Numbers

In this book, we'll be dealing with two types of numbers: integers and floating-point values. An integer represents a whole number and has no decimal point. A "+" sign preceding the number indicates a positive value and a "-" indicates a negative value. If no sign is given, the value is assumed to be positive.

Just as large numbers are normally broken up with commas, Ruby allows you to partition large numbers with underscores. For example, the number 1,000,000 can be expressed as 1000000 or 1_000_000. To test this, enter the following command at the command line:

```
5_000 / 4
```

The result is 1250, just as if you'd entered `5000 / 4`.

Floating-point numbers have decimal points that separate the number's integer part from its fractional part. In Ruby, each floating-point number must have at least one digit before and after the decimal point. That is, you can express $\frac{1}{2}$ as `0.5` or `0.500`, but never as `.5`.

A floating-point value can be preceded by `+` or `-`, and can be followed by `e` to define an exponent. The following floating-point numbers are valid: `-25.4`, `1.4959e11`, `123_456.789_012`, and `3.14159`.

Ruby provides for other numeric types, including complex numbers and rational numbers. But for the purposes of this book, you'll only need integers and floating-point values.

Arithmetic Operators

Ruby recognizes all the common arithmetic operators used in C and other programming languages: `+`, `-`, `*`, and `/`. These are listed in Table 2.1, along with the modulo and exponent operators.

Table 2.1

Ruby Arithmetic Operators

Operator	Purpose	Integer Example	Floating-Point Example
<code>+</code>	Addition	<code>4 + 5 = 9</code>	<code>4.0 + 5.0 = 9.0</code>
<code>-</code>	Subtraction	<code>12 - 4 = 8</code>	<code>12.0 - 4.0 = 8.0</code>
<code>*</code>	Multiplication	<code>7 * 3 = 21</code>	<code>7.0 * 3.0 = 21.0</code>
<code>/</code>	Division	<code>20 / 8 = 2</code>	<code>20.0 / 8.0 = 2.5</code>
<code>%</code>	Modulo	<code>20 % 8 = 4</code>	<code>20.0 % 8.0 = 4.0</code>
<code>**</code>	Exponent	<code>3 ** 2 = 9</code>	<code>3.0 ** 2.0 = 9.0</code>

It's important to see how the type of the result (integer or floating-point) is determined by the numerical inputs, called *operands*. If one integer is subtracted from another, the result will always be an integer, and the same goes for addition, subtraction, and division.

If one operand is an integer and the other is a floating-point value, the result will always be a floating-point value. This is demonstrated in the following examples:

- `90 - 82` returns 8

- `90.0 - 82` returns `8.0`
- `3 * 4` returns `12`
- `3 * 4.0` returns `12.0`
- `4 / 3` returns `1`
- `4 / 3.0` returns `1.3333333333333333`
- `3 / 4` returns `0`
- `3 / 4.0` returns `0.75`

The last four of these expressions make use of the division operator, and the results may not be obvious at first. If one of the operands is floating-point, the result is the regular floating-point quotient. But if both operands are integers, such as in `a / b`, the remainder is discarded and only the integer result is returned.

To obtain the remainder of integer division, you need the fifth operator in Table 2.1: the modulo operator or `%`. The best way to understand this is by example. If 17 is divided by 5, the result is 3 with a remainder of 2. In Ruby, this means `17 / 5 = 3` and `17 % 5 = 2`. If `a` divides evenly into `b`, the result of `b % a` will always be zero. Therefore, the following expressions should make sense:

- `16 / 8` returns `2`
- `16 / 8.0` returns `2.0`
- `16 % 8` returns `0`

The last operator in Table 2.1, `**`, performs exponentiation, and `a ** b` is the same operation as a^b . The second operand is the exponent, and defines how many times the first operand should be multiplied by itself. For example, `2 ** 3` returns 8 because 2^3 equals 8.

The exponent doesn't have to be an integer, and you can compute square roots by setting it equal to $\frac{1}{2}$ and cube roots by setting the exponent to $\frac{1}{3}$. Similarly, if the exponent is negative, the result is the multiplicative inverse ($1/x$) of the operation with a positive exponent. For example, `2.0 ** -3 = 1/(2.0 ** 3) = 1/8`.

The following examples show further uses of the `**` operator:

- `4 ** 2` returns `16`
- `4 ** -2` returns `0.0625`

- `4 ** 0.5` returns `2.0`
- `4 ** 0` returns `1`
- `4 ** (1/2)` returns `1` (the exponent evaluates to 0)

Operating on numeric values is a crucial task in many SketchUp designs. As you write code, keep in mind that integer operations generally only return integers. To obtain a floating-point result, one of the operands needs to be floating-point.

Order of Operations

If a Ruby command contains multiple numeric operators, the operations aren't necessarily performed from left to right. The following rules must be applied in order:

1. Perform all operations surrounded by parentheses from left to right.
2. Perform all exponent operations from left to right.
3. Perform all multiplication and division operations from left to right.
4. Perform all addition and subtraction operations from left to right.

For example, look at the following command:

```
1 + 3 * (6 - 4) ** 3 / (1 + 3)
```

Ruby will compute the operations in parentheses first: $(6 - 4 = 2)$ and $(1 + 3 = 4)$. Next, it will perform the exponentiation ($2^3 = 8$), and the equation simplifies to:

```
1 + 3 * 8 / 4
```

Lastly, Ruby performs the multiplication ($3 * 8 = 24$) and the division ($24 / 4 = 6$). The final answer is computed as $1 + 6 = 7$.

