The Perl Philosophy

Perl is a language for getting things done. It's flexible, forgiving, and malleable. In the hands of a capable programmer, it can perform almost any task, from one-liner calculations and automations to multi-programmer, multi-year projects and everything in between.

Perl is powerful, and modern Perl--Perl which takes advantage of the best knowledge, deepest experience, and reusable idioms of the global Perl community--can be maintainable, fast, and easy to use. Perhaps most importantly, it can help you do what you need to do with little frustration and no ceremony.

Perl is a pragmatic language. You, the programmer, are in charge. Rather than manipulating your mind and your problems to fit how the language designer thinks you should write programs, Perl allows you to solve your problems as you see fit.

Perl is a language which can grow with you. You can write useful programs with the knowledge that you can learn in an hour of reading this book. Yet if you take the time to understand the philosophies behind the syntax, semantics, and design of the language, you can be far more productive.

First, you need to know how to learn more.

Perldoc

One of Perl's most useful and least appreciated features is the `perldoc` utility. This program is part of every complete Perl 5 installationYou may have to install an additional package on a free GNU/Linux distribution or another Unix-like system; on Debian and Ubuntu this is `perl-doc`.. It displays the documentation of every Perl module installed on the system--whether a core module or one installed from the Comprehensive Perl Archive Network (CPAN)--as well as thousands of pages of Perl's copious core documentation.

If you prefer an online version, http://perldoc.perl.org/ hosts recent versions of the Perl documentation. http://search.cpan.org/ displays the documentation of every module on the CPAN. Windows users, both ActivePerl and Strawberry Perl provide a link in your Start menu to the documentation.

The default behavior of `perldoc` is to display the documentation for a named module or a specific section of the core documentation:

```
    $ B<perldoc List::Util>
    $ B<perldoc perltoc>
    $ B<perldoc Moose::Manual>
```

The first example extracts documentation written for the `List::Util` module and displays it in a form appropriate for your screen. Community standards for CPAN modules (*cpan*) suggest that additional libraries use the same documentation format and form as core modules, so there's no distinction between reading the documentation for a core library such as `Data::Dumper` or one installed from the CPAN. The standard documentation template includes a description of the module, demonstrates sample uses, and then contains a detailed explanation of the module and its interface. While the amount of documentation varies by author, the form of the documentation is remarkably consistent.

The second example displays a pure documentation file, in this case the table of contents of the core documentation itself. This file describes each individual piece of the core documentation; browse it for a good understanding of Perl's breadth.

The third example resembles the second; `Moose::Manual` is part of the Moose CPAN distribution (*moose*). It is also purely documentation; it contains no code.

Similarly, `perldoc perlfaq` will display the table of contents for Frequently Asked Questions about Perl 5. Skimming these questions is invaluable.

The `perldoc` utility has many more abilities (see `perldoc perldoc`). Two of the most useful are the `-q` and the `-f` flags. The `-q` flag takes a keyword or keywords and searches only the Perl FAQ, displaying all results. Thus `perldoc -q sort` returns three questions: *How do I sort an array by (anything)?*, *How do I sort a hash (optionally by value instead of key)?*, and *How can I always keep my hash sorted?*.

The `-f` flag displays the core documentation for a builtin Perl function. `perldoc -f sort` explains the behavior of the `sort` operator. If you don't know the name of the function you want, use `perldoc perlfunc` to see a list of functions.

`perldoc perlop` and `perldoc perlsyn` document Perl's symbolic operators and syntactic constructs; `perldoc perldiag` explains the meanings of Perl's warning messages.

Perl 5's documentation system is *POD*, or *Plain Old Documentation*. `perldoc perlpod` describes how POD works. The `perldoc` utility will display the POD in any Perl module you create and install for your project, and other POD tools such as `podchecker`, which validates the form of your POD, and `Pod::Webserver`, which displays local POD as HTML through a minimal webserver, will handle valid POD correctly.

`perldoc` has other uses. With the `-l` command-line flag, it displays the *path* to the documentation file rather than the contents of the documentationBe aware that a module may have a separate *.pod* file in addition to its *.pm* file.. With the `-m` flag, it displays the entire *contents* of the module, code and all, without processing any POD instructions.

