Learn Perl in about 2 hours 30 minutes

By Sam Hughes

Perl is a dynamic, dynamically-typed, high-level, scripting (interpreted) language most comparable with PHP and Python. Perl's syntax owes a lot to ancient shell scripting tools, and it is famed for its overuse of confusing symbols, the majority of which are impossible to Google for. Perl's shell scripting heritage makes it great for writing *glue code*: scripts which link together other scripts and programs. Perl is ideally suited for processing text data and producing more text data. Perl is widespread, popular, highly portable and well-supported. Perl was designed with the philosophy "There's More Than One Way To Do It" (TMTOWTDI) (contrast with Python, where "there should be one - and preferably only one - obvious way to do it").

Perl has horrors, but it also has some great redeeming features. In this respect it is like every other programming language ever created.

This document is intended to be informative, not evangelical. It is aimed at people who, like me:

- dislike the official Perl documentation at http://perl.org/ for being intensely technical and giving far too much space to very unusual edge cases
- learn new programming languages most quickly by "axiom and example"
- wish Larry Wall would get to the point
- already know how to program in general terms
- don't care about Perl beyond what's necessary to get the job done.

This document is intended to be as short as possible, but no shorter.

Preliminary notes

- The following can be said of almost every declarative statement in this document: "that's not, strictly speaking, true; the situation is actually a lot more complicated". I've deliberately omitted or neglected to bother to research the "full truth" of the matter for the same reason that there's no point in starting off a Year 7 physics student with the Einstein field equations. If you see a serious lie, point it out, but I reserve the right to preserve certain critical lies-to-children.

- Throughout this document I'm using example `print` statements to output data but not explicitly appending line breaks. This is done to prevent me from going crazy and to give greater attention to the actual string being printed in each case, which is invariably more important. In many examples, this results in alotofwordsallsmusheduptogetherononeline if the code is run in reality. Try to ignore this. Or, in your head or in practice, set `$\` (also known as `$OUTPUT_RECORD_SEPARATOR`) to `"\n"`, which adds the line breaks automatically. Or substitute the `say` function.

- Perl docs all have short, memorable names, such as perlsyn which explains Perl syntax, perlop (operators/precedence), perlfunc (built-in functions) et cetera. **perlvar is the most important of these**, because this is where you can look up un-Googlable variable names like `$_`, `$"` and `$|`.

Hello world

A Perl *script* is a text file with the extension `.pl`.

Here's the text of `helloworld.pl`:

```
use strict;
use warnings;

print "Hello world";
```

Perl has no explicit compilation step (there *is* a "compilation" step, but it is performed automatically before execution and no compiled binary is generated). Perl scripts are interpreted by the Perl interpreter, `perl` or `perl.exe`:

```
perl helloworld.pl [arg0 [arg1 [arg2 ...]]]
```

A few immediate notes. Perl's syntax is highly permissive and it will allow you to do things which result in ambiguous-looking statements with unpredictable behaviour. There's no point in me explaining what these behaviours are, because you want to avoid them. The way to avoid them is to put `use strict; use warnings;` at the very top of every Perl script or module that you create. Statements of the form `use` *<whatever>* are *pragmas*. A pragma is a signal to the Perl compiler, and changes the way in which the initial syntactic validation is performed. These lines take effect at compile time, and have no effect when the interpreter encounters them at run time.

The hash symbol `#` begins a comment. A comment lasts until the end of the line. Perl has no block comment syntax.

Variables

Perl variables come in three types: *scalars*, *arrays* and *hashes*. Each type has its own *sigil*: `$`, `@` and `%` respectively. Variables are declared using `my`.

Scalar variables

A scalar variable can contain:

- `undef` (corresponds to `None` in Python, `null` in PHP)
- a number (Perl does not distinguish between an integer and a float)
- a string
- a reference to any other variable.

```
my $undef = undef;
print $undef; # error

# implicit undef:
my $undef2;
print $undef2; # exactly the same error

my $num = 4040.5;
print $num; # "4040.5"

my $string = "world";
print $string; # "world"
```

(References are coming up shortly.)

String concatenation using the `.` operator (same as PHP):

```
print "Hello ".$string; # "Hello world"
```

String concatenation by passing multiple arguments to `print`:

```
print "Hello ", $string; # "Hello world"
```

It is impossible to determine whether a scalar contains a "number" or a "string". More precisely, it is irrelevant. Perl is weakly typed in this respect. Whether a scalar behaves like a number or a string depends on the operator with which it is used. When used as a string, a scalar will behave like a string. When used as a number, a scalar will behave like a number (or raise a warning if this isn't possible):

```
my $str1 = "4G";
my $str2 = "4H";

print $str1 .  $str2; # "4G4H"
print $str1 +  $str2; # "8" with two warnings
print $str1 eq $str2; # "" (empty string, i.e. false)
print $str1 == $str2; # "1" with NO WARNING!
```

The lesson is to always using the correct operator in the correct situation. There are separate operators for comparing scalars as numbers and comparing scalars as strings:

```
# Numerical operators:  <,  >, <=, >=, ==, !=, <=>
# String operators:    lt, gt, le, ge, eq, ne, cmp
```

Perl has no boolean data type. A scalar in an `if` statement evaluates to boolean "false" if and only if it is one of the following:

- `undef`
- number `0`
- string `""`
- string `"0"`.

The Perl documentation *repeatedly* claims that functions return "true" or "false" values in certain situations. In practice, when a function is claimed to return "true" it usually returns `1`, and when it is claimed to return false it usually returns the empty string, `""`.

Array variables

An array variable is a list of scalars indexed by integers beginning at 0. In Python this is known as a *list*, and in PHP this is known as an *array*.

```
my @array = (
        "print",
        "these",
        "strings",
        "out",
        "for",
        "me", # trailing comma is okay
);
```

You have to use a dollar sign to access a value from an array, because the value being *retrieved* is not an array but a scalar:

```
print $array[0]; # "print"
print $array[1]; # "these"
print $array[2]; # "strings"
print $array[3]; # "out"
print $array[4]; # "for"
print $array[5]; # "me"
print $array[6]; # warning
```

You can use negative indices to retrieve entries starting from the end and working backwards:

```
print $array[-1]; # "me"
print $array[-2]; # "for"
print $array[-3]; # "out"
print $array[-4]; # "strings"
print $array[-5]; # "these"
print $array[-6]; # "print"
print $array[-7]; # warning
```

There is no collision between a scalar `$array` and an array `@array` containing a scalar entry `$array[0]`. There may, however, be reader confusion, so avoid this.

