Table of Contents

Date and Time..... 1
Finding Today's Date..... 3
Parsing Dates, Precisely or Fuzzily..... 7
Printing a Date..... 10
Iterating Over Dates..... 14
Doing Date Arithmetic..... 16
Counting the Days Since an Arbitrary Date..... 18
Converting Between Time Zones..... 20
Checking Whether Daylight Saving Time Is in Effect..... 22
Converting Between Time and DateTime Objects..... 24
Finding the Day of the Week..... 27
Handling Commercial Dates..... 28
Running a Code Block Periodically..... 29
Waiting a Certain Amount of Time..... 31
Adding a Timeout to a Long-Running Operation..... 34

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

3. Date and Time

With no concept of time, our lives would be a mess. Without software programs to constantly manage and record this bizarre aspect of our universe...well, we might actually be better off. But why take the risk?

Some programs manage real-world time on behalf of the people who'd otherwise have to do it themselves: calendars, schedules, and data gatherers for scientific experiments. Other programs use the human concept of time for their own purposes: they may run experiments of their own, making decisions based on microsecond variations. Objects that have nothing to do with time are sometimes given timestamps recording when they were created or last modified. Of the basic data types, a time is the only one that directly corresponds to something in the real world.

Ruby supports the date and time interfaces you might be used to from other programming languages, but on top of them are Ruby-specific idioms that make programming easier. In this chapter, we'll show you how to use those interfaces and idioms, and how to fill in the gaps left by the language as it comes out of the box.

Ruby actually has two different time implementations. There's a set of time libraries written in C that have been around for decades. Like most modern programming languages, Ruby provides a native interface to these C libraries. The libraries are powerful, useful, and reliable, but they also have some significant shortcomings, so Ruby compensates with a second time library written in pure Ruby. The pure Ruby library isn't used for everything because it's slower than the C interface, and it lacks some of the features buried deep in the C library, such as the management of Daylight Saving Time.

The `Time` class contains Ruby's interface to the C libraries, and it's all you need for most applications. The `Time` class has a lot of Ruby idiom attached to it, but most of its methods have strange unRuby-like names like `strftime` and `strptime`. This is for the benefit of people who are already used to the C library, or one of its other interfaces (like Perl or Python's).

The internal representation of a `Time` object is a number of seconds before or since "time zero." Time zero for Ruby is the Unix epoch: the first second GMT of January 1, 1970. You can get the current local time with `Time.now`, or create a `Time` object from seconds-since-epoch with `Time.at`.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
Time.now                                # => Sat Mar 18 14:49:30 EST 2006
Time.at(0)                              # => Wed Dec 31 19:00:00 EST 1969
```

This numeric internal representation of the time isn't very useful as a human-readable representation. You can get a string representation of a `Time`, as seen above, or call accessor methods to split up an instant of time according to how humans reckon time:

```
t = Time.at(0)
t.sec                                   # => 0
t.min                                   # => 0
t.hour                                  # => 19
t.day                                   # => 31
t.month                                 # => 12
t.year                                  # => 1969
t.wday                                  # => 3      # Numeric day of week; Sunday
is 0
t.yday                                  # => 365    # Numeric day of year
t.isdst                                 # => false  # Is Daylight Saving Time in
                                        # effect?
t.zone                                  # => "EST"  # Time zone
```

See Recipe 3.3 for more human-readable ways of slicing and dicing `Time` objects.

Apart from the awkward method and member names, the biggest shortcoming of the `Time` class is that on a 32-bit system, its underlying implementation can't handle dates before December 1901 or after January 2037.[1]

[1] A system with a 64-bit `time_t` can represent a much wider range of times (about half a trillion years):

```
Time.local(1865,4,9)                    # => Sun Apr 09 00:00:00 EWT 1865
Time.local(2100,1,1)                    # => Fri Jan 01 00:00:00 EST 2100
```

You'll still get into trouble with older times, though, because `Time` doesn't handle calendrical reform. It'll also give time zones to times that predate the creation of time zones (EWT stands for Eastern War Time, an American timezone used during World War II).

```
Time.local(1865, 4, 9)
# ArgumentError: time out of range
Time.local(2100, 1, 1)
# ArgumentError: time out of range
```

To represent those times, you'll need to turn to Ruby's other time implementation: the `Date` and `DateTime` classes. You can probably use `DateTime` for everything, and not use `Date` at all:

```
require 'date'
DateTime.new(1865, 4, 9).to_s           # => "1865-04-09T00:00:00Z"
DateTime.new(2100, 1, 1).to_s           # => "2100-01-01T00:00:00Z"
```

Recall that a `Time` object is stored as a fractional number of seconds since a "time zero" in 1970. The internal representation of a `Date` or `DateTime` object is a astronomical Julian date: a fractional number of *days* since a "time zero" in 4712 BCE, over 6,000 years ago.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
# Time zero for the date library:
DateTime.new.to_s                       # => "-4712-01-01T00:00:00Z"

# The current date and time:
DateTime::now.to_s                       # => "2006-03-18T14:53:18-0500"
```

A `DateTime` object can precisely represent a time further in the past than the universe is old, or further in the future than the predicted lifetime of the universe. When `DateTime` handles historical dates, it needs to take into account the calendar reform movements that swept the Western world throughout the last 500 years. See Recipe 3.1 for more information on creating `Date` and `DateTime` objects.

Clearly `DateTime` is superior to `Time` for astronomical and historical applications, but you can use `Time` for most everyday programs. This table should give you a picture of the relative advantages of `Time` objects and `DateTime` objects.

Table 3-1.

	Time	DateTime
Date range	1901–2037 on 32-bit systems	Effectively infinite
Handles Daylight Saving Time	Yes	No
Handles calendar reform	No	Yes
Time zone conversion	Easy with the tz gem	Difficult unless you only work with time zone offsets
Common time formats like RFC822	Built-in	Write them yourself
Speed	Faster	Slower

Both `Time` and `DateTime` objects support niceties like iteration and date arithmetic: you can basically treat them like numbers, because they're stored as numbers internally. But recall that a `Time` object is stored as a number of seconds, while a `DateTime` object is stored as a number of days, so the same operations will operate on different time scales on `Time` and `DateTime` objects. See Recipes 3.4 and 3.5 for more on this.

So far, we've talked about writing code to manage specific moments in time: a moment in the past or future, or right now. The other use of time is duration, the relationship between two times: "start" and "end," "before" and "after." You can measure duration by subtracting one `DateTime` object from another, or one `Time` object from another: you'll get a result measured in days or seconds (see Recipe 3.5). If you want your program to actually *experience* duration (the difference between now and a time in the future), you can put a thread to sleep for a certain amount of time: see Recipes 3.12 and 3.13.

You'll need duration most often, perhaps, during development. Benchmarking and profiling can measure how long your program took to run, and which parts of it took the longest. These topics are covered in Chapter 17: see Recipes 17.12 and 17.13.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Recipe 3.1. Finding Today's Date

Problem

You need to create an object that represents the current date and time, or a time in the future or past.

Solution

The factory method `Time.now` creates a `Time` object containing the current local time. If you want, you can then convert it to GMT time by calling `Time#gmtime`. The `gmtime` method actually modifies the underlying time object, though it doesn't follow the Ruby naming conventions for such methods (it should be called something like `gmtime`!).

```
now = Time.now                    # => Sat Mar 18 16:58:07 EST 2006
now.gmtime                        # => Sat Mar 18 21:58:07 UTC 2006

#The original object was affected by the time zone conversion.
now                               # => Sat Mar 18 21:58:07 UTC 2006
```

To create a `DateTime` object for the current local time, use the factory method `DateTime.now`. Convert a `DateTime` object to GMT by calling `DateTime#new_offset` with no argument. Unlike `Time#gmtime`, this method returns a second `DateTime` object instead of modifying the original in place.

```
require 'date'
now = DateTime.now
# => #<DateTime: 70669826362347677/28800000000,-5/24,2299161>
now.to_s                          # => "2006-03-18T16:58:07-0500"
now.new_offset.to_s               # => "2006-03-18T21:58:07Z"

#The original object was not affected by the time zone conversion.
now.to_s                          # => "2006-03-18T16:58:07-0500"
```

Discussion

Both `Time` and `DateTime` objects provide accessor methods for the basic ways in which the Western calendar and clock divide a moment in time. Both classes provide `year`, `month`, `day`, `hour` (in 24-hour format), `min`, `sec`, and `zone` accessors. `Time#isdst` lets you know if the underlying time of a `Time` object has been modified by Daylight Saving Time in its time zone. `DateTime` pretends Daylight Saving Time doesn't exist.

```
now_time = Time.new
now_datetime = DateTime.now
now_time.year                      # => 2006
now_datetime.year                  # => 2006
now_time.hour                      # => 18
now_datetime.hour                  # => 18

now_time.zone                      # => "EST"
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
now_datetime.zone                  # => "-0500"
now_time.isdst                     # => false
```

