



Perl 5






<p>See also: Perl - Perl</p> <ul style="list-style-type: none"> Perl @ Wikipedia perl.org perl @ GitHub PerlMonks.org O'Reilly Books Perl mailing lists Perl Weekly 	<ul style="list-style-type: none"> <i>Quick Intros to Perl:</i> Perl Intro, PerlCheat, Learn Perl in Y minutes, or in 2 hours 30 minutes Online Perl books & tutorials: Beginning Perl, Modern Perl (html), Perl Maven Tutorial, Intro to Perl (old) Impatient Perl, Extreme Perl/Minimum Viable Perl, Just Enough Perl for Rex, Perl Cookbook (PLEAC Perl: Perl code & solutions), Learning Perl LPo, Intermediate Perl IntPo, Mastering Perl o, Effective Perl Programming o, Object Oriented Perl, Higher-order Perl HoP. Others not recommended <p><i>Perl Guidelines and tools:</i> Perl Style Guide, 10 Essential Development Practices. Books: Perl Best Practices o, Modern Perl Best Practices (course) o</p> <ul style="list-style-type: none"> perlritic script uses Perl::Critic to scan Perl code. The pel-perl-critic command invokes it to check code in buffer. The perltidy application reformats Perl code. Older perltidy home page. PerlTidy @ Wikipedia, PBP recommended .perltidyrc 	<p>perl, Perl command line options, perlrun, perlvip, perldoc, perlbug / perthanks perlsec</p> <ul style="list-style-type: none"> Online Perl Interpreter perl-live-coding out & in Emacs Online PerlTidy option info.
<p>perldoc browser</p> <ul style="list-style-type: none"> In Emacs: C-c C-h F 	<p>perldoc : About perldoc itself. perltoc : Table of content: names of all pages. perlsyn : Perl syntax. perlfunc : Perl built-in functions.</p>	<p> Use perldoc to find if a Perl module is installed, as in: <code>perldoc local::lib</code></p> <ul style="list-style-type: none"> <code>perldoc local::lib</code> prints the documentation of <code>local::lib</code> if it is installed. <code>perl -Mlocal::lib</code> is useful to get modules installed in your home directory o
<p>CPAN (@ Wikipedia)</p> <ul style="list-style-type: none"> Search: meta:cpan CPAN Testers CPANdeps 	<ul style="list-style-type: none"> The Zen of Comprehensive Archive Networks PAUSE - Perl Authors Upload Server Installing Local Perl Modules with CPAN CPAN Issue tracker: CPAN RT See Also: IntPo 	<p>Command line tools interacting with CPAN to install Perl modules o. (see also this StackOverflow Q/A):</p> <ul style="list-style-type: none"> <code>cpan</code>: (requires config, but has defaults). Use <code>local::lib</code>; cpan will be able to install into your <code>~/perl5</code> tree. <ul style="list-style-type: none"> Type <code>cpan</code> to open the cpan shell, then type <code>install The::Module</code> to install packages. <code>cpanplus</code>, or <code>cpanminus</code>: <code>cpanm</code> (no config required). <code>cpanm</code>: <code>cpanm -S The::Module</code>

Last updated on: 2025-02-24

Perl scripts

<p>Writing Perl scripts</p>	<p>Impose strictures in Perl files to prevent errors by adding one of the following use lines. Also see the strictures package.</p>		
<p>Use the following at the beginning of Perl script files.</p> <p><code>perldiag @ perldoc</code></p>	<pre>#!/usr/bin/env perl use strict; use warnings; # for testing only: use diagnostics;</pre>	<pre>#!/usr/bin/perl -w use v5.12; # loads strict ... use v5.35; # & loads warnings ! use diagnostics produces more info but increases startup time. Alternative: perl -Mdiagnostics . Emacs pel-perl-critic command can report diagnostic.</pre>	<p>Executable Perl script should have a valid shebang line identifying the appropriate location of the Perl interpreter. It may have to be modified at installation time (OpenGroup/SUS).</p> <p> It's best to: use warnings; <code>perl -w</code> generates warning for all Perl code in the program including modules used by the program. Also use the <code>-c</code> option to check syntax. But most Perl code should also activate the strict Perl rules and warnings to detect warnings. See: Barewords in Perl</p>
<p>use version/features</p>	<pre>use v5.36;</pre>	<p>This can be used to enable both the strict and warning pramas as well as several named features.</p> <ul style="list-style-type: none"> See the table listing the feature bundles per Perl versions. 	
<p>Perl version history</p> <ul style="list-style-type: none"> at perldoc <p>M: minor, P: patch level</p>	<ul style="list-style-type: none"> Perl Versions Guide Perl versions @ perldoc 	<ul style="list-style-type: none"> 5.even: maintenance track version 5.odd : development track version decimal: 1.02. # old way dot-decimal: v5.38.2 	<ul style="list-style-type: none"> <code>\$1</code> : current Perl version as a decimal number <code>\$^v</code> : current Perl version as a version object <p>Equivalence between decimal and dot-decimal versions: AAA.MMMPP ⇔ vAAA.MMM.PP . Note that 3 <i>Minor</i> digits are used in the decimal versions. Patch use 2 or 3.</p>

Perl 5 Operators

<p>Perl 5 Operators</p> <p>Note:</p>	<p>Perl operators, listed below with their precedence and associativity.</p> <p>C Operators missing from Perl : unary &, unary * and (type)</p> <p>Quote and Quote-like operators : in Perl quotes are operators and they provide various kind of interpolating and pattern matching capabilities.</p>		
<p>Associativity: one of:</p> <ul style="list-style-type: none"> right left NA : not associative: cannot use more than one of these operators in sequence. CH: chained <p>To get this information, use:</p> <p><code>perldoc perllop</code></p> <p>Note: o The Bitwise String Operators are:</p> <pre>~. &. . ^. &.= .= ^.=</pre> <ul style="list-style-type: none"> Stable: Perl >= 5.28 Experimental: Perl >= 5.22 	<p>terms and list operators (leftward) ()</p> <p>Arrow Operator: -></p> <p>Auto-increment and Auto-decrement: ++ --</p> <p>Exponentiation: **</p> <p>Symbolic Unary Operators: ! - -. \ and unary + and -</p> <p>Binding operators: == !-</p> <p>Multiplicative Operators: * / % x</p> <p>Additive Operators: + - .</p> <p>Shift Operators: << >></p> <p>named unary operators</p> <p>Class instance Operator: isa</p> <p>Relational Operators: as numbers: < > <= >= as strings: lt gt le ge</p> <p>Equality Operators: as numbers: == != <=> as strings: eq ne cmp --</p> <p>Bitwise And: & &.</p> <p>Bitwise Or and Exclusive Or: . ^ ^.</p> <p>C-style Logical And: &&</p> <p>Logical Or, Xor, Defined-Or: ^^ //</p> <p>Range Operators:</p> <p>Conditional Operator: ? :</p> <p>Assignment Operators: = **= += *= &= &.= <<= &&=</p> <pre>-- /= /= = = . = >>= = .= %= ^= ^= = // =</pre> <p>x= goto last next redo dump</p>	<p>Note: print, sort, reverse, chmod, are list operators</p> <p>Note: The operator <code>\</code> creates a reference. See example.</p>	
<p>trick operators </p> <p>Do not use in production code!</p> <p>But understanding how these work does help understand Perl.</p> <p>These are not real Perl operators; they are concatenation of other operators that achieve a specific effect.</p>	<p>-+ 0+ Converts a string that starts with digits into a number.</p> <pre>print -+ '22les poulets!'; # prints 22</pre> <p>= () = Called the 'goatse' operator. It causes the right side expression to be evaluated in array context. Used to assign the array/list size to a scalar.</p> <pre>my \$str = "A 22 before 33 does not make 9, it is 44!"; my \$digit_count = () = \$str =~ /\d/g; print "\$digit_count"; # prints '7', the number of digits in \$str</pre> <p>@{[]} Interpolate an array in a string: "<code>@{[something]}</code>" is the same as: <code>join "\$", something</code></p> <pre>print "these people @{{get_names()}} get promoted"</pre> <p>-- Force scalar context.</p> <p>In scalar context <code>localtime</code> returns human readable time, but in list context it returns a 9-tuple with date elements.</p> <pre>\$ perl -le 'print --localtime' Mon Nov 30 09:06:13 2009</pre>	<p>Truth and falsehood</p> <p> The strings '0' and '' mean false. The output of <code>glob()</code> may return a file named '0'!</p> <p> The bareword <code>false</code> has a truth value of <code>true</code>!</p> <p> One way to define valid true and false <i>constant symbols</i> that can be used in assignments (but see o):</p> <pre>use constant { true => 1, false => 0 };</pre>	
<p>File test operators</p> <p>See filetest -X</p>	<p>File tests can be stacked (<code>-r -w -e \$fname</code>) or combined as in the following example o:</p> <p> Notice the underscore in the example: it's the virtual filehandle <code>_</code> accessing the last <code>stat</code> or <code>lstat</code> result :</p> <pre>if (-e \$fname && -f _ && -r _) { print("\$fname exists, is readable\n"); }</pre>		
<p>The operators check if the file...</p> <p>See also:</p> <ul style="list-style-type: none"> File Tests o File test operators @ perl tutorial <p>See also:</p> <ul style="list-style-type: none"> localtime File::stat IO::Interactive 	<p>-r is readable by effective uid/gid</p> <p>-w is writable by effective uid/gid</p> <p>-x is executable by effective uid/gid</p> <p>-o is owned by effective uid</p> <p>-R is readable by real uid/gid</p> <p>-W is writable by real uid/gid</p> <p>-X is executable by real uid/gid</p> <p>-O file is owned by real uid.</p> <p>-M Days between start time and file modification time</p>	<p>-e exists.</p> <p>-z is empty.</p> <p>-s has nonzero size (returns size in bytes).</p> <p>-f is a plain file.</p> <p>-d is a directory.</p> <p>-l is a symbolic link.</p> <p>-p is a named pipe (FIFO) or Filehandle is a pipe.</p> <p>-S is a socket.</p> <p>-A Days between start time and file access time</p>	<p>-b is a block special file.</p> <p>-c is a character special file.</p> <p>-t handle is opened to a tty.</p> <p>-u has setuid bit set.</p> <p>-g has setgid bit set.</p> <p>-k has sticky bit set.</p> <p>-T is an ASCII text file (heuristic guess).</p> <p>-B is a "binary" file (opposite of -T).</p> <p>-C Days between start time and node change time (in Unix).</p>

