Perl 5 🚧

See also: 𝔓𝔩 - Perl • **Perl @ Wikipedia** • **perl.org** • **PerlMonks.org** σ : **O'Reilly Books**	• Perl Intro - a quick introduction to Perl. PerlCheat • Online Perl books : Beginning Perl , Modern Perl (html) , `Perl tutorial.org` , Perl Maven Tutorial • Perl Cookbook σ (PLEAC Perl: *list of Perl code solutions*) • Learning Perl σ, Intermediate Perl σ , Mastering Perl σ , Effective Perl Programming σ	perl , Perl command line options , **perlrun** , perlivp , perldoc , perlbug / perlthanks perlsec	• **Online Perl Interpreter** • **Online PerlTidy** option info.
Perl Guidelines and tools	**Perl Style Guide, 10 Essential Development Practices**, • Books: **Perl Best Practices** σ, **Modern Perl Best Practices (course)** σ • **perlcritic 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**		
perldoc browser • In Emacs: **`C-c C-h F`**	• **perldoc** : about perldoc itself • **perltoc** : table of content: names of all pages • **perlsyn** : Perl syntax • **perlfunc** : Perl built-in functions	☝Use perldoc to find if a Perl module is installed, as in: `perldoc` **`local::lib`** • `perldoc local::lib` prints the documentation of **local::lib** if it is installed. • `perl -Mlocal::lib` is useful to get modules installed in your home directory σ	
CPAN (@ Wikipedia) • **Search CPAN — meta::cpan**	• **The Zen of Comprehensive Archive Networks** • **PAUSE - Perl Authors Upload Server**	**Command line tools** interacting with CPAN to install Perl modules σ: • **cpan**: (requires config), **cpanplus**, or cpanminus : **cpanm** :(no config required). • To install a Perl module with **cpanm**: cpanm -S *The::Module*	

Perl scripts

Writing Perl scripts — Impose strictures in Perl files to prevent errors by adding one of the following use lines. Also see the **strictures package.**

Use the following at the beginning of Perl script files.
 	
perldiag @ perldoc
	`#!/usr/bin/env perl`
	`use strict;`
	`use warnings;`
	
	`# for testing only:`
	`use diagnostics;`
		`#! /usr/bin/perl -w`
		`use v5.12; # loads strict …`
		`use v5.35; # loads warnings`
		
		⚠ `use diagnostics` produces more info but increases startup time.
			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).
			⚠ It's best to: use warnings; **perl -w** generates warning for all Perl code in the program including modules used by the program. Also use the **-c** option to check syntax. But most Perl code should also activate the strict Perl rules and warnings to detect warnings. See: Barewords in Perl

		Alternative: perl -Mdiagnostics . Emacs **pel-perl-critic** command can report diagnostic.	
use version/features	`use v5.36;`	This can be used to enable both the strict and warning pramas as well as several named features. • See the **table listing the feature bundles per Perl versions**.	

Perl 5 Operators

Perl 5 Operators

Note: Perl has a large number of operators, listed below with their **precedence and associativity**.
- C Operators missing from Perl : unary &, unary * and (type)
- Quote and Quote-like operators : in Perl quotes are operators and they provide various kind of interpolating and pattern matching capabilities.

Associativity: one of:
- right
- left
- NA : not associative: cannot use more than one of these operators in sequence.
- CH: chained

To get this information, use:
`perldoc perlop`

left	**terms and list operators (leftward)**	`( )`
left	**Arrow Operator:**	`->`
NA	**Auto-increment and Auto-decrement:**	`++ --`
right	**Exponentiation:**	`**`
right	**Symbolic Unary Operators**:	`! ~ ~. \` and unary + and – **Note:** The operator `\` creates a reference. See example.
left	**Binding operators:**	`=~ !~`
left	**Multiplicative Operators:**	`* / % x`
left	**Additive Operators:**	`+ - .`
left	**Shift Operators:**	`<< >>`
NA	**named unary operators**	
NA	**Class instance Operator:**	`isa`
CH	**Relational Operators:**	as numbers: `< > <= >=` as strings: `lt gt le ge`
CH/NA	**Equality Operators:**	as numbers: `== != <=>` as strings: `eq ne cmp ~~`
left.	**Bitwise And:**	`& &.`
left	**Bitwise Or and Exclusive Or:**	`| |. ^ ^.`
left	**C-style Logical And:**	`&&`
left	**Logical Defined-Or:**	`|| ^^ //`
NA	**Range Operators:**	`.. …`
right	**Conditional Operator:**	`?:`