Expressivity

Before Larry Wall created Perl, he studied linguistics and human languages. His experiences continue to influence Perl's design. There are many ways to write a Perl program depending on your project's style, the available time to create the program, the expected maintenance burden, or even your own personal sense of expression. You may write in a straightforward, top-to-bottom style. You may write many small and independent functions. You may model your problem with classes and objects. You may eschew or embrace advanced features.

Perl hackers have a slogan for this: *TIMTOWTDI*, pronounced "Tim Toady", or "There's more than one way to do it!"

Where this expressivity can provide a large palette with which master craftsman can create amazing and powerful edifices, unwise conglomerations of various techniques can impede maintainability and comprehensibility. You can write good code or you can make a mess. The choice is yours... but be kind to other people, if you must make a mess..

Where other languages might suggest that one enforced way to perform any operation is the right solution, Perl allows you to optimize for your most important criteria. Within the realm of your own problems, you can choose from several good approaches--but be mindful of readability and future maintainability.

As a novice to Perl, you may find certain constructs difficult to understand. The greater Perl community has discovered and promoted several idioms (*idioms*) which offer great power. Don't expect to understand them immediately. Some of Perl's features interact in subtle ways.

Learning Perl is like learning a second or third spoken language. You'll learn a few words, then string together some sentences, and eventually will be able to have small, simple conversations. Mastery comes with practice, both reading and writing. You don't have to understand all of the details of this chapter immediately to be productive with Perl. Keep the principles as you read the rest of this book.

Another design goal of Perl is to surprise experienced (Perl) programmers very little. For example, adding two scalars together with a numeric operator (`$first_num + $second_num`) is obviously a numeric operation; the operator must treat both scalars as numeric values to produce a numeric result. No matter what the contents of `$first_num` and `$second_num`, Perl will coerce them to numeric values (*numeric_coercion*) without requiring the user or programmer to perform this conversion manually. You've expressed your intent to treat them as numbers by choosing a numeric operator (*numeric_operators*), so Perl happily handles the rest.

In general, Perl programmers can expect Perl to do what you mean; this is the notion of *DWIM--do what I mean*. You may also see this mentioned as the *principle of least astonishment*. Given a cursory understanding of Perl (especially its *context_philosophy*), it should be possible to read a single unfamiliar Perl expression and understand its intent.

If you're new to Perl, you will develop this skill over time. The flip side of Perl's expressivity is that Perl novices can write useful programs before they learn all of Perl's powerful features. The Perl community often refers to this as *baby Perl*. Though it may sound dismissive, please don't take offense; everyone was a novice once. Take the opportunity to learn from more experienced programmers and ask for explanations of idioms and constructs you don't yet understand.

A Perl novice might multiply a list of numbers by three by writing:

```
my @tripled;
my $count = @numbers;

for (my $i = 0; $i < $count; $i++)
{
    $tripled[$i] = $numbers[$i] * 3;
}
```

A Perl adept might write:

```
my @tripled;

for my $num (@numbers)
{
    push @tripled, $num * 3;
}
```

An experienced Perl hacker might write:

```
my @tripled = map { $_ * 3 } @numbers;
```

Experience writing Perl will help you to focus on what you want to do rather than how to do it.

Perl is a language intended to grow with your understanding of programming. It won't punish you for writing simple programs. It allows you to refine and expand programs for clarity, expressivity, reuse, and maintainability. Take advantage of this philosophy. It's more important to accomplish your task well than to write a conceptually pure and beautiful program.

The rest of this book demonstrates how to use Perl to your advantage.

Context

Spoken languages have a notion of *context* where the correct usage or meaning of a word or phrase depends on its surroundings. You may understand this in a spoken language, where the inappropriate pluralization of "Please give me one hamburgers!"The pluralization of the noun differs from the amount. sounds wrong or the incorrect gender of "la gato"The article is feminine, but the noun is masculine. makes native speakers chuckle.

Context in Perl is similar; the language understands expectations of the amount of data to provide as well as what kind of data to provide. Perl will happily attempt to provide exactly what you ask for.

One type of context in Perl means that certain operators have different behavior if you want zero, one, or many results. It's possible that a specific construct in Perl will do something different if you say "Fetch me zero results; I don't care about any" than if you say "Fetch me one result" or "Fetch me many results."

Likewise, certain contexts make it clear that you expect a numeric value, a string value, or a value that's either true or false.

Context can be tricky if you try to write or read Perl code as a series of single expressions which stand apart from from their environments. You may find yourself slapping your forehead after a long debugging session when you discover that your assumptions about context were incorrect. However, if you're cognizant of contexts, they can make your code clearer, more concise, and more flexible.

