The Ruby Programming Language

CHAPTER 1

Introduction

1.1.1 Ruby Is Object-Oriented

```
1.class       # => Fixnum: the number 1 is a Fixnum
0.0.class     # => Float: floating-point numbers have class Float
true.class    # => TrueClass: true is a the singleton instance of TrueClass
false.class   # => FalseClass
nil.class     # => NilClass
```

1.1.2 Blocks and Iterators

```
3.times { print "Ruby! " }    # Prints "Ruby! Ruby! Ruby! "
1.upto(9) {|x| print x }      # Prints "123456789"

a = [3, 2, 1]                 # This is an array literal
a[3] = a[2] - 1               # Use square brackets to query and set array
elements
a.each do |elt|               # each is an iterator. The block has a parameter
elt
  print elt+1                 # Prints "4321"
end                           # This block was delimited with do/end instead
of {}

a = [1,2,3,4]                 # Start with an array
b = a.map {|x| x*x }          # Square elements: b is [1,4,9,16]
c = a.select {|x| x%2==0 }    # Select even elements: c is [2,4]
a.inject do |sum,x|           # Compute the sum of the elements => 10
  sum + x
end

h = {                         # A hash that maps number names to digits
  :one => 1,                  # The "arrows" show mappings: key=>value
  :two => 2                   # The colons indicate Symbol literals
}
h[:one]                       # => 1.  Access a value by key
h[:three] = 3                 # Add a new key/value pair to the hash
h.each do |key,value|         # Iterate through the key/value pairs
  print "#{value}:#{key}; "   # Note variables substituted into string
end                           # Prints "1:one; 2:two; 3:three; "

File.open("data.txt") do |f|  # Open named file and pass stream to block
```

```
  line = f.readline           # Use the stream to read from the file
end                           # Stream automatically closed at block end

t = Thread.new do             # Run this block in a new thread
  File.read("data.txt")       # Read a file in the background
end                           # File contents available as thread value

print "#{value}:#{key}; "     # Note variables substituted into string
```

1.1.3 Expressions and Operators in Ruby

```
minimum = if x < y then x else y end

1 + 2                      # => 3: addition
1 * 2                      # => 2: multiplication
1 + 2 == 3                 # => true: == tests equality
2 ** 1024                  # 2 to the power 1024: Ruby has arbitrary size ints
"Ruby" + " rocks!"         # => "Ruby rocks!": string concatenation
"Ruby! " * 3               # => "Ruby! Ruby! Ruby! ": string repetition
"%d %s" % [3, "rubies"]    # => "3 Rubies": Python-style, printf formatting
max = x > y ? x : y        # The conditional operator
```

1.1.4 Methods

```
def square(x)         # Define a method named square with one parameter x
  x*x                 # Return x squared
end                   # End of the method

def Math.square(x)    # Define a class method of the Math module
  x*x
end
```

1.1.5 Assignment

```
x = 1

x += 1             # Increment x: note Ruby does not have ++.
y -= 1             # Decrement y: no -- operator, either.

x, y = 1, 2        # Same as x = 1; y = 2
a, b = b, a        # Swap the value of two variables
x,y,z = [1,2,3]    # Array elements automatically assigned to variables

# Define a method to convert Cartesian (x,y) coordinates to Polar
```

```
def polar(x,y)
  theta = Math.atan2(y,x)   # Compute the angle
  r = Math.hypot(x,y)       # Compute the distance
  [r, theta]                # The last expression is the return value
end

# Here's how we use this method with parallel assignment
distance, angle = polar(2,2)

o.x=(1)         # Normal method invocation syntax
o.x = 1         # Method invocation through assignment
```

