# 4 Operators

## 4.1 Number and string operators

In program (1) we saw the operator `+`, which as you would expect takes a pair of numerical values and gives their sum. Likewise `-` is used as a minus sign. Some further operators (not a complete list, but the ones you are most likely to need) include:

| | |
|---|---|
| `*` | multiplication |
| `/` | division |
| `**` | exponentiation: `2 ** 3` means $2^3$, i.e. eight |

These operators apply to numerical values, but others apply to character-strings. Notably, the full stop `.` represents *concatenation* (making one string out of two):

```
$p = "witch";
$q = "craft";
$r = $p . $q;
print $r;
```

```
witchcraft
```

(Beware of possible confusion here. Some programming languages make the plus sign do double duty, to represent concatenation of strings as well as addition of numbers, but in Perl the plus sign is used only for numerical values.)

Another string operator is `x` (the letter x), which is used to concatenate a string with itself a given number of times: `"a" x 6` is equivalent to `"aaaaaa"`, `"pom" x 3` is equivalent to `"pompompom"`. (And `"pom" x 0` would yield the *empty string* – the length-zero string containing no characters – which is more straightforwardly specified as `""`.)

Note, by the way, that for Perl a single character is just a string of length one – there is no difference, as there is for instance in C, between `"a"` and `'a'`, these are equivalent ways of representing the length-one string containing just the character `a`. However, single and double quotation marks are not always equivalent. Perl uses backslash as an *escape* character to create codes for string elements which would be awkward to type: for instance, `\n` represents a newline character, and `\t` a tab. Between double quotation marks these sequences are interpreted as codes:

```
print "witch\ncraft";
```

```
witch
craft
```

but between single quotation marks they are taken literally:

```
print 'witch\ncraft';
```

```
witch\ncraft
```

In practice this means that you will almost always want to use double rather than single quotation marks. If you do want to include a backslash character within a string defined within double quotation marks, you code it as `\\`; and likewise `\"` and `\'` code quotation marks that are part of a string. When you display a line you will commonly want to end it with a newline, so that it doesn't run into whatever is displayed next. Thus:

```
print "Don\'t say \"never\".\n";
```

```
Don't say "never".
```

There are *rules of precedence* among the various operator symbols. Thus, the sequence `2 + 3 * 4` will yield the result 14 (not 20), because `*` has higher precedence than `+`. Here the relative precedence probably seems obvious, because it is the same in school algebra: multiplications are done before additions, not the other way round. But it is not always so easy to predict the precedence. Are you confident that you know whether `12 / 3 * 2` would give eight or two? Rather than learning all the precedence rules by heart, it is much easier to avoid the issue by using brackets: `(12 / 3) * 2` is eight, `12 / (3 * 2)` is two.

A detailed Perl manual will give the full rules of precedence, together with a number of less-used operators not covered here. But many successful Perl programmers are hazy about a few of the more arcane operators – and I wonder whether *anyone* is confident about every detail of the precedence rules. Brackets are easier.

Incidentally, although the main purpose of an assignment statement, such as `$a = 0`, is to give the symbol on the left a value, Perl regards the entire statement as an expression with a value (its value is the value assigned by the equals sign). This means that if we want to initialize various variables with the same value, we don't need to write separate assignment statements

```
$a = 0;
$b = 0;
$c = 0;
```

– it is enough to write `$a = $b = $c = 0`. An expression like this is interpreted as if it were written `$a = ($b = ($c = 0)))`: `$c` is straightforwardly assigned the value zero, then `$b` is assigned the value `($c = 0)`, which is itself zero – and `$a` is assigned the value `($b = 0)`, which is again zero.

## 4.2 Combining operator and assignment

One thing that a programmer very often needs to do is to change the value of a variable by applying some arithmetic operation to its current value – say, adding the value of another variable:

```
$a = $a + $b;
```

Because this is such a frequent thing, it can be abbreviated by combining the operator and the assignment symbol:

```
$a += $b;
```

and likewise `$a -= $b` means "reduce the value of `$a` by that of `$b`", and so forth:

```
$a = 21;
$b = 3;
$a /= $b;
print $a;
```

```
7
```

Very often, the arithmetic operation consists of either adding or subtracting one; these operations can be further abbreviated to `++` and `--`:

```
$a = 20;
++ $a;
print $a;
```

```
21
```

(There is a subtle difference between `++ $a` and `$a ++`, in terms of when the addition happens. A beginner is recommended always to put `++` or `--` *before* the variable to which it applies, in which case the addition or subtraction is carried out before the variable is used in any further operations.)