right	**Assignment Operators**:	`=`
		`**= += *= &= &.= <<= &&=`
		`-= /= |= |.= >>= ||=`
		`.= %= ^= ^.= //=`
		`x=`
		`goto last next redo dump`

left	**Comma, fat-comma Operators:**	`, =>`
NA	**list operators (rightward)**	
right	**Logical Not:**	`not`
left	**Logical And:**	`and`
left	**Logical or and Exclusive or:**	`or xor`

trick operators ⚠
Do not use in production code!
But understanding how these work does help understand Perl.
These are not real Perl operators; they are concatenation of other operators that achieve a specific effect.

`-+-`
`0+`
	Converts a string that starts with digits into a number.	`print -+- '22les poulets!';`
		# prints 22
			-+- is essentially - + - or - - but a + to allow placing them together. The `0+` does the same as `-+-`, but the second has higher precedence.

`=()=`	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.	`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	

`@{[]}`	Interpolate an array in a string: `"@{[something]}"` is the same as: `join $", something`	`print "these people @{[get_names()]} get promoted"`	

`~~`	Force scalar context.	In scalar context **localtime** returns human readable time, but in list context it returns a 9-tuple with date elements.	`$ perl -le 'print ~~localtime'`
			`Mon Nov 30 09:06:13 2009`

Truth and falsehood

⚠ Remember that the strings '0' and " mean false. The output of glob() may return a file named '0' !

⚠ a bareword false has a truth value of true!!!!

• False in a **boolean context**: • the number **0**, • the strings **'0'** and **' '**, • the empty list **()**, • **"undef"** • All other values are true.	• Negation of a true value by "!" or "not" returns a special false value. • When evaluated as a string it is treated as '', but as a number, it is treated as 0.	So the following scalar values are considered **false**: • undef - the undefined value • 0 the number 0, even if you write it as 000 or 0.0 • '' the empty string. • '0', a **single** 0 in the string.	All other scalar values, including the following are **true**: • 1 any non-0 number • ' ' the string with a space in it • '00' two or more 0 characters in a string • "0\n" a 0 followed by a newline • 'true' • 'false' . Even the string 'false' evaluates to true.

☝ One way to define valid true and false *constant symbols* that can be used in assignments (but see ⬅): `use constant { true => 1, false => 0 };`

File test operators
See **filetest -X**

File tests can be **stacked** `(-r -w -e $fname)` or combined as in the following example σ:
☝ Notice the underscore in the example: it's the **virtual filehandle** _ accessing the last **stat** or **lstat** result :

```
if (-e $fname && -f _ && -r _ ) {
  print("$fname exists, is readable\n"); }
```

The operators check if the file…

See also:
- File Tests σ
- `File test operators @ perl tutorial`

See also:
- **File::stat**
- **IO::Interactive**

-r	is readable *by effective uid/gid*	**-e**	exists.	**-b**	is a block special file.
-w	is writable *by effective uid/gid*	**-z**	is empty.	**-c**	is a character special file.
-x	is executable *by effective uid/gid*	**-s**	has nonzero size (returns size in bytes).	**-t**	handle is opened to a tty.
-o	is owned *by effective uid*	**-f**	is a plain file.	**-u**	has setuid bit set.
-R	is readable *by* ***real*** *uid/gid*	**-d**	is a directory.	**-g**	has setgid bit set.
-W	is writable *by* ***real*** *uid/gid*	**-l**	is a symbolic link.	**-k**	has sticky bit set.
-X	is executable *by* ***real*** *uid/gid*	**-p**	is a named pipe (FIFO) or Filehandle is a pipe.	**-T**	is an ASCII text file (heuristic guess).
-O	file is owned *by* ***real*** *uid*.	**-S**	is a socket.	**-B**	is a "binary" file (opposite of -T).
-M	Days between start time and file modification time	**-A**	Days between start time and file access time	**-C**	Days between start time and node change time (in Unix).

