Chapter 6

Standard Types

So far we've been having fun implementing programs using arrays, hashes, and procs, but we haven't really covered the other basic types in Ruby: numbers, strings, ranges, and regular expressions. Let's spend a few pages on these basic building blocks now.

Numbers

Ruby supports integers and floating-point, rational, and complex numbers. Integers can be any length (up to a maximum determined by the amount of free memory on your system). Integers within a certain range (normally $-2^{30} \ldots 2^{30}-1$ or $-2^{62} \ldots 2^{62}-1$) are held internally in binary form and are objects of class Fixnum. Integers outside this range are stored in objects of class Bignum (currently implemented as a variable-length set of short integers). This process is transparent, and Ruby automatically manages the conversion back and forth:

```
num = 81
6.times do
  puts "#{num.class}: #{num}"
  num *= num
end
```

produces:

```
Fixnum: 81
Fixnum: 6561
Fixnum: 43046721
Bignum: 1853020188851841
Bignum: 3433683820292512484657849089281
Bignum: 11790184577738583171520872861412518665678211592275841109096961
```

You write integers using an optional leading sign, an optional base indicator (`0` for octal, `0d` for decimal [the default], `0x` for hex, or `0b` for binary), followed by a string of digits in the appropriate base. Underscore characters are ignored in the digit string (some folks use them in place of commas in larger numbers).

```
123456                  => 123456   # Fixnum
0d123456                => 123456   # Fixnum
123_456                 => 123456   # Fixnum - underscore ignored
-543                    => -543     # Fixnum - negative number
0xaabb                  => 43707    # Fixnum - hexadecimal
0377                    => 255      # Fixnum - octal
-0b10_1010              => -42      # Fixnum - binary (negated)
123_456_789_123_456_789 => 123456789123456789 # Bignum
```

A numeric literal with a decimal point and/or an exponent is turned into a Float object, corresponding to the native architecture's double data type. You must both precede and follow the decimal point with a digit (if you write 1.0e3 as 1.e3, Ruby will try to invoke the method e3 on the object 1).

1.9

As of Ruby 1.9, rational and complex number support is built into the interpreter. Rational numbers are the ratio of two integers—they are fractions—and hence have an exact representation (unlike floats). Complex numbers represent points on the complex plane. They have two components, the real and imaginary parts.

Ruby doesn't have a literal syntax for representing rational and complex numbers. Instead, you create them using explicit calls to the constructor methods Rational and Complex (although, as we'll see, you can use the mathn library to make working with rational numbers easier).

```
Rational(3, 4) * Rational(2, 3)     # =>   (1/2)
Rational("3/4") * Rational("2/3")   # =>   (1/2)

Complex(1, 2) * Complex(3, 4)       # =>   (-5+10i)
Complex("1+2i") * Complex("3+4i")   # =>   (-5+10i)
```

All numbers are objects and respond to a variety of messages (listed in full starting on pages 466 [Bignum], 473 [Complex], 525 [Fixnum], 528 [Float], 543 [Integer], 615 [Numeric], and 660 [Rational]). So, unlike (say) C++, you find the absolute value of a number by writing num.abs, not abs(num).

Finally, we'll offer a warning for Perl users. Strings that contain just digits are *not* automatically converted into numbers when used in expressions. This tends to bite most often when reading numbers from a file. For example, we may want to find the sum of the two numbers on each line for a file such as the following:

```
3 4
5 6
7 8
```