1.1.7 Regexp and Range

```
/[Rr]uby/         # Matches "Ruby" or "ruby"
/\d{5}/           # Matches 5 consecutive digits
1..3              # All x where 1 <= x <= 3
1...3             # All x where 1 <= x < 3

# Determine US generation name based on birth year
# Case expression tests ranges with ===
generation = case birthyear
             when 1946..1963: "Baby Boomer"
             when 1964..1976: "Generation X"
             when 1978..2000: "Generation Y"
             else nil
             end

# A method to ask the user to confirm something
def are_you_sure?                           # Define a method. Note question mark!
  while true                                # Loop until we explicitly return
    print "Are you sure? [y/n]: "           # Ask the user a question
    response = gets                         # Get her answer
    case response                           # Begin case conditional
    when /^[yY]/                            # If response begins with y or Y
      return true                           # Return true from the method
    when /^[nN]/, /^$/                      # If response begins with n,N or is empty
      return false                          # Return false
    end
  end
end
```

1.1.8 Classes and Modules

```
#
# This class represents a sequence of numbers characterized by the three
# parameters from, to, and by. The numbers x in the sequence obey the
```

```ruby
# following two constraints:
#
#    from <= x <= to
#    x = from + n*by, where n is an integer
#
class Sequence
  # This is an enumerable class; it defines an each iterator below.
  include Enumerable   # Include the methods of this module in this class

  # The initialize method is special; it is automatically invoked to
  # initialize newly created instances of the class
  def initialize(from, to, by)
    # Just save our parameters into instance variables for later use
    @from, @to, @by = from, to, by  # Note parallel assignment and @ prefix
  end

  # This is the iterator required by the Enumerable module
  def each
    x = @from        # Start at the starting point
    while x <= @to   # While we haven't reached the end
      yield x        # Pass x to the block associated with the iterator
      x += @by       # Increment x
    end
  end

  # Define the length method (following arrays) to return the number of
  # values in the sequence
  def length
    return 0 if @from > @to        # Note if used as a statement modifier
    Integer((@to-@from)/@by) + 1   # Compute and return length of sequence
  end

  # Define another name for the same method.
  # It is common for methods to have multiple names in Ruby
  alias size length   # size is now a synonym for length

  # Override the array-access operator to give random access to the sequence
  def[](index)
    return nil if index < 0  # Return nil for negative indexes
    v = @from + index*@by    # Compute the value
    if v <= @to              # If it is part of the sequence
      v                      # Return it
    else                     # Otherwise...
      nil                    # Return nil
    end
  end

  # Override arithmetic operators to return new Sequence objects
  def *(factor)
    Sequence.new(@from*factor, @to*factor, @by*factor)
  end
```

```
  def +(offset)
    Sequence.new(@from+offset, @to+offset, @by)
  end
end

s = Sequence.new(1, 10, 2)          # From 1 to 10 by 2's
s.each {|x| print x }               # Prints "13579"
print s[s.size-1]                   # Prints 9
t = (s+1)*2                         # From 4 to 22 by 4's

module Sequences                    # Begin a new module
  def self.fromtoby(from, to, by)   # A singleton method of the module
    x = from
    while x <= to
      yield x
      x += by
    end
  end
end

Sequences.fromtoby(1, 10, 2) {|x| print x }  # Prints "13579"

class Range                  # Open an existing class for additions
  def by(step)               # Define an iterator named by
    x = self.begin           # Start at one endpoint of the range
    if exclude_end?          # For ... ranges that exclude the end
      while x < self.end     # Test with the < operator
        yield x
        x += step
      end
    else                     # Otherwise, for .. ranges that include the end
      while x <= self.end    # Test with <= operator
        yield x
        x += step
      end
    end
  end                        # End of method definition
end                          # End of class modification

# Examples
(0..10).by(2) {|x| print x}  # Prints "0246810"
(0...10).by(2) {|x| print x} # Prints "02468"
```

1.2.1 The Ruby Interpreter

```
% ruby -e 'puts "hello world!"'

hello world!
```

```
% ruby hello.rb

hello world!
```

1.2.2 Displaying Output

```
9.downto(1) {|n| print n }   # No newline between numbers
puts " blastoff!"            # End with a newline

% ruby count.rb

987654321 blastoff!
```