To get an array's length:

```
print "This array has ", (scalar @array), "elements"; # "This array has 6 elements"
print "The last populated index is ", $#array;        # "The last populated index is 
5"
```

String concatenation using the `.` operator:

```
print $array[0].$array[1].$array[2]; # "printthesestrings"
```

String concatenation by passing multiple arguments to `print`:

```
print @array; # "printthesestringsoutforme"
```

The arguments with which the original Perl script was invoked are stored in the built-in array variable `@ARGV`.

Variables can be interpolated into strings:

```
print "Hello $string"; # "Hello world"
print "@array";        # "print these strings out for me"
```

Caution. One day you will put somebody's email address inside a string, `"jeff@gmail.com"`. This will cause Perl to look for an array variable called `@gmail` to interpolate into the string, and not find it, resulting in a runtime error. Interpolation can be prevented in two ways: by backslash-escaping the sigil, or by using single quotes instead of double quotes.

```
print "Hello \$string"; # "Hello $string"
print 'Hello $string';  # "Hello $string"
print "\@array";        # "@array"
print '@array';         # "@array"
```

Hash variables

A hash variable is a list of scalars indexed by strings. In Python this is known as a *dictionary*, and in PHP it is known as an *array*.

```
my %scientists = (
        "Newton"   => "Isaac",
        "Einstein" => "Albert",
        "Darwin"   => "Charles",
);
```

Notice how similar this declaration is to an array declaration. In fact, the double arrow symbol `=>` is called a "fat comma", because it is just a synonym for the comma separator. A hash is merely a list with an even number of elements, where the even-numbered elements (0, 2, ...) are all considered as strings.

Once again, you have to use a dollar sign to access a value from a hash, because the value being *retrieved* is not a hash but a scalar:

```
print $scientists{"Newton"};   # "Isaac"
print $scientists{"Einstein"}; # "Albert"
print $scientists{"Darwin"};   # "Charles"
print $scientists{"Dyson"};    # runtime error - key not set
```

Note the braces used here. Again, there is no collision between a scalar `$hash` and a hash `%hash` containing a scalar entry `$hash{"foo"}`.

You can convert a hash straight to an array with twice as many entries, alternating between key and value (and the reverse is equally easy):

```
my @scientists = %scientists;
```

However, unlike an array, the keys of a hash have *no underlying order*. They will be returned in whatever order is more efficient. So, notice the rearranged *order* but preserved *pairs* in the resulting array:

```
print @scientists; # something like "EinsteinAlbertDarwinCharlesNewtonIsaac"
```

To recap, you have to use **square brackets** to retrieve a value from an array, but you have to use **braces** to retrieve a value from a hash. The square brackets are effectively a numerical operator and the braces are effectively a string operator. The fact that the *index* supplied is a number or a string is of absolutely no significance:

```
my $data = "orange";
my @data = ("purple");
my %data = ( "0" => "blue");

print $data;      # "orange"
print $data[0];   # "purple"
print $data["0"]; # "purple"
print $data{0};   # "blue"
print $data{"0"}; # "blue"
```

Lists

A *list* in Perl is a different thing again from either an array or a hash. You've just seen several lists:

```
(
        "print",
        "these",
        "strings",
        "out",
        "for",
        "me",
)

(
        "Newton"   => "Isaac",
        "Einstein" => "Albert",
        "Darwin"   => "Charles",
)
```

A list is not a variable. A list is an ephemeral *value* which can be *assigned* to an array or a hash variable. This is why the syntax for declaring array and hash variables is identical. There are many situations where the terms "list" and "array" can be used interchangeably, but there are equally many where lists and arrays display subtly different and extremely confusing behaviour.

Okay. Remember that `=>` is just `,` in disguise and then look at this example:

```
(0, 1, 2, 3, 4, 5)
(0 => 1, 2 => 3, 4 => 5)
```

The use of `=>` hints that one of these lists is an array declaration and the other is a hash declaration. But on their own, neither of them are declarations of anything. They are just lists. *Identical* lists. Also:

```
()
```

There aren't even hints here. This list could be used to declare an empty array or an empty hash and the `perl` interpreter clearly has no way of telling either way. Once you understand this odd aspect of Perl, you will also understand why the following fact must be true: **List values cannot**

be nested. Try it:

```
my @array = (
        "apples",
        "bananas",
        (
                "inner",
                "list",
                "several",
                "entries",
        ),
        "cherries",
);
```

Perl has no way of knowing whether `("inner", "list", "several", "entries")` is supposed to be an inner array or an inner hash. Therefore, Perl assumes that it is neither and **flattens the list out into a single long list**:

```
print $array[0]; # "apples"
print $array[1]; # "bananas"
print $array[2]; # "inner"
print $array[3]; # "list"
print $array[4]; # "several"
print $array[5]; # "entries"
print $array[6]; # "cherries"

print $array[2][0]; # error
print $array[2][1]; # error
print $array[2][2]; # error
print $array[2][3]; # error
```

The same is true whether the fat comma is used or not:

```
my %hash = (
        "beer" => "good",
        "bananas" => (
                "green"  => "wait",
                "yellow" => "eat",
        ),
);

# The above raises a warning because the hash was declared using a 7-element list

print $hash{"beer"};    # "good"
print $hash{"bananas"}; # "green"
print $hash{"wait"};    # "yellow";
print $hash{"eat"};     # undef, so raises a warning

print $hash{"bananas"}{"green"};  # error
print $hash{"bananas"}{"yellow"}; # error
```

More on this shortly.

Context

Perl's most distinctive feature is that its code is *context-sensitive*. **Every expression in Perl is evaluated either in scalar context or list context**, depending on whether it is expected to produce a scalar or a list. Many Perl expressions and built-in functions display radically different behaviour depending on the context in which they are evaluated.

A scalar declaration such as `my $scalar =` evaluates its expression in scalar context. A scalar value such as `"Mendeleev"` evaluated in scalar context returns the scalar:

```
my $scalar = "Mendeleev";
```

An array or hash declaration such as `my @array =` or `my %hash =` evaluates its expression in list context. A list value evaluated in list context returns the list, which then gets fed in to populate the array or hash:

```
my @array = ("Alpha", "Beta", "Gamma", "Pie");
my %hash = ("Alpha" => "Beta", "Gamma" => "Pie");
```

No surprises so far.

