Table of Contents

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

2. Numbers

Numbers are as fundamental to computing as breath is to human life. Even programs that have nothing to do with math need to count the items in a data structure, display average running times, or use numbers as a source of randomness. Ruby makes it easy to represent numbers, letting you breathe easy and tackle the harder problems of programming.

An issue that comes up when you're programming with numbers is that there are several different implementations of "number," optimized for different purposes: 32bit integers, floating-point numbers, and so on. Ruby tries to hide these details from you, but it's important to know about them because they often manifest as mysteriously incorrect calculations.[1]

[1] See, for instance, the Discussion section of Recipe 2.11, where it's revealed that `Matrix#inverse` doesn't work correctly on a matrix full of integers. This is because `Matrix#inverse` uses division, and integer division works differently from floating-point division.

The first distinction is between small numbers and large ones. If you've used other programming languages, you probably know that you must use different data types to hold small numbers and large numbers (assuming that the language supports large numbers at all). Ruby has different classes for small numbers (`Fixnum`) and large numbers (`Bignum`), but you don't usually have to worry about the difference. When you type in a number, Ruby sees how big it is and creates an object of the appropriate class.

```
1000.class                      # => Fixnum
10000000000.class               # => Bignum
(2**30 - 1).class               # => Fixnum
(2**30).class                   # => Bignum
```

When you perform arithmetic, Ruby automatically does any needed conversions. You don't have to worry about the difference between small and large numbers:[2]

[2] Python also has this feature.

```
small = 1000
big = small ** 5                # => 1000000000000000
big.class                       # => Bignum
smaller = big / big             # => 1
smaller.class                   # => Fixnum
```

The other major distinction is between whole numbers (integers) and fractional numbers. Like all modern programming languages, Ruby implements the IEEE floating-point standard for representing fractional numbers. If you type a number that includes a decimal point, Ruby creates a `Float` object instead of a `Fixnum` or `Bignum`:

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

```
0.01.class                          # => Float
1.0.class                           # => Float
10000000000.00000000001.class       # => Float
```

But floating-point numbers are imprecise (see Recipe 2.2), and they have their own size limits, so Ruby also provides a class that can represent any number with a finite decimal expansion (Recipe 2.3). There's also a class for numbers like two-thirds, which have an infinite decimal expansion (Recipe 2.4), and a class for complex or "irrational" numbers (Recipe 2.12).

Every kind of number in Ruby has its own class (`Integer`, `Bignum`, `Complex`, and so on), which inherits from the `Numeric` class. All these classes implement the basic arithmetic operations, and in most cases you can mix and match numbers of different types (see Recipe 8.9 for more on how this works). You can reopen these classes to add new capabilities to numbers (see, for instance, Recipe 2.17), but you can't usefully subclass them.

Ruby provides simple ways of generating random numbers (Recipe 2.5) and sequences of numbers (Recipe 2.15). This chapter also covers some simple mathematical algorithms (Recipes 2.7 and 2.11) and statistics (Recipe 2.8).

Recipe 2.1. Parsing a Number from a String

Problem

Given a string that contains some representation of a number, you want to get the corresponding integer or floating-point value.

Solution

Use `String#to_i` to turn a string into an integer. Use `String#to_f` to turn a string into a floating-point number.

```
'400'.to_i                          # => 400
'3.14'.to_f                         # => 3.14
'1.602e-19'.to_f                    # => 1.602e-19
```

Discussion

Unlike Perl and PHP, Ruby does not automatically make a number out of a string that contains a number. You must explicitly call a conversion method that tells Ruby *how* you want the string to be converted.

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

Along with `to_i` and `to_f`, there are other ways to convert strings into numbers. If you have a string that represents a hex or octal string, you can call `String#hex` or `String#oct` to get the decimal equivalent. This is the same as passing the base of the number into `to_i`:

```
'405'.oct                          # => 261
'405'.to_i(8)                      # => 261
'405'.hex                          # => 1029
'405'.to_i(16)                     # => 1029
'fed'.hex                          # => 4077
'fed'.to_i(16)                     # => 4077
```

If `to_i`, `to_f`, `hex`,or `oct` find a character that can't be part of the kind of number they're looking for, they stop processing the string at that character and return the number so far. If the string's first character is unusable, the result is zero.

```
"13: a baker's dozen".to_i                          # => 13
'1001 Nights'.to_i                                  # => 1001
'The 1000 Nights and a Night'.to_i                  # => 0
'60.50 Misc. Agricultural Equipment'.to_f           # => 60.5
'$60.50'.to_f                                       # => 0.0
'Feed the monster!'.hex                             # => 65261
'I fed the monster at Canoga Park Waterslides'.hex # => 0
'0xA2Z'.hex                                         # => 162
'-10'.oct                                           # => -8
'-109'.oct                                          # => -8
'3.14'.to_i                                         # => 3
```

Note especially that last example: the decimal point is just one more character that stops processing of a string representing an integer.

If you want an exception when a string can't be completely parsed as a number, use `Integer( )` or `Float( )`:

```
Integer('1001')                                     # => 1001
Integer('1001 nights')
# ArgumentError: invalid value for Integer: "1001 nights"

Float('99.44')                                      # => 99.44
Float('99.44% pure')
# ArgumentError: invalid value for Float(): "99.44% pure"
```

To extract a number from within a larger string, use a regular expression. The `NumberParser` class below contains regular expressions for extracting floating-point strings, as well as decimal, octal, and hexadecimal numbers. Its `extract_numbers` method uses `String#scan` to find all the numbers of a certain type in a string.

```
class NumberParser
  @@number_regexps = {
    :to_i => /([+-]?[0-9]+)/,
    :to_f => /([+-]?([0-9]*\.)?[0-9]+(e[+-]?[0-9]+)?)/i,
    :oct => /([+-]?[0-7]+)/,
    :hex => /\b([+-]?(0x)?[0-9a-f]+)\b/i
```


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

```
    #The \b characters prevent every letter A-F in a word from being
    #considered a hexadecimal number.
  }

  def NumberParser.re(parsing_method=:to_i)
    re = @@number_regexps[parsing_method]
    raise ArgumentError, "No regexp for #{parsing_method.inspect}!" unless re
    return re
  end

  def extract(s, parsing_method=:to_i)
    numbers = []
    s.scan(NumberParser.re(parsing_method)) do |match|
      numbers << match[0].send(parsing_method)
    end
    numbers
  end
end
```

Here it is in action:

```
p = NumberParser.new

pw = "Today's numbers are 104 and 391."
NumberParser.re(:to_i).match(pw).captures                 # => ["104"]
p.extract(pw, :to_i)                                      # => [104, 391]

p.extract('The 1000 nights and a night')                  # => [1000]
p.extract('$60.50', :to_f)                                # => [60.5]
p.extract('I fed the monster at Canoga Park Waterslides', :hex)
# => [4077]
p.extract('In octal, fifteen is 017.', :oct)              # => [15]

p.extract('From 0 to 10e60 in -2.4 seconds', :to_f)
# => [0.0, 1.0e+61, -2.4]
p.extract('From 0 to 10e60 in -2.4 seconds')
# => [0, 10, 60, -2, 4]
```

If you want to extract more than one kind of number from a string, the most reliable strategy is to stop using regular expressions and start using the `scanf` module, a free third-party module that provides a parser similar to C's `scanf` function.

```
require 'scanf'
s = '0x10 4.44 10'.scanf('%x %f %d')                      # => [16, 4.44, 10]
```

See Also

- Recipe 2.6, "Converting Between Numeric Bases"
- Recipe 8.9, "Converting and Coercing Objects to Different Types"
- The `scanf` module (http://www.rubyhacker.com/code/scanf/)

Recipe 2.2. Comparing Floating-Point Numbers

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

Problem

Floating-point numbers are not suitable for exact comparison. Often, two numbers that should be equal are actually slightly different. The Ruby interpreter can make seemingly nonsensical assertions when floating-point numbers are involved:

```
1.8 + 0.1                       # => 1.9
1.8 + 0.1 == 1.9                # => false
1.8 + 0.1 > 1.9                 # => true
```

You want to do comparison operations approximately, so that floating-point numbers infintesimally close together can be treated equally.

Solution

You can avoid this problem altogether by using `BigDecimal` numbers instead of floats (see Recipe 2.3). `BigDecimal` numbers are completely precise, and work as well as as floats for representing numbers that are relatively small and have few decimal places: everyday numbers like the prices of fruits. But math on `BigDecimal` numbers is much slower than math on floats. Databases have native support for floating-point numbers, but not for `BigDecimals`. And floating-point numbers are simpler to create (simply type `10.2` in an interactive Ruby shell to get a `Float` object). `BigDecimals` can't totally replace floats, and when you use floats it would be nice not to have to worry about tiny differences between numbers when doing comparisons.

But how tiny is "tiny"? How large can the difference be between two numbers before they should stop being considered equal? As numbers get larger, so does the range of floating-point values that can reasonably be expected to model that number. 1.1 is probably not "approximately equal" to 1.2, but $10^{20} + 0.1$ is probably "approximately equal" to $10^{20} + 0.2$.

The best solution is probably to compare the relative magnitudes of large numbers, and the absolute magnitudes of small numbers. The following code accepts both two thresholds: a relative threshold and an absolute threshold. Both default to `Float::EPSILON`, the smallest possible difference between two `Float` objects. Two floats are considered approximately equal if they are within `absolute_epsilon` of each other, or if the difference between them is `relative_epsilon` times the magnitude of the larger one.

```
class Float
  def approx(other, relative_epsilon=Float::EPSILON, epsilon=Float::EPSILON)
    difference = other - self
    return true if difference.abs <= epsilon
    relative_error = (difference / (self > other ? self : other)).abs
    return relative_error <= relative_epsilon
  end
end
```


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

```
100.2.approx(100.1 + 0.1)            # => true
10e10.approx(10e10+1e-5)             # => true
100.0.approx(100+1e-5)               # => false
```

Discussion

Floating-point math is very precise but, due to the underlying storage mechanism for `Float` objects, not very accurate. Many real numbers (such as 1.9) can't be represented by the floating-point standard. Any attempt to represent such a number will end up using one of the nearby numbers that does have a floating-point representation.

You don't normally see the difference between `1.9` and `1.8 + 0.1`, because `Float#to_s` rounds them both off to "1.9". You can see the difference by using `Kernel#printf` to display the two expressions to many decimal places:

```
printf("%.55f", 1.9)
# 1.8999999999999999911821580299874767661094665527343750000
printf("%.55f", 1.8 + 0.1)
# 1.9000000000000001332267629550187848508358001708984375000
```

Both numbers straddle 1.9 from opposite ends, unable to accurately represent the number they should both equal. Note that the difference between the two numbers is precisely `Float::EPSILON`:

```
Float::EPSILON                       # => 2.22044604925031e-16
(1.8 + 0.1) - 1.9                    # => 2.22044604925031e-16
```

This `EPSILON`'s worth of inaccuracy is often too small to matter, but it does when you're doing comparisons. `1.9+Float::EPSILON` is not equal to `1.9-Float::EPSILON`, even if (in this case) both are attempts to represent the same number. This is why most floating-point numbers are compared in relative terms.

