Perl gets things done--it's flexible, forgiving, and malleable. Capable programmers use it every day for everything from one-liners and one-off automations to multi-year, multi-programmer projects.

Perl is pragmatic. You're in charge. You decide how to solve your problems and Perl will mold itself to do what you mean, with little frustration and no ceremony.

Perl will grow with you. In the next hour, you'll learn enough to write real, useful programs--and you'll understand *how* the language works and *why* it works as it does. Modern Perl takes advantage of this knowledge and the combined experience of the global Perl community to help you write working, maintainable code.

First, you need to know how to learn more.

Perldoc

Perl respects your time; Perl culture values documentation. The language ships with thousands of pages of core documentation. The `perldoc` utility is part of every complete Perl installation. Your OS may provide this as an additional package; install `perl-doc` on Debian or Ubuntu GNU/Linux, for example. `perldoc` can display the core docs as well as the documentation of every Perl module you have installed--whether a core module or one installed from the Comprehensive Perl Archive Network (CPAN).

http://perldoc.perl.org/ hosts recent versions of the Perl documentation. CPAN indexes at http://search.cpan.org/ and http://metacpan.org/ provide documentation for all CPAN modules. Other distributions such as ActivePerl and Strawberry Perl provide local documentation in HTML formats.

Use `perldoc` to read the documentation for a module or part of the core documentation:

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

The first example displays the documentation of the `List::Util` module; these docs are in the module itself. The second example is the table of contents of the core docs. This file is purely documentation. The third example requires you to install the `Moose` (*moose*) CPAN distribution; it displays the pure-documentation manual. `perldoc` hides these all of these details for you; there's no distinction between reading the documentation for a core library such as `Data::Dumper` or one installed from the CPAN. Perl culture values documentation so much that even external libraries follow the good example of the core language documentation.

The standard documentation template includes a description of the module, sample uses, and 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.

Perl has lots of documentation. Where do you start?

`perldoc perltoc` displays the table of contents of the core documentation, and `perldoc perlfaq` is the table of contents for Frequently Asked Questions about Perl. `perldoc perlop` and `perldoc perlsyn` document Perl's symbolic operators and syntactic constructs. `perldoc perldiag` explains the meanings of Perl's warning messages. `perldoc perlvar` lists all of Perl's symbolic variables.

You don't have to memorize anything in these docs. Skim them for a great overview of the language and come back to them when you have questions.

The `perldoc` utility can do much, much more (see `perldoc perldoc`). Use the `-q` option with a keyword to search the Perl FAQ. For example, `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` option shows the documentation for a builtin Perl function, such as `perldoc -f sort`. If you don't know the name of the function you want, browse the list of available builtins in `perldoc perlfunc`.

The `-v` option looks up a builtin variable. For example, `perldoc -v $PID` explains `$PID`, which is the variable containing the current program's process id. Depending on your shell, you may have to quote the variable appropriately.

The `-l` option shows the *path* to the file containing the documentation. (A module may have a separate *.pod* file in addition to its *.pm* file.)

The `-m` option displays the entire *contents* of the module, code and all, without any special formatting.

Perl uses a documentation format called *POD*, short for *Plain Old Documentation*. `perldoc perlpod` describes how POD works. Other POD tools include `podchecker`, which validates the structure of POD documents, and the `Pod::Webserver` CPAN module, which displays local POD as HTML through a minimal web server.

Expressivity

Before Larry Wall created Perl, he studied linguistics. Unlike other programming languages designed around a mathematical notion, Perl's design emulates how people communicate with people. This gives you the freedom to write programs depending on your current needs. You may write simple, straightforward code or combine many small pieces into larger programs. You may select from multiple design paradigms, and you may eschew or embrace advanced features.

Learning Perl is like learning any spoken language. You'll learn a few words, then string together sentences, and then enjoy simple conversations. Mastery comes from practice of both reading and writing code. You don't have to understand every detail of Perl to be productive, but the principles in this chapter are essential to your growth as a programmer.

Other languages may claim that there should be only one best way to solve any problem. Perl allows *you* to decide what's most readable, most useful, most appealing, or most fun.

Perl hackers call this *TIMTOWTDI*, pronounced "Tim Toady", or "There's more than one way to do it!"

This expressivity allows master craftworkers to create amazing programs but also allows the unwary to make messes. You'll develop your own sense of good taste with experience. Express yourself, but be mindful of readability and maintainability, especially for those who come after you.

Perl novices often find certain syntactic constructs opaque. These idioms (*idioms*) offer great (if subtle) power to experienced programmers, but it's okay to avoid them until you're comfortable with them.