Perl 5 Constants and Variables

Perl Constants

- Perl pragma to declare constants. ⚠️ But be aware that these are still not read-only, that they inject sub-routines and have several limitations. Read the doc!!
- CPAN modules for defining constants by Neil Bowers . Of particular interest: Const::Fast and Attribute::Constant for efficient read-only constants.

Perl Variables Names

Perl Variables Names	Scalar Naming Conventions		Array Naming Conventions	All: underscore or letter of the first character.
Case is significant in all names. ASCII by default, **UTF-8** if the **utf8 pragma** is used.	• Local variables: • Global variables: • Constants: • All variables:	$lowercase $Title_Case $UPPER_CASE words separated by underscores.	Similar conventions, except that array names should be **plural**. • @locals • @Global_Arrays • @CONSTANT_ARRAYS	• Module names are MixedCaseNoUnderscores • Constants are UPPERCASE_WITH_UNDERSCORES • Package wide vars are Mixed_Case_With_Underscores • Functions/methods are lowercase_with_underscores • Avoid ALLUPPERCASE: used by Perl special variables.

Perl types

Perl types	Sigil	Examples	Meaning	Extra Info
Scalar	`$`	`$foo`	Simple scalar value	
		`$days[28]`	29th element of array @days	
		`$days{'Feb'}`	Value associated with the *Feb* key of hash %days	
		`${days}`	Same as `$days`, but unambiguous before alphanumerics. Useful inside strings for interpolation of variables followed by other letters.	
		`$Dog::days`	The `$days` variable inside the Dog package.	
		`$Dog'days`	Same as above. However this is an archaic use of the single quote.	
		`$#days`	Last index of array `@days.`	
		`$days->[28]`	29th element of array pointed to by reference `$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*	`@`	`@days`	Array containing `($days[0], $days[1], … #days[$#days])` .	• A *list* is an ordered collection of scalars (of any type). • An *array* is a variable that contains a list. • Reading beyond the end of array returns **undef**
		`@days[3,4,5]`	Array slice containing `($days[3], $days[4], $days[5])` .	
		`@days[3..5]`	Array slice containing `($days[3], $days[4], $days[5])` .	
	• *Negative* indices used in read access from the end: -1 is last item. • Use these negative indices to access from the end. Do **not** compute index with $#name -3, if the list size is 2, this will give invalid results.			
• **slices**	• Use a slice to select multiple elements from a list, array, or hash. • Don't use a slice when you know you need exactly one element.			• An lvalue slice imposes list context on the righthand side.
• **Anonymous arrays**	• What are the advantages of anonymous array? @ StackOverflow • Perlref @ Perldoc, Perl reference tutorial @ Perldoc			• Anonymous array := a type of array reference. • Array reference allows Perl to treat the array as a single item. • This can be used to build, nested data structures.
Hash/associative array	`%`	`%days`	Associative array (hash): keys-value pairs. Can be initialized as: • `%days = (Jan => 31, Feb => $leap? 29 : 28, …)` • `%days = ("Jan", 31, 'Feb', $leap? 29 : 28, … )`	Initialize a hash slice with array context: @char_to_num{'A' .. 'Z'} = 1 .. 26;
		`@days{'J','F'}`	Hash slice containing `($days{'J'}, $days{'F'})` .	
Subroutine	`&`	`&foo`	**&** is needed to create reference to subroutine.	
Typeglob	`*`	`*foo`		See: Advanced Perl Programming, 1st Edition Section 3.2

7 kinds of package variables or variable-like elements in Perl:

1. **scalar variables**
2. **array variables**
3. **hash variables**
4. **subroutine name**
5. **format names**
   - how to format output in Perl?, Perl-Formats
   - See **write** and **select**
6. **file handles**
7. **directory handles**

Scalar values

Numeric literals examples.
Note: leading 0 work only for literals, not for string-to-number conversions.