The most efficient way to do a relative comparison is to see whether the two numbers differ by more than an specified error range, using code like this:

```
class Float
  def absolute_approx(other, epsilon=Float::EPSILON)
    return (other-self).abs <= epsilon
  end
end

(1.8 + 0.1).absolute_approx(1.9)     # => true
10e10.absolute_approx(10e10+1e-5)    # => false
```

The default value of `epsilon` works well for numbers close to 0, but for larger numbers the default value of `epsilon` will be too small. Any other value of `epsilon` you might specify will only work well within a certain range.

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

Thus, `Float#approx`, the recommended solution, compares both absolute and relative magnitude. As numbers get bigger, so does the allowable margin of error for two numbers to be considered "equal." Its default `relative_epsilon` allows numbers between 2 and 3 to differ by twice the value of `Float::EPSILON`. Numbers between 3 and 4 can differ by three times the value of `Float::EPSILON`, and so on.

A very small value of `relative_epsilon` is good for mathematical operations, but if your data comes from a real-world source like a scientific instrument, you can increase it. For instance, a Ruby script may track changes in temperature read from a thermometer that's only 99.9% accurate. In this case, `relative_epsilon` can be set to 0.001, and everything beyond that point discarded as noise.

```
98.6.approx(98.66)                          # => false
98.6.approx(98.66, 0.001)                   # => true
```

See Also

- Recipe 2.3, "Representing Numbers to Arbitrary Precision," has more information on `BigDecimal` numbers
- If you need to represent a fraction with an infinite decimal expansion, use a `Rational` number (see Recipe 2.4, "Representing Rational Numbers")
- "Comparing floating-point numbers" by Bruce Dawson has an excellent (albeit C-centric) overview of the tradeoffs involved in different ways of doing floating-point comparisons (http://www.cygnus-software.com/papers/comparingfloats/comparingfloats.htm)

Recipe 2.3. Representing Numbers to Arbitrary Precision

Problem

You're doing high-precision arithmetic, and floating-point numbers are not precise enough.

Solution

A `BigDecimal` number can represent a real number to an arbitrary number of decimal places.

```
require 'bigdecimal'

BigDecimal("10").to_s               # => "0.1E2"
BigDecimal("1000").to_s             # => "0.1E4"
BigDecimal("1000").to_s("F")        # => "1000.0"
```


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

```
BigDecimal("0.123456789").to_s       # => "0.123456789E0"
```

Compare how `Float` and `BigDecimal` store the same high-precision number:

```
nm = "0.123456789012345678901234567890123456789"
nm.to_f                          # => 0.123456789012346
BigDecimal(nm).to_s
# => "0.123456789012345678901234567890123456789E0"
```

Discussion

`BigDecimal` numbers store numbers in scientific notation format. A `BigDecimal` consists of a sign (positive or negative), an arbitrarily large decimal fraction, and an arbitrarily large exponent. This is similar to the way floating-point numbers are stored, but a double-precision floating-point implementation like Ruby's cannot represent an exponent less than `Float::MIN_EXP` (–1021) or greater than `Float::MAX_EXP` (1024). Float objects also can't represent numbers at a greater precision than `Float::EPSILON`, or about $2.2*10^{-16}$.

You can use `BigDecimal#split` to split a `BigDecimal` object into the parts of its scientific-notation representation. It returns an array of four numbers: the sign (1 for positive numbers,–1 for negative numbers), the fraction (as a string), the base of the exponent (which is always 10), and the exponent itself.

```
BigDecimal("105000").split
# => [1, "105", 10, 6]
# That is, 0.105*(10**6)

BigDecimal("-0.005").split
# => [-1, "5", 10, -2]
# That is, -1 * (0.5*(10**-2))
```

A good way to test different precision settings is to create an infinitely repeating decimal like 2/3, and see how much of it gets stored. By default, `BigDecimals` give 16 digits of precision, roughly comparable to what a Float can give.

```
(BigDecimal("2") / BigDecimal("3")).to_s
# => "0.6666666666666667E0"

2.0/3
# => 0.666666666666667
```

You can store additional significant digits by passing in a second argument `n` to the `BigDecimal` constructor. `BigDecimal` precision is allocated in chunks of four decimal digits. Values of `n` from 1 to 4 make a `BigDecimal` use the default precision of 16 digits. Values from 5 to 8 give 20 digits of precision, values from 9 to 12 give 24 digits, and so on:

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

```
def two_thirds(precision)
  (BigDecimal("2", precision) / BigDecimal("3")).to_s
end

two_thirds(1)                          # => "0.6666666666666667E0"
two_thirds(4)                          # => "0.6666666666666667E0"
two_thirds(5)                          # => "0.66666666666666666667E0"
two_thirds(9)                          # => "0.666666666666666666666667E0"
two_thirds(13)                         # => "0.6666666666666666666666666667E0"
```

Not all of a number's significant digits may be used. For instance, Ruby considers `BigDecimal`("2") and `BigDecimal`("2.000000000000") to be equal, even though the second one has many more significant digits.

You can inspect the precision of a number with `BigDecimal#precs`. This method returns an array of two elements: the number of significant digits actually being used, and the toal number of significant digits. Again, since significant digits are allocated in blocks of four, both of these numbers will be multiples of four.

```
BigDecimal("2").precs                      # => [4, 8]
BigDecimal("2.000000000000").precs         # => [4, 20]
BigDecimal("2.000000000001").precs         # => [16, 20]
```

If you use the standard arithmetic operators on `BigDecimals`, the result is a `BigDecimal` accurate to the largest possible number of digits. Dividing or multiplying one `BigDecimal` by another yields a `BigDecimal` with more digits of precision than either of its parents, just as would happen on a pocket calculator.

```
(a = BigDecimal("2.01")).precs         # => [8, 8]
(b = BigDecimal("3.01")).precs         # => [8, 8]

(product = a * b).to_s("F")            # => "6.0501"
product.precs                          # => [8, 24]
```

To specify the number of significant digits that should be retained in an arithmetic operation, you can use the methods `add`, `sub`, `mul`, and `div` instead of the arithmetic operators.

```
two_thirds = (BigDecimal("2", 13) / 3)
two_thirds.to_s                # => "0.6666666666666666666666666666666666667E0"

(two_thirds + 1).to_s          # => "0.1666666666666666666666666666666666666667E1"

two_thirds.add(1, 1).to_s      # => "0.2E1"
two_thirds.add(1, 4).to_s      # => "0.1667E1"
```

Either way, `BigDecimal` math is significantly slower than floating-point math. Not only are `BigDecimals` allowed to have more significant digits than floats, but `BigDecimals` are stored as an array of decimal digits, while floats are stored in a binary encoding and manipulated with binary arithmetic.

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

The `BigMath` module in the Ruby standard library defines methods for performing arbitrary-precision mathematical operations on `BigDecimal` objects. It defines power-related methods like `sqrt`, `log`, and `exp`, and trigonometric methods like `sin`, `cos`, and `atan`.

All of these methods take as an argument a number `prec` indicating how many digits of precision to retain. They may return a `BigDecimal` with more than `prec` significant digits, but only `prec` of those digits are guaranteed to be accurate.

```
require 'bigdecimal/math'
include BigMath
two = BigDecimal("2")
BigMath::sqrt(two, 10).to_s("F")        # => "1.41421356237309504880168835l5"
```

That code gives 28 decimal places, but only 10 are guaranteed accurate (because we passed in an `n` of 10), and only 24 are actually accurate. The square root of 2 to 28 decimal places is actually 1.41421356237309504880168872 42. We can get rid of the inaccurate digits with `BigDecimal#round`:

```
BigMath::sqrt(two, 10).round(10).to_s("F") # => "1.4142135624"
```

We can also get a more precise number by increasing `n`:

```
BigMath::sqrt(two, 28).round(28).to_s("F")   # => "1.41421356237309504880168872 42"
```

`BigMath` also annotates `BigDecimal` with class methods `BigDecimal.PI` and `BigDecimal.E`. These methods construct `BigDecimals` of those transcendental numbers at any level of precision.

```
Math::PI                            # => 3.14159265358979
Math::PI.class                      # => Float
BigDecimal.PI(1).to_s               # => "0.3141592653589793236419814396560 3E1"
BigDecimal.PI(20).to_s
# => "0.31415926535897932384626433832795028839198592935214 27E1"
```

See Also

- At the time of writing, `BigMath::log` was very slow for `BigDecimals` larger than about 10; see Recipe 2.7, "Taking Logarithms," for a much faster implementation
- See Recipe 2.4, "Representing Rational Numbers," if you need to exactly represent a rational number with an infinite decimal expansion, like 2/3
- The `BigDecimal` library reference is extremely useful; if you look at the generated RDoc for the Ruby standard library, `BigDecimal` looks almost undocumented, but it actually has a comprehensive reference file (in English and Japanese): it's just not

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

in RDoc format, so it doesn't get picked up; this document is available in the Ruby source package, or do a web search for "BigDecimal: An extension library for Ruby"

Recipe 2.4. Representing Rational Numbers

Problem

You want to precisely represent a rational number like 2/3, one that has no finite decimal expansion.

Solution

Use a `Rational` object; it represents a rational number as an integer numerator and denominator.

```
float = 2.0/3.0                    # => 0.666666666666667
float * 100                        # => 66.6666666666667
float * 100 / 42                   # => 1.58730158730159

require 'rational'
rational = Rational(2, 3)          # => Rational(2, 3)
rational.to_f                      # => 0.666666666666667
rational * 100                     # => Rational(200, 3)
rational * 100 / 42                # => Rational(100, 63)
```

Discussion

`Rational` objects can store numbers that can't be represented in any other form, and arithmetic on `Rational` objects is completely precise.

Since the numerator and denominator of a `Rational` can be `Bignum`s, a `Rational` object can also represent numbers larger and smaller than those you can represent in floating-point. But math on `BigDecimal` objects is faster than on `Rationals`. `BigDecimal` objects are also usually easier to work with than `Rationals`, because most of us think of numbers in terms of their decimal expansions.

You should only use `Rational` objects when you need to represent rational numbers with perfect accuracy. When you do, be sure to use only `Rationals, Fixnums`, and `Bignums` in your calculations. Don't use any `BigDecimals` or floating-point numbers: arithmetic operations between a `Rational` and those types will return floating-point numbers, and you'll have lost precision forever.

```
10 + Rational(2,3)                 # => Rational(32, 3)
require 'bigdecimal'
BigDecimal('10') + Rational(2,3)   # => 10.6666666666667
```


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

The methods in Ruby's `Math` module implement operations like square root, which usually give irrational results. When you pass a `Rational` number into one of the methods in the `Math` module, you get a floating-point number back:

```
Math::sqrt(Rational(2,3))            # => 0.816496580927726
Math::sqrt(Rational(25,1))           # => 5.0
Math::log10(Rational(100, 1))        # => 2.0
```

The `mathn` library adds miscellaneous functionality to Ruby's math functions. Among other things, it modifies the `Math::sqrt` method so that if you pass in a square number, you get a `Fixnum` back instead of a `Float`. This preserves precision whenever possible:

```
require 'mathn'
Math::sqrt(Rational(2,3))              # => 0.816496580927726
Math::sqrt(Rational(25,1))             # => 5
Math::sqrt(25)                         # => 5
Math::sqrt(25.0)                       # => 5.0
```

See Also

- The `rfloat` third-party library lets you use a `Float`-like interface that's actually backed by `Rational` (http://blade.nagaokaut.ac.jp/~sinara/ruby/rfloat/)
- RCR 320 proposes better interoperability between `Rationals` and floating-point numbers, including a `Rational#approximate` method that will let you convert the floating-point number `0.1` into `Rational(1, 10)` (http://www.rcrchive.net/rcr/show/320)

Recipe 2.5. Generating Random Numbers

Problem

You want to generate pseudorandom numbers, select items from a data structure at random, or repeatedly generate the same "random" numbers for testing purposes.