As another design goal, Perl tries to avoid surprising experienced (Perl) programmers. For example, adding two variables (`$first_num + $second_num`) is obviously a numeric operation (*numeric_operators*). You've expressed your intent to treat the values of those variables as numbers by using a numeric operator. Perl happily does so. No matter the contents of `$first_num` and `$second_num`, Perl will coerce them to numeric values (*numeric_coercion*).

Perl adepts often call this principle *DWIM*, or *do what I mean*. You could just as well call this the *principle of least astonishment*. Given a cursory understanding of Perl (especially context; *context_philosophy*), it should be possible to understand the intent of an unfamiliar Perl expression. You will develop this skill as you learn Perl.

Perl's expressivity allows novices to write useful programs without having to understand the entire language. This is by design! Experienced developers often call the results *baby Perl* as a term of endearment. Everyone begins as a novice. Through practice and learning from more experienced programmers, you will understand and adopt more powerful idioms and techniques. It's okay for you to write simple code that you understand. Keep practicing and you'll become a native speaker.

A novice Perl hacker might triple a list of numbers with:

```
my @tripled;

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

... and a Perl adept might write:

```
my @tripled;

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

... while an experienced Perl hacker could write:

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

Every program gets the same result. Each uses Perl in a different way.

As you get more comfortable with Perl, you can let the language do more for you. With experience, you can focus on *what* you want to do rather than *how* to do it. Perl doesn't care if you write baby or expert code. Design and refine your programs for clarity, expressivity, reuse, and maintainability, in part or in whole. Take advantage of this flexibility and pragmatism: it's far better to accomplish your task effectively now than to write a conceptually pure and beautiful program next year.

Context

In spoken languages, the meaning of a word or phrase depends on how you use it; the local *context* of other grammatical constructs helps clarify the intent. For example, the inappropriate pluralization of "Please give me one hamburgers!" sounds wrong (the pluralization of the noun differs from the amount), just as the incorrect gender of "la gato" (the article is feminine, but the noun is masculine) makes native speakers chuckle. Some words do double duty; one sheep is a sheep just as two sheep are also sheep and you program a program.

Perl uses context to express how to treat a piece of data. This governs the *amount* of data as well as the *kind* of data. For example, several Perl operations produce different behaviors when you expect zero, one, or many results. A specific construct in Perl may do something different if you write "Do this, but I don't care about any results" compared to "Do this and give me multiple results." Other operations allow you to specify whether you expect to work with numeric, textual, or true or false data.

You must keep context in mind when you read Perl code. Every expression is part of a larger context. You may find yourself slapping your forehead after a long debugging session when you discover that your assumptions about context were incorrect. If instead you're aware of context, your code will be more correct--and cleaner, flexible, and more concise.

Void, Scalar, and List Context

Amount context governs *how many* items you expect an operation to produce. Think of subject-verb number agreement in English. Even without knowing the formal description of this principle, you probably understand the error in the sentence "Perl are a fun language." (In terms of amount context, you could say that the verb "are" expects a plural noun or noun phrase.) In Perl, the number of items you request influences how many you receive.

Suppose the function (*functions*) called `find_chores()` sorts your household todo list in order of priority. The number of chores you expect to read from your list influences what the function produces. If you expect nothing, you're just pretending to be busy. If you expect one task, you have something to do for the next fifteen minutes. If you have a burst of energy on a free weekend, you could get all of your chores.

Why does context matter? A context-aware function can examine its calling context and decide how much work it must do. When you call a function and never use its return value, you've used *void context*:

```
find_chores();
```

Assigning the function's return value to a single item (*scalars*) enforces *scalar context*:

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

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             = find_chores();
my ($single_element, @rest) = find_chores();

# list of results passed to a function
process_list_of_results( find_chores() );
```

The parentheses in the second line of the previous example group the two variable declarations (*lexical_scope*) into a single unit so that assignment assigns to both of the variables. A single-item list is still a list, though. You could also correctly write:

```
my B<(>$single_elementB<)>   = find_chores();
```

.... in which case the parentheses tell Perl parser that you intend list context for the single variable `$single_element`. This is subtle, but now that you know about it, the difference of amount context between these two statements should be obvious:

```
my $scalar_context = find_chores();
my B<(>$list_contextB<)> = find_chores();
```

Lists propagate list context to the expressions they contain. This often confuses novices until they understand it. Both of these calls to `find_chores()` occur in list context:

```
process_list_of_results( find_chores() );

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

Yes, initializing a hash (*hashes*) with a list of values imposes list context on `find_chores`. Use the `scalar` operator to impose scalar context:

```
my %results = (
    cheap_operation     => $cheap_results,
    expensive_operation => B<scalar> find_chores(),
);
```

Again, context can help you determine how much work a function should do. In void context, `find_chores()` may legitimately do nothing. In scalar context, it can find only the most important task. In list context, it must sort and return the entire list.