The following code doesn't work:

```
some_file.each do |line|
  v1, v2 = line.split    # split line on spaces
  print v1 + v2, " "
end
```

produces:

```
34 56 78
```

The problem is that the input was read as strings, not numbers. The plus operator concatenates strings, so that's what we see in the output. To fix this, use the Integer method to convert the strings to integers:

```
some_file.each do |line|
  v1, v2 = line.split
  print Integer(v1) + Integer(v2), " "
end
```

produces:

```
7 11 15
```

How Numbers Interact

Most of the time, numbers work the way you'd expect. If you perform some operation between two numbers of the same class, the answer will typically be a number of that same class (although, as we've seen, fixnums can become bignums, and vice versa). If the two numbers are different classes, the result will have the class of the more general one. If you mix integers and floats, the result will be a float; if you mix floats and complex numbers, the result will be complex.

```
1 + 2                 # =>   3
1 + 2.0               # =>   3.0
1.0 + 2               # =>   3.0
1.0 + Complex(1,2)    # =>   (2.0+2i)
1 + Rational(2,3)     # =>   (5/3)
1.0 + Rational(2,3)   # =>   1.66666666666667
```

The return-type rule still applies when it comes to division. However this often confuses folks, because division between two integers yields an integer result:

```
1.0 / 2   # =>   0.5
1 / 2.0   # =>   0.5
1 / 2     # =>   0
```

If you'd prefer that integer division instead return a fraction (a Rational number), require the mathn library (described on page 767). This will cause arithmetic operations to attempt to find the most *natural* representation for their results. For integer division where the result isn't an integer, a fraction will be returned.

```
22 / 7                    # =>   3
Complex::I * Complex::I   # =>   (-1+0i)

require 'mathn'
22 / 7                    # =>   (22/7)
Complex::I * Complex::I   # =>   -1
```

Note that 22/7 is effectively a rational literal once mathn is loaded (albeit one that's calculated at runtime).

Looping Using Numbers

Integers also support several useful iterators. We've seen one already: 6.times in the code example on page 106. Others include upto and downto for iterating up and down between two integers. Class Numeric also provides the more general method step, which is more like a traditional for loop.

```
3.times        { print "X " }
1.upto(5)      {|i| print i, " " }
99.downto(95)  {|i| print i, " " }
50.step(80, 5) {|i| print i, " " }
```

produces:

```
X X X 1 2 3 4 5 99 98 97 96 95 50 55 60 65 70 75 80
```

1.9 As with other iterators, if you leave the block off, the call returns an Enumerator object:

```
10.downto(7).with_index {|num, index| puts "#{index}: #{num}"}
```

produces:

```
0: 10
1: 9
2: 8
3: 7
```

Strings

1.9 Ruby strings are simply sequences of characters.[1] They normally hold printable characters, but that is not a requirement; a string can also hold binary data. Strings are objects of class String.

Strings are often created using string literals—sequences of characters between delimiters. Because binary data is otherwise difficult to represent within program source, you can place various escape sequences in a string literal. Each is replaced with the corresponding binary value as the program is compiled. The type of string delimiter determines the degree of substitution performed. Within single-quoted strings, two consecutive backslashes are replaced by a single backslash, and a backslash followed by a single quote becomes a single quote.

```
'escape using "\\"'   # =>  escape using "\"
'That\'s right'       # =>  That's right
```

Double-quoted strings support a boatload more escape sequences. The most common is probably \n, the newline character. Table 22.2 on page 329 gives the complete list. In addition, you can substitute the value of any Ruby code into a string using the sequence #{ *expr* }. If the code is just a global variable, a class variable, or an instance variable, you can omit the braces.

1. Prior to Ruby 1.9, strings were sequences of 8-bit bytes.

```
"Seconds/day: #{24*60*60}"      # =>  Seconds/day: 86400
"#{'Ho! '*3}Merry Christmas!"   # =>  Ho! Ho! Ho! Merry Christmas!
"This is line #$."              # =>  This is line 3
```

The interpolated code can be one or more statements, not just an expression:

```
puts  "now is #{ def the(a)
                   'the ' + a
                 end
                 the('time')
               } for all good coders..."
```

produces:

```
now is the time for all good coders...
```

You have three more ways to construct string literals: %q, %Q, and *here documents*.

