Standard Types

So far we've been having fun implementing pieces of our jukebox code, but we've been negligent. We've looked at 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 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} to $2^{30}-1$ or -2^{62} to $2^{62}-1$) 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 = 8
7.times do
  print num.type, " ", num, "\n"
  num *= num
end
```

produces:

```
Fixnum 8
Fixnum 64
Fixnum 4096
Fixnum 16777216
Bignum 281474976710656
Bignum 79228162514264337593543950336
Bignum 6277101735386680763835789423207666416102355444464034512896
```

You write integers using an optional leading sign, an optional base indicator (`0` for octal, `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.

```
123456                  # Fixnum
123_456                 # Fixnum (underscore ignored)
-543                    # Negative Fixnum
123_456_789_123_345_789 # Bignum
0xaabb                  # Hexadecimal
0377                    # Octal
-0b101_010              # Binary (negated)
```

You can also get the integer value corresponding to an ASCII character or escape sequence by preceding it with a question mark. Control and meta combinations can also be generated using ?\C-*x*, ?\M-*x*, and ?\M-\C-*x*. The control version of a value is the same as ``value & 0x9f''. The meta version of a value is ``value | 0x80''. Finally, the sequence ?\C-? generates an ASCII delete,`0177`.

```
?a                      # character code
?\n                     # code for a newline (0x0a)
?\C-a                   # control a = ?A & 0x9f = 0x01
?\M-a                   # meta sets bit 7
?\M-\C-a                # meta and control a
?\C-?                   # delete character
```

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 follow the decimal point with a digit, as `1.e3` tries to invoke the method `e3` in class`Fixnum`.

All numbers are objects, and respond to a variety of messages (listed in full starting on pages 290, 313, 315, 323, and 349). So, unlike (say) C++, you find the absolute value of a number by writing `aNumber.abs`, not `abs(aNumber)`.

Integers also support several useful iterators. We've seen one already---`7.times`in the code example on page 47. Others include `upto` and `downto`, for iterating up and down between two integers, and `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
```

Finally, a warning for Perl users. Strings that contain numbers are not automatically converted into numbers when used in expressions. This tends to bite most often when reading numbers from a file. The following code (probably) doesn't do what was intended.

```
DATA.each do |line|
  vals = line.split    # split line, storing tokens in val
  print vals[0] + vals[1], " "
end
```

Feed it a file containing

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

and you'll get the output ``34 56 78." What happened?

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 `String#to_i` method to convert the string to an integer.

```
DATA.each do |line|
  vals = line.split
  print vals[0].to_i + vals[1].to_i, " "
end
```

produces:

```
7 11 15
```

Strings

Ruby strings are simply sequences of 8-bit bytes. 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 18.2 on page 203 gives the complete list. In addition, you can substitute the value of any Ruby expression into a string using the sequence `#{` *expr* `}`. If the expression is just a global variable, a class variable, or an instance variable, you can omit the braces.

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

There are three more ways to construct string literals: `%q`, `%Q`, and ``here documents.''

`%q` and `%Q` start delimited single- and double-quoted strings.

```
%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
```

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.

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

```
aString = <<END_OF_STRING
    The body of the string
    is the input lines up to
    one ending 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.

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

produces:

```
     Concat
        enate
```

Working with Strings

`String` is probably the largest built-in Ruby class, with over 75 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.

Let's get back to our jukebox. Although it's designed to be connected to the Internet, it also holds copies of some popular songs on a local hard drive. That way, if a squirrel chews through our 'net connection we'll still be able to entertain the customers.

For historical reasons (are there any other kind?), the list of songs is stored as rows in a flat file. Each row 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 might start:

```
/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 create `Song` objects based on them. At a minimum, we'll need to:

- break the line into fields,
- convert the running time from mm:ss to seconds, and
- remove those extra spaces from the artist's name.

