1.5. Training Your Intuition: Things to Remember

It may truly be said that "everything is intuitive once you understand it." This verity is the heart of this section because Ruby has many features and personality quirks that may be different from what the traditional programmer is used to.

Some readers may feel their time is wasted by a reiteration of some of these points; if that is the case for you, you are free to skip the paragraphs that seem obvious to you. Programmers' backgrounds vary widely; an old-time C hacker and a Smalltalk guru will each approach Ruby from different viewpoints. We hope, however, that a perusal of these following paragraphs will assist many readers in following what some call *the Ruby Way*.

1.5.1. Syntax Issues

The Ruby parser is complex and relatively forgiving. It tries to make sense out of what it finds instead of forcing the programmer into slavishly following a set of rules. However, this behavior may take some getting used to. Here is a list of things to know about Ruby syntax:

- Parentheses are usually optional with a method call. These calls are all valid:

  ```
  foobar
  foobar()
  foobar(a,b,c)
  foobar a, b, c
  ```

- Given that parentheses are optional, what does `x y z` mean, if anything? As it turns out, this means, "Invoke method `y`, passing `z` as a parameter, and pass the result as a parameter to method `x`." In short, the statement `x(y(z))` means the same thing.

  This behavior will change in the future. Refer to the discussion of poetry mode in section 1.6 "Ruby Jargon and Slang" later in the chapter.

- Let's try to pass a hash to a method:

  ```
  my_method {a=>1, b=>2}
  ```

  This results in a syntax error, because the left brace is seen as the start of a block. In this instance, parentheses are necessary:

  ```
  my_method({a=>1, b=>2})
  ```

- Now let's suppose that the hash is the *only* parameter (or the last parameter) to a method. Ruby forgivingly lets us omit the braces:

  ```
  my_method(a=>1, b=>2)
  ```

  Some people might think that this looks like a method invocation with named parameters. Really it isn't, though it could be used as such.

- There are other cases in which blank spaces are semi-significant. For example, these four expressions may all seem to mean the same:

  ```
  x = y + z
  x = y+z
  x = y+ z
  x = y +z
  ```

  And in fact, the first three do mean the same. However, in the fourth case, the parser thinks that `y` is a method call and `+z` is a parameter passed to it! It will then give an error message for that line if there is no method named `y`. The moral is to use blank spaces in a reasonable way.

- Similarly, `x = y*z` is a multiplication of y and z, whereas `x = y *z` is an invocation of method `y`, passing an expansion of array `z` as a parameter.

- In constructing identifiers, the underscore is considered to be lowercase. Thus an identifier may start with an underscore, but it will *not* be a constant even if the next letter is uppercase.

- In linear nested-`if` statements, the keyword `elsif` is used rather than `else if` or `elif` as in some languages.

- Keywords in Ruby are not really reserved words. When a method is called with a receiver (or in other cases where there is no ambiguity), a keyword may be used as a method name. Do this with caution, remembering that programs should be readable by humans.

- The keyword `then` is optional (in `if` and `case` statements). Those who want to use it for readability may do so. The same is true for `do` in `while` and `until` loops.

- The question mark and exclamation point are not really part of the identifier that they modify but should be considered suffixes. Thus we see that although, for example, `chop` and `chop!` are considered different identifiers, it is not permissible to use these characters in any other position in the word. Likewise, we use `defined?` in Ruby, but `defined` is the keyword.

- Inside a string, the pound sign (`#`) is used to signal expressions to be evaluated. That means that in some circumstances, when a pound sign occurs in a string, it has to be escaped with a backslash, but this is *only* when the next character is a `{` (left brace), `$` (dollar sign), or `@` (at sign).

- Because of the fact that the question mark may be appended to an identifier, care should be taken with spacing around the ternary operator. For example, suppose we have a variable `my_flag`, which stores either `true` or `false`. Then the first line of code shown here will be correct, but the second will give a syntax error:

```
x = my_flag ? 23 : 45    # OK
x = my_flag? 23 : 45     # Syntax error
```

- The ending marker `=end` for embedded documentation should not be considered a token. It marks the entire line and thus any characters on the rest of that line are not considered part of the program text but belong to the embedded document.

