Hidden features of Perl?

Asked 16 years, 2 months ago Modified 7 years, 2 months ago

Viewed 63k times

142

votes

Locked. This question and its answers are locked because the question is off-topic but has historical significance. It is not currently accepting new answers or interactions.

What are some really useful but esoteric language features in Perl that you've actually been able to employ to do useful work?

Guidelines:

- Try to limit answers to the Perl core and not CPAN
- Please give an example and a short description

Hidden Features also found in other languages' Hidden Features:

(These are all from Corion's answer)

- C
  - Duff's Device
  - Portability and Standardness

- C#
  - Quotes for whitespace delimited lists and strings
  - Aliasable namespaces
- Java
  - Static Initalizers
- JavaScript
  - Functions are First Class citizens
  - Block scope and closure
  - Calling methods and accessors indirectly through a variable
- Ruby
  - Defining methods through code
- PHP
  - Pervasive online documentation
  - Magic methods
  - Symbolic references
- Python
  - One line value swapping
  - Ability to replace even core functions with your own functionality

Other Hidden Features:

Operators:

- The bool quasi-operator
- The flip-flop operator
  - Also used for list construction
- The `++` and unary `-` operators work on strings
- The repetition operator
- The spaceship operator
- The || operator (and // operator) to select from a set of choices
- The diamond operator
- Special cases of the `m//` operator
- The tilde-tilde "operator"

Quoting constructs:

- The qw operator
- Letters can be used as quote delimiters in q{}-like constructs
- Quoting mechanisms

Syntax and Names:

- There can be a space after a sigil
- You can give subs numeric names with symbolic references
- Legal trailing commas
- Grouped Integer Literals

- hash slices
- Populating keys of a hash from an array

Modules, Pragmas, and command-line options:

- use strict and use warnings
- Taint checking
- Esoteric use of -n and -p
- CPAN
- `overload::constant`
- IO::Handle module
- Safe compartments
- Attributes

Variables:

- Autovivification
- The `$[` variable
- tie
- Dynamic Scoping
- Variable swapping with a single statement

Loops and flow control:

- Magic goto
- `for` on a single variable
- continue clause

- Desperation mode

Regular expressions:

- The `\G` anchor
- `(?{})` and '(??{})` in regexes

Other features:

- The debugger
- Special code blocks such as BEGIN, CHECK, and END
- The `DATA` block
- New Block Operations
- Source Filters
- Signal Hooks
- map (twice)
- Wrapping built-in functions
- The `eof` function
- The `dbmopen` function
- Turning warnings into errors

Other tricks, and meta-answers:

- cat files, decompressing gzips if needed
- Perl Tips

See Also:

- Hidden features of C
- Hidden features of C#
- Hidden features of C++
- Hidden features of Java
- Hidden features of JavaScript
- Hidden features of Ruby
- Hidden features of PHP
- Hidden features of Python
- Hidden features of Clojure

perl **hidden-features**

Share

edited Sep 25, 2017 at 20:53

community wiki
17 revs, 7 users 53%
Adam Bellaire

Most of these features are in everyday use, some occur in the majority of Perl scripts, and most listed under "Other" still stem from other languages, calling these "hidden" changes the intent of the question. – reinierpost May 26, 2010 at 9:24

Comments disabled on deleted / locked posts / reviews

78 Answers

Sorted by: Highest score (default)

54 votes

The flip-flop operator is useful for skipping the first iteration when looping through the records (usually lines) returned by a file handle, without using a flag variable:

```
while(<$fh>)
{
  next if 1..1; # skip first record
  ...
}
```

Run `perldoc perlop` and search for "flip-flop" for more information and examples.

Share

answered Oct 2, 2008 at 12:41

community wiki
John Siracusa

Actually that's taken from Awk, where you can do flip-flop between two patterns by writing pattern1, pattern2
– Bruno De Fraine Oct 2, 2008 at 12:50

15 To clarify, the "hidden" aspect of this is that if either operand to scalar '..' is a constant the value is implicitly compared to the input line number ($.) – Michael Carman Oct 2, 2008 at 13:41

There are many non-obvious features in Perl.

47
votes

For example, did you know that there can be a space after a sigil?

```
$ perl -wle 'my $x = 3; print $ x'
3
```

Or that you can give subs numeric names if you use symbolic references?

```
$ perl -lwe '*4 = sub { print "yes" }; 4->()'
yes
```

There's also the "bool" quasi operator, that return 1 for true expressions and the empty string for false:

```
$ perl -wle 'print !!4'
1
$ perl -wle 'print !!"0 but true"'
1
$ perl -wle 'print !!0'
(empty line)
```

Other interesting stuff: with `use overload` you can overload string literals and numbers (and for example make them BigInts or whatever).