Perl 5 Constants and Variables 🚧

Perl Constants	Perl pragma to declare constants ! but not read-only! See CPAN modules for defining constants by Neil Bowers and Const::Fast and Attribute::Constant			
Perl Variables Names	Scalar Naming Conventions	Array Naming Conventions	All: 1 st char: underscore or letter. Never use ALLCAPS	
Case sensitive. ASCII by default, UTF-8 if the utf8 pragma is used.	<ul style="list-style-type: none"> All variables: words_with_underscores Local variables: \$lowercase Global variables: \$Title_Case Constants: \$UPPER_CASE 	Same. Array names should be plural. <ul style="list-style-type: none"> @locals @Global_Arrays @CONSTANT_ARRAYS 	<ul style="list-style-type: none"> Module names are MixedCaseNoUnderscores Constants are UPPERCASE_WITH_UNDERSCORES Package wide vars are Mixed_Case_With_Underscores Functions/methods are lowercase_with_underscores 	
Scope of variables	A variable defined without any of the following prefixed keyword is global by default .	With use strict ; Perl warns when globals are used. If using a global is needed, do something like this:	Write use vars qw(\$AUTOLOAD); to pre-declare the \$AUTOLOAD scalar variable and prevent warning.	
Declarations Scope of variables in Perl @Perl Maven	my local, lexical scope, non persistent	Examples: my @values = (42, 36, 99); my (\$v1, \$v2) = (42, 36);		
state	Local, lexical scope, persistent	<i>Perl >= v5.10</i> Restriction: in <i>Perl < v5.28</i> : array and hashes state cannot be initialized in list context.		
our	Creates a lexical scoped alias to a package (i.e. global) variable. Prevents global variable access warnings when strict 'vars' is active.			
local	Localizes an existing package variable to the current scope. It's not a declaration. The variable previous value is restored when leaving the scope. <ul style="list-style-type: none"> In modern Perl 5, use it to localize modifications to a global variable or hash value. It's a simple dynamic binding mechanism. 			
6 kinds of variables types:	1. scalar \$ 2. array @	3. hash % 4. subroutine (code). &	5. format (See write and select) <ul style="list-style-type: none"> how to format output in Perl?, Perl-Formats 	6. I/O: file, directory, other handles
Perl types Scalar See: Scalar::Util Archaic use of single quote: <code>\$Dog'days</code>	\$foo Simple scalar value \$days[28] 29 th element of array @days \$days{'Feb'} Value associated with the <i>Feb</i> key of hash %days \${days} Same as \$days, use before alphanumeric. \$Dog::days The \$days variable inside the Dog package.		 \$#days Last index of array @days. \$days->[28] 29 th element of array pointed to by reference \$days. \$days[0][2] Multi-dimensional array \$d{99}{'Feb'} Multi-dimensional hash \$d{99, 'Feb'} Multi-dimensional hash emulation	
list and Array • 0-based indexed (first index is 0). • Last index of array @name is \$#name	<ul style="list-style-type: none"> Arrays are initialized by literal lists. Lists are always flattened in Perl: 	<ul style="list-style-type: none"> You can assign a list of values to a list of variables. Useful to swap: <code>(\$val1, \$val2) = (\$val2, \$val1);</code> If there are more variables than values: the extra variables are set to undef. Extra values are ignored. 		
array slices LPo Simple explanation	<ul style="list-style-type: none"> This means that <code>(1, 2, (10, 20, (100, 200), 30, 40), 4)</code> is exactly the same as <code>(1, 2, 10, 20, 100, 200, 30, 40, 4)</code>. Use references to create nested data structures. 			
Anonymous arrays	<ul style="list-style-type: none"> What are the advantages of anonymous array? @ StackOverflow Periref @ Perldoc, Perl reference tutorial @ Perldoc 		<ul style="list-style-type: none"> Anonymous array := a type of array reference. Use it to build nested data structures. Array reference allows Perl to treat the array as a single item. 	
Hash/associative array Hashes @ Perl Maven Note: keys are always strings.	% %days	Associative array (hash): keys-value pairs. Can be initialized as: <ul style="list-style-type: none"> <code>my %days = (Jan => 31, Feb => \$leap? 29 : 28, ...)</code> <code>my %days = ("Jan", 31, 'Feb', \$leap? 29 : 28, ...)</code> Multiple values of a hash can be changed with the following construct:	Initialize a hash slice with array context: <pre>@char_to_num{'A'..'Z'} = 1..26; my %rating = (ron => 20, al => 50, steve => 80); # use fat comma to quote word left of it.</pre>	
hash slice LPo	@days{'J', 'F'}	Hash slice returning a list containing <code>(\$days{'J'}, \$days{'F'})</code> .	my @names = ('ron', 'al');	
key-value slices LPo	extract/write values:	my scores = @rating{@names}; @rating{@names} = (45, 55);	@rating{ @names } = (25, 35); # update ron & al's ratings	
Subroutine	& &foo	& is needed to create reference to subroutine with <code>\&subroutine_name</code>		
I/O				
Format				
Typoglob	A typglob is a symbol table structure with the slots of that symbol for the scalar, array, hash, code, format and I/O form of the symbol in the namespace.			
	* *symbol1	See: Object Oriented Perl , section 2.2.4. Typeglobs. Advanced Perl Programming, 1st Edition Section 3.2		
References Perl references intro Perl reference tutorial Reference purpose IntPo	A reference is a scalar variable whose value is a pointer to another Perl variable. Use it to build more complex data types . Make reference with <code>\</code> . The ref built-in returns a string describing the referent: 'ARRAY', 'HASH', 'CODE', 'FORMAT', 'IO', the class name of a blessed object, an empty string if arg is not a reference.			
• brace around refs: circumfix dereferencing:	my @array = qw(a, b, c); print \$array[1].# b You can create complex data with references: <code>###</code>	my \$array_ref = ['a', 'b', 'c\n']; print \$\$array_ref[1]; # b print \$\$array_ref[1]; # b, simpler print \$array_ref->[1]; # b, arrow notation	my %hash = (a=>1, b=>2, c=>3); print \$hash{c}; # 3 ◀ drop brace around bareword ref.▶ ◀ arrow notation is shorter/cleaner ▶	my \$hash_ref = {a=>1, b=>2, c=>3}; print \$\$hash_ref{c}; # 3 print \$\$hash_ref{c}; # 3, simpler print \$hash_ref->{c}; # 3 with arrow notation
• simplify with <code>-></code>	my \$data = [0, 1, 2, [40, 50, 60, [100, 200], 70], 8]; print @{{\${\$data}[3]}[3]}[0], "\n"; # 100 print \$data->[3]->[3]->[0], "\n"; # 100 print \$data->[3]->[3]->[0], "\n"; # 100 print \$data->[3][3][0], "\n"; # 100.		<ul style="list-style-type: none"> Create a lexical reference: <code>my \$hash_ref = \%hash;</code> Store a ref to an array or hash into an array: <code>push @array, \%hash;</code> Pass array or hash to subroutine: <code>fact(@a, \%h);</code> Return from sub: <code>return (\@a, \%h);</code> 	
☺ Disambiguate hash references with <code>+{ ... }</code>			◀ Arrows between subscript are optional.	
Symbolic References With a simple string it refers to the symbols table of the <i>main</i> package. The string can also be fully qualified name , then it uses the specified symbol table.	<ul style="list-style-type: none"> Symbolic references are very flexible but dangerous and not allowed when use strict is imposed. It's not used often but it's important to know they exist. A <i>symbolic</i> reference is a string containing the name of a variable or subroutine in a package's symbol table. They cannot access lexical variables. If a symbolic reference is necessary, restrict it's use to a block and relax the warning checks in block with: no strict "refs"; 			
	package main; \$name = "data"; print \${\$name}; push @{\$name}, 42; &{\$name}();	Same as: print \$main::data; push @main::data, 42; &main::data();	\$sref = "Pkg::var"; \$sref->{level} = "high"; \$val = \$sref->[3]; \$sref->(\$val, 22); &{"Pkg" . "var"}();	Same as: \$Pkg::var{level} = "high"; \$val = \$Pkg::var[3]; \$Pkg::var(\$val, 22); &Pkg::var();
postfix dereferencing See: cool new Perl feature: postfix dereferencing	<i>(Perl >= v5.20.0)</i> Instead of using a sigil prefix, it uses a postfix sigil and star. sref : ref to scalar, aref : ref to array, href : ref to hash, cref : ref to code, gref : ref to glob			
	\$sref->*\$; # same as \${ \$sref } \$aref->*\$; # same as @{ \$aref }	\$aref->\$\$*; # same as #{ \$aref } #last array idx \$href->\$\$*; # same as { \$href }	\$cref->*&; # same as &{ \$cref } \$gref->*&; # same as *{ \$gref }	
Reference to subroutine	Store a ref to a subroutine:	my \$fct_ref = \&the_function;	Indirect calls: with the simpler arrow notation :	<ul style="list-style-type: none"> &{ \$the_function } (arg1, arg2); \$the_function->(arg1, arg2);
	Using an anonymous subroutine, always calling it indirectly:		my \$op = sub { my \$v1 = shift; my \$v2 = shift; return \$v1 ** \$v2;}; say \$op->(10, 4); # prints 10000	
Autovivification. ! What is autovivification? Perl surprise/problem with autovivification	Unlike most programming languages Perl automatically creates missing parts of arrays, hashes when an undefined value is referenced . Also see: autovivification in for loop but not assignment?			
	no autovivification; # turn off vivification except for setting value		no autovivification 'exists'; # turn it off just for exists checks. See others.	
Closures • Perl closure ☺ Note how easy it is to create a closure in Perl: a simple block that defines a lexical variable referenced by subroutines defined in that block. The variable is not accessible outside of the block but the subroutines are!	A closure binds its environment and keeps it to use it when invoked. <ul style="list-style-type: none"> In the example at right, a greeter function is built and returned, remembering how to greet. It is used like this: <pre>my \$fr = make_greeting("Bonjour"); my \$it = make_greeting("Buongiorno"); \$fr->('Brigitte'); # prints: "Bonjour, Brigitte!\n" \$it->('Madonna'); # prints: "Buongiorno, Madonna!\n"</pre> 			
	A code block defining lexical variable(s) and subroutines consist of a closure too! With the following example, the add_1() subroutine increments the \$count and that's returned by get_count() . The \$count variable cannot be accessed from anywhere else!		<pre>sub make_greeting { my \$greet = shift; my \$greet_fct = sub { my \$name = shift; print "\$greet, \$name!\n"; }; return \$greet_fct; # return ref to internal function }</pre>	
		{ my \$count; # lexically scoped variables are only accessible inside the block sub add_1 { count += 1; } # but the subroutine is not lexical it's visible sub get_count { return count; } # in the package (main by default). } # The lifetime of the subroutines is the program, keeping the referred-to variables alive!		

