Chapter 17

Character Encoding

1.9

Prior to Ruby 1.9, Ruby programs were basically written using the ASCII character encoding. You could always override this with the -K command-line option, but this led to inconsistencies when manipulating strings and doing file I/O.

Ruby 1.9 changes all this. Ruby now supports the idea of character encodings. And, what's more, these encodings can be applied relatively independently to your program source files, to objects in your running programs, and to the interpretation of I/O streams.

Before delving into the details, let's spend a few minutes thinking about why we need to separate the encodings of source files, variables, and I/O streams. Let's imagine Yui is a developer in Japan who wants to code in her native language. Her editor lets her write code using Shift JIS (which we'll call SJIS from now on), a Japanese character encoding, so she writes her variable names using katakana and kanji characters. But, by default, Ruby assumes that source files are written in ASCII, and the SJIS characters would not be recognized as such. However, by setting the encoding to be used when compiling the source file, Ruby can now parse her program.

She converts her program into a gem, and users around the world try it out. Dan, in the United States, doesn't read Japanese, so the content of her source files makes no sense to him. However, because the source files carry their encoding around with them, there's no problem; his Ruby happily compiles her code. But Dan wants to test her code against a file that contains regular old ASCII characters. That's no problem, because the file encoding is determined by Dan's locale, not by the encoding of the Ruby source. Similarly, Sophie in Paris uses the same library, but her data file is encoded in ISO-8859-1 (which is basically ASCII plus a useful subset of accented European characters in character positions above 127). Again, no problem.

But, back in Japan, Yui has a new feature to add to her library. Users want to create short PDF summaries of the data she reads, but the PDF writing library she's using supports only ISO-8859-1 characters. So, regardless of the encoding of the source code of her program and the encoding of the files she reads, she needs to be able to create strings at runtime with 8859-1 encoding. So, again, we need to be able to decouple the encoding of individual objects from the encoding of everything else.

If this sounds complex, well...it is. But the good news is that the Ruby team spent a long time thinking up ways to make it all relatively easy to use when you're writing code. In this section, we'll look at how to work with the various encodings, and I'll try to list some conventions that will make your code work in the brave new multinational world.

Encodings

At the heart of the Ruby encoding system is the new Encoding class. Objects of class Encoding each represent a different character encoding. The Encoding.list method returns a list of the built-in encodings, and the Encoding.aliases method returns a hash where the keys are aliases and the values are the corresponding base encoding. We can use these two methods to build a table of known encoding names:

Download **samples/encoding_1.rb**

```
encodings = {}
Encoding.list.each {|enc| encodings[enc.name] = [enc.name] }
Encoding.aliases.each do |alias_name, base_name|
  fail "#{base_name} #{alias_name}" unless encodings[base_name]
  encodings[base_name] << alias_name
end
names = encodings
          .values
          .sort_by {|base_name, *| base_name.downcase}
          .map do |base_name, *rest|
            if rest.empty?
              base_name
            else
             "#{base_name} (#{rest.join(', ')})"
            end
          end

puts names
```

We can see the output, wrapped into columns, in Figure 17.1 on the next page:

However, that's not the full story. Encodings in Ruby can be dynamically loaded—Ruby actually comes with more encodings than those shown in the output from this code.

Strings, regular expressions, symbols, I/O streams, and program source files are all associated with one of these encoding objects.

Encodings commonly used in Ruby programs include ASCII (7 bit characters), ASCII-8BIT,[1] UTF-8, and Shift JIS.

1. There isn't actually a character encoding called ASCII-8BIT. It's a Ruby fantasy, but a useful one. We'll talk about it shortly.