SketchUp Numeric Conversion

Before we leave the subject of numbers, it's important to mention the operators that SketchUp provides in addition to those described previously. These are listed in Table 2.2.

Table 2.2

SketchUp Conversion Operators

Operator	Purpose	Example
cm	Convert centimeters to inches	2.54.cm = 1
degrees	Convert degrees to radians	180.degrees = 3.14159265358979
feet	Convert feet to inches	1.feet = 12.0
inch	Convert inches to length	--
km	Convert kilometers to inches	1.km = 39370.0787401575
m	Convert meters to inches	1.m = 39.3700787401575
mile	Convert miles to inches	1.mile = 63360.0
mm	Convert millimeters to inches	1.mm = 0.0393700787401575
radians	Convert radians to degrees	3.14159265358979.radians = 180
to_cm	Convert inches to centimeters	0.393700787401575.to_cm = 1
to_feet	Convert inches to feet	12.to_feet = 1.0
to_inch	Convert length to inches	--
to_km	Convert inches to kilometers	39370.0787401575.to_km = 1
to_l	Convert inches to length	--
to_m	Convert inches to meters	39.3700787401575.to_m = 1
to_mile	Convert inches to miles	63360.to_mile = 1.0
to_mm	Convert inches to millimeters	0.0393700787401575.to_mm = 1
to_yard	Convert inches to yards	36.to_yard = 1.0
yard	Convert yards to inches	1.yard = 36.0

Internally, SketchUp stores length values in inches, even if you choose a template based on the metric system. For this reason, most of the conversion utilities in Table 2.2 either convert from inches or convert to inches. These operators are different than those in Table 2.1 in that they are accessed by placing a dot (.) after the number. The third column shows how these operators are used in practice.

For example, the following command converts a length of 72 inches to meters:

```
72.to_m
```

The result is 1.8288 because 1.8288 meters has the same length as 72 inches.

At the time of this writing, SketchUp can't draw or store lengths less than 0.001 inches. This means you can't set any dimension less than 0.001 inches.

This book usually doesn't specify dimensions, but there is one important point to keep in mind. All of the SketchUp routines that deal with angles require angular values to be given in *radians*, not degrees. But because it's simpler to deal with integers, this book starts with degrees and converts the angle to radians. For example, to convert 30° to radians, you'd use the following command:

```
30.degrees
```

This may cause confusion since we're converting *from* degrees instead of *to* degrees. However, angular measurements will be used frequently in this book, and eventually, the usage of the `degrees` operator will become second nature.

2.3 Strings

Coding with numbers is important, but there are many occasions when you'll need to work with text. Text operations become necessary when you read characters from a file, define labels for a SketchUp design, or add tooltips to a new SketchUp tool. In my scripts, I frequently use text operations to display messages during the course of a script's execution.

In many programming languages, single characters are treated differently than groups of characters. For example, in Java, 'a' is a `char` and "abcd" is a `String`. Ruby doesn't make this distinction: both 'a' and "abcd" are `Strings`. A `String` contains one or more characters, including letters, numbers, punctuation, and special characters.

In Ruby, a `String` can be enclosed in single quotes or double quotes. If a `String` is enclosed in double quotes, the Ruby interpreter recognizes escape sequences (e.g., `\t` for tab, `\n` for newline), and displays them accordingly. If the `String` is enclosed in single quotes, escape sequences are ignored. Therefore, "Line1\nLine2" will be printed on two lines because of the

`\n` escape character. However, `'Line1\nLine2'` will be printed on one line because the escape character is ignored.

Basic String Operations

Ruby provides a number of ways to manipulate `Strings` in code. Two of the most common operators are `+` and `*`, and their roles are easy to understand. The `+` operator joins `Strings` together, as shown in the following command:

```
"Hello," + " world"
→ Hello, world
```

The multiplication operator, `*`, repeats a `String` a given number of times, as shown in the following command:

```
"Hello!" * 3
→ Hello!Hello!Hello!
```

Substrings and Ranges

A common task is to access characters in a `String` according to their positions. The position of a character is called its *index*. Within a `String`, index values run from zero to the length of the `String` minus one.

A set of adjacent characters in a `String` is called a *substring*, and you can access a substring by specifying a `Range` of index values. A `Range` represents a sequence of values and can be defined in one of two ways. A `Range` of the first type is specified by `start..end`, and represents the interval from `start` to `end`, including `end`. A `Range` of the second type is specified by `start...end`, and represents the interval from `start` to `end`, not including `end`. The following examples make this clear:

- `0..4` represents the interval `[0, 1, 2, 3]`
- `0...4` represents the interval `[0, 1, 2, 3, 4]`
- `-5..-3` represents the interval `[-5, -4, -3]`

- 'a'...'e' represents the interval ['a', 'b', 'c', 'd']
- 'a'..'e' represents the interval ['a', 'b', 'c', 'd', 'e']

To obtain a substring, follow the `String` by the `Range` of desired index values enclosed within brackets. Try these commands in the Ruby Console:

```
"HelloWorld"[0..2]
  → Hel
"HelloWorld"[0...2]
  → He
"HelloWorld"[1..4]
  → ello
"HelloWorld"[1...4]
  → ell
```

If an index is positive, the character's location is determined from the left of the `String`. If the index is negative, the location is determined from the right. This is shown in Figure 2.2.