Solution

Use the `Kernel#rand` function with no arguments to select a psuedorandom floating-point number from a uniform distribution between 0 and 1.

```
rand                                 # => 0.517297883846589
rand                                 # => 0.946962603814814
```

Pass in a single integer argument *n* to `Kernel#rand`, and it returns an integer between 0 and *n–1*:

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

```
rand(5)                              # => 0
rand(5)                              # => 4
rand(5)                              # => 3
rand(1000)                           # => 39
```

Discussion

You can use the single-argument form of `Kernel#rand` to build many common tasks based on randomness. For instance, this code selects a random item from an array.

```
a = ['item1', 'item2', 'item3']
a[rand(a.size)]                      # => "item3"
```

To select a random key or value from a hash, turn the keys or values into an array and select one at random.

```
m = { :key1 => 'value1',
      :key2 => 'value2',
      :key3 => 'value3' }
values = m.values
values[rand(values.size)]            # => "value1"
```

This code generates pronounceable nonsense words:

```
def random_word
 letters = { ?v => 'aeiou',
             ?c => 'bcdfghjklmnprstvwyz' }
 word = ''
 'cvcvcvc'.each_byte do |x|
   source = letters[x]
   word << source[rand(source.length)].chr
 end
 return word
end

random_word                          # => "josuyip"
random_word                          # => "haramic"
```

The Ruby interpreter initializes its random number generator on startup, using a seed derived from the current time and the process number. To reliably generate the same random numbers over and over again, you can set the random number seed manually by calling the `Kernel#srand` function with the integer argument of your choice. This is useful when you're writing automated tests of "random" functionality:

```
#Some random numbers based on process number and current time
rand(1000)                           # => 187
rand(1000)                           # => 551
rand(1000)                           # => 911

#Start the seed with the number 1
srand 1
rand(1000)                           # => 37
rand(1000)                           # => 235
rand(1000)                           # => 908
```


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

```
#Reset the seed to its previous state
srand 1
rand(1000)                          # => 37
rand(1000)                          # => 235
rand(1000)                          # => 908
```

See Also

- Recipe 4.10, "Shuffling an Array"
- Recipe 5.11, "Choosing Randomly from a Weighted List"
- Recipe 6.9, "Picking a Random Line from a File"
- The Facets library implements many methods for making random selections from data structures: `Array#pick`, `Array#rand_subset`, `Hash#rand_pair`, and so on; it also defines `String.random` for generating random strings
- Christian Neukirchen's `rand.rb` also implements many random selection methods (http://chneukirchen.org/blog/static/projects/rand.html)

Recipe 2.6. Converting Between Numeric Bases

Problem

You want to convert numbers from one base to another.

Solution

You can convert specific binary, octal, or hexadecimal numbers to decimal by representing them with the `0b`, `0o`, or `0x` prefixes:

```
0b100                               # => 4
0o100                               # => 64
0x100                               # => 256
```

You can also convert between decimal numbers and string representations of those numbers in any base from 2 to 36. Simply pass the base into `String#to_i` or `Integer#to_s`.

Here are some conversions between string representations of numbers in various bases, and the corresponding decimal numbers:

```
"1045".to_i(10)                     # => 1045
"-1001001".to_i(2)                  # => -73
"abc".to_i(16)                      # => 2748
"abc".to_i(20)                      # => 4232
"number".to_i(36)                   # => 1442151747
"zz1z".to_i(36)                     # => 1678391
"abcdef".to_i(16)                   # => 11259375
"AbCdEf".to_i(16)                   # => 11259375
```


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

Here are some reverse conversions of decimal numbers to the strings that represent those numbers in various bases:

```
42.to_s(10)                     # => "42"
-100.to_s(2)                    # => "-1100100"
255.to_s(16)                    # => "ff"
1442151747.to_s(36)             # => "number"
```

Some invalid conversions:

```
"6".to_i(2)                     # => 0
"0".to_i(1)                     # ArgumentError: illegal radix 1
40.to_s(37)                     # ArgumentError: illegal radix 37
```

Discussion

`String#to_i` can parse and `Integer#to_s` can create a string representation in every common integer base: from binary (the familiar base 2, which uses only the digits 0 and 1) to hexatridecimal (base 36). Hexatridecimal uses the digits 0–9 and the letters a–z; it's sometimes used to generate alphanumeric mneumonics for long numbers.

The only commonly used counting systems with bases higher than 36 are the variants of base-64 encoding used in applications like MIME mail attachments. These usually encode strings, not numbers; to encode a string in MIME-style base-64, use the `base64` library.

See Also

- Recipe 12.5, "Adding Graphical Context with Sparklines," and Recipe 14.5, "Sending Mail," show how to use the `base64` library

Recipe 2.7. Taking Logarithms

Problem

You want to take the logarithm of a number, possibly a huge one.

Solution

`Math.log` calculates the natural log of a number: that is, the log base *e*.

```
Math.log(1)                     # => 0.0
Math.log(Math::E)               # => 1.0
Math.log(10)                    # => 2.30258509299405
Math::E ** Math.log(25)         # => 25.0
```


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

`Math.log10` calculates the log base 10 of a number:

```
Math.log10(1)                          # => 0.0
Math.log10(10)                         # => 1.0
Math.log10(10.1)                       # => 1.00432137378264
Math.log10(1000)                       # => 3.0
10 ** Math.log10(25)                   # => 25.0
```

To calculate a logarithm in some other base, use the fact that, for any bases b_1 and b_2, $\log b_1(x) = \log b_2(x) / \log b_2(k)$.

```
module Math
  def Math.logb(num, base)
    log(num) / log(base)
  end
end
```

Discussion

A logarithm function inverts an exponentiation function. The log base *k* of *x*,or $\log k(^x)$, is the number that gives *x* when raised to the *k* power. That is, `Math. log10(1000)==3.0` because 10 cubed is `1000.Math.log(Math::E)==1` because *e* to the first power is *e*.

The logarithm functions for all numeric bases are related (you can get from one base to another by dividing by a constant factor), but they're used for different purposes.

Scientific applications often use the natural log: this is the fastest log implementation in Ruby. The log base 10 is often used to visualize datasets that span many orders of magnitude, such as the pH scale for acidity and the Richter scale for earthquake intensity. Analyses of algorithms often use the log base 2, or binary logarithm.

If you intend to do a lot of algorithms in a base that Ruby doesn't support natively, you can speed up the calculation by precalculating the dividend:

```
dividend = Math.log(2)
(1..6).collect { |x| Math.log(x) / dividend }
# => [0.0, 1.0, 1.58496250072116, 2.0, 2.32192809488736, 2.58496250072116]
```

The logarithm functions in `Math` will only accept integers or floating-point numbers, not `BigDecimal` or `Bignum` objects. This is inconvenient since logarithms are often used to make extremely large numbers managable. The `BigMath` module has a function to take the natural logarithm of a `BigDecimal` number, but it's very slow.

Here's a fast drop-in replacement for `BigMath::log` that exploits the logarithmic identity `log(x*y)==log(x) + log(y)`. It decomposes a `BigDecimal` into three much smaller

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

numbers, and operates on those numbers. This avoids the cases that give `BigMath::log` such poor performance.

```
require 'bigdecimal'
require 'bigdecimal/math'
require 'bigdecimal/util'

module BigMath
  alias :log_slow :log
  def log(x, prec)
    if x <= 0 || prec <= 0
      raise ArgumentError, "Zero or negative argument for log"
    end
    return x if x.infinite? || x.nan?
    sign, fraction, power, exponent = x.split
    fraction = BigDecimal(".#{fraction}")
    power = power.to_s.to_d
    log_slow(fraction, prec) + (log_slow(power, prec) * exponent)
  end
end
```

Like `BigMath::log`, this implementation returns a `BigMath` accurate to at least `prec` digits, but containing some additional digits which might not be accurate. To avoid giving the impression that the result is more accurate than it is, you can round the number to `prec` digits with `BigDecimal#round`.

```
include BigMath

number = BigDecimal("1234.5678")
Math.log(number)                            # => 7.11847622829779

prec = 50
BigMath.log_slow(number, prec).round(prec).to_s("F")
# => "7.11847622829778629250879253638708184134073214145175"

BigMath.log(number, prec).round(prec).to_s("F")
# => "7.11847622829778629250879253638708184134073214145175"
BigMath.log(number ** 1000, prec).round(prec).to_s("F")
# => "7118.47622829778629250879253638708184134073214145175161"
```

As before, calculate a log other than the natural log by dividing by `BigMath.log(base)` or `BigMath.log_slow(base)`.

```
huge_number = BigDecimal("1000") ** 1000
base = BigDecimal("10")
BigMath.log(huge_number, 100) / BigMath.log(base, 100)).to_f
# => 3000.0
```

How does it work? The internal representation of a `BigDecimal` is as a number in scientific notation: `fraction*10**power`. Because `log(x*y)=log(x) + log(y)`, the log of such a number is `log(fraction) + log(10**power)`.

`10**power` is just 10 multiplied by itself `power` times (that is, `10*10*10*...*10`). Again, `log(x*y)=log(x) + log(y)`, so `log(10*10*10*...*10)=log(10)+log(10) +`

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

`log(10)+...+log(10),or log(10)*power`. This means we can take the logarithm of a huge `BigDecimal` by taking the logarithm of its (very small) fractional portion and the logarithm of 10.

See Also

- Mathematicians used to spend years constructing tables of logarithms for scientific and engineering applications; so if you find yourself doing a boring job, be glad you don't have to do that (see http://en.wikipedia.org/wiki/Logarithm#Tables_of_logarithms)

Recipe 2.8. Finding Mean, Median, and Mode

Problem

You want to find the average of an array of numbers: its mean, median, or mode.