Figure 17.1. Encodings and Their Aliases

ASCII-8BIT (BINARY)	Big5 (CP950)	CP51932
CP850 (IBM850)	CP852	CP855
CP949	Emacs-Mule	EUC-JP (eucJP)
EUC-KR (eucKR)	EUC-TW (eucTW)	eucJP-ms (euc-jp-ms)
GB12345	GB18030	GB1988
GB2312 (EUC-CN, eucCN)	GBK (CP936)	IBM437 (CP437)
IBM737 (CP737)	IBM775 (CP775)	IBM852
IBM855	IBM857 (CP857)	IBM860 (CP860)
IBM861 (CP861)	IBM862 (CP862)	IBM863 (CP863)
IBM864 (CP864)	IBM865 (CP865)	IBM866 (CP866)
IBM869 (CP869)	ISO-2022-JP (ISO2022-JP)	ISO-2022-JP-2 (ISO2022-JP2)
ISO-8859-1 (ISO8859-1)	ISO-8859-10 (ISO8859-10)	ISO-8859-11 (ISO8859-11)
ISO-8859-13 (ISO8859-13)	ISO-8859-14 (ISO8859-14)	ISO-8859-15 (ISO8859-15)
ISO-8859-16 (ISO8859-16)	ISO-8859-2 (ISO8859-2)	ISO-8859-3 (ISO8859-3)
ISO-8859-4 (ISO8859-4)	ISO-8859-5 (ISO8859-5)	ISO-8859-6 (ISO8859-6)
ISO-8859-7 (ISO8859-7)	ISO-8859-8 (ISO8859-8)	ISO-8859-9 (ISO8859-9)
KOI8-R (CP878)	KOI8-U	macCentEuro
macCroatian	macCyrillic	macGreek
macIceland	MacJapanese (MacJapan)	macRoman
macRomania	macThai	macTurkish
macUkraine	Shift_JIS (SJIS)	stateless-ISO-2022-JP
TIS-620	US-ASCII (ASCII, ANSI_X3.4-1968, 646)	UTF-16BE (UCS-2BE)
UTF-16LE	UTF-32BE (UCS-4BE)	UTF-32LE (UCS-4LE)
UTF-7 (CP65000)	UTF-8 (CP65001, locale, external)	UTF8-MAC (UTF-8-MAC)
Windows-1250 (CP1250)	Windows-1251 (CP1251)	Windows-1252 (CP1252)
Windows-1253 (CP1253)	Windows-1254 (CP1254)	Windows-1255 (CP1255)
Windows-1256 (CP1256)	Windows-1257 (CP1257)	Windows-1258 (CP1258)
Windows-31J (CP932, csWindows31J)	Windows-874 (CP874)	

Source Files

First and foremost, there's a simple rule: if you only ever use 7-bit ASCII characters in your source, then the source file encoding is irrelevant. So, the simplest way to write Ruby source files that just work everywhere is to stick to boring old ASCII.

However, once a source file contains a byte whose top bit is set, you've just left the comfortable world of ASCII and entered the wild and wacky nightmare of character encodings. Here's how it works.

If your source files are not written using 7-bit ASCII, you probably want to tell Ruby about it. Because the encoding is an attribute of the source file, and not anything to do with the environment where the file is used, Ruby has a way of setting the encoding on a file-by-file basis using a new *magic comment*. If the first line of a file[2] is a comment (or the second line if the first line is a #! shebang line), Ruby scans it looking for the string coding:. If it finds it, Ruby then skips any spaces and looks for the (case-insensitive) name of an encoding. Thus, to specify that a source file is in UTF-8 encoding, you can write this:

```
# coding: utf-8
```

As Ruby is just scanning for coding:, you could also write this:

```
# encoding: ascii
```

2. Or a string passed to eval.

Emacs users might like the fact that this also works:

```
#  -*- encoding:  shift_jis -*-
```

(Your favorite editor may also support some kind of flag comment to set a file's encoding.)

If there's a shebang line, the encoding comment must be the second line of the file:

```
#!/usr/local/rubybook/bin/ruby
#  encoding:  utf-8
```

Additionally, Ruby detects any files that start with a UTF-8 byte order mark (BO). If Ruby sees the byte sequence \xEF\xBB\xBF at the start of a source file, it assumes that file is UTF-8 encoded.

The special constant __ENCODING__ returns the encoding of the current source file.