- There are no arbitrary blocks in Ruby; that is, you can't start a block whenever you feel like it as in C. Blocks are allowed only where they are needed—for example, attached to an iterator. The exception is the `begin-end` block, which can be used basically anywhere.

- Remember that the keywords `BEGIN` and `END` are completely different from the `begin` and `end` keywords.

- When strings bump together (static concatenation), the concatenation is of a lower precedence than a method call. For example:

```
str = "First " 'second'.center(20)      # Examples 1 and 2
str = "First " + 'second'.center(20)    # are the same.
str = "First second".center(20)         # Examples 3 and 4
str = ("First " + 'second').center(20) # are the same.
```

- Ruby has several pseudovariables, which look like local variables but really serve specialized purposes. These are `self`, `nil`, `true`, `false`, `__FILE__`, and `__LINE__`.

1.5.2. Perspectives in Programming

Presumably everyone who knows Ruby (at this point in time) has been a student or user of other languages in the past. This of course makes learning Ruby easy in the sense that numerous features in Ruby are just like the corresponding features in other languages. On the other hand, the programmer may be lulled into a false sense of security by some of the familiar constructs in Ruby and may draw unwarranted conclusions based on past experience—which we might term "geek baggage."

Many people are coming to Ruby from Smalltalk, Perl, C/C++, and various other languages. Their presuppositions and expectations may all vary somewhat, but they will always be present. For this reason, we discuss here a few of the things that some programmers may "trip over" in using Ruby:

- A character in Ruby truly is an integer. It is not a type of its own, as in Pascal, and is not the same as a string of length `1`. *This is slated to change in the near future so that a character constant will be much the same as a string. As of the time of this writing, this has not happened yet*. Consider the following code fragment:

```
x = "Hello"
y = ?A
puts "x[0] = #{x[0]}"    # Prints: x[0] = 72
puts "y = #{y}"          # Prints: y = 65
if y == "A"              # Prints: no
```

```
  puts "yes"
else
  puts "no"
end
```

- There is no Boolean type such as many languages have. `TrueClass` and `FalseClass` are distinct classes, and their only instantiations are `true` and `false`.

- Many of Ruby's operators are similar or identical to those in C. Two notable exceptions are the increment and decrement operators (`++` and `--`). These are not available in Ruby, neither in "pre" nor "post" forms.

- The modulus operator is known to work somewhat differently in different languages with respect to negative numbers. The two sides of this argument are beyond the scope of this book; Ruby's behavior is as follows:

```
puts (5 % 3)     # Prints 2
puts (-5 % 3)    # Prints 1
puts (5 % -3)    # Prints -1
puts (-5 % -3)   # Prints -2
```

- Some may be used to thinking that a false value may be represented as a zero, a null string, a null character, or various other things. But in Ruby, all of these are true; in fact, *everything is true* except `false` and `nil`.

- In Ruby, variables don't have classes; only values have classes.

- There are no declarations of variables in Ruby. It is good practice, however, to assign `nil` to a variable initially. This certainly does not assign a type to the variable and does not truly initialize it, but it does inform the parser that this is a variable name rather than a method name.

- `ARGV[0]` is truly the first of the command-line parameters, numbering naturally from zero; it is not the file or script name preceding the parameters, like `argv[0]` in C.

- Most of Ruby's operators are really methods; the "punctuation" form of these methods is provided for familiarity and convenience. The first exception is the set of reflexive assignment operators (`+=`, `-=`, `*=`, and so on); the second exception is the following set: `= .. ... ! not && and || or != !~.`

- Like most (though not all) modern languages, Boolean operations are always short-circuited; that is, the evaluation of a Boolean expression stops as soon as its truth value is known. In a sequence of `or` operations, the first `true` will stop evaluation; in a string of `and` operations, the first `false` will stop evaluation.

- The prefix `@@` is used for class variables (which are associated with the class rather than the instance).