Solution

Usually when people speak of the "average" of a set of numbers they're referring to its mean, or arithmetic mean. The mean is the sum of the elements divided by the number of elements.

```
def mean(array)
  array.inject(array.inject(0) { |sum, x| sum += x } / array.size.to_f
end

mean([1,2,3,4])                     # => 2.5
mean([100,100,100,100.1])           # => 100.025
mean([-100, 100])                   # => 0.0
mean([3,3,3,3])                     # => 3.00
```

The median is the item *x* such that half the items in the array are greater than *x* and the other half are less than *x*. Consider a sorted array: if it contains an odd number of elements, the median is the one in the middle. If the array contains an even number of elements, the median is defined as the mean of the two middle elements.

```
def median(array, already_sorted=false)
  return nil if array.empty?
  array = array.sort unless already_sorted
  m_pos = array.size / 2
  return array.size % 2 == 1 ? array[m_pos] : mean(array[m_pos-1..m_pos])
end

median([1,2,3,4,5])                 # => 3
median([5,3,2,1,4])                 # => 3
median([1,2,3,4])                   # => 2.5
median([1,1,2,3,4])                 # => 2
median([2,3,-100,100])              # => 2.5
median([1, 1, 10, 100, 1000])       # => 10
```


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

The mode is the single most popular item in the array. If a list contains no repeated items, it is not considered to have a mode. If an array contains multiple items at the maximum frequency, it is "multimodal." Depending on your application, you might handle each mode separately, or you might just pick one arbitrarily.

```
def modes(array, find_all=true)
  histogram = array.inject(Hash.new(0)) { |h, n| h[n] += 1; h }
  modes = nil
  histogram.each_pair do |item, times|
    modes << item if modes && times == modes[0] and find_all
    modes = [times, item] if (!modes && times>1) or (modes && times>modes[0])
  end
  return modes ? modes[1…modes.size] : modes
end

modes([1,2,3,4])                    # => nil
modes([1,1,2,3,4])                  # => [1]
modes([1,1,2,2,3,4])                # => [1, 2]
modes([1,1,2,2,3,4,4])              # => [1, 2, 4]
modes([1,1,2,2,3,4,4], false)       # => [1]
modes([1,1,2,2,3,4,4,4,4,4])        # => [4]
```

Discussion

The mean is the most popular type of average. It's simple to calculate and to understand. The implementation of `mean` given above always returns a floating-point number object. It's a good general-purpose implementation because it lets you pass in an array of `Fixnums` and get a fractional average, instead of one rounded to the nearest integer. If you want to find the mean of an array of `BigDecimal` or `Rational` objects, you should use an implementation of `mean` that omits the final `to_f` call:

```
def mean_without_float_conversion(array)
  array.inject(0) { |x, sum| sum += x } / array.size
end
require 'rational'
numbers = [Rational(2,3), Rational(3,4), Rational(6,7)]
mean(numbers)
# => 0.757936507936508
mean_without_float_conversion(numbers)
# => Rational(191, 252)
```

The median is mainly useful when a small proportion of outliers in the dataset would make the mean misleading. For instance, government statistics usually show "median household income" instead of "mean household income." Otherwise, a few super-wealthy households would make everyone else look much richer than they are. The example below demonstrates how the mean can be skewed by a few very high or very low outliers.

```
mean([1, 100, 100000])              # => 33367.0
median([1, 100, 100000])            # => 100

mean([1, 100, -1000000])            # => -333299.666666667
median([1, 100, -1000000])          # => 1
```


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

The mode is the only definition of "average" that can be applied to arrays of arbitrary objects. Since the mean is calculated using arithmetic, an array can only be said to have a mean if all of its members are numeric. The median involves only comparisons, except when the array contains an even number of elements: then, calculating the median requires that you calculate the mean.

If you defined some other way to take the median of an array with an even number of elements, you could take the median of `Arrays` of strings:

```
median(["a", "z", "b", "l", "m", "j", "b"])
# => "j"
median(["a", "b", "c", "d"])
# TypeError: String can't be coerced into Fixnum
```

The standard deviation

A concept related to the mean is the standard deviation, a quantity that measures how close the dataset as a whole is to the mean. When a mean is distorted by high or low outliers, the corresponding standard deviation is high. When the numbers in a dataset cluster closely around the mean, the standard deviation is low. You won't be fooled by a misleading mean if you also look at the standard deviation.

```
def mean_and_standard_deviation(array)
  m = mean(array)
  variance = array.inject(0) { |variance, x| variance += (x - m) ** 2 }
  return m, Math.sqrt(variance/(array.size-1))
end

#All the items in the list are close to the mean, so the standard
#deviation is low.
mean_and_standard_deviation([1,2,3,1,1,2,1])
# => [1.57142857142857, 0.786795792469443]
#The outlier increases the mean, but also increases the standard deviation.
mean_and_standard_deviation([1,2,3,1,1,2,1000])
# => [144.285714285714, 377.33526837801]
```

A good rule of thumb is that two-thirds (about 68 percent) of the items in a dataset are within one standard deviation of the mean, and almost all (about 95 percent) of the items are within two standard deviations of the mean.

See Also

- "Programmers Need to Learn Statistics or I Will Kill Them All," by Zed Shaw (http://www.zedshaw.com/blog/programming/programmer_stats.html)
- More Ruby implementations of simple statistical measures (http://dada.perl.it/shootout/moments.ruby.html)
- To do more complex statistical analysis in Ruby, try the Ruby bindings to the GNU Scientific Library (http://ruby-gsl.sourceforge.net/)

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

- The `Stats` class in the Mongrel web server (http://mongrel.rubyforge.org) implements other algorithms for calculating mean and standard deviation, which are faster if you need to repeatedly calculate the mean of a growing series

Recipe 2.9. Converting Between Degrees and Radians

Problem

The trigonometry functions in Ruby's `Math` library take input in radians (2π radians in a circle). Most real-world applications measure angles in degrees (360 degrees in a circle). You want an easy way to do trigonometry with degrees.

Solution

The simplest solution is to define a conversion method in `Numeric` that will convert a number of degrees into radians.

```
class Numeric
  def degrees
    self * Math::PI / 180
  end
end
```

You can then treat any numeric object as a number of degrees and convert it into the corresponding number of radians, by calling its `degrees` method. Trigonometry on the result will work as you'd expect:

```
90.degrees                          # => 1.5707963267949
Math::tan(45.degrees)               # => 1.0
Math::cos(90.degrees)               # => 6.12303176911189e-17
Math::sin(90.degrees)               # => 1.0
Math::sin(89.9.degrees)             # => 0.999998476913288

Math::sin(45.degrees)               # => 0.707106781186547
Math::cos(45.degrees)               # => 0.707106781186548
```

Discussion

I named the conversion method `degrees` by analogy to the methods like `hours` defined by Rails. This makes the code easy to read, but if you look at the actual numbers, it's not obvious why `45.degrees` should equal the floating-point number 0.785398163397448.

If this troubles you, you could name the method something like `degrees_to_radians`. Or you could use Lucas Carlson's `units` gem, which lets you define customized unit conversions, and tracks which unit is being used for a particular number.

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

```
require 'rubygems'
require 'units/base'

class Numeric
  remove_method(:degrees) # Remove the implementation given in the Solution
  add_unit_conversions(:angle => { :radians => 1, :degrees => Math::PI/180 })
  add_unit_aliases(:angle => { :degrees => [:degree], :radians => [:radian] })
end

90.degrees                          # => 90.0
90.degrees.unit                     # => :degrees
90.degrees.to_radians               # => 1.5707963267949
90.degrees.to_radians.unit          # => :radians

1.degree.to_radians                 # => 0.0174532925199433
1.radian.to_degrees                 # => 57.2957795130823
```

The units you define with the `units` gem do nothing but make your code more readable. The trigonometry methods don't understand the units you've defined, so you'll still have to give them numbers in radians.

```
# Don't do this:
Math::sin(90.degrees)               # => 0.893996663600558

# Do this:
Math::sin(90.degrees.to_radians)    # => 1.0
```

Of course, you could also change the trigonometry methods to be aware of units:

```
class << Math
  alias old_sin sin
  def sin(x)
    old_sin(x.unit == :degrees ? x.to_radians : x)
  end
end

90.degrees                          # => 90.0
Math::sin(90.degrees)               # => 1.0
Math::sin(Math::PI/2.radians)       # => 1.0
Math::sin(Math::PI/2)               # => 1.0
```

That's probably overkill, though.

See Also

- Recipe 8.9, "Converting and Coercing Objects to Different Types"
- The Facets More library (available as the `facets_more gem`) also has a `Units` module

Recipe 2.10. Multiplying Matrices

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

Problem

You want to turn arrays of arrays of numbers into mathematical matrices, and multiply the matrices together.

Solution

You can create `Matrix` objects from arrays of arrays, and multiply them together with the `*` operator:

```
require 'matrix'
require 'mathn'

a1 = [[1, 1, 0, 1],
      [2, 0, 1, 2],
      [3, 1, 1, 2]]
m1 = Matrix[*a1]
# => Matrix[[1, 1, 0, 1], [2, 0, 1, 2], [3, 1, 1, 2]]

a2 = [[1, 0],
      [3, 1],
      [1, 0],
      [2, 2.5]]
m2 = Matrix[*a2]
# => Matrix[[1, 0], [3, 1], [1, 0], [2, 2.5]]

m1 * m2
# => Matrix[[6, 3.5], [7, 5.0], [11, 6.0]]
```

Note the unusual syntax for creating a `Matrix` object: you pass the rows of the matrix into the array indexing operator, not into `Matrix#new` (which is private).

Discussion

Ruby's `Matrix` class overloads the arithmetic operators to support all the basic matrix arithmetic operations, including multiplication, between matrices of compatible dimension. If you perform an arithmetic operation on incompatible matrices, you'll get an `ExceptionForMatrix::ErrDimensionMismatch`.

Multiplying one matrix by another is simple enough, but multiplying a chain of matrices together can be faster or slower depending on the order in which you do the multiplications. This follows from the fact that multiplying a matrix with dimensions K x M, by a matrix with dimensions MxN, requires K * M * N operations and gives a matrix with dimension K * N. If K is large for some matrix, you can save time by waiting til the end before doing multiplications involving that matrix.

Consider three matrices A, B, and C, which you want to multiply together. A has 100 rows and 20 columns. B has 20 rows and 10 columns. C has 10 rows and one column.

Since matrix multiplication is associative, you'll get the same results whether you multiply A by B and then the result by C, or multiply B by C and then the result by A. But multiplying

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

A by B requires 20,000 operations (100 * 20 * 10), and multiplying (AB) by C requires another 1,000 (100 * 10 * 1). Multiplying B by C only requires 200 operations (20 * 10 * 1), and multiplying the result by A requires 2,000 more (100 * 20 * 1). It's almost 10 times faster to multiply A(BC) instead of the naive order of (AB)C.

