Ruby Fundamentals - Introduction To Ruby

April 14, 2025

1 Introduction

Ruby is a dynamic programming language with a complex but expressive grammar and a core class library with a rich and powerful API. Ruby draws inspiration from Lisp, Smalltalk, and Perl, but uses a grammar that is easy for C and Java programmers to learn. Ruby is a pure object-oriented language, but it is also suitable for procedural and functional programming styles. It includes powerful meta-programming capabilities and can be used to create domain-specific languages or DSLs.

2 A Tour of Ruby

This section is a guided, but meandering, tour through some of the most interesting features of Ruby. Everything discussed here will be documented in detail later in the book, but this first look will give you the flavor of the language.

2.1 Ruby is Object-Oriented

We'll begin with the fact that Ruby is a *completely* object-oriented language. Every value is a an object, even simple numeric literals and the values true, false, and nil (nil is a special value that indicates the absence of value; it is Ruby's version of null). Here we invoke a method named class on these values. Comments begin with `#` in Ruby, and the `=>` arrows in the comments indicate the value returned by the commented code (this is a convention used throughout this book):

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

In many languages, function and method invocations require parentheses, but there are no parentheses in any of the code above. In Ruby, parentheses are usually optional and they are commonly omitted, especially when the method being invoked takes no arguments. The fact that the parentheses are omitted in the method invocations here makes them look like references to named fields or named variables of the object.

2.2 Blocks and Iterators

The fact that we can invoke methods on integers isn't just an esoteric aspect of Ruby. It is actually something that Ruby programmers do with some frequency.

```
3.times {print "Ruby!"} # Prints "Ruby!Ruby!Ruby!"
```

They are a special kind of method known as an *iterator*, and they behave like loops. The code within curly braces - known as a *block* - is associated with the method invocation and serves as the body of the loop. The use of iterators and blocks is another notabler feature of Ruby; although the language does support an ordinary while loop, it is more common to perform loops with constructs that are actually method calls.

Integers are not only the only values that have iterator methods. Arrays (and similar ”enumerable” objects) define an iterator named each, which invokes the associated block once for each element in the array. Each invocation of the blocks is passed a single element from the array:

```
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 {}
```

Various other useful iterators are defined on top of each:

```
a[1, 2, 3, 4] # Start with an array
b = a.map{|x| x*x} # Square elements
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
```

Hashes, like arrays, are a fundamental data structure in Ruby. As their name implies, they are based on the hashtable data strucutre and serve to map arbitrary key objects to value objects. (To put this another way, we can say that a hash associates arbitrary value objects with key objects) Hashes user square brackets, like arrays do, to query and set values in the hash. Instead of using an integer index, they expect a key objects within the square brackets. Like the Array class, the Hash class also defines on each iterator method. This method invokes the associated block of code once for each key/value pair in the hash, and passes both key and the value as parameters to the block:

```ruby
h = {
 :one => 1,
 :two => 2
}
h[:one]
h[:three] = 3
h.each do |key, value|
    print "#{value}: #{key};"
end
```

Ruby's hashes can use any object as a key, but Symbol objects are the most commonly used. Symbols are immutable, interned strings. They can compared by identity rather than by textual content (two distinct Symbol objects will never have the same content).

The ability to associate a block of code with a method invocation is a fundamental and very powerful feature of Ruby. Although its most obvious use is for loop-like constructs, it is also useful for methods that only invoke the block once.

Double quoted strings can include arbitrary Ruby expressions delimited by # and . The value of the expression within these delimiters is converted to a string. The resulting string is then used in the string literal. This substitution of expression values into strings is usually called *string interpolation*.

2.3 Expressions and Operators in Ruby

Ruby's syntax is expression-oriented. Control structures such as if that would be called statements in other languages are actually expressions in Ruby. They have values like other simpler expressions do, and we can write code like this:

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

Although all statements in Ruby are actually expressions, they do not all return meaningful values. While loops and method definitions, for example, are expressions that normally return the value nil.

As in most languages, expressions in Ruby are usually built out of values and operators. For the most part, Ruby's operators will be familiar to anyone who knows C, Java, JavaScript, or any similar programming language. Here are examples of some commonplace and some more unusual Ruby operators:

```ruby
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
"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
```

Many of Ruby's operators are implemented as methods, and classes can define (or redefine) these methods however they want. (They can't define completely new operators, however; there is only a fixed set of recognized operators). As examples, notice that the `+` and `*` operators behave differently for integers and strings. And you can define these operators any way you want in your own classes. The `<<` operator is another good example. The integer classes Fixnum and Bignum uses this operator for the bitwise left-shift operation, following the C programming language. At the same time, other classes - such as strings, arrays, and streams - use this operator for an append operation.

If you create a new class that can have values appended to it in some way, it is a very good idea to define `<<`.

One of the most powerful operators to override is `[]`. The Array and Hash classes use this operator to access array elements by index and hash values by key. But you can define `[]` in your classes for any purpose you want. You can even define it as a method that expects multiple arguments, comma-separated between the square brackets. And if you want to allow square brackets to be used on the left hand side of an assignment expression, you can define the corresponding `[]=` operator. The value on the right hand side of the assignment will be passed as the final argument to the method that implements this operator.

