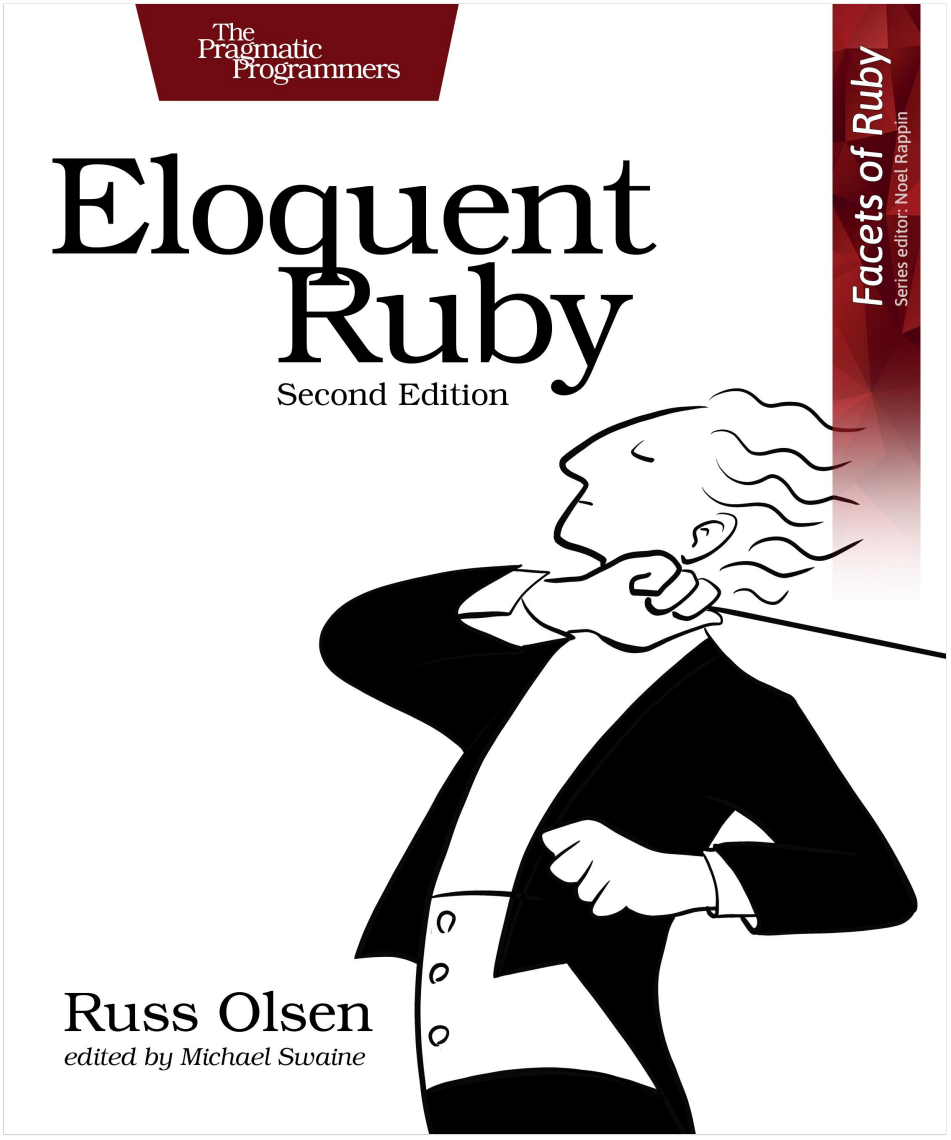

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

If you step back and think about writing programs, it all comes down to a few jobs. Typically we need to get data from somewhere, locate the part of the data that we are interested in, do some computing and then send the results off. Ruby has always provided a great toolkit for pulling data, for doing the computing and for pushing the results back out into the world.

However, until recently, the tools provided by Ruby for the *locate the interesting part of the data* have been pretty scant. Certainly you could ask if this object is an `instance_of?` some class, or if that array `length` is three and if this hash harbors a `:first_name` key. The trouble is data frequently shows up in complex webs of interconnected objects and collections, and working your way through those webs one object, array, or hash at a time can be tedious and error-prone.

The good news is that recent releases of Ruby have included a tool that enables us to do the *locate the interesting* part in a relatively painless, declarative way: pattern matching. As the name suggests, pattern matching is there to help you identify patterns in your data and, even better, to lay your hands on the parts of your data that you care about.

So in this chapter we are going to take a long look at pattern matching, the syntax that it introduces into Ruby, and how you can use it to locate the bits that you are looking for. As usual we will also look at some of the ways that pattern matching has found its way into real-world code and at some of the potholes you might want to swerve to avoid.