That kind of potential savings justifies doing some up-front work to find the best order for the multiplication. Here is a method that recursively figures out the most efficient multiplication order for a list of `Matrix` objects, and another method that actually carries out the multiplications. They share an array containing information about where to divide up the list of matrices: where to place the parentheses, if you will.

```
class Matrix
  def self.multiply(*matrices)
    cache = []
    matrices.size.times { cache << [nil] * matrices.size }
    best_split(cache, 0, matrices.size-1, *matrices)
    multiply_following_cache(cache, 0, matrices.size-1, *matrices)
  end
```

Because the methods that do the actual work pass around recursion arguments that the end user doesn't care about, I've created `Matrix.multiply`, a wrapper method for the methods that do the real work. These methods are defined below (`Matrix.best_split` and `Matrix.multiply_following_cache`).
`Matrix.multiply_following_cache` assumes that the optimal way to multiply that list of `Matrix` objects has already been found and encoded in a variable `cache`. It recursively performs the matrix multiplications in the optimal order, as determined by the cache.

```
:private
def self.multiply_following_cache(cache, chunk_start, chunk_end, *matrices)
  if chunk_end == chunk_start
    # There's only one matrix in the list; no need to multiply.
    return matrices[chunk_start]
  elsif chunk_end-chunk_start == 1
    # There are only two matrices in the list; just multiply them together.
    lhs, rhs = matrices[chunk_start..chunk_end]
  else
    # There are more than two matrices in the list. Look in the
    # cache to see where the optimal split is located. Multiply
    # together all matrices to the left of the split (recursively,
    # in the optimal order) to get our equation's left-hand
    # side. Similarly for all matrices to the right of the split, to
    # get our right-hand side.
    split_after = cache[chunk_start][chunk_end][1]
    lhs = multiply_following_cache(cache, chunk_start, split_after, *matrices)
    rhs = multiply_following_cache(cache, split_after+1, chunk_end, *matrices)
  end

  # Begin debug code: this illustrates the order of multiplication,
  # showing the matrices in terms of their dimensions rather than their
  # (possibly enormous) contents.
  if $DEBUG
    lhs_dim = "#{lhs.row_size}x#{lhs.column_size}"
    rhs_dim = "#{rhs.row_size}x#{rhs.column_size}"
    cost = lhs.row_size * lhs.column_size * rhs.column_size
    puts "Multiplying #{lhs_dim} by #{rhs_dim}: cost #{cost}"
```


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

```
    end

    # Do a matrix multiplication of the two matrices, whether they are
    # the only two matrices in the list or whether they were obtained
    # through two recursive calls.
    return lhs * rhs
  end
```

Finally, here's the method that actually figures out the best way of splitting up the multiplcations. It builds the cache used by the `multiply_following_cache` method defined above. It also uses the cache as it builds it, so that it doesn't solve the same subproblems over and over again.

```
  def self.best_split(cache, chunk_start, chunk_end, *matrices)
    if chunk_end == chunk_start
      cache[chunk_start][chunk_end] = [0, nil]
    end
    return cache[chunk_start][chunk_end] if cache[chunk_start][chunk_end]

    #Try splitting the chunk at each possible location and find the
    #minimum cost of doing the split there. Then pick the smallest of
    #the minimum costs: that's where the split should actually happen.
    minimum_costs = []
    chunk_start.upto(chunk_end-1) do |split_after|
      lhs_cost = best_split(cache, chunk_start, split_after, *matrices)[0]
      rhs_cost = best_split(cache, split_after+1, chunk_end, *matrices)[0]

      lhs_rows = matrices[chunk_start].row_size
      rhs_rows = matrices[split_after+1].row_size
      rhs_cols = matrices[chunk_end].column_size
      merge_cost = lhs_rows * rhs_rows * rhs_cols
      cost = lhs_cost + rhs_cost + merge_cost
      minimum_costs << cost
    end
    minimum = minimum_costs.min
    minimum_index = chunk_start + minimum_costs.index(minimum)
    return cache[chunk_start][chunk_end] = [minimum, minimum_index]
  end
end
```

A simple test confirms the example set of matrices spelled out earlier. Remember that we had a 100 x 20 matrix (A), a 20 x 10 matrix (B), and a 20 x 1 matrix (C). Our method should be able to figure out that it's faster to multiply A(BC) than the naive multiplication (AB)C. Since we don't care about the contents of the matrices, just the dimensions, we'll first define some helper methods that make it easy to generate matrices with specific dimensions but random contents.

```
class Matrix
  # Creates a randomly populated matrix with the given dimensions.
  def self.with_dimensions(rows, cols)
    a = []
    rows.times { a << []; cols.times { a[-1] << rand(10) } }
    return Matrix[*a]
  end

  # Creates an array of matrices that can be multiplied together
  def self.multipliable_chain(*rows)
    matrices = []
    0.upto(rows.size-2) do |i|
      matrices << Matrix.with_dimensions(rows[i], rows[i+1])
    end
```


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

```
    return matrices
  end
end
```

After all that, the test is kind of anticlimactic:

```
# Create an array of matrices 100x20, 20x10, 10x1.
chain = Matrix.multipliable_chain(100, 20, 10, 1)

# Multiply those matrices two different ways, giving the same result.
Matrix.multiply(*chain) == (chain[0] * chain[1] * chain[2])
# Multiplying 20x10 by 10x1: cost 200
# Multiplying 100x20 by 20x1: cost 2000
# => true
```

We can use the `Benchmark` library to verify that matrix multiplication goes much faster when we do the multiplications in the right order:

```
# We'll generate the dimensions and contents of the matrices randomly,
# so no one can accuse us of cheating.
dimensions = []
10.times { dimensions << rand(90)+10 }
chain = Matrix.multipliable_chain(*dimensions)

require 'benchmark'
result_1 = nil
result_2 = nil
Benchmark.bm(11) do |b|
  b.report("Unoptimized") do
    result_1 = chain[0]
    chain[1..chain.size].each { |c| result_1 *= c }
  end
  b.report("Optimized") { result_2 = Matrix.multiply(*chain) }
end
#                user      system       total        real
# Unoptimized   4.350000    0.400000   4.750000 ( 11.104857)
# Optimized     1.410000    0.110000   1.520000 (  3.559470)

# Both multiplications give the same result.
result_1 == result_2                              # => true
```

See Also

- Recipe 2.11, "Solving a System of Linear Equations," uses matrices to solve linear equations
- For more on benchmarking, see Recipe 17.13, "Benchmarking Competing Solutions"

Recipe 2.11. Solving a System of Linear Equations

Problem

You have a number of linear equations (that is, equations that look like "2x + 10y + 8z = 54"), and you want to figure out the solution: the values of x, y, and z. You have as many equations as you have variables, so you can be certain of a unique solution.

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

Solution

Create two `Matrix` objects. The first `Matrix` should contain the coefficients of your equations (the 2, 10, and 8 of "2x + 10y + 8z = 54"), and the second should contain the constant results (the 54 of the same equation). The numbers in both matrices should be represented as floating-point numbers, rational numbers, or `BigDecimal` objects: anything other than plain Ruby integers.

Then invert the coefficient matrix with `Matrix#inverse`, and multiply the result by the matrix full of constants. The result will be a third `Matrix` containing the solutions to your equations.

For instance, consider these three linear equations in three variables:

```
2x + 10y + 8z = 54
7y + 4z = 30
5x + 5y + 5z = 35
```

To solve these equations, create the two matrices:

```
require 'matrix'
require 'rational'
coefficients = [[2, 10, 8], [0, 7, 4], [5, 5, 5]].collect! do |row|
  row.collect! { |x| Rational(x) }
end
coefficients = Matrix[*coefficients]
# => Matrix[[Rational(2, 1), Rational(10, 1), Rational(8, 1)],
# =>        [Rational(0, 1), Rational(7, 1), Rational(4, 1)],
# =>        [Rational(5, 1), Rational(5, 1), Rational(5, 1)]]

constants = Matrix[[Rational(54)], [Rational(30)], [Rational(35)]]
```

Take the inverse of the coefficient matrix, and multiply it by the results matrix. The result will be a matrix containing the values for your variables.

```
solutions = coefficients.inverse * constants
# => Matrix[[Rational(1, 1)], [Rational(2, 1)], [Rational(4, 1)]]
```

This means that, in terms of the original equations, x=1, y=2, and z=4.

Discussion

This may seem like magic, but it's analagous to how you might use algebra to solve a single equation in a single variable. Such an equation looks something like $Ax = B$: for instance, $6x = 18$. To solve for x, you divide both sides by the coefficient: $\frac{6x}{6} = \frac{18}{6}$

The sixes on the left side of the equation cancel out, and you can show that x is 18/6, or 3.

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

In that case there's only one coefficient and one constant. With n equations in n variables, you have n^2 coefficients and n constants, but by packing them into matrices you can solve the problem in the same way.

Here's a side-by-side comparision of the set of equations from the Solution, and the corresponding matrices created in order to solve the system of equations.

```
2x + 10y + 8z = 54 | [ 2 10 8] [x]   [54]
x + 7y + 4z = 31   | [ 1 7  4] [y] = [31]
5x + 5y + 5z = 35  | [ 5 5  5] [z]   [35]
```

If you think of each matrix as a single value, this looks exactly like an equation in a single variable. It's Ax = B, only this time A, x, and B are matrices. Again you can solve the problem by dividing both sides by A: x = B/A. This time, you'll use matrix division instead of scalar division, and your result will be a matrix of solutions instead of a single solution.

For numbers, dividing B by A is equivalent to multiplying B by the inverse of A. For instance, 9/3 equals 9 * 1/3. The same is true of matrices. To divide a matrix B by another matrix A, you multiply B by the inverse of A.

The `Matrix` class overloads the division operator to do multiplication by the inverse, so you might wonder why we don't just use that. The problem is that `Matrix#/` calculates B/A as `B*A.inverse`, and what we want is `A.inverse*B`. Matrix multiplication isn't commutative, and so neither is division. The developers of the `Matrix` class had to pick an order to do the multiplication, and they chose the one that won't work for solving a system of equations.

For the most accurate results, you should use `Rational` or `BigDecimal` numbers to represent your coefficients and values. You should never use integers. Calling `Matrix#inverse` on a matrix full of integers will do the inversion using integer division. The result will be totally inaccurate, and you won't get the right solutions to your equations.

Here's a demonstration of the problem. Multiplying a matrix by its inverse should get you an identity matrix, full of zeros but with ones going down the right diagonal. This is analagous to the way multiplying 3 by 1/3 gets you 1.

When the matrix is full of rational numbers, this works fine:

```
matrix = Matrix[[Rational(1), Rational(2)], [Rational(2), Rational(1)]]
matrix.inverse
# => Matrix[[Rational(-1, 3), Rational(2, 3)],
# =>        [Rational(2, 3), Rational(-1, 3)]]

matrix * matrix.inverse
# => Matrix[[Rational(1, 1), Rational(0, 1)],
# =>        [Rational(0, 1), Rational(1, 1)]]
```


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

But if the matrix is full of integers, multiplying it by its inverse will give you a matrix that looks nothing like an identity matrix.

```
matrix = Matrix[[1, 2], [2, 1]]
matrix.inverse
# => Matrix[[-1, 1],
# =>        [0, -1]]

matrix * matrix.inverse
# => Matrix[[-1, -1],
# =>        [-2, 1]]
```

Inverting a matrix that contains floating-point numbers is a lesser mistake: `Matrix#inverse` tends to magnify the inevitable floating-point rounding errors. Multiplying a matrix full of floating-point numbers by its inverse will get you a matrix that's almost, but not quite, an identity matrix.

```
float_matrix = Matrix[[1.0, 2.0], [2.0, 1.0]]
float_matrix.inverse
# => Matrix[[-0.333333333333333, 0.666666666666667],
# =>        [0.666666666666667, -0.333333333333333]]

float_matrix * float_matrix.inverse
# => Matrix[[1.0, 0.0],
# =>        [1.11022302462516e-16, 1.0]]
```

See Also

- Recipe 2.10, "Multiplying Matrices"
- Another way of solving systems of linear equations is with Gauss-Jordan elimination; Shin-ichiro Hara has written an `algebra` library for Ruby, which includes a module for doing Gaussian elimination, along with lots of other linear algebra libraries (http://blade.nagaokaut.ac.jp/~sinara/ruby/math/algebra/)
- There is also a package, called `linalg`, which provides Ruby bindings to the C/Fortran LAPACK library for linear algebra (http://rubyforge.org/projects/linalg/)

Recipe 2.12. Using Complex Numbers

Problem

You want to represent complex ("imaginary") numbers and perform math on them.