Source Elements That Have Encodings

If nothing overrides the setting, the default encoding for source is US-ASCII. This is basically the same as Ruby 1.8—you write your programs using 7-bit ASCII characters. However, unlike Ruby 1.8, if any characters with the top bit set (that is, with a character code greater than 127) do sneak into your source, Ruby will report an error, probably saying something like "invalid multibyte char." Here's an example where we typed some UTF-8 characters into a Ruby program:

```
π = 3.14159
puts "π = #{π}"
```

produces:

```
prog.rb:1: invalid multibyte char (US-ASCII)
```

The character π actually consists of the two bytes: \xcf \x80. In the default Ruby source encoding of US-ASCII, these characters raise an error because the top bit is set and Ruby doesn't know how to handle them.

We can fix that by setting the encoding:

```
# encoding: utf-8
π = 3.14159
puts "π = #{π}"
```

produces:

```
π = 3.14159
```

Note that Ruby is correctly interpreting π as a single character:

```
# encoding: utf-8
PI = "π"
puts "The size of a string containing π is #{PI.size}"
```

produces:

```
The size of a string containing π is 1
```

Now, let's get perverse. The two-byte sequence "\xcf\x8" represents π in UTF-8 but is not a valid byte sequence in the SJIS encoding. Let's see what happens if we tell Ruby that this same source file is SJIS encoded. (Remember: when we do this, we're not changing the actual bytes in the string—we're just telling Ruby to interpret them with a different set of encoding rules.)

```
# encoding: sjis
PI = "π"
puts "The size of a string containing π is #{PI.size}"
```

produces:

```
puts "The size of a string containing π is #{PI.size}"
          ^
prog.rb:2: invalid multibyte char (Shift_JIS)
prog.rb:3: syntax error, unexpected tCONSTANT, expecting $end
```

This time, Ruby complains because the file contains byte sequences that are illegal in the given encoding. And, to make matters even more confusing, the parser swallowed up the double quote after the π character, presumably while trying to build a valid SJIS character. This led to the second error message, because the word The is now interpreted as program text.

String literals are always encoded using the encoding of the source file that contains them, regardless of the content of the string:

```
# encoding: utf-8
def show_encoding(str)
  puts "'#{str}' (size #{str.size}) is #{str.encoding.name}"
end
show_encoding "cat"    # latin 'c', 'a', 't'
show_encoding "δog"    # greek delta, latin 'o', 'g'
```

produces:

```
'cat' (size 3) is UTF-8
'δog' (size 3) is UTF-8
```

Symbols and regular expression literals that contain only 7-bit characters are encoded using US-ASCII. Otherwise, they will have the encoding of the file that contains them.

```
# encoding: utf-8
def show_encoding(str)
  puts "#{str.inspect} is #{str.encoding.name}"
end
show_encoding :cat
show_encoding :δog
show_encoding /cat/
show_encoding /δog/
```

produces:

```
:cat is US-ASCII
:δog is UTF-8
/cat/ is US-ASCII
/δog/ is UTF-8
```

You can create arbitrary Unicode characters in strings and regular expressions using the \u escape. This has two forms: \uxxxx lets you encode a character using four hex digits, and \u{x... x... x...} lets you specify a variable number of characters, each with a variable number of hex digits:

```
# encoding: utf-8
"Greek pi: \u03c0"            # =>   "Greek pi: π"
"Greek pi: \u{3c0}"           # =>   "Greek pi: π"
"Greek \u{70 69 3a 20 3c0}"   # =>   "Greek pi: π"
```

Literals containing a \u sequence will always be encoded UTF-8, regardless of the source file encoding.

The String#bytes method is a convenient way to inspect the bytes in a string object. Notice that in the following code, the 16-bit codepoint is converted to a two-byte UTF-8 encoding:

```
# encoding: utf-8
"pi: \u03c0".bytes.to_a   # =>   [112, 105, 58, 32, 207, 128]
```

Eight-bit Clean Encodings

Ruby supports a virtual encoding called *ASCII-8BIT*. Despite the *ASCII* in the name, this is really intended to be used on data streams that contain binary data (hence its alias of *BINARY*). However, you can also use this as an encoding for source files. If you do, Ruby interprets all characters with codes below 128 as regular ASCII and all other characters as valid constituents of variable names. This is basically a neat hack, because it allows you to compile a file written in an encoding you don't know—the characters with the high-order bit set will be assumed to be printable.