- `loop` is not a keyword; it is a `Kernel` method, not a control structure.

- Some may find the syntax of `unless-else` to be slightly unintuitive. Because `unless` is the opposite of `if`, the `else` clause will be executed if the condition is *true*.

- The simpler `Fixnum` type is passed as an immediate value and thus may not be changed from within methods. The same is true for `true`, `false`, and `nil`.

- Do not confuse the `&&` and `||` operators with the `&` and `|` operators. These are used as in C; the former are for Boolean operations, and the latter are for arithmetic or bitwise operations.

- The `and-or` operators have lower precedence than the `&&-||` operators. See the following code fragment:

```
a = true
b = false
c = true
d = true
a1 = a && b or c && d   # &&'s are done first
a2 = a && (b or c) && d # or is done first
puts a1                 # Prints false
puts a2                 # Prints true
```

- Additionally, be aware that the assignment "operator" has a *higher* precedence than the `and` and `or` operators! (This is also true for the reflexive assignment operators `+=`, `-=`, and the others.) For example, in the following code `x = y or z` looks like a normal assignment statement, but it is really a freestanding expression (equivalent to `(x=y) or z`, in fact). The third section shows a real assignment statement `x = (y or z)`, which may be what the programmer really intended.

```
y = false
z = true

x = y or z     # = is done BEFORE or!
puts x         # Prints false
```

```
(x = y) or z   # Line 5: Same as previous
puts x         # Prints false

x = (y or z)   # or is done first
puts x         # Prints true
```

- Don't confuse object attributes and local variables. If you are accustomed to C++ or Java, you might forget this. The variable `@my_var` is an instance variable (or attribute) in the context of whatever class you are coding, but `my_var`, used in the same circumstance, is only a local variable within that context.

- Many languages have some kind of `for` loop, as does Ruby. The question sooner or later arises as to whether the index variable can be modified. Some languages do not allow the control variable to be modified at all (printing a warning or error either at compile time or runtime), and some will cheerfully allow the loop behavior to be altered in midstream by such a change. Ruby takes yet a third approach. When a variable is used as a `for` loop control variable, it is an ordinary variable and can be modified at will; however, such a modification does not affect the loop behavior! The `for` loop sequentially assigns the values to the variable on each iteration without regard for what may have happened to that variable inside the loop. For example, this loop will execute exactly 10 times and print the values `1` through `10`:

  ```
  for var in 1..10
    puts "var = #{var}"
    if var > 5
      var = var + 2
    end
  end
  ```

- Variable names and method names are not always distinguishable "by eye" in the immediate context. How does the parser decide whether an identifier is a variable or a method? The rule is that if the parser sees the identifier being assigned a value prior to its being used, it will be considered a variable; otherwise, it is considered to be a method name. (Note also that the assignment does not have to be *executed* but only *seen* by the interpreter.)

1.5.3. Ruby's `case` Statement

Every modern language has some kind of multiway branch, such as the `switch` statement in C/C++ and Java or the `case` statement in Pascal. These serve basically the same purpose and function much the same in most languages.

Ruby's `case` statement is similar to these others, but on closer examination it is so unique that it makes C and Pascal look like close friends. The `case` statement in Ruby has no precise analogue in any other language that I am familiar with, and this makes it worth additional attention here.

We have already seen the syntax of this statement. We will concentrate here on its actual semantics:

- To begin with, consider the trivial `case` statement shown here. The expression shown is compared with the value, not surprisingly, and if they correspond, `some_action` is performed.

  ```
  case expression
    when value
      some_action
  end
  ```

  Ruby uses the special operator `===` (called the *relationship operator*) for this. This operator is also referred to (somewhat inappropriately) as the *case equality* operator. We say "inappropriately" because it does not always denote equality.

- Thus the preceding simple statement is equivalent to this statement:

  ```
  if value === expression
    some_action
  end
  ```

- However, do not confuse the relationship operator with the equality operator (`==`). They are utterly different, although their behavior may be the same in many circumstances. The relationship operator is defined differently for different classes and, for a given class, may behave differently for different operand types passed to it.