Scalar values	Numeric	literals examples:	Note: leading 0 work only for literals, not for string-to-number conversions.	Useful related builtin functions																																						
<ul style="list-style-type: none"> numeric: <p>Note: underline separators can be used inside decimal, hexadecimal and binary literals.</p>	<ul style="list-style-type: none"> integer : using the system's native format. <ul style="list-style-type: none"> bigint - transparent big integer support. bignum - transparent big number support. floating-point : using the system's native format. <ul style="list-style-type: none"> bigrat - transparent big rational number support. <p><i>A variable holding an integer can be converted to floating-point if the operation done to it requires it (such as dividing 1 by 2).</i></p>	<pre>my \$x = 12345; # integer my \$x = 12345.67; # floating point my \$x = 6.02e23; # scientific notation my \$x = 0x1f.0p3; # power² exponent: <i>Perl >= v5.22</i> my \$x = 4_294_967_296; # underline for legibility my \$x = 0x1234_5678; # underline in hex is also OK my \$x = 0377; # octal my \$x = 0o377; # octal also <i>Perl >= v5.34</i> my \$x = 0b1100_0010; # binary with underlines my \$x = 0xff55; # hexadecimal</pre>	<ul style="list-style-type: none"> oct - for: binary, octal, hex hex POSIX::ceil POSIX::floor abs 																																							
<ul style="list-style-type: none"> string 	<ul style="list-style-type: none"> double-quoted strings: perform backslash and variable interpolation of expression that begin with \$ (a scalar) or @ (an array). Hashes cannot be interpolated. single-quote strings: only perform \' and \\ substitution (to ' and \ respectively), nothing else. Single quote and double quote strings can spread multiple lines: it embeds the newline character on each new line. \n is only expanded in double quoted strings. In single quote string it is treated as two characters; no substitution is done (as explained above). 																																									
<ul style="list-style-type: none"> Unicode support 	Use Unicode literally in a program; add the utf8 pragma : <code>use utf8</code> ; See: Perl Unicode Tutorial , Perl Unicode Introduction , Perl Unicode Support @ perldoc																																									
<ul style="list-style-type: none"> Quote constructs <p>See:</p> <ul style="list-style-type: none"> Strings in Perl: quoted, interpolated and escaped 	<table border="1"> <thead> <tr> <th>Usual</th> <th>Generic</th> <th>Meaning</th> <th>Interpolates?</th> <th>Notes</th> </tr> </thead> <tbody> <tr> <td>''</td> <td>q//</td> <td>Literal string</td> <td>No</td> <td rowspan="10"> <ul style="list-style-type: none"> Not all characters can be used as the / separator. { }, () and < > can also be used. You can use whitespace between the quote specifier and its initial bracketing character: <pre>my \$chuck_of_code = q { if (\$condition) { print "Bonjour!"; } };</pre> </td> </tr> <tr> <td>""</td> <td>qq//</td> <td>Literal string</td> <td>Yes</td> </tr> <tr> <td>``</td> <td>qx//</td> <td>Command execution</td> <td>Yes</td> </tr> <tr> <td>()</td> <td>qw//</td> <td>Word list</td> <td>No</td> </tr> <tr> <td>//</td> <td>m//</td> <td>Pattern match</td> <td>Yes</td> </tr> <tr> <td>s///</td> <td>s///</td> <td>Pattern substitution</td> <td>Yes</td> </tr> <tr> <td>tr///</td> <td>y///</td> <td>Character translation</td> <td>No</td> </tr> <tr> <td>""</td> <td>qr//</td> <td>Regular expression</td> <td>Yes</td> </tr> </tbody> </table>	Usual	Generic	Meaning	Interpolates?	Notes	''	q//	Literal string	No	<ul style="list-style-type: none"> Not all characters can be used as the / separator. { }, () and < > can also be used. You can use whitespace between the quote specifier and its initial bracketing character: <pre>my \$chuck_of_code = q { if (\$condition) { print "Bonjour!"; } };</pre> 	""	qq//	Literal string	Yes	``	qx//	Command execution	Yes	()	qw//	Word list	No	//	m//	Pattern match	Yes	s///	s///	Pattern substitution	Yes	tr///	y///	Character translation	No	""	qr//	Regular expression	Yes			
Usual	Generic	Meaning	Interpolates?	Notes																																						
''	q//	Literal string	No	<ul style="list-style-type: none"> Not all characters can be used as the / separator. { }, () and < > can also be used. You can use whitespace between the quote specifier and its initial bracketing character: <pre>my \$chuck_of_code = q { if (\$condition) { print "Bonjour!"; } };</pre> 																																						
""	qq//	Literal string	Yes																																							
``	qx//	Command execution	Yes																																							
()	qw//	Word list	No																																							
//	m//	Pattern match	Yes																																							
s///	s///	Pattern substitution	Yes																																							
tr///	y///	Character translation	No																																							
""	qr//	Regular expression	Yes																																							
<ul style="list-style-type: none"> Character escapes (only inside double quoted strings) 	<table border="1"> <tbody> <tr> <td>\a</td> <td>Alert (bell)</td> <td>\t</td> <td>Horizontal tab</td> <td>\x{263a}</td> <td>Character number 0x263A</td> </tr> <tr> <td>\b</td> <td>Backspace</td> <td>\e</td> <td>ESC character</td> <td></td> <td></td> </tr> <tr> <td>\c</td> <td>ESC character</td> <td>\O33</td> <td>ESC in octal</td> <td></td> <td>Any Unicode code point, by name: \N{LATIN SMALL LETTER E WITH ACUTE} é</td> </tr> <tr> <td>\f</td> <td>Form feed</td> <td>\o{33}</td> <td>ESC in octal</td> <td></td> <td>\N{ U+E9 } é</td> </tr> <tr> <td>\n</td> <td>Newline (usually LF)</td> <td>\x7f</td> <td>DEL in hexadecimal</td> <td></td> <td></td> </tr> <tr> <td>\r</td> <td>Carriage return (Usually CR)</td> <td>\cC</td> <td>Control-C</td> <td></td> <td></td> </tr> </tbody> </table>	\a	Alert (bell)		\t	Horizontal tab	\x{263a}	Character number 0x263A	\b	Backspace	\e	ESC character			\c	ESC character	\O33	ESC in octal		Any Unicode code point, by name: \N{LATIN SMALL LETTER E WITH ACUTE} é	\f	Form feed	\o{33}	ESC in octal		\N{ U+E9 } é	\n	Newline (usually LF)	\x7f	DEL in hexadecimal			\r	Carriage return (Usually CR)	\cC	Control-C						
\a	Alert (bell)	\t	Horizontal tab		\x{263a}	Character number 0x263A																																				
\b	Backspace	\e	ESC character																																							
\c	ESC character	\O33	ESC in octal		Any Unicode code point, by name: \N{LATIN SMALL LETTER E WITH ACUTE} é																																					
\f	Form feed	\o{33}	ESC in octal		\N{ U+E9 } é																																					
\n	Newline (usually LF)	\x7f	DEL in hexadecimal																																							
\r	Carriage return (Usually CR)	\cC	Control-C																																							
<ul style="list-style-type: none"> translation escapes (inside double quoted strings) 	<table border="1"> <tbody> <tr> <td>\u</td> <td>Force next character to titlecase</td> <td>\U</td> <td>Force all following characters to uppercase. Ends at \E</td> <td>\E</td> <td>Ends \U, \L, \F or \Q</td> </tr> <tr> <td>\l</td> <td>Force next character to lowercase</td> <td>\L</td> <td>Force all following characters to lowercase. Ends at \E</td> <td></td> <td></td> </tr> <tr> <td></td> <td></td> <td>\F</td> <td>Force all following characters to Unicode fold case. Ends at \E</td> <td></td> <td></td> </tr> <tr> <td></td> <td></td> <td>\Q</td> <td>Backslash all following non alphanumeric characters. Ends at \E</td> <td></td> <td></td> </tr> </tbody> </table>	\u	Force next character to titlecase	\U	Force all following characters to uppercase. Ends at \E	\E	Ends \U , \L , \F or \Q	\l	Force next character to lowercase	\L	Force all following characters to lowercase. Ends at \E					\F	Force all following characters to Unicode fold case. Ends at \E					\Q	Backslash all following non alphanumeric characters. Ends at \E																			
\u	Force next character to titlecase	\U	Force all following characters to uppercase. Ends at \E	\E	Ends \U , \L , \F or \Q																																					
\l	Force next character to lowercase	\L	Force all following characters to lowercase. Ends at \E																																							
		\F	Force all following characters to Unicode fold case. Ends at \E																																							
		\Q	Backslash all following non alphanumeric characters. Ends at \E																																							
<ul style="list-style-type: none"> bareword 	In Perl, a <i>bareword</i> refers to a sequence of characters suitable for an identifier. It's not quoted. By default Perl allows barewords to behave like strings. This is not allowed when any of <code>use strict</code> ; or <code>use strict "subs"</code> ; or <code>use v5.12</code> ; is specified.																																									
<ul style="list-style-type: none"> Here documents <ul style="list-style-type: none"> Here docs @ Perl maven Perl here doc @Wikipedia 	Perl here-documents are a form of line oriented quoting. There are several forms of here documents, where the identifier (like EOF used below, but can be any word) must be placed at the beginning of the terminating line: Note: They can also be stacked and text can be transformed. See the documentation .																																									
<ul style="list-style-type: none"> Perl Regexp 	Regexp Tutorial , Learn PCRE in X minutes , PCRE cheatsheet ,		Debugex regexp tester, regex101 , RegEx Pal																																							
<ul style="list-style-type: none"> index/substr 	<code>\$pos = index(\$page, \$line);</code>	<code>\$last_slash = rindex("usr/bin/ls", "/");</code>	<code>\$part = substr(\$text, \$pos, \$len)</code>	A value of -1 in pos identifies last character.																																						
<ul style="list-style-type: none"> Replacement manipulate strings with substr LPc 	<code>my \$pref = "I like awk and erlang";</code> <code>substr(\$pref, index(\$pref, "awk"), length("awk")) = "Perl";</code> <code>substr(\$pref, 0, 0) = "Sally and ";</code> # insert text anywhere		<code>substr(\$pref, -15) =~ s/Perl/Perl5/g;</code> # replace text inside a restricted portion of the string.																																							

Perl 5 *Special* Literal and Variables

Special Literals	<ul style="list-style-type: none"> __FILE__ : current file name __LINE__ : current line number 	<ul style="list-style-type: none"> __PACKAGE__ : current package name __SUB__ : reference to current subroutine 	<ul style="list-style-type: none"> __END__ : use to indicate logical end of script __DATA__ : same, but supports reading text 	
<ul style="list-style-type: none"> Perl Special Variables Perl Variables 	<p>👉 To get information about a Perl special variable from the command line use the perldoc -v command.</p> <p>To get information about \$< use: perldoc -v '\$<'</p>			
<ul style="list-style-type: none"> Deprecated and removed variables: 	<pre>\$# \$* \$_ \${^ENCODING} \${^WIN32_SLOPPY_STAT}</pre>			
<ul style="list-style-type: none"> General variables 	Note that the \$, @ and % prefixes are the sigil that identify the scalar, array and hash access context. The name of the variable is placed after that character.			
default input and pattern searching space	<ul style="list-style-type: none"> \$ARG \$_ 	subroutine parameters :		
list separator	<ul style="list-style-type: none"> \$LIST_SEPARATOR \$" 	Subscript separator for multidimensional array emulation :	<ul style="list-style-type: none"> @_ @_ \$\$SUBSCRIPT_SEPARATOR \$\$SUBSEP 	
Name of executed program	<ul style="list-style-type: none"> \$PROGRAM_NAME \$0 	Name used to execute the current copy of Perl		
Perl process ID	<ul style="list-style-type: none"> \$PROCESS_ID \$PID \$\$ 	Process real GID	<ul style="list-style-type: none"> \$REAL_GROUP_ID \$GID \$(
Process real UID	<ul style="list-style-type: none"> \$REAL_USER_ID \$UID \$< 	Process effective UID	<ul style="list-style-type: none"> \$EFFECTIVE_USER_ID\$ \$EUID \$> 	
Special variables in sort	<ul style="list-style-type: none"> \$a \$b 	The Perl sort function uses global variables \$a and \$b as argument to the function used by sort to compare strings. Pass a sorting function that uses the <=> equality operator to force numerical comparisons, as in: <code>@sorted = sort { \$a <=> \$b } @unsorted;</code>		
Current environment	%ENV	Environment variable accessed as an associative array (a hash). • See: Perl: How to access shell environment variables through Perl associative arrays.		
Perl interpreter revision, version and subversion	<ul style="list-style-type: none"> \$OLD_PERL_VERSION \$] 	Perl interpreter revision, version and subversion		
Maximum file descriptor	<ul style="list-style-type: none"> \$\$SYSTEM_FD_MAX \$^F 	Fields of each line when auto-split mode is on.		
Include Directories	@INC	Included filenames	%INC	Hook localization (?)
inplace-edit extension value	<ul style="list-style-type: none"> \$INPLACE_EDIT \$^I 	Package's class parent classes	@ISA	Emergency memory pool
Maximum block nesting	\${^MAX_NESTED_EVAL_BEGIN_BLOCKS}	Time when program started		<ul style="list-style-type: none"> \$BASETIME \$^T
Name of OS where this Perl was built	<ul style="list-style-type: none"> \$OSNAME \$^O 	Signal handlers	%SIG	Coderefs for various perl keywords

