10

Regular Expressions

Reading

The text for this chapter has not been written yet. You can find all the information you need for this project in the lecture notes titled “Regular Expressions.”

There are no files to download this week.

Tutorial Project

Start an IRB session.

T1. Define a variable named s that will hold the first test string:

```ruby
>> s = "peter piper picked a peck of pickled peppers"
=> "peter piper picked a peck of pickled peppers"
```

T2. If we want to look for a specific sequence of letters, just use the index method. Type this to see if s contains the string “pip”:

```ruby
>> s.index("pip")
=> 6
```

The 6 means the string was found at location 6. Since Ruby uses 0 for the index of the first letter, that means the first “p” in “pip” was the 7th letter in s.

T3. For the first test with regular expressions, use the regular expression syntax to make a pattern that only matches these three letters:

```ruby
>> s.index(/pip/)
=> 6
```

It’s not much, but it shows that Ruby does understand regular expression syntax, i.e. it allowed us to enclose the letters inside slashes instead of quotes.

T4. Next try the operator Ruby uses as a short-cut. This is equivalent to the previous statement:

```ruby
>> s =~ /pip/
=> 6
```

Make sure you understand what that last expression asks Ruby to do: s (the item on the left side) is the name of a string. To the right of the =~ operator is a regular expression. When you ask Ruby to evaluate s =~ r it does a pattern matching operator to see if the pattern described by r can be found anywhere in s. If so, Ruby prints the location where the pattern was found, but if the pattern was not found the result is nil.
T5. Next let's experiment with wild cards. The regular expression `/p.c/` means “p followed by any character followed by c.” See if `s` contains this pattern:

```ruby
>> s = "p.c/
=> 12
```

Yes – the pattern starts at location 12.

T6. The pattern `/p.c/` occurs one other place in `s`. To verify that Ruby returned the first match we can do a couple of tests. First, ask for the string that is in locations 12 through 14:

```ruby
>> s[12..14]
=> "pic"
```

T7. Another way to check is to enclose the regular expression in brackets:

```ruby
>> s[/p.c/]
=> "pic"
```

Make sure you understand these last two examples. We've now seen three different things that Ruby lets us put between brackets following the name of a string:

- a single number, e.g. `s[5]`; Ruby evaluates this by returning the ASCII code of the character at location 5 in the string.
- a range of numbers, e.g. `s[6..8]`; in this case Ruby returns a string consisting of the characters at positions 6 through 8.
- a regular expression, e.g. `s[/p.c/]`. Here Ruby returns the first substring of `s` that matches the regular expression.

T8. Next let's experiment with character classes and repetition. The two-letter sequence `\w` means “any single word character.” Try this expression:

```ruby
>> s = "\w/
=> 0
```

Not too surprising – the 0 just means the first character in `s` is a letter or digit.

T9. Putting a plus sign after an item tells Ruby to match one or more of that item. So the pattern `\w+` means “one or more word characters in a row.” This expression asks Ruby to return the first word in `s`:

```ruby
>> s[\w+]
=> "peter"
```

T10. The `scan` method asks Ruby to make a list of all the places in `s` that match a pattern. To get a list of all the words in `s`:

```ruby
>> s.scan(/\w+/
=> ["peter", "piper", "picked", "a", "peck", "of", "pickled", "peppers"]
```

Ruby doesn't really know what a “word” is. Make sure you realize what we asked Ruby to do in that last example: find all the substrings that contained only “word characters”, which to Ruby means upper and lower case letters or digits. Since spaces are not word characters, the spaces in `s` broke up the pattern.

T11. What do you think would happen if we asked Ruby to use the pattern `\w+` to find the words in the sentence “I’m afraid of cows”? See if you can figure it out first, then do the experiment:

```ruby
>> t = "I’m afraid of cows."
=> "I’m afraid of cows."
>> t.scan(/\w+/)
=> ["I", "m", "afraid", "of", "cows"]
```
Because an apostrophe isn’t a word character it marked the end of the pattern, and Ruby started over again with the “m”.