- Do not fall into the trap of thinking that the tested expression is the receiver and the value is passed as a parameter to it. The opposite is true (as we saw previously).

- This points up the fact that `x === y` is *not* typically the same as `y === x`! There will be situations in which this is true, but overall the relationship operator is not commutative. (That is why we do not favor the term *case equality operator*—because equality is always commutative.) In other words, reversing our original example, the following code does not behave the same way:

  ```
  case value
  ```

```
  when expression
    some_action
end
```

- As an example, consider a string `str` and a pattern (regular expression) `pat`, which matches that string. The expression `str =~ pat` is true, just as in Perl. Because Ruby defines the opposite meaning for `=~` in Regexp, you can also say that `pat =~ str` is true. Following this logic further, we find that (because of how `Regexp` defines `===`) `pat === str` is also true. However, note that `str === pat` is *not* true. This means that this code fragment:

  ```
  case "Hello"
    when /Hell/
      puts "We matched."
  else
    puts "We didn't match."
  end
  ```

  does not do the same thing as this fragment:

  ```
  case /Hell/
    when "Hello"
      puts "We matched."
  else
    puts "We didn't match."
  end
  ```

  If this confuses you, just memorize the behavior. If it does not confuse you, so much the better.

- Programmers accustomed to C may be puzzled by the absence of `break` statements in the `case` statement; such a usage of `break` in Ruby is unnecessary (and illegal). This is due to the fact that "falling through" is rarely the desired behavior in a multiway branch. There is an implicit jump from each `when`-clause (or *case limb*, as it is sometimes called) to the end of the `case` statement. In this respect, Ruby's `case` statement resembles the one in Pascal.

- The values in each case limb are essentially arbitrary. They are not limited to any certain type. They need not be constants but can be variables or complex expressions. Ranges or multiple values can be associated with each case limb.

- Case limbs may have empty actions (null statements) associated with them. The values in the limbs need not be unique, but may overlap. Look at this example:

  ```
  case x
    when 0
    when 1..5
      puts "Second branch"
    when 5..10
      puts "Third branch"
  else
    puts "Fourth branch"
  end
  ```

  Here a value of `0` for `x` will do nothing; a value of `5` will print `Second branch`, even though `5` is also included in the next limb.

- The fact that case limbs may overlap is a consequence of the fact that they are evaluated in sequence *and* that short-circuiting is done. In other words, if evaluation of the expressions in one limb results in success, the limbs that follow are never evaluated. Thus it is a bad idea for case limb expressions to have method calls that have side effects. (Of course, such calls are questionable in most circumstances anyhow.) Also, be aware that this behavior may mask runtime errors that would occur if expressions were evaluated. For example:

  ```
  case x
    when 1..10
      puts "First branch"
    when foobar()               # Possible side effects?
      puts "Second branch"
    when 5/0                    # Dividing by zero!
      puts "Third branch"
  else
    puts "Fourth branch"
  end
  ```

  As long as `x` is between `1` and `10`, `foobar()` will not be called, and the expression `5/0` (which would naturally result in a runtime error) will not be evaluated.

1.5.4. Rubyisms and Idioms

Much of this material overlaps conceptually with the preceding pages. Don't worry too much about why we divided it as we did; many of these tidbits were difficult to classify or organize. Our most important motivation was simply to break the information into digestible chunks.

Ruby was designed to be consistent and orthogonal. But it is also complex, and so, like every language, it has its own set of idioms and quirks. We discuss some of these in the following list:

- `alias` can be used to give alternate names for global variables and methods.
- The numbered global variables `$1`, `$2`, `$3`, and so on, cannot be aliased.
- We do not recommend the use of the "special variables" such as `$=`, `$_`, `$/`, and the rest. Though they can sometimes make code more compact, they rarely make it any clearer; we use them sparingly in this book and recommend the same practice.
- Do not confuse the `..` and `...` range operators. The former is *inclusive* of the upper bound, and the latter is *exclusive*. For example, `5..10` includes the number `10`, but `5...10` does not.
- There is a small detail relating to ranges that may cause confusion. Given a range `m..n`, the method `end` will return the endpoint of the range; its alias `last` will do the same thing. However, these methods will return the same value `n` for the range `m...n`, even though `n` is not included in the latter range. The method `end_excluded?` is provided to distinguish between these two situations.
- Do not confuse ranges with arrays. These two assignments are entirely different:

  ```
  x = 1..5
  x = [1, 2, 3, 4, 5]
  ```