Download **samples/encoding_15.rb**

```
# encoding: ascii-8bit
π = 3.14159
puts "π = #{π}"
puts "Size of 'π' = #{'π'.size}"
```

produces:

```
π = 3.14159
Size of 'π' = 2
```

The last line of output illustrates why ASCII-8BIT is a dangerous encoding for source files. Because it doesn't know to use UTF-8 encoding, the π character looks to Ruby like two separate characters.

Source Encoding Is Per-File

Clearly, a large application will be built from many source files. Some of these files may come from other people (possibly as libraries or gems). In these cases, you may not have control over the encoding used in a file.

Ruby supports this by allowing different encodings in the files that make up a project. Each file starts with the default encoding of US-ASCII. The file's encoding may then be set with either a coding: comment or a UTF-8 BOM.

Here's a file called iso-8859-1.rb. Notice the explicit encoding.

```
# -*- encoding: iso-8859-1 -*-
STRING_ISO = "olé"    #  \x6f \x6c \xe9
```

And here's its UTF-8 counterpart:

```
# file: utf.rb, encoding: utf-8
STRING_U = "δog"    # \xe2\x88\x82\x6f\x67
```

Now let's require both of these files into a third file. Just for the heck of it, let's declare the third file to have SJIS encoding:

```
# encoding: sjis
require 'iso-8859-1'
require 'utf'
def show_encoding(str)
  puts "'#{str}' (size #{str.size}) is #{str.encoding.name}"
end
show_encoding(STRING_ISO)
show_encoding(STRING_U)
show_encoding("cat")
```

produces:

```
'olé' (size 3) is ISO-8859-1
'δog' (size 3) is UTF-8
'cat' (size 3) is Shift_JIS
```

Notice how each file has an independent encoding. String literals in each retain their own encoding, even when used in a different file. All the encoding directive does is tell Ruby how to interpret the characters in the file and what encoding to use on literal strings and regular expressions containing non-ASCII characters. Ruby will never change the actual bytes in a source file when reading them in.

Transcoding

As we've already seen, strings, symbols, and regular expressions are now labeled with their encoding. You can convert a string from one encoding to another using the String#encode method. For example, we can convert the word *olé* from UTF-8 to ISO-8859-1:

```
# encoding: utf-8
ole_in_utf = "olé"
ole_in_utf.encoding      # =>   #<Encoding:UTF-8>
ole_in_utf.bytes.to_a    # =>   [111, 108, 195, 169]

ole_in_8859 = ole_in_utf.encode("iso-8859-1")
ole_in_8859.encoding     # =>   #<Encoding:ISO-8859-1>
ole_in_8859.bytes.to_a   # =>   [111, 108, 233]
```

You have to be careful when using encode—if the target encoding doesn't contain characters that appear in your source string, Ruby will throw an exception. For example, the π character is available in UTF-8 but not in ISO-8859-1:

```
# encoding: utf-8
pi = "pi = π"
pi.encode("iso-8859-1")
```

produces:

```
prog.rb:3:in `encode': "\xCF\x80" from UTF-8 to ISO-8859-1 (Encoding::UndefinedConversionErr
from /tmp/prog.rb:3:in `<main>'
```

You can, however, override this behavior, for example supplying a placeholder character to use when no direct translation is possible. (See the description of String#encode on page 678 for more details.)

```
# encoding: utf-8
pi = "pi = π"
puts pi.encode("iso-8859-1", undef: :replace, replace: "??")
```

produces:

```
pi = ??
```

Sometimes you'll have a string containing binary data and you want that data to be interpreted as if it had a particular encoding. You can't use the encode method for this, because you don't want to change the byte contents of the string—you're just changing the encoding associated with those bytes. Use the String#force_encoding method to do this:

Download **samples/encoding_22.rb**

```
# encoding: ascii-8bit
str = "\xc3\xa9"     # e-acute in UTF-8
str.encoding     # =>   #<Encoding:ASCII-8BIT>
str.force_encoding("utf-8")
str.bytes.to_a   # =>   [195, 169]
str.encoding     # =>   #<Encoding:UTF-8>
```

Finally, you can use encode (with two parameters) to convert between two encodings if your source string is ASCII-8BIT. This might happen if, for example, you're reading data in binary mode from a file and choose not to encode it at the time you read it. Here we fake that out by creating an ASCII-8BIT string that contains an ISO-8859-1 sequence (our old friend *olé*). We then convert the string to UTF-8. To do this, we have to tell encode the actual encoding of the bytes by passing it a second parameter:

Download **samples/encoding_23.rb**

```
# encoding: ascii-8bit
original = "ol\xe9"     # e-acute in ISO-8859-1
original.bytes.to_a   # =>   [111, 108, 233]
original.encoding     # =>   #<Encoding:ASCII-8BIT>
new = original.encode("utf-8", "iso-8859-1")
new.bytes.to_a        # =>   [111, 108, 195, 169]
new.encoding          # =>   #<Encoding:UTF-8>
```

If you're writing programs that will support multiple encodings, you probably want to read the section on *Default Internal Encoding* on page 274—it will greatly simplify your life.

Input and Output Encoding

Playing around with encodings within a program is all very well, but in most code we'll want to read data from and write data to external files. And, often, that data will be in a particular encoding.

Ruby's I/O objects support both encoding and transcoding of data. What does this mean?

Every I/O object has an associated external encoding. This is the encoding of the data being read from or written to the outside world. Through a piece of magic I'll describe on page 274, all Ruby programs run with the concept of a default external encoding. This is the external encoding that will be used by I/O objects unless you override it when you create the object (for example, by opening a file).

Now, your program may want to operate internally in a different encoding. For example, some of my files may be encoded with ISO-8859-1, but I want my Ruby program to work internally using UTF-8. Ruby I/O objects manage this by having an optional associated *internal encoding*. If set, then input will be transcoded from the external to the internal encodings on read operations, and output will be transcoded from internal to external encoding on write operations.

Let's start with the simple cases. On my OS X box, the default external encoding is UTF-8. If I don't override it, all my file I/O will therefore also be in UTF-8. I can query the external encoding of an I/O object using the IO#external_encoding method:

```
f = File.open("/etc/passwd")
puts "File encoding is #{f.external_encoding}"
line = f.gets
puts "Data encoding is #{line.encoding}"
```

produces:

```
File encoding is UTF-8
Data encoding is UTF-8
```

Notice that the data is tagged with a UTF-8 encoding even though it (presumably) contains just 7-bit ASCII characters. Only literals in your Ruby source files have the "change encoding if they contain 8-bit data" rule.

You can force the external encoding associated with an I/O object when you open it—simply add the name of the encoding, preceded by a colon, to the mode string. Note that this in no way changes the data that's read—it simply tags it with the encoding you specify:

```
f = File.open("/etc/passwd", "r:ascii")
puts "File encoding is #{f.external_encoding}"
line = f.gets
puts "Data encoding is #{line.encoding}"
```

produces:

```
File encoding is US-ASCII
Data encoding is US-ASCII
```

You can force Ruby to transcode—change the encoding—of data it reads and writes by putting two encoding names in the mode string, again with a colon before each. For example, the file `iso-8859-1.txt` contains the word *olé* in ISO-8859-1 encoding. In this encoding, the e-acute character is encoded by the single byte \xe9. I can view this file's contents in hex using the od command-line tool. (Windows users can use the d command in debug to do the same.)

```
% od -t x1 iso-8859-1.txt
0000000    6f  6c  e9  0a
0000004
```

If we try to read it with our default external encoding of UTF-8, we'll encounter a problem:

```
f = File.open("iso-8859-1.txt")
puts f.external_encoding.name
line = f.gets
puts line.encoding
puts line
```

produces:

```
UTF-8
UTF-8
ol?
```

The problem is that the binary sequence for the e-acute isn't the same in ISO-8859-1 and UTF-8. Ruby just assumed the file contained UTF-8 characters, tagging the string it read accordingly.