Many of these things are actually documented somewhere, or follow logically from the documented features, but nonetheless some are not very well known.

Update: Another nice one. Below the `q{...}` quoting constructs were mentioned, but did you know that you can use letters as delimiters?

```
$ perl -Mstrict  -wle 'print q bJet another perl hacker.
Jet another perl hacker.
```

Likewise you can write regular expressions:

```
m xabcx
# same as m/abc/
```

Share

edited Aug 23, 2009 at 21:39

community wiki
3 revs, 2 users 92%
moritz

2 “Did you know that there can be a space after a sigil?” I am utterly flabbergasted. Wow. – Aristotle Pagaltzis Oct 2, 2008 at 19:00

1 Cool! !!$undef_var doesn't create a warning. – Axeman Oct 2, 2008 at 20:40

4 I think your example of using letters to delimit strings should be "*Just* another perl hacker" rather than "Jet another perl hacker" =P – Chris Lutz Aug 23, 2009 at 21:58

The worst part is that you can use other things as delimiters, too. Even closing brackets. The following are valid: s}regex}replacement}xsmg; q]string literal]; – Ryan C. Thompson Oct 27, 2009 at 22:03

46
votes

Add support for compressed files via *magic ARGV*:

```
s{
    ^            # make sure to get whole filename
    (
      [^'] +     # at least one non-quote
      \.         # extension dot
      (?:        # now either suffix
          gz
        | Z
       )
    )
    \z           # through the end
}{gzcat '$1' |}xs for @ARGV;
```

(quotes around $_ necessary to handle filenames with shell metacharacters in)

Now the `<>` feature will decompress any `@ARGV` files that end with ".gz" or ".Z":

```
while (<>) {
    print;
}
```

Share

edited Oct 30, 2010 at 21:34

community wiki
4 revs, 3 users 50%
timkay

2 I don't think you need to escape the `|` in the replacement.
– Chris Lutz Aug 23, 2009 at 21:56

I'm staring at this and I can't figure out how it works. At what point is `zcat |` parsed as a command to pipe through?
– Ether Jun 3, 2010 at 3:44

1 @Ether => detecting pipes is a feature of the two argument open, which the diamond operator uses as it opens each file in `@ARGV` – Eric Strom Jun 5, 2010 at 6:29

40
votes

One of my favourite features in Perl is using the boolean `||` operator to select between a set of choices.

```
$x = $a || $b;

# $x = $a, if $a is true.
# $x = $b, otherwise
```

This means one can write:

```
$x = $a || $b || $c || 0;
```

to take the first true value from `$a`, `$b`, and `$c`, or a default of `0` otherwise.

In Perl 5.10, there's also the `//` operator, which returns the left hand side if it's defined, and the right hand side otherwise. The following selects the first *defined* value from `$a`, `$b`, `$c`, or `0` otherwise:

```
$x = $a // $b // $c // 0;
```

These can also be used with their short-hand forms, which are very useful for providing defaults:

```
$x ||= 0;   # If $x was false, it now has a value of 0.

$x //= 0;   # If $x was undefined, it now has a value of zero.
```

Cheerio,

Paul

Share

edited Nov 11, 2010 at 15:09

community wiki
3 revs, 3 users 85%
pjf

4 This is such a common idiom that it hardly qualifies as a "hidden" feature. – Michael Carman Oct 2, 2008 at 13:28

3 shame the pretty printer thinks // is a comment :) – John Ferguson Oct 2, 2008 at 14:31

2 Question, is there a "use feature" to use these new operators, or are they default enabled? I am still leaning Perl 5.10's features. – J.J. Oct 2, 2008 at 15:34

6 // is in there by default, no special tweaks needed. You can also backport it into 5.8.x with the dor-patch... see the

authors/id/H/HM/HMBRAND/ directory on any CPAN mirror. FreeBSD 6.x and beyond does this for you in their perl package. – dland Oct 2, 2008 at 16:46

2 When || or // is combined with do { }, you can encapsulate a more complex assignment, ie $x = $a || do { my $z; 3 or 4 lines of derivation; $z }; – RET Oct 3, 2008 at 1:40

39 votes

The operators ++ and unary - don't only work on numbers, but also on strings.

```
my $_ = "a"
print -$_
```

prints *-a*

```
print ++$_
```

prints *b*

```
$_ = 'z'
print ++$_
```

prints *aa*

Share

answered Oct 2, 2008 at 12:26

community wiki
Leon Timmermans

3 To quote perlvar: "The auto-decrement operator is not magical." So `--` doesn't work on strings. – moritz Oct 2, 2008 at 12:56