A scalar expression evaluated in list context turns into a single-element list:

```
my @array = "Mendeleev";
print $array[0];     # "Mendeleev"
print scalar @array; # "1"
```

A list expression evaluated in scalar context returns *the final scalar in the list*:

```
my $scalar = ("Alpha", "Beta", "Gamma", "Pie");
print $scalar; # "Pie"
```

An array expression (an array is different from a list, remember?) evaluated in scalar context returns *the length of the array*:

```
my @array = ("Alpha", "Beta", "Gamma", "Pie");
my $scalar = @array;
print $scalar; # "4"
```

You can force any expression to be evaluated in scalar context using the `scalar` built-in function. In fact, this is why we use `scalar` to retrieve the length of an array.

You are not bound by law or syntax to return a scalar value when a subroutine is evaluated in scalar context, nor to return a list value in list context. As seen above, Perl is perfectly capable of fudging the result for you.

References and nested data structures

In the same way that lists cannot contain lists as elements, **arrays and hashes cannot contain other arrays and hashes as elements.** They can only contain scalars. For example:

```
my @outer = ();
my @inner = ("Mercury", "Venus", "Earth");

$outer[0] = @inner;

print $outer[0];    # "3", not "MercuryVenusEarth" as you would hope
print $outer[0][0]; # error, not "Mercury" as you would hope
```

`$outer[0]` is a scalar, so it demands a scalar value. When you try to assign an array value like `@inner` to it, `@inner` is evaluated in scalar context. This is the same as assigning `scalar @inner`, which is the length of array `@inner`, which is 3.

However, a scalar variable may contain a *reference* to any variable, including an array variable or a hash variable. This is how more complicated data structures are created in Perl.

A reference is created using a backslash.

```
my $colour    = "Indigo";
my $scalarRef = \$colour;
```

Any time you would use the name of a variable, you can instead just put some braces in, and, within the braces, put a *reference* to a variable instead.

```
print $colour;         # "Indigo"
print $scalarRef;      # e.g. "SCALAR(0x182c180)"
print ${ $scalarRef }; # "Indigo"
```

As long as the result is not ambiguous, you can omit the braces too:

```
print $$scalarRef; # "Indigo"
```

Hence:

```
my %owner1 = (
        "name" => "Santa Claus",
        "DOB"  => "1882-12-25",
);

my %owner2 = (
        "name" => "Mickey Mouse",
        "DOB"  => "1928-11-18",
);

my @owners = ( \%owner1, \%owner2 );

my %account = (
        "number" => "12345678",
        "opened" => "2000-01-01",
        "owners" => \@owners,
);
```

It is also possible to declare *anonymous* arrays and hashes using different symbols. Use square brackets for an anonymous array and braces for an anonymous hash. The value returned in each case is a *reference* to the anonymous data structure in question. Watch carefully, this results in exactly the same `%account` as above:

```
# Braces denote an anonymous hash
my $owner1 = {
        "name" => "Santa Claus",
        "DOB"  => "1882-12-25",
};

my $owner2 = {
        "name" => "Mickey Mouse",
        "DOB"  => "1928-11-18",
};

# Square brackets denote an anonymous array
my $owners = [ $owner1, $owner2 ];

my %account = (
        "number" => "12345678",
        "opened" => "2000-01-01",
        "owners" => $owners,
);
```

All of that is quite long-winded, so here's how it can all be achieved without all of those tedious intermediate variables:

```
my %account = (
        "number" => "31415926",
        "opened" => "3000-01-01",

        "owners" => [

                {
                        "name" => "Philip Fry",
                        "DOB"  => "1974-08-06",
                },

                {
                        "name" => "Hubert Farnsworth",
                        "DOB"  => "2841-04-09",
                },
        ],
);
```

And here's how you'd print that data out:

```
print "Account #", $account{"number"}, "\n";
print "Opened on ", $account{"opened"}, "\n";
print "Joint owners:\n";
print "\t", $account{"owners"}[0]{"name"}, " (born ", $account{"owners"}[0]{"DOB"},
")\n";
```

```
print "\t", $account{"owners"}[1]{"name"}, " (born ", $account{"owners"}[1]{"DOB"},
")\n";
```

How to shoot yourself in the foot with references to arrays and hashes

This array has five elements:

```
my @array1 = (1, 2, 3, 4, 5);
print @array1; # "12345"
```

This array has one element (which happens to be a reference to an anonymous, five-element array):

```
my @array2 = [1, 2, 3, 4, 5];
print @array2; # e.g. "ARRAY(0x182c180)"
```

This *scalar* is a reference to an anonymous, five-element array:

```
my $array3 = [1, 2, 3, 4, 5];
print $array3;      # e.g. "ARRAY(0x22710c0)"
print @{ $array3 }; # "12345"
print @$array3;     # "12345"
```

Some syntactic sugar

The arrow shortcut operator `->` is much quicker and more readable than using tedious braces all the time to reference things. You will see people accessing hashes through references very frequently, so try to get used to it.

```
my @colours = ("Red", "Orange", "Yellow", "Green", "Blue");
my $arrayRef = \@colours;

print $colours[0];       # direct array access
print ${ $arrayRef }[0]; # use the reference to get to the array
print $arrayRef->[0];    # exactly the same thing

my %atomicWeights = ("Hydrogen" => 1.008, "Helium" => 4.003, "Manganese" => 54.94);
my $hashRef = \%atomicWeights;

print $atomicWeights{"Helium"}; # direct hash access
print ${ $hashRef }{"Helium"};  # use a reference to get to the hash
print $hashRef->{"Helium"};     # exactly the same thing - this is very common
```

Flow control

`if` ... `elsif` ... `else` ...

No surprises here, other than the spelling of `elsif`:

```
my $word = "antidisestablishmentarianism";
my $strlen = length $word;

if($strlen >= 15) {
        print "'", $word, "' is a very long word";
} elsif(10 <= $strlen && $strlen < 15) {
        print "'", $word, "' is a medium-length word";
} else {
        print "'", $word, "' is a a short word";
}
```

Perl provides a shorter "*statement* `if` *condition*" syntax which is highly recommended:

```
print "'", $word, "' is actually enormous" if $strlen >= 20;
```

`unless` ... `else` ...

```
my $temperature = 20;

unless($temperature > 30) {
        print $temperature, " degrees Celsius is not very hot";
} else {
        print $temperature, " degrees Celsius is actually pretty hot";
}
```

`unless` blocks are generally best avoided like the plague because they are very confusing. An "`unless` [... `else`]" block can be trivially refactored into an "`if` [... `else`]" block by negating the condition [or by keeping the condition and swapping the blocks]. Mercifully, there is no `elsunless` keyword.