• Regexp Variables			
captured sub-patterns	<code><digit>(\$1, \$2, ...)</code>	Capture buffer content	<code>@{^CAPTURE}</code>
String matched	<ul style="list-style-type: none"> <code>\$MATCH</code> <code>\$&</code> 	String matched (compiled regexp)	<code>\${^MATCH}</code>
String preceding match	<ul style="list-style-type: none"> <code>\$PREMATCH</code> <code>\$'</code> 	String preceding match (compiled regexp)	<code>\${^PREMATCH}</code>
String following match	<ul style="list-style-type: none"> <code>\$POSTMATCH</code> <code>\$'</code> 	String following match (compiled regexp)	<code>{^POSTMATCH}</code>
Last capture group	<ul style="list-style-type: none"> <code>\$LAST_PAREN_MATCH</code> <code>\$+</code> 	Most recently closed capture group	<ul style="list-style-type: none"> <code>\$LAST_SUBMATCH_RESULT</code> <code>\$^N</code>
Match capture key values	<ul style="list-style-type: none"> <code>%+</code> <code>%{^CAPTURE}</code> <code>%LAST_PAREN_MATCH</code> 	Maximum regexp nested group	<code>\${^RE_COMPILE_RECURSION_LIMIT}</code>
Match start offsets	<ul style="list-style-type: none"> <code>@LAST_MATCH_START</code> <code>@-</code> 	Match ends offsets	<ul style="list-style-type: none"> <code>@LAST_MATCH_END</code> <code>@+</code>
Last successful pattern	<code>\${^LAST_SUCESSFUL_PATTERN}</code>	Result of last successful regexp assertion	<ul style="list-style-type: none"> <code>\$^R</code> <code>\$LAST_REGEXP_CODE_RESULT</code>
regexp debug flag	<code>\${^RE_DEBUG_FLAG}</code>	regexp internal optimization/memory	<code>\${^RE_TRIE_MAXBUF}</code>
• Format Variables			
The format mechanism is use to generate printed layouts. It's an old Perl feature but still useful in various places.			
Current value of the <code>write()</code> accumulator for <code>format()</code> lines.	<ul style="list-style-type: none"> <code>\$ACCUMULATOR</code> <code>\$^A</code> 		
Form feed format defaults to <code>\f</code>	<ul style="list-style-type: none"> <code>IO::Handle->format_formfeed(EXPR)</code> <code>\$FORMAT_FORMFEED</code> <code>\$^L</code> 	Set of characters after which a string may be broken to fill continuation fields	<ul style="list-style-type: none"> <code>IO::Handle->format_line_break_characters EXPR</code> <code>\$FORMAT_LINE_BREAK_CHARACTERS</code> <code>\$:</code>
Number of lines left on the page on currently selected output channel	<ul style="list-style-type: none"> <code>HANDLE->format_lines_left(EXPR)</code> <code>\$FORMAT_LINES_LEFT</code> <code>\$-</code> 	Current page length of current output channel	<ul style="list-style-type: none"> <code>HANDLE->format_lines_per_page(EXPR)</code> <code>\$FORMAT_LINES_PER_PAGE</code> <code>\$=</code>
Name of current top-page format of output channel	<ul style="list-style-type: none"> <code>HANDLE->format_top_name(EXPR)</code> <code>\$FORMAT_TOP_NAME</code> <code>\$^</code> 	Report format name of output channel	<ul style="list-style-type: none"> <code>HANDLE->format_name(EXPR)</code> <code>\$FORMAT_NAME</code> <code>\$~</code>
• Error Variables			
The variables <code>\$?</code> , <code>!</code> , <code>^E</code> , and <code>\$?</code> contain information about different types of error conditions that may appear during execution of a Perl program. They correspond to errors detected by the Perl interpreter, C library, operating system, or an external program, respectively.			
Perl error from the last <code>eval</code> operator	<ul style="list-style-type: none"> <code>\$EVAL_ERROR</code> <code>\$@</code> 	Current state of interpreter	<ul style="list-style-type: none"> <code>\$EXCEPTIONS_BEING_CAUGHT</code> <code>\$^S</code>
Current value of C errno integer variable	<ul style="list-style-type: none"> <code>\$OS_ERROR</code> <code>\$ERRNO</code> <code>\$!</code> 	<code>\$!</code> returns the system variable <code>errno</code> when used in a numeric context, but returns the string from <code>pererror()</code> when used in string context.	<ul style="list-style-type: none"> Hash of error names to 0 or 1, set to 1 if current error is this error. <code>%OS_ERROR</code> <code>%ERRNO</code> <code>%!</code>
OS detected error	<ul style="list-style-type: none"> <code>\$EXTENDED_OS_ERROR</code> <code>\$^E</code> 		
Status returned by last pipe close, backtick command, <code>wait</code> , <code>waited</code> , or <code>system()</code> call.	<ul style="list-style-type: none"> <code>\$CHILD_ERROR</code> <code>\$?</code> 	native status returned by last pipe close , backtick command, <code>wait()</code> or <code>waitpid()</code> or <code>system()</code> call	<code>\${^CHILD_ERROR_NATIVE}</code>
Current value of warning switch	<ul style="list-style-type: none"> <code>\$WARNING</code> <code>\$^W</code> 	Current set of warning checks enabled by the use warnings pragma	<code>\${^WARNING_BITS}</code>
• Variables related to the interpreter state			
These variables provide information about the current interpreter state.			
Flag associated with the <code>-c</code> switch	<ul style="list-style-type: none"> <code>\$COMPILING</code> <code>\$^C</code> 	The current value of the debugging flags	<ul style="list-style-type: none"> <code>\$DEBUGGING</code> <code>\$^D</code>
Current phase of the perl interpreter	<code>\${^GLOBAL_PHASE}</code>	Debugging support. Internal variable.	<ul style="list-style-type: none"> <code>\$PERLDB</code> <code>\$^P</code>
Compile-time hints for the perl interpreter. Internal use only	<code>\$^H</code>	Values of compiled statements	<code>%^H</code>
Taint mode	<code>\${^TAINT}</code>	Safe locale operations availability	<code>\${^SAFE_LOCALES}</code>
Input/Output Layers. Internal use by <code>PerlIO</code> only.	<code>\${^OPEN}</code>	Unicode Settings of Perl	<code>\${^UNICODE}</code>
Internal UTF-8 offset caching code state	<code>\${^UTF8CACHE}</code>	State of UTF-8 locale detected by perl at startup.	<code>\${^UTF8LOCALE}</code>
• File handle Variables			
See also: Perl File Handles The following variables are used in the Input/Output handling as well as program arguments.			
Name of current file read from <code><></code>	<code>\$ARGV</code>	Command line arguments of the script ← See diamond operator <> . →	<ul style="list-style-type: none"> <code>@ARGV</code> Number of arguments minus one <code>#\$ARGV</code>
Special file handle that iterates over command-line filenames in <code>@ARGV</code>	<code>ARGV</code>	Special file handle that points to currently open output file when doing edit-in-place processing	<code>ARGVOUT</code>
Output field separator for the print operator	<ul style="list-style-type: none"> <code>IO::Handle->output_field_separator(EXPR)</code> <code>\$OUTPUT_FIELD_SEPARATOR</code> <code>\$OFS</code> <code>\$,</code> 	Current line number for the last file handled accessed	<ul style="list-style-type: none"> <code>HANDLE->input_line_number(EXPR)</code> <code>\$INPUT_LINE_NUMBER</code> <code>\$NR</code> <code>\$.</code>
Input record separator (newline by default)	<ul style="list-style-type: none"> <code>\$RS</code> <code>\$/</code> <code>IO::Handle->input_record_separator(EXPR)</code> <code>\$INPUT_RECORD_SEPARATOR</code> 	Output record separator	<ul style="list-style-type: none"> <code>\$ORS</code> <code>\$\</code> <code>IO::Handle->output_record_separator(EXPR)</code> <code>\$OUTPUT_RECORD_SEPARATOR</code>
Auto-flush control <ul style="list-style-type: none"> order of output @ Perl Maven Suffering from Buffering? 	<ul style="list-style-type: none"> <code>HANDLE->autoflush(EXPR)</code> <code>\$OUTPUT_AUTOFLUSH</code> <code>\$!</code> 	Perl activates file buffering by default. Assign 1 to <code>\$!</code> to activate auto-flush.	<ul style="list-style-type: none"> Last read file handle <code>\${^LAST_FH}</code>