- **numeric**:
  - integer : using the system's native format.
    - **bigint** - transparent big integer support.
    - **bignum** - transparent big number support.
  - floating-point : using the system's native format.
    - **bigrat** - transparent big rational number support.

```
my $x = 12345;           # integer
my $x = 12345.67;        # floating point
my $x = 6.02e23;         # scientific notation
my $x = 0x1f.0p3;        # power² exponent: Perl >= v5.22
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        Perl >= v5.34
my $x = 0xffff;          # hexadecimal
my $x = 0b1100_0010;     # binary
```

Useful related builtin functions

- **oct** - supports binary, octal, hex
- **hex**
- **POSIX::ceil**
- **POSIX::floor**
- **abs**

- **string**
  - 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.
  - But `\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).

- **Unicode support**

To use Unicode literally in a program, add the **utf8 pragma**:

```
      use utf8;
```

See: Perl Unicode Tutorial, Perl Unicode Introduction, Perl Unicode Support @ perldoc

- **Quote constructs**

See:
- Strings in Perl: quoted, interpolated and escaped

Customary	Generic	Meaning	Interpolates?
`''`	`q//`	Literal string	No
`""`	`qq//`	Literal string	Yes
` `` `	`qx//`	Command execution	Yes
`()`	`qw//`	World list	No
`//`	`m//`	Pattern match	Yes
`s///`	`s///`	Pattern substitution	Yes
`tr///`	`y///`	Character translation	No
`""`	`qr//`	Regular expression	Yes

Notes

- 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:

```
my $chuck_of_code = q {
    if ($condition) {
        print "Salut!";
    }
};
```

- It's also possible to write: `s<foo>(bar)` and `tr(a-f)[A-F]` as well as separating them on 2 lines:

```
tr (a-f)
   [A-F];
```

- Array variables are interpolated by joining all elements with the separator specified by the $" special variable ($LIST_SEPARATOR) .

- **Character escapes** (only inside double quoted strings)

\a	Alert (bell)	**\e**	ESC character
\b	Backspace	**\033**	ESC in octal
\e	ESC character	**\o{33}**	ESC in octal
\f	Form feed	**\x7f**	DEL in hexadecimal
\n	Newline (usually LF)	**\x{263a}**	Character number 0x263A
\r	Carriage return (Usually CR)	**\cC**	Control-C
\t	Horizontal tab		

Any Unicode code point, by name:

\N{LATIN SMALL LETTER E WITH ACUTE}	é
\N{ U+E9 }	é

- **translation escapes** (inside **double quoted** strings)

\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**		

- **bareword**

In Perl, a *bareword* 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 **`use strict;`** or **`use strict "subs";`** or **`use v5.12;`** is specified.

- **Here documents**
- 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:

- Default : **<<EOF;** Supports variable interpolation.
- Double quotes: **<<"EOF";** Supports variable interpolation. Can also be written with whitespace as in **<< "EOF";**
- Single quotes: **<<'EOF';** Does not support interpolation. Can also be written with whitespace as in **<< 'EOF';**
- backticks: **<<\`EOF\`;** Execute commands in a shell and return text printed on stdout. Can also be written with whitespace as in **<< \`EOF\`;**
- indented: **<<~EOF;** Allows indenting the here-doc string. Can also use the ~ with the other forms: **<<~\EOF**, **<<~"EOF"**, **<<~"EOF"**, **<<~\`EOF\`**
- They can also be stacked and text can be transformed. See the documentation.