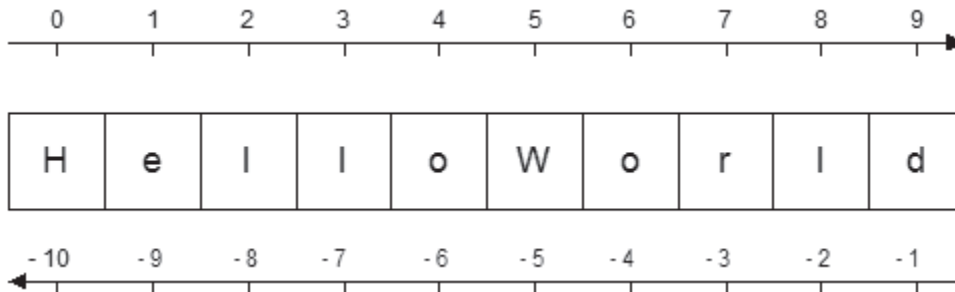


Figure 2.2: Positive and Negative Index Values within a String

The following commands demonstrate how negative indices are used to obtain substrings:

```
"HelloWorld"[-10..-6]
  → Hello
```

```
"HelloWorld"[-3...-1]
→ rl
```

In addition to retrieving a substring with a `Range`, you can also obtain a substring using the index of the first character and the length of the substring. The format is `[index, length]` and the following examples show how this is used in code:

```
"HelloWorld"[3, 4]
→ loWo
```

```
"HelloWorld"[0, 5]
→ Hello
```

```
"HelloWorld"[-5, 5]
→ World
```

Advanced String Operations

In addition to the operations presented thus far, you can access many others by following the `String` with a dot and the name of a *method*. Methods will be explained later in this chapter, but for now, you can think of a method as an operation with a specific name. For example, `length` and `size` are two methods that return the number of characters in a `String`. The following examples show how these two methods are accessed in code:

```
"HelloWorld".length
→ 10
```

```
"HelloWorld".size
→ 10
```

Table 2.3 lists `length`, `size`, and a number of other `String` methods with descriptions of their purpose and examples of their usage. More methods of the `String` class and other fundamental Ruby classes can be found at the Ruby Documentation Site at <http://www.ruby-doc.org/core/classes/String.html>.

Table 2.3*String Operations*

Operation	Purpose	Example
<code>downcase</code>	Change all letters to lower-case	<code>"Hello".downcase → "hello"</code>
<code>hex</code>	Convert a hexadecimal expression to a number	<code>"0x42".hex → 66</code>
<code>include?</code>	Identifies whether the String contains the given expression	<code>"hello".include? "ell" → true</code>
<code>index</code>	Returns the index of the first occurrence of the given String	<code>"Hello".index("e") → 2</code>
<code>length</code>	Returns the number of characters in the String	<code>"Hello".length → 5</code>
<code>lstrip</code>	Removes leading whitespace from the String	<code>" Hello ".lstrip → "Hello "</code>
<code>replace</code>	Replaces the String with another String	<code>"Hello!".replace("Hola!") → "Hola!"</code>
<code>reverse</code>	Reverses the characters in the String	<code>"Hello".reverse → "olleH"</code>
<code>rindex</code>	Returns the index of the last occurrence of the given String	<code>"Hello".rindex("l") → 3</code>
<code>size</code>	Returns the number of characters in the String	<code>"Hello".size → 5</code>
<code>split</code>	Splits the String into substrings according to a character, returns the array of substrings	<code>"Hello,world,again".split(",") → ["Hello","world","again"]</code>
<code>strip</code>	Removes leading and trailing whitespace from the String	<code>" Hello ".strip → "Hello"</code>
<code>to_f</code>	If possible, converts the String to a Float and returns the Float	<code>"-1.3e10".to_f → -13000000000.0</code>
<code>to_i</code>	If possible, converts the String to a Fixnum and returns the Fixnum	<code>"2000".to_i → 2000</code>
<code>tr</code>	Replaces specified characters in String with other specified characters	<code>"Hello".tr("le","ma") → "Hammo"</code>
<code>upcase</code>	Change all letters to upper-case	<code>"Hello".upcase → "HELLO"</code>

Printing Strings

Ruby provides three commands that display Strings in the console: `puts`, `print`, and `printf`. In this book, the example code relies primarily on `puts`, which displays a String followed by a newline character. The following example shows how `puts` is used:

```
puts "Number of characters in Hello: " + "Hello".length.to_s
→ Number of characters in Hello: 5
```

The `to_s` method converts the number returned by `"Hello".length` to a String. Ruby doesn't always convert numbers to Strings, so this must be done in code.

The `print` command performs the same operation as `puts`, but doesn't place a newline after the String. The `printf` command works like `print`, but allows you to format the String using special formatting characters. `printf` formatting is an involved subject, and I won't discuss it here in depth. But the following examples should give you an idea how it's used:

```
printf "The length of %s is %d\n", "Hello", "Hello".length
→ The length of Hello is 5
printf "The last index of b in bubble is %d\n", "bubble".rindex("b")
→ The last index of b in bubble is 3
```

Ruby's `printf` command works exactly like the common `printf` command in C. A brief web search will show you all the different formatting commands and options available.