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*/`, which 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.

```
songs = SongList.new


songFile.each do |line|
  file, length, name, title = line.chomp.split(/\s*\|\s*/)
  songs.append Song.new(title, name, length)
end
puts songs[1]
```

produces:

```
Song: Wonderful World--Louis    Armstrong (2:58)
```

Unfortunately, whoever created the original file entered the artists' names in columns, so some of them contain extra spaces. These will look ugly on our high-tech, super-twist, flat-panel Day-Glo display, so we'd better remove these extra spaces before we go much further. There are 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.

```
songs = SongList.new


songFile.each do |line|
  file, length, name, title = line.chomp.split(/\s*\|\s*/)
  name.squeeze!(" ")
  songs.append Song.new(title, name, length)
end
puts songs[1]
```

produces:

```
Song: Wonderful World--Louis Armstrong (2:58)
```

Finally, there's 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 component. The pattern for one or more digits is `/\d+/`.

```
songs = SongList.new
songFile.each do |line|
  file, length, name, title = line.chomp.split(/\s*\|\s*/)
  name.squeeze!(" ")
  mins, secs = length.scan(/\d+/)
  songs.append Song.new(title, name, mins.to_i*60+secs.to_i)
end
puts songs[1]
```

produces:

```
Song: Wonderful World--Louis Armstrong (178)
```

Our jukebox has a keyword search capability. Given a word from a song title or an artist's name, it will list all matching tracks. Type in ``fats," and it might come back with songs by Fats Domino, Fats Navarro, and Fats Waller, for example. We'll implement this by creating an indexing class. Feed it an object and some strings, and it will index that object under every word (of two or more characters) that occurs in those strings. This will illustrate a few more of class `String`'s many methods.

```
class WordIndex
  def initialize
    @index = Hash.new(nil)
  end
  def index(anObject, *phrases)
    phrases.each do |aPhrase|
      aPhrase.scan /\w[-\w']+/ do |aWord|   # extract each word
        aWord.downcase!
        @index[aWord] = [] if @index[aWord].nil?
        @index[aWord].push(anObject)
      end
    end
  end
  def lookup(aWord)
    @index[aWord.downcase]
  end
end
```

The `String#scan` method extracts elements from a string that match a regular expression. In this case, the pattern ``\w[-\w']+'' matches any character that can appear in a word, followed by one or more of the things specified in the brackets (a hyphen, another word character, or a single quote). We'll talk more about regular expressions beginning on page 56. To make our searches case insensitive, we map both the words we extract and the words used as keys during the lookup to lowercase. Note the exclamation mark at the end of the first`downcase!` method name. As with the `squeeze!` method we used previously, this is an indication that the method will modify the receiver in place, in this case converting the string to lowercase.*[There's a minor bug in this code example: the song ``Gone, Gone, Gone" would get indexed three times. Can you come up with a fix?]*

We'll extend our `SongList` class to index songs as they're added, and add a method to look up a song given a word.

```
class SongList
  def initialize
    @songs = Array.new
    @index = WordIndex.new
  end
  def append(aSong)
    @songs.push(aSong)
    @index.index(aSong, aSong.name, aSong.artist)
    self
  end
  def lookup(aWord)
    @index.lookup(aWord)
  end
end
```

Finally, we'll test it all.

```
songs = SongList.new
songFile.each do |line|
  file, length, name, title = line.chomp.split(/\s*\|\s*/)
  name.squeeze!(" ")
  mins, secs = length.scan(/\d+/)
  songs.append Song.new(title, name, mins.to_i*60+secs.to_i)
end
puts songs.lookup("Fats")
puts songs.lookup("ain't")
puts songs.lookup("RED")
puts songs.lookup("WoRlD")
```

produces:

```
Song: Ain't Misbehavin'--Fats Waller (225)
Song: Ain't Misbehavin'--Fats Waller (225)
Song: Texas Red--Strength in Numbers (249)
Song: Wonderful World--Louis Armstrong (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 datatype: ranges.

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, while the three-dot form creates a range that excludes the specified high value.

```
1..10
'a'..'z'
0...anArray.length
```

In Ruby, unlike in some earlier versions of Perl, ranges are not represented internally as lists: the sequence 1..100000 is held as a `Range` object containing references to two `Fixnum` objects. If you need to, you can convert a range to a list using the `to_a` method.