Matching Data Like Strings

Let's start by thinking about regular expressions. As we saw in Chapter 4, Find the Right String with Regular Expressions, on page ?, regular expressions exist to make it easy to answer the question *does this string look look like that?*. We saw, for example, that you can write a regular expression that will match strings that look like times:

patmatch/match_data_spec.rb

```
# This regular expression string that looks like a (US) time,
# perhaps "09:24 AM".

TIME_RE = /\d\d:\d\d (AM|PM)/

def is_time?(s) = TIME_RE.match?(s)
```

The magic of regular expressions is that they provide a simple, declarative way of finding the string you are looking for. You can think of pattern matching as sort of regular expression facility, but for the rest of your data:

Pattern matching provides a simple declarative syntax for asking if *this* data structure looks like *that*.

Remarkably, pattern matching is built on just a few bits of new syntax. The first is the use of in to match a value. So if we had these local variables:

```
title = "Emma"
genre = :drama
published = 1816
```

We can use in to match them against specific values:

```
title in "Emma"               # True! The two strings are equal.
title in "Persuasion"         # False! The strings are not equal.
genre in :drama               # True! It's the same symbol.
published in 1816             # True again!
published in 1066             # Nope!
```

You can also use in as a stand-in for kind_of?:

```
title in String             # True
title in Integer            # Nope, the title is not an integer.
published in Integer        # True: 1816 is an integer.
published in Numeric        # True: Integer is a subclass of Numeric.
```

So far we haven't done anything that couldn't have done just as easily with == or kind_of?. But in exactly the same way that regular expressions only got interesting when we started writing things like (AM|PM) and .*, the pattern matching plot is about to thicken. To start with, we can ask if a value is one thing or another:

```
published in Integer | String         # true! It's an integer.
title in Integer | String             # true again! This time a string.
genre in Integer | String             # Nope, genre is a symbol!
genre in :comedy | :drama | :tragedy    # Yes, it is the second one!
```

You can also mix and match explicit values and classes in your alternatives. Thus we could write this:

```
genre in String | :comedy | :drama | :tragedy
```

And have the match succeed if genre is either a string or one of three specific symbols.

Since pattern matches always return a boolean, they work well as the conditions in if statements:

```
if genre in String | :comedy | :drama | :tragedy
  puts "The genre is #{genre}"
end
```

And wonderfully in case statements:

```
case genre
in :comedy | :drama | :tragedy
  puts "The genre is #{genre}"
in String
  puts "You need to read the chapter on symbols!"
in Integer | Float
  puts "A numeric genre makes no sense!"
else
  puts "Are you sure you understand this genre thing?"
end
```

Note that the in replaces the when in the case statement and how succinctly we can cover a whole range of alternatives with just a few lines of code.

Pattern Matching Arrays

The real power of pattern matching only kicks in when we realize that the pattern–the thing to the right of the in–can also describe an entire array. Suppose, for example, you were writing an application to process data about books, data that comes packaged in tidy little arrays:

patmatch/match_arrays_spec.rb

```
info = ["Emma", "Austen", :drama, 1816]
```

Your job is to write a method to do the processing:

```
def process_book(info)
  # Do something with the book info, if it is indeed book info.
end
```

Except that the source of info is incredibly unreliable. Thus process_book might get called with a clean, four element book array, or it might get called with nil or an integer or some other random value. A key part of your job is to reject anything that doesn't look like a book array.

Clearly you could write some straightforward Ruby to deal with the noise:

```
def valid_book?(info)
  return false unless info in Array
  return false unless info.length == 4
  return false unless info[0] in String
  return false unless info[1] in String
  return false unless info[2] in Symbol
  return false unless info[3] in Integer
  true
end

def process_book(info)
  if valid_book?(info)
```

```
    # Do something with the book info...
  end
end
```

The only halfway interesting thing about the valid_book? method is that we are using our new in expressions instead of the more familiar kind_of? method.

Happily, it turns out that we don't have to go through all of return…unless nonsense. Instead we can simply match the alleged array against a pattern:

```
# A better way...

def valid_book?(info)
  info in [String, String, Symbol, Integer]
end
```

While the thing to the right of the in in this example looks like an array, it's really a sort of higher level pattern, a pattern that will only match if info is a four element array that starts with two strings followed by a symbol and an integer.

Even better, it's trivial to make our pattern more picky. Perhaps the only valid genres are :drama, :comedy, and tragedy. No problem:

```
# A pickier pattern.

def valid_book?(info)
  info in [String, String, :drama | :comedy | :tragedy, Integer]
end
```

Or perhaps we are only interested in books published in 1816:

```
def published_in_1816?(info)
  info in [String, String, :drama | :comedy | :tragedy, 1816]
end
```

Keep in mind that when you write an array pattern you need to account for *all* of the elements or the pattern won't match. But you don't have to account for all of the elements explicitly. You can use an asterisk to mean *any number of any kind of items*. Thus if we only wanted to be sure that our info array began with a string, we could write:

```
info in [String, *]
```

Or perhaps we wanted to be sure that info starts with a string and ends with an integer:

```
info in [String, *, Integer]
```

Or simply harbors a symbol somewhere:

```
info in [*, Symbol, *]
```

Note that this last example is going to trigger a linear search through the array, which is fine for our small arrays but perhaps less so for very long arrays.