2.4 Variables and Constants

We've dealt with bare numbers and text so far, but in the real world, we assign names to data and operate on the names instead of the raw data. In Ruby, named data comes in two categories: variables and constants. This section will explain how they're used and the differences between them.

Variables

In SketchUp scripts, it's more convenient to work with names instead of numbers. For example, if you want to change a door's height from 86 inches to 94 inches, you don't want to change each occurrence of "86" to "94". Instead, it's easier to use a name such as `door_height`. Now you can change all of the height values easily: set `door_height` to 94 in one line and this value will hold for all future occurrences.

Let's see how this works in SketchUp. In the Ruby Console, assign the variable `x` to a value of 2 with the following command:

```
x = 2
```

When you do this, SketchUp sets aside a portion of memory for the variable `x` and places the value of 2 in the allocated memory. Now you can operate on this variable as if it was a regular number. For example,

- `x + 5` returns 7
- `x * 2` returns 4
- `x ** 3` returns 8

In these operations, `x` keeps its value after each command. To change the value of a variable, you can perform operations such as the following:

- `x = x + 1`
- `x = x - 3`
- `x = x * 7`
- `x = x / 9`

You can accomplish the same results with Ruby's shorthand operators:

- `x += 1`
- `x -= 3`
- `x *= 7`
- `x /= 9`

These operations are all integer-based, but if you prefer, you can set `x` equal to a floating-point value or a `String`. For example, the following commands create a variable containing "Hello" and use the `String` addition operator, `+`, to append another `String`:

```
str = "Hello"
    → Hello
str += ", world!"
    → Hello, world!
```

The variable names `x` and `door_height` share an important characteristic: both start with a lower-case letter. In Ruby, a variable may start with the underscore or any lower-case letter, but not an upper-case letter. If data is assigned to a name with an upper-case letter, Ruby interprets it as a constant, which is explained next.

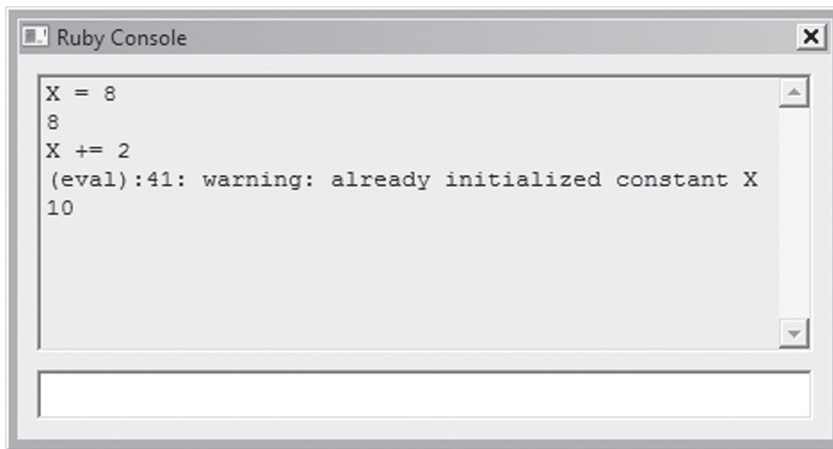
Constants

There are many instances where you'll deal with values that shouldn't be changed. For example, π will always equal approximately 3.14159 and there will always be 2.54 centimeters to an inch. In these cases, using a variable isn't a good idea because its value might be changed during a script's execution.

For this reason, Ruby provides *constants*, which operate like variables and can be assigned to the same types of values. But if a constant's value is reassigned, Ruby produces a warning telling you that its value has been changed. To see how this works, enter the following two commands in the console window:

```
X = 8
X += 2
```

After these commands are executed, Figure 2.3 shows the resulting message in the Ruby Console: "already initialized constant X".

A screenshot of a Ruby Console window. The window title is "Ruby Console". The text inside the console is as follows:

```
X = 8
8
X += 2
(eval):41: warning: already initialized constant X
10
```

The console shows the assignment of 8 to X, the output of 8, the attempt to increment X by 2, a warning message, and the final output of 10.

Figure 2.3: Updating a Constant's Value

Despite the warning, the second command changes the value of `X` from 8 to 10, and you can verify this with further commands.

If you repeat these commands with `x` instead of `X`, no warning will appear. This is because Ruby uses the first letter of the data structure to distinguish constants from variables. If the first letter is upper-case, Ruby treats it as a constant. If it's lower-case, Ruby considers it a variable.

2.5 Arrays

Every point, line, and shape in a SketchUp design must be positioned with `x`, `y`, and `z` coordinates. Rather than manage coordinates as individual numbers, it's easier to place them in collections called *arrays*. An array is a data structure that contains an ordered sequence of values called *elements*. Arrays are similar to the *Strings* we looked at earlier, but while a *String* is composed of characters, an array can contain anything, including numbers, *Strings*, variables, constants, and even other arrays.