%q and %Q start delimited single- and double-quoted strings (you can think of %q as a thin quote ' and %Q as a thick quote "):

```
%q/general single-quoted string/   # =>  general single-quoted string
%Q!general double-quoted string!   # =>  general double-quoted string
%Q{Seconds/day: #{24*60*60}}       # =>  Seconds/day: 86400
```

In fact, the Q is optional:

```
%!general double-quoted string!   # =>  general double-quoted string
%{Seconds/day: #{24*60*60}}       # =>  Seconds/day: 86400
```

The character following the *q* or *Q* is the delimiter. If it is an opening bracket ([), brace ({), parenthesis ((), or less-than sign (<), the string is read until the matching close symbol is found. Otherwise, the string is read until the next occurrence of the same delimiter. The delimiter can be any nonalphanumeric or nonmultibyte character.

Finally, you can construct a string using a *here document*:

```
string = <<END_OF_STRING
    The body of the string
    is the input lines up to
    one starting with the same
    text that followed the '<<'
END_OF_STRING
```

A here document consists of lines in the source up to but not including the terminating string that you specify after the << characters. Normally, this terminator must start in the first column. However, if you put a minus sign after the << characters, you can indent the terminator:

```
string = <<-END_OF_STRING
    The body of the string is the input lines up to
    one starting with the same text that followed the '<<'
    END_OF_STRING
```

You can also have multiple here documents on a single line. Each acts as a separate string. The bodies of the here documents are fetched sequentially from the source lines that follow.

```
print <<-STRING1, <<-STRING2
   Concat
   STRING1
      enate
      STRING2
```

produces:

```
Concat
   enate
```

Note that Ruby does not strip leading spaces off the contents of the strings in these cases.

Strings and Encodings

1.9

In Ruby 1.9, every string has an associated encoding. The default encoding of a string literal depends on the encoding of the source file that contains it. With no explicit encoding, a source file (and its strings) will be US-ASCII.

```
plain_string = "dog"
puts "Encoding of #{plain_string.inspect} is #{plain_string.encoding}"
```

produces:

```
Encoding of "dog" is US-ASCII
```

If you override the encoding, you'll do that for all strings in the file:

```
#encoding: utf-8
plain_string = "dog"
puts "Encoding of #{plain_string.inspect} is #{plain_string.encoding}"
utf_string = "δog"
puts "Encoding of #{utf_string.inspect} is #{utf_string.encoding}"
```

produces:

```
Encoding of "dog" is UTF-8
Encoding of "δog" is UTF-8
```

We'll have a lot more to say about encoding in Chapter 17 on page 264.

Character Constants

Technically, Ruby does not have a class for characters—characters are simply strings of length one. For historical reasons, character constants can be created by preceding the character (or sequence that represents a character) with a question mark:

?a	# => "a"	(printable character)
?\n	# => "\n"	(code for a newline (0x0a))
?\C-a	# => "\x01"	(control a)
?\M-a	# => "\xE1"	(meta sets bit 7)
?\M-\C-a	# => "\x81"	(meta and control a)
?\C-?	# => "\x7F"	(delete character)

Do yourself a favor and immediately forget this section. It's far easier to use regular octal and hex escape sequences than to remember these ones. Use "a" rather than ?a, and use "\n" rather than ?\n.

Working with Strings

String is probably the largest built-in Ruby class, with more than 100 standard methods. We won't go through them all here; the library reference has a complete list. Instead, we'll look at some common string idioms—things that are likely to pop up during day-to-day programming.

Maybe we've been given a file containing information on a song playlist. For historical reasons (are there any other kind?), the list of songs is stored as lines in the file. Each line holds the name of the file containing the song, the song's duration, the artist, and the title, all in vertical bar–separated fields. A typical file may start like this:

```
/jazz/j00132.mp3  | 3:45 | Fats     Waller     | Ain't Misbehavin'
/jazz/j00319.mp3  | 2:58 | Louis    Armstrong  | Wonderful World
/bgrass/bg0732.mp3| 4:09 | Strength in Numbers | Texas Red
         :                  :           :                    :
```

Looking at the data, it's clear that we'll be using some of class String's many methods to extract and clean up the fields before we use them. At a minimum, we'll need to

- break each line into fields,
- convert the running times from mm:ss to seconds, and
- remove those extra spaces from the artists' names.