T12. We can put numbers in braces following an item to tell Ruby to look for that number of items. For example, /\w{3}/ means “three word characters in a row.”

```ruby
>> s[/\w{3}/]
=> "pet"
```

Ruby found three letters at the beginning of *s*.

T13. To find strings of word characters between 3 and 10 letters long:

```ruby
>> s[/\w{3,10}/]
=> "peter"
```

Note that the complete first word was returned this time.

T14. Let’s repeat a previous experiment, where we use *scan* to find all the places where *s* matches the pattern, but this time we only want words between 3 and 10 letters long:

```ruby
>> s.scan(/\w{3,10}/)
=> ["peter", "piper", "picked", "peck", "pickled", "peppers"]
```

The next set of experiments show the need to “quote” certain characters.

T15. Define a new version of the string *t*:

```ruby
>> t = "Are you afraid of cows, too?"
=> "Are you afraid of cows, too?"
```

T16. The question mark is one of the symbols (like * or +) that has special meaning inside a regular expression. If you try to search for a question mark Ruby will print an error message saying it didn’t understand the regular expression:

```ruby
>> t = "/?/
SyntaxError: compile error
(irb):69: invalid regular expression...
```

T17. To see if a string contains a special symbol we need to “quote” it by putting a backslash in front of it:

```ruby
>> t = "/\?/
=> 27
```

T18. Can you think of a way to verify that previous expression did what we expected, without counting all the letters in *t*?

T19. One of the frustrating things about learning to write regular expressions is remembering which characters have a special meaning and which can be used without quoting. Try doing a search for a comma in the string *t*, with and without the quotes. What did you find out?

**Optional Project**

This next set of exercises introduces a subtle point about how regular expressions are evaluated by Ruby. You do not need to do these exercises unless you want to try some of the extra credit questions on the lab.

Recall one of the expressions shown earlier, which asked Ruby to look for a word between 3 and 10 letters long in the test string *s*:

```ruby
>> s[/\w{3,10}/]
=> "peter"
```
Here Ruby returned the first complete word in the sentence. But note carefully what we asked for: any sequence that consists entirely of word characters and this is between 3 and 10 characters long. Ruby could just as easily have returned "pet", "pete", or even "eter" as valid responses. So why did it return the full word?

The answer is that pattern matching in Ruby is greedy. What the regular expression algorithm does is find the first location in a string where the pattern can start, and then begin adding characters to the match. It does not stop as soon as the pattern is matched; instead it keeps on adding characters until no more characters can be added.

T20. This expression illustrates the fact that pattern matching uses a “greedy” algorithm. The pattern /p.*r/ means “a string that starts with a p and ends with an r:”

```ruby
>> s[/p.*r/]
=> "peter piper picked a peck of pickled pepper"
```

There are two things to notice about this result. First, since we asked for .* and not \w* Ruby included spaces in the result. Second, Ruby could have stopped at either of the first two r's it found – the result could have been "peter" or "peter piper" but instead Ruby continued to the last r in the string.

T21. We can tell Ruby we want it to return the shortest possible string that matches a pattern by putting a question mark in the pattern:

```ruby
>> s[/p.*?r/]  
=> "peter"
```

The question mark tells Ruby to make a lazy match. It returned the part of s that starts with a p, has as few characters as possible to match .*, and then the letter r.

T22. If we ask Ruby for all substrings that match /p.*r/ with the normal greedy match we’ll get just one long substring:

```ruby
>> s.scan(/p.*r/)
=> ["peter piper picked a peck of pickled pepper"]
```

T23. If we repeat that test, but this time use a lazy match, we’ll get all the substrings that start with a p and continue to the next r:

```ruby
>> s.scan(/p.*?r/)
=> ["peter", "piper", "picked a peck of pickled pepper"]
```