```
(1..10).to_a                »  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
('bar'..'bat').to_a         »  ["bar", "bas", "bat"]
```

Ranges implement 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.each do |digit|
  dial(digit)
end
```

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`<=>`, the general comparison operator. 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.

Here's a simple class that represents rows of ``#'' signs. We might use it as a text-based stub when testing the jukebox volume control.

```
class VU

  include Comparable

  attr :volume

  def initialize(volume)  # 0..9
    @volume = volume
  end

  def inspect
    '#' * @volume
  end

  # Support for ranges

  def <=>(other)
    self.volume <=> other.volume
  end

  def succ
    raise(IndexError, "Volume too big") if @volume >= 9
```

```
    VU.new(@volume.succ)
  end
end
```

We can test it by creating a range of VU objects.

```
medium = VU.new(4)..VU.new(7)
medium.to_a                         » [####, #####, ######, #######]
medium.include?(VU.new(3))          » false
```

Ranges as Conditions

As well as representing sequences, ranges may also be used as conditional expressions. 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 the word ``end.''

```
while gets
  print if /start/../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 82.

Ranges as Intervals

A final use of the versatile range is as an interval test: seeing if some value falls within the interval represented by the range. This is done 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
```

The example of a case expression on page 81 shows this test in action, determining a jazz style given a year.

Regular Expressions

Back on page 50 when we were creating a song list from a file, we used a regular expression to match the field delimiter in the input file. We claimed that the expression `line.split(/\s*\|\s*/)` matched a vertical bar surrounded by optional whitespace. Let's explore regular expressions in more detail to see why this claim is true.

Regular expressions are used to match patterns against strings. Ruby provides built-in support that makes pattern matching and substitution convenient and concise. In this section we'll work through all the main features of regular expressions. There are some details we won't cover: have a look at page 205 for more information.

Regular expressions are objects of type `Regexp`. They can be created by calling the constructor explicitly or by using the literal forms /*pattern*/ and %r*pattern*\.

```
a = Regexp.new('^\s*[a-z]')                    »  /^\s*[a-z]/
b = /^\s*[a-z]/                                »  /^\s*[a-z]/
c = %r{^\s*[a-z]}                              »  /^\s*[a-z]/
```

Once you have a regular expression object, you can match it against a string using `Regexp#match(aString)` or the match operators `=~` (positive match) and `!~`(negative match). The match operators are defined for both `String` and `Regexp`objects. If both operands of the match operator are `Strings`, the one on the right will be converted to a regular expression.

```
a = "Fats Waller"
a =~ /a/                                       »  1
a =~ /z/                                       »  nil
a =~ "ll"                                      »  7
```

The match operators return the character position at which the match occurred. They also have the side effect of setting a whole load of Ruby variables. `$&`receives the part of the string that was matched by the pattern, `$`` receives the part of the string that preceded the match, and `$'` receives the string after the match. We can use this to write a method, `showRE`, which illustrates where a particular pattern matches.

```
def showRE(a,re)
  if a =~ re
    "#{$`}<<#{$&}>>#{$'}"
  else
    "no match"
  end
end
showRE('very interesting', /t/)                »  very in<<t>>eresting
showRE('Fats Waller', /ll/)                    »  Fats Wa<<ll>>er
```

The match also sets the thread-global variables `$~` and `$1` through `$9`. The variable `$~` is a `MatchData` object (described beginning on page 336) that holds everything you might want to know about the match. `$1` and so on hold the values of parts of the match. We'll talk about these later. And for people who cringe when they see these Perl-like variable names, stay tuned. There's good news at the end of the chapter.

Patterns

Every regular expression contains a pattern, which is used to match the regular expression against a string.

Within a pattern, all characters except ., |, (,), [, {, +, \, ^, $, *, and ? match themselves.

```
showRE('kangaroo', /angar/)                    »  k<<angar>>oo
showRE('!@%&-_=+', /%&/)                       »  !@<<%&>>-_=+
```

If you want to match one of these special characters literally, precede it with a backslash. This explains part of the pattern we used to split the song line,`/\s*\|\s*/`. The `\|` means ``match a vertical bar.'' Without the backslash, the ``|'' would have meant *alternation* (which we'll describe later).

```
showRE('yes | no', /\|/)                    »  yes <<|>> no
showRE('yes (no)', /\(no\)/)                »  yes <<(no)>>
showRE('are you sure?', /e\?/)              »  are you sur<<e?>>
```

A backslash followed by an alphanumeric character is used to introduce a special match construct, which we'll cover later. In addition, a regular expression may contain `#{...}` expression substitutions.