Perl 5 Input/Output 🚧

Perl I/O	<ul style="list-style-type: none"> • open @ perldoc browser • Writing to files with Perl @ Perl Maven • open file in-memory @ stackOverflow 	<ul style="list-style-type: none"> • Stupid open() tricks @Perl.com: <ul style="list-style-type: none"> • No explicit filename • create an anonymous temporary file 	<ul style="list-style-type: none"> • print to a string • read lines from a string 		
print, printf, sprintf	<p>print, printf, sprintf (which describes the format). Note: print, a list operator, is more efficient than printf. print and printf output to stdout by default, but accept a file handle as the first argument if it is NOT followed by a separating comma (a ',' puts it in the list to print!)</p>				
say	<p>use <code>feature qw(say);</code> or use <code>v5.10;</code> (or higher). Like <code>print</code>, but implicitly appends a newline at the end of the list.</p>				
diamond operator <>	<ul style="list-style-type: none"> • Both <code><></code> and <code><<>></code> operators read the content of files listed on the command line via <code>@ARGV</code>. • The <code><></code> operator supports shell redirection and pipe operations which <code><<>></code> does not allow (for security reasons). <ul style="list-style-type: none"> • The <code><<>></code> operator is always empty. • With <code><<></code> is used, if there is nothing on the command line of the program or a dash (-) is present the command line identifies stdin. Not so for the <code><<>></code> operator. • The <code><></code> operator, depending on what's inside it, is an exact synonym for either the readline or glob function (but this does not apply to the <code><<>></code> operator): <ul style="list-style-type: none"> • If <code><></code> contains only a bareword or a simple scalar variable, it compiles to readline, otherwise it compiles to glob. 				
The double diamond, a more secure <<>> (Perl >= v5.22)					
In-place-editing ⚠ The <code><></code> operator tries to duplicate the original file's permission and ownership.	<pre>print <>;</pre>	<pre>← Simple implementation of /bin/cat</pre>	<pre>print <<>>;</pre>	<pre>← safer one</pre>	Redirection cannot be forced via file names embedding them with the <code><<>></code> operator.
	<pre>print sort <>;</pre>	<pre>← Simple implementation of /bin/sort</pre>	<pre>print sort <<>>;</pre>	<pre>← safer one</pre>	
	Set <code>\$^I</code> to a backup file extension (such as Emacs <code>"~"</code> or <code>".bak"</code>) to change the behaviour of the <code><></code> and <code><<>></code> operators and <code>print</code> . In a <code>while (<>) {...}</code> loop, when <code>\$^I</code> is not <code>undef</code> (its default), Perl: <ul style="list-style-type: none"> • renames currently processed file with the specified extension added, • opens a new file with the original name • prints into the new file. • Any modification goes into the new file: in-place-editing it! 		<pre>use strict; \$I = "~"; # rename old file: add '~' to it's name (Emacs-style backup) while (<>) { s/something/Something else/; # perform any substitution print; }</pre>		
perl -i cmdline option	It's also possible to do this on the command line! For example:		<pre>perl -p -i -w -e 's/something/Something else/g' data*.dat</pre>		
Special filehandle names	ARGV	The special filehandle that iterates over command-line filenames in <code>@ARGV</code> . Usually written as the null filehandle in the angle operator <code><></code> (or <code><<>></code>)			
Also See:	ARGVOUT	The special filehandle that points to the currently open output file when doing edit-in-place processing with <code>-i</code> . <ul style="list-style-type: none"> • Useful when you have to do a lot of inserting and don't want to keep modifying <code>\$_</code> 			
• File handle Variables section above.	STDIN	<STDIN> : line input operator for the STDIN filehandle (for the standard input). <ul style="list-style-type: none"> • Each time <code><STDIN></code> is used in scalar context, Perl reads 1 complete line of the standard input and uses it as the value of <code><STDIN></code>. <ul style="list-style-type: none"> • The string includes a line termination character. Use the chomp built-in function to strip it off the variable. • If <code><STDIN></code> is read in list context, it returns all lines inside a list! For example, <code>foreach (<STDIN>) { ... }</code> reads the entire stdin in 1 step: <code>\$_</code> holds it all! 			
• <code>open</code>		<pre>while (<STDIN>) { # print all print; # lines of } # stdin</pre>	<pre>while (defined(\$_ = <STDIN>)) { print \$_; }</pre>	The code in the left-most cell is the shortest form. It is equivalent to the code beside it; each line of stdin is stored in the default variable <code>\$_</code> and the loop stops on end at which time <code><STDIN></code> returns <code>undef</code> .	
• <code>open::layers</code>	STDOUT	standard output			
Also see process and filehandles inside the Topic: Process Control below.	STDERR	standard error Note: generally <code>STDERR</code> is not buffered, while <code>STDOUT</code> is buffered by default. Text sent on <code>STDERR</code> may show up before <code>STDOUT</code> . <ul style="list-style-type: none"> • Print a new line on <code>STDOUT</code> to help flushing it or assign 1 to <code>\$ </code> to activate auto-flush. 			
	DATA				
Using lexical scalar filehandles	<code>open</code> also supports the use of lexical scalar filehandles, a more versatile and safer mechanism. <ul style="list-style-type: none"> • The file handle can be declared inside the statement as shown below. • It can also be declared before, but the file handle variable must be <code>undef</code> when the <code>open</code> statement executes, otherwise <code>open</code> uses it as a file handle value. 				
Example from Grinnz :	<ul style="list-style-type: none"> • <code>open my \$in_fh, '<', \$filename</code> or <code>die "Failed to open \$filename for reading: \$!";</code> • <code>open my \$out_fh, '>>:encoding(UTF-8)', \$outfile</code> or <code>die "Failed to open \$outfile for appending: \$!";</code> 				

Perl 5 Built-in Functions 🚧

Perl Functions Perl syntax	🗨️ To get information about a Perl function from the command line: use the <code>perldoc -f</code> command. To get information about <code>print</code> use: <code>perldoc -f print</code> This PDF refers to several Perl built-in functions in various places.
⚠ Cautionary notes	Some of the Perl functions exhibit various limitations and the vary over Perl versions. This section describes the ones I am aware and the proposed alternatives.
<ul style="list-style-type: none"> • <code>each</code> keyword is broken • Use <code>Var::Pairs</code> instead. 	Do NOT use the built-in <code>each</code> . It is broken, as described by Damian Conway in his Modern Perl Best Practice O'Reilly course , section control structure. <ul style="list-style-type: none"> • <code>each</code> is not re-entrant: <ul style="list-style-type: none"> • nested loops of <code>each</code> over the same hash does not work as expected and will create infinite loop since the nested loop <code>each</code> juts iterates from where the first loop each left it. • Exiting the loop leaves the state of the <code>each</code> internal pointer at the current location. • If you use <code>each</code> on the same hash later it will resume from where it left, it will not start from the beginning.

Perl 5 Statements 🚧

Perl Syntax	<code>perldoc perlsyn</code> : Perl syntax is free-form. It borrowed concepts from many languages. See <code>perldoc perlttrap</code> for comparisons and differences.	
Comments	Comments start with a <code>#</code> on a line, outside of a string or regular expression.	
Statement separator	Every statement must be terminated by a semicolon, except for the last statement of a block where it is optional. It is however customary to put it anyway.	
No semicolon after a block	A block is not followed by a semicolon. Note, however that <code>eval {}</code> , <code>sub {}</code> , and <code>do {}</code> need explicit termination because these are not compound statements but just terms inside an expression.	
Statement modifiers	A simple statement may be followed by a <i>single</i> modifier just before the terminating semicolon:	
⚠ Do not use with a <code>my state</code> and <code>our</code> .	<pre>if EXPR</pre>	<pre>while EXPR</pre>
	<pre>unless EXPR</pre>	<pre>until EXPR</pre>
Compound statements	A sequence of statements inside a file, a <code>{}</code> delimited block, or an <code>eval</code> string constitute a scope. <ul style="list-style-type: none"> • Because hash references are also identified by <code>{}</code>, it may be necessary to put a semicolon after the opening brace to identify a block. As in: <code>{; ... }</code> • Inside all following, a BLOCK is always enclosed by braces, as in <code>{...}</code>, even for if statements. • In loops, the <code>continue</code> control statement identifies a BLOCK that is executed before the loop condition is evaluated again. 	
Control flow Statements:	<pre>if (EXPR) BLOCK if (EXPR) BLOCK else BLOCK if (EXPR) BLOCK elsif (EXPR) BLOCK ... if (EXPR) BLOCK elsif (EXPR) BLOCK ... else BLOCK</pre>	<pre>unless (EXPR) BLOCK unless (EXPR) BLOCK else BLOCK unless (EXPR) BLOCK elsif (EXPR) BLOCK ... unless (EXPR) BLOCK elsif (EXPR) BLOCK ... else BLOCK</pre>
while and unless loops	<pre>LABEL while (EXPR) BLOCK # run while EXPR is true LABEL while (EXPR) BLOCK continue BLOCK</pre>	<pre>LABEL until (EXPR) BLOCK # run while EXPR is false LABEL until (EXPR) BLOCK continue BLOCK</pre>
for and foreach loops	<pre>LABEL for (EXPR; EXPR; EXPR) BLOCK LABEL for VAR (LIST) BLOCK LABEL for VAR (LIST) BLOCK continue BLOCK</pre>	<pre>LABEL foreach (EXPR; EXPR; EXPR) BLOCK LABEL foreach VAR (LIST) BLOCK LABEL foreach VAR (LIST) BLOCK continue BLOCK</pre>
switch statement	<pre>given (EXPR) BLOCK # in switch statements. (Perl >= v5.14). It was available in Perl 5.10.0 but did not work properly until 5.10.1</pre>	
Iterate over multiple values at a time	<pre>LABEL for my (VAR, VAR) (LIST) BLOCK (Perl >= 5.36) LABEL for my (VAR, VAR) (LIST) BLOCK continue BLOCK LABEL foreach my (VAR, VAR) (LIST) BLOCK LABEL foreach my (VAR, VAR) (LIST) BLOCK continue BLOCK</pre>	
Basic Blocks	A BLOCK by itself is semantically equivalent to a loop that executes once, allowing loop control keywords (see below).	<pre>LABEL BLOCK LABEL BLOCK continue BLOCK</pre>
Defer blocks	A block prefixed by the <code>defer</code> modifier provides a section of code which runs at a later time during scope exit. Requires: <code>use feature 'refer'; (Perl >= 5.36)</code>	

Try Catch exceptions	<ul style="list-style-type: none"> The try/catch syntax provides flow control exception handling. This syntax must be first enabled with <code>use feature 'try'</code>; The finally block is experimental. It cannot return, goto or use loop controls. <pre>try BLOCK catch (VAR) BLOCK try BLOCK catch (VAR) BLOCK finally BLOCK</pre>	<pre>use feature 'try'; try { my \$x = call_a_function(); \$x > 0 or die "Negative not supported"; do_something_with(\$x); } catch (\$e) { warn "Unable to output a value; \$e"; }</pre>
Loop control	The following built-in functions can be used inside the above loops.	
<ul style="list-style-type: none"> Use the last and redo inside a naked block of code to control looping. 	loop control keywords: <ul style="list-style-type: none"> last <code>o</code>: exits the loop. next <code>o</code>: starts the next iteration of the loop. redo <code>o</code>: restarts the loop block without evaluating the condition again. 	The last , next , and redo loop control keywords work in the following constructs: <ul style="list-style-type: none"> <code>while (condition) { ... }</code> <code>until (condition) { ... }</code> <code>for (init; condition; continue) { ... }</code> <code>foreach array { ... }</code> naked block: <code>{ ... }</code>
Specially Named Blocks PHASE BLOCK	5 specially named blocks are run at the various phase of a running program: BEGIN , UNITCHECK , CHECK , INIT and END . See: BEGIN block - running code during compilation. Note the security risk warnings . The BEGIN block is used to implement other Perl functionality.	
Statement modifiers	<ul style="list-style-type: none"> if EXPR unless EXPR while EXPR until EXPR for LIST foreach LIST when EXPR 	The for and foreach statements impose a list context ; the complete list is processed. Therefore a loop like the following trying to stop on a line that has <code>"_END_"</code> on it will not work since it reads all of STDIN: <pre>foreach (<STDIN>) { last if ?_END_/; ...; }</pre>
do block	<ul style="list-style-type: none"> The do block is *very useful* to set a value based on several conditions, just as the <code>?: conditional operator</code> but with an explicit block that may use scoped variables. Takes advantage of a block value is the value of the last expression executed inside the block. Do <code>"not"</code> return from the block. The last, next and redo cannot be used inside do blocks. The do blocks are <i>not</i> semantically equivalent to loop blocks. 	<pre>my \$next_step = do { my (\$perl_nirvana, \$emacs_nirvana) = check-nirvana-levels(); if (\$perl_nirvana < 5 && \$emacs_nirvana < 8) { 'study-Perl' } elsif (some_other_cond()) { 'time-to-cook' } elsif (\$emacs_nirvana < 7) { 'look-into-eieio' } else { \$isit_winter? 'go-skiing' : 'go-canoeing' } }</pre>
goto statement	Perl supports 3 forms of goto statements: <code>goto-LABEL</code> , <code>goto-EXPR</code> , and <code>goto-&NAME</code> . Note that loops labels cannot be used.	