Just as *Strings* are surrounded with single quotes or double quotes, arrays are surrounded by square brackets. For example, the following command creates a seven-element array:

```
arr = [1, 2, "ab", 4.0, 'Hello', 6.0, [1, 2, 3]]
```

This creates an array called `arr` whose elements are `1, 2, "ab", 4.0, 'Hello',` and `[1, 2, 3]`.

Accessing Array Elements

Each element is accessed according to its position inside the array, starting from position 0. An element's position is referred to as its *index*. The following command accesses the element of `arr` whose index equals 2:

```
x = arr[2]
```

The following command sets the value of the fourth element, whose index equals 3:

```
arr[3] = 12
```

Array element indices follow the same rules as character indices in `Strings`. Index 0 represents the first element, index 1 represents the second element, and index 2 represents the third element. Negative indices access elements from the end of the array. That is, an index of `-1` returns the last element of the array, `-2` returns the second-to-last element of the array, and so on.

As with `Strings`, you can access multiple array elements by defining a `Range` of indices. This can be done by placing two or three dots between the start and end values. The following example commands access elements in the `arr` array defined earlier:

```
arr[2..5]
  → ["ab", 4.0, "Hello", 6.0]
arr[0...3]
  → [1, 2, "ab"]
arr[-6..-4]
  → [2, "ab", 4.0]
```

Alternatively, you can set a starting index and identify how many further elements should be in the subarray. The following command forms a subarray with four elements starting with the element at the index 2:

```
x = arr[2, 4]
  → ["ab", 4.0, "Hello", 6.0]
```

This command sets `x` equal to an array containing elements "ab", 4.0, "Hello", and 6.0. Notice the difference between `a[2..4]` and `a[2, 4]`. Keep this in mind if you encounter any Range-related errors.

Basic Array Operations

Ruby provides a number of different ways to manipulate arrays, and many of them are exactly similar to the `String` operations discussed earlier. Table 2.4 lists six array operators with descriptions and examples of their usage.

Table 2.4

Ruby Array Operators

Operator	Description	Example
+	Combine two arrays into a larger array	<code>[6, 7] + ["aa", "bb", "cc"] → [6, 7, "aa", "bb", "cc"]</code>
-	Remove the elements of the second array from the first	<code>[1, 2, 3, 4] - [1, 2] → [3, 4]</code>
*	Repeat elements of an array	<code>[a, b] * 3 → [a, b, a, b, a, b]</code>
<<	Append an element to the end of an array	<code>[x, y, 12] << 13 → [x, y, 12, 13]</code>
	Combine arrays without duplicates	<code>[1, 2, 3] [2, 3, 4] → [1, 2, 3, 4]</code>
&	Combine only the duplicate elements	<code>[1, 2, 3] & [2, 3, 4] → [2, 3]</code>

In the third column, array elements include textual names contained within quotes ("aa" and "bb") and letters contained without quotes (`x` and `y`). As discussed earlier, names without quotes are used to identify variables and constants, and must be initialized before they're inserted into an array.

The third operator, `*`, is particularly helpful when you want to initialize every element of an array with the same value. For example, the following command fills `zero_array` with twelve zeros:

```
zero_array = [0] * 12
```

The `<<` operator adds an element to the end of an array. If the second argument is another array, that array will become a single element of the first array. For example, the command

```
[1, 2, 3] << [4, 5, 6]
```

returns `[1, 2, 3, [4, 5, 6]]`, not `[1, 2, 3, 4, 5, 6]`. If you want to append the elements of the second array, use the `concat` method described below.

The last two operators can be confusing. The `|` operator concatenates the input arrays and removes duplicates from the concatenation. That is, if both input arrays contain `x` and `y`, the result will only contain one instance each of `x` and `y`. The `&` operator creates an array composed only of duplicate elements. If `x` and `y` are the only duplicates in the input arrays, the result of the `&` operator will contain only `x` and `y`.

Array Methods

Ruby arrays have methods that can be called like the `String` methods discussed earlier. Table 2.5 lists twelve important methods, and further methods will be introduced as they become needed.

Table 2.5

Ruby Array Methods

Method	Description	Example
<code>new</code>	Create a new array	<code>num_list = Array.new(30)</code>
<code>length</code>	Return the number of elements in the array	<code>[1, 2, 3, 4].length → 4</code>
<code>index</code>	Return the index of the given element value	<code>[1, 2, 3].index(1) → 0</code>
<code>concat</code>	Concatenate two arrays	<code>[1, 2, 3].concat([4, 5]) → [1, 2, 3, 4, 5]</code>

<code>delete_at</code>	Remove an element from the array by index	<code>a = [1, 2, 3]</code> <code>a.delete_at(1)</code> <code>a → [1, 3]</code>
<code>delete</code>	Remove an element from the array by element value	<code>a = [1, 2, 3]</code> <code>a.delete(1)</code> <code>a → [2, 3]</code>
<code>clear</code>	Remove all elements from the array	<code>["a", "b", "c"].clear → []</code>
<code>uniq</code>	Remove duplicate elements from array	<code>[1, 1, 2, 2, 3].uniq →</code> <code>[1, 2, 3]</code>
<code>fill</code>	Replace elements of the array with a value	<code>[1, 2, 3].fill(7) → (7, 7, 7)</code>
<code>sort</code>	Sort the array's elements	<code>["ab", "yz", "wx", "ac"].sort →</code> <code>["ab", "ac", "wx", "yz"]</code>
<code>first</code>	Return the first element of the array	<code>[1, 2, 3].first → 1</code>
<code>last</code>	Return the last element of the array	<code>[1, 2, 3].last → 3</code>

The `new` method creates new arrays. It can be used in three different ways, as shown in the following examples:

1. `num_list = Array.new 20` - creates an array with a given size without initializing its elements
2. `num_list = Array.new 4, "xyz"` - creates a new array with four elements, each set to "xyz"
3. `num_list = Array.new old_list` - creates a new array with the same elements as those in the array `old_list`

The `length` method identifies how many elements are in the array. `index` returns the position associated with a given element value. This is the reverse of regular array usage, which returns the element value associated with an index.