This, by comparison, is highly recommended because it is so easy to read:

```
print "Oh no it's too cold" unless $temperature > 15;
```

Ternary operator

The ternary operator `?:` allows simple `if` statements to be embedded in a statement. The canonical use for this is singular/plural forms:

```
my $gain = 48;
print "You gained ", $gain, " ", ($gain == 1 ? "experience point" : "experience
points"), "!";
```

Aside: singulars and plurals are best spelled out in full in both cases. Don't do something clever like the following, because anybody searching the codebase to replace the words "tooth" or "teeth" will never find this line:

```
my $lost = 1;
print "You lost ", $lost, " t", ($lost == 1 ? "oo" : "ee"), "th!";
```

Ternary operators can be nested:

```
my $eggs = 5;
print "You have ", $eggs == 0 ? "no eggs" :
                   $eggs == 1 ? "an egg"  :
                   "some eggs";
```

`if`, `unless` and `?:` statements evaluate their conditions in scalar context. For example, `if(@array)` returns true if and only if `@array` has 1 or more elements. It doesn't matter what those elements are - they may contain `undef` or other false values for all we care.

Array iteration

There's More Than One Way To Do It.

Basic C-style `for` loops are available, but these are obtuse and old-fashioned and should be avoided. Notice how we have to put a `my` in front of our iterator `$i`, in order to declare it:

```
for(my $i = 0; $i < scalar @array; $i++) {
        print $i, ": ", $array[$i];
}
```

Native iteration over an array is much nicer. Note: unlike PHP, the `for` and `foreach` keywords are synonyms. Just use whatever looks most readable:

```
foreach my $string ( @array ) {
        print $string;
}
```

If you do need the indices, the range operator `..` creates an anonymous array of integers:

```
foreach my $i ( 0 .. $#array ) {
        print $i, ": ", $array[$i];
}
```

If you don't provide an explicit iterator, Perl uses a default iterator, `$_`. `$_` is the first and friendliest of the built-in variables:

```
foreach ( @array ) {
        print $_;
}
```

If using the default iterator, and you only wish to put a single statement inside your loop, you can use the super-short loop syntax:

```
print $_ foreach @array;
```

Perl also provides `while` loops but those are coming up in a second.

Hash iteration

You can't iterate over a hash. However, you can iterate over its keys. Use the `keys` built-in function to retrieve an array containing all the keys of a hash. Then use the `foreach` approach that we used for arrays:

```
foreach my $key (keys %scientists) {
        print $key, ": ", $scientists{$key};
}
```

Since a hash has no underlying order, the keys may be returned in any order. Use the `sort` built-in function to sort the array of keys alphabetically beforehand:

```
foreach my $key (sort keys %scientists) {
        print $key, ": ", $scientists{$key};
}
```

There is also a special `each` built-in function which retrieves key/value pairs one at a time. Every time `each` is called, it returns an array containing two values, until the end of the array is reached, when a false value is returned. We assign the values of two scalars to the values of the array, simultaneously:

```
while( my ($key, $value) = each %scientists ) {
        print $key, ": ", $value;
}
```

Loop control

`next` and `last` can be used to control the progress of a loop. In most programming languages these are known as `continue` and `break` respectively. We can also optionally provide a label for any loop. By convention, labels are written in `ALLCAPITALS`. Having labelled the loop, `next` and `last` may target that label. This example lists all the non-fictional animals from an array:

```
my @input = (
        "dragon", "camel", "cow", "pangolin", "unicorn",
        "pig", "sheep", "donkey", "pig", "basilisk",
        "monkey", "jellyfish", "squid", "crab", "dragon",
);
my @fictional = ("basilisk", "dragon", "unicorn");

INPUT: foreach my $input ( @input ) {

        # See if this input animal is fictional
        foreach my $fictional ( @fictional ) {

                # It is?
                if($input eq $fictional) {

                        # Then jump to the next input animal
                        next INPUT;
                }
        }

        # Animal is not fictional, print it
        print $input;
}
```

Array functions

In-place array modification

We'll use `@stack` to demonstrate these:

```
my @stack = ("Fred", "Eileen", "Denise", "Charlie");
print @stack; # "FredEileenDeniseCharlie"
```

`pop` extracts and returns the final element of the array. This can be thought of as the top of the stack:

```
print pop @stack; # "Charlie"
print @stack;     # "FredEileenDenise"
```

`push` appends extra elements to the end of the array:

```
push @stack, "Bob", "Alice";
print @stack; # "FredEileenDeniseBobAlice"
```

`shift` extracts and returns the first element of the array:

```
print shift @stack; # "Fred"
print @stack;       # "EileenDeniseBobAlice"
```

`unshift` inserts new elements at the beginning of the array:

```
unshift @stack, "Hank", "Grace";
print @stack; # "HankGraceEileenDeniseBobAlice"
```

`pop`, `push`, `shift` and `unshift` are all special cases of `splice`. `splice` removes and returns an array slice, replacing it with a different array slice:

```
print splice(@stack, 1, 4, "<<<", ">>>"); # "GraceEileenDeniseBob"
print @stack;                             # "Hank<<<>>>Alice"
```

Creating new arrays from old

Perl provides the following functions which act on arrays to create other arrays.

`join`

The `join` function concatenates many strings into one:

```
my @elements = ("Antimony", "Arsenic", "Aluminum", "Selenium");
print @elements;             # "AntimonyArsenicAluminumSelenium"
print "@elements";           # "Antimony Arsenic Aluminum Selenium"
print join(", ", @elements); # "Antimony, Arsenic, Aluminum, Selenium"
```

`reverse`

The `reverse` function returns an array in reverse order:

```
my @numbers = ("one", "two", "three");
print reverse(@numbers); # "threetwoone"
```

map

The `map` function takes an array as input and applies an operation to every scalar `$_` in this array. It then constructs a new array out of the results. The operation to perform is provided in the form of a single expression inside braces:

```
my @capitals = ("Baton Rouge", "Indianapolis", "Columbus", "Montgomery", "Helena",
"Denver", "Boise");

print join ", ", map { uc $_ } @capitals;
# "BATON ROUGE, INDIANAPOLIS, COLUMBUS, MONTGOMERY, HELENA, DENVER, BOISE"
```

grep

The `grep` function takes an array as input and returns a filtered array as output. The syntax is similar to `map`. This time, the second argument is evaluated for each scalar `$_` in the input array. If a boolean true value is returned, the scalar is put into the output array, otherwise not.

```
print join ", ", grep { length $_ == 6 } @capitals;
# "Helena, Denver"
```

Instead of a single Perl expression, you may supply a regular expression. In this case, the scalar is put into the output array only if the regular expression matches `$_`:

```
print join ", ", grep m/^[B-H]/, @capitals;
# "Baton Rouge, Columbus, Helena, Denver, Boise"
```

Obviously, the length of the resulting array is the *number of successful matches*, which means you can use `grep` to quickly check whether an array contains an element:

```
print scalar grep { $_ eq "Columbus" } @capitals; # "1"
```

`grep` and `map` may be combined to form *list comprehensions*, an exceptionally powerful feature conspicuously absent from many other programming languages.

sort

By default, the `sort` function returns the input array, sorted into alphabetical order:

```
my @elevations = (19, 1, 2, 100, 3, 98, 100, 1056);

print join ", ", sort @elevations;
# "1, 100, 100, 1056, 19, 2, 3, 98"
```

However, similar to `grep` and `map`, you may supply some code of your own. Sorting is always performed using a series of comparisons between elements. Your block receives `$a` and `$b` as inputs and should return -1 if `$a` is "less than" `$b`, 0 if they are "equal" or 1 if `$a` is "greater than" `$b`.

