10 Pattern matching

10.1 Matching and substitution

So far, we have been looking at standard programming functions that just about any language includes. There is nothing very special in the way that Perl implements these. Perl's particular glory lies in *pattern matching*, and we turn to that now.

Pattern matching is about finding particular arrangements of characters within strings (and changing the strings in some way, or taking some other action, when the target patterns are found). Pattern matching in Perl uses the symbol `=~` to link the string being examined (the *target string*) to one of two matching actions, identified by the letters `m` (match) or `s` (substitute):

`m/.../`	return "true" if the pattern ... is present, "false" otherwise
`s/.../.../`	if the pattern ... to the left is found, change it to ... on the right

Pattern matching is about finding the pattern *within* the target string. The pattern usually will not comprise the whole of the target string (though we shall see that, if that is what we want, we can specify that.)

The simplest kind of pattern to look for is a particular substring (though the possibilities become far more sophisticated than that). Let's look at a couple of examples of that simple kind:

```
$a = "location";
if ($a =~ m/cat/)
  {
  print "Found a cat.\n";
  }

Found a cat.
```

```
$a = "location";
$a =~ s/cat/cut/;
print "$a\n";

locution
```

I shall call the material represented by dots in `m/.../` and to the left in `s/.../.../` the *pattern* section, and I shall call the material represented by the right-hand dots in `s/.../.../` the *replacement* section.

In the case of the `m/.../` construction, it is permissible to omit the `m`; rather than `if ($a =~ m/cat/)` it works just as well to write `if ($a =~ /cat/)`. This abbreviation saves so little typing that it seems pointless for the language to include it; however, because one can omit the `m`, Perl programmers almost always do omit it – so that readers who move on to other Perl textbooks could be confused if the abbreviation were not mentioned here.[12]

Also with the `m/.../` construction, alongside `=~` for "matches" there is also the symbol `!~` for "does not match":

```
$a = "location";
if ($a !~ /dog/)
  {
  print "No dog here.\n";
  }

No dog here.
```

If the `!~` symbol did not exist, it would of course be easy enough to achieve the same by writing:

```
if (not($a =~ /dog/))
```

but perhaps this abbreviation earns its keep a little better than the omission of `m`.

10.2 Character classes

Matching a specific substring gets us only so far in practice. Things get more interesting when our patterns refer to *classes* of characters, *repetitions* of pattern elements, and so forth.

To make the discussion concrete, let us suppose that (via the INFILE handle) we are reading the text file shown as Figure 1, which is a list of English counties and their populations (in thousands) at the year 1966. (We could of course have used newer data – but it happens that the 1960s data have features which will be helpful for illustrating certain aspects of Perl.)

Intentionally, I have set the file out rather messily – some lines have whitespace before the county name, and the whitespace within some county names and separating names from population figures varies from line to line, containing spaces, tab characters, or both. Text files in real life often are rather messy, and Perl is good for dealing with that kind of mess: messy input will give the pattern matcher good practice.

```
Bedfordshire  428
 Berkshire   585
Buckinghamshire    542
Cambridgeshire and  Isle of Ely    294
Cheshire     1472
Cornwall    353
Cumberland   296
Derbyshire   912
Devon        865
   Dorset    333
Durham  1541
Essex   1244
Gloucestershire    1054
Hampshire    1483
Herefordshire      140
Hertfordshire      872
Huntingdonshire and     Peterborough    184
Kent     1325
Lancashire   5189
Leicestershire 716
Lincolnshire (Holland)  105
Lincolnshire (Kesteven) 226
Lincolnshire (Lindsey) 453
Greater London     7914
Norfolk 586
Northamptonshire   428
Northumberland     828
Nottinghamshire    954
Oxfordshire  349
Rutland 28
Shropshire   322
Somerset     638
Staffordshire1802
East Suffolk 371
West Suffolk 148
Surrey  977
East Sussex  710
West Sussex  450
Warwickshire 2095
Westmorland  67
Isle of Wight      97
Wiltshire    471
Worcestershire     663
Yorkshire, East Riding  543
Yorkshire,   North Riding    584
Yorkshire, West Riding  3736
```

Figure 1

Let’s say that we want to extract a list of just those counties with populations of at least a million. Since the numbers shown are thousands, this means in practice: lines containing at least four consecutive digits.