- **Perl Regexp** info, cheatsheets & regexp testers

- **Regexp Tutorial**
- **Learn PCRE in X minutes**
- **PCRE cheatsheet**
- Debuggex regexp tester
- regex101
- RegEx Pal

Perl Special Variables • **Perl Variables**	☝ To get information about a Perl special variable from the command line use the **perldoc -v** command. • To get information about **$<** use: **`perldoc -v '$<'`**				
• **Deprecated and removed variables:**	$# $* $[${^ENCODING} ${^WIN32_SLOPPY_STAT}				
• **General variables**					
default input and pattern searching space	• $ARG • $_		subroutine parameters	• @ARG • @_	
list separator	• $LIST_SEPARATOR • $”		Subscript separator for multidimensional array emulation	• $SUBSCRIPT_SEPARATOR • $SUBSEP • $;	
Name of executed program	• $PROGRAM_NAME • $0		Name used to execute the current copy of Perl	• $EXECUTABLE_NAME • $^X	
Perl process ID	• $PROCESS_ID • $PID • $$	Process real GID	• $REAL_GROUP_ID • $GID • $(	Process effective GID	• $EFFECTIVE_GROUP_ID • $EGID • $)
Process real UID	• $REAL_USER_ID • $UIG • $<		Process effective UID	• $EFFECTIVE_USER_ID$ • $EUID • $>	
Special variables in sort	• $a • $b	The Perl **sort** function uses global variables $a and $b. **sort** sorts strings. Pass a sorting function that uses the **<=>** equality operator to force numerical comparisons: `@sorted = sort { $a <=> $b } @unsorted;`			
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	• $OLD_PERL_VERSION • $]		Perl interpreter revision, version and subversion	• $PERL_VERSION • $^V	
Maximum file descriptor	• $SYSTEM_FD_MAX • $^F		Fields of each line when auto-split mode is on.	@F	
Include Directories	@INC	Included filenames	%INC	Hook localization (?)	$INC
inplace-edit extension value	• $INPLACE_EDIT • $^I	Package’s class parent classes	@ISA	Emergency memory pool	$^M
Maximum block nesting	${^MAX_NESTED_EVAL_BEGIN_BLOCKS}			Time when program began running	• $BASETIME • $^T
Name of OS where this Perl was built	• $OSNAME • $^O	Signal handlers	%SIG	Coderefs for various perl keywords	%{^HOOK}
• **Regexp Variables**					
captured sub-patterns	$<digit>($1, $2, …)		Capture buffer content	@{^CAPTURE}	
String matched	• $MATCH • $&		String matched (compiled regexp)	${^MATCH}	
String preceding match	• $PREMATCH • $`		String preceding match (compiled regexp)	${^PREMATCH}	
String following match	• $POSTMATCH • $’		String following match (compiled regexp)	{^POSTMATCH}	
Last capture group	• $LAST_PAREN_MATCH • $+		Most recently closed capture group	• $LAST_SUBMATCH_RESULT • $^N	
Match capture key values	• %{^CAPTURE} • %LAST_PAREN_MATCH • %+		Maximum regexp nested group	${^RE_COMPILE_RECURSION_LIMIT}	
Match start offsets	• @LAST_MATCH_START • @-	Match ends offsets	• @LAST_MATCH_END • @+	Named captured groups	• %{^CAPTURE_ALL} • %-
Last successful pattern	${^LAST_SUCESSFUL_PATTERN}		Result of last successful regexp assertion	• $LAST_REGEXP_CODE_RESULT • $^R	
regexp debug flag	${^RE_DEBUG_FLAG}		regexp internal optimization/memory	${^RE_TRIE_MAXBUF}	
• **Format Variables**					
Current value of the write() accumulator for format() lines.	• $ACCUMULATOR • $^A				
Form feed format. defaults to \f	• IO::Handle->format_formfeed(EXPR) • $FORMAT_FORMFEED • $^L		Set of characters after which a string may be broken to fill continuation fields	• IO::Handle->format_line_break_characters EXPR • $FORMAT_LINE_BREAK_CHARACTERS • $:	
Number of lines left on the page on currently selected output channel	• HANDLE->format_lines_left(EXPR) • $FORMAT_LINES_LEFT • $-		Current page length of current output channel	• HANDLE->format_lines_per_page(EXPR) • $FORMAT_LINES_PER_PAGE • $=	
Name of current top-page format of output channel	• HANDLE->format_top_name(EXPR) • $FORMAT_TOP_NAME • $^		Report format name of output channel	• HANDLE->format_name(EXPR) • $FORMAT_NAME • $~	
• **Error Variables**	The variables **`$@`**, **`$!`**, **`$^E`**, and **`$?`** 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 eval operator	• $EVAL_ERROR • $@		Current state of interpreter	• $EXCEPTIONS_BEING_CAUGHT • $^S	
Current value of C errno integer variable	• $OS_ERROR • $ERRNO • $!	**`$!`** returns the system variable **errno** when used in a numeric context, but returns the string from **perror()** when used in string context.	Hash of error names to 0 or 1, set to 1 if current error is this error.	• %OS_ERROR • %ERRNO • %!	
OS detected error	• $EXTENDED_OS_ERROR • $^E				
Status returned by last pipe close, backtick command, wait, waited, or system() call.	• $CHILD_ERROR • $?		native status returned by last pipe close , backtick command, wait() or waitpid() or system() call	${^CHILD_ERROR_NATIVE}	