  However, there is a convenient method `to_a` for converting ranges to arrays. (Many other types also have such a method.)
- Often we want to assign a variable a value only if it does not already have a value. Because an unassigned variable has the value `nil`, we can do for example, `x = x || 5` shortened to `x ||= 5`. Beware that the value `false` will be overwritten just as `nil` will.
- In most languages, swapping two variables takes an additional temporary variable. In Ruby, multiple assignment makes this unnecessary: `x, y = y, x` will interchange the values of `x` and `y`.
- Keep a clear distinction in your mind between *class* and *instance*. For example, a class variable such as `@@foobar` has a classwide scope, but an instance variable such as `@foobar` has a separate existence in each object of the class.
- Similarly, a class method is associated with the class in which it is defined; it does not belong to any specific object and cannot be invoked as though it did. A class method is invoked with the name of a class, and an instance method is invoked with the name of an object.
- In writing about Ruby, the *pound notation* is sometimes used to indicate an instance method—for example, we say `File.chmod` to denote the class method `chmod` of class `File`, and `File#chmod` to denote the instance method that has the same name. This notation is not part of Ruby syntax but only Ruby folklore. We have tried to avoid it in this book.
- In Ruby, constants are not truly constant. They cannot be changed from within instance methods, but otherwise their values *can* be changed.
- In writing about Ruby, the word *toplevel* is common as both an adjective and a noun. We prefer to use *top level* as a noun and *top-level* as an adjective, but our meaning is the same as everyone else's.
- The keyword `yield` comes from CLU and may be misleading to some programmers. It is used within an iterator to invoke the block with which the iterator is called. It does not mean "yield" as in producing a result or returning a value but is more like the concept of "yielding a timeslice."
- The reflexive assignment operators `+=`, `-=`, and the rest, are not methods (nor are they really operators); they are only "syntax sugar" or "shorthand" for their longer forms. Thus, to say `x += y` is really identical to saying `x = x + y`, and if the `+` operator is overloaded, the `+=` operator is defined "automagically" as a result of this predefined shorthand.
- Because of way the reflexive assignment operators are defined, they cannot be used to initialize variables. If the first reference to `x` is `x += 1`, an error will result. This will be intuitive to most programmers unless they are accustomed to a language where variables are initialized to some sort of zero or null value.
- It is actually possible in some sense to get around this behavior. One can define operators for `nil` such that the initial `nil` value of the variable produces the result we want. Here is a `nil.+` method that will allow `+= to`

initialize a `String` or a `Fixnum` value, basically just returning `other` and thus ensuring that `nil + other` is equal to other:

```
def nil.+(other)
  other
end
```

This illustrates the power of Ruby; but whether it is useful or appropriate to code this way is left as an exercise for the reader.