Anchors

By default, a regular expression will try to find the first match for the pattern in a string. Match `/iss/` against the string ``Mississippi,'' and it will find the substring ``iss'' starting at position one. But what if you want to force a pattern to match only at the start or end of a string?

The patterns `^` and `$` match the beginning and end of a line, respectively. These are often used to *anchor* a pattern match: for example, `/^option/` matches the word ``option'' only if it appears at the start of a line. The sequence `\A` matches the beginning of a string, and `\z` and `\Z` match the end of a string. (Actually, `\Z`matches the end of a string *unless* the string ends with a ``\n'', it which case it matches just before the ``\n''.)

```
showRE("this is\nthe time", /^the/)          »  this is\n<<the>> time
showRE("this is\nthe time", /is$/)           »  this <<is>>\nthe time
showRE("this is\nthe time", /\Athis/)        »  <<this>> is\nthe time
showRE("this is\nthe time", /\Athe/)         »  no match
```

Similarly, the patterns `\b` and `\B` match word boundaries and nonword boundaries, respectively. Word characters are letters, numbers, and underscore.

```
showRE("this is\nthe time", /\bis/)          »  this <<is>>\nthe time
showRE("this is\nthe time", /\Bis/)          »  th<<is>> is\nthe time
```

Character Classes

A character class is a set of characters between brackets: `[` *characters* `]`matches any single character between the brackets. `[aeiou]` will match a vowel,`[,.:;!?]` matches punctuation, and so on. The significance of the special regular expression characters---`.|()[{+^$*?`---is turned off inside the brackets. However, normal string substitution still occurs, so (for example) `\b` represents a backspace character and `\n` a newline (see Table 18.2 on page 203). In addition, you can use the abbreviations shown in Table 5.1 on page 59, so that (for example) `\s`matches any whitespace character, not just a literal space.

```
showRE('It costs $12.', /[aeiou]/)           »  It c<<o>>sts $12.
showRE('It costs $12.', /[\s]/)              »  It<< >>costs $12.
```

Within the brackets, the sequence c_1-c_2 represents all the characters between $c_1$and c_2, inclusive.

If you want to include the literal characters `]` and `-` within a character class, they must appear at the start.

```
a = 'Gamma [Design Patterns-page 123]'
showRE(a, /[]]/)          »  Gamma [Design Patterns-page 123<<]>>
```

```
showRE(a, /[B-F]/)        »  Gamma [<<D>>esign Patterns-page 123]
showRE(a, /[-]/)          »  Gamma [Design Patterns<<->>page 123]
showRE(a, /[0-9]/)        »  Gamma [Design Patterns-page <<1>>23]
```

Put a `^` immediately after the opening bracket to negate a character class: `[^a-z]`matches any character that isn't a lowercase alphabetic.

Some character classes are used so frequently that Ruby provides abbreviations for them. These abbreviations are listed in Table 5.1 on page 59---they may be used both within brackets and in the body of a pattern.

```
showRE('It costs $12.', /\s/)                  »  It<< >>costs $12.
showRE('It costs $12.', /\d/)                  »  It costs $<<1>>2.
```

Character class abbreviations

Sequence	As [...]	Meaning
`\d`	[0-9]	Digit character
`\D`	[^0-9]	Nondigit
`\s`	[\s\t\r\n\f]	Whitespace character
`\S`	[^\s\t\r\n\f]	Nonwhitespace character
`\w`	[A-Za-z0-9_]	Word character
`\W`	[^A-Za-z0-9_]	Nonword character

Finally, a period (``.'') appearing outside brackets represents any character except a newline (and in multiline mode it matches a newline, too).

```
a = 'It costs $12.'
showRE(a, /c.s/)                    »   It <<cos>>ts $12.
showRE(a, /./)                      »   <<I>>t costs $12.
showRE(a, /\./)                     »   It costs $12<<.>>
```

Repetition

When we specified the pattern that split the song list line, `/\s*\|\s*/`, we said we wanted to match a vertical bar surrounded by an arbitrary amount of whitespace. We now know that the `\s` sequences match a single whitespace character, so it seems likely that the asterisks somehow mean ``an arbitrary amount.'' In fact, the asterisk is one of a number of modifiers that allow you to match multiple occurrences of a pattern.