2.4 Methods

Methods are defined with the `def` keyword. The return value of a method is the value of the last expression evaluated in its body:

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

When a method, like the one above, is defined outside of a class or module, it is effectively a global function rather than a method to be invoked on an object. (Technically, however, a method like this become a private method of the Object class). Methods can also be defined on individual objects by prefixing the name of the method with the object on which it is defined. Methods like these are known as *singleton methods*, and they are how Ruby defines class methods:

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

The Math module is part of the core Ruby library, and this code adds a new method to it. This is a key feature of Ruby - classes and modules are "open" and can be modified and extended at runtime.

Method parameters may have default values specified, and methods may accept arbitrary numbers of arguments.

2.5 Assignment

The = operator in Ruby assigns a value to a variable:

```ruby
x = 1
```

Assignment can be combined with other operators such as `+` and `-`:

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

Ruby supports parallel assignment, allowing more than one value and more than one variable in assignment expressions:

```ruby
x,y = 1,2
a,b = b,a
x,y,z = [1,2,3] # Array elements automatically assigned to variables
```

Methods in Ruby are allowed to return more than one value, and parallel assignment is helpful in conjunction with such methods. Methods that end with an equal sign are special because Ruby allows them to be invoked using assignment syntax.

2.6 Punctuation Suffixes and Prefixes

We saw previously that methods whose names end with `=` can be invoked by assignment expressions. Ruby methods can also end with a question mark or an exclamation point. A question mark is used to mark predicates - methods that return a Boolean value. For example, the Array and Hash classes both define a method named `empty?` that test whether the data structure has any elements. An exclamation mark at the end of a method name is used to indicate that caution is required with the use of the method. A number of core Ruby classes define pairs of methods with the same name, except that one ends with an exclamation mark and one does not. Usually, the method without the exclamation mark returns a modified copy of the object it is invoked on, and the one with exclamation mark is a mutator method that alters the object in place. The Array class, for example, defines methods `sort` and `sort!`. In addition to these punctuation characters at the end of method names, you'll notice punctuation characters at the start of Ruby variable names: global variables are prefixed with `$`, instance variables are prefixed with , and class variables are prefixed with . These prefixes can take a little getting used to, but after a while you may come to appreciate the fact that the prefix tells you the scope of the variable. The prefixes are required in order to disambiguate Ruby's very flexible grammar. One way to think of variable prefixes is that they are one price we pay for being able to omit parentheses around method invocations.

2.7 Regexp and Range

We mentioned arrays and hashs earlier as fundamental data structures in Ruby. We demonstrated the use of numbers and strings as well. Two other datatypes are worth mentioning here. A Regexp (regular expression) object describves a textual pattern and has methods for determining whether a given string matches that pattern or not. And a Range represents the value (usually integers) between two endpoints. Regular expressions and ranges have a literal syntax in Ruby:

```ruby
/[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
```

Regexp and Range objects define the normal `==` operator for testing equality. In addition, they also define the `===` operator for testing matching and membership. Ruby's case statement (like the switch statement of C or Java) matches its expression against each of the possible cases using `===`, so this operator is often called the *case equality operator*. It leads to conditional tests like these:

```
generation = case birthyear
    when 1946..1963: "Baby Boomer"
    when 1964..1976: "Generation X"
    when 1978..2000: "Generation Y"
    else nil
    end
```

2.8 Classes and Modules

A class is a collection of related methods that operate on the state of an object. An object's state is held by its instance variables: variables whose names begin with and whose values are specific to that particular object. The following code defines an example class named Sequence and demonstrates how to write iterator methods and define operators:

```
#
# This class represents a sequence of numbers characterized by the three parameters
# from, to, and by. The numbers x in the sequence obey the following two constraints:
#
# from <= x <= to
# x = from  + n*by, where n is an integer
#
# This is an enumerable class; it defines an each iterator below.
include Enumerable # Include the methods of this module in this class

class

    # Thew initialize method is special; it is automatically invoked to initialize
    # newly created instance of the class
    def initialize(from, to, by)
        # Just save our parameters into instance variables for later use
        @from, @to, @by = from, to , by
    end
    # This is the iterator required by the Enumerable module
    def each
        x = @from # Start at the starting point
```

```ruby
        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
        Integer((@to-@from)/@by) + 1
    end

    # Define another name for the same method
    # It is common for methods to havem ultiple 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
```

Here's some code that uses this Sequence class:

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

The key feature of our Sequence class is its each iterator. If we are only interested in the iterator method, there is no need to define the whole class. Instead, we can simply write an iterator method that accepts the from, to, and by parameters. Instead of making this a global function, let's define it in a module of its own:

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

With the iterator defined this way, we write code like this:

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

An iterator like this makes it unnecessary to create a Sequence object to iterate a sequence of numbers. But the name of the method is quite long, and its invocation syntax is unsatisfying. What we really want is a way to iterate numeric Range objects by steps other than 1. One of the amazing feature of Ruby is that its classes, even the built-in core classes, are open: any program can add methods to them. So we really can define a new iterator method for ranges.

2.9 Ruby Surprises

Ruby's strings are mutable, which may be surprising to Java programmers in particular. The `[]=` operator allows you to alter the characters of a string or to insert, delete, and replace substrings. The `<<` operator allows you to append to a string, and the String class defines various other methods that alter strings in place. Because strings are mutable, string literals in a program are not unique objects. If you include a string literal within a loop, it evaluates to a new object on each iteration of the loop. Call the freeze method on a string (or on any object) to prevent any future modifications to that object.

Ruby's coinditionals and loops (such as if and while) evaluate conditional expressions to determine which branch to evaluate or whether to continue looping. Conditional expressions often evaluate to true or false, but this is not required. The value of nil is treated the same as false, and *any other value is the same as true.*