1.2.3 Interactive Ruby with irb

```
$ irb --simple-prompt    # Start irb from the terminal
2**3                     # Try exponentiation => 8

"Ruby! " * 3             # Try string repetition => "Ruby! Ruby! Ruby! "

1.upto(3){|x| puts x }   # Try an iterator
# 1                      # Three lines of output
# 2                      # Because we called puts 3 times
# 3                      # The return value of 1.upto(3) => 1
quit                     # Exit irb
$                        # Back to the terminal prompt
```

1.2.4 Viewing Ruby Documentation with ri

```
ri Array

ri Array.sort

ri Hash#each

ri Math::sqrt
```

1.2.5 Ruby Package Management with gem

```
gem install rails
```

```
Successfully installed activesupport-1.4.4

Successfully installed activerecord-1.15.5

Successfully installed actionpack-1.13.5

Successfully installed actionmailer-1.3.5

Successfully installed actionwebservice-1.2.5

Successfully installed rails-1.2.5

6 gems installed

Installing ri documentation for activesupport-1.4.4...

Installing ri documentation for activerecord-1.15.5...

...etc...

gem list                # List installed gems

gem enviroment          # Display RubyGems configuration information

gem update rails        # Update a named gem

gem update              # Update all installed gems

gem update --system     # Update RubyGems itself

gem uninstall rails     # Remove an installed gem

require 'rubygems'                # Not necessary in Ruby >= 1.9

gem 'RedCloth', '> 2.0', '< 4.0' # Activate RedCloth version 2.x or 3.x in
Gemfile

require 'RedCloth'                # And now load it in code file
```