"aa" doesn't seem to be the natural element following "z". I would expect the next highest ascii value, which is "{". – Ether Feb 23, 2009 at 23:08

4 Don't ask a programmer what comes after "z"; ask a human. This feature is great for numbering items in a long list. – Barry Brown Mar 18, 2009 at 23:44

17 When new to Perl I implemented this feature myself with the exact z to aa behavior then showed it to a co-worker who laughed and me and said "let me show you something". I cried a bit but learned something. – Copas May 30, 2009 at 3:07

2 @Ether - If you want that, use numbers and autoconvert them to ASCII with `ord()`. Or, write a small class and overload the operators to do it for you. – Chris Lutz Aug 23, 2009 at 22:05

36 votes

As Perl has almost all "esoteric" parts from the other lists, I'll tell you the one thing that Perl can't:

~~The one thing Perl can't do is have bare arbitrary URLs in your code, because the `//` operator is used for regular expressions.~~

Just in case it wasn't obvious to you what features Perl offers, here's a selective list of the maybe not totally obvious entries:

Duff's Device - in Perl

Portability and Standardness - There are likely more computers with Perl than with a C compiler

A file/path manipulation class - File::Find works on even more operating systems than .Net does

Quotes for whitespace delimited lists and strings - Perl allows you to choose almost arbitrary quotes for your list and string delimiters

Aliasable namespaces - Perl has these through glob assignments:

```
*My::Namespace:: = \%Your::Namespace
```

Static initializers - Perl can run code in almost every phase of compilation and object instantiation, from `BEGIN` (code parse) to `CHECK` (after code parse) to `import` (at module import) to `new` (object instantiation) to `DESTROY` (object destruction) to `END` (program exit)

Functions are First Class citizens - just like in Perl

Block scope and closure - Perl has both

Calling methods and accessors indirectly through a variable - Perl does that too:

```
my $method = 'foo';
my $obj = My::Class->new();
$obj->$method( 'baz' ); # calls $obj->foo( 'baz' )
```

Defining methods through code - Perl allows that too:

```
*foo = sub { print "Hello world" };
```

Pervasive online documentation - Perl documentation is online and likely on your system too

Magic methods that get called whenever you call a "nonexisting" function - Perl implements that in the AUTOLOAD function

Symbolic references - you are well advised to stay away from these. They will eat your children. But of course, Perl allows you to offer your children to blood-thirsty demons.

One line value swapping - Perl allows list assignment

Ability to replace even core functions with your own functionality

```
use subs 'unlink';
sub unlink { print 'No.' }
```

or

```
BEGIN{
    *CORE::GLOBAL::unlink = sub {print 'no'}
};

unlink($_) for @ARGV
```

Share edited May 23, 2017 at 12:10

community wiki
5 revs, 3 users 85%
Corion

I'm a fan of Perl's documentation compared to other languages, but I still think that for Regexes and references it could be rationalised a whole lot. e.g. the best primer for regexes is not Perlre, but Perlop – John Ferguson Oct 2, 2008 at 14:43

9 "The one thing Perl can't do is have bare arbitrary URLs in your code, because the // operator is used for regular expressions." - this is utter nonsense. – Account deleted Oct 12, 2008 at 8:58

Thanks for your insight.I've looked at some ways to have a bare http://... URL in Perl code without using a source filter,and didn't find a way.Maybe you can show how this is possible? // is used for regular expressions in Perl versions up to 5.8.x.In 5.10 it's repurposed for defined-or assignment. – Corion Oct 13, 2008 at 14:23

8 Why/where would you *want* bare URLs in your code? I can't think of an example. – castaway Jul 8, 2009 at 6:57

18 Nobody would want that, it's just a Java meme. "foo.com" is the label http: and then "foo.com" in a comment. Some people find this interesting because... they are dumb. – jrockway Oct 10, 2009 at 18:44

35 votes

Autovivification. AFAIK **no other language has it**.

Share

answered Oct 2, 2008 at 13:48

community wiki
J.J.