The `cmp` operator does exactly this for strings:

```
print join ", ", sort { $a cmp $b } @elevations;
# "1, 100, 100, 1056, 19, 2, 3, 98"
```

The "spaceship operator", `<=>`, does the same for numbers:

```
print join ", ", sort { $a <=> $b } @elevations;
```

```
# "1, 2, 3, 19, 98, 100, 100, 1056"
```

`$a` and `$b` are always scalars, but they can be references to quite complex objects which are difficult to compare. If you need more space for the comparison, you can create a separate subroutine and provide its name instead:

```
sub comparator {
	# lots of code...
	# return -1, 0 or 1
}

print join ", ", sort comparator @elevations;
```

You can't do this for `grep` or `map` operations.

Notice how the subroutine and block are never explicitly provided with `$a` and `$b`. Like `$_`, `$a` and `$b` are, in fact, global variables which are *populated* with a pair of values to be compared each time.

Built-in functions

By now you have seen at least a dozen built-in functions: `print`, `sort`, `map`, `grep`, `each`, `keys`, `scalar` and so on. Built-in functions are one of Perl's greatest strengths. They

- are numerous
- are very useful
- are extensively documented (in "perlfunc")
- vary greatly in syntax, so check the documentation
- sometimes accept regular expressions as arguments
- sometimes accept entire blocks of code as arguments
- sometimes don't require commas between arguments
- sometimes will consume an arbitrary number of comma-separated arguments and sometimes will not
- sometimes will fill in their own arguments if too few are supplied
- generally don't require brackets around their arguments except in ambiguous circumstances

The best advice regarding built-in functions is to know that they exist, so that you can **use them**. If you are carrying out a task which feels like it's low-level and common enough that it's been done many times before, the chances are that it has.

User-defined subroutines

Subroutines are declared using the `sub` keyword. In contrast with built-in functions, user-defined subroutines always accept the same input: a list of scalars. That list may of course have a single element, or be empty. A single scalar is taken as a list with a single element. A hash with *N* elements is taken as a list with 2*N* elements.

Although the brackets are optional, subroutines should always be invoked using brackets, even when called with no arguments. This makes it clear that a subroutine call is happening.

Once you're inside a subroutine, the arguments are available using the built-in array variable `@_`. Examples follow.

Unpacking arguments

There's More Than One Way To unpack these arguments, but some are superior to others.

The example subroutine `leftPad` below pads a string out to the required length using the supplied pad character. (The `x` function concatenates multiple copies of the same string in a row.) (Note: for brevity, these subroutines all lack some elementary error checking, i.e. ensuring the pad character is only 1 character, checking that the width is greater than or equal to the length of existing string, checking that all needed arguments were passed at all.)

`leftPad` is typically invoked as follows:

```
print leftPad("hello", 10, "+"); # "+++++hello"
```

1. Some people don't unpack the arguments at all and use `@_` "live". This is unreadable and discouraged:

   ```
   sub leftPad {
           my $newString = ($_[2] x ($_[1] - length $_[0])) . $_[0];
           return $newString;
   }
   ```

2. Unpacking `@_` is only slightly less strongly discouraged:

   ```
   sub leftPad {
           my $oldString = $_[0];
           my $width     = $_[1];
           my $padChar   = $_[2];
           my $newString = ($padChar x ($width - length $oldString)) . $oldString;
           return $newString;
   }
   ```

3. Unpacking `@_` by removing data from it using `shift` is highly recommended for up to 4 arguments:

   ```
   sub leftPad {
           my $oldString = shift @_;
           my $width     = shift @_;
           my $padChar   = shift @_;
           my $newString = ($padChar x ($width - length $oldString)) . $oldString;
           return $newString;
   }
   ```

   If no array is provided to the `shift` function, then it operates on `@_` implicitly. This approach is seen very commonly:

   ```
   sub leftPad {
           my $oldString = shift;
           my $width     = shift;
           my $padChar   = shift;
           my $newString = ($padChar x ($width - length $oldString)) . $oldString;
           return $newString;
   }
   ```

   Beyond 4 arguments it becomes hard to keep track of what is being assigned where.

4. You can unpack `@_` all in one go using multiple simultaneous scalar assignment. Again, this is okay for up to 4 arguments:

   ```
   sub leftPad {
           my ($oldString, $width, $padChar) = @_;
           my $newString = ($padChar x ($width - length $oldString)) . $oldString;
           return $newString;
   }
   ```

5. For subroutines with large numbers of arguments or where some arguments are optional or cannot be used in combination with others, best practice is to require the user to provide a hash of arguments when calling the subroutine, and then unpack `@_` back into that hash of arguments. For this approach, our subroutine call would look a little different:

   ```
   print leftPad("oldString" => "pod", "width" => 10, "padChar" => "+");
   ```

And the subroutine itself looks like this:

```
sub leftPad {
        my %args = @_;
        my $newString = ($args{"padChar"} x ($args{"width"} - length
$args{"oldString"})) . $args{"oldString"};
        return $newString;
}
```

Returning values

Like other Perl expressions, subroutine calls may display contextual behaviour. You can use the `wantarray` function (which should be called `wantlist` but never mind) to detect what context the subroutine is being evaluated in, and return a result appropriate to that context:

```
sub contextualSubroutine {
        # Caller wants a list. Return a list
        return ("Everest", "K2", "Etna") if wantarray;

        # Caller wants a scalar. Return a scalar
        return "Everest ::: K2 ::: Etna";
}

my @array = contextualSubroutine();
print @array; # "EverestK2Etna"

my $scalar = contextualSubroutine();
print $scalar; # "Everest ::: K2 ::: Etna"
```

A quick note on variables

All the variables you have seen so far are *lexical* variables, which are declared using the `my` keyword and last until the end of the enclosing block or file.