You can see that `Time#zone` and `DateTime#zone` are a little different. `Time#zone` returns a time zone name or abbreviation, and `DateTime#zone` returns a numeric offset from GMT in string form. You can call `DateTime#offset` to get the GMT offset as a number: a fraction of a day.

```
now_datetime.offset                # => Rational(-5, 24) # -5 hours
```

Both classes can also represent fractions of a second, accessible with `Time#usec` (that is, μsec or microseconds) and `DateTime#sec_fraction`. In the example above, the `DateTime` object was created after the `Time` object, so the numbers are different even though both objects were created within the same second.

```
now_time.usec                      # => 247930
# That is, 247930 microseconds
now_datetime.sec_fraction          # => Rational(62191, 21600000000)
# That is, about 287921 microseconds
```

The `date` library provides a `Date` class that is like a `DateTime`, without the time. To create a `Date` object containing the current date, the best strategy is to create a `DateTime` object and use the result in a call to a `Date` factory method. `DateTime` is actually a subclass of `Date`, so you only need to do this if you want to strip time data to make sure it doesn't get used.

```
class Date
  def Date.now
    return Date.jd(DateTime.now.jd)
  end
end
puts Date.now
# 2006-03-18
```

In addition to creating a time object for *this very moment*, you can create one from a string (see Recipe 3.2) or from another time object (see Recipe 3.5). You can also use factory methods to create a time object from its calendar and clock parts: the year, month, day, and so on.

The factory methods `Time.local` and `Time.gm` take arguments `Time` object for that time. For local time, use `Time.local`; for GMT, use `Time.gm`. All arguments after year are optional and default to zero.

```
Time.local(1999, 12, 31, 23, 21, 5, 1044)
# => Fri Dec 31 23:21:05 EST 1999

Time.gm(1999, 12, 31, 23, 21, 5, 22, 1044)
# => Fri Dec 31 23:21:05 UTC 1999
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
Time.local(1991, 10, 1)
# => Tue Oct 01 00:00:00 EDT 1991

Time.gm(2000)
# => Sat Jan 01 00:00:00 UTC 2000
```

The `DateTime` equivalent of `Time.local` is the `civil` factory method. It takes almost but not quite the same arguments as `Time.local`:

```
[year, month, day, hour, minute, second, timezone_offset, date_of_calendar_reform].
```

The main differences from `Time.local` and `Time.gmt` are:

- There's no separate `usec` argument for fractions of a second. You can represent fractions of a second by passing in a rational number for `second`.
- All the arguments are optional. However, the default year is 4712 BCE, which is probably not useful to you.
- Rather than providing different methods for different time zones, you must pass in an offset from GMT as a fraction of a day. The default is zero, which means that calling `DateTime.civil` with no time zone will give you a time in GMT.

```
DateTime.civil(1999, 12, 31, 23, 21, Rational(51044, 100000)).to_s
# => "1999-12-31T23:21:00Z"

DateTime.civil(1991, 10, 1).to_s
# => "1991-10-01T00:00:00Z"

DateTime.civil(2000).to_s
# => "2000-01-01T00:00:00Z"
```

The simplest way to get the GMT offset for your local time zone is to call `offset` on the result of `DateTime.now`. Then you can pass the offset into `DateTime.civil`:

```
my_offset = DateTime.now.offset                   # => Rational(-5, 24)

DateTime.civil(1999, 12, 31, 23, 21, Rational(51044, 100000), my_offset).to_s
# => "1999-12-31T23:21:00-0500"
```

Oh, and there's the calendar-reform thing, too. Recall that `Time` objects can only represent dates from a limited range (on 32-bit systems, dates from the 20th and 21st centuries). `DateTime` objects can represent any date at all. The price of this greater range is that `DateTime` needs to worry about calendar reform when dealing with historical dates. If you're using old dates, you may run into a gap caused by a switch from the Julian calendar (which made every fourth year a leap year) to the more accurate Gregorian calendar (which occasionally skips leap years).

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

This switch happened at different times in different countries, creating differentlysized gaps as the local calendar absorbed the extra leap days caused by using the Julian reckoning for so many centuries. Dates created within a particular country's gap are invalid for that country.

By default, Ruby assumes that `Date` objects you create are relative to the Italian calendar, which switched to Gregorian reckoning in 1582. For American and Commonwealth users, Ruby has provided a constant `Date::ENGLAND`, which corresponds to the date that England and its colonies adopted the Gregorian calendar. `DateTime's` constructors and factory methods will accept `Date::ENGLAND` or `Date::ITALY` as an extra argument denoting when calendar reform started in that country. The calendar reform argument can also be any old Julian day, letting you handle old dates from any country:

```
#In Italy, 4 Oct 1582 was immediately followed by 15 Oct 1582.
#
Date.new(1582, 10, 4).to_s
# => "1582-10-04"
Date.new(1582, 10, 5).to_s
# ArgumentError: invalid date
Date.new(1582, 10, 4).succ.to_s
# => "1582-10-15"

#In England, 2 Sep 1752 was immediately followed by 14 Sep 1752.
#
Date.new(1752, 9, 2, Date::ENGLAND).to_s
# => "1752-09-02"
Date.new(1752, 9, 3, Date::ENGLAND).to_s
# ArgumentError: invalid date
Date.new(1752, 9, 2, DateTime::ENGLAND).succ.to_s
# => "1752-09-14"
Date.new(1582, 10, 5, Date::ENGLAND).to_s
# => "1582-10-05"
```

You probably won't need to use Ruby's Gregorian conversion features: it's uncommon that computer applications need to deal with old dates that are both known with precision and associated with a particular locale.

See Also

- A list of the dates of Gregorian conversion for various countries (http://www.polysyllabic.com/GregConv.html)
- Recipe 3.7, "Converting Between Time Zones
- Recipe 3.8, "Checking Whether Daylight Saving Time Is in Effect"

Recipe 3.2. Parsing Dates, Precisely or Fuzzily

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Problem

You want to transform a string describing a date or date/time into a `Date` object. You might not know the format of the string ahead of time.

Solution

The best solution is to pass the date string into `Date.parse` or `DateTime.parse`. These methods use heuristics to guess at the format of the string, and they do a pretty good job:

```
require 'date'

Date.parse('2/9/2007').to_s
# => "2007-02-09"

DateTime.parse('02-09-2007 12:30:44 AM').to_s
# => "2007-09-02T00:30:44Z"

DateTime.parse('02-09-2007 12:30:44 PM EST').to_s
# => "2007-09-02T12:30:44-0500"

Date.parse('Wednesday, January 10, 2001').to_s
# => "2001-01-10"
```

Discussion

The `parse` methods can save you a lot of the drudgework associated with parsing times in other programming languages, but they don't always give you the results you want. Notice in the first example how `Date.parse` assumed that 2/9/2007 was an American (month first) date instead of a European (day first) date. `parse` also tends to misinterpret two-digit years:

```
Date.parse('2/9/07').to_s                  # => "0007-02-09"
```

Let's say that `Date.parse` doesn't work for you, but you know that all the dates you're processing will be formatted a certain way. You can create a format string using the standard `strftime` directives, and pass it along with a date string into `DateTime.strptime` or `Date.strptime`. If the date string matches up with the format string, you'll get a `Date` or `DateTime` object back. You may already be familiar with this technique, since this many languages, as well as the Unix `date` command, do date formatting this way.