I had no idea that Python, et al, didn't support this. – skiphoppy Oct 3, 2008 at 16:17

@davidnicol: Really? Can you provide a link? My quick search on google didn't return anything. For those that don't know ECMAscript is the correct name for Javascript. en.wikipedia.org/wiki/ECMAScript – J.J. Oct 4, 2008 at 9:25

1 And there is a module to disable autovivication – Alexandr Ciornii Jun 29, 2009 at 10:06

1 @Gregg Lind - Given that Python automatically creates variables whenever you first assign to them, autovivification would create monstrous problems out of a single typo. – Chris Lutz Aug 23, 2009 at 22:04

3 @tchrist - a = [[x*y for y in xrange(1,11)] for x in xrange(1,11)] – Omnifarious Nov 11, 2010 at 15:25

31 votes

It's simple to quote almost any kind of strange string in Perl.

```
my $url = q{http://my.url.com/any/arbitrary/path/in/the/
```

In fact, the various quoting mechanisms in Perl are quite interesting. The Perl regex-like quoting mechanisms allow you to quote anything, specifying the delimiters. You can use almost any special character like #, /, or open/close characters like (), [], or {}. Examples:

```
my $var  = q#some string where the pound is the final es
my $var2 = q{A more pleasant way of escaping.};
my $var3 = q(Others prefer parens as the quote mechanism
```

Quoting mechanisms:

q : literal quote; only character that needs to be escaped is the end character. qq : an interpreted quote; processes variables and escape characters. Great for strings that you need to quote:

```
my $var4 = qq{This "$mechanism" is broken.  Please infor
about it.};
```

qx : Works like qq, but then executes it as a system command, non interactively. Returns all the text generated from the standard out. (Redirection, if supported in the OS, also comes out) Also done with back quotes (the ` character).

```
my $output  = qx{type "$path"};       # get just the outp
my $moreout = qx{type "$path" 2>&1}; # get stuff on stde
```

qr : Interprets like qq, but then compiles it as a regular expression. Works with the various options on the regex as well. You can now pass the regex around as a variable:

```
sub MyRegexCheck {
    my ($string, $regex) = @_;
    if ($string)
    {
       return ($string =~ $regex);
    }
    return; # returns 'null' or 'empty' in every context
}

my $regex = qr{http://[\w]\.com/([\w]+/)+};
@results = MyRegexCheck(q{http://myurl.com/subpath1/subp
```

qw : A very, very useful quote operator. Turns a quoted set of whitespace separated words into a list. Great for filling in data in a unit test.

```

   my @allowed = qw(A B C D E F G H I J K L M N O P Q R
   my @badwords = qw(WORD1 word2 word3 word4);
   my @numbers = qw(one two three four 5 six seven); # w
   my @list = ('string with space', qw(eight nine), "a $
lists
   my $arrayref = [ qw(and it works in arrays too) ];
```

They're great to use them whenever it makes things clearer. For qx, qq, and q, I most likely use the {} operators. The most common habit of people using qw is usually the () operator, but sometimes you also see qw//.

Share

edited Aug 23, 2009 at 21:47

community wiki
3 revs, 2 users 96%
Robert P

1 I sometimes use qw"" so that syntax highlighters will highlight it correctly. – Brad Gilbert Oct 2, 2008 at 22:47

Works for me in SlickEdit. :) – Robert P Jan 15, 2009 at 0:35

1 @fengshaun, The editors I generally use **do** highlight these correctly. I was referring, in part to the syntax highlighter on StackOverflow. – Brad Gilbert Jun 13, 2010 at 22:40

@Brad Gilbert: Stack Overflow can’t (well, (doesn’t) parse Perl worth diddly squat. ☹ – tchrist Nov 11, 2010 at 15:15

`my $moreout = qx{type "$path" 2>&1};` ... I didn't know you could do that! [TM] – dland Jul 22, 2011 at 8:52

---

27 votes

Not really hidden, but many every day Perl programmers don't know about CPAN. This especially applies to people who aren't full time programmers or don't program in Perl full time.

Share

edited Mar 18, 2009 at 23:26

community wiki
2 revs, 2 users 86%
mpeters

---

27 votes

The "for" statement can be used the same way "with" is used in Pascal:

```
for ($item)
{
    s/&nbsp;/ /g;
    s/<.*?>/ /g;
    $_ = join(" ", split(" ", $_));
}
```

You can apply a sequence of s/// operations, etc. to the same variable without having to repeat the variable name.

NOTE: the non-breaking space above (&nbsp;) has hidden Unicode in it to circumvent the Markdown. Don't copy paste it :)

Share

edited Aug 23, 2009 at 21:43

community wiki
5 revs, 3 users 88%
timkay

And "map" does the same trick as well... map { .... } $item; One advantage of using "for" over "map" would be that you could use next to break out. – draegtun Oct 2, 2008 at 19:39

2 Also, for has the item being manipulated listed before the code doing the manipulating, leading to better readability. – Robert P Apr 3, 2009 at 22:59

@RobertP: That’s quite right. A topicalizer is useful in discourse. – tchrist Nov 11, 2010 at 15:27

26 votes

The quoteword operator is one of my favourite things. Compare:

```
my @list = ('abc', 'def', 'ghi', 'jkl');
```

and

```
my @list = qw(abc def ghi jkl);
```

Much less noise, easier on the eye. Another really nice thing about Perl, that one really misses when writing SQL, is that a trailing comma is legal:

```
print 1, 2, 3, ;
```

That looks odd, but not if you indent the code another way:

```
print
    results_of_foo(),
    results_of_xyzzy(),
    results_of_quux(),
    ;
```

Adding an additional argument to the function call does not require you to fiddle around with commas on previous or trailing lines. The single line change has no impact on its surrounding lines.