Package variables, which we are about to meet, are declared using the `our` keyword and are effectively global in scope.

Packages and modules

In Perl, packages and modules are different things.

Packages

A *package* is a namespace in which subroutines and package variables can be declared. Any subroutine or package variable you declare is implicitly declared within the current package. At the beginning of execution, you are in the `main` package, but you can switch package using the `package` built-in function:

```
sub subroutine {
        print "universe";
}

our $variable = "empty";

package Food::Potatoes;

# no collision:
sub subroutine {
        print "kingedward";
}

our $variable = "mashed";
```

Any time you call a subroutine, you implicitly call a subroutine which is inside the current package.

The same is true of package variables. Alternatively, you can explicitly provide a package. See what happens if we continue the above script:

```
subroutine();    # "kingedward"
print $variable; # "mashed"

main::subroutine();                # "universe"
print $main::variable;             # "empty"
Food::Potatoes::subroutine();     # "kingedward"
print $Food::Potatoes::variable; # "mashed"
```

Modules

A *module* is a `.pm` file that you can include in another Perl file (script or module). A module is a text file with exactly the same syntax as a `.pl` Perl script. An example module might be located at `C:\foo\bar\baz\Mathematics\Powers.pm` or `/foo/bar/baz/Mathematics/Powers.pm`, and read as follows:

```
use strict;
use warnings;

package Mathematics::Powers;

our $e = 2.71828;

sub exp {
        return $e ** shift;
}

1;
```

Because a module is executed from top to bottom when it is loaded, you need to return a true value at the end to show that it was loaded successfully. `return 1` would suffice. If you don't use `return`, the value returned is *the value returned when the most recent statement was evaluated*. So, you will often see `1` at the bottom of a Perl module, as shown above.

So that the Perl interpreter can find them, directories containing Perl modules should be listed in your environment variable `PERL5LIB` beforehand. List the root directory containing the modules, don't list the module directories or the modules themselves:

```
set PERL5LIB=C:\foo\bar\baz;%PERL5LIB%
```

or

```
export PERL5LIB=/foo/bar/baz:$PERL5LIB
```

Once the Perl module is created and `perl` knows where to look for it, you can use the `require` built-in function to search for and execute it during a Perl script. For example, calling `require Mathematics::Powers` causes the Perl interpreter to search each directory listed in `PERL5LIB` in turn, looking for a file called `Mathematics/Powers.pm`. After the module has been loaded, the subroutines and variables that were defined there suddenly become available in the main script. Our example script might be called `powers.pl` and read as follows:

```
use strict;
use warnings;

require Mathematics::Powers;

print Mathematics::Powers::exp(2); # "7.3890461584"
```

Now read this next bit carefully.

Packages and modules are two completely separate and distinct features of the Perl programming language. The fact that they both use the same double colon delimiter is a monumental red herring. It is possible to switch packages multiple times over the course of a script or module, and it is possible to use the same package declaration in multiple locations in multiple files. Calling `require Foo::Bar` *does not* look for and load a file with a `package Foo::Bar` declaration somewhere inside it. Calling `require Foo::Bar` *does not* necessarily load subroutines or package variables in the `Foo::Bar` namespace. Calling `require Foo::Bar` merely loads a file called `Foo/Bar.pm`, which need not have *any* kind of package declaration inside it at all, and in fact might declare `package Baz::Qux` and other nonsense inside it for all you know.

Likewise, a subroutine call `Baz::Qux::processThis()` need not necessarily have been declared inside a file named `Baz/Qux.pm`. It could have been declared *literally anywhere*.

Separating these two concepts is one of the stupidest features of Perl, and treating them as separate concepts invariably results in chaotic, maddening code. Fortunately for us, the majority of Perl programmers obey the following two laws:

1. **A Perl script (`.pl` file) must always contain exactly zero `package` declarations.**
2. **A Perl module (`.pm` file) must always contain exactly one `package` declaration, corresponding exactly to its name and location.** E.g. module `Mathematics/Powers.pm` must begin with `package Mathematics::Powers`.

Because of this, in practice you will find that most "packages" and "modules" produced by reliable third parties *can* be regarded and referred to interchangeably. However, it is important that you do not take this for granted, because one day you *will* meet code produced by a madman.

Object-oriented Perl

Perl is not a great language for OO programming. Perl's OO capabilities were grafted on after the fact, and this shows.

- An *object* is simply a reference (i.e. a scalar variable) which happens to know which class its referent belongs to. To tell a reference that its referent belongs to a class, use `bless`. To find out what class a reference's referent belongs to (if any), use `ref`.
- A *method* is simply a subroutine that expects an object (or, in the case of class methods, a package name) as its first argument. Object methods are invoked using `$obj->method()`; class methods are invoked using `Package::Name->method()`.
- A *class* is simply a package that happens to contain methods.

A quick example. An example module `Animals/Animals.pm` containing a class `Animals::Animal` reads like this:

```
use strict;
use warnings;

package Animals::Animal;

sub eat {
        # First argument is always the object to act upon.
        my $self = shift;

        foreach my $food ( @_ ) {
                if($self->canEat($food)) {
                        print "Eating ", $food;
                } else {
                        print "Can't eat ", $food;
                }
        }
}

sub canEat {
        return 1;
}
```

```
1;
```

And a Perl script making use of this class might read:

```
use strict;
use warnings;

require Animals::Animal;

my $animal = {};                     # $animal is an ordinary hash reference
print ref $animal;                   # "HASH"
bless $animal, "Animals::Animal"; # now it is an object of class "Animals::Animal"
print ref $animal;                   # "Animals::Animal"

$animal->eat("insects", "curry", "salmon");
```

This final call is equivalent to `Animals::Animal::eat($animal, "insects", "curry", "salmon")`.

Note: literally any reference can be blessed into any class. It's up to you to ensure that (1) the referent can actually be used as an instance of this class and (2) that the class in question exists and has been loaded.

Inheritance

To create a class inheriting from a base class, populate the `@ISA` package variable. Let's suppose we subclassed `Animals::Animal` with `Animals::Bear`, located at `Animals/Bear.pm`:

```
use strict;
use warnings;

package Animals::Bear;

require Animals::Animal;

# Inherit from Animals::Animal
our @ISA = ("Animals::Animal");

# Override one method
sub canEat {
        shift;
        return 1 if shift eq "salmon";
        return 0;
}

1;
```

And some sample code:

```
use strict;
use warnings;

require Animals::Bear;

my $bear = bless {}, "Animals::Bear";

$bear->eat("insects", "curry", "salmon"); # eat only the salmon
```

This final method call tries to invoke `Animals::Bear::eat($bear, "insects", "curry", "salmon")`, but a subroutine `eat()` isn't defined in the `Animals::Bear` package. However, because `@ISA` has been populated with a parent package `Animals::Animal`, the Perl interpreter tries calling `Animals::Animal::eat($bear, "insects", "curry", "salmon")` instead, which works. Note how the class `Animals::Animal` had to be loaded explicitly by `Animals::Bear`.