The `concat` method is similar to the `+` operator in Table 2.4, and appends elements of one array to the end of second array. The difference is that `+` creates a third array to store the concatenated result and `concat` modifies the second array. The following commands show how this is used:

```
first_array = [5, 4, 3, 2, 1]
```

```
second_array = [8, 7, 6].concat(first_array)
```

The second command appends the elements of `first_array` to the end of `second_array`, which now equals `[8, 7, 6, 5, 4, 3, 2, 1]`.

The `delete_at` and `delete` methods both remove array elements, but operate in different ways. The `delete_at` method removes an array element at a specific index, while the `delete` method removes all elements with a specific value. The following commands show how these two methods are used:

```
test = [10, "x", 9, "x", 8, "y", "x"]
test.delete_at 4
    → test = [10, "x", 9, "x", "y", "x"]
test.delete "x"
    → test = [10, 9, "y"]
```

The `delete_at` method removes the array's fifth element, whose value is 8. The `delete` method removes the elements whose value equals "x".

The `uniq` method removes duplicate elements from an array and the `clear` method removes all elements from an array, returning an empty array. The `fill` method sets elements of an array equal to a given value. This method can be used in four primary ways, as shown by the following commands:

```
fill_test = [1, 2, 3, 4, 5, 6]
fill_test.fill 9 sets all of the elements equal to 9
fill_test.fill 10, 0..2 sets the first through third elements equal to 10
fill_test.fill 21, 4, 2 sets two elements equal to 21, starting with element 4
```

The `sort` method places the array's elements in numeric or alphabetic order, but only if all of the elements are numeric or all alphabetic. The `first` method returns the first element of the array and `last` returns the last element of the array. These methods can be replaced by accessing array indices 0 and -1 respectively.

2.6 Objects, Classes, and Methods

In the early days of computer programming, the only way to store data was to create variables, constants, strings, and arrays. But as time progressed and applications grew in complexity, coders became tired of managing thousands of disorganized values. So they decided to group related data and operations into structures called *objects*. The nature of the data and operations in an object are determined by the object's *class*, and the operations defined in a class are called its *methods*.

Entire books have been written on the subject of object-oriented programming, so this section's treatment of Ruby objects will be shamefully brief. For more information, I recommend *Object-Oriented Analysis and Design with Applications* by Grady Booch, Robert A. Maksimchuk, Michael W. Engel, and Bobbi J. Young. Alternatively, there are many free resources on the web that discuss the principles of object-oriented design.

Objects

To build a large-scale model in SketchUp, you need to specify values for many characteristics, including coordinates, materials, textures, and colors. These settings are easy to access in the design window—click on a line and SketchUp tells you its length. But in software, managing this much data is a difficult task.

To make our lives bearable, we organize related characteristics into hierarchical data structures. For example, if we're modeling a house, we'll create one overall data structure for the house and lower-level substructures for its walls, doors, and roof. A door substructure may contain sub-substructures that model the door's knob, lock, and paneling.

In software, these data structures are called *objects*. An object can be accessed just like one of the variables we looked at earlier. But unlike a variable, an object contains multiple related values. For example, while `door_height` identifies the height of a door, a object of type `Door` may contain values for the door's height, width, depth, material, and color.

It's helpful to distinguish objects from arrays. In Ruby, an array may contain elements of any type, but an object only contains data needed to model a *thing*—a physical object or abstract principle. For example, if the design of a house needs to keep track of each door's height, width, and material, then these are the values stored by objects of `Door` type. The design may also contain objects of type `Window`, `Porch`, and `Garage`.

Two objects of the same type must have the same characteristics, but not necessarily the same values. If the `Door` type defines a height and width, and objects `door1` and `door2` are both of type `Door`, then `door1` and `door2` must store values for height and width—but not necessarily the same values.

The specific term for an object's type is its *class*. It's important to understand the relationship between objects and classes, and this is the focus of the following discussion.

Classes

A class defines the structure of an object in the same way that a set of blueprints defines the structure of a building or a strand of DNA defines the structure of an organism. More specifically, a class identifies the data contained within an object and the methods available for operating on the object's data.

The topic of coding new classes will have to wait until Chapter 8. For now, you only need to know what classes are and how to create objects from existing classes. Between the Ruby libraries and the SketchUp API, there are hundreds of classes available.