One way to specify a class of characters is by listing them within square brackets. So we could achieve what we want, cumbersomely, by writing:

(9)

```
while ($line = <INFILE>)
  {
  chomp($line);
  if ($line =~ m/[0123456789][0123456789][0123456789][0123456789]/)
    {
    print "$line\n";
    }
  }
```

in which case the machine would respond:

```
Cheshire     1472
Durham  1541
Essex   1244
Gloucestershire   1054
Hampshire    1483
Kent    1325
Lancashire   5189
Greater London    7914
Staffordshire1802
Warwickshire 2095
Yorkshire, West Riding  3736
```

"I studied English for 16 years but... ...I finally learned to speak it in just six lessons"
Jane, Chinese architect
ENGLISH OUT THERE
Click to hear me talking before and after my unique course download

But Perl knows the character sequence; a cumbersome expression like `[0123456789]` can be abbreviated as `[0-9]` (and similarly, `[defghij]` could be given as `[d-j]`). So line 9.4 could be reduced to:

```
if ($line =~ /[0-9][0-9][0-9][0-9]/)
```

(remember that `m` for "matches" can be omitted).

Better still, since the class of digits from 0 to 9 is often useful, Perl provides a backslash code `\d` for that class: the line can be shortened further to:

```
if ($line =~ /\d\d\d\d/)
```

Other frequently-useful class codes are `\s` for whitespace characters (space, tab, newline), and `\w` for "word characters", meaning letters of the alphabet, digits, or the underline character (but not e.g. punctuation marks).[13] Capitals in class codes denote the complementary classes: `\D` is any character *other* than a digit, `\S` is any "black" (i.e. non-whitespace) character, `\W` any nonalphanumeric, non-underline character.

Furthermore, rather than repeating `\d` four times, we can specify the number of repetitions wanted within curly brackets:

```
if ($line =~ /\d{4}/)
```

(If we wanted to say "at least 4", so as to allow explicitly for populations of ten million or more, that would be `{4,}`; and `{4,6}` following a pattern element would mean "at least 4 and not more than 6". But this level of explicitness is not needed in the present example; if a string `$line` contains five consecutive digits it must certainly contain four consecutive digits, so it will match the pattern shown above.)

10.3 Complement classes and indefinite repetition

Some counties have multi-word names including spaces. Let's suppose that we want to pick out those lines. We might think of doing that by changing 9.4 so that instead of looking for a sequence of digits, it looks for a whitespace followed by an alphabetic character:

```
if ($line =~ /\s[A-Za-z]/)
```

(Perl offers no backslash code covering letters only; `\w` covers numbers too. `[A-Za-z]` is as concise as we can get for "any (upper or lower case) letter".)

One problem with this, though, is that the names for the divisions of Lincolnshire would be missed: in those names the internal space is followed by an opening bracket rather than a letter. It would be better to define the test as "space followed by any black character other than a digit". We can code this by using the notation `[^...]` meaning "any character *other than* ... ". Our line then becomes:

```
if ($line =~ /\s[^\s\d]/)
```

i.e. the pattern is "whitespace followed by a character that is neither whitespace nor a digit".