Some common date and time formats include:

```
american_date = '%m/%d/%y'
Date.strptime('2/9/07', american_date).to_s           # => "2007-02-09"
DateTime.strptime('2/9/05', american_date).to_s       # => "2005-02-09T00:00:00Z"
Date.strptime('2/9/68', american_date).to_s           # => "2068-02-09"
Date.strptime('2/9/69', american_date).to_s           # => "1969-02-09"

european_date = '%d/%m/%y'
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
Date.strptime('2/9/07', european_date).to_s            # => "2007-09-02"
Date.strptime('02/09/68', european_date).to_s          # => "2068-09-02"
Date.strptime('2/9/69', european_date).to_s            # => "1969-09-02"

four_digit_year_date = '%m/%d/%Y'
Date.strptime('2/9/2007', four_digit_year_date).to_s   # => "2007-02-09"
Date.strptime('02/09/1968', four_digit_year_date).to_s # => "1968-02-09"
Date.strptime('2/9/69', four_digit_year_date).to_s     # => "0069-02-09"

date_and_time = '%m-%d-%Y %H:%M:%S %Z'
DateTime.strptime('02-09-2007 12:30:44 EST', date_and_time).to_s
# => "2007-02-09T12:30:44-0500"
DateTime.strptime('02-09-2007 12:30:44 PST', date_and_time).to_s
# => "2007-02-09T12:30:44-0800"
DateTime.strptime('02-09-2007 12:30:44 GMT', date_and_time).to_s
# => "2007-02-09T12:30:44Z"

twelve_hour_clock_time = '%m-%d-%Y %I:%M:%S %p'
DateTime.strptime('02-09-2007 12:30:44 AM', twelve_hour_clock_time).to_s
# => "2007-02-09T00:30:44Z"
DateTime.strptime('02-09-2007 12:30:44 PM', twelve_hour_clock_time).to_s
# => "2007-02-09T12:30:44Z"

word_date = '%A, %B %d, %Y'
Date.strptime('Wednesday, January 10, 2001', word_date).to_s
# => "2001-01-10"
```

If your date strings might be in one of a limited number of formats, try iterating over a list of format strings and attempting to parse the date string with each one in turn. This gives you some of the flexibility of `Date.parse` while letting you override the assumptions it makes. `Date.parse` is still faster, so if it'll work, use that.

```
Date.parse('1/10/07').to_s                             # => "0007-01-10"
Date.parse('2007 1 10').to_s
# ArgumentError: 3 elements of civil date are necessary

TRY_FORMATS = ['%d/%m/%y', '%Y %m %d']
def try_to_parse(s)
  parsed = nil
  TRY_FORMATS.each do |format|
    begin
      parsed = Date.strptime(s, format)
      break
    rescue ArgumentError
    end
  end
  return parsed
end

try_to_parse('1/10/07').to_s                           # => "2007-10-01"
try_to_parse('2007 1 10').to_s                         # => "2007-01-10"
```

Several common date formats cannot be reliably represented by `strptime` format strings. Ruby defines class methods of `Time` for parsing these date strings, so you don't have to write the code yourself. Each of the following methods returns a `Time` object.

`Time.rfc822` parses a date string in the format of RFC822/RFC2822, the Internet email standard. In an RFC2822 date, the month and the day of the week are always in English (for instance, "Tue" and "Jul"), even if the locale is some other language.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
require 'time'
mail_received = 'Tue, 1 Jul 2003 10:52:37 +0200'
Time.rfc822(mail_received)
# => Tue Jul 01 04:52:37 EDT 2003
```

To parse a date in the format of RFC2616, the HTTP standard, use `Time.httpdate`. An RFC2616 date is the kind of date you see in HTTP headers like Last-Modified. As with RFC2822, the month and day abbreviations are always in English:

```
last_modified = 'Tue, 05 Sep 2006 16:05:51 GMT'
Time.httpdate(last_modified)
# => Tue Sep 05 12:05:51 EDT 2006
```

To parse a date in the format of ISO 8601 or XML Schema, use `Time.iso8601` or `Time.xmlschema`:

```
timestamp = '2001-04-17T19:23:17.201Z'
t = Time.iso8601(timestamp)     # => Tue Apr 17 19:23:17 UTC 2001
t.sec                           # => 17
t.tv_usec                       # => 201000
```

Don't confuse these class methods of `Time` with the instance methods of the same names. The class methods create `Time` objects from strings. The instance methods go the other way, formatting an existing `Time` object as a string:

```
t = Time.at(1000000000)         # => Sat Sep 08 21:46:40 EDT 2001
t.rfc822                        # => "Sat, 08 Sep 2001 21:46:40 -0400"
t.httpdate                      # => "Sun, 09 Sep 2001 01:46:40 GMT"
t.iso8601                       # => "2001-09-08T21:46:40-04:00"
```

See Also

- The RDoc for the `Time#strftime` method lists most of the supported `strftime` directives (`ri Time#strftime`); for a more detailed and complete list, see the table in Recipe 3.3, "Printing a Date"

Recipe 3.3. Printing a Date

Problem

You want to print a date object as a string.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Solution

If you just want to look at a date, you can call `Time#to_s` or `Date#to_s` and not bother with fancy formatting:

```
require 'date'
Time.now.to_s                           # => "Sat Mar 18 19:05:50 EST 2006"
DateTime.now.to_s                       # => "2006-03-18T19:05:50-0500"
```

If you need the date in a specific format, you'll need to define that format as a string containing time-format directives. Pass the format string into `Time#strftime` or `Date#strftime`. You'll get back a string in which the formatting directives have been replaced by the correpsonding parts of the `Time` or `DateTime` object.

A formatting directive looks like a percent sign and a letter: `%x`. Everything in a format string that's not a formatting directive is treated as a literal:

```
Time.gm(2006).strftime('The year is %Y!')   # => "The year is 2006!"
```

The Discussion lists all the time formatting directives defined by `Time#strftime` and `Date#strftime`. Here are some common time-formatting strings, shown against a sample date of about 1:30 in the afternoon, GMT, on the last day of 2005:

```
time = Time.gm(2005, 12, 31, 13, 22, 33)
american_date = '%D'
time.strftime(american_date)                # => "12/31/05"
european_date = '%d/%m/%y'
time.strftime(european_date)                # => "31/12/05"
four_digit_year_date = '%m/%d/%Y'
time.strftime(four_digit_year_date)         # => "12/31/2005"
date_and_time = '%m-%d-%Y %H:%M:%S %Z'
time.strftime(date_and_time)                # => "12-31-2005 13:22:33 GMT"
twelve_hour_clock_time = '%m-%d-%Y %I:%M:%S %p'
time.strftime(twelve_hour_clock_time)       # => "12-31-2005 01:22:33 PM"
word_date = '%A, %B %d, %Y'
time.strftime(word_date)                    # => "Saturday, December 31, 2005"
```

Discussion

Printed forms, parsers, and people can all be very picky about the formatting of dates. Having a date in a standard format makes dates easier to read and scan for errors. Agreeing on a format also prevents ambiguities (is 4/12 the fourth of December, or the twelfth of April?)

If you `require 'time'`, your `Time` objects will sprout special-purpose formatting methods for common date representation standards: `Time#rfc822`, `Time#httpdate`, and `Time#iso8601`. These make it easy for you to print dates in formats compliant with email, HTTP, and XML standards:

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
require 'time'
time.rfc822                              # => "Sat, 31 Dec 2005 13:22:33 -0000"
time.httpdate                            # => "Sat, 31 Dec 2005 13:22:33 GMT"
time.iso8601                             # => "2005-12-31T13:22:33Z"
```

`DateTime` provides only one of these three formats. ISO8601 is the the default string representation of a `DateTime` object (the one you get by calling `#to_s`). This means you can easily print `DateTime` objects into XML documents without having to convert them into `Time` objects.

For the other two formats, your best strategy is to convert the `DateTime` into a `Time` object (see Recipe 3.9 for details). Even on a system with a 32-bit time counter, your `DateTime` objects will probably fit into the 1901–2037 year range supported by `Time`, since RFC822 and HTTP dates are almost always used with dates in the recent past or near future.

Sometimes you need to define a custom date format. `Time#strftime` and `Date#strftime` define many directives for use in format strings. The big table below says what they do. You can combine these in any combination within a formatting string.

Some of these may be familiar to you from other programming languages; virtually all languages since C have included a `strftime` implementation that uses some of these directives. Some of the directives are unique to Ruby.

Table 3-2.

Formatting directive	What it does	Example for 13:22:33 on December 31, 2005
%A	English day of the week	"Saturday"
%a	Abbreviated English day of the week	"Sat"
%B	English month of the year	"December"
%b	English month of the year	"Dec"
%C	The century part of the year, zero-padded if necessary.	"20"
%c	This prints the date and time in a way that looks like the default string representation of Time, but without the timezone. Equivalent to '%a %b %e %H:%M:%S %Y'	"Sat Dec 31 13:22:33 2005"
%D	American-style short date format with two-digit year. Equivalent to "%m/%d/%y"	"12/31/05"
%d	Day of the month, zero-padded	"31"
%e	Day of the month, not zero-padded	"31"
%F	Short date format with 4-digit year.; equivalent to "%Y-%m-%d"	"2005-12-31"
%G	Commercial year with century, zero-padded to a minimum of four digits and with a minus sign prepended for dates BCE (see Recipe 3.11. For the calendar year, use %Y)	"2005"
%g	Year without century, zero-padded to two digits	"05"
%H	Hour of the day, 24-hour clock, zero-padded to two digits	"13"
%h	Abbreviated month of the year; the same as "%b"	"Dec"
%I	Hour of the day, 12-hour clock, zero-padded to two digits	"01"