Current value of warning switch	• $WARNING • $^W		Current set of warning checks enabled by the use warnings pragma	${^WARNING_BITS}	
• **Variables related to the interpreter state**	These variables provide information about the current interpreter state.				
Flag associated with the -c switch	• $COMPILING • $^C		The current value of the debugging flags	• $DEBUGGING • $^D	
Current phase of the perl interpreter	${^GLOBAL_PHASE}		Debugging support. Internal variable.	• $PERLDB • $^P	
Compile-time hints for the perl interpreter. Internal use only	$^H		Values of compiled statements	%^H	
Taint mode	${^TAINT}		Safe locale operations availability	${^SAFE_LOCALES}	
Input/Output Layers. Internal use by PerlIO only.	${^OPEN}		Unicode Settings of Perl	${^UNICODE}	
Internal UTF-8 offset caching code state	${^UTF8CACHE}		State of UTF-8 locale detected by perl at startup.	${^UTF8LOCALE}	
• **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 <>	$ARGV	Command line arguments of the script ⬅ See **diamond operator <>**. ➡	@ARGV	Number of arguments minus one	$#ARGV
Special file handle that iterates over command-line filenames in @ARGV	ARGV	Special file handle that points to currently open output file when doing edit-in-place processing	ARGVOUT		
Output field separator for the print operator	• IO::Handle->output_field_separator(EXPR) • $OUTPUT_FIELD_SEPARATOR • $OFS • $,		Current line number for the last file handled accessed	• HANDLE->input_line_number(EXPR) • $INPUT_LINE_NUMBER • $NR • $.	
Input record separator (newline by default)	• IO::Handle->input_record_separator(EXPR) • $INPUT_RECORD_SEPARATOR • $RS • $/		Output record separator	• IO::Handle->output_record_separator(EXPR) • $OUTPUT_RECORD_SEPARATOR • $ORS • $\	
Auto-flush control • order of output @ Perl Maven • Suffering from Buffering?	• HANDLE->autoflush(EXPR) • $OUTPUT_AUTOFLUSH • $|	Perl activates file buffering by default. Assign 1 to **$|** to activate auto-flush.	Last read file handle	${^LAST_FH}	