Numeric, String, and Boolean Context

Perl's other context--*value context*--influences how Perl interprets a piece of data. Perl can figure out if you have a number or a string and convert data between the two types. In exchange for not having to declare explicitly what *type* of data a variable contains or a function produces, Perl's value contexts provide hints about how to treat that data.

Perl will coerce values to specific proper types (*coercion*) depending on the operators you use. For example, the `eq` operator tests that two values contain equivalent string values:

```
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';
```

The `eq` operator treats its operands as strings by enforcing *string context* on them, but the `==` operator imposes *numeric context*. In numeric context, both strings evaluate to `0` (*numeric_coercion*). Be sure to use the proper operator for your desired value context.

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

In rare circumstances, you may not be able to use the appropriate operator to enforce value context. 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 up 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
```

Value contexts are easier to identify than amount contexts. Once you know which operators provide which contexts (*operator_types*), you'll rarely make mistakes.

Implicit Ideas

Perl code can seem dense at first, but it's full of linguistic shortcuts. These allow experienced programmers to glance at code and understand its important implications. Context is one shortcut. Another is default variables--the programming equivalent of pronouns.

The Default Scalar Variable

The *default scalar variable* (or *topic variable*), $_, is most notable in its *absence*: many of Perl's builtin operations work on the contents of $_ in the absence of an explicit variable. You can still type $_ if it makes your code clearer to you, but it's often unnecessary.

Many of Perl's scalar operators (including `chr`, `ord`, `lc`, `length`, `reverse`, and `uc`) work on the default scalar variable if you do not provide an alternative. For example, the `chomp` builtin removes any trailing newline sequence (technically the contents of `$/`; see `perldoc -f chomp`) from its operand:

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

$_ behaves the same way in Perl as the pronoun *it* does in English. Without an explicit variable, `chomp` removes the trailing newline sequence from $_. When you write "`chomp;`", Perl will always chomp *it*. These two lines of code are equivalent:

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

`say` and `print` also operate on $_ in the absence of other arguments:

```
print;  # prints $_ to the current filehandle
say;    # prints $_ and a newline to the current filehandle
```

Perl's regular expression facilities (*chp.regex*) default to $_ to match, substitute, and transliterate:

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

s/Paquito/Paquita/;
```

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

Perl's looping directives (*looping_directives*) default to using `$_` as the iteration variable, whether `for` iterating over a list:

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

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

... or `while` waiting for an expression to evaluate to false:

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

... or `map` transforming a list:

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

... or `grep` filtering a list:

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

Just as English gets confusing when you have too many pronouns and antecedents, so does Perl when you mix explicit and implicit uses of `$_`. In general, there's only one `$_`. If you use it in multiple places, one operator's `$_` may override another's. For example, if one function uses `$_` and you call it from another function which uses `$_`, the callee may clobber the caller's value:

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

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

If `calculate_value()` or any other function changed `$_`, that change would persist through that iteration of the loop. Using a named lexical is safer and may be clearer:

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

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

The triple-dot (`...`) operator is a placeholder for code you intend to fill in later. Perl will parse it as a complete statement, but will throw an exception that you're trying to run unimplemented code if you try to run it. See `perldoc perlop` for more details.

The Default Array Variables

Perl also provides two implicit array variables. Perl passes arguments to functions (*functions*) in an array named `@_`. Array operations (*arrays*) inside functions use this array by default. These two snippets of code are equivalent:

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

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

Just as `$_` corresponds to the pronoun *it*, `@_` corresponds to the pronouns *they* and *them*. *Unlike* `$_`, each function has a separate copy of `@_`. The builtins `shift` and `pop` operate on `@_`, if provided no explicit operands.

Outside of all functions, the default array variable `@ARGV` contains the command-line arguments provided to the program. Perl's array operations (including `shift` and `pop`) operate on `@ARGV` implicitly outside of functions. You cannot use `@_` when you mean `@ARGV`.

Perl's `<$fh>` operator is the same as the `readline` builtin. `readline $fh` does the same thing as `<$fh>`. A bare `readline` behaves just like `<>`. For historic reasons, `<>` is still more common, but consider using `readline` as a more readable alternative. (What's more readable, `glob '*.html'` to `<*.html>`? The same idea applies.)

`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; see *filehandle*.) This implicit `@ARGV` behavior is useful for writing short programs, such as a command-line filter which reverses its input:

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

Perl 5.22 made this expression a little safer with the `<<>>` operator. If a filename provided contains a special punctuation symbol like `|filename` or `filename|`, Perl would do something special with it. The double-diamond operator avoids this behavior.

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 the

behavior of `reverse` sounds confusing, your instincts are correct. Perl arguably should have separated "reverse a string" from "reverse a list".

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. That's a lot of flexibility in a small program--and you're only getting started.