This makes it very pleasant to work with variadic functions. This is perhaps one of the most under-rated features of Perl.

Share

answered Oct 2, 2008 at 16:54

community wiki

dland

2 An interesting corner case of Perl's syntax is that the following is valid: for $_ qw(a list of stuff) {...} – ephemient Oct 3, 2008 at 3:21

1 You can even abuse glob syntax for quoting words, as long as you don't use special characters such as *?. So you can write `for (<a list of stuff>) { ... }` – moritz Oct 7, 2008 at 12:24

1 @ephemient: nearly. That only works with lexicals: for my $x qw(a b c) {...} For instance: for $_ qw(a b c) {print} # prints nothing – dland Oct 8, 2008 at 7:43

why add that extra lexical when you can enjoy perl's favourite default? for (qw/a b c d/) { print; } – fengshaun Jun 9, 2010 at 6:08

2 @ephemient, @fengshaun, @moritz, @dland: That's "fixed" in *blead*; see this p5p thread. – tchrist Nov 11, 2010 at 15:21

26 votes

The ability to parse data directly pasted into a **DATA** block. No need to save to a test file to be opened in the program or similar. For example:

```
my @lines = <DATA>;
for (@lines) {
    print if /bad/;
}

__DATA__
some good data
some bad data
more good data
more good data
```

Share

edited Aug 23, 2009 at 21:42

community wiki
2 revs, 2 users 93%
allan

And very useful in little tests! – fengshaun Jun 9, 2010 at 6:13

@peter mortensen how would you have multiple blocks? And how do you end a block? – Toad Oct 1, 2010 at 17:55

@Toad: it is allan's answer (see the revision list). It is better to address that user. Or, as that user has left Stack Overflow, maybe address no one in particular (so a real Perl expert can straighten it out later). – Peter Mortensen Oct 1, 2010 at 18:25

3 @Hai: No it is **not** ugly — in fact, it's precisely the opposite of ugly: it's clean, svelte, minimal, and beautiful; in a word, it's wonderful, and languages without it are a PITA. @peter mortensen, @toad: One answer to how to have multiple data blocks in the same program is to use the Inline::Files module off CPAN. – tchrist Nov 11, 2010 at 15:26

Inline::Files is implemented using source filters. There's also Data::Section that provides multiple inline blocks and does not use source filters. – Prakash K Jan 12, 2012 at 19:09

24 votes

New Block Operations

I'd say the ability to expand the language, creating pseudo block operations is one.

1. You declare the prototype for a sub indicating that it takes a code reference first:

   ```
   sub do_stuff_with_a_hash (&\%) {
       my ( $block_of_code, $hash_ref ) = @_;
       while ( my ( $k, $v ) = each %$hash_ref ) {
           $block_of_code->( $k, $v );
       }
   }
   ```

2. You can then call it in the body like so

   ```
   use Data::Dumper;

   do_stuff_with_a_hash {
       local $Data::Dumper::Terse = 1;
       my ( $k, $v ) = @_;
       say qq(Hey, the key   is "$k"!);
       say sprintf qq(Hey, the value is "%v"!), Dumper

   } %stuff_for
   ;
   ```

(`Data::Dumper::Dumper` is another semi-hidden gem.) Notice how you don't need the `sub` keyword in front of the block, or the comma before the hash. It ends up looking a lot like: `map { } @list`

Source Filters

Also, there are source filters. Where Perl will pass you the code so you can manipulate it. Both this, and the block operations, are pretty much don't-try-this-at-home type of things.

I have done some neat things with source filters, for example like creating a very simple language to check the time, allowing short Perl one-liners for some decision making:

```
perl -MLib::DB -MLib::TL -e 'run_expensive_database_dele
AM_7';
```

`Lib::TL` would just scan for both the "variables" and the constants, create them and substitute them as needed.

Again, source filters can be messy, but are powerful. But they can mess debuggers up something terrible--and even warnings can be printed with the wrong line numbers. I stopped using Damian's Switch because the debugger would lose all ability to tell me where I really was. But I've found that you can minimize the damage by modifying small sections of code, keeping them on the same line.