Solution

Use the `Complex` class, defined in the `complex` library. All mathematical and trigonometric operations are supported.

```
require 'complex'
```


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

```
Complex::I                      # => Complex(0, 1)

a = Complex(1, 4)               # => Complex(1, 4)
a.real                          # => 1
a.image                         # => 4

b = Complex(1.5, 4.25)          # => Complex(1.5, 4.25)
b + 1.5                         # => Complex(3.0, 4.25)
b + 1.5*Complex::I              # => Complex(1.5, 5.75)

a - b                           # => Complex(-0.5, -0.25)
a * b                           # => Complex(-15.5, 10.25)
b.conjugate                     # => Complex(1.5, -4.25)
Math::sin(b)                    # => Complex(34.9720129257216, 2.47902583958724)
```

Discussion

You can use two floating-point numbers to keep track of the real and complex parts of a complex number, but that makes it complicated to do mathematical operations such as multiplication. If you were to write functions to do these operations, you'd have more or less reimplemented the `Complex` class. `Complex` simply keeps two instances of `Numeric`, and implements the basic math operations on them, keeping them together as a complex number. It also implements the complex-specific mathematical operation `Complex#conjugate`.

Complex numbers have many uses in scientific applications, but probably their coolest application is in drawing certain kinds of fractals. Here's a class that uses complex numbers to calculate and draw a character-based representation of the Mandelbrot set, scaled to whatever size your screen can handle.

```
class Mandelbrot

  # Set up the Mandelbrot generator with the basic parameters for
  # deciding whether or not a point is in the set.

  def initialize(bailout=10, iterations=100)
    @bailout, @iterations = bailout, iterations
  end
```

A point (x,y) on the complex plane is in the Mandelbrot set unless a certain iterative calculation tends to infinity. We can't calculate "tends towards infinity" exactly, but we can iterate the calculation a certain number of times waiting for the result to exceed some "bail-out" value.

If the result ever exceeds the bail-out value, `Mandelbrot` assumes the calculation goes all the way to infinity, which takes it out of the Mandelbrot set. Otherwise, the iteration will run through without exceeding the bail-out value. If that happens, `Mandelbrot` makes the opposite assumption: the calculation for that point will never go to infinity, which puts it in the Mandelbrot set.

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

The default values for `bailout` and `iterations` are precise enough for small, chunky ASCII renderings. If you want to make big posters of the Mandelbrot set, you should increase these numbers.

Next, let's define a method that uses `bailout` and `iterations` to guess whether a specific point on the complex plane belongs to the Mandelbrot set. The variable `x` is a position on the real axis of the complex plane, and `y` is a position on the imaginary axis.

```
# Performs the Mandelbrot operation @iterations times. If the
# result exceeds @bailout, assume this point goes to infinity and
# is not in the set. Otherwise, assume it is in the set.
def mandelbrot(x, y)
  c = Complex(x, y)
  z = 0
  @iterations.times do |i|
    z = z**2 + c                    # This is the Mandelbrot operation.
    return false if z > @bailout
  end
  return true
end
```

The most interesting part of the Mandelbrot set lives between–2 and 1 on the real axis of the complex plane, and between–1 and 1 on the complex axis. The final method in `Mandelbrot` produces an ASCII map of that portion of the complex plane. It maps each point on an ASCII grid to a point on or near the Mandelbrot set. If `Mandelbrot` estimates that point to be in the Mandelbrot set, it puts an asterisk in that part of the grid. Otherwise, it puts a space there. The larger the grid, the more points are sampled and the more precise the map.

```
def render(x_size=80, y_size=24, inside_set="*", outside_set=" ")
  0.upto(y_size) do |y|
    0.upto(x_size) do |x|
      scaled_x = -2 + (3 * x / x_size.to_f)
      scaled_y = 1 + (-2 * y / y_size.to_f)
      print mandelbrot(scaled_x, scaled_y) ? inside_set : outside_set
    end
    puts
    end
  end
end
```

Even at very small scales, the distinctive shape of the Mandelbrot set is visible.

```
Mandelbrot.new.render(25, 10)
#                 **
#                ****
#              ********
#         *** *********
# ********************
#         *** *********
#              ********
#                ****
#                 **
```


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

See Also

- The scaling equation, used to map the complex plane onto the terminal screen, is similar to the equations used to scale data in Recipe 12.5, "Adding Graphical Context with Sparklines," and Recipe 12.14, "Representing Data as MIDI Music"

Recipe 2.13. Simulating a Subclass of Fixnum

Problem

You want to create a class that acts like a subclass of `Fixnum`, `Float`, or one of Ruby's other built-in numeric classes. This wondrous class can be used in arithmetic along with real `Integer` or `Float` objects, and it will usually act like one of those objects, but it will have a different representation or implement extra functionality.

Solution

Let's take a concrete example and consider the possibilities. Suppose you wanted to create a class that acts just like `Integer`, except its string representation is a hexadecimal string beginning with "0x". Where a `Fixnum`'s string representation might be "208", this class would represent 208 as "0xc8".

You could modify `Integer#to_s` to output a hexadecimal string. This would probably drive you insane because it would change the behavior for *all* `Integer` objects. From that point on, nearly all the numbers you use would have hexadecimal string representations. You probably want hexadecimal string representations only for a few of your numbers.

This is a job for a subclass, but you can't usefully subclass `Fixnum` (the Discussion explains why this is so). The only alternative is delegation. You need to create a class that contains an instance of `Fixnum`, and almost always delegates method calls to that instance. The only method calls it doesn't delegate should be the ones that it wants to override.

The simplest way to do this is to create a custom delegator class with the `delegate` library. A class created with `DelegateClass` accepts another object in its constructor, and delegates all methods to the corresponding methods of that object.

```
require 'delegate'
class HexNumber < DelegateClass(Fixnum)
  # The string representations of this class are hexadecimal numbers
  def to_s
    sign = self < 0 ? "-" : ""
    hex = abs.to_s(16)
    "#{sign}0x#{hex}"
  end
```


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

```
  def inspect
    to_s
  end
end

HexNumber.new(10)                 # => 0xa
HexNumber.new(-10)                # => -0xa
HexNumber.new(1000000)            # => 0xf4240
HexNumber.new(1024 ** 10)         # => 0x100000000000000000000000

HexNumber.new(10).succ            # => 11
HexNumber.new(10) * 2             # => 20
```

Discussion

Some object-oriented languages won't let you subclass the "basic" data types like integers. Other languages implement those data types as classes, so you can subclass them, no questions asked. Ruby implements numbers as classes (`Integer`, with its concrete subclasses `Fixnum` and `Bignum`), and you can subclass those classes. If you try, though, you'll quickly discover that your subclasses are useless: they don't have constructors.

Ruby jealously guards the creation of new `Integer` objects. This way it ensures that, for instance, there can be only one `Fixnum` instance for a given number:

```
100.object_id                             # => 201
(10 * 10).object_id                       # => 201
Fixnum.new(100)
# NoMethodError: undefined method `new' for Fixnum:Class
```

You can have more than one `Bignum` object for a given number, but you can only create them by exceeding the bounds of `Fixnum`. There's no `Bignum` constructor, either. The same is true for `Float`.

```
(10 ** 20).object_id                      # => -606073730
((10 ** 19) * 10).object_id               # => -606079360
Bignum.new(10 ** 20)
# NoMethodError: undefined method `new' for Bignum:Class
```

If you subclass `Integer` or one of its subclasses, you won't be able to create any instances of your class—not because those classes aren't "real" classes, but because they don't really have constructors. You might as well not bother.

So how can you create a custom number-like class without redefining all the methods of `Fixnum`? You can't, really. The good news is that in Ruby, there's nothing painful about redefining all the methods of `Fixnum`. The `delegate` library takes care of it for you. You can use this library to generate a class that responds to all the same method calls as `Fixnum`. It does this by delegating all those method calls to a `Fixnum` object it holds as a member. You can then override those classes at your leisure, customizing behavior.

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

Since most methods are delegated to the member `Fixnum`, you can perform math on `HexNumber` objects, use `succ` and `upto`, create ranges, and do almost anything else you can do with a `Fixnum`. Calling `HexNumber#is_a?(Fixnum)` will return false, but you can change even that by manually overriding `is_a?`.

Alas, the illusion is spoiled somewhat by the fact that when you perform math on `HexNumber` objects, you get `Fixnum` objects back.

```
HexNumber.new(10) * 2                       # => 20
HexNumber.new(10) + HexNumber.new(200)      # => 210
```

Is there a way to do math with `HexNumber` objects and get `HexNumber` objects as results? There is, but it requires moving a little bit beyond the comfort of the `delegate` library. Instead of simply delegating all our method calls to an `Integer` object, we want to delegate the method calls, then intercept and modify the return values. If a method call on the underlying `Integer` object returns an `Integer` or a collection of `Integer`s, we want to convert it into a `HexNumber` object or a collection of `HexNumber`s.

The easiest way to delegate all methods is to create a class that's nearly empty and define a `method_missing` method. Here's a second `HexNumber` class that silently converts the results of mathematical operations (and any other `Integer` result from a method of `Integer`) into `HexNumber` objects. It uses the `BasicObject` class from the Facets More library (available as the `facets-more` gem): a class that defines almost no methods at all. This lets us delegate almost everything to `Integer`.

```
require 'rubygems'
require 'facet/basicobject'

class BetterHexNumber < BasicObject

  def initialize(integer)
    @value = integer
  end

  # Delegate all methods to the stored integer value. If the result is a
  # Integer, transform it into a BetterHexNumber object. If it's an
  # enumerable containing Integers, transform it into an enumerable
  # containing BetterHexNumber objects

  def method_missing(m, *args)
    super unless @value.respond_to?(m)
    hex_args = args.collect do |arg|
      arg.kind_of?(BetterHexNumber) ? arg.to_int : arg
    end
    result = @value.send(m, *hex_args)
    return result if m == :coerce
    case result
    when Integer
      BetterHexNumber.new(result)
    when Array
      result.collect do |element|
        element.kind_of?(Integer) ? BetterHexNumber.new(element) : element
      end
    else
```


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

```
      result
    end
  end

  # We don't actually define any of the Fixnum methods in this class,
  # but from the perspective of an outside object we do respond to
  # them. What outside objects don't know won't hurt them, so we'll
  # claim that we actually implement the same methods as our delegate
  # object. Unless this method is defined, features like ranges won't
  # work.
  def respond_to?(method_name)
    super or @value.respond_to? method_name
  end

  # Convert the number to a hex string, ignoring any other base
  # that might have been passed in.
  def to_s(*args)
    hex = @value.abs.to_s(16)
    sign = self < 0 ? "-" : ""
    "#{sign}0x#{hex}"
  end

  def inspect
    to_s
  end
end
```

Now we can do arithmetic with `BetterHexNumber` objects, and get `BetterHexNumber` object back:

```
hundred = BetterHexNumber.new(100)        # => 0x64
hundred + 5                               # => 0x69
hundred + BetterHexNumber.new(5)          # => 0x69
hundred.succ                              # => 0x65
hundred / 5                               # => 0x14
hundred * -10                             # => -0x3e8
hundred.divmod(3)                         # => [0x21, 0x1]
(hundred…hundred+3).collect             # => [0x64, 0x65, 0x66]
```

A `BetterHexNumber` even claims to be a `Fixnum`, and to respond to all the methods of `Fixnum`! The only way to know it's not is to call `is_a?`.

```
hundred.class                         # => Fixnum
hundred.respond_to? :succ             # => true
hundred.is_a? Fixnum                  # => false
```

See Also

- Recipe 2.6, "Converting Between Numeric Bases"
- Recipe 2.14, "Doing Math with Roman Numbers"
- Recipe 8.8, "Delegating Method Calls to Another Object"
- Recipe 10.8, "Responding to Calls to Undefined Methods"

Recipe 2.14. Doing Math with Roman Numbers

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

Problem

You want to convert between Arabic and Roman numbers, or do arithmetic with Roman numbers and get Roman numbers as your result.