If *r* stands for the immediately preceding regular expression within a pattern, then:

r `*`	matches zero or more occurrences of *r*.
r `+`	matches one or more occurrences of *r*.
r `?`	matches zero or one occurrence of *r*.
r `{m,n}`	matches at least ``m'' and at most ``n'' occurrences of *r*.
r `{m,}`	matches at least ``m'' occurrences of *r*.

These repetition constructs have a high precedence---they bind only to the immediately preceding regular expression in the pattern. `/ab+/` matches an ``a'' followed by one or more ``b''s, not a sequence of ``ab''s. You have to be careful with the `*` construct too---the pattern /a*/ will match any string; every string has zero or more ``a''s.

These patterns are called *greedy*, because by default they will match as much of the string as they can. You can alter this behavior, and have them match the minimum, by adding a question mark suffix.

```
a = "The moon is made of cheese"
showRE(a, /\w+/)                    »  <<The>> moon is made of cheese
showRE(a, /\s.*\s/)                 »  The<< moon is made of >>cheese
showRE(a, /\s.*?\s/)                »  The<< moon >>is made of cheese
showRE(a, /[aeiou]{2,99}/)          »  The m<<oo>>n is made of cheese
showRE(a, /mo?o/)                   »  The <<moo>>n is made of cheese
```

Alternation

We know that the vertical bar is special, because our line splitting pattern had to escape it with a backslash. That's because an unescaped vertical bar ``|'' matches either the regular expression that precedes it or the regular expression that follows it.

```
a = "red ball blue sky"
showRE(a, /d|e/)                          »  r<<e>>d ball blue sky
showRE(a, /al|lu/)                        »  red b<<al>>l blue sky
showRE(a, /red ball|angry sky/)           »  <<red ball>> blue sky
```

There's a trap for the unwary here, as ``|'' has a very low precedence. The last example above matches ``red ball'' or ``angry sky'', not ``red ball sky'' or ``red angry sky''. To match ``red ball sky'' or ``red angry sky'', you'd need to override the default precedence using grouping.

Grouping

You can use parentheses to group terms within a regular expression. Everything within the group is treated as a single regular expression.

```
showRE('banana', /an*/)                        »   b<<an>>ana
showRE('banana', /(an)*/)                      »   <<>>banana
showRE('banana', /(an)+/)                      »   b<<anan>>a

a = 'red ball blue sky'
showRE(a, /blue|red/)                    »  <<red>> ball blue sky
showRE(a, /(blue|red) \w+/)              »  <<red ball>> blue sky
showRE(a, /(red|blue) \w+/)              »  <<red ball>> blue sky
showRE(a, /red|blue \w+/)                »  <<red>> ball blue sky

showRE(a, /red (ball|angry) sky/)           »  no match
a = 'the red angry sky'
```

```
showRE(a, /red (ball|angry) sky/)             »  the <<red angry sky>>
```

Parentheses are also used to collect the results of pattern matching. Ruby counts opening parentheses, and for each stores the result of the partial match between it and the corresponding closing parenthesis. You can use this partial match both within the remainder of the pattern and in your Ruby program. Within the pattern, the sequence `\1` refers to the match of the first group, `\2` the second group, and so on. Outside the pattern, the special variables `$1`, `$2`, and so on, serve the same purpose.

```
"12:50am" =~ /(\d\d):(\d\d)(..)/             »  0
"Hour is #$1, minute #$2"                    »  "Hour is 12, minute 50"
"12:50am" =~ /((\d\d):(\d\d))(..)/           »  0
"Time is #$1"                                »  "Time is 12:50"
"Hour is #$2, minute #$3"                    »  "Hour is 12, minute 50"
"AM/PM is #$4"                               »  "AM/PM is am"
```