Void, Scalar, and List Context

One of the aspects of context governs *how many* items you expect. This is *amount context*. Compare this context to subject-verb number agreement in English. Even if you haven't learned the formal description of the rule, you probably understand the error in the sentence "Perl are a fun language". The rule in Perl is that the number of items you request determines how many you get.

Suppose you have a function (*functions*) called `some_expensive_operation()` which performs an expensive calculation and can produce many, many results. If you call the function on its own and never use its return value, you've called the function in *void context*:

```
some_expensive_operation();
```

Assigning the function's return value to a single element evaluates the function in *scalar context*:

```
my $single_result = some_expensive_operation();
```

Assigning the results of calling the function to an array (*arrays*) or a list, or using it in a list evaluates the function in *list context*:

```
my @all_results        = some_expensive_operation();
my ($single_element)   = some_expensive_operation();
process_list_of_results( some_expensive_operation() );
```

The second line of the previous example may look confusing; the parentheses there give a hint to the compiler that although there's only a single scalar, this assignment should occur in list context. It's semantically equivalent to assigning to a scalar and a temporary array, and then throwing away the array, except that no assignment to the array actually occurs:

```
my ($single_element, @rest) = some_expensive_operation();
```

Why is context interesting for functions? Suppose `some_expensive_operation()` calculates all of the tasks you have to do around the house, sorted in order of their priority. If you have only time to do one task, you can call the function in scalar context to get a useful task--perhaps not necessarily the most important, but important enough without having to go through your whole task list. In list context, the function can perform all of the sorting and searching and comparison and give you an exhaustive list in the proper order. If you want all of that work, but only have time for a couple of tasks, you can use a one or two-element list.

Evaluating a function or expression--except for assignment--in list context can produce confusion. Lists propagate list context to the expressions they contain. Both calls to `some_expensive_operation()` occur in list context:

```
process_list_of_results( some_expensive_operation() );

my %results =
(
    cheap_operation     => $cheap_operation_results,
    expensive_operation => some_expensive_operation(), # OOPS!
);
```

The latter example often surprises novice programmers who expect scalar context for the call. Use the `scalar` operator to enforce it:

```
my %results =
(
    cheap_operation     => $cheap_operation_results,
    expensive_operation => scalar some_expensive_operation(),
);
```

Numeric, String, and Boolean Context

Another type of context determines how Perl understands a piece of data--not *how many* pieces of data you want, but what the data means. You've probably already noticed that Perl's flexible about figuring out if you have a number or a string and converting between the two as you want them. This *value context* helps to explain how it does so. In exchange for not having to declare (or at least track) explicitly what *type* of data a variable contains or a function produces, Perl offers specific type contexts that tell the compiler how to treat a given value during an operation.

Suppose you want to compare the contents of two strings. The `eq` operator tells you if the strings contain the same information:

```
say "Catastrophic crypto fail!" if $alice eq $bob;
```

You may have had a baffling experience where you *know* that the strings are different, but they still compare the same:

```
my $alice = 'alice';
say "Catastrophic crypto fail!" if $alice == 'Bob';  # OOPS
```

The `eq` operator treats its operands as strings by enforcing *string context* on them. The `==` operator enforces *numeric context*. The example code fails because the value of both strings when treated as numbers is `0`.

Boolean context occurs when you use a value in a conditional statement. In the previous examples, the `if` statement evaluated the results of the `eq` and `==` operators in boolean context.

Perl will do its best to coerce values to the proper type (*coercion*), depending on the operators you use. Be sure to use the proper operator for the type of context you want.

In rare circumstances, you may need to force an explicit context where no appropriately typed operator exists. To force a numeric context, add zero to a variable. To force a string context, concatenate a variable with the empty string. To force a boolean context, double the negation operator:

```
    my $numeric_x =  0 + $x;  # forces numeric context
    my $stringy_x = '' . $x;  # forces string  context
    my $boolean_x =    !!$x;  # forces boolean context
```

In general, type contexts are less difficult to understand and see than the amount contexts. Once you understand that they exist and know which operators provide which contexts (*operator_types*), you'll rarely make mistakes with them.

Implicit Ideas

Like many spoken languages, Perl provides linguistic shortcuts. Context is one such feature. Both the compiler and a programmer reading the code can understand the expected number of results or the type of an operation from existing information without adding additional explicit information to disambiguate. Others also exist.