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Formatting directive	What it does	Example for 13:22:33 on December 31, 2005
%j	Julian day of the year, padded to three digits (from 001 to 366)	"365"
%k	Hour of the day, 24-hour clock, not zero-padded; like %H but with no padding	"13"
%l	Hour of the day, 12-hour clock, not zero-padded; like %I but with no padding	"1"
%M	Minute of the hour, padded to two digits	"22"
%m	Month of the year, padded to two digits	"12"
%n	A newline; don't use this; just put a newline in the formatting string	"\n"
%P	Lowercase meridian indicator ("am" or "pm")	"pm"
%p	Upper meridian indicator. Like %P, except gives "AM" or "PM"; yes, the uppercase P gives the lowercase meridian, and vice versa	"PM"
%R	Short 24-hour time format; equivalent to "%H:%M"	"13:22"
%r	Long 12-hour time format; equivalent to "%I:%M:%S %p"	"01:22:33 PM"
%S	Second of the minute, zero-padded to two digits	"33"
%s	Seconds since the Unix epoch	"1136053353"
%T	Long 24-hour time format; equivalent to "%H:%M:%S"	"13:22:33"
%t	A tab; don't use this; just put a tab in the formatting string	"\t"
%U	Calendar week number of the year: assumes that the first week of the year starts on the first Sunday; if a date comes before the first Sunday of the year, it's counted as part of "week zero" and "00" is returned	"52"
%u	Commercial weekday of the year, from 1 to 7, with Monday being day 1	"6"
%V	Commercial week number of the year (see Recipe 3.11)	"52"
%W	The same as %V, but if a date is before the first Monday of the year, it's counted as part of "week zero" and "00" is returned	"52"
%w	Calendar day of the week, from 0 to 6, with Sunday being day 0	"6"
%X	Preferred representation for the time; equivalent to "%H:%M:%S"	"13:22:33"
%x	Preferred representation for the date; equivalent to "%m/%d/%y"	"12/31/05"
%Y	Year with century, zero-padded to four digits and with a minus sign prepended for dates `BCE`	"2005"
%y	Year without century, zero-padded to two digits	"05"
%Z	The timezone abbreviation (`Time`) or GMT offset (`Date`). `Date` will use "Z" instead of "+0000" if a time is in GMT	"GMT" for Time, "Z" for Date
%z	The timezone as a GMT offset	"+0000"
%%	A literal percent sign	"%"
%v	European-style date format with month abbreviation; equivalent to "%e-%b-%Y"	31-Dec-2005
%+	Prints a Dateobject as though it were a Timeobject converted to a string; like %c, but includes the timezone information; equivalent to "%a %b %e %H:%M:%S %Z %Y"	Sat Dec 31 13:22:33 Z 2005

`Date` defines two formatting directives that won't work at all in `Time#strftime`. Both are shortcuts for formatting strings that you could create manually.

If you need a date format for which there's no formatting directive, you should be able to compensate by writing Ruby code. For instance, suppose you want to format our example

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

date as "The 31st of December". There's no special formatting directive tol print the day as an ordinal number, but you can use Ruby code to build a formatting string that gives the right answer.

```
class Time
  def day_ordinal_suffix
    if day == 11 or day == 12
      return "th"

    else
      case day % 10
      when 1 then return "st"
      when 2 then return "nd"
      when 3 then return "rd"
      else return "th"
      end
    end
  end
end

time.strftime("The %e#{time.day_ordinal_suffix} of %B") # => "The 31st of December"
```

The actual formatting string differs depending on the date. In this case, it ends up "The %est of %B", but for other dates it will be "The %end of %B", "The %erd of %B", or "The %eth of %B".

See Also

- `Time` objects can parse common date formats as well as print them out; see Recipe 3.2, "Parsing Dates, Precisely or Fuzzily," to see how to parse the output of `strftime`, `rfc822`, `httpdate`, and `iso8661`
- Recipe 3.11, "Handling Commercial Dates"

Recipe 3.4. Iterating Over Dates

Problem

Given a point in time, you want to get somewhere else.

Solution

All of Ruby's time objects can be used in ranges as though they were numbers. `Date` and `DateTime` objects iterate in increments of one day, and `Time` objects iterate in increments of one second:

```
require 'date'
(Date.new(1776, 7, 2)..Date.new(1776, 7, 4)).each { |x| puts x }
# 1776-07-02
# 1776-07-03
# 1776-07-04
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
span = DateTime.new(1776, 7, 2, 1, 30, 15)..DateTime.new(1776, 7, 4, 7, 0, 0)
span.each { |x| puts x }
# 1776-07-02T01:30:15Z
# 1776-07-03T01:30:15Z
# 1776-07-04T01:30:15Z

(Time.at(100)..Time.at(102)).each { |x| puts x }
# Wed Dec 31 19:01:40 EST 1969
# Wed Dec 31 19:01:41 EST 1969
# Wed Dec 31 19:01:42 EST 1969
```

Ruby's `Date` class defines `step` and `upto`, the same convenient iterator methods used by numbers:

```
the_first = Date.new(2004, 1, 1)
the_fifth = Date.new(2004, 1, 5)

the_first.upto(the_fifth) { |x| puts x }
# 2004-01-01
# 2004-01-02
# 2004-01-03
# 2004-01-04
# 2004-01-05
```

Discussion

Ruby date objects are stored internally as numbers, and a range of those objects is treated like a range of numbers. For `Date` and `DateTime` objects, the internal representation is the Julian day: iterating over a range of those objects adds one day at a time. For `Time` objects, the internal representation is the number of seconds since the Unix epoch: iterating over a range of `Time` objects adds one second at a time.

`Time` doesn't define the `step` and `upto` method, but it's simple to add them:

```
class Time
  def step(other_time, increment)
   raise ArgumentError, "step can't be 0" if increment == 0
    increasing = self < other_time
    if (increasing && increment < 0) || (!increasing && increment > 0)
      yield self
      return
      end
      d = self
      begin
        yield d
        d += increment
      end while (increasing ? d <= other_time : d >= other_time)
    end

    def upto(other_time)
      step(other_time, 1) { |x| yield x }
    end
  end

  the_first = Time.local(2004, 1, 1)
  the_second = Time.local(2004, 1, 2)
  the_first.step(the_second, 60 * 60 * 6) { |x| puts x }
  # Thu Jan 01 00:00:00 EST 2004
  # Thu Jan 01 06:00:00 EST 2004
  # Thu Jan 01 12:00:00 EST 2004
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
# Thu Jan 01 18:00:00 EST 2004
# Fri Jan 02 00:00:00 EST 2004

the_first.upto(the_first) { |x| puts x }
# Thu Jan 01 00:00:00 EST 2004
```

See Also

- Recipe 2.15, "Generating a Sequence of Numbers"

Recipe 3.5. Doing Date Arithmetic

Problem

You want to find how much time has elapsed between two dates, or add a number to a date to get an earlier or later date.

Solution

Adding or subtracting a `Time` object and a number adds or subtracts that number of seconds. Adding or subtracting a `Date` object and a number adds or subtracts that number of days:

```
require 'date'
y2k = Time.gm(2000, 1, 1)                  # => Sat Jan 01 00:00:00 UTC 2000
y2k + 1                                    # => Sat Jan 01 00:00:01 UTC 2000
y2k - 1                                    # => Fri Dec 31 23:59:59 UTC 1999
y2k + (60 * 60 * 24 * 365)                 # => Sun Dec 31 00:00:00 UTC 2000

y2k_dt = DateTime.new(2000, 1, 1)
(y2k_dt + 1).to_s                          # => "2000-01-02T00:00:00Z"
(y2k_dt - 1).to_s                          # => "1999-12-31T00:00:00Z"
(y2k_dt + 0.5).to_s                        # => "2000-01-01T12:00:00Z"
(y2k_dt + 365).to_s                        # => "2000-12-31T00:00:00Z"
```

Subtracting one `Time` from another gives the interval between the dates, in seconds. Subtracting one `Date` from another gives the interval in days:

```
day_one = Time.gm(1999, 12, 31)
day_two = Time.gm(2000, 1, 1)
day_two - day_one                          # => 86400.0
day_one - day_two                          # => -86400.0

day_one = DateTime.new(1999, 12, 31)
day_two = DateTime.new(2000, 1, 1)
day_two - day_one                          # => Rational(1, 1)
day_one - day_two                          # => Rational(-1, 1)

# Compare times from now and 10 seconds in the future.
before_time = Time.now
before_datetime = DateTime.now
sleep(10)
Time.now - before_time                     # => 10.003414
DateTime.now - before_datetime             # => Rational(5001557, 43200000000)
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

The `activesupport` gem, a prerequisite of Ruby on Rails, defines many useful functions on `Numeric` and `Time` for navigating through time:[2]

[2] So does the Facets More library.