The ability to use part of the current match later in that match allows you to look for various forms of repetition.

```
# match duplicated letter
showRE('He said "Hello"', /(\w)\1/)              »  He said "He<<ll>>o"
# match duplicated substrings
showRE('Mississippi', /(\w+)\1/)                 »  M<<ississ>>ippi
```

You can also use back references to match delimiters.

```
showRE('He said "Hello"', /(["']).*?\1/)           »  He said <<"Hello">>
showRE("He said 'Hello'", /(["']).*?\1/)           »  He said <<'Hello'>>
```

Pattern-Based Substitution

Sometimes finding a pattern in a string is good enough. If a friend challenges you to find a word that contains the letters a, b, c, d, and e in order, you could search a word list with the pattern `/a.*b.*c.*d.*e/` and find ``absconded'' and ``ambuscade.'' That has to be worth something.

However, there are times when you need to change things based on a pattern match. Let's go back to our song list file. Whoever created it entered all the artists' names in lowercase. When we display them on our jukebox's screen, they'd look better in mixed case. How can we change the first character of each word to uppercase?

The methods `String#sub` and `String#gsub` look for a portion of a string matching their first argument and replace it with their second argument. `String#sub`performs one replacement, while `String#gsub` replaces every occurrence of the match. Both routines return a new copy of the `String` containing the substitutions. Mutator versions `String#sub!` and `String#gsub!` modify the original string.

```
a = "the quick brown fox"
a.sub(/[aeiou]/,  '*')               »  "th* quick brown fox"
a.gsub(/[aeiou]/, '*')               »  "th* q**ck br*wn f*x"
a.sub(/\s\S+/,  '')                  »  "the brown fox"
a.gsub(/\s\S+/, '')                  »  "the"
```

The second argument to both functions can be either a `String` or a block. If a block is used, the block's value is substituted into the `String`.

```
a = "the quick brown fox"
a.sub(/^./) { $&.upcase }                    »  "The quick brown fox"
a.gsub(/[aeiou]/) { $&.upcase }              »  "thE qUIck brOwn fOx"
```

So, this looks like the answer to converting our artists' names. The pattern that matches the first character of a word is `\b\w`---look for a word boundary followed by a word character. Combine this with `gsub` and we can hack the artists' names.

```
def mixedCase(aName)
  aName.gsub(/\b\w/) { $&.upcase }
end
mixedCase("fats waller")                     »  "Fats Waller"
mixedCase("louis armstrong")                 »  "Louis Armstrong"
mixedCase("strength in numbers")             »  "Strength In Numbers"
```

Backslash Sequences in the Substitution

Earlier we noted that the sequences `\1`, `\2`, and so on are available in the pattern, standing for the *n*th group matched so far. The same sequences are available in the second argument of `sub` and `gsub`.

```
"fred:smith".sub(/(\w+):(\w+)/, '\2, \1')          »  "smith, fred"
"nercpyitno".gsub(/(.)(.)/, '\2\1')                »  "encryption"
```

There are additional backslash sequences that work in substitution strings: `\&`(last match), `\+` (last matched group), `\`` (string prior to match), `\'` (string after match), and `\\` (a literal backslash). It gets confusing if you want to include a literal backslash in a substitution. The obvious thing is to write

```
str.gsub(/\\/, '\\\\')
```

Clearly, this code is trying to replace each backslash in `str` with two. The programmer doubled up the backslashes in the replacement text, knowing that they'd be converted to ``\\'' in syntax analysis. However, when the substitution occurs, the regular expression engine performs another pass through the string, converting ``\\'' to ``\'', so the net effect is to replace each single backslash with another single backslash. You need to write `gsub(/\\/, '\\\\\\\\')`!

```
str = 'a\b\c'                                 »   "a\b\c"
str.gsub(/\\/, '\\\\\\\\')                    »   "a\\b\\c"
```

However, using the fact that `\&` is replaced by the matched string, you could also write

```
str = 'a\b\c'                                 »   "a\b\c"
str.gsub(/\\/, '\&\&')                        »   "a\\b\\c"
```