Solution

The simplest way to define a `Roman` class that acts like `Fixnum` is to have its instances delegate most of their method calls to a real `Fixnum` (as seen in the previous recipe, Recipe 2.13). First we'll implement a container for the `Fixnum` delegate, and methods to convert between Roman and Arabic numbers:

```
class Roman
  # These arrays map all distinct substrings of Roman numbers
  # to their Arabic equivalents, and vice versa.
  @@roman_to_arabic = [['M', 1000], ['CM', 900], ['D', 500], ['CD', 400],
    ['C', 100], ['XC', 90], ['L', 50], ['XL', 40], ['X', 10], ['IX', 9],
    ['V', 5], ['IV', 4], ['I', 1]]
  @@arabic_to_roman = @@roman_to_arabic.collect { |x| x.reverse }.reverse

  # The Roman symbol for 5000 (a V with a bar over it) is not in
  # ASCII nor Unicode, so we won't represent numbers larger than 3999.
  MAX = 3999

  def initialize(number)
    if number.respond_to? :to_str
      @value = Roman.to_arabic(number)
    else
      Roman.assert_within_range(number)
      @value = number
    end
  end

  # Raise an exception if a number is too large or small to be represented
  # as a Roman number.
  def Roman.assert_within_range(number)
    unless number.between?(1, MAX)
      msg = "#{number} can't be represented as a Roman number."
      raise RangeError.new(msg)
    end
  end

  #Find the Fixnum value of a string containing a Roman number.
  def Roman.to_arabic(s)
    value = s
    if s.respond_to? :to_str
      c = s.dup
      value = 0
      invalid = ArgumentError.new("Invalid Roman number: #{s}")
      value_of_previous_number = MAX+1
      value_from_previous_number = 0
      @@roman_to_arabic.each_with_index do |(roman, arabic), i|
        value_from_this_number = 0
        while c.index(roman) == 0
          value_from_this_number += arabic
          if value_from_this_number >= value_of_previous_number
            raise invalid
          end
          c = c[roman.size..s.size]
        end

        #This one's a little tricky. We reject numbers like "IVI" and
        #"IXV", because they use the subtractive notation and then
        #tack on a number that makes the total overshoot the number
        #they'd have gotten without using the subtractive
```


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

```
        #notation. Those numbers should be V and XIV, respectively.
        if i > 2 and @@roman_to_arabic[i-1][0].size > 1 and
            value_from_this_number + value_from_previous_number >=
            @@roman_to_arabic[i-2][1]
          raise invalid
        end

        value += value_from_this_number
        value_from_previous_number = value_from_this_number
        value_of_previous_number = arabic
        break if c.size == 0
      end
      raise invalid if c.size > 0
    end
    return value
  end

  def to_arabic
    @value
  end
  #Render a Fixnum as a string depiction of a Roman number
  def to_roman
    value = to_arabic
    Roman.assert_within_range(value)
    repr = ""
    @@arabic_to_roman.reverse_each do |arabic, roman|
      num, value = value.divmod(arabic)
      repr << roman * num
    end
    repr
  end
```

Next, we'll make the class respond to all of `Fixnum`'s methods by implementing a `method_missing` that delegates to our internal `Fixnum` object. This is substantially the same `method_missing` as in Recipe 2.13 Whenever possible, we'll transform the results of a delegated method into `Roman` objects, so that operations on `Roman` objects will yield other `Roman` objects.

```
# Delegate all methods to the stored integer value. If the result is
# a Integer, transform it into a Roman object. If it's an array
# containing Integers, transform it into an array containing Roman
# objects.
def method_missing(m, *args)
  super unless @value.respond_to?(m)
  hex_args = args.collect do |arg|
    arg.kind_of?(Roman) ? arg.to_int : arg
  end
  result = @value.send(m, *hex_args)
  return result if m == :coerce
  begin
    case result
    when Integer
      Roman.new(result)
    when Array
      result.collect do |element|
        element.kind_of?(Integer) ? Roman.new(element) : element
      end
    else
      result
    end
  rescue RangeError
    # Too big or small to fit in a Roman number. Use the original number
    result
  end
end
```


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

The only methods that won't trigger `method_missing` are methods like `to_s`, which we're going to override with our own implementations:

```
  def respond_to?(method_name)
    super or @value.respond_to? method_name
  end

  def to_s
    to_roman
  end

  def inspect
    to_s
  end
end
```

We'll also add methods to `Fixnum` and `String` that make it easy to create `Roman` objects:

```
class Fixnum
  def to_roman
    Roman.new(self)
  end
end

class String
  def to_roman
    Roman.new(self)
  end
end
```

Now we're ready to put the `Roman` class through its paces:

```
72.to_roman                               # => LXXII
444.to_roman                              # => CDXLIV
1979.to_roman                             # => MCMLXXIX
'MCMXLVIII'.to_roman                      # => MCMXLVIII

Roman.to_arabic('MCMLXXIX')               # => 1979
'MMI'.to_roman.to_arabic                  # => 2001

'MMI'.to_roman + 3                        # => MMIV
'MCMXLVIII'.to_roman                      # => MCMXLVIII
612.to_roman * 3.to_roman                 # => MDCCCXXXVI
(612.to_roman * 3).divmod('VII'.to_roman) # => [CCLXII, II]
612.to_roman * 10000                      # => 6120000    # Too big
612.to_roman * 0                          # => 0          # Too small

'MCMXCIX'.to_roman.succ                   # => MM

('I'.to_roman..'X'.to_roman).collect
# => [I, II, III, IV, V, VI, VII, VIII, IX, X]
```

Here are some invalid Roman numbers that the `Roman` class rejects:

```
'IIII'.to_roman
# ArgumentError: Invalid Roman number: IIII
'IVI'.to_roman
# ArgumentError: Invalid Roman number: IVI
'IXV'.to_roman
# ArgumentError: Invalid Roman number: IXV
'MCMM'.to_roman
```


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

```
# ArgumentError: Invalid Roman number: MCMM
'CIVVM'.to_roman
# ArgumentError: Invalid Roman number: CIVVM
-10.to_roman
# RangeError: -10 can't be represented as a Roman number.
50000.to_roman
# RangeError: 50000 can't be represented as a Roman number.
```

Discussion

The rules for constructing Roman numbers are more complex than those for constructing positional numbers such as the Arabic numbers we use. An algorithm for parsing an Arabic number can scan from the left, looking at each character in isolation. If you were to scan a Roman number from the left one character at a time, you'd often find yourself having to backtrack, because what you thought was "XI" (11) would frequently turn out to be "XIV" (14).

The simplest way to parse a Roman number is to adapt the algorithm so that (for instance) "IV" as treated as its own "character," distinct from "I" and "V". If you have a list of all these "characters" and their Arabic values, you can scan a Roman number from left to right with a greedy algorithm that keeps a running total. Since there are few of these "characters" (only 13 of them, for numbers up to 3,999), and none of them are longer than 2 letters, this algorithm is workable. To generate a Roman number from an Arabic number, you can reverse the process.

The `Roman` class given in the Solution works like `Fixnum`, thanks to the `method_missing` strategy first explained in Recipe 2.13. This lets you do math entirely in Roman numbers, except when a result is out of the supported range of the `Roman` class.

Since this `Roman` implementation only supports 3999 distinct numbers, you could make the implementation more efficient by pregenerating all of them and retrieving them from a cache as needed. The given implementation lets you extend the implementation to handle larger numbers: you just need to decide on a representation for the larger Roman characters that will work for your encoding.

The Roman numeral for 5,000 (a V with a bar over it) isn't present in ASCII, but there are Unicode characters U+2181 (the Roman numeral 5,000) and U+2182 (the Roman numeral 10,000), so that's the obvious choice for representing Roman numbers up to 39,999. If you're outputting to HTML, you can use a CSS style to put a bar above "V", "X", and so on. If you're stuck with ASCII, you might choose "_V" to represent 5,000, "_X" to represent 10,000, and so on. Whatever you chose, you'd add the appropriate "characters" to the `roman_to_arabic` array (remembering to add "M_V" and "_V_X" as well as "_V" and "_X"), increment `MAX`, and suddenly be able to instantiate `Roman` objects for large numbers.

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

The `Roman#to_arabic` method implements the "new" rules for Roman numbers: that is, the ones standardized in the Middle Ages. It rejects certain number representations, like IIII, used by the Romans themselves.

Roman numbers are common as toy or contest problems, but it's rare that a programmer will have to treat a Roman number as a number, as opposed to a funny-looking string. In parts of Europe, centuries and the month section of dates are written using Roman numbers. Apart from that, outline generation is probably the only real-world application where a programmer needs to treat a Roman number as a number. Outlines need several of visually distinct ways to represent the counting numbers, and Roman numbers (upper- and lowercase) provide two of them.

If you're generating an outline in plain text, you can use `Roman#succ` to generate a succession of Roman numbers. If your outline is in HTML format, though, you don't need to know anything about Roman numbers at all. Just give an `<ol>` tag a CSS style of `list-style-type:lower-roman` or `list-style-type:upper-roman`. Output the elements of your outline as `<li>` tags inside the `<ol>` tag. All modern browsers will do the right thing:

```
<ol style="list-style-type:lower-roman">
<li>Primus</li>
<li>Secundis</li>
<li>Tertius</li>
</ol>
```

See Also

- Recipe 2.13, "Simulating a Subclass of Fixnum"
- An episode of the Ruby Quiz focused on algorithms for converting between Roman and Arabic numbers; one solution uses an elegant technique to make it easier to create Roman numbers from within Ruby: it overrides `Object#const_ missing` to convert any undefined constant into a Roman number; this lets you issue a statement like `XI + IX`, and get `XX` as the result (http://www.rubyquiz.com/quiz22.html)

Recipe 2.15. Generating a Sequence of Numbers

Problem

You want to iterate over a (possibly infinite) sequence of numbers the way you can iterate over an array or a range.

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

Solution

Write a generator function that `yield`s each number in the sequence.

```
def fibonacci(limit = nil)
  seed1 = 0
  seed2 = 1
  while not limit or seed2 <= limit
    yield seed2
    seed1, seed2 = seed2, seed1 + seed2
  end
end

fibonacci(3) { |x| puts x }
# 1
# 1
# 2
# 3

fibonacci(1) { |x| puts x }
# 1
# 1

fibonacci { |x| break if x > 20; puts x }
# 1
# 1
# 2
# 3
# 5
# 8
# 13
```

Discussion

A generator for a sequence of numbers works just like one that iterates over an array or other data structure. The main difference is that iterations over a data structure usually have a natural stopping point, whereas most common number sequences are infinite.

One strategy is to implement a method called `each` that yields the entire sequence. This works especially well if the sequence is finite. If not, it's the responsibility of the code block that consumes the sequence to stop the iteration with the `break` keyword.