Since `@ISA` is an array, Perl supports multiple inheritance, with all the benefits and horrors this entails.

`BEGIN` blocks

A `BEGIN` block is executed as soon as the compiler has finished parsing it, even before the compiler parses the rest of the file. It is ignored at execution time.

```
use strict;
use warnings;

# a package declaration might go here

BEGIN {
        # do something extremely important
}

# actual code
```

A `BEGIN` block is always executed first. If you create multiple `BEGIN` blocks (don't), they are executed in order from top to bottom as the compiler encounters them. A `BEGIN` block always executes first even if it is placed halfway through a script (don't do this) or even at the end (or this).

Because they are executed at compilation time, a `BEGIN` block placed inside a conditional block will *still* be executed first, even if the conditional evaluates to false and despite the fact that the conditional *has not been evaluated at all yet* and in fact *may never be evaluated*. **Do not put `BEGIN` blocks in conditionals!** If you want to do something conditionally at compile time, you need to put the conditional *inside* the `BEGIN` block:

```
BEGIN {
        if($condition) {
                # etc.
        }
}
```

use

Okay. Now that you understand the obtuse behaviour and semantics of packages, modules, class methods and `BEGIN` blocks, I can explain the exceedingly commonly-seen `use` function.

The following three statements:

```
use Bugs::Caterpillar ("crawl", "pupate");
use Bugs::Caterpillar ();
use Bugs::Caterpillar;
```

are respectively equivalent to:

```
BEGIN {
        require Bugs::Caterpillar;
        Bugs::Caterpillar->import("crawl", "pupate");
}
BEGIN {
        require Bugs::Caterpillar;
}
BEGIN {
        require Bugs::Caterpillar;
        Bugs::Caterpillar->import();
}
```

- No, the three examples are not in the wrong order. It is just that Perl is dumb.
- A `use` call is a disguised `BEGIN` block. The same caveats apply. `use` statements must always be placed at the top of the file, and **never inside conditionals**.
- `import()` is not a built-in Perl function. It is a **user-defined class method**. The burden is on the programmer of the `Bugs::Caterpillar` package to define or inherit `import()`, and the method could theoretically accept anything as arguments and do anything with those arguments.
- Notice how `require Bugs::Caterpillar` loads a **module** named `Bugs/Caterpillar.pm`, whereas `Bugs::Caterpillar->import()` calls the `import()` subroutine that was defined inside

the `Bugs::Caterpillar` **package**. Let's hope the module and the package coincide!

Exporter

The most common way to define an `import()` method is to inherit it from Exporter module. Exporter is a *de facto* core feature of the Perl programming language. In Exporter's implementation of `import()`, the list of arguments that you pass in is interpreted as a list of subroutine names. When a subroutine is `import()`ed, it becomes available in the current namespace as well as in its own original namespace.

This concept is easiest to grasp using an example. Here's what `Bugs/Caterpillar.pm` looks like:

```
use strict;
use warnings;

package Bugs::Caterpillar;

require Exporter;

our @ISA = ("Exporter");

our @EXPORT_OK = ("crawl", "eat");
our @EXPORT    = ("crawl");

sub crawl  { print "inch inch";   }
sub eat    { print "chomp chomp"; }
sub pupate { print "bloop bloop"; }

1;
```

And here's a script which makes use of the `Bugs/Caterpillar.pm` module:

```
use strict;
use warnings;

use Bugs::Caterpillar;

Bugs::Caterpillar::crawl();  # "inch inch"
Bugs::Caterpillar::eat();    # "chomp chomp"
Bugs::Caterpillar::pupate(); # "bloop bloop"

crawl(); # "inch inch"
```

The package variable `@EXPORT_OK` can be populated with a list of all subroutines which the user can import explicitly by passing subroutine names to `import()`. If `import()` is called with the name of a subroutine not in this list, a runtime error will occur. For example, try `use Bugs::Caterpillar ("pupate")`.

The package variable `@EXPORT` can be populated with a list of subroutines to be exported by default. These are exported if `import()` is called with no arguments at all, which is what happens in this example.

As a result of being `import()`ed, a method such as `Bugs::Caterpillar::crawl()` become available without qualification as `crawl()`. This saves typing. (Note: regardless of the content of `@EXPORT_OK`, every method can always be called "longhand", as shown above. There are no private methods in Perl. Customarily, a method intended for private use is named with a leading underscore or two.)

A caution. Notice how `crawl()` was neither defined in the script, nor explicitly `import()`ed from another file with e.g. `use Bugs::Caterpillar ("crawl")`. Suppose the middle three lines weren't there to provide clues, and suppose there were a dozen `use` calls alongside `use Bugs::Caterpillar`. And remember that any module is free to have more `use` calls of its own. In such a situation, it is extremely difficult to locate the place where `crawl()` was originally defined. The moral of this story is twofold:

1. When creating a module which makes use of Exporter, never use `@EXPORT` to export subroutines by default. Always make the user call subroutines "longhand" or `import()` them explicitly (using e.g. `use Bugs::Caterpillar ("crawl")`, which is a strong clue to look in `Bugs/Caterpillar.pm` for the definition of `crawl()`).

2. When `use`ing a module, always explicitly name the subroutines you want to `import()`. If you don't want to `import()` any subroutines and wish to refer to them longhand, you must supply an explicit empty list: `use Bugs::Caterpillar ()`.

Files

A *file handle* is a completely different object from a scalar, array or hash variable. File handles are customarily represented in `ALLCAPS`; three familiar built-in filehandles are `STDIN`, `STDOUT` and `STDERR`.

