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 has a culture of useful documentation. The `perldoc` utility is part of every complete Perl 5 installationHowever your Unix-like system may require you to install an additional package such as `perl-doc` on Debian or Ubuntu GNU/Linux.. `perldoc` 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.

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 Perl 5 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 embedded within the `List::Util` module. The second example displays a pure documentation file, in this case the table of contents of the core documentation. The third example displays a pure documentation file included as part of a CPAN distribution (*moose*). `perldoc` hides these details; 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.

`perldoc perltoc` displays the table of contents of the core documentation, and `perldoc perlfaq` displays the table of contents for Frequently Asked Questions about Perl 5. `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. Skimming these files will give you a great overview of Perl 5.

The `perldoc` utility has many more abilities (see `perldoc perldoc`). The `-q` option searches only the Perl FAQ for any provided keywords. 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` option displays the 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, browse the list of

available builtins in `perldoc perlfunc`.

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

The `-l` option causes `perldoc` to display 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..

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

Perl 5's documentation system is *POD*, or *Plain Old Documentation*. `perldoc perlpod` describes how POD works. Other POD tools include `podchecker`, which validates the form of your POD, and `Pod::Webserver`, which displays local POD as HTML through a minimal web server.

Expressivity

Larry Wall's his studies of linguistics and human languages influenced the design of Perl. The language allows you tremendous freedom to solve your problems, depending on your group style, the available time, the expected lifespan of the program, or even how creative you feel. You may write simple, straightforward code or integrate into larger, well-defined programs. You may select from multiple design paradigms, and you may eschew or embrace advanced features.

Where other languages enforce one best way to write any code, Perl allows *you* to decide what's most readable or useful or fun.

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

Though this expressivity allows master craftworkers to create amazing programs, it allows the unwise or uncautious to make messes. Experience and good taste will guide you to write great code. The choice is yours--but be mindful of readability and maintainability, especially for those who come after you.

Perl novices often may find certain constructs opaque. Many of these idioms (*idioms*) offer great (if subtle) power. It's okay to avoid them until you're comfortable with them.

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

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

Perl adepts often call this principle *DWIM*, or *do what I mean*. Another phrasing is that Perl follows 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.

Perl's expressivity also allows novices to write useful programs without having to understand everything. The resulting code is often called *baby Perl*, in the sense that most everyone wants to help babies learn to communicate well. Everyone begins as a novice. Through practice and learning from more experienced programmers, you will understand and adopt more powerful idioms and techniques.

For example, an experienced Perl hacker might triple a list of numbers with:

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

... and a Perl adept might write:

```
my @tripled;

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

... while a novice might try:

```
my @tripled;

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

All three approaches accomplish the same thing, but each uses Perl in a different way.

Experience writing Perl will help you to focus on *what* you want to do rather than *how* to do it. Even so, Perl will happily run simple programs. You can 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 may depend on how you use it; the local *context* helps clarify the intent. For example, the inappropriate pluralization of "Please give me one hamburgers!"The pluralization of the noun differs from the amount. sounds wrong, just as the incorrect gender of "la gato"The article is feminine, but the noun is masculine. makes native speakers chuckle. Consider also the pronoun "you" or the noun "sheep" which can be singular or plural depending on context.

Context in Perl is similar. It governs the amount as well as the kind of data to use. Perl will happily attempt to provide exactly what you ask for--provided you do so by choosing the appropriate context.

Certain Perl operations produce different behaviors when you want 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 I expect multiple results." Other operations allow you to specify whether you expect to work with numeric data, textual data, or true or false data.

Context can be tricky if you try to write or read Perl code as a series of single expressions extracted

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. If instead you're cognizant of context, your code will be more correct--and cleaner, flexible, and more concise.

Void, Scalar, and List Context

Amount context context governs *how many* items you expect from an operation. The English language's subject-verb number agreement is a close parallel. Even without knowing the formal description of this linguistic principle, you probably understand the error in the sentence "Perl are a fun language". In Perl, the number of items you request determines how many you get.

Suppose you have a function (*functions*) called `find_chores()` which sorts your household todo list in order of task priority. The means by which you call this function determines what it will produce. You may have no time to do chores, in which case calling the function is an attempt to look industrious. You may have enough time to do one task, or you could have a burst of energy on a free weekend and desire to accomplish as much as possible.

If you call the function on its own and never use its return value, you've called the function in *void context*:

```
find_chores();
```

Assigning the function's return value to a single item (*scalars*) evaluates the function in *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();
process_list_of_results( find_chores() );
```

The parentheses in the second line of the previous example group the two variable declarations (*lexical_scope*) so that assignment will behave as you expect. If `@rest` were to go unused, you could also correctly write:

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

.... in which case the parentheses give a hint to the Perl 5 parser that you intend list context for the assignment even though you assign only one element of a list. 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();
```

Evaluating a function or expression--except for assignment--in list context can produce confusion. Lists propagate list context to the expressions they contain. 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!
);
```

The latter example often surprises novice programmers, as 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(),
);
```

Why does context matter? A context-aware function can examine its calling context and decide how much work it must 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*--governs how Perl interprets a piece of data. 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. In exchange for not having to declare (or at least track) explicitly what *type* of data a variable contains or a function produces, Perl's type contexts provide hints that tell the compiler how to treat data.

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

```
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. The == operator imposes *numeric context*. In numeric context, both strings evaluate to `0` (*numeric_coercion*). Be sure to use the proper operator for the type of context you want.

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 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
```

Type 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

Context is only one linguistic shortcut in Perl. Programmers who understand these shortcuts can glance at code and instantly understand its most important characteristics. Another important linguistic feature is the Perl equivalent of pronouns.

The Default Scalar Variable

The *default scalar variable* (also called the *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 use $_ as the variable, 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 from its operandSee `perldoc -f chomp` and `$/` for more precise details of its behavior.:

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

$_ has the same function in Perl as the pronoun *it* in English. Without an explicit variable, `chomp` removes the trailing newline sequence from $_. Perl understands what you mean when you say "`chomp`"; Perl will always chomp *it*, so these two lines of code are equivalent:

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

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

```
print;  # prints $_ to the current filehandle
say;    # prints "$_\n" to the current filehandle
```

Perl's regular expression facilities (*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. Consider `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;
```

As English gets confusing when you have too many pronouns and antecedents, you must take care mixing uses of $_ implicitly or explicitly. Uncautious simultaneous use of $_ may lead to one piece of code silently overwriting the value written by another. If you write a function which uses $_, you may clobber a caller function's use of $_.

As of Perl 5.10, you may declare $_ as a lexical variable (*lexical_scope*) to prevent this clobbering behavior:

```
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. Adding a `my` declaration prevents clobbering an existing instance of $_:

```
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.

Perl 5.12 introduced the triple-dot (`...`) operator as 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 manipulation operations (*arrays*) inside functions affect this array by default, so 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* $_, Perl automatically localizes @_ for you when you call other functions. The builtins `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. 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>. As of Perl 5.10, a bare `readline` behaves just like <>, so you can now use `readline` everywhere. For historic reasons, <> is still more common, but consider using `readline` as a more readable alternative. You probably prefer `glob '*.html'` to <*.html>, right? It's the same idea.

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 the behavior of `reverse` sounds confusing, your instincts are correct. Perl 5 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. Yet Perl is good for far more than small command-line programs....