Perl 5 Subroutines 🚧

Perl subroutines Object Oriented Perl, 2.1.4	<ul style="list-style-type: none"> Parentheses are optional when calling a subroutine. In some cases, using them prevents mis-interpretations. Also note that blocks are often passed as first argument to a subroutine. 																												
<ul style="list-style-type: none"> Declaring subroutine In all cases, it's less ambiguous to define the subroutine before use and use parentheses in calls. 	<ul style="list-style-type: none"> Declare a subroutine to use as a list operator. <ul style="list-style-type: none"> use <code>or</code> or not <code> </code> because it binds too tightly. Declare a subroutine to use as a unary operator: 	<pre>sub seed_for; \$val = seed_for \$0 or die 'seed_for failed'; sub seed_for(\$); # use subroutine prototype to declare it as unary operator. \$val = seed_for \$0 die 'seed_for failed';</pre>																											
<ul style="list-style-type: none"> Defining subroutine 	<ul style="list-style-type: none"> Defined with the <code>sub</code> keyword followed by a block. 	<pre>sub greet { print "hello!\n"; }</pre>																											
<ul style="list-style-type: none"> Calling a subroutine 	<ul style="list-style-type: none"> If the subroutine definition follows its invocation, parentheses after the subroutine name are required, as in: <code>greet()</code>; 	<ul style="list-style-type: none"> But if the definition was above the call, the parentheses are optional; as in: <code>greet</code>; Subroutine sigil is <code>&</code>. It can optionally be used in a call; as in <code>&greet</code>; or <code>&greet()</code>; 																											
<ul style="list-style-type: none"> pass current <code>@_array</code> 	<ul style="list-style-type: none"> Call with <code>&</code> prefix without args, as in <code>&sub_function</code>; to pass current <code>@_array</code>. Used to call a helper subroutine with in the primary one, providing all its arguments. 																												
<ul style="list-style-type: none"> goto 	<ul style="list-style-type: none"> From a subroutine use <code>goto &sub_function</code>; to transfer control to that subroutine instead of calling it. It also passes the current <code>@_array</code> to it. 																												
<ul style="list-style-type: none"> calling a method 	<ul style="list-style-type: none"> Parentheses are required if arguments are passed to method, but optional if there is no arguments. 	<pre>\$obj->method_with_args(\$val1, \$valb); \$obj->method_without_arg; \$obj->method_without_args();</pre>																											
<ul style="list-style-type: none"> subroutine & 	<ul style="list-style-type: none"> Why we teach the subroutine ampersand Why should I use the & to call a Perl subroutine? @ StackOverflow 	<ul style="list-style-type: none"> Another point of view: Subroutines and Ampersands Note it must be used to make a reference to a subroutine: <code>\$greeter = \&greet</code>; 																											
<ul style="list-style-type: none"> subroutine arguments passed by list <ul style="list-style-type: none"> always variable by nature named arguments <p>Note: The <code>@_</code> is an alias to the passed values; changing them inside the subroutine affects the caller's values.</p>	<ul style="list-style-type: none"> The arguments passed to a subroutine are available to its code via the special <code>@_array</code>. The caller code supplies a list of values. Lists lists are flattened in Perl. Since hash declaration take a list of key/value pairs, it's easy to implement a passing named arguments! It's also possible for the subroutine to set defaults for some of the expected arguments by taking advantage of the fact that hash are lists, list are flattened and hash can be assigned a list with the last values are used. 	<pre>@sorted = alpha_order('Nice', 'Québec', 'Montréal'); @sorted = number_order @unsorted_numbers; @sorted = alpha_order('Trois-Rivières', @sorted, 'Gaspé', 'Rimouski');</pre> <p>Implementation: <code>sub move { my (%directions) = @_; ... }</code> Caller: <code>move(up=>3, left=>4); move('down', 2);</code> # it's by convention! To set a default: <pre>sub move { %default = (up=>0, down=0, left=>0, right=>0); my (%directions) = (%default, @_); ... }</pre></p>																											
Subroutine Prototypes	An older Perl feature. Clashes with subroutine signatures as of Perl v5.20. In <i>Perl >= v5.20</i> put the :prototype attribute before subroutine prototype parenthesis.																												
Subroutine signatures <ul style="list-style-type: none"> <i>Perl >= 5.36</i>: Stable <i>Perl >= 5.20</i>: Experimental See: Use v5.20 subroutine signatures	<table border="1"> <tr> <td>Exactly zero arguments</td> <td><code>()</code></td> <td>Zero or 1 argument, no default, unnamed:</td> <td><code>(\$=)</code></td> </tr> <tr> <td>Zero or 1 argument, no default, named</td> <td><code>(\$val=)</code></td> <td>Zero or 1 argument, named, with default</td> <td><code>(\$val=1)</code></td> </tr> <tr> <td>exactly 1 named argument:</td> <td><code>(\$val)</code></td> <td>Exactly 2 arguments</td> <td><code>(\$v1, \$v2)</code></td> </tr> <tr> <td>2, 3 or 4 arguments no defaults:</td> <td><code>(\$v1, \$v2, \$=, \$=)</code></td> <td>2,3 or 4 arguments, 1 default:</td> <td><code>(\$v1, \$v2, \$v3='a', \$=)</code></td> </tr> <tr> <td>Two or more, any number of arguments.</td> <td><code>(\$v1, \$v2, @)</code></td> <td>Two or more arguments, remainders into a named array:</td> <td><code>(\$v1, \$v2, @rest)</code></td> </tr> <tr> <td>Two or more arguments: an even number</td> <td><code>(\$v1, \$v2, %)</code></td> <td>Two or more arguments, remainders into a named hash:</td> <td><code>(\$v1, \$v2, %rest)</code></td> </tr> <tr> <td>Class method</td> <td><code>(\$class, ...)</code></td> <td>Object method</td> <td><code>(\$self, ...)</code></td> </tr> </table>	Exactly zero arguments	<code>()</code>	Zero or 1 argument, no default, unnamed:	<code>(\$=)</code>	Zero or 1 argument, no default, named	<code>(\$val=)</code>	Zero or 1 argument, named, with default	<code>(\$val=1)</code>	exactly 1 named argument:	<code>(\$val)</code>	Exactly 2 arguments	<code>(\$v1, \$v2)</code>	2, 3 or 4 arguments no defaults:	<code>(\$v1, \$v2, \$=, \$=)</code>	2,3 or 4 arguments, 1 default:	<code>(\$v1, \$v2, \$v3='a', \$=)</code>	Two or more, any number of arguments.	<code>(\$v1, \$v2, @)</code>	Two or more arguments, remainders into a named array:	<code>(\$v1, \$v2, @rest)</code>	Two or more arguments: an even number	<code>(\$v1, \$v2, %)</code>	Two or more arguments, remainders into a named hash:	<code>(\$v1, \$v2, %rest)</code>	Class method	<code>(\$class, ...)</code>	Object method	<code>(\$self, ...)</code>
Exactly zero arguments	<code>()</code>	Zero or 1 argument, no default, unnamed:	<code>(\$=)</code>																										
Zero or 1 argument, no default, named	<code>(\$val=)</code>	Zero or 1 argument, named, with default	<code>(\$val=1)</code>																										
exactly 1 named argument:	<code>(\$val)</code>	Exactly 2 arguments	<code>(\$v1, \$v2)</code>																										
2, 3 or 4 arguments no defaults:	<code>(\$v1, \$v2, \$=, \$=)</code>	2,3 or 4 arguments, 1 default:	<code>(\$v1, \$v2, \$v3='a', \$=)</code>																										
Two or more, any number of arguments.	<code>(\$v1, \$v2, @)</code>	Two or more arguments, remainders into a named array:	<code>(\$v1, \$v2, @rest)</code>																										
Two or more arguments: an even number	<code>(\$v1, \$v2, %)</code>	Two or more arguments, remainders into a named hash:	<code>(\$v1, \$v2, %rest)</code>																										
Class method	<code>(\$class, ...)</code>	Object method	<code>(\$self, ...)</code>																										
Returned value.	<ul style="list-style-type: none"> The result of the last evaluated expression is implicitly returned. The return operator can be used but it's not required unless used to change execution flow (return immediately from the subroutine). The subroutine can return a scalar in scalar context or a list if called in list context. <ul style="list-style-type: none"> Inside the subroutine, use the <code>wantarray</code> function to determine the calling context of the subroutine call and why it should return: 																												
Detecting calling context with <code>wantarray</code>																													
Identify caller	The <code>caller</code> built-in returns information about the subroutine caller inside an array: (package, file_name, file_line). In scalar context it returns the package only.																												
AutoLoading	On a call to undefined subroutine Perl checks if the package defines an <code>\$AUTOLOAD</code> subroutine it calls that.	Also see: AutoLoader .																											
Continuation with goto	The <code>goto</code> built-in can be used by a subroutine to continue its execution into another subroutine. Not for all but useful in some specific cases such as autoloading .																												

Perl 5 Classes, Objects and Methods 🚧

Object Oriented Perl <ul style="list-style-type: none"> Perl OO Tutorial Perl Module Library Module creation guideline 	To build a Perl class with common Perl: 1) create a package with the name of the class inside a module, 2) write functions in the package, 3) bless a referent.
<ul style="list-style-type: none"> By convention, something a name that starts with an underscore is <i>internal</i>, not meant to be used directly. <ul style="list-style-type: none"> There is nothing preventing direct access, but users of the class should not access it directly (as OO design principles recommend). Perl ignore prototypes of methods. It's possible to create class methods and class attributes: Their scope must be the scope of the module they are defined in. Destructors are normally not required, as Perl automatically destroys objects at their end-of-life based on scope. It's needed when classes use circular references. <ul style="list-style-type: none"> It is possible to create explicit destructor by defining a DESTROY method in the class. See The destructor called DESTROY and Object Oriented Perl book. Inheritance: parent classes are identified in the <code>@ISA array</code>. In code set them by identifying them via the <code>use parent pragma</code>. <ul style="list-style-type: none"> See the <code>isa</code> class instance operator. 	