However, either of these versions assumes that no line has spaces at the beginning, before the name – any line written that way would get picked by this `if` statement, whether the county name were one word or more – in the present case it would wrongly pick Berkshire and Dorset. As we have seen, it is usually best not to make too many assumptions about where an input file includes whitespace and whether this consists of one or more space(s) and/or tab(s). What we are really looking for is any stretch of whitespace having *non-numeric black characters on both sides*. "One or more" is represented by the plus sign, so the following version of the `if` statement will do the trick:

```
if ($line =~ /[^\s\d]\s+[^\s\d]/)
```

With this line substituted for line 9.4, when we run (9) the output will be:

```
Cambridgeshire and  Isle of Ely    294
Huntingdonshire and     Peterborough    184
Lincolnshire (Holland)  105
Lincolnshire (Kesteven) 226
Lincolnshire (Lindsey) 453
Greater London     7914
East Suffolk 371
West Suffolk 148
East Sussex  710
West Sussex  450
Isle of Wight      97
Yorkshire, East Riding  543
Yorkshire,   North Riding    584
Yorkshire, West Riding  3736
```

Having said that it is safest to allow for messy use of whitespace in input files, perhaps we might actually want to locate lines with initial whitespace, in order to tidy the file up by deleting it. In a pattern, `^` outside square brackets represents the beginning (and `$` represents the end) of a string.[14] So, if `$line` is, say, the string "`   Dorset 333`" (beginning with three space characters), the following substitution statement:

```
$line =~ s/^\s+//;
```

will remove its leading spaces. (`//` as the replacement section changes the substring matching the pattern section to nothing, in other words it deletes that substring.)

If `$line` were a line having no leading whitespace, the statement above would ignore it: `+` means "one or more", so the pattern would not be found and no substitution would occur.

Alongside `+` for "one or more" we have the symbol `*` meaning "zero or more", so we could alternatively write the statement as:

```
$line =~ s/^\s*//;
```

This statement will successfully apply to any line whatever (every line begins with at least zero whitespace characters). But, if `\s*` does match zero whitespace characters, then nothing will be replaced by nothing, in other words there will be no actual change to strings that lack leading whitespace. Here, statements using `+` and `*` are interchangeable in practice; in other situations the distinction between these symbols is crucial.[15]

10.4 Capturing subpatterns

You might object here: if `+` (or `*`) can be any number from one (or zero) upwards, how do we know that `\s+` will match all three leading spaces in a string which contains them, rather than just one or two of them, or none of them if the pattern uses the asterisk? In fact either version of the statement will match all three leading spaces, because Perl pattern-matching is "greedy" – where alternative fits to a pattern are possible, it will always take the one which matches more rather than less of the string under consideration. However, it would clearly be safer to be explicit, and require a black character to follow the whitespace to be eliminated.

But that black character will be the first letter of the county name – we need to retain it in the replacement. Putting round brackets around part of the pattern "captures" whatever substring matches that subpattern so that we can refer back to it in the replacement section: `$1, $2`, etc. mean "whatever matched the first/second/... bracketed subpattern".

Thus, here is a snippet of code which will go through a file removing any leading or trailing whitespace from a line that contains some black characters. It uses the full-stop symbol “ `.` ”, meaning “any character whatever, black or white” – between the first and last black characters of the line there can be any number of intermediate characters, including letters, numbers, and whitespace:

(10)

```
1  while ($a = <INFILE>)
2    {
3    chomp($a);
4    $a =~ s/^\s*(\S.*\S)\s*$/$1/;
5    print "$a\n";
6    }
```

The pattern section of the substitution construction in 10.4 specifies beginning-of-line followed by any number of whitespaces followed by a black character, then any sequence of characters of any kind, followed by a black character and then zero or more whitespaces to the end of the line. The replacement section keeps the first and last black characters and whatever came between them. In 10.4 we must write `\s*` rather than `\s+`, because we don’t want the statement to apply only to lines that contain *both* leading and trailing whitespace: we want it to apply whenever there is whitespace at either end, or at both ends.

(In our county-population file, we know that the first black character of a line will always be a letter and the last will always be a digit, and we also know that there will always be several characters between the first and last black characters; so (10) would work just as well if the subpattern within brackets were written as `(\w.*\d)` or `(\w.+\d)`. But 10.4 is a more general way of pruning leading and trailing whitespace – it would work equally well for a file in which some lines have punctuation marks at beginning or end, or in which there are only two black characters with nothing between them.)