```
require 'rubygems'
require 'active_support'

10.days.ago                                 # => Wed Mar 08 19:54:17 EST 2006
1.month.from_now                            # => Mon Apr 17 20:54:17 EDT 2006
2.weeks.since(Time.local(2006, 1, 1))       # => Sun Jan 15 00:00:00 EST 2006

y2k - 1.day                                 # => Fri Dec 31 00:00:00 UTC 1999
y2k + 6.3.years                             # => Thu Apr 20 01:48:00 UTC 2006
6.3.years.since y2k                         # => Thu Apr 20 01:48:00 UTC 2006
```

Discussion

Ruby's date arithmetic takes advantage of the fact that Ruby's time objects are stored internally as numbers. Additions to dates and differences between dates are handled by adding to and subtracting the underlying numbers. This is why adding 1 to a `Time` adds one second and adding 1 to a `DateTime` adds one day: a `Time` is stored as a number of seconds since a time zero, and a `Date` or `DateTime` is stored as a number of days since a (different) time zero.

Not every arithmetic operation makes sense for dates: you could "multiply two dates" by multiplying the underlying numbers, but that would have no meaning in terms of real time, so Ruby doesn't define those operators. Once a number takes on aspects of the real world, there are limitations to what you can legitimately do to that number.

Here's a shortcut for adding or subtracting big chunks of time: using the right-or left-shift operators on a `Date` or `DateTime` object will add or subtract a certain number number of months from the date.

```
(y2k_dt >> 1).to_s                          # => "2000-02-01T00:00:00Z"
(y2k_dt << 1).to_s                          # => "1999-12-01T00:00:00Z"
```

You can get similar behavior with `activesupport's Numeric#month` method, but that method assumes that a "month" is 30 days long, instead of dealing with the lengths of specific months:

```
y2k + 1.month                               # => Mon Jan 31 00:00:00 UTC 2000
y2k - 1.month                               # => Thu Dec 02 00:00:00 UTC 1999
```

By contrast, if you end up in a month that doesn't have enough days (for instance, you start on the 31st and then shift to a month that only has 30 days), the standard library will use the last day of the new month:

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
# Thirty days hath September…
halloween = Date.new(2000, 10, 31)
(halloween << 1).to_s                       # => "2000-09-30"
(halloween >> 1).to_s                       # => "2000-11-30"
(halloween >> 2).to_s                       # => "2000-12-31"

leap_year_day = Date.new(1996, 2, 29)
(leap_year_day << 1).to_s                   # => "1996-01-29"
(leap_year_day >> 1).to_s                   # => "1996-03-29"
(leap_year_day >> 12).to_s                  # => "1997-02-28"
(leap_year_day << 12 * 4).to_s              # => "1992-02-29"
```

See Also

- Recipe 3.4, "Iterating Over Dates"
- Recipe 3.6, "Counting the Days Since an Arbitrary Date"
- The RDoc for Rails' `ActiveSupport::CoreExtensions::Numeric::Time` module (http://api.rubyonrails.com/classes/ActiveSupport/CoreExtensions/Numeric/Time.html)

Recipe 3.6. Counting the Days Since an Arbitrary Date

Problem

You want to see how many days have elapsed since a particular date, or how many remain until a date in the future.

Solution

Subtract the earlier date from the later one. If you're using `Time` objects, the result will be a floating-point number of seconds, so divide by the number of seconds in a day:

```
def last_modified(file)
  t1 = File.stat(file).ctime
  t2 = Time.now
  elapsed = (t2-t1)/(60*60*24)
  puts "#{file} was last modified #{elapsed} days ago."
end

last_modified("/etc/passwd")
# /etc/passwd was last modified 125.873605469919 days ago.
last_modified("/home/leonardr/")
# /home/leonardr/ was last modified 0.113293513796296 days ago.
```

If you're using `DateTime` objects, the result will be a rational number. You'll probably want to convert it to an integer or floating-point number for display:

```
require 'date'
def advent_calendar(date=DateTime.now)
  christmas = DateTime.new(date.year, 12, 25)
  christmas = DateTime.new(date.year+1, 12, 25) if date > christmas
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
  difference = (christmas-date).to_i
  if difference == 0
    puts "Today is Christmas."
  else
    puts "Only #{difference} day#{"s" unless difference==1} until Christmas."
  end
end

advent_calendar(DateTime.new(2006, 12, 24))
# Only 1 day until Christmas.
advent_calendar(DateTime.new(2006, 12, 25))
# Today is Christmas.
advent_calendar(DateTime.new(2006, 12, 26))
# Only 364 days until Christmas.
```

Discussion

Since times are stored internally as numbers, subtracting one from another will give you a number. Since both numbers measure the same thing (time elapsed since some "time zero"), that number will actually mean something: it'll be the number of seconds or days that separate the two times on the timeline.

Of course, this works with other time intervals as well. To display a difference in hours, for `Time` objects divide the difference by the number of seconds in an hour (3,600, or `1.hour` if you're using Rails). For `DateTime` objects, divide by the number of days in an hour (that is, multiply the difference by 24):

```
sent = DateTime.new(2006, 10, 4, 3, 15)
received = DateTime.new(2006, 10, 5, 16, 33)
elapsed = (received-sent) * 24
puts "You responded to my email #{elapsed.to_f} hours after I sent it."
# You responded to my email 37.3 hours after I sent it.
```

You can even use `divmod` on a time interval to hack it down into smaller and smaller pieces. Once when I was in college, I wrote a script that displayed how much time remained until the finals I should have been studying for. This method gives you a countdown of the days, hours, minutes, and seconds until some scheduled event:

```
require 'date'
def remaining(date, event)
  intervals = [["day", 1], ["hour", 24], ["minute", 60], ["second", 60]]
  elapsed = DateTime.now - date
  tense = elapsed > 0 ? "since" : "until"
  interval = 1.0
  parts = intervals.collect do |name, new_interval|
    interval /= new_interval
    number, elapsed = elapsed.abs.divmod(interval)
  "#{number.to_i} #{name}#{"s" unless number == 1}"
  end
  puts "#{parts.join(", ")} #{tense} #{event}."
end

remaining(DateTime.new(2006, 4, 15, 0, 0, 0, DateTime.now.offset),
          "the book deadline")
# 27 days, 4 hours, 16 minutes, 9 seconds until the book deadline.
remaining(DateTime.new(1999, 4, 23, 8, 0, 0, DateTime.now.offset),
          "the Math 114A final")
# 2521 days, 11 hours, 43 minutes, 50 seconds since the Math 114A final.
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

See Also

- Recipe 3.5, "Doing Date Arithmetic"

Recipe 3.7. Converting Between Time Zones

Problem

You want to change a time object so that it represents the same moment of time in some other time zone.

Solution

The most common time zone conversions are the conversion of system local time to UTC, and the conversion of UTC to local time. These conversions are easy for both `Time` and `DateTime` objects.

The `Time#gmtime` method modifies a `Time` object in place, converting it to UTC. The `Time#localtime` method converts in the opposite direction:

```
now = Time.now       # => Sat Mar 18 20:15:58 EST 2006
now = now.gmtime              # => Sun Mar 19 01:15:58 UTC 2006
now = now.localtime           # => Sat Mar 18 20:15:58 EST 2006
```

The `DateTime.new_offset` method converts a `DateTime` object from one time zone to another. You must pass in the dstination time zone's offset from UTC; to convert local time to UTC, pass in zero. Since `DateTime` objects are immutable, this method creates a new object identical to the old `DateTime` object, except for the time zone offset:

```
require 'date'
local = DateTime.now
local.to_s                    # => "2006-03-18T20:15:58-0500"
utc = local.new_offset(0)
utc.to_s                      # => "2006-03-19T01:15:58Z"
```

To convert a UTC `DateTime` object to local time, you'll need to call `DateTime#new_offset` and pass in the numeric offset for your local time zone. The easiest way to get this offset is to call `offset` on a `DateTime` object known to be in local time. The offset will usually be a rational number with a denominator of 24:

```
local = DateTime.now
utc = local.new_offset

local.offset                 # => Rational(-5, 24)
local_from_utc = utc.new_offset(local.offset)
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
local_from_utc.to_s           # => "2006-03-18T20:15:58-0500"
local == local_from_utc       # => true
```

Discussion

`Time` objects created with `Time.at`, `Time.local`, `Time.mktime`, `Time.new`, and `Time.now` are created using the current system time zone. `Time` objects created with `Time.gm` and `Time.utc` are created using the UTC time zone. `Time` objects can represent any time zone, but it's difficult to use a time zone with `Time` other than local time or UTC.

Suppose you need to convert local time to some time zone other than UTC. If you know the UTC offset for the destination time zone, you can represent it as a fraction of a day and pass it into `DateTime#new_offset`:

```
#Convert local (Eastern) time to Pacific time
eastern = DateTime.now
eastern.to_s                  # => "2006-03-18T20:15:58-0500"

pacific_offset = Rational(-7, 24)
pacific = eastern.new_offset(pacific_offset)
pacific.to_s                  # => "2006-03-18T18:15:58-0700"
```

`DateTime#new_offset` can convert between arbitrary time zone offsets, so for time zone conversions, it's easiest to use `DateTime` objects and convert back to `Time` objects if necessary. But `DateTime` objects only understand time zones in terms of numeric UTC offsets. How can you convert a date and time to UTC when all you know is that the time zone is called "WET", "Zulu", or "Asia/Taskent"?

On Unix systems, you can temporarily change the "system" time zone for the current process. The C library underlying the `Time` class knows about an enormous number of time zones (this "zoneinfo" database is usually located in `/usr/share/zoneinfo/`, if you want to look at the available time zones). You can tap this knowledge by setting the environment variable `TZ` to an appropriate value, forcing the `Time` class to act as though your computer were in some other time zone. Here's a method that uses this trick to convert a `Time` object to any time zone supported by the underlying C library:

```
class Time
  def convert_zone(to_zone)
    original_zone = ENV["TZ"]
    utc_time = dup.gmtime
    ENV["TZ"] = to_zone
    to_zone_time = utc_time.localtime
    ENV["TZ"] = original_zone
    return to_zone_time
  end
end
```

Let's do a number of conversions of a local (Eastern) time to other time zones across the world:

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
t = Time.at(1000000000)                 # => Sat Sep 08 21:46:40 EDT 2001

t.convert_zone("US/Pacific")            # => Sat Sep 08 18:46:40 PDT 2001
t.convert_zone("US/Alaska")             # => Sat Sep 08 17:46:40 AKDT 2001

t.convert_zone("UTC")                   # => Sun Sep 09 01:46:40 UTC 2001
t.convert_zone("Turkey")                # => Sun Sep 09 04:46:40 EEST 2001
```

Note that some time zones, like India's, are half an hour offset from most others:

```
t.convert_zone("Asia/Calcutta")         # => Sun Sep 09 07:16:40 IST 2001
```

By setting the `TZ` environment variable before creating a `Time` object, you can represent the time in any time zone. The following code converts Lagos time to Singapore time, regardless of the "real" underlying time zone.

```
ENV["TZ"] = "Africa/Lagos"
t = Time.at(1000000000)                 # => Sun Sep 09 02:46:40 WAT 2001
ENV["TZ"] = nil

t.convert_zone("Singapore")             # => Sun Sep 09 09:46:40 SGT 2001

# Just to prove it's the same time as before:
t.convert_zone("US/Eastern")            # => Sat Sep 08 21:46:40 EDT 2001
```

Since the `TZ` environment variable is global to a process, you'll run into problems if you have multiple threads trying to convert time zones at once.

See Also

- Recipe 3.9, "Converting Between Time and DateTime Objects"
- Recipe 3.8, "Checking Whether Daylight Saving Time Is in Effect"
- Information on the "zoneinfo" database (http://www.twinsun.com/tz/tz-link.htm)

Recipe 3.8. Checking Whether Daylight Saving Time Is in Effect

Problem

You want to see whether the current time in your locale is normal time or Daylight Saving/Summer Time.

Solution

Create a `Time` object and check its `isdst` method:

```
Time.local(2006, 1, 1)                  # => Sun Jan 01 00:00:00 EST 2006
Time.local(2006, 1, 1).isdst            # => false
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
Time.local(2006, 10, 1)               # => Sun Oct 01 00:00:00 EDT 2006
Time.local(2006, 10, 1).isdst         # => true
```

Discussion

`Time` objects representing UTC times will always return false when `isdst` is called, because UTC is the same year-round. Other `Time` objects will consult the daylight saving time rules for the time locale used to create the `Time` object. This is usually the sysem locale on the computer you used to create it: see Recipe 3.7 for information on changing it. The following code demonstrates some of the rules pertaining to Daylight Saving Time across the United States:

```
eastern = Time.local(2006, 10, 1)      # => Sun Oct 01 00:00:00 EDT 2006
eastern.isdst                          # => true

ENV['TZ'] = 'US/Pacific'
pacific = Time.local(2006, 10, 1)      # => Sun Oct 01 00:00:00 PDT 2006
pacific.isdst                          # => true

# Except for the Navajo Nation, Arizona doesn't use Daylight Saving Time.
ENV['TZ'] = 'America/Phoenix'
arizona = Time.local(2006, 10, 1)      # => Sun Oct 01 00:00:00 MST 2006
arizona.isdst                          # => false

# Finally, restore the original time zone.
ENV['TZ'] = nil
```

The C library on which Ruby's `Time` class is based handles the complex rules for Daylight Saving Time across the history of a particular time zone or locale. For instance,

Daylight Saving Time was mandated across the U.S. in 1918, but abandoned in most locales shortly afterwards. The "zoneinfo" file used by the C library contains this information, along with many other rules:

```
# Daylight saving first took effect on March 31, 1918.
Time.local(1918, 3, 31).isdst         # => false
Time.local(1918, 4, 1).isdst          # => true
Time.local(1919, 4, 1).isdst          # => true

# The federal law was repealed later in 1919, but some places
# continued to use Daylight Saving Time.
ENV['TZ'] = 'US/Pacific'
Time.local(1920, 4, 1)                # => Thu Apr 01 00:00:00 PST 1920

ENV['TZ'] = nil
Time.local(1920, 4, 1)                # => Thu Apr 01 00:00:00 EDT 1920

# Daylight Saving Time was reintroduced during the Second World War.
Time.local(1942,2,9)                  # => Mon Feb 09 00:00:00 EST 1942
Time.local(1942,2,10)                 # => Tue Feb 10 00:00:00 EWT 1942
# EWT stands for "Eastern War Time"
```

A U.S. law passed in 2005 expands Daylight Saving Time into March and November, beginning in 2007. Depending on how old your zoneinfo file is, `Time` objects you create for dates in 2007 and beyond might or might not reflect the new law.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
Time.local(2007, 3, 13)              # => Tue Mar 13 00:00:00 EDT 2007
# Your computer may incorrectly claim this time is EST.
```

This illustrates a general point. There's nothing your elected officials love more than passing laws, so you shouldn't rely on `isdst` to be accurate for any `Time` objects that represent times a year or more into the future. When that time actually comes around, Daylight Saving Time might obey different rules in your locale.

The `Date` class isn't based on the C library, and knows nothing about time zones or locales, so it also knows nothing about Daylight Saving Time.

See Also

- Recipe 3.7, "Converting Between Time Zones"
- Information on the "zoneinfo" database (http://www.twinsun.com/tz/tz-link.htm)

Recipe 3.9. Converting Between Time and DateTime Objects

Problem

You're working with both `DateTime` and `Time` objects, created from Ruby's two standard date/time libraries. You can't mix these objects in comparisons, iterations, or date arithmetic because they're incompatible. You want to convert all the objects into one form or another so that you can treat them all the same way.

Solution

To convert a `Time` object to a `DateTime`, you'll need some code like this:

```
require 'date'
class Time
  def to_datetime
    # Convert seconds + microseconds into a fractional number of seconds
    seconds = sec + Rational(usec, 10**6)

    # Convert a UTC offset measured in minutes to one measured in a
    # fraction of a day.
    offset = Rational(utc_offset, 60 * 60 * 24)
    DateTime.new(year, month, day, hour, min, seconds, offset)
  end
end

time = Time.gm(2000, 6, 4, 10, 30, 22, 4010)
# => Sun Jun 04 10:30:22 UTC 2000
time.to_datetime.to_s
# => "2000-06-04T10:30:22Z"
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Converting a `DateTime` to a `Time` is similar; you just need to decide whether you want the `Time` object to use local time or GMT. This code adds the conversion method to `Date`, the superclass of `DateTime`, so it will work on both `Date` and `DateTime` objects.

```
class Date
  def to_gm_time
    to_time(new_offset, :gm)
  end

  def to_local_time
    to_time(new_offset(DateTime.now.offset-offset), :local)
  end

  private
  def to_time(dest, method)
    #Convert a fraction of a day to a number of microseconds
    usec = (dest.sec_fraction * 60 * 60 * 24 * (10**6)).to_i
    Time.send(method, dest.year, dest.month, dest.day, dest.hour, dest.min,
              dest.sec, usec)
  end
end

(datetime = DateTime.new(1990, 10, 1, 22, 16, Rational(41,2))).to_s
# => "1990-10-01T22:16:20Z"
datetime.to_gm_time
# => Mon Oct 01 22:16:20 UTC 1990
datetime.to_local_time
# => Mon Oct 01 17:16:20 EDT 1990
```

Discussion

Ruby's two ways of representing dates and times don't coexist very well. But since neither can be a total substitute for the other, you'll probably use them both during your Ruby career. The conversion methods let you get around incompatibilities by simply converting one type to the other:

```
time < datetime
# ArgumentError: comparison of Time with DateTime failed
time.to_datetime < datetime
# => false
time < datetime.to_gm_time
# => false

time - datetime
# TypeError: can't convert DateTime into Float
(time.to_datetime - datetime).to_f
# => 3533.50973962975                    # Measured in days
time - datetime.to_gm_time
# => 305295241.50401                     # Measured in seconds
```