Signal Hooks

It's often enough done, but it's not all that obvious. Here's a die handler that piggy backs on the old one.

```
my $old_die_handler = $SIG{__DIE__};
$SIG{__DIE__}
    = sub { say q(Hey! I'm DYIN' over here!); goto &$olc
    ;
```

That means whenever some other module in the code wants to die, they gotta come to you (unless someone else does a destructive overwrite on `$SIG{__DIE__}`). And you can be notified that somebody things something is an error.

Of course, for enough things you can just use an `END { }` block, if all you want to do is clean up.

`overload::constant`

You can inspect literals of a certain type in packages that include your module. For example, if you use this in your `import` sub:

```
overload::constant
    integer => sub {
        my $lit = shift;
        return $lit > 2_000_000_000 ? Math::BigInt->new(
    };
```

it will mean that every integer greater than 2 billion in the calling packages will get changed to a `Math::BigInt` object. (See overload::constant).

Grouped Integer Literals

While we're at it. Perl allows you to break up large numbers into groups of three digits and still get a parsable integer out of it. Note `2_000_000_000` above for 2 billion.

Share

edited Aug 23, 2009 at 21:53

community wiki

4 revs, 2 users 95%
Axeman

5 When using $SIG{**DIE**} handlers, its strongly recommended that you inspect $^S to see if your program is actually dying, or just throwing an exception which is going to be caught. Usually you don't want to interfere with the latter. – pjf Oct 2, 2008 at 22:25

The new block is very instructive ! I was thinking It was a language semantic! many thanks. – ZeroCool Jan 3, 2010 at 19:40

An instructive use of the source filter is pdl's NiceSlice (pdl.perl.org/?docs=NiceSlice&title=PDL::NiceSlice) so that one doesn't need to use the `->slice` as a method every time a slice is needed. – Joel Berger May 8, 2011 at 3:08

24 votes

Binary "x" is the repetition operator:

```
print '-' x 80;     # print row of dashes
```

It also works with lists:

```
print for (1, 4, 9) x 3; # print 149149149
```

Share

edited Mar 19, 2010 at 17:01

community wiki
2 revs, 2 users 77%
Bruno De Fraine

This is one reason why Perl has been so popular with hackers. perl -e 'print 0x000 x 25'; – J.J. Oct 2, 2008 at 15:28

4 My favorite use for this is generating placeholders for the last part of an SQL INSERT statement: @p = ('?') x $n; $p = join(", ", @p); $sql = "INSERT ... VALUES ($p)"; – skiphoppy Oct 3, 2008 at 16:20

24 votes

Taint checking. With taint checking enabled, perl will die (or warn, with `-t`) if you try to pass tainted data (roughly speaking, data from outside the program) to an unsafe function (opening a file, running an external command, etc.). It is very helpful when writing setuid scripts or CGIs or anything where the script has greater privileges than the person feeding it data.

Magic goto. `goto &sub` does an optimized tail call.

The debugger.

`use strict` and `use warnings`. These can save you from a bunch of typos.

Share

edited Mar 19, 2010 at 17:03

community wiki
3 revs, 3 users 80%
Glomek

1 Why don't other languages have this feature? This feature used makes perl web scripts an order of magnitude more secure. – Matthew Lock Dec 3, 2009 at 6:25

22 votes

Based on the way the `"-n"` and `"-p"` switches are implemented in Perl 5, you can write a seemingly incorrect program including `}{`:

```
ls |perl -lne 'print $_; }{ print "$. Files"'
```

which is converted internally to this code:

```
LINE: while (defined($_ = <ARGV>)) {
    print $_; }{ print "$. Files";
}
```

Share

edited Oct 9, 2008 at 13:44

community wiki
2 revs, 2 users 95%
Sec

@martin clayton: Why is it called that? – tchrist Nov 11, 2010 at 15:29

@tchrist - because it, supposedly, looks like two people rubbing noses. In profile, if you see what I mean. – martin clayton Nov 11, 2010 at 23:56

18

votes

Let's start easy with the Spaceship Operator.

```
$a = 5 <=> 7;  # $a is set to -1
$a = 7 <=> 5;  # $a is set to 1
$a = 6 <=> 6;  # $a is set to 0
```

Share

answered Oct 2, 2008 at 12:09

community wiki
Sec

1 @Leon: C/C++ doesn't do a 3 value return for numbers. If memory serves String comapre functions are the only 3 value return that I know of in the whole STL language. AFAIK Python doesn't have a 3 return numeric compare. Java doesn't have a number specific 3 return compare either. – J.J. Oct 2, 2008 at 14:53

7 It's worth mentioning what's so useful about -1/0/1 comparison operators, since not everyone might know: you can chain them together with the or-operator to do primary/secondary/etc. sorts. So `($a->lname cmp $b->lname) || ($a->fname cmp $b->fname)` sorts people by their last names, but if two people have the same last name then they will be ordered by their first name. – hobbs Jul 26, 2010 at 11:13

@J.J. Python does have a 3-value compare: cmp() >>> print (cmp(5,7), cmp(6,6), cmp(7,5)) (-1, 0, 1) – bukzor Jul 29, 2010 at 0:53

18 votes

This is a meta-answer, but the Perl Tips archives contain all sorts of interesting tricks that can be done with Perl. The archive of previous tips is on-line for browsing, and can be subscribed to via mailing list or atom feed.