`Range#each` is an example of an iterator over a finite sequence, while `Prime#each` enumerates the infinite set of prime numbers. `Range#each` is implemented in C, but here's a (much slower) pure Ruby implementation for study. This code uses `self.begin` and `self.end` to call `Range#begin` and `Range#end`, because `begin` and `end` are reserved words in Ruby.

```
class Range
  def each_slow
    x = self.begin
    while x <= self.end
      yield x
      x = x.succ
    end
  end
end
```


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

```
(1..3).each_slow {|x| puts x}
# 1
# 2
# 3
```

The other kind of sequence generator iterates over a finite portion of an infinite sequence. These are methods like `Fixnum#upto` and `Fixnum#step`: they take a start and/ or an end point as input, and generate a finite sequence within those boundaries.

```
class Fixnum
  def double_upto(stop)
    x = self
    until x > stop
      yield x
    x = x * 2
   end
  end
end
10.double_upto(50) { |x| puts x }
# 10
# 20
# 40
```

Most sequences move monotonically up or down, but it doesn't have to be that way:

```
def oscillator
  x = 1
  while true
    yield x
      x *= -2
  end
end
oscillator { |x| puts x; break if x.abs > 50; }
# 1
# -2
# 4
# -8
# 16
# -32
# 64
```

Though integer sequences are the most common, any type of number can be used in a sequence. For instance, `Float#step` works just like `Integer#step`:

```
1.5.step(2.0, 0.25) { |x| puts x }
# => 1.5
# => 1.75
# => 2.0
```

`Float` objects don't have the resolution to represent every real number. Very small differences between numbers are lost. This means that some `Float` sequences you might think would go on forever will eventually end:

```
def zeno(start, stop)
  distance = stop - start
  travelled = start
  while travelled < stop and distance > 0
```


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

```
    yield travelled
    distance = distance / 2.0
    travelled += distance
  end
end

steps = 0
zeno(0, 1) { steps += 1 }
steps                                    # => 54
```

See Also

- Recipe 1.16, "Generating a Succession of Strings"
- Recipe 2.16, "Generating Prime Numbers," shows optimizations for generating a very well-studied number sequence
- Recipe 4.1, "Iterating Over an Array"
- Chapter 7 has more on this kind of generator method

Recipe 2.16. Generating Prime Numbers

Problem

You want to generate a sequence of prime numbers, or find all prime numbers below a certain threshold.

Solution

Instantiate the `Prime` class to create a prime number generator. Call `Prime#succ` to get the next prime number in the sequence.

```
require 'mathn'
primes = Prime.new
primes.succ                              # => 2
primes.succ                              # => 3
```

Use `Prime#each` to iterate over the prime numbers:

```
primes.each { |x| puts x; break if x > 15; }
# 5
# 7
# 11
# 13
# 17
primes.succ                             # => 19
```

Discussion

Because prime numbers are both mathematically interesting and useful in cryptographic applications, a lot of study has been lavished on them. Many algorithms have been devised

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

for generating prime numbers and determining whether a number is prime. The code in this recipe walks a line between efficiency and ease of implementation.

The best-known prime number algorithm is the Sieve of Eratosthenes, which finds all primes in a certain range by iterating over that range multiple times. On the first pass, it eliminates every even number greater than 2, on the second pass every third number after 3, on the third pass every fifth number after 5, and so on. This implementation of the Sieve is based on a sample program packaged with the Ruby distribution:

```
def sieve(max=100)
  sieve = []
  (2..max).each { |i| sieve[i] = i }
  (2..Math.sqrt(max)).each do |i|
    (i*i).step(max, i) { |j| sieve[j] = nil } if sieve[i]
  end
  sieve.compact
end

sieve(10)
# => [2, 3, 5, 7]
sieve(100000).size
# => 9592
```

The Sieve is a fast way to find the primes smaller than a certain number, but it's memory-inefficient and it's not suitable for generating an infinite sequence of prime numbers. It's also not very compatible with the Ruby idiom of generator methods. This is where the `Prime` class comes in.

A `Prime` object stores the current state of one iteration over the set of primes. It contains all information necessary to calculate the next prime number in the sequence. `Prime#each` repeatedly calls `Prime#succ` and `yield`s it up to whatever code block was passed in.

Ruby 1.9 has an efficient implementation of `Prime#each`, but Ruby 1.8 has a very slow implementation. The following code is based on the 1.9 implementation, and it illustrates many of the simple tricks that drastically speed up algorithms that find or use primes. You can use this code, or just paste the code from Ruby 1.9's `mathn.rb` into your 1.8 program.

The first trick is to share a single list of primes between all `Prime` objects by making it a class variable. This makes it much faster to iterate over multiple `Prime` instances, but it also uses more memory because the list of primes will never be garbage-collected.

We initialize the list with the first few prime numbers. This helps early performance a little bit, but it's mainly to get rid of edge cases. The class variable `@@check_next` tracks the next number we think might be prime.

```
require 'mathn'
```


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

```
class Prime
  @@primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59,
              61, 67, 71, 73, 79, 83, 89, 97, 101]
  @@check_next = 103
end
```

A number is prime if it has no factors: more precisely, if it has no *prime* factors between 2 and its square root. This code uses the list of prime numbers not only as a cache, but as a data structure to help find larger prime numbers. Instead of checking all the possible factors of a number, we only need to check some of the *prime* factors.

To avoid calculating square roots, we have `@@limit` track the largest prime number less than the square root of `@@check_next`. We can decide when to increment it by calculating squares instead of square roots:

```
class Prime
# @@primes[3] < sqrt(@@check_next) < @@primes[4]
  @@limit = 3

  # sqrt(121) == @@primes[4]
  @@increment_limit_at = 121
end
```

Now we need a new implementation of `Prime#succ`. Starting from `@@check_next`, the new implementation iterates over numbers until it finds one that's prime, then returns the prime number. But it doesn't iterate over the numbers one at a time: we can do better than that. It skips even numbers and numbers divisible by three, which are obviously not prime.

```
class Prime
   def succ
    @index += 1
    while @index >= @@primes.length
      if @@check_next + 4 > @@increment_limit_at
        @@limit += 1
        @@increment_limit_at = @@primes[@@limit + 1] ** 2
      end
      add_if_prime
      @@check_next += 4
      add_if_prime
      @@check_next += 2
    end
    return @@primes[@index]
  end
end
```

How does it do this? Well, consider a more formal definition of "even" and "divisible by three." If *x* is congruent to 2 or 4, mod 6 (that is, if *x* % 6 is 2 or 4), then x is even and not prime. If *x* is congruent to 3, mod 6, then *x* is divisible by 3 and not prime. If *x* is congruent to 1 or 5, mod 6, then *x* might be prime.

Our starting point is `@@check_next`, which starts out at 103. 103 is congruent to 1, mod 6, so it might be prime. Adding 4 gives us 107, a number congruent to 5, mod 6. We skipped two even numbers (104 and 106) and a number divisible by 3 (105). Adding 2 to 107 skips

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

another even number and gives us 109. Like 103, 109 is congruent to 1, mod 6. We can add 4 and 2 again to get two more numbers that might be prime. By continually adding 4 and then 2 to `@@check_next`, we can skip over the numbers that are obviously not prime.

Although all `Prime` objects share a list of primes, each object should start yielding primes from the beginning of the list:

```
class Prime
  def initialize
    @index = -1
  end
end
```

Finally, here's the method that actually checks `@@check_next` for primality, by looking for a prime factor of that number between 5 and `@@limit`. We don't have to check 2 and 3 because `succ` skips numbers divisible by 2 and 3. If no prime factor is found, the number is prime: we add it to the class-wide list of primes, where it can be returned by `succ` or yielded to a code block by `each`.

```
class Prime
  private
  def add_if_prime
    factor = @@primes[2..@@limit].find { |prime| @@check_next % prime == 0 }
    @@primes << @@check_next unless factor
  end
 end
end
```

Here's the new `Prime` class in action, finding the ten-thousandth prime:

```
primes = Prime.new
p = nil
10000.times { p = primes.succ }
p                                    # => 104729
```

Checking primality

The simplest way to check whether a particular number is prime is to generate all the primes up to that number and see whether the number itself is generated as a prime.

```
class Prime
  def prime?(n)
    succ( ) while @seed < n
    return @primes.member?(n)
  end
end
```

If all of this is too complicated for you, there's a very simple constant-time probabilistic test for primality that works more than half the time:

```
def probably_prime?(x)
```


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

```
  x < 8
end

probably_prime? 2                   # => true
probably_prime? 5                   # => true

probably_prime? 6                   # => true
probably_prime? 7                   # => true
probably_prime? 8                   # => false
probably_prime? 100000              # => false
```

See Also

- Recipe 2.15, "Generating a Sequence of Numbers"
- K. Kodama has written a number of simple and advanced primality tests in Ruby (http://www.math.kobe-u.ac.jp/~kodama/tips-prime.html)

Recipe 2.17. Checking a Credit Card Checksum

Problem

You want to know whether a credit card number was entered correctly.

Solution

The last digit of every credit card is a checksum digit. You can compare the other digits against the checksum to catch mistakes someone might make when typing their credit card number.

Lucas Carlson's CreditCard library, available as the `creditcard` gem, contains Ruby implementations of the checksum algorithms. It adds methods to the `String` and `Integer` classes to check the internal consistency of a credit card number:

```
require 'rubygems'
require 'creditcard'

'5276 4400 6542 1319'.creditcard?    # => true
'5276440065421313'.creditcard?       # => false
1276440065421319.creditcard?         # => false
```

CreditCard can also determine which brand of credit card a certain number is for:

```
5276440065421313.creditcard_type     # => "mastercard"
```


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

Discussion

The CreditCard library uses a well-known algorithm for finding the checksum digit of a credit card. If you can't or don't want to install the `creditcard` gem, you can just implement the algorithm yourself:

```
module CreditCard
  def creditcard?
    numbers = self.to_s.gsub(/[^\d]+/, '').split(//)

    checksum = 0
    0.upto numbers.length do |i|
      weight = numbers[-1*(i+2)].to_i * (2 - (i%2))

      checksum += weight % 9
    end

    return numbers[-1].to_i == 10 - checksum % 10
  end
end

class String
  include CreditCard
end

class Integer
  include CreditCard
end

'5276 4400 6542 1319'.creditcard?      # => true
```

How does it work? First, it converts the object to an array of numbers:

```
numbers = '5276 4400 6542 1319'.gsub(/[^\d]+/, '').split(//)
# =>  ["5", "2", "7", "6", "4", "4", "0", "0",
# =>  "6", "5", "4", "2", "1", "3", "1", "9"]
```

It then calculates a weight for each number based on its position, and adds that weight to a running checksum:

```
checksum = 0
0.upto numbers.length do |i|
  weight = numbers[-1*(i+2)].to_i * (2 - (i%2))
  checksum += weight % 9
end
checksum                               # => 51
```

If the last number of the card is equal to 10 minus the last digit of the checksum, the number is self-consistent:

```
numbers[-1].to_i == 10 - checksum % 10                          # => true
```

A self-consistent credit card number is just a number with a certain mathematical property. It can catch typos, but there's no guarantee that a real credit card exists with that number.

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

To check that, you need to use a payment gateway like Authorize.net, and a gateway library like `Payment::AuthorizeNet`.

See Also

- Recipe 16.8, "Charging a Credit Card"

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