10.5 Alternatives

The substitution statement 10.4 should work successfully with the material of Figure 1, but it is not as general as it might be. It will ignore a line consisting *only* of whitespace, since there will be no characters for the `\S` symbols to match. Furthermore, it will also ignore a line with just one black character surrounded by whitespace on either side. Each `\S` in the sequence `\S.*\S` must match a separate black character, so the line must contain at least two. In the present context where we are dealing with lists of counties and their populations, although wholly-blank lines might occur, we should be safe in assuming that there will never be a line with just one black character. But suppose that we were dealing with some other kind of input file in which this latter kind of line did occur, and we wanted it too to have leading and trailing whitespace removed. To achieve that, we could modify (10.4) using the symbol “ `|` ” to represent alternatives:

```
$a =~ s/^\s*(\S.*\S|\S)\s*$/$1/;
```

When we use `|` for “or” in a pattern, normally (as here) we will need round brackets around the alternatives, to show where they begin and end within the pattern. In this case we need the brackets anyway, so that we can use `$1` to refer to the material from first to last black character (or to the sole black character in a line that has only one). But sometimes round brackets will be needed within a pattern just to delimit alternatives – the symbol `$1` (or `$2`, or whatever `$`... symbol corresponds to that bracket-pair) will never be used.

10.6 Escaping special characters

In codes like `\s` (any whitespace character) or `\t` (tab character), the backslash indicates that the following character (`s` or `t`) is not to be given its literal meaning. But where a character such as asterisk or full stop has special pattern-matching significance, a backslash is needed before it when it *should* be given its literal meaning within a pattern. Each of the following characters needs a backslash if it appears with its literal meaning in a pattern:

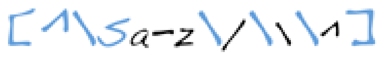
```
\ | ( ) [ { ^ $ * + ? .
```

– and the hyphen needs a backslash if it occurs in its literal meaning between square brackets. However, if you cannot remember precisely which punctuation-type characters require a backslash, it does no harm to put one in anyway – it is only alphanumeric characters which have non-literal meanings when preceded by a backslash in a pattern.

So for instance the class of “all characters other than whitespace, lower-case letters, forward slash, backslash, or circumflex” could be represented as:

```
[^\sa-z\/\\\^]
```

It is logical, but it takes a bit of care to work out precisely what a jumble like that is saying. (When I composed this expression I could not remember whether or not the forward slash needs a backslash in order to be interpreted literally; in fact it does not, but I put one in anyway.) Real-life Perl applications often do tend to need just such complicated patterns involving many non-alphanumeric characters. One tip, when working the pattern out on paper before typing it in, is to write the characters which are used with special pattern-matching meanings larger, and/or in a different colour, than the characters representing themselves:



This helps the eye to grasp the logic of what is going on.

10.7 Greed versus anorexia

We saw that Perl pattern-matching is normally “greedy”:

```
$a = "Hertfordshire";
$a =~ /(e.*r)(.*)$/;
print($1, "\n", $2, "\n");

ertfordshir
e
```

We can make the pattern-matching quantifiers `*`, `+`, etc. "anorexic" by suffixing `?` to them:

```
$a = "Hertfordshire";
$a =~ /(e.*?r)(.*)$/;
print($1, "\n", $2, "\n");

er
tfordshire
```

This works even with the quantifier `?` (meaning zero or one) itself:

```
$b = "Essex";
$b =~ /E(s?)(.*)x/;
print "$2\n";

se


$b = "Essex";
$b =~ /E(s??)(.*)x/;
print "$2\n";

sse
```

10.8 Pattern-internal back-reference

We have seen that the symbols `$1, $2`, etc. can be used to refer back to pattern elements demarcated by round brackets, when the `$...` symbols occur outside the pattern – in the replacement part of a `s/.../.../` structure, or in a later code line. Sometimes we need to refer back to a pattern element from a later point *within that pattern*. For that, the symbols are `\1, \2`, ... rather than `$1, $2`, ...