We can tell the program that the file contains ISO-8859-1:

```
f = File.open("iso-8859-1.txt", "r:iso-8859-1")
puts f.external_encoding.name
line = f.gets
puts line.encoding
puts line
```

produces:

```
ISO-8859-1
ISO-8859-1
ol?
```

This doesn't help us much. The string is now tagged with the correct encoding, but our operating system is still expecting UTF-8 output.

The solution is to map the ISO-8859-1 to UTF-8 on input:

```
f = File.open("iso-8859-1.txt", "r:iso-8859-1:utf-8")
puts f.external_encoding.name
line = f.gets
puts line.encoding
puts line
```

produces:

```
ISO-8859-1
UTF-8
olé
```

If you specify two encoding names when opening an I/O object, the first is the external encoding, and the second is the internal encoding. Data is transcoded from the former to the latter on reading and the opposite way on writing. That's how I created the file containing *olé* in the first place:

% ruby -e 'File.open("iso-8859-1.txt", "w:iso-8859-1:utf-8") { |f| f.puts "olé"}'

Binary Files

In the old days, we Unix users used to make little snide comments about the way that Windows users had to open binary files using a special binary mode. Well, now the Windows folks can get their own back. If you want to open a file containing binary data in Ruby, you must now specify the binary flag, which will automatically select the 8-bit clean ASCII-8BIT encoding. To make things explicit, you can use "binary" as an alias for the encoding:

Download **samples/encoding_31.rb**

```
f = File.open("iso-8859-1.txt", "rb")
puts "Implicit encoding is #{f.external_encoding.name}"
f = File.open("iso-8859-1.txt", "rb:binary")
puts "Explicit encoding is #{f.external_encoding.name}"
line = f.gets
puts "String encoding is #{line.encoding.name}"
```

produces:

```
Implicit encoding is ASCII-8BIT
Explicit encoding is ASCII-8BIT
String encoding is ASCII-8BIT
```

Default External Encoding

If you look at the text files on your computer, the chances are that they'll all use the same encoding. In the United States, that'll probably be UTF-8 or ASCII. In Europe, it might be UTF-8 or ISO-8859-x. If you use a Windows box, you may be using a different set of encodings (use the console chcp command to find your current code page). But whatever encoding you use, the chances are good that you'll stick with it for the majority of your work.

On Unix-like boxes, you'll probably find you have the LANG environment variable set. On my OS X box, I have this:

```
% echo $LANG
en_US.UTF-8
```

This says that I'm using the English language in the U.S. territory and my default codeset is UTF-8. On startup, Ruby looks for this environment variable and, if present, sets the default external encoding from the codeset component. Thus, on my box, Ruby 1.9 programs run with a default external encoding of UTF-8. If instead I were in Japan and my LANG variable were set to ja_JP.sjis, my encoding would be set to Shift JIS. We can look at the default external encoding by querying the Encoding class. While we're at it, we'll experiment with different values in the LANG environment variable:

```
% echo $LANG
en_US.UTF-8
% ruby -e 'p Encoding.default_external.name'
"UTF-8"
% LANG=ja_JP.sjis ruby -e 'p Encoding.default_external.name'
"Shift_JIS"
% LANG= ruby -e 'p Encoding.default_external.name'
"US-ASCII"
```

The encoding set from the environment *does not* affect the encoding Ruby uses for source files—it affects only the encoding of data read and written by your programs.