Our first task is to split each line into fields, and String#split will do the job nicely. In this case, we'll pass split a regular expression, /\s*\|\s*/, that splits the line into tokens wherever split finds a vertical bar, optionally surrounded by spaces. And, because the line read from the file has a trailing newline, we'll use String#chomp to strip it off just before we apply the split. We'll store details of each song in a Struct that contains an attribute for each of the three fields. (A Struct is simply a data structure that contains a given set of attributes—in this case the title, name, and length. See page 696 for the gory details.)

Download **samples/tutstdtypes_24.rb**

```
Song = Struct.new(:title, :name, :length)
File.open("songdata") do |song_file|
  songs = []
  song_file.each do |line|
    file, length, name, title = line.chomp.split(/\s*\|\s*/)
    songs << Song.new(title, name, length)
  end
  puts songs[1]
end
```

produces:

```
#<struct Song title="Wonderful World", name="Louis    Armstrong", length="2:58">
```

Unfortunately, whoever created the original file entered the artists' names in columns, so some of them contain extra spaces that we'd better remove before we go much further. We have many ways of doing this, but probably the simplest is String#squeeze, which trims runs of repeated characters. We'll use the squeeze! form of the method, which alters the string in place:

Download samples/tutstdtypes_25.rb

```
Song = Struct.new(:title, :name, :length)
File.open("songdata") do |song_file|
  songs = []
  song_file.each do |line|
    file, length, name, title = line.chomp.split(/\s*\|\s*/)
    name.squeeze!(" ")
    songs << Song.new(title, name, length)
  end
  puts songs[1]
end
```

produces:

```
#<struct Song title="Wonderful World", name="Louis Armstrong", length="2:58">
```

Finally, we have the minor matter of the time format: the file says 2:58, and we want the number of seconds, 178. We could use split again, this time splitting the time field around the colon character:

```
mins, secs = length.split(/:/)
```

Instead, we'll use a related method. String#scan is similar to split in that it breaks a string into chunks based on a pattern. However, unlike split, with scan you specify the pattern that you want the chunks to match. In this case, we want to match one or more digits for both the minutes and seconds components. The pattern for one or more digits is /\d+/:

Download samples/tutstdtypes_27.rb

```
Song = Struct.new(:title, :name, :length)
File.open("songdata") do |song_file|
  songs = []
  song_file.each do |line|
    file, length, name, title = line.chomp.split(/\s*\|\s*/)
    name.squeeze!(" ")
    mins, secs = length.scan(/\d+/)
    songs << Song.new(title, name, mins.to_i*60 + secs.to_i)
  end
  puts songs[1]
end
```

produces:

```
#<struct Song title="Wonderful World", name="Louis Armstrong", length=178>
```

We could spend the next 50 pages looking at all the methods in class String. However, let's move on instead to look at a simpler data type: the range.

Ranges

Ranges occur everywhere: January to December, 0 to 9, rare to well done, lines 50 through 67, and so on. If Ruby is to help us model reality, it seems natural for it to support these ranges. In fact, Ruby goes one better: it actually uses ranges to implement three separate features: sequences, conditions, and intervals.