If you use the block form of `gsub`, the string for substitution is analyzed only once (during the syntax pass) and the result is what you intended.

```
str = 'a\b\c'                                  »   "a\b\c"
str.gsub(/\\/) { '\\\\' }                      »   "a\\b\\c"
```

Finally, as an example of the wonderful expressiveness of combining regular expressions with code blocks, consider the following code fragment from the CGI library module, written by Wakou Aoyama. The code takes a string containingHTML escape sequences and converts it into normal ASCII. Because it was written for a Japanese audience, it uses the ``n'' modifier on the regular expressions, which turns off wide-character processing. It also illustrates Ruby's `case`expression, which we discuss starting on page 81.

```
def unescapeHTML(string)
  str = string.dup
  str.gsub!(/&(.*?);/n) {
    match = $1.dup
    case match
    when /\Aamp\z/ni           then '&'
    when /\Aquot\z/ni          then '"'
    when /\Agt\z/ni            then '>'
    when /\Alt\z/ni            then '<'
    when /\A#(\d+)\z/n         then Integer($1).chr
    when /\A#x([0-9a-f]+)\z/ni then $1.hex.chr
    end
  }
  str
end


puts unescapeHTML("1&lt;2 &amp;&amp; 4&gt;3")
puts unescapeHTML("&quot;A&quot; = &#65; = &#x41;")
```

produces:

```
1<2 && 4>3
"A" = A = A
```

Object-Oriented Regular Expressions

We have to admit that while all these weird variables are very convenient to use, they aren't very object oriented, and they're certainly cryptic. And didn't we say that everything in Ruby was an object? What's gone wrong here?

Nothing, really. It's just that when Matz designed Ruby, he produced a fully object-oriented regular expression handling system. He then made it look familiar to Perl programmers by wrapping all these $-variables on top of it all. The objects and classes are still there, underneath the surface. So let's spend a while digging them out.

We've already come across one class: regular expression literals create instances of class `Regexp` (documented beginning on page 361).

```
re = /cat/
re.type                                »         Regexp
```

The method `Regexp#match` matches a regular expression against a string. If unsuccessful, the method returns `nil`. On success, it returns an instance of class`MatchData`, documented beginning on page 336. And that `MatchData` object gives you access to all available information about the match. All that good stuff that you can get from the $-variables is bundled in a handy little object.

```
re = /(\d+):(\d+)/     # match a time hh:mm
```

```
md = re.match("Time: 12:34am")
md.type                                          »   MatchData
md[0]         # == $&                            »   "12:34"
md[1]         # == $1                            »   "12"
md[2]         # == $2                            »   "34"
md.pre_match  # == $`                            »   "Time: "
md.post_match # == $'                            »   "am"
```

Because the match data is stored in its own object, you can keep the results of two or more pattern matches available at the same time, something you can't do using the $-variables. In the next example, we're matching the same `Regexp`object against two strings. Each match returns a unique `MatchData` object, which we verify by examining the two subpattern fields.

```
re = /(\d+):(\d+)/     # match a time hh:mm
md1 = re.match("Time: 12:34am")
md2 = re.match("Time: 10:30pm")
md1[1, 2]                       »     ["12", "34"]
md2[1, 2]                       »     ["10", "30"]
```

So how do the $-variables fit in? Well, after every pattern match, Ruby stores a reference to the result (`nil` or a `MatchData` object) in a thread-local variable (accessible using `$~`). All the other regular expression variables are then derived from this object. Although we can't really think of a use for the following code, it demonstrates that all the other `MatchData`-related $-variables are indeed slaved off the value in `$~`.

```
re = /(\d+):(\d+)/
md1 = re.match("Time: 12:34am")
md2 = re.match("Time: 10:30pm")
[ $1, $2 ]   # last successful match                    »  ["10", "30"]
$~ = md1
[ $1, $2 ]   # previous successful match                »  ["12", "34"]
```

Having said all this, we have to 'fess up. Andy and Dave normally use the $-variables rather than worrying about `MatchData` objects. For everyday use, they just end up being more convenient. Sometimes we just can't help being pragmatic.