APPENDIX

Ruby Execution

As promised in Chapter 2, this appendix takes time to introduce the general execution environment of a Ruby script. It covers the various command line options available with the Ruby interpreter as well as the many pertinent variables and constants that make very short scripts possible.

Command Line Options

Perl users should feel right at home with many of the flags that can be passed to the Ruby interpreter, as the flags were modeled very much on those used by Perl. It is my opinion that Perl's biggest strength lies in its one-liners that, while often baffling to the newcomer, provide immense power to construct unbelievably dense code. This facility stems from two things: command line modes that do very common things for you and shorthand variables for accessing/setting pertinent data and behavior.

The fact that Ruby respects the success of Perl's approach in this area often leads to the charge that Ruby is too Perl-like. This is roughly equivalent to saying a human is too much like a turkey because they've both recognized the advantages of bipedal locomotion.

With that said, I can now present a list of command line options that is complete save for four: `--copyright`, `--version`, `--help` (`-h`), and `--yydebug` (`-y`). I'm going to assume that the first three hold no mystery. The fourth basically turns on a load of debugging for the interpreter itself and is of no use unless you plan to debug Ruby.

Octal Record Separator Specification: -0[octal]

Consists of the digit 0 optionally followed by an octal value for the character to use as the default global record separator `$/`, which is used by methods like `IO.each`. Without an explicit value, `NULL` (`000`) is assumed. Passed a value of zero (`-00`), Ruby goes into paragraph mode, where each record is considered to be separated by two default record separators (e.g., `"\n\n"`). Since `511` is not a valid character code, passing `-0777` will guarantee that the entire input file is read in as one record—a process somewhat indelicately referred to as *slurping*.

Auto Split Mode: -a

When used in conjunction with `-n` or `-p`, runs `$F = $_.split` at the beginning of each loop.

Change Directory: -C path

Causes the present working directory to be set to the specified `path` before the script is run.

Check Syntax Only: -c

Does exactly what you think it does (without running the script). Remember that the more sophisticated your scripts become, the less useful this flag will be, as it only invokes a simple syntax check. If all is well, you will receive the message `Syntax OK`. Otherwise, you will see one or more reports indicating a problem and the line it occurred on like this:

```
upload.rb:19: odd number list for Hash
upload.rb:28: syntax error
```

Debug Mode: -d, --debug

Sets `$DEBUG` to `true`, which is generally useful if you want a simple way to enable some extra tracing or differing behaviors during development. This flag also implicitly sets `$VERBOSE = true`.

Execute Line: -e line

Takes a `line` of Ruby and executes it (as you've already seen). This can be specified multiple times, in which case each instance is treated as a new separate line for bookkeeping purposes (try `ruby -e 'p __LINE__' -e 'p __LINE__'`).

Field Separator Specification: -F pattern

Sets the global field separator `$;`, which is used by methods like `String.split`. Note that this can be a simple string delimiter or a full-blown regular expression.

Include from Additional Directory: -I path(s)

Adds to the array of search paths for the inclusion of code via the `load` or `require` statement. This array is accessible in code as `$:` (`$LOAD_PATH`). Note that many paths can be specified per `-I` and that many `-I`s can be used.

In-place Editing: -i[extension]

For each file path specified as an argument to the interpreter (in `ARGV`), anything written to standard out while dealing with a particular file becomes the contents of that particular file. The original file is preserved with a new `extension` if and only if this is specified. Obviously, this flag is pretty meaningless without the use of `-n` or `-p`. Suppose I wanted to change all instances of my name in a set of header files to my secret squirrel name. I would write something like `ruby -p -i.bak -e 'gsub(/Andre/, "Nutkin")' *.h`.

KCode Encoding Choice: -K encoding

Specifies the `encoding` of the script to support character sets like SJIS as used by our Japanese colleagues. This is equivalent to setting the `$KCODE` variable. An `encoding` of `u` specifies UTF-8.

Line Ending Automated Handling: -l

Sets the output record separator (`$\`) to the same value as the input record separator (`$/`) and automatically performs a `chop!` on each input record (`$_`). In simpler terms, this flag causes each record to be presented to you without its line ending or other record separator, and then adds it back once you've finished.

Nested Retrieval Loop: -n

Places an implicit `while gets ... end` block around your code. Note that this allows you to read each line of not only a single file, but multiple files. It is well worth having a close look at `ri Kernel.gets` for the rather cute behavior conventions it provides. In particular, it sets `$_` to the value of the retrieved record (e.g., the next line).

Printed Nested Retrieval Loop: -p

Does the same as `-n`, but with an extra `print $_` at the end of each loop.

Require Library: -r library

Identical to having placed `require library` at the beginning of your code. This is particularly useful in `-n`/`-p` situations where you don't want to `require` the `library` through every loop (although this wouldn't be disastrous, as rerequiring a library involves only a single hash lookup).

Script Search: -S

Instead of assuming a given script is in the current directory, causes the interpreter to search in the location specified in the `PATH` environment variable. This is mostly to allow platforms that don't support the shebang-style script header (silly as they are) to emulate it.

Smart Variables: -s

Magically removes every command flag after the script name but before any file arguments from `ARGV` and turns them into global variables. So `ruby -s test.rb -col=blue file.c` would run the `test.rb` script with a single argument (`file.c`) and give the script the global variable `$col = "blue"`.

Taint Level: -T[level]

Turns on taint checking at the given `level` (or `1` if none is specified) and sets `$SAFE` to this value.