Filehandles don't need declaring explicitly using `my` or `our`. They pop into existence automatically. A file handle can be opened using `open`. `open` must be supplied with a *method*. The method `<` indicates that we wish to open the file to read from it:

```
my $f = "text.txt";
my $result = open INPUT, "<", $f;

if(!defined $result || !$result) {
        die "Couldn't open ", $f, " for reading";
}
```

As seen above, you should always check that the `open` operation completed successfully. If successful, `open` returns a true value. Otherwise, it returns `undef`. This checking procedure being rather tedious, a frequently-seen idiom is this:

```
open(INPUT, "<", $f) || die "Couldn't open ", $f, " for reading";
```

Notice how without the brackets, this would be:

```
open INPUT, "<", $f || die "Couldn't open ", $f, " for reading";
```

Which is the same as the nonsensical:

```
open INPUT, "<", ($f || die "Couldn't open ", $f, " for reading");
```

For this reason (and, as far as I can tell, solely this reason), Perl provides a completely separate operator, `or`, which works exactly like `||` except that it has extremely low precedence, making this possible:

```
open INPUT, "<", $f or die "Couldn't open ", $f, " for reading";
```

To read a line of text from a filehandle, use the `readline` built-in function. `readline` returns a full line of text, with a line break intact at the end of it (except possibly for the final line of the file), or `undef` if you've reached the end of the file.

```
while(1) {
        my $line = readline INPUT;
        last unless defined $line;
        # process the line...
}
```

To truncate that possible trailing line break, use `chomp`:

```
chomp $line;
```

Note that `chomp` acts on `$line` in place. `$line = chomp $line` is probably not what you want.

You can also use `eof` to detect the end of the file:

```
while(!eof INPUT) {
        my $line = readline INPUT;
        # process $line...
}
```

But beware of just using `while(my $line = readline INPUT)`, because if `$line` turns out to be `"0"`, the loop will terminate early. If you want to write something like that, Perl provides the `<>` operator which wraps up `readline` in a fractionally safer way. This is very commonly-seen and perfectly safe:

```
while(my $line = <INPUT>) {
        # process $line...
}
```

And even:

```
while(<INPUT>) {
        # process $_...
}
```

To read a single line of user input:

```
my $line = <STDIN>;
```

To just wait for the user to hit Enter:

```
<STDIN>;
```

Calling `<>` with no filehandle reads data from standard input, or from any files named in arguments when the Perl script was called.

Writing to a file involves first opening it in a different mode. The method `>` indicates that we wish to open the file to write to it. (`>` will clobber the content of the target file if it already exists and has content. To merely append to an existing file, use mode `>>`). Then, simply provide the filehandle as a zeroth argument for the `print` function.

```
open OUTPUT, ">", $f or die "Couldn't open ", $f, " for writing";
print OUTPUT "The eagles have left the nest";
```

Notice the absence of a comma between the filehandle and the first argument in `print`. As you've gathered, if the filehandle is omitted, `STDOUT` is used by default.

File handles are actually closed automatically at script exit time, but otherwise:

```
close INPUT;
close OUTPUT;
```

A scalar variable may hold a reference to a file handle instead of a variable:

```
my $fh;
open $fh, "<", "text.txt" or die;
while(<$fh>) {
        # etc...
}
close $fh;
```

System calls

Apologies if you already know the following non-Perl-related facts. Every time a process finishes on a Windows or Linux system (and, I assume, on most other systems), it concludes with a 16-bit *status word*. The highest 8 bits constitute a *return code* between 0 and 255 inclusive, with 0 conventionally representing unqualified success, and other values representing various degrees of failure. The other 8 bits are less frequently examined - they "reflect mode of failure, like signal death and core dump information".

You can exit from a Perl script with the return code of your choice (from 0 to 255) using `exit`.

Perl provides More Than One Way To - in a single call - spawn a child process, pause the current script until the child process has finished, and then resume interpretation of the current script. Whichever method is used, you will find that immediately afterwards, the built-in variable `$?` (`$CHILD_ERROR`) has been populated with the status word that was returned from that child process's termination. You can get the return code by taking just the highest 8 of those 16 bits: `$? >> 8`.

The `system` function can be used to invoke another program with the arguments listed. The value returned by `system` is the same value with which `$?` is populated:

```
my $rc = system "perl", "anotherscript.pl", "foo", "bar", "baz";
$rc >>= 8;
print $rc; # "37";
```

Alternatively, you can use backticks ` `` ` to run an actual command at the command line and capture the standard output from that command. In scalar context the entire output is returned as a single string. In list context, the entire output is returned as an array of strings, each one representing a line of output.

```
my $text = `perl anotherscript.pl foo bar baz`;
print $text; # "foobarbaz"
```

This is the behaviour which would be seen if `anotherscript.pl` contained, for example:

```
use strict;
use warnings;

print @ARGV;
exit(37);
```

Miscellaneous notes

Perl provides a wide selection of quote-like operators in addition to what you've seen in these documents:

- There's an alternate syntax, `qw{ }`, for declaring arrays. This often seen in `use` statements:

  ```
  use Account qw{create open close suspend delete};
  ```

- `qr//` can be used to put a regex into a scalar variable. This is especially useful because recompiling a regular expression multiple times actually takes substantial time:

  ```
  my @capitals = ("Baton Rouge", "Indianapolis", "Columbus", "Montgomery",
  "Helena", "Denver", "Boise");
  my $regex = qr/^[B-H]/;
  print join ", ", grep /$regex/, @capitals;
  ```

- `qx{ }` can be used instead of \`backticks\` to invoke a program and capture its output:

  ```
  my $text = qx{perl anotherscript.pl foo bar baz};
  ```

- And many more!

Instead of braces, you can use any character you like as the delimiter in these alternate quote-like operators, as well as in `m//` regex matches and `s///` regex replacements. This is actually quite useful if your regex contains a lot of slashes or backslashes. For example, `m!///!` matches three literal forward slashes.

Perl does have `CONSTANTS`. These are discouraged now, but weren't always. Constants are actually just subroutine calls with omitted brackets.

Sometimes people omit quotes around hash keys. They can get away with it because in this situation a bareword (a string with no sigil) occurs as a string, as opposed to a subroutine call or a filehandle or a package name.

If you see a block of unformatted code wrapped in a delimiter with double chevrons, like `<<EOF`, the magic word to Google for is "heredoc".

The Data::Dumper module can be used to output an arbitrary scalar variable to the screen. This is an essential debug tool.

Warning! Many built-in functions can be called with no arguments, **causing them to operate on `$_` instead**. Hopefully this will help you understand formations like:

```
print foreach @array;
```

```
foreach ( @array ) {
        next unless defined;
}
```

I dislike this formation because it can lead to problems when refactoring.

File tests

The function `-e` is a built-in function which tests whether the named file exists.

```
print "what" unless -e "/usr/bin/perl";
```

The function `-d` is a built-in function which tests whether the named file is a directory.

The function `-f` is a built-in function which tests whether the named file is a plain file.

These are just three of a large class of functions of the form `-X` where `X` is some lower- or upper-case letter. These functions are called *file tests*. Note the leading minus sign. In a Google query, the minus sign indicates to exclude results containing this search term. This makes file tests hard to Google for! Just search for "perl file test" instead.