Here, for instance, is an example of pattern matching which finds, within a string containing names of French towns separated by whitespace, a town name which contains the same pair of adjacent vowels at two separate points in the name:

(11)

```
1  $towns = "Paris Poitiers Bordeaux Toulouse Lyons Marseilles";
2  if ($towns =~ /(\s|^)(\S*([aeiou][aeiou])\S*\3\S*)(\s|$)/)
3    {
4    print "$2\n$3\n";
5    }
```

```
Toulouse
ou
```

The second word in the string, "Poitiers", contains two pairs of adjacent vowels, but different pairs; the use of `\3` in the pattern-match means that only the fourth word, "Toulouse", with the same vowel-pair "ou" in two places, fits the pattern.

The pattern in 11.2 takes a bit of unpicking.

First, since the `$towns` string uses whitespace to separate names but does not have whitespace at beginning or end, in order to ensure that every name including the first and the last is tested we need to specify that the substrings examined are preceded either by whitespace or beginning-of-line, i.e. `(\s|^)`, and followed either by whitespace or end-of-line, i.e. `(\s|$)`. These are cases of round brackets which are included in the pattern purely to delimit alternatives: they are respectively the first and fourth pair of round brackets in 11.2, but no use is made of the codes `$1` and `$4`.

Between these two pairs of round brackets is the sequence:

```
(\S*([aeiou][aeiou])\S*\3\S*)
```

This looks for a continuous sequence of black characters, containing a pair of vowels at some point and the same pair of vowels at a later point. Notice that this segment of line 11.2 has one pair of round brackets nested inside another pair. The numbering reflects the sequential order of the *opening* brackets; so the outer brackets surrounding the entire segment are pair 2 (pair 1 was `(\s|^)`, remember), and the internal brackets round `[aeiou][aeiou]` are pair 3. Hence, *within* the pattern, `\3` refers back to whatever `[aeiou][aeiou]` matched (and later, outside the pattern, in line 11.4 `$2` and `$3` refer respectively to the whole word, and to that vowel-pair).

10.9 Transliteration

Finally, apart from the structures `m/.../` and `s/.../.../`, there is a third structure, `tr/.../.../` ("tr" for "transliterate"), which can appear after `=~` and which is commonly discussed in Perl textbooks under the heading "pattern matching". This is misleading, though: what appears between the left-hand pair of slashes in a `tr/.../.../` construction is not a pattern, the whole of which is looked for in the target string, but a sequence of individual characters, any one of which, if found, is replaced by the corresponding character from the right-hand pair of slashes. Thus `tr/.../.../` could be used, for instance, to capitalize each vowel in a string:

```
$a = "triceratops";
$a =~ tr/aeiou/AEIOU/;
print "$a\n";
```

```
trIcErAtOps
```

The only real reason to deal with `tr/.../.../` in this chapter rather than somewhere else is that, like `m/.../` and `s/.../.../`, it uses the symbol `=~` as the link to its target string.

Appendix: A checklist of pattern-matching symbols

[acgt]	any of the characters a, c, g, t		
[^acgt]	any character other than a, c, g, t		
[a-t]	any of the twenty characters between a and t inclusive		
.	any character		
\d	digit	\D	nondigit
\s	whitespace character	\S	"black" character
\w	"word character"	\W	non-word character
^	beginning of target string	$	end of target string
X*	any number (zero or more) X's		
X+	one or more X's		
X?	zero or one X		
X*?, X+?, X??	as above but matching as little as possible		
X{3}	sequence of three X's		
X{3,}	sequence of three or more X's		
X{3,5}	sequence of at least three and at most five X's		
(X|Y)	either X or Y		
\1, \2, ...	(within a pattern) whatever matched the contents of the 1st/2nd/... pair of round brackets in this pattern		
$1, $2, ...	(outside a pattern) whatever matched the contents of the 1st/2nd/... pair of round brackets in the last pattern matched		