14 Scalar versus list context

Often, we shall need to work through an array, in order to process all the entries in some way or to look for entries having some feature of interest. Let's return to the idea of identifying counties with populations over a million, and let's now make it a task to be done in code that follows on from code-chunk (13). In chapter 10 we used pattern matching to count sequences of digits in lines directly they were read in from the external data file. But (13) set us up with arrays containing the population information, which remain available for consultation as long as our program is running – we don't need to go back to the external file. If we knew how many rows there were in the `@countyPops` array, one way to list the million-plus counties would use a `for` loop:

(16)

```
1   for ($j = 0; $j < N_rows; ++$j)
2     {
3     if ($countyPops[$j] >= 1000000)
4       {
5       print ($countyNames[$j], "\n");
6       }
7     }
```

```
Cheshire
Durham
Essex
   ⋮
```

Here I am temporarily writing *N_rows* in italics to stand for whatever number of rows the array contains. The initial row is row 0, so the number of the final row will be one less than *N_rows*; therefore this `for` line will run through all the rows of `@countyPops` and stop after the final row, as we want. So what Perl expression should we write in place of *N_rows*?

In fact (see Figure 1) there will be 46 rows, from row 0 for Bedfordshire to row 45 for the West Riding of Yorkshire. But we don't want to hard-wire the number 46 into our program, otherwise the program will break as soon as we vary the data-set – say, by giving it Scottish rather than English counties. (One wise software-engineering expert urges that the *only* numbers which should ever appear as specific numbers in program code are zero and one.[22] That is possibly an extreme point of view, but certainly it would be foolish to incorporate the number 46 into our code explicitly in a case like this one.) What we want in place of *N_rows* is an expression that means "the number of rows in `@countyPops`, *whatever that number happens to be*".

We can achieve this very simply: the expression `@countyPops` itself is all that is needed. Line 16.1 can read:

```
for ($j = 0; $j < @countyPops; ++$j)
```

How can this be? `$j` always represents an integer (0, 1, 2, and so on); how can an individual integer be "less than" (or more than, or equal to) a whole array of numbers?

Here we come to one of the most distinctive features of Perl: the fact that a given term can have different meanings in different contexts. The position following a `<` symbol is a *scalar context*: if we write `$j < ...` , then whatever expression we put in place of the dots has to provide a scalar value. An array is not a scalar, but an array name in a scalar context will be understood by Perl to mean "the number of elements in the array" – and this of course *is* a scalar value.

The alternative to scalar context is *list context*. For instance, `print ...` is a list context, so whatever we put in place of the dots will be interpreted by Perl as a list. If we write, say:

```
print 100;
```

we seem to be asking Perl to print an individual (scalar) value, but Perl will not interpret the request that way; Perl will understand us as asking it to print a list that happens to be one element long.

In this latter case the difference is perhaps academic. So long as Perl actually prints out `100` as expected, why should we care whether it "thinks" it is printing out a list of one element or an individual item? Going in the other direction, though, it can be tricky to know how Perl will interpret a non-scalar item that it encounters in a scalar context. You just have to learn that, for instance, what an array gives in a scalar context is its size – you could not reliably predict this from more general features of the Perl language.[23]

Consequently, at the elementary level we are concerned with in this textbook, we shall not make much use of the ability to change the usual meanings of expressions by putting them into unusual contexts. But this is one of those features that is used so heavily, by people who are proficient in Perl, that beginners might be confused if we did not even mention it here. And the specific example of putting an array name into scalar context to find its size is so useful that even beginners need to know about it. If we want a scalar variable `$N_counties` to contain the number of counties in the array `@countyPops`, we might have expected that Perl would have a built-in function with a name such as `size()` that would take an array as argument and return the number of elements it contains. There is in fact no function `size()`; the nearest equivalent is the function `scalar()`, which gives whatever value its argument gives in a scalar context – so that, if the argument is an array, `scalar()` gives its size. We can assign `$N_counties` a value by writing:

```
$N_counties = scalar(@countyPops);
```

But for that matter, we can achieve the same result by writing simply:

```
$N_counties = @countyPops;
```

– assignment to a scalar variable is a "scalar context", so `@countyPops` will yield its size.

The point about how Perl interprets an array in a scalar context arose because we wanted to find some way of specifying the number of elements in the `@countyPops` array, in order to allow a `for` loop to work through the array and stop when it reached the end. This made a convenient introduction to the issue of finding the size of an array, which is something we often need to do. But, before moving on, we should note that there is a more direct method of working through the elements of an array, which bypasses the array-size issue because it dispenses with the need for a named index variable such as `$j`. Rather than using `for`, to work through each of the entries in an array we can use a special-purpose keyword `foreach`, which is relevant only to arrays.

Suppose we want to calculate the total national population, by adding together the populations of each of the counties (again, by adding code to the program which began with code-chunk (13)). We can do it like this:

(17)

```
1  $totalPop = 0;
2  foreach $countyPop (@countyPops)
3    {
4    $totalPop += $countyPop;
5    }
6  print "$totalPop\n";
```

```
45373000
```

The `foreach` line looks at each row of `@countyPops` in turn, assigning its value to the variable `$countyPop`; successive values of that variable are added to `$totalPop`, which begins as zero and ends containing the overall cumulative total.