<p>Other:</p> <ul style="list-style-type: none"> Object Oriented Perl by Damian Conway Corinna Class Tutorial <p>See also Perl extension for OO:</p> <ul style="list-style-type: none"> Perl Moose @ wikipedia Moose home Moose @ meta::Cpan 	<pre>use Employee; use strict; # By using the package name and the arrow operator to refer # to the new method, Perl passes the string "Employee", the # class name, to the first argument. This is used by the bless # built-in to turn the anonymous hash objref into an # Employee class reference. my \$empl = Employee->new('Pete', 'V.P.');</pre> <p># The Employee::new method returns a reference to the object. It can be used to call other methods, which also pass the object reference as the first argument.</p> <pre>\$empl->set_office('L1-100');</pre> <p>Note that calling Employee::new directory, no object reference is passed; therefore the arrow notation is required.</p>	<pre>package Employee; # a very simple/naive class implementation sub new { # A class construction method, conventional name: new my \$class = \$_[0]; # first argument is class name (a string) my \$objref = { # following arguments passed to Employee->new() _name = \$_[1], # by convention, names of class attributes start with _role = \$_[2], # an underscore. Access them only inside the methods }; # but Perl provides no access protection. ... bless \$objref, \$class; # bless object referent as a class, return it from new() } sub set_office { # first argument is the class instance my (\$self, \$office_ID) = @_; # it's assigned to self: the reference to the object \$self->{_office_ID} = \$office_ID; }</pre>
<p>The tie function</p> <ul style="list-style-type: none"> Some references: Some modules 	<p>The tie function is used to associate a behaviour provided by package subroutines to a specific variable or handle.</p> <ul style="list-style-type: none"> It's possible to tie operations on a scalar, array, hash or handle to a specific variable. The operations are controlled by the package subroutines that have preselected names for various operations depending on the type selected (scalar, hash, array or handle). <ul style="list-style-type: none"> Object Oriented Perl by Damian Conway. Ties, chapter 9. The Magic of Tied Variables. Mastering Perl <ul style="list-style-type: none"> Magical tied scalars, Brian D. Foy <p>• Tie::Simple • Tie::File • Tie::IxHash</p>	<ul style="list-style-type: none"> Changing Hash Behaviour with tie, by Dave Cross Sorted hash in Perl using Tie::IxHash @ PerlMaven <p>• Tie::Array • Tie::Array::CSV</p>
<p>Operator overloading</p>	<p>Operator overloading is provided by the overload.pm module written by Dr. Ilya Zakharevich.</p>	<ul style="list-style-type: none"> Object Oriented Perl by Damian Conway. Operator Overloading, chapter 10. Overloading by Dave Cross

Perl 5 Modules 🚧

Perl Modules	Note that module files must end with a true value. It is customary to place a 1; on the last non-commented line that.	
Perl core modules	<ul style="list-style-type: none"> How to detect where a module is installed : <code>perldoc -l Module</code> How to check if a module is part of Perl core : <code>corelist Module</code> (Perl >= v5.9.2) 	
Access to Modules	Provide access to modules in your code with one of the following: do , require or use	
<p>Modules @perltutorial</p> <p>Modules</p> <p>Using simple modules</p>	<p>do</p> <p>Looks for the module file by searching the @INC path. Performed at run time (and therefore can be done conditionally).</p> <ul style="list-style-type: none"> If Perl finds the file, it places the code inside the calling program and executes it. Otherwise, Perl will skip the do statement silently. The "included" code does not have access to the lexical variables from the main program. Skip the @INC path lookup if given a file path starting with <code>./</code>, <code>../</code>, or <code>/</code> 	
	<p>require</p> <p>Loads the module file once, also searching the @INC path. Performed at run time (and therefore can be done conditionally).</p> <ul style="list-style-type: none"> If the require for the same file appears twice, Perl ignores it. Perl will issue an error message if it cannot find the file (as opposed to do). Skip the @INC path lookup if given a file path starting with <code>./</code>, <code>../</code>, or <code>/</code> 	
<p>The normal way to access Perl modules</p>	<p>use</p> <p>Similar to require except that Perl applies it before the program starts: it's done at compile time. Modify it dynamically in a BEGIN block. See IntPo.</p> <ul style="list-style-type: none"> Therefore the use statement cannot be invoked inside conditional statements such as if-else. Used often to include a module in a program. That imports the defaults as defined by the module's code. <p>Select what to import with one of the two equivalent forms: (See IntPo):</p> <ul style="list-style-type: none"> <code>use Module::Name ('function_a', 'function_b');</code> <code>use Module::Name qw(function_a function_b);</code> <code>use Module::Name ();</code> # import nothing. All accesses to the module must be done with <code>Module::Name::something</code> 	
<p>Error handling for: Can't locate in @INC</p> <ul style="list-style-type: none"> How to fix that <p>See Also: IntPo</p> <ul style="list-style-type: none"> See: <code>show-perl-inc @ USRHOME</code> 	<p>For the above statements to work Perl must be able to identify the location of the requested module(s).</p> <ul style="list-style-type: none"> Perl looks for a module code inside the directories identified by the @INC array. <p>if you have. <code>use The::Module;</code> inside your code, Perl looks for a sub-directory named 'The' containing a file named 'Module.pm' inside each @INC directory.</p> <p>If Perl does not find it, there are multiple ways to solve the problem:</p> <ul style="list-style-type: none"> Add the required directory to the list of directories identified in the ':' separated list in the PERL5LIB environment variable. (use ';' as separators in Windows). Add a <code>use lib 'path/to/the/directory';</code> statement inside your Perl file to add the required directory when executing a specific piece of Perl code, at compile time. Run Perl with the -I (capital i) option to run the code with the extra directory added to @INC array. <p>To List the directories used by Perl from one of the following equivalent command lines:</p> <ul style="list-style-type: none"> <code>perl -e 'print join("\n", @INC), "\n";'</code> <code>perl -le 'print for INC;'</code> <p style="text-align: right;">You can also get more information with <code>perl -v</code></p>	
<p>Declare packages</p>	<p>In Perl a package can span several files and one file may contain the code of several packages. The package starts with the package keyword. The special __PACKAGE__ literal contains the name of the current package.</p> <ul style="list-style-type: none"> The default package is the main package. Code at the before the first package declaration in a file belongs to the main package. 	

Topic: Data Introspection 🚧

Data Introspection			
<p>Using Perl Debugger</p> <ul style="list-style-type: none"> Debugger Tutorial 	<p>Debug a program:</p> <pre>perl -d program_name program_args</pre>		
	<p>Debug interactive session:</p> <pre>perl -d -e 0</pre>		
<p>Debugger commands</p>	<p>q Quit debugger</p> <p>h help. List all available commands.</p>	<p>s single step</p> <p>x evaluate expression</p>	
<p>Modules for Data introspection</p>	<p>Data::Dumper (Perl >= 5.005) It provides the Dumper function that prints strings that can be used by eval to rebuild the data.</p> <p>Data::Dump (Requires Perl >= v5.6.0)</p> <p>Data::Printer A nicer data dumper, not eval compatible.</p>	<ul style="list-style-type: none"> It is similar to the x command of the debugger. Pass reference to the variables, otherwise it extends them to list and show each entry as its own variable. <p>Provides a dump function that has nicer output, but is not eval compatible.</p> <ul style="list-style-type: none"> dump() prints on the stdout. No need to use print. <ul style="list-style-type: none"> It provides the p subroutine that does not require a reference to the variable as it inspects it first. p() prints on the stdout. No need to use print. 	<ul style="list-style-type: none"> <code>print Dumper(\@array);</code> <code>print Dumper \%hash;</code> <p><code>use Data::Dump qw(dump);</code> <code>dump(\@array);</code> <code>dump(\%hash);</code></p> <p><code>use Data::Printer;</code> <code>p(@array);</code> <code>p(%hash);</code></p>
<p>Data Marshalling</p> <ul style="list-style-type: none"> Data Serialization 	<p>There are several modules, either part of Perl core or outside, that provides mechanism to marshal/serialize and unmarshal/de-serialize data.</p> <ul style="list-style-type: none"> See the links at left for more info. 		
<p>perl-live-coding</p> <ul style="list-style-type: none"> Demo screencast 	<p>This third party package creates a very good Perl REPL. It can be used outside and inside of Emacs.</p> <ul style="list-style-type: none"> When used inside Emacs it can evaluate Perl code by line, marked area and display the results in a secondary buffer. Highly recommended. 		
<p>Analysing Perl Code with Doxygen</p>	<ul style="list-style-type: none"> Install Doxygen::Filter::Perl with cpan. See https://github.com/jordan2175/doxygen-filter-perl 		