Finally, you can use the -E command-line option (or the long-form --encoding) to set the default external encoding of your I/O objects:

```
% ruby -E utf-8 -e 'p Encoding.default_external.name'
"UTF-8"
% ruby -E sjis -e 'p Encoding.default_external.name'
"Shift_JIS"
% ruby -E sjis:iso-8859-1 -e 'p Encoding.default_internal.name'
"ISO-8859-1"
```

Encoding Compatibility

Before Ruby performs operations involving strings or regular expressions, it first has to check that the operation makes sense. For example, it is valid to perform an equality test between two strings with different encodings, but it is not valid to append one to the other.

The basic steps in this checking are as follows:

1. If the two objects have the same encoding, the operation is valid.
2. If the two objects each contain only 7-bit characters, the operation is permitted regardless of the encodings.
3. If the encodings in the two objects are compatible (which we'll discuss next), the operation is permitted.
4. Otherwise, an exception is raised.

Let's say you have a set of text files containing markup. In some of the files, authors used the sequence \dots to represent an ellipsis. In other files, which have UTF-8 encoding, authors used an actual ellipsis character (\u2026). We want to convert both forms to three periods.

We can start off with a simplistic solution:

```
# encoding: utf-8
while line = gets
  result = line.gsub(/\\dots/, "...")
                .gsub(/…/, "...") # unicode ellipsis
  puts result
end
```

In my environment, the content of files is by default assumed to be UTF-8. Feed our code ASCII files and UTF-encoded files, and it works just fine. But what happens when we feed it a file that contains ISO-8859-1 characters?

```
dots.rb:4:in `gsub': broken UTF-8 string (ArgumentError)
```

Ruby tried to interpret the input text, which is ISO-8859-1 encoded, as UTF-8. Because the byte sequences in the file aren't valid UTF, it failed.

There are three solutions to this problem. The first is to say that it makes no sense to feed files with both ISO-8859 and UTF-8 encoding to the same program without somehow differentiating them. That's perfectly true. This approach means we'll need some command-line options, liberal use of force_encoding, and probably some kind of code to delegate the pattern matching to different sets of patterns depending on the encoding of each file.

A second hack is to simply treat both the data and the program as ASCII-8BIT and perform all the comparisons based on the underlying bytes. This isn't particularly reliable, but it might work in some circumstances.

The third solution is to choose a master encoding and to transcode strings into it before doing the matches. Ruby provides built-in support for this with the default_internal encoding mechanism.

Default Internal Encoding

By default, Ruby performs no automatic transcoding when reading and writing data. However, two command-line options allow you to change this.

We've already seen the -E option, which sets the default encoding applied to the content of external files. When you say -E xxx, the default external encoding is set to xxx.

However, -E takes a second option. In the same way that you can give File#open both an external and an internal encoding, you can also set a default internal encoding using the folllowing option:

-E external:internal

Thus, if all your files are written with ISO-8859-1 encoding but you want your program to have to deal with their content as if it were UTF-8, you can use this:

ruby -E iso-8859-1:utf-8

You can specify just an internal encoding by omitting the external option but leaving the colon:

```
ruby -E :utf-8
```

Indeed, because UTF-8 is probably the best of the available transcoding targets, Ruby has the -U command-line option, which sets the internal encoding to UTF-8.

You can query the default internal encoding in your code with the Encoding.default_internal method. This returns nil if no default internal encoding has been set.

One last note before we leave this section: if you compare two strings with different encodings, Ruby does not normalize them. Thus, "é" tagged with a UTF-8 encoding will not compare equal to "é" tagged with ISO-8859-1, because the underlying bytes are different.

Fun with Unicode

As Daniel Berger pointed out,[3] the fact that UTF-8 is now supported in Ruby means that we can do interesting things with our method and variable names:

Download **samples/encoding_36.rb**

```
# encoding: utf-8
def Σ(*args)
  args.inject(:+)
end
puts Σ 1, 3, 5, 9
```

produces:

```
18
```

Of course, this way can lead to some pretty obscure and hard-to-use code. (For example, is the summation character in the previous code a real summation, \u2211, or a Greek sigma, \u03a3?) Just because we *can* do something doesn't mean we necessarily *should*....

3. http://www.oreillynet.com/ruby/blog/2007/10/fun_with_unicode_1.html