Some of my favourite tips include building executables with PAR, using autodie to throw exceptions automatically, and the use of the switch and smart-match constructs in Perl 5.10.

Disclosure: I'm one of the authors and maintainers of Perl Tips, so I obviously think very highly of them. ;)

Share

answered Oct 2, 2008 at 13:30

community wiki
pjf

2 It's probably one of the best documented languages out there, and set the pattern for tools to search documentation. That the list in this question is probably not as needed as for other languages. – Axeman Oct 2, 2008 at 21:40

18 votes

map - not only because it makes one's code more expressive, but because it gave me an impulse to read a little bit more about this "functional programming".

Share

edited Aug 23, 2009 at 21:56

community wiki
2 revs, 2 users 67%
brunorc

15
votes

The continue clause on loops. It will be executed at the bottom of every loop, even those which are next'ed.

```
while( <> ){
  print "top of loop\n";
  chomp;

  next if /next/i;
  last if /last/i;

  print "bottom of loop\n";
}continue{
  print "continue\n";
}
```

Share

answered Oct 4, 2008 at 2:29

community wiki
Shawn H Corey

15
votes

My vote would go for the (?{}) and (??{}) groups in Perl's regular expressions. The first executes Perl code, ignoring the return value, the second executes code, using the return value as a regular expression.

Share

edited Aug 23, 2009 at 21:53

community wiki
2 revs, 2 users 67%
Leon Timmermans

perl invented so many regexp extensions that other programs now often use pcre (perl compatible regex) instead of the original regex language. – Sec Oct 2, 2008 at 12:36

Read the little blurb here perldoc.perl.org/... :-D – J.J. Oct 2, 2008 at 15:30

Perl really has (as far as I know), lead the pack, when it comes to regexps. – Brad Gilbert Oct 2, 2008 at 21:53

This, as far as I'm aware, is still experimental, and may not work the same way in future Perls. Not to say that it isn't useful, but a slightly safer and just as useable version can be found in the s/// command's /e flag: `s/(pattern)/reverse($1);/ge;` # reverses all `patterns`. – Chris Lutz Feb 9, 2009 at 23:22

@Chris Lutz, @Leon Timmerman: Note that those two constructs are now reëntrant. Also note that the second one need no longer be used to effect recursive patterns, now that we can recurse on capture groups. @Brad Gilbert: That’s right, although PCRE does a decent job of tracking us; one area of regex excellence where Perl is completely unchallenged is its access to Unicode properties; see my unitrio distribution of `uninames`, `unichars`, and especially `uniprops` to see just part of what I mean. – tchrist Nov 11, 2010 at 15:33

13
votes

```
while(/\G(\b\w*\b)/g) {
     print "$1\n";
}
```

the \G anchor. It's **hot**.

Share

answered Oct 2, 2008 at 14:25

community wiki
J.J.

3 ...and it indicates the position of the end of the previous match. – Dave Sherohman Oct 2, 2008 at 16:05

1 But you have to call your regex in scalar context. – davidnicol Oct 3, 2008 at 21:03

@davidnicol: The above code works. Can you clarify what you mean? – J.J. Oct 4, 2008 at 9:27

13 votes

The `m//` operator has some obscure special cases:

- If you use `?` as the delimiter it only matches once unless you call `reset`.
- If you use `'` as the delimiter the pattern is not interpolated.
- If the pattern is empty it uses the pattern from the last successful match.

Share

edited Nov 11, 2010 at 15:35

community wiki
2 revs, 2 users 80%
Michael Carman

2 These are more like hidden gotchas than hidden features! I don't know anyone who likes them. A thread on p5p some time back discussed the usefulness of a putative m/$foo/r flag, where /r would mean no interpolation (the letter isn't important) since no-one can ever remember the single quotes thing.
– dland Oct 2, 2008 at 16:43

2 @dland: Agreed; I'd call these hidden *mis*features and would never use them in production code. – Michael Carman Oct 2, 2008 at 16:59