The Default Scalar Variable

The default scalar variable, `$_`, is the best example of a linguistic shortcut in Perl. It's most notable in its *absence*: many of Perl's built in operations perform their work on the contents of `$_` in the absence of an explicit variable. You can still use `$_` as the variable, but it's often unnecessary.

For example, the `chomp` operator removes any trailing newline sequence from the given string:

```
    my $uncle = "Bob\n";
    say "'$uncle'";
    chomp $uncle;
    say "'$uncle'";
```

Without an explicit variable, `chomp` removes the trailing newline sequence from `$_`. Thus, these two lines of code are equivalent:

```
    chomp $_;
    chomp;
```

Similarly, the `say` and `print` builtins operate on `$_` in the absence of other arguments:

```
    print;  # prints $_ to the currently selected filehandle
    say;    # prints $_ to the currently selected filehandle
            # with a trailing newline
```

Perl's regular expression facilities (*regular_expressions*) can also operate on $_, performing matches, substitutions, and transliterations:

```
$_ = 'My name is Paquito';
say if /My name is/;

s/Paquito/Paquita/;

tr/A-Z/a-z/;
say;
```

Check how many other string functions this includes?

Many of Perl's scalar operators work on the default scalar variable if you do not provide an alternative.

Perl's looping directives (*looping_directives*) also set $_, such as `for` iterating over a list:

```
say "#B<$_>" for 1 .. 10;

for (1 .. 10)
{
    say "#B<$_>";
}
```

... or `while`:

```
while (<STDIN>)
{
    chomp;
    say scalar reverse;
}
```

... or `map` transforming a list:

```
my @squares = map { B<$_> * B<$_> } 1 .. 10;
say for @squares;
```

... or `grep` filtering a list:

```
say 'Brunch time!' if grep { /pancake mix/ } @pantry;
```

If you call functions within code that uses $_ whether implicitly or explicitly, they may overwrite the value of $_. Similarly, if you write a function which uses $_, you may clobber a caller function's use of $_. Perl 5.10 allows you to use `my` to declare $_ as a lexical variable, which prevents this clobbering behavior. Be wise.

```
while (<STDIN>)
{
    chomp;

    # BAD EXAMPLE
    my $munged = calculate_value( $_ );
    say "Original: $_";
    say "Munged  : $munged";
}
```

In this example, if `calculate_value()` or any other function it happened to call changed $_, it would remain changed throughout the `while` loop. Adding a `my` declaration prevents that behavior:

```
while (my $_ = <STDIN>)
{
    ...
}
```

Of course, using a named lexical can be just as clear:

```
while (my $line = <STDIN>)
{
    ...
}
```

Use $_ as you would the word "it" in formal writing: sparingly, in small and well-defined scopes.

The Default Array Variables

While Perl has a single implicit scalar variable, it has two implicit array variables. Perl passes arguments to functions in an array named @_. Array manipulation operations (*arrays*) performed inside functions affect this array by default. Thus, these two snippets of code are equivalent:

```
sub foo
{
    my $arg = shift;
    ...
}

sub foo_explicit
{
    my $arg = shift @_;
    ...
}
```

Unlike $_, Perl automatically localizes @_ for you when you call other functions. The array operators `shift` and pop operate on @_ with no other operands provided.

Outside of all functions, the default array variable @ARGV contains the command-line arguments to the program. The same array operators which use @_ implicitly *within* functions use @ARGV implicitly outside of functions. You cannot use @_ when you mean @ARGV.

ARGV has one special case. If you read from the null filehandle <>, Perl will treat every element in @ARGV as the *name* of a file to open for reading. (If @ARGV is empty, Perl will read from standard input.) This implicit @ARGV behavior is useful for writing short programs, such as this command-line filter which reverses its input:

```
while (<>)
{
    chomp;
    say scalar reverse;
}
```

Why `scalar`? say imposes list context on its operands. `reverse` passes its context on to its operands, treating them as a list in list context and a concatenated string in scalar context. If this sounds confusing, it can be: if Perl 5 arguably should have had different operators for these different operations.

If you run it with a list of files:

```
    $ B<perl reverse_lines.pl encrypted/*.txt>
```

... the result will be one long stream of output. Without any arguments, you can provide your own standard input by piping in from another program or typing directly.