Perl 5 isn't perfect, but it's malleable--in part because no single configuration is ideal for every programmer and every purpose. Some useful behaviors are available as core libraries. More are available from the CPAN. Your effectiveness as a Perl programmer depends on you taking advantage of these enhancements.

Missing Defaults

Perl 5's design process tried to anticipate new directions for the language, but it was as impossible to predict the future in 1994 as it is in 2011. Perl 5 expanded the language, but remained compatible with Perl 1 from 1987.

The best Perl 5 code of 2011 is very different from the best Perl 5 code of 1994, or the best Perl 1 code of 1987.

Although Perl 5 contains an extensive core library, it's not comprehensive. Many of the best Perl 5 modules exist on the CPAN (*cpan*) and not in the core. The `Task::Kensho` meta-distribution includes several other distributions which represent the best the CPAN has to offer. When you need to solve a problem, look there first.

With that said, a few core pragmas and modules are indispensable to serious Perl programmers.

The strict Pragma

The `strict` pragma (*pragmas*) allows you to forbid (or re-enable) various powerful language constructs which offer potential for accidental abuse.

`strict` forbids symbolic references, requires variable declarations (*lexical_scope*), and prohibits the use of undeclared barewords (*barewords*). While the occasional use of symbolic references is necessary to manipulate symbol tables (*import*), the use of a variable as a variable name offers the possibility of subtle errors of action at a distance--or, worse, the possibility of poorly-validated user input manipulating internal-only data for malicious purposes.

Requiring variable declarations helps to detect typos in variable names and encourages proper scoping of lexical variables. It's much easier to see the intended scope of a lexical variable if all variables have `my` or `our` declarations in the appropriate scope.

`strict` has a lexical effect based on the compile-time scope of its use (*import*) and disabling (with `no`). See `perldoc strict` for more details.

The warnings Pragma

The `warnings` pragma (*handling_warnings*) controls the reporting of various classes of warnings in Perl 5, such as attempting to stringify the `undef` value or using the wrong type of operator on values. It also warns about the use of deprecated features.

The most useful warnings explain that Perl had trouble understanding what you meant and had to guess at the proper interpretation. Even though Perl often guesses correctly, disambiguation on your part will ensure that your programs run correctly.

The `warnings` pragma has a lexical effect on the compile-time scope of its use or disabling (with `no`). See `perldoc perllexwarn` and `perldoc warnings` for more details.

Combine `use warnings` with `use diagnostics` to receive expanded diagnostic messages for each warning present in your programs. These expanded diagnostics come from `perldoc perldiag`. This behavior is useful when learning Perl. Disable it before you deploy your program, because it produces verbose output which might fill up your logs and expose too much information to users.

IO::File and IO::Handle

Before Perl 5.14, lexical filehandles were objects of the `IO::Handle` class, but you had to load `IO::Handle` explicitly before you could call methods on them. As of Perl 5.14, lexical filehandles are instances of `IO::File` and Perl loads `IO::File` for you.

Add `IO::Handle` to code running on Perl 5.12 or earlier if you call methods on lexical filehandles.

The autodie Pragma

Perl 5 leaves error handling (or error ignoring) up to you. If you're not careful to check the return value of every `open()` call, for example, you could try to read from a closed filehandle--or worse, lose data as you try to write to one. The `autodie` pragma changes the default behavior. If you write:

```
use autodie;

open my $fh, '>', $file;
```

... an unsuccessful `open()` call will throw an exception. Given that the most appropriate approach to a failed system call is throwing an exception, this pragma can remove a lot of boilerplate code and allow you the peace of mind of knowing that you haven't forgotten to check a return value.

This pragma entered the Perl 5 core as of Perl 5.10.1. See `perldoc autodie` for more information.