7 I can't imagine a Perl programmer being unable to remember (or even guess) that single quotes stand for no interpolation. Its usage with this semantics is almost universal in the language that I'd rather *expect* this to be so... – Sundar R Oct 3, 2008 at 17:53

and if the pattern is empty and the last successful match was compiled with the /o modifier, from then on it will be stuck on that pattern. – davidnicol Oct 3, 2008 at 21:11

1 I think the empty pattern behaviour has been deprecated. Primarily because a pattern like m/$foo/ becomes a nasty bug when $foo is empty. – Matthew S May 18, 2010 at 5:12

12
votes

The null filehandle diamond operator `<>` has its place in building command line tools. It acts like `<FH>` to read from a handle, except that it magically selects whichever is found first: command line filenames or STDIN. Taken from perlop:

```
while (<>) {
...          # code for each line
}
```

Share

answered Oct 2, 2008 at 13:06

community wiki
spoulson

4 It also follows the UNIX semantics of using "-" to mean "read from stdin. So you could do `perl myscript.pl file1.txt - file2.txt`, and perl would process the first file, then stdin, then the second file. – Ryan C. Thompson Oct 27, 2009 at 22:28

You can `overload` the `<>` operator on your own objects (`<$var>`) to work like an iterator. However it does not work as you could expect in list context. – dolmen Mar 29, 2011 at 12:50

11 votes

Special code blocks such as `BEGIN`, `CHECK` and `END`. They come from Awk, but work differently in Perl, because it is not record-based.

The `BEGIN` block can be used to specify some code for the parsing phase; it is also executed when you do the syntax-and-variable-check `perl -c`. For example, to load in configuration variables:

```
BEGIN {
    eval {
            require 'config.local.pl';
        };
        if ($@) {
            require 'config.default.pl';
        }
    }
```

Share

edited Oct 2, 2008 at 13:26

community wiki
3 revs
Bruno De Fraine

11
votes

```
rename("$_.part", $_) for "data.txt";
```

renames data.txt.part to data.txt without having to repeat myself.

Share

answered Oct 2, 2008 at 17:12

community wiki
timkay

10
votes

A bit obscure is the tilde-tilde "operator" which forces scalar context.

```
print ~~ localtime;
```

is the same as

```
print scalar localtime;
```

and different from

```
print localtime;
```

Share

answered Oct 2, 2008 at 12:42

community wiki
Sec

5 This is especially obscure because perl5.10.0 also introduces the "smart match operator" `~~`, which can do regex matches, can look if an item is contained in an array and so on. – moritz Oct 2, 2008 at 12:52

That's not obscure, that's obfuscated (and useful for golf and JAPHs). – Michael Carman Oct 2, 2008 at 13:18

This is not correct! ~~ is not safe on references! It stringifies them. – Leon Timmermans Oct 2, 2008 at 15:04

Well, yes. Stringification is what happens to references when forced into scalar context. How does that make "~~ forces scalar context" incorrect? – Dave Sherohman Oct 2, 2008 at 16:00

3 @Nomad Dervish: Scalar context /= stringification. e.g. "$n = @a" is scalar context. "$s = qq'@a'" is stringification. With regard to references, "$ref1 = $ref2" is scalar context, but does not stringify. – Michael Carman Oct 2, 2008 at 17:09

9

votes

tie, the variable tying interface.

Share

answered Oct 3, 2008 at 21:15

community wiki
davidnicol

9

votes

The "desperation mode" of Perl's loop control constructs which causes them to look up the stack to find a matching label allows some curious behaviors which Test::More takes advantage of, for better or worse.

```
SKIP: {
    skip() if $something;

    print "Never printed";
}

sub skip {
    no warnings "exiting";
    last SKIP;
}
```

There's the little known .pmc file. "use Foo" will look for Foo.pmc in @INC before Foo.pm. This was intended to allow compiled bytecode to be loaded first, but Module::Compile takes advantage of this to cache source filtered modules for faster load times and easier debugging.

The ability to turn warnings into errors.

```
local $SIG{__WARN__} = sub { die @_ };
$num = "two";
$sum = 1 + $num;
print "Never reached";
```

That's what I can think of off the top of my head that hasn't been mentioned.

Share

answered Oct 15, 2008 at 15:13

community wiki
Schwern

9
votes

The goatse operator `*`:

```
$_ = "foo bar";
my $count =()= /[aeiou]/g; #3
```

or

```
sub foo {
    return @_;
}

$count =()= foo(qw/a b c d/); #4
```

It works because list assignment in scalar context yields the number of elements in the list being assigned.

`*` Note, not really an operator

Share

answered May 31, 2009 at 3:13

community wiki
Chas. Owens

That is the most (well, least) beautiful "operator" ever.
– Chris Lutz Aug 24, 2009 at 23:59

1 2 3 Next