Ranges as Sequences

The first and perhaps most natural use of ranges is to express a sequence. Sequences have a start point, an end point, and a way to produce successive values in the sequence. In Ruby, these sequences are created using the .. and ... range operators. The two-dot form creates an inclusive range, and the three-dot form creates a range that excludes the specified high value:

```
1..10
'a'..'z'
0..."cat".length
```

1.9

You can convert a range to an array using the to_a method and convert it to an Enumerator using to_enum:[2]

```
(1..10).to_a          # =>   [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
('bar'..'bat').to_a   # =>   ["bar", "bas", "bat"]
enum = ('bar'..'bat').to_enum
enum.next             # =>   "bar"
enum.next             # =>   "bas"
```

Ranges have methods that let you iterate over them and test their contents in a variety of ways:

```
digits = 0..9
digits.include?(5)          # =>   true
digits.min                  # =>   0
digits.max                  # =>   9
digits.reject {|i| i < 5 }  # =>   [5, 6, 7, 8, 9]
digits.inject(:+)           # =>   45
```

So far we've shown ranges of numbers and strings. However, as you'd expect from an object-oriented language, Ruby can create ranges based on objects that you define. The only constraints are that the objects must respond to succ by returning the next object in sequence and the objects must be comparable using <=>. Sometimes called the *spaceship operator*, <=>, compares two values, returning -1, 0, or $+1$ depending on whether the first is less than, equal to, or greater than the second.

2. Sometimes people worry that ranges take a lot of memory. That's not an issue: the range 1..100000 is held as a Range object containing references to two Fixnum objects. However, convert a range into an array, and all that memory will get used.

In reality, this isn't something you do very often, so examples tend to be a bit contrived. Here's one—a class that presents numbers that are powers of 2. Because it defines <=> and succ, we can use objects of this class in ranges:

Download samples/tutstdtypes_31.rb

```
class PowerOfTwo
  attr_reader :value
  def initialize(value)
    @value = value
  end
  def <=>(other)
    @value <=> other.value
  end
  def succ
    PowerOfTwo.new(@value + @value)
  end
  def to_s
    @value.to_s
  end
end
p1 = PowerOfTwo.new(4)
p2 = PowerOfTwo.new(32)
puts (p1..p2).to_a
```

produces:

```
4
8
16
32
```

Ranges as Conditions

As well as representing sequences, ranges can also be used as conditional expressions. Here, they act as a kind of toggle switch—they turn on when the condition in the first part of the range becomes true, and they turn off when the condition in the second part becomes true. For example, the following code fragment prints sets of lines from standard input, where the first line in each set contains the word *start* and the last line contains the word *end*:

```
while line = gets
  puts line if line =~ /start/ .. line =~ /end/
end
```

Behind the scenes, the range keeps track of the state of each of the tests. We'll show some examples of this in the description of loops that starts on page 160 and in the language section on page 348.

Ranges as Intervals

A final use of the versatile range is as an interval test: seeing whether some value falls within the interval represented by the range. We do this using ===, the case equality operator:

```
(1..10)    === 5         # =>   true
(1..10)    === 15        # =>   false
(1..10)    === 3.14159   # =>   true
('a'..'j') === 'c'       # =>   true
('a'..'j') === 'z'       # =>   false
```

This is most often used in case statements:

```
car_age = gets.to_f    # let's assume it's 5.2
case car_age
when 0...1
  puts "Mmm.. new car smell"
when 1...3
  puts "Nice and new"
when 3...6
  puts "Reliable but slightly dinged"
when 6...10
  puts "Can be a struggle"
when 10...30
  puts "Clunker"
else
  puts "Vintage gem"
end
```

produces:

```
Reliable but slightly dinged
```

Note the use of exclusive ranges in the previous example. These are normally the correct choice in case statements. If instead we'd written the following, we'd get the wrong answer because 5.2 does not fall within any of the ranges, so the else clause triggers:

Download **samples/tutstdtypes_35.rb**

```
car_age = gets.to_f    # let's assume it's 5.2
case car_age
when 0..0
  puts "Mmm.. new car smell"
when 1..2
  puts "Nice and new"
when 3..5
  puts "Reliable but slightly dinged"
when 6..9
  puts "Can be a struggle"
when 10..29
  puts "Clunker"
else
  puts "Vintage gem"
end
```

produces:

```
Vintage gem
```