In Ruby, everything we work with is an object. Therefore, everything we work with has a class. The `class` method displays the name of an object's class, as shown in the following code:

```
5.class
  → Fixnum
3.14159.class
  → Float
"Hello, world".class
  → String
[5, 6, 7].class
  → Array
```

As shown, `5` is an object of class `Fixnum` (fixed-point number), `3.14159` is an object of class `Float` (floating-point number), `"Hello, world"` is an object of class `String`, and `[5, 6, 7]` is an object of class `Array`. If you analyze someone else's code, the `class` method makes it easy to determine precisely what type of data you're dealing with.

The `Fixnum`, `Float`, `String`, and `Array` classes are all provided by the Ruby Standard Library. But the classes we'll be focusing on in this book are made available through the SketchUp API. The following two are particularly important, and will be discussed in the next chapter:

- `Edge` - an object created from the `Edge` class represents a line segment in a SketchUp design
- `Face` - an object created from the `Face` class represents a two-dimensional surface in a SketchUp design

Note: Rather than use phrases such as "an object created from the `Face` class" or "an object of `Face` type," this book will refer to these objects as `Face` objects or `Faces`. But remember that `Face` is the name of the class, not the object.

There are over eighty different classes in the SketchUp API, and Appendix A lists them all. You can also visit the <http://code.google.com/apis/sketchup/docs/index.html> web site. There, you can click through the links to see what each class accomplishes.

Instance Methods

The earlier discussion of `Strings` explained basic operators like `+` and `*`, and also presented a list of named operations called *methods*. These methods operate on `String` objects—if `str` is a `String`, `str.length` returns the number of characters in `str`. Similarly, `str.downcase` converts the characters to lower case. For a full listing of `String` methods, enter the following command in the Ruby Console:

```
"Hello".methods
```

This lists all the methods available to a `String` object. If `str` is a `String` variable, you can accomplish the same result by invoking `str.methods`. These methods are defined in the `String` class, and all `String` objects, such as `str`, can invoke them.

The `Array` class provides another set of methods for its objects, as shown here:

```
arr = [0, 1, 2]
arr.length
  → 3
```

```
arr.first
    → 0
arr.last
    → 2
```

A method is a procedure defined in a class that operates on object data. Put simply, an object represents a thing and a method provides a means of interacting with the thing's characteristics. Methods are called or invoked using *dot-notation*: the object is followed by a dot and the method name. For example, if the `Auto` class defines a method called `reverse` and `auto1` and `auto2` are `Auto` objects, you can call `auto1.reverse` and `auto2.reverse`.

Many methods require additional data to operate. In Table 2.5, the `fill` method in the `Array` class needs an input value to replace the values of the elements in the input array. This additional data, called an *argument* or a *parameter*, can be provided with or without parentheses, as is shown in the following commands:

```
arr = [0, 1, 2, 3]
arr.fill(7)
    → [7, 7, 7, 7]
arr.fill 7
    → [7, 7, 7, 7]
```

If a method requires multiple arguments, the arguments must be separated by commas, whether the list of arguments is surrounded by parentheses or not.

Appendix A lists the methods in each of the classes defined in the SketchUp API. If you look closely, you'll notice that many method names end with a `?` while others end with a `=`. This gives you an idea of how the method works. If the method ends with a `?`, it returns `true` or `false`. For example, the `include?` method in the `String` class returns `true` if the argument is part of the `String` and `false` otherwise. This is shown by the following examples:

```
str = "Hello, world"
str.include? "ell"
    → true
```

```
str.include?("word")  
→ false
```

If a Ruby method ends with `=`, it updates the object with the data supplied by the parameter. This can be demonstrated by using the `[]=` operator of the `Array` class, which changes the value of an array element to that of the parameter. The following command sets the third element of `arr` equal to 5:

```
arr[2] = 5
```

Ruby methods can be chained together. That is, if `method_B` can operate on the value returned by `method_A`, you can invoke both methods with `method_A.method_B`. For example, let's say you want to reverse the characters in the uppercase conversion of "Hello". You can do this using multiple commands, as shown by the following:

```
str = "Hello"  
str1 = str.upcase  
→ HELLO  
str2 = str1.reverse  
→ OLLEH
```

Or you can accomplish the same result in one command:

```
str = "Hello".upcase.reverse  
→ OLLEH
```

In this example, the `reverse` method operates on the result of `"Hello".upcase`. Method chaining reduces the amount of code you need to enter, but makes your code slightly less readable.

Class Methods

The previous discussion implied that all methods defined in a class can only be accessed through objects. This is frequently the case, but it's not entirely correct. Some methods in a class operate on the class itself. Methods that operate on classes are called *class methods*. Methods that operate on objects are called *instance methods* because an object is an instance of a class.

There is one important class method contained in every Ruby class. This is the `new` method, and it creates a new object from a class in the same manner that a construction team creates a new building from blueprints. For example, the following command calls the `new` method of the `String` class to create a new `String` object:

```
new_str = String.new
new_str.class
      → String
```

In this book, the vast majority of the methods we'll be using are instance methods. Any method discussed in this book can be assumed to be an instance method unless specifically described as a class method.

2.7 Class Inheritance

In many situations, you may have to access classes that have characteristics in common. For example, if you're an architect, you might need to model hotels and hospitals. The two structures are different enough to require separate classes: `Hotel` and `Hospital`. But the two classes also contain similar characteristics, such as location, material, and number of stories. For this reason, the following methods could apply equally well to either the `Hotel` or `Hospital` classes:

- `num_stories` - the number of stories in the structure
- `location` - the structure's geographic location
- `material` - the type of material used to build the structure

Rather than code the same methods twice, it's more efficient to place them in a third class so they will be available to `Hotel` and `Hospital` objects. This third class should embody the commonality of the two classes, and in this example, we'll call the common class `Building`. We'll also set up a relationship between `Building`, `Hotel`, and `Hospital` such that `Hotel` and `Hospital` both receive the methods defined in `Building`.