Topic: Directory Operations 🚧

Directory Operations	In Books: LPo		
Opening Files	All file open operations are relative to the <i>current working directory</i> (for relative file names)		<code>open my \$filehandle, '<:utf8', 'a_relative/path.txt'</code>
Creating temporary files	File::Temp (Perl >= v5.6.1). Using <code>File::Temp</code> . Also see IO::File		
Built-in Functions	Related Functions/Packages / Descriptions	Notes	
Getting file names by: <ul style="list-style-type: none"> • Globbing : <ul style="list-style-type: none"> • with <code>glob</code> • with the glob operator <code><></code> 	File::Glob (Perl >= v5.6.0) - provides more control.	Example:	<pre>my @all_files = glob '*'; my @perl_files = glob '*.pm *.pl'; # 2 globs, space-separated</pre>
	The <code><></code> operator is identifying: <ul style="list-style-type: none"> • a filehandle, when: the item inside <code><></code> is a Perl identifier or an indirect file handle read scalar, • a glob expression otherwise. 	Glob examples:	<pre>my @all_files = <'*>; my @all_files = <*>; # 1 glob: no space, no need for string my @perl_files = <*.pm *.pl*>; # 2 globs, space-separated my \$etc_dir = '/etc'; my @etc_dir_files = <\$etc_dir/* \$etc_dir/.*>; my @files = <LARRY/*>; # a glob</pre>
	See: readline	Filehandle examples:	<pre>my @his_lines = <LARRY>; # a filehandle read my \$name = 'LARRY'; my @his_lines = <\$name>; # indirect filehandle read of LARRY handle my @same_lines = readline LARRY; # another way to write above my @same_lines = readline \$name;</pre>
<ul style="list-style-type: none"> • with a directory handle LPo 	<ul style="list-style-type: none"> • opendir : open a directory: get a directory handle • readdir : read the directory handle. But see this. • closedir : close the directory handle. • DirHandle (Perl <= 5.5) • File::Spec::Functions (Perl >= v5.5.4) • Path::Class 	Example: iterate explicitly over a list of file names extracted from the directory using these 3 functions.	<pre>my \$dir = '/usr/bin'; opendir my \$dh, \$dir or die "Failed opening \$dir: \$!"; foreach \$file (readdir \$dh) { print "File \$file is inside \$dir\n"; # ⚠ no path in name! } closedir \$dh;</pre>
Creating directory	• mkdir	Example:	<pre>mkdir \$dir_name, oct(\$permissions); # octal for permissions mkdir \$dir_name, 0700; # do not use "0700", it's 700 decimal!</pre>
Removing directory	<ul style="list-style-type: none"> • rmdir Removes an empty directory. • File::Path remove_tree, rmtree remove dir & files (Perl >= v5.0.1) 		
Removing files	• unlink a list or <code>\$_</code>		<pre>unlink 'file1.txt', 'file2.txt'; unlink qw(file1.txt file2.txt); unlink glob 'file?.txt'</pre>
Renaming files	<ul style="list-style-type: none"> • rename an old file name to a new one. <ul style="list-style-type: none"> • The fat comma operator is sometimes used to highlight what is the old and the new name. 	As in here:	<pre>rename 'old_name' , 'new_name'; rename old_name => 'new_name'; # use fat comma to quote word left of it.</pre>
Changing permissions	• chmod changes file permissions		
Changing ownership	• chown changes file ownership		
Creating Hard link	• link to create a hard link		
Creating symbolic link	• symlink to create a symbolic link		
chdir Change current working directory	<ul style="list-style-type: none"> • File::chdir • File::HomeDir 	<ul style="list-style-type: none"> • Change the current working directory. • chdir without argument attempt to change to user home directory using the <code>\$ENV{HOME}</code> and <code>\$ENV{LOGDIR}</code> environment values if ⚠ they are set. The File::HomeDir module helps in setting them. • The built-in chdir is global ⚠ for the entire program. Use File::chdir facilities for localized operations. 	
Modules	Functions		Extra Information
	Legend: Exported by default , exported on request, <i>Win32 specific</i>		
Cwd	<ul style="list-style-type: none"> • getcwd, cwd, fastcwd, fastgetcwd, getcwd • abs_path, realpath, fast_abs_path 		<pre>use Cwd; my \$curdir = getcwd; print "cwd is \$curdir\n";</pre>
File::Basename	• fileparse , basename , dirname ,		
File::Spec File::Spec::Functions	<ul style="list-style-type: none"> • functional interface to methods: canonpath, catdir, catfile, curdir, rootdir, updir, no_upwards, file_name_is_absolute, path. <code>devnul</code>, <code>tmpdir</code>, <code>case_tolerant</code>, <code>splitpath</code>, <code>splitdir</code>, <code>catpath</code>, <code>abs2rel</code>, <code>rel2abs</code>. All can be imported by using the <code>:ALL</code> tag. 		
File::Find : Traverse a directory tree. See: File::Find::Closures	find , finddepth , %options . In wanted : File::Find::dir , File::Find::name Note that <code>\$_</code> gets the base name of the file (no path). It is used to perform filetest operations in the example here (as explicit argument to <code>-s</code> , and implicit argument to <code>-d</code> and <code>-f</code>). This traverses the entire tree.		<pre>use File::Find; find(sub {printf("- %10s : %4d, %s\n", \$_, -s \$_, File::Find::name) if (-d or -f) and (\$_ ne "."); }, '.'); # in the above it lists the names of files inside all directories not showing the directory name</pre>

Topic: List Operations 🚧

List Operators			
Sorting lists	sort	Sort a list	<pre>my @sorted = sort @unsorted_list;</pre> in place: <code>my @data = sort @data;</code>
	reverse	Sort a list in reverse order	<pre>my @rsorted = reverse @unsorted_list;</pre> in place: <code>my @data = reverse @data;</code>
Filtering list with grep	<pre>my @adult_ages = grep \$_ > 18, @ages;</pre>		<pre>my @lucky_ages = grep /7\$/, @ages; # all that end with 7</pre> <pre>my @read_ages = grep { \$_ >= 7 && \$_ <= 77 } @ages;</pre>
Counting matches	<pre>my \$count = grep \$_ > 18, @ages;</pre> <p>An expression, subroutine or block with trailing boolean can be used as the <code>grep</code> criteria. Each item in the list is identified inside <code>grep</code> by <code>\$_</code>.</p> <ul style="list-style-type: none"> • The block is an anonymous subroutine. 🙋 Return a boolean from the subroutine, but fall-off, do not return, from a block! 		
Transform a list with map	<ul style="list-style-type: none"> • map block LIST • map EXPR, LIST 	Evaluates the BLOCK or EPR for each element of LIST , setting <code>\$_</code> to each element, composing a list with the results.	Each element can generate a single value, a list or 0 or more elements. The result list flattened anyway.

Topic: Process control 🚧

Process Control	In Books: LPo	Important security information: perldoc perlsec
------------------------	-------------------------------	---

Environment Variables	Inside the <code>%ENV</code> hash.	Perl <code>%Config</code> hash: Perl configuration information. For example, whether it support threads, what are path separators, etc... • To use it: <code>use Config;</code>	
Built-in Functions	Example	Description/ Notes	
system (2 functions) <ul style="list-style-type: none"> using the shell security risk? <ul style="list-style-type: none"> avoiding the shell other syntax system return value: <ul style="list-style-type: none"> A value of 0 usually means all was OK. 	system 'ls -l \$HOME'; system "cd \$project; make &"; system 'tar', 'cvf', \$tarfile, @directories; system ('tar', @arguments); system ({ \$prog }, \$arg0, @args);	Run child process asynchronously using parent's stdin, stdout and stderr, using the OS native command shell. Use the Unix shell to execute a long running build asynchronously. 🙌 However: avoid using the shell like this . <ul style="list-style-type: none"> Using the shell to build commands from unvalidated user input data may lead to security issues. No shell invoked when more than 1 argument is passed to system. No shell interpretation, piping, re-direction done. 0 means success: <code>unless (system 'tar', arguments) { print "tar command success\n"; }</code>	
	👉 Note that if the string contain no shell metacharacters it is executed directly (not through a shell).		
	2 bytes:	MSByte: child program exit code. LSByte: system-specific information bits: <ul style="list-style-type: none"> 0x80 : set on core dump. 0x7f : signal number 	<pre>my \$retval = system(...);</pre> <pre>my \$childp_exitcode = \$retval >> 8;</pre> <pre>my \$had_core_dump = (\$retval & 0x80) == 0x80? 1 : 0;</pre> <pre>my \$signal_number = \$retval & 0x7f;</pre> ← shift most significant byte ← use least significant byte
	exec	Unlike system, exec does not return to the parent Perl process. Use:	<code>exec 'the_program' or die "Could not run: \$!"; #or warn or exit</code>
	backquotes ``	Use backquotes to capture the stdout of a program. That's the main point of using it. <ul style="list-style-type: none"> The trailing newline is not filtered out; it can be filter by <code>chomp</code>. <ul style="list-style-type: none"> The value inside the backquotes is treated like the single double quote string argument of system: it will invoke the shell if there are any shell meta-characters and supports interpolation. <ul style="list-style-type: none"> The following example builds a dictionary (hash) of topics with the text extracted from <code>perldoc</code>. Note that <code>`...`</code> is also written as <code>qx/ ... /</code> backquote operation in scalar context returns 1 string. In list context it returns a list of strings (1 per line). 	<pre>chomp(my \$current_date = `date`);</pre> <pre>my @topics = qw(die warn exit);</pre> <pre>my %info;</pre> <pre>foreach (@topics) {</pre> <pre> \$info{\$_} = `perldoc -t -f \$_`;</pre> <pre>}</pre>
Modules			
Capture streams	• Capture::Tiny	Can be used to capture the stdout and stderr streams for various ways if executing other programs	
Inter-process support	• IPC::System::Simple	Can also be used to capture streams and provide more inter-process support. • It provides systemx which never uses the shell, along with other useful functions.	
Processes as filehandles	In Books: LPo		
Perl ← program	Launching a process that pipes into the Perl process	<pre>open DATE, 'date ' or die "Cannot pipe from date: \$!";</pre> <pre>open my \$date_fh, ' -', 'date' or die "Cannot pipe from date: \$!";</pre> <pre>open my \$ps_fh, ' -', 'ps', 'aux' or die "Cannot pipe from ps: \$!";</pre> <pre>open my \$find_fh, ' -', 'find', qw(. -name "*.p[lm]" -print) or die "Cannot pipe from find: \$!";</pre> Use a bare word to define the DATE file handle. This one and the others define a local file handle variable. The file handle variable can later be used to read, as the above one, but is not global.	
Perl → program	Launching a process that the Perl process pipes into.	<code>open my \$dispatcher_fh, ' -', 'dispatcher', qw ('-to-perl-groups' 'Help!') or die "Cannot pipe to the dispatcher: \$!";</code>	
Forking	In Books: LPo . See also: Linux fork(2) system call, QA: Why do we need fort to create new processes? Why fork woks the way it does?		
fork with exec and waitpid See also: <ul style="list-style-type: none"> Other IPC functions Perl IPC 	<ul style="list-style-type: none"> fork the process into parent and child. in the child process start the program with <code>exec</code> In the parent process wait for the program termination with <code>waitpid</code> 	<pre>defined(my \$process_id = fork) or die "Fork failed: \$!";</pre> <pre>unless (\$process_id) {</pre> <pre> # Inside the child process (created by fork)</pre> <pre> exec 'long_running_process' or die "Failed starting long_running_process: \$!";</pre> <pre>}</pre> <pre># Inside the parent process, wait for completion of long_running_process.</pre> <pre>waitpid(\$process_id, 0);</pre>	
Signals	In Books: LPo		
kill	Sends a signal to a list of processes. <ul style="list-style-type: none"> The signal may be identified by number or name (string), which is more portable. The <code>%Config{sign_name}</code> provides the supported signal names. <ul style="list-style-type: none"> Note that the <i>fat comma</i> operator (<code>=></code>) can be used to automatically quote signal name: <ul style="list-style-type: none"> If the signal is 0 or "ZERO" no signal is sent to the process; instead Perl checks if it's possible to send a signal to the process: ie: if the process exists. <ul style="list-style-type: none"> If the signal is a negative number or a string that starts with '-' the signal is sent to the process group identified by the process scalar argument. 	<pre>kill 'INT', \$pid or die "Can't signal \$pid with SIGINT: \$!";</pre> <pre>kill INT => \$pid or die "Can't signal \$pid with SIGINT: \$!";</pre> <pre>unless (kill 0, \$process_id) {</pre> <pre> warn "Process \$process_id is no longer running!";</pre> <pre>}</pre> <ul style="list-style-type: none"> <code>kill '-KILL', \$process_group</code> <code>kill -9, \$process_group</code> 	
Signal handlers	<ul style="list-style-type: none"> Set the signal handler by setting <code>%SIG</code> for the signal name (with no 'SIG' prefix) to a string holding the name of the subroutine. 	<code>\$_SIG{'INT'} = 'dispatcher_int_handler';</code>	
Error Logging and Reporting	<ul style="list-style-type: none"> Perl supports the <code>warn</code> built-in to generate warnings on stderr. The <code>Carp::carp</code> from the <code>Carp</code> package, provides more information. 	• Log::log4perl is an implementation of the popular Apache Log4j for Perl.	

PerlTidy formatting control 🚧

perltidy option	Option	Impact
indentation style	<ul style="list-style-type: none"> <code>-bl</code>, <code>--opening-brace-on-new-line</code> <code>--brace-left</code> 	<ul style="list-style-type: none"> Without this option (the default) the code indentation style selected is K&R style. With this option, the indentation style is Allman/BSD style.