- It is wise to recall that `Class` *is an object*, and `Object` *is a class*. We will try to make this clear in a later chapter; for now, simply recite it every day as a mantra.
- Some operators can't be overloaded because they are built into the language rather than implemented as methods. These are: `= .. ... and or not && || ! != !~`. Additionally, the reflexive assignment operators (`+=, -=`, and so on) cannot be overloaded. These are not methods, and it can be argued they are not true operators either.
- Be aware that although assignment is not overloadable, it is still possible to write an instance method with a name such as `foo=` (allowing statements such as `x.foo = 5`). Consider the equal sign to be like a suffix.
- Recall that a "bare" scope operator has an implied `Object` before it, so that `::Foo` means `Object::Foo`.
- Recall that fail is an `alias` for `raise`.
- Recall that definitions in Ruby are executed. Because of the dynamic nature of the language, it possible (for example) to define two methods completely differently based on a flag that is tested at runtime.
- Remember that the `for` construct (`for x in a`) is really calling the default iterator `each`. Any class having this iterator can be walked through with a `for` loop.
- Be aware that a method defined at the top level is added to `Kernel` and is thus a member of `Object`.
- A "*setter*" method (such as `foo=`) must be called with a receiver; otherwise, it will look like a simple assignment to a local variable of that name.
- Recall that `retry` can be used in iterators but not in general loops. In iterators, it causes the reassignment of all the parameters and the restarting of the current iteration.
- The keyword `retry` is also used in exception handling. Don't confuse the two usages.
- An object's `initialize` method is always private.
- Where an iterator ends in a left brace (or in `end`) and results in a value, that value can be used as the receiver for further method calls. For example:

```
squares = [1,2,3,4,5].collect do |x| x**2 end.reverse
# squares is now [25,16,9,4,1]
```

- The idiom `if $0 == __FILE__` is sometimes seen near the bottom of a Ruby program. This is a check to see whether the file is being run as a standalone piece of code (`true`) or is being used as some kind of auxiliary piece of code such as a library (`false`). A common use of this is to put a sort of "main program" (usually with test code in it) at the end of a library.
- Normal subclassing or inheritance is done with the `<` symbol:

```
class Dog < Animal
  # ...
end
```

But creation of a singleton class (an anonymous class that extends a single instance) is done with the `<<` symbol:

```
class << platypus
  # ...
end
```

- When passing a block to an iterator, there is a slight difference between braces (`{}`) and a `do-end` pair. This is a precedence issue:

```
mymethod param1, foobar do ... end
# Here, do-end binds with mymethod
```

```
mymethod param1, foobar { ... }
# Here, {} binds with foobar, assumed to be a method
```

- It is somewhat traditional in Ruby to put single-line blocks in braces and multiline blocks in `do-end` pairs. Examples:

```
my_array.each { |x| puts x }

my_array.each do |x|
  print x
  if x % 2 == 0
    puts " is even."
  else
    puts " is odd."
  end
end
```

  This habit is not required, and there may be occasions where it is inappropriate to follow this rule.

- Bear in mind that strings are in a sense two-dimensional; they can be viewed as sequences of characters or sequences of lines. Some may find it surprising that the default iterator `each` operates on lines (where a "line" is a group of characters terminated by a record separator that defaults to newline); an alias for `each` is `each_line`. If you want to iterate by characters, you can use `each_byte`. The iterator `sort` also works on a line-by-line basis. There is no iterator `each_index` for strings because of the ambiguity involved. Do we want to handle the string by character or by line? This all becomes habitual with repeated use.

- A closure remembers the context in which it was created. One way to create a closure is by using a `Proc` object. As a crude example, consider the following:

```
def power(exponent)
  proc {|base| base**exponent}
end

square = power(2)
cube = power(3)

a = square.call(11)    # Result is 121
b = square.call(5)     # Result is 25
c = cube.call(6)       # Result is 216
d = cube.call(8)       # Result is 512
```

  Observe that the closure "knows" the value of exponent that it was given at the time it was created.

- However, let's assume that a closure uses a variable defined in an outer scope (which is perfectly legal). This property can be useful, but here we show a misuse of it:

```
$exponent = 0

def power
  proc {|base| base**$exponent}
end

$exponent = 2
square = power

$exponent = 3
cube = power

a = square.call(11)    # Wrong! Result is 1331

b = square.call(5)     # Wrong! Result is 125

# The above two results are wrong because the CURRENT value
# of $exponent is being used. This would be true even if it
# had been a local variable that had gone out of scope (e.g.,
# using define_method).

c = cube.call(6) # Result is 216
d = cube.call(8) # Result is 512
```

- Finally, consider this somewhat contrived example. Inside the block of the `times` iterator, a new context is started, so that `x` is a local variable. The variable `closure` is already defined at the top level, so it will not be defined as local to the block.

```
closure = nil          # Define closure so the name will be known
1.times do             # Start a new context
  x = 5                # x is local to this block
  closure = Proc.new { puts "In closure, x = #{x}" }
```

```
end

x = 1

# Define x at top level

closure.call          # Prints: In closure, x = 5
```

Now note that the variable `x` that is set to `1` is a new variable, defined at the top level. It is not the same as the other variable of the same name. The closure therefore prints `5` because it remembers its creation context with the previous variable `x` and its previous value.