## 4.3 Truth-value operators

The operators seen so far give either a number or a string as their result. There are also operators which yield the answers “true” or “false”. To see how these work, consider that very often we want a program to branch: if so-and-so then do *this*, otherwise do *that* (or, do nothing). Branching is handled by a construction like this:

```
if ($a > 100)
  {
  print "It\'s big.\n";
  }
⋮
```

When the program reaches this section of code, it checks whether the current value of `$a` is over 100; if so, the code between curly brackets is executed, i.e. the message is printed out, otherwise that block of code is ignored; and in either case the program then moves on to whatever statements follow after the closing curly bracket. Obviously `>` means “is greater than”, so it yields either the value “true” or the value “false”: `6 > 5` gives “true”, `6 > 6` or `6 > 7` give “false”.[5]

The meaning of `>` is straightforward, and likewise `<` means “is less than”, `>=` and `<=` mean “is greater/less than or equal to”, and `!=` means “is not equal to”. The big stumbling block, which often leads experienced programmers into careless mistakes, comes from the fact that, most often, one wants to ask whether some value “is equal to” another. The Perl for “is equal to” is `==` (*two* equals signs).

It is all too easy to write something like:

```
if ($a = 100)
  {
  ⋮
  }
```

thinking that you are testing whether `$a` is equal to 100. You aren’t. A single equals sign is the assignment symbol: it means “*make* the thing on my left be equal to the thing on my right”. So a computer encountering `if ($a = 100)` will first change whatever value `$a` previously had to the value 100, and then decide what to do with the `if` by considering the “truth value” of 100. A number does not really have a truth-value, of course, but for reasons that we can skip over here Perl will treat the number 100 as “true”; so it will do whatever is given within the curly brackets. Try it:

```
$a = 2;
if ($a = 100)
  {
  print "It\'s one hundred.\n";
  }

It's one hundred.
```

or even

```
$a = "pomegranate";
if ($a = 100)
  {
  print "It\'s one hundred.\n";
  }

It's one hundred.
```

I have spelled this out at length, because the mistake is so easy to make. To *test for equality* between numerical values you need *two* equals signs. A single equals sign does not test anything, it *assigns* a value.

A further complication is that `==`, `!=`, `>`, and so forth can only be used to compare *numerical* values. Often, one wants to check whether two *strings* are the same or different. For strings, “is equal to” is symbolized as `eq` (and “is not equal to” is `ne`). Thus:

(2)

```
1  $a = "pomegranate";
2  if ($a eq "Pomegranate")
3    {
4    print "They\'re the same.\n";
5    }
6  if ($a ne "Pomegranate")
7    {
8    print "They\'re different.\n";
9    }

   They're different.
```

(Lower case versus capital makes the strings unequal.) You must not use `==` or `!=` in 2.2 or 2.6 – if you do, Perl will apply the wrong test (and `perl -w` will issue a warning).

The keywords `eq` and `ne` are the string-comparison counterparts of the number-comparison operators `==` and `!=`.[6] There are also string-comparison counterparts to `<`, `>=`, etc. For instance, when comparing strings, `lt` means "is less than" and `ge` means "is greater than or equal to". What "less" and "greater" refer to in this case is the sorting sequence for strings; for instance, the string `band` is "greater than" `ban` but "less than" `bang`. But this is a specialized kind of string comparison, which many programmers never need to use. For most purposes, `eq` and `ne` are the only string-comparison operators needed.

The symbols `and`, `or`, and `not` apply to expressions which have truth-values to give further truth-values. X `and` Y is true if both X and Y are true, and false if either or both is false. X `or` Y is true if either one of X and Y is true. The expression `not` X is true if X is false, and false if X is true. So, for instance,

```
   (3 > 2) and not (4 < 5)
```

gives "false". The expression to the left of `and` is true; but `4 < 5` is true, so the expression to the right of `and`, namely "`not (4 < 5)`", is false. "True `and` false" gives false.[7]

We saw above that Perl includes some shortcuts, such as `*=` or `++`, which achieve conciseness by merging operation and assignment symbolically. There is also one construction which does something similar with a truth-value operator: the three-place `?:` construction. Instead of (2), we could have written:

(3)

```
1  $a = "pomegranate";
2  $b = "They\'re the same.\n";
3  $c = "They\'re different.\n";
4  $d = ($a eq "Pomegranate" ? $b : $c);
5  print $d;

   They're different.
```

What the structure `X ? Y : Z` does is to say “Is X true or false? If it is true, then the value of the whole construction is Y; if X is false, then the value of the whole construction is Z.” In this case, the value of `$a eq "Pomegranate"` is “false” (because `$a` begins with lower-case `p`); so `$d` is assigned the value of `$c` rather than that of `$b`, and hence `$c` is what is printed out.

In this toy example, the code using `?:` is not much shorter than the code it replaced. But in realistic programming situations, `?:` can often be very handy.