1.4 A Sudoku Solver in Ruby

```
#
# This module defines a Sudoku::Puzzle class to represent a 9x9
# Sudoku puzzle and also defines exception classes raised for
# invalid input and over-constrained puzzles. This module also defines
# the method Sudoku.solve to solve a puzzle. The solve method uses
# the Sudoku.scan method, which is also defined here.
#
# Use this module to solve Sudoku puzzles with code like this:
#
```

```ruby
#  require 'sudoku'
#  puts Sudoku.solve(Sudoku::Puzzle.new(ARGF.readlines))
#
module Sudoku

  #
  # The Sudoku::Puzzle class represents the state of a 9x9 Sudoku puzzle.
  #
  # Some definitions and terminology used in this implementation:
  #
  # - Each element of a puzzle is called a "cell".
  # - Rows and columns are numbered from 0 to 8, and the coordinates [0,0]
  #   refer to the cell in the upper-left corner of the puzzle.
  # - The nine 3x3 subgrids are known as "boxes" and are also numbered from
  #   0 to 8, ordered from left to right and top to bottom. The box in
  #   the upper-left is box 0. The box in the upper-right is box 2. The
  #   box in the middle is box 4. The box in the lower-right is box 8.
  #
  # Create a new puzzle with Sudoku::Puzzle.new, specifying the initial
  # state as a string or as an array of strings. The string(s) should use
  # the characters 1 through 9 for the given values, and '.' for cells
  # whose value is unspecified. Whitespace in the input is ignored.
  #
  # Read and write access to individual cells of the puzzle is through the
  # [] and []= operators, which expect two-dimensional [row,column] indexing.
  # These methods use numbers (not characters) 0 to 9 for cell contents.
  # 0 represents an unknown value.
  #
  # The has_duplicates? predicate returns true if the puzzle is invalid
  # because any row, column, or box includes the same digit twice.
  #
  # The each_unknown method is an iterator that loops through the cells of
  # the puzzle and invokes the associated block once for each cell whose
  # value is unknown.
  #
  # The possible method returns an array of integers in the range 1..9.
  # The elements of the array are the only values allowed in the specified
  # cell. If this array is empty, then the puzzle is over-specified and
  # cannot be solved. If the array has only one element, then that element
  # must be the value for that cell of the puzzle.
  #
  class Puzzle

    # These constants are used for translating between the external
    # string representation of a puzzle and the internal representation.
    ASCII = ".123456789"
    BIN = "\000\001\002\003\004\005\006\007\010\011"

    # This is the initialization method for the class. It is automatically
    # invoked on new Puzzle instances created with Puzzle.new. Pass the input
    # puzzle as an array of lines or as a single string. Use ASCII digits 1
    # to 9 and use the '.' character for unknown cells. Whitespace,
```

```ruby
    # including newlines, will be stripped.
    def initialize(lines)
      if (lines.respond_to? :join)  # If argument looks like an array of lines
        s = lines.join              # Then join them into a single string
      else                          # Otherwise, assume we have a string
        s = lines.dup               # And make a private copy of it
      end

      # Remove whitespace (including newlines) from the data
      # The '!' in gsub! indicates that this is a mutator method that
      # alters the string directly rather than making a copy.
      s.gsub!(/\s/, "")  # /\s/ is a Regexp that matches any whitespace

      # Raise an exception if the input is the wrong size.
      # Note that we use unless instead of if, and use it in modifier form.
      raise Invalid, "Grid is the wrong size" unless s.size == 81

      # Check for invalid characters, and save the location of the first.
      # Note that we assign and test the value assigned at the same time.
      if i = s.index(/[^123456789\.]/)
        # Include the invalid character in the error message.
        # Note the Ruby expression inside #{} in string literal.
        raise Invalid, "Illegal character #{s[i,1]} in puzzle"
      end

      # The following two lines convert our string of ASCII characters
      # to an array of integers, using two powerful String methods.
      # The resulting array is stored in the instance variable @grid
      # The number 0 is used to represent an unknown value.
      s.tr!(ASCII, BIN)      # Translate ASCII characters into bytes
      @grid = s.unpack('c*') # Now unpack the bytes into an array of numbers

      # Make sure that the rows, columns, and boxes have no duplicates.
      raise Invalid, "Initial puzzle has duplicates" if has_duplicates?
    end

    # Return the state of the puzzle as a string of 9 lines with 9
    # characters (plus newline) each.
    def to_s
      # This method is implemented with a single line of Ruby magic that
      # reverses the steps in the initialize() method. Writing dense code
      # like this is probably not good coding style, but it demonstrates
      # the power and expressiveness of the language.
      #
      # Broken down, the line below works like this:
      # (0..8).collect invokes the code in curly braces 9 times--once
      # for each row--and collects the return value of that code into an
      # array. The code in curly braces takes a subarray of the grid
      # representing a single row and packs its numbers into a string.
      # The join() method joins the elements of the array into a single
      # string with newlines between them. Finally, the tr() method
      # translates the binary string representation into ASCII digits.
```

```
    (0..8).collect{|r| @grid[r*9,9].pack('c9')}.join("\n").tr(BIN,ASCII)
  end

  # Return a duplicate of this Puzzle object.
  # This method overrides Object.dup to copy the @grid array.
  def dup
    copy = super       # Make a shallow copy by calling Object.dup
    @grid = @grid.dup  # Make a new copy of the internal data
    copy               # Return the copied object
  end

  # We override the array access operator to allow access to the
  # individual cells of a puzzle. Puzzles are two-dimensional,
  # and must be indexed with row and column coordinates.
  def [](row, col)
    # Convert two-dimensional (row,col) coordinates into a one-dimensional
    # array index and get and return the cell value at that index
    @grid[row*9 + col]
  end

  # This method allows the array access operator to be used on the
  # lefthand side of an assignment operation. It sets the value of
  # the cell at (row, col) to newvalue.
  def []=(row, col, newvalue)
    # Raise an exception unless the new value is in the range 0 to 9.
    unless (0..9).include? newvalue
      raise Invalid, "illegal cell value"
    end
    # Set the appropriate element of the internal array to the value.
    @grid[row*9 + col] = newvalue
  end

  # This array maps from one-dimensional grid index to box number.
  # It is used in the method below. The name BoxOfIndex begins with a
  # capital letter, so this is a constant. Also, the array has been
  # frozen, so it cannot be modified.
  BoxOfIndex = [
    0,0,0,1,1,1,2,2,2,0,0,0,1,1,1,2,2,2,0,0,0,1,1,1,2,2,2,
    3,3,3,4,4,4,5,5,5,3,3,3,4,4,4,5,5,5,3,3,3,4,4,4,5,5,5,
    6,6,6,7,7,7,8,8,8,6,6,6,7,7,7,8,8,8,6,6,6,7,7,7,8,8,8
  ].freeze

  # This method defines a custom looping construct (an "iterator") for
  # Sudoku puzzles.  For each cell whose value is unknown, this method
  # passes ("yields") the row number, column number, and box number to the
  # block associated with this iterator.
  def each_unknown
    0.upto 8 do |row|             # For each row
      0.upto 8 do |col|           # For each column
        index = row*9+col         # Cell index for (row,col)
        next if @grid[index] != 0 # Move on if we know the cell's value
        box = BoxOfIndex[index]   # Figure out the box for this cell
```

```ruby
        yield row, col, box       # Invoke the associated block
      end
    end
  end

  # Returns true if any row, column, or box has duplicates.
  # Otherwise returns false. Duplicates in rows, columns, or boxes are not
  # allowed in Sudoku, so a return value of true means an invalid puzzle.
  def has_duplicates?
    # uniq! returns nil if all the elements in an array are unique.
    # So if uniq! returns something then the board has duplicates.
    0.upto(8) {|row| return true if rowdigits(row).uniq! }
    0.upto(8) {|col| return true if coldigits(col).uniq! }
    0.upto(8) {|box| return true if boxdigits(box).uniq! }

    false  # If all the tests have passed, then the board has no duplicates
  end

  # This array holds a set of all Sudoku digits. Used below.
  AllDigits = [1, 2, 3, 4, 5, 6, 7, 8, 9].freeze

  # Return an array of all values that could be placed in the cell
  # at (row,col) without creating a duplicate in the row, column, or box.
  # Note that the + operator on arrays does concatenation but that the -
  # operator performs a set difference operation.
  def possible(row, col, box)
    AllDigits - (rowdigits(row) + coldigits(col) + boxdigits(box))
  end

  private  # All methods after this line are private to the class

  # Return an array of all known values in the specified row.
  def rowdigits(row)
    # Extract the subarray that represents the row and remove all zeros.
    # Array subtraction is set difference, with duplicate removal.
    @grid[row*9,9] - [0]
  end

  # Return an array of all known values in the specified column.
  def coldigits(col)
    result = []                # Start with an empty array
    col.step(80, 9) {|i|       # Loop from col by nines up to 80
      v = @grid[i]             # Get value of cell at that index
      result << v if (v != 0)  # Add it to the array if non-zero
    }
    result                     # Return the array
  end

  # Map box number to the index of the upper-left corner of the box.
  BoxToIndex = [0, 3, 6, 27, 30, 33, 54, 57, 60].freeze

  # Return an array of all the known values in the specified box.
```

```
  def boxdigits(b)
    # Convert box number to index of upper-left corner of the box.
    i = BoxToIndex[b]
    # Return an array of values, with 0 elements removed.
    [
      @grid[i],    @grid[i+1],  @grid[i+2],
      @grid[i+9],  @grid[i+10], @grid[i+11],
      @grid[i+18], @grid[i+19], @grid[i+20]