- Variables starting with a single `@`, defined inside a class, are generally instance variables. However, if they are defined outside any method, they are really *class instance* variables. (This usage is somewhat contrary to most OOP terminology in which a class instance is regarded to be the same as an instance or an object.) Example:

```
class Myclass

  @x = 1        # A class instance variable
  @y = 2        # Another one

  def mymethod
    @x = 3      # An instance variable
    # Note that @y is not accessible here.
  end

end
```

The class instance variable `@y` in the preceding code example is really an attribute of the class object `Myclass`, which is an instance of the class `Class`. (Remember, `Class` is an object, and `Object` is a class.) Class instance variables cannot be referenced from within instance methods and, in general, are not very useful.

- `attr`, `attr_reader`, `attr_writer`, and `attr_accessor` are shorthand for the actions of defining "setters" and "getters"; they take symbols as arguments (that is, instances of class `Symbol`).

- There is never any assignment with the scope operator; for example, the assignment `Math::PI = 3.2` is illegal.

1.5.5. Expression Orientation and Other Miscellaneous Issues

In Ruby, expressions are nearly as significant as statements. If you are a C programmer, this will be somewhat familiar to you; if your background is in Pascal, it may seem utterly foreign. But Ruby carries expression orientation even further than C.

In addition, we use this section to remind you of a couple of minor issues regarding regular expressions. Consider them to be a small bonus:

- In Ruby, any kind of assignment returns the same value that was assigned. Thus we can sometimes take little shortcuts like the ones shown here, but be careful when you are dealing with objects! Remember that these are nearly always *references* to objects.

```
x = y = z = 0       # All are now zero.

a = b = c = []      # Danger! a, b, and c now all refer
                    # to the SAME empty array.
x = 5
y = x += 2          # Now x and y are both 7
```

Remember, however, that values such as `Fixnum`s are actually stored as immediate values, not as object references.

- Many control structures return values—`if`, `unless`, and `case`. The following code is all valid; it demonstrates that the branches of a decision need not be statements but can simply be expressions:

```
a = 5
x = if a < 8 then 6 else 7 end    # x is now 6

y = if a < 8        # y is 6 also; the
  6                 # if-statement can be
else                # on a single line
  7                 # or on multiple lines.
end

# unless also works; z will be assigned 4
z = unless x == y then 3 else 4 end

t = case a          # t gets assigned
```

```
  when 0..3         # the value
    "low"           # "medium"
  when 4..6
    "medium"
else
  "high"
end
```

Here we indent as though the `case` started with the assignment. This looks proper to our eyes, but you may disagree.

- Note by way of contrast that the `while` and `until` loops do not return useful values but typically return `nil`:

```
i = 0
x = while (i < 5)    # x is nil
  puts i+=1
end
```

- The ternary decision operator can be used with statements or expressions. For syntactic reasons (or parser limitations) the parentheses here are necessary:

```
x = 6
y = x == 5 ? 0 : 1                  # y is now 1
x == 5 ? puts("Hi") : puts("Bye")   # Prints Bye
```

- The `return` at the end of a method can be omitted. A method always returns the last expression evaluated in its body, regardless of where that happens.

- When an iterator is called with a block, the last expression evaluated in the block is returned as the value of the block. Thus if the body of an iterator has a statement like `x = yield`, that value can be captured.

- *Regular expressions*: Recall that the multiline modifier `/m` can be appended to a regex, in which case `.` (dot) will match a newline character.

- *Regular expressions*: Beware of zero-length matches. If all elements of a regex are optional, then "nothingness" will match that pattern, and a match will always be found at the beginning of a string. This is a common error for regex users, particularly novices.