This relationship between classes is called *inheritance*, and both `Hotel` and `Hospital` are said to inherit from `Building`. In this example, `Building` is called a *superclass* of `Hotel` and `Hospital`, and `Hotel` and `Hospital` are called *subclasses* of `Building`. Figure 2.4 shows what a class inheritance hierarchy looks like.

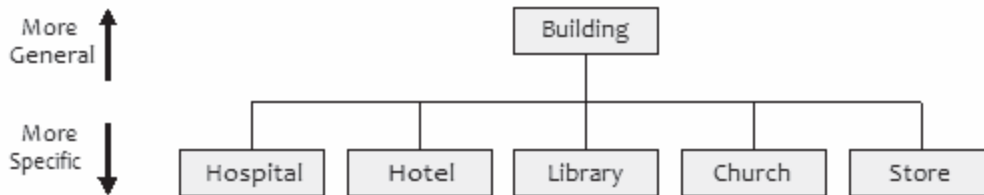


Figure 2.4: Simple Example Inheritance Hierarchy

Now you can define the `num_stories`, `location`, and `material` methods in the single `Building` class, and if you need to rewrite any of them, you only have to modify a single class. Also, by creating the `Building` class, you can easily add more classes that represent buildings like libraries, churches, and stores.

Let's look at a real-world example of class inheritance. The Ruby interpreter processes numbers differently depending on how much memory they occupy. If an integer occupies 31 bits or less, it's a `Fixnum` object. If an integer occupies more than 31 bits, it's a `Bignum` object. This is why `24.class` returns `Fixnum` and `1234567890.class` returns `Bignum`.

The two classes require different methods for certain operations, but between the two, many of the methods can remain the same. For example, the `next` method returns the succeeding integer in numerical order. `24.next` returns 25 and `1234567890.next` returns 1234567891.

For this reason, Ruby has a class specifically for integers called `Integer`. Common methods like `next` are placed in the `Integer` class, and because `Fixnum` and `Bignum` both inherit from `Integer`, the method is available for objects of both classes. Figure 2.5 shows how `Integer`, `Fixnum`, and `Bignum` are positioned in Ruby's numeric class hierarchy.

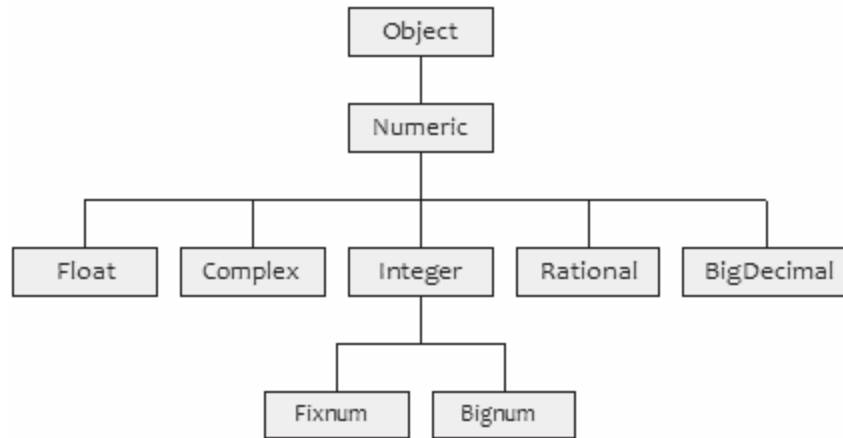


Figure 2.5: Ruby Number Class Hierarchy

Chapter 8 will explain how to create classes and subclasses in code. For now, all you have to understand about class inheritance is this: If Class B inherits from Class A (i.e. Class A is the superclass of Class B), then all the methods in A are available to Class B. This means that any Class B object can access all the same methods as an object of Class A.

2.8 Conclusion

The goal of this chapter is to give you enough of a foundation in Ruby so that you can understand SketchUp objects and how they're used in code. This lesson began with a discussion of numbers, `Strings`, arrays, variables and constants, and then continued to explain objects and classes. We'll be working with these data structures throughout this book.

As mentioned earlier, you can think of an object as a *thing*. An object's method is a means of interacting with the *thing*. In Ruby, every data structure is an object, including numbers and variables. The number 5 has Ruby methods that can be called using the same dot-notation as the methods of a `String` or array.

A class defines the structure of an object, including the object's data and the methods available to operate on the object. Some methods accept arguments, and in Ruby, these arguments can optionally be surrounded by parentheses. Multiple arguments must be separated by commas. Some methods, called class methods, operate on a class instead of an object. The

most important of these is `new`, which creates a new object from the class.

Object-oriented programming is an involved topic, and if it's not all perfectly clear yet, don't be concerned. As you work through the examples in this book, you'll be able to see the principles of OO coding made manifest in real-world code. In the next chapter, we'll put aside theoretical concerns and see how SketchUp's objects work together to create three-dimensional models.