The methods defined above are reversible: you can convert back and forth between `Date` and `DateTime` objects without losing accuracy.

```
time                                     # => Sun Jun 04 10:30:22 UTC 2000
time.usec                                # => 4010'

time.to_datetime.to_gm_time              # => Sun Jun 04 10:30:22 UTC 2000
time.to_datetime.to_gm_time.usec         # => 4010
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
datetime.to_s                              # => "1990-10-01T22:16:20Z"
datetime.to_gm_time.to_datetime.to_s       # => "1990-10-01T22:16:20Z"
```

Once you can convert between `Time` and `DateTime` objects, it's simple to write code that normalizes a mixed array, so that all its elements end up being of the same type. This method tries to turn a mixed array into an array containing only `Time` objects. If it encounters a date that won't fit within the constraints of the `Time` class, it starts over and converts the array into an array of `DateTime` objects instead (thus losing anyinformation about Daylight Saving Time):

```
def normalize_time_types(array)
  # Don't do anything if all the objects are already of the same type.
  first_class = array[0].class
  first_class = first_class.super if first_class == DateTime
  return unless array.detect { |x| !x.is_a?(first_class) }

  normalized = array.collect do |t|
    if t.is_a?(Date)
      begin
        t.to_local_time
      rescue ArgumentError # Time out of range; convert to DateTimes instead.
        convert_to = DateTime
        break
      end
    else
      t
    end
  end

  unless normalized
    normalized = array.collect { |t| t.is_a?(Time) ? t.to_datetime : t }
  end
  return normalized
end
```

When all objects in a mixed array can be represented as either `Time` or `DateTime` objects, this method makes them all `Time` objects:

```
mixed_array = [Time.now, DateTime.now]
# => [Sat Mar 18 22:17:10 EST 2006,
#        #<DateTime: 2355661091453457l/9600000000,-5/24,2299161>]
normalize_time_types(mixed_array)
# => [Sat Mar 18 22:17:10 EST 2006, Sun Mar 19 03:17:10 EST 2006]
```

If one of the `DateTime` objects can't be represented as a `Time`, `normalize_time_types` turns all the objects into `DateTime` instances. This code is run on a system with a 32-bit time counter:

```
mixed_array << DateTime.civil(1776, 7, 4)
normalize_time_types(mixed_array).collect { |x| x.to_s }
# => ["2006-03-18T22:17:10-0500", "2006-03-18T22:17:10-0500",
#     "1776-07-04T00:00:00Z"]
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

See Also

- Recipe 3.1, "Finding Today's Date"

Recipe 3.10. Finding the Day of the Week

Problem

You want to find the day of the week for a certain date.

Solution

Use the `wday` method (supported by both `Time` and `DateTime`) to find the day of the week as a number between 0 and 6. Sunday is day zero.

The following code yields to a code block the date of every Sunday between two dates. It uses `wday` to find the first Sunday following the start date (keeping in mind that the first date may itself be a Sunday). Then it adds seven days at a time to get subsequent Sundays:

```
def every_sunday(d1, d2)
  # You can use 1.day instead of 60*60*24 if you're using Rails.
  one_day = d1.is_a?(Time) ? 60*60*24 : 1
  sunday = d1 + ((7-d1.wday) % 7) * one_day
  while sunday < d2
    yield sunday
    sunday += one_day * 7
  end
end

def print_every_sunday(d1, d2)
  every_sunday(d1, d2) { |sunday| puts sunday.strftime("%x")}
end

print_every_sunday(Time.local(2006, 1, 1), Time.local(2006, 2, 4))
# 01/01/06
# 01/08/06
# 01/15/06
# 01/22/06
# 01/29/06
```

Discussion

The most commonly used parts of a time are its calendar and clock readings: year, day, hour, and so on. `Time` and `DateTime` let you access these, but they also give you access to a few other aspects of a time: the Julian day of the year (`yday`), and, more usefully, the day of the week (`wday`).

The `every_sunday` method will accept either two `Time` objects or two `DateTime` objects. The only difference is the number you need to add to an object to increment it by one day. If you're only going to be using one kind of object, you can simplify the code a little.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

To get the day of the week as an English string, use the `strftime` directives `%A` and `%a`:

```
t = Time.local(2006, 1, 1)
t.strftime("%A %A %A!")                  # => "Sunday Sunday Sunday!"
t.strftime("%a %a %a!")                  # => "Sun Sun Sun!"
```

You can find the day of the week and the day of the year, but Ruby has no built-in method for finding the week of the year (there is a method to find the *commercial* week of the year; see Recipe 3.11). If you need such a method, it's not hard to create one using the day of the year and the day of the week. This code defines a `week` method in a module, which it mixes in to both `Date` and `Time`:

```
require 'date'
module Week
 def week
    (yday + 7 - wday) / 7
  end
end

class Date
  include Week
end

class Time
  include Week
end

saturday = DateTime.new(2005, 1, 1)
saturday.week                            # => 0
(saturday+1).week                        # => 1 #Sunday, January 2
(saturday-1).week                        # => 52 #Friday, December 31
```

See Also

- Recipe 3.3, "Printing a Date"
- Recipe 3.5, "Doing Date Arithmetic"
- Recipe 3.11, "Handling Commercial Dates"

Recipe 3.11. Handling Commercial Dates

Problem

When writing a business or financial application, you need to deal with commercial dates instead of civil or calendar dates.

Solution

`DateTime` offers some methods for working with commercial dates. `Date#cwday` gives the commercial day of the week, `Date#cweek` gives the commercial week of the year, and `Date#cwyear` gives the commercial year.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Consider January 1, 2006. This was the first day of calendar 2006, but since it was a Sunday, it was the last day of commercial 2005:

```
require 'date'
sunday = DateTime.new(2006, 1, 1)
sunday.year                             # => 2006
sunday.cwyear                           # => 2005
sunday.cweek                            # => 52
sunday.wday                             # => 0
sunday.cwday                            # => 7
```

Commercial 2006 started on the first *weekday* in 2006:

```
monday = sunday + 1
monday.cwyear                           # => 2006
monday.cweek                            # => 1
```

Discussion

Unless you're writing an application that needs to use commercial dates, you probably don't care about this, but it's kind of interesting (if you think dates are interesting). The commercial week starts on Monday, not Sunday, because Sunday's part of the weekend. `DateTime#cwday` is just like `DateTime#wday`, except it gives Sunday a value of seven instead of zero.

This means that `DateTime#cwday` has a range from one to seven instead of from zero to six:

```
(sunday…sunday+7).each do |d|
  puts "#{d.strftime("%a")} #{d.wday} #{d.cwday}"
end
# Sun 0 7
# Mon 1 1
# Tue 2 2
# Wed 3 3
# Thu 4 4
# Fri 5 5
# Sat 6 6
```

The `cweek` and `cwyear` methods have to do with the commercial year, which starts on the first Monday of a year. Any days before the first Monday are considered part of the previous commercial year. The example given in the Solution demonstrates this: January 1, 2006 was a Sunday, so by the commercial reckoning it was part of the last week of 2005.

See Also

- See Recipe 3.3, "Printing a Date," for the `strftime` directives used to print parts of commercial dates

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Recipe 3.12. Running a Code Block Periodically

Problem

You want to run some Ruby code (such as a call to a shell command) repeatedly at a certain interval.

Solution

Create a method that runs a code block, then sleeps until it's time to run the block again:

```
def every_n_seconds(n)
  loop do
    before = Time.now
    yield
    interval = n-(Time.now-before)
    sleep(interval) if interval > 0
  end
end
every_n_seconds(5) do
 puts "At the beep, the time will be #{Time.now.strftime("%X")}…beep!"
end
# At the beep, the time will be 12:21:28… beep!
# At the beep, the time will be 12:21:33… beep!
# At the beep, the time will be 12:21:38… beep!
# …
```

Discussion

There are two main times when you'd want to run some code periodically. The first is when you actually want something to happen at a particular interval: say you're appending your status to a log file every 10 seconds. The other is when you would prefer for something to happen continuously, but putting it in a tight loop would be bad for system performance. In this case, you compromise by putting some slack time in the loop so that your code isn't *always* running.

The implementation of `every_n_seconds` deducts from the sleep time the time spent running the code block. This ensures that calls to the code block are spaced evenly apart, as close to the desired interval as possible. If you tell `every_n_seconds` to call a code block every five seconds, but the code block takes four seconds to run, `every_n_seconds` only sleeps for one second. If the code block takes six seconds to run, `every_n_seconds` won't sleep at all: it'll come back from a call to the code block, and immediately `yield` to the block again.

If you always want to sleep for a certain interval, no matter how long the code block takes to run, you can simplify the code:

```
def every_n_seconds(n)
  loop do
    yield
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
    sleep(n)
  end
end
```

In most cases, you don't want `every_n_seconds` to take over the main loop of your program. Here's a version of `every_n_seconds` that spawns a separate thread to run your task. If your code block stops the loop by with the `break` keyword, the thread stops running:

```
def every_n_seconds(n)
  thread = Thread.new do
    while true
      before = Time.now
      yield
      interval = n-(Time.now-before)
      sleep(interval) if interval > 0
    end
  end
  return thread
end
```

In this snippet, I use `every_n_seconds` to spy on a file, waiting for people to modify it:

```
def monitor_changes(file, resolution=1)
  last_change = Time.now
  every_n_seconds(resolution) do
    check = File.stat(file).ctime
    if check > last_change
      yield file
      last_change = check
    elsif Time.now - last_change > 60
      puts "Nothing's happened for a minute, I'm bored."
      break
    end
  end
end
```

That example might give output like this, if someone on the system is working on the file `/tmp/foo`:

```
thread = monitor_changes("/tmp/foo") { |file| puts "Someone changed #{file}!" }
# "Someone changed /tmp/foo!"
# "Someone changed /tmp/foo!"
# "Nothing's happened for a minute; I'm bored."
thread.status                # => false
```

See Also

- Recipe 3.13, "Waiting a Certain Amount of Time"
- Recipe 23.4, "Running Periodic Tasks Without cron or at"

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Recipe 3.13. Waiting a Certain Amount of Time

Problem

You want to pause your program, or a single thread of it, for a specific amount of time.

Solution

The `Kernel#sleep` method takes a floating-point number and puts the current thread to sleep for some (possibly fractional) number of seconds:

```
3.downto(1) { |i| puts "#{i}…"; sleep(1) }; puts "Go!"
# 3…
# 2…
# 1…
# Go!

Time.new                    # => Sat Mar 18 21:17:58 EST 2006
sleep(10)
Time.new                    # => Sat Mar 18 21:18:08 EST 2006
sleep(1)
Time.new                    # => Sat Mar 18 21:18:09 EST 2006
# Sleep for less then a second.
Time.new.usec               # => 377185
sleep(0.1)
Time.new.usec               # => 479230
```

Discussion

Timers are often used when a program needs to interact with a source much slower than a computer's CPU: a network pipe, or human eyes and hands. Rather than constantly poll for new data, a Ruby program can sleep for a fraction of a second between each poll, giving other programs on the CPU a chance to run. That's not much time by human standards, but sleeping for a fraction of a second at a time can greatly improve a system's overall performance.

You can pass any floating-point number to `sleep`, but that gives an exaggerated picture of how finely you can control a thread's sleeping time. For instance, you can't sleep for 10^{-50} seconds, because it's physically impossible (that's less than the Planck time). You can't sleep for `Float::EPSILON` seconds, because that's almost certainly less than the resolution of your computer's timer.

You probably can't even reliably `sleep` for a microsecond, even though most modern computer clocks have microsecond precision. By the time your `sleep` command is processed by the Ruby interpreter and the thread actually starts waiting for its timer to go off, some small amount of time has already elapsed. At very small intervals, this time can be greater than the time you asked Ruby to sleep in the first place.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Here's a simple benchmark that shows how long `sleep` on your system will actually make a thread sleep. It starts with a `sleep` interval of one second, which is fairly accurate. It then sleeps for shorter and shorter intervals, with lessening accuracy each time:

```
interval = 1.0
10.times do |x|
  t1 = Time.new
  sleep(interval)
  actual = Time.new - t1

  difference = (actual-interval).abs
  percent_difference = difference / interval * 100
  printf("Expected: %.9f Actual: %.6f Difference: %.6f (%.2f%%)\n",
         interval, actual, difference, percent_difference)

  interval /= 10
end
# Expected: 1.000000000 Actual: 0.999420 Difference: 0.000580 (0.06%)
# Expected: 0.100000000 Actual: 0.099824 Difference: 0.000176 (0.18%)
# Expected: 0.010000000 Actual: 0.009912 Difference: 0.000088 (0.88%)
# Expected: 0.001000000 Actual: 0.001026 Difference: 0.000026 (2.60%)
# Expected: 0.000100000 Actual: 0.000913 Difference: 0.000813 (813.00%)
# Expected: 0.000010000 Actual: 0.000971 Difference: 0.000961 (9610.00%)
# Expected: 0.000001000 Actual: 0.000975 Difference: 0.000974 (97400.00%)
# Expected: 0.000000100 Actual: 0.000015 Difference: 0.000015 (14900.00%)
# Expected: 0.000000010 Actual: 0.000024 Difference: 0.000024 (239900.00%)
# Expected: 0.000000001 Actual: 0.000016 Difference: 0.000016 (1599900.00%)
```

A small amount of the reported time comes from overhead, caused by creating the second `Time` object, but not enough to affect these results. On my system, if I tell Ruby to sleep for a millisecond, the time spent running the `sleep` call greatly exceeds the time I wanted to `sleep` in the first place! According to this benchmark, the shortest length of time for which I can expect `sleep` to accurately sleep is about 1/100 of a second.

You might think to get better sleep resolution by putting the CPU into a tight loop with a certain number of repetitions. Apart from the obvious problems (this hurts system performance, and the same loop will run faster over time since computers are always getting faster), this isn't even reliable.

The operating system doesn't know you're trying to run a timing loop: it just sees you using the CPU, and it can interrupt your loop at any time, for any length of time, to let some other process use the CPU. Unless you're on an embedded operating system where you can control exactly what the CPU does, the only reliable way to wait for a specific period of time is with `sleep`.

Waking up early

The `sleep` method will end early if the thread that calls it has its `run` method called. If you want a thread to sleep until another thread wakes it up, use `Thread.stop`:

```
alarm = Thread.new(self) { sleep(5); Thread.main.wakeup }
puts "Going to sleep for 1000 seconds at #{Time.new}…"
sleep(10000); puts "Woke up at #{Time.new}!"
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
# Going to sleep for 1000 seconds at Thu Oct 27 14:45:14 PDT 2005…
# Woke up at Thu Oct 27 14:45:19 PDT 2005!

alarm = Thread.new(self) { sleep(5); Thread.main.wakeup }
puts "Goodbye, cruel world!";
Thread.stop;
puts "I'm back; how'd that happen?"
# Goodbye, cruel world!
# I'm back; how'd that happen?
```

See Also

- Recipe 3.12, "Running a Code Block Periodically"
- Chapter 20
- The Morse Code example in Recipe 21.11, "Making Your Keyboard Lights Blink," displays an interesting use of `sleep`

Recipe 3.14. Adding a Timeout to a Long-Running Operation

Problem

You're running some code that might take a long time to complete, or might never complete at all. You want to interrupt the code if it takes too long.

Solution

Use the built-in `timeout` library. The `Timeout.timeout` method takes a code block and a deadline (in seconds). If the code block finishes running in time, it returns true. If the deadline passes and the code block is still running, `Timeout.timeout` terminates the code block and raises an exception.

The following code would never finish running were it not for the `timeout` call. But after five seconds, `timeout` raises a `Timeout::Error` and execution halts:

```
# This code will sleep forever… OR WILL IT?
require 'timeout'
before = Time.now
begin
  status = Timeout.timeout(5) { sleep }
rescue Timeout::Error
  puts "I only slept for #{Time.now-before} seconds."
end
# I only slept for 5.035492 seconds.
```

Discussion

Sometimes you must make a network connection or take some other action that might be incredibly slow, or that might never complete at all. With a timeout, you can impose an upper limit on how long that operation can take. If it fails, you can try it again later, or

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

forge ahead without the information you were trying to get. Even when you can't recover, you can report your failure and gracefully exit the program, rather than sitting around forever waiting for the operation to complete.

By default, `Timeout.timeout` raises a `Timeout::Error`. You can pass in a custom exception class as the second argument to `Timeout.timeout`: this saves you from having to rescue the `Timeout:Error` just so you can raise some other error that your application knows how to handle.

If the code block had side effects, they will still be visible after the timeout kills the code block:

```
def count_for_five_seconds
  $counter = 0
  begin
    Timeout::timeout(5) { loop { $counter += 1 } }
  rescue Timeout::Error
    puts "I can count to #{$counter} in 5 seconds."
  end
end

count_for_five_seconds
# I can count to 2532825 in 5 seconds.
$counter                                  # => 2532825
```

This may mean that your dataset is now in an inconsistent state.

See Also

- `ri Timeout`
- Recipe 3.13, "Waiting a Certain Amount of Time"
- Recipe 14.1, "Grabbing the Contents of a Web Page"

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01