    ] - [0]
  end
end  # This is the end of the Puzzle class

# An exception of this class indicates invalid input,
class Invalid < StandardError
end

# An exception of this class indicates that a puzzle is over-constrained
# and that no solution is possible.
class Impossible < StandardError
end

#
# This method scans a Puzzle, looking for unknown cells that have only
# a single possible value. If it finds any, it sets their value. Since
# setting a cell alters the possible values for other cells, it
# continues scanning until it has scanned the entire puzzle without
# finding any cells whose value it can set.
#
# This method returns three values. If it solves the puzzle, all three
# values are nil. Otherwise, the first two values returned are the row and
# column of a cell whose value is still unknown. The third value is the
# set of values possible at that row and column. This is a minimal set of
# possible values: there is no unknown cell in the puzzle that has fewer
# possible values. This complex return value enables a useful heuristic
# in the solve() method: that method can guess at values for cells where
# the guess is most likely to be correct.
#
# This method raises Impossible if it finds a cell for which there are
# no possible values. This can happen if the puzzle is over-constrained,
# or if the solve() method below has made an incorrect guess.
#
# This method mutates the specified Puzzle object in place.
# If has_duplicates? is false on entry, then it will be false on exit.
#
def Sudoku.scan(puzzle)
  unchanged = false  # This is our loop variable

  # Loop until we've scanned the whole board without making a change.
  until unchanged
    unchanged = true      # Assume no cells will be changed this time
    rmin,cmin,pmin = nil  # Track cell with minimal possible set
    min = 10              # More than the maximal number of possibilities
```

```ruby
    # Loop through cells whose value is unknown.
    puzzle.each_unknown do |row, col, box|
      # Find the set of values that could go in this cell
      p = puzzle.possible(row, col, box)

      # Branch based on the size of the set p.
      # We care about 3 cases: p.size==0, p.size==1, and p.size > 1.
      case p.size
      when 0  # No possible values means the puzzle is over-constrained
        raise Impossible
      when 1  # We've found a unique value, so set it in the grid
        puzzle[row,col] = p[0] # Set that position on the grid to the value
        unchanged = false      # Note that we've made a change
      else    # For any other number of possibilities
        # Keep track of the smallest set of possibilities.
        # But don't bother if we're going to repeat this loop.
        if unchanged && p.size < min
          min = p.size                    # Current smallest size
          rmin, cmin, pmin = row, col, p  # Note parallel assignment
        end
      end
    end
  end

  # Return the cell with the minimal set of possibilities.
  # Note multiple return values.
  return rmin, cmin, pmin
end

# Solve a Sudoku puzzle using simple logic, if possible, but fall back
# on brute-force when necessary. This is a recursive method. It either
# returns a solution or raises an exception. The solution is returned
# as a new Puzzle object with no unknown cells. This method does not
# modify the Puzzle it is passed. Note that this method cannot detect
# an under-constrained puzzle.
def Sudoku.solve(puzzle)
  # Make a private copy of the puzzle that we can modify.
  puzzle = puzzle.dup

  # Use logic to fill in as much of the puzzle as we can.
  # This method mutates the puzzle we give it, but always leaves it valid.
  # It returns a row, a column, and set of possible values at that cell.
  # Note parallel assignment of these return values to three variables.
  r,c,p = scan(puzzle)

  # If we solved it with logic, return the solved puzzle.
  return puzzle if r == nil

  # Otherwise, try each of the values in p for cell [r,c].
  # Since we're picking from a set of possible values, the guess leaves
  # the puzzle in a valid state. The guess will either lead to a solution
```

```
    # or to an impossible puzzle. We'll know we have an impossible
    # puzzle if a recursive call to scan throws an exception. If this happens
    # we need to try another guess, or re-raise an exception if we've tried
    # all the options we've got.
    p.each do |guess|        # For each value in the set of possible values
      puzzle[r,c] = guess    # Guess the value

      begin
        # Now try (recursively) to solve the modified puzzle.
        # This recursive invocation will call scan() again to apply logic
        # to the modified board, and will then guess another cell if needed.
        # Remember that solve() will either return a valid solution or
        # raise an exception.
        return solve(puzzle)  # If it returns, we just return the solution
      rescue Impossible
        next                  # If it raises an exception, try the next guess
      end
    end

    # If we get here, then none of our guesses worked out
    # so we must have guessed wrong sometime earlier.
    raise Impossible
  end
end
```