Extracted from:

Modern Perl, Fourth Edition

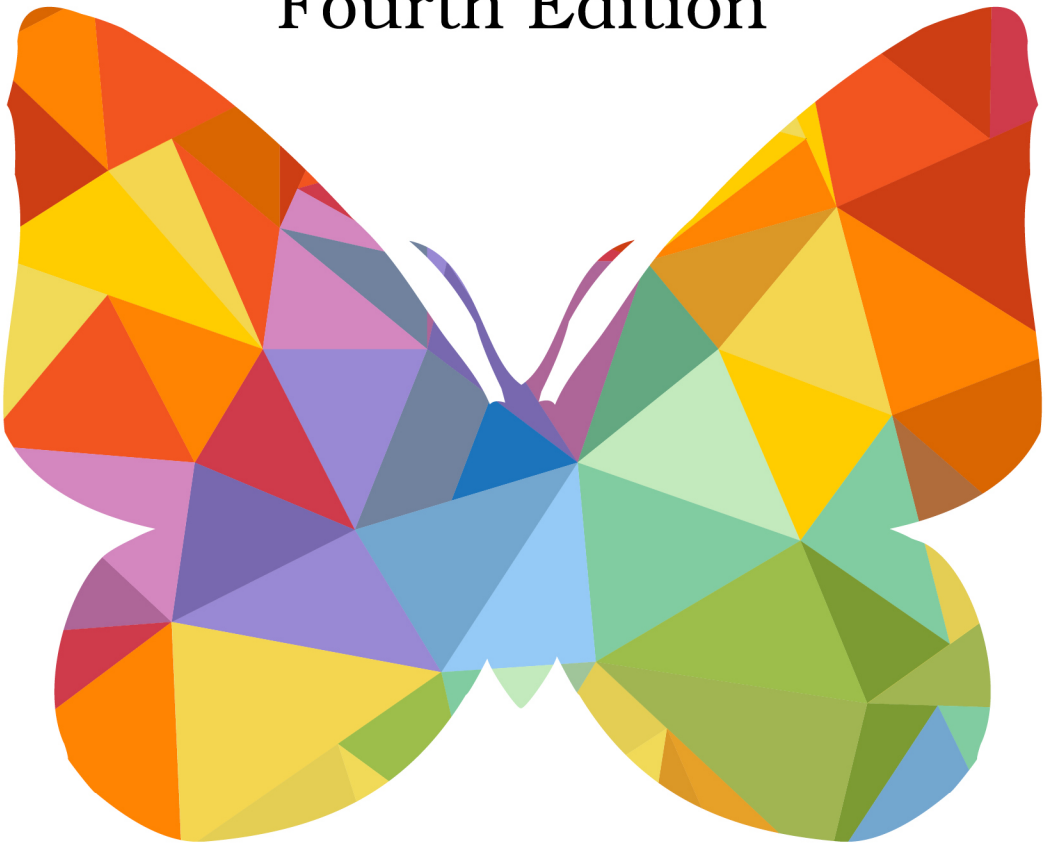
The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Modern Perl

Fourth Edition



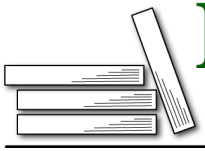
chromatic

The Classic Reference, Updated for Perl 5.22

Modern Perl, Fourth Edition

chromatic

The Pragmatic Bookshelf
Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *https://pragprog.com*.

The team that produced this book includes:

Michael Swaine (editor)
Potomac Indexing, LLC (index)
Linda Recktenwald (copyedit)
Dave Thomas (layout)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-088-2
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—October 2015

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 (Declaring Functions on page ?) 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 on page ?) enforces *scalar context*:

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

Assigning the results of calling the function to an array (Arrays on page ?) 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 on page ?) 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 this:

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

In this case the parentheses tell the 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 $scalar_context = find_chores();
my ($list_context) = 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 on page ?) 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 => 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 on page ?) 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 on page ?). 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 on page ?), you'll rarely make mistakes.