Verbose Mode: -v, --verbose

Activates all kinds of extra messages and warnings, sets `$VERBOSE = true`, and causes the version string to be printed before anything else is done (thus making a simple `ruby -v` do what you would expect).

Warnings Mode: -w

Does the same as `-v`, except that no version string is printed and the interpreter will expect input on standard in if no other code/scripts are specified.

Ignore Garbage: -x[path]

The interpreter will ignore everything it encounters until it sees a shebang line containing the string `ruby`, which it will interpret as the beginning of the script. It will then execute the script until it encounters an end-of-file or the keyword `__END__`. If a `path` is specified, the present working directory is changed to this `path` before execution commences.

Environment

Ruby provides access to its runtime environment through a set of variables that, again, should be familiar to any Perl user. First, there are the set of arguments to the script whereby the invocation

```
$ ruby count.rb 1 2 3
```

would give us

```
ARGV = ["1", "2", "3"]
```

inside `count.rb`. `ARGV` is just an `Array` and so supports all the methods you have/will come to love. Imagine we wanted our script to be able to take some command flags, one of which was a `-r` to remove something or other. One quick and dirty way to determine whether this had been passed is

```
remove_stuff = ARGV.include?("-r")
```

In practice, beyond the simplest command line options, a proper argument-parsing library tends to be a better idea.

Next on the list of most-used variables would have to be the environment itself—that dictionary of values you can see from the console by typing `env` (or something similar). Here's what part of mine looks like:

```
TERM_PROGRAM=Apple_Terminal
TERM=xterm-color
SHELL=/bin/bash
TERM_PROGRAM_VERSION=133
USER=andre
```

Perhaps unsurprisingly, this is accessed in Ruby using the `ENV` object (which acts like a `Hash` but isn't one). Values can be retrieved and set as you would expect:

```
terminal_type = ENV["TERM"]
ENV["USER"] = "Nutkin"
```

Beyond these two oft-used collections, a few global variables and a couple more constants are of interest.

Process Information

The following entities all relate to process-specific information such as the name of the command used to run the script and global variables for each of the command flags:

- `$0`: The command that originally ran the Ruby script (what C users would think of as the first argument of `ARGV`). On many platforms, this is writable so that how the process appears in `top`/`ps` can be changed by assigning something new to `$0`.
- `$$`: The process ID (`PID`) of the currently executing program.
- `$:`: The array of directories to check when including code via the `require` or `load` command. This array provides a programmatic alternative to using the `-I` flag on the command line to include new search paths. Append your chosen directory/directories as you would with any other array: `$: << "/tmp/foo"` or `$: += ["/tmp/foo", "/tmp/bar"]`.
- `$-<something>`: Almost all of the command flags have a corresponding variable of this form—for example, `$-a`, which corresponds to the value passed by the `-a` command flag, or `$-d`, which is `true` in debug mode.
- `__FILE__`: The name of the currently executing file.
- `__LINE__`: The number of the line that is currently executing.

File Handles

These variables specify the input and output streams used by your program:

- `STDIN`, `STDOUT`, `STDERR`: The actual standard file handles (of class `IO`) for the program and the initial values of `$stdin`, `$stdout`, and `$stderr`
- `$stdin`, `$stdout`, `$stderr`: The current `IO` objects for the standard file handles used by methods like `Kernel.puts`

Magic Processing Variables

This subsection lists global variables controlling some of the more "magical" field, file, and record processing behavior we unpacked in Chapter 7:

- `$/`: The input record separator used by methods like `Kernel.gets`.
- `$\`: The output record separator used by methods like `Kernel.puts`.
- `$;`: The input field separator (a simple string delimiter or a regular expression).
- `$,`: The output field separator used by methods like `Array.join` (defaults to `nil`).
- `$<`, `ARGF`: A special object that provides access to all of the lines of all of the files specified in `ARGV` (or the data from `$stdin` when no such files are specified). For example, `$<.file` returns an `IO` object for the current file being read, and `$FILENAME` is a synonym for `$<.filename`.

- `$_`: The last line read by `Kernel.gets` or `Kernel.readline`. Indeed, a number of the string manipulation functions in Kernel (such as `gsub`) operate on `$_`.
- `$F`: The array of fields as split under the `-a` command flag using `$;` to determine the field separator.

Other Sundries

Finally, here are some other variables (and a constant) that you might find useful in day-to-day scripting:

- `DATA`: The contents of the main program file past an `__END__` directive (if such exists)
- `$1` to `$9`: The up to nine matches from the last executed regular expression (local to the current scope)
- `$!`: The last exception object raised in the current context
- `$@`: The last exception object's backtrace

Improving Readability

Looking back over the preceding subsections, I see that I used to use a lot of these tersely named global variables (or their equivalents) when I was a Perl monkey. I'm pretty sure I've never employed most of these as a Ruby-holic, mostly because I found myself in a job where one-liners were never one-offs. Even trivial bits of code ended up getting reused, and the ease with which one can write things "properly" the first time in Ruby meant that beasts like `$<` never really had a chance.

If you absolutely insist on using these global variables in scripts, then you'd be well advised to check out the `English` library, which ships as part of the standard distribution. It maps many of the symbols to more intelligible variable names and can be bolted on to your script with a quick `require "English"`. Mappings include `$!` to `$ERROR_INFO`, `$;` to `$FIELD_SEPARATOR`, and `$/` to `$INPUT_RECORD_SEPARATOR`.

INDEX

B

C

D

E

F

G

H

I

J

K

L

M

N

P

Q

R

S

T

U

V

W

X

Y

Z