Perl 5 Input/Output 🚧

References	• open @ perldoc browser • Writing to files with Perl @ Perl Maven • open file in-memory @ stackOverflow	• Stupid open() tricks @Perl.com: • No explicit filename • create an anonymous temporary file	• print to a string • read lines from a string		
print, **printf**, **sprintf**	**`print`, `printf`, `sprintf`** (which describes the format) . Note: **`print`** 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!)				
diamond operator <> **The double diamond, a more secure <>** *(Perl >= v5.22)*	Both **<>** and **<<>>** operators read the content of files listed on the command line via @ARGV. Nothing or **-** on the command line identifies stdin. The **<>** operator supports shell redirection and pipe operations which **<<>>** does not allow (for security reasons).				
	`print <>;`	⬅ Simple implementation of /bin/cat	`print <<>>;`	⬅ safer one	Redirection cannot be forced via file names embedding them with. the **<<>>** operator.
	`print sort <>;`	⬅ Simple implementation of /bin/sort	`print sort <<>>;`	⬅ safer one	

☝ In-place-editing
The <> operator tries to duplicate the original file's permission and ownership.
	Set **`$^I`** to a backup file extension (such as Emacs "~" or ".bak") to change the behaviour of the **<>** and **<<>>** operators and print.
	In a `while (<>) {…}` loop, when **`$^I`** is not undef (its default), Perl:
	• 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!
			`use strict;`
			`$^I = "~";` # rename old file: add '~' to it's name (Emacs-style backup)
			
			`while (<>) {`
			`  s/something/Something else/;` # perform any substitution
			`  print;`
			`}`		

perl -i cmdline option	It's also possible to do this on the command line! For example:		`perl -p -i~ -w -e 's/something/Something else/g' data*.dat`		
Special filehandle names Also See: • **File handle Variables** section above.	**ARGV**	The special filehandle that iterates over command-line filenames in @ARGV. Usually written as the null filehandle in the angle operator **<>** (or **<<>>**)			
	ARGVOUT	The special filehandle that points to the currently open output file when doing edit-in-place processing with **-i**. • Useful when you have to do a lot of inserting and don't want to keep modifying **$_**			
	STDIN	**<STDIN>** : line input operator for the STDIN filehandle (for the **standard input**). • Each time <STDIN> is used in scalar context, Perl reads 1 complete line of the standard input and uses it as the value of <STDIN>. • The string includes a line termination character. Use the **chomp()** built-in function to strip it off the variable. • If <STDIN> is read in list context, it returns **all lines** inside a list! For example, foreach (<STDIN>) { … } reads the entire stdin in 1 step: $_ holds it all!			

		`while (<STDIN>) { # print all`
		`   print;         # lines of`
		`}                # stdin`
			`while (defined($_ = <STDIN>)) {`
			`   print $_;`
			`}`
				The code in the left-most cell is the shortest form. It is equivalent to the code beside it; each line of stdin is stored in the default variable **$_** and the loop stops on end at which time <STDIN> returns undef.	

	STDOUT	**standard output**			
	STDERR	**standard error** Note: generally STDERR is not buffered, while STDOUT is buffered by default. Text sent on STDERR may show up before STDOUT. • Print a new line on STDOUT to help flushing it or assign 1 to **$|** to activate auto-flush.			
	DATA				
say	• say `use feature qw(say);` or `use v5.10;` (or higher). Like print, but implicitly appends a newline at the end of the list.				

Perl 5 Statements 🚧

Loop control ☝ Use the **last** and **redo** inside a naked block of code to control looping.	See **perlsyn** for more information on Perl syntax which includes declarations, blocks, loops, labels, subroutines, etc…		
	loop control keywords: • **`last`**: exits the loop. • **`next`**: starts the next iteration of the loop. • **`redo`**: restarts the loop block without evaluating the condition again.	The **`last`,`next`**, and **`redo`** loop control keywords work in the following constructs: • while (condition) { … } • until (condition) { … } • for (init; condition; continue) { … } • foreach array { … } • naked block: { … }	Notes: • The while and foreach loops may have a **continue block**: executed before evaluating condition again, which corresponds to the 3rd part of a for loop statement. See this @ stackOverflow. • Blocks can be labelled as targets to **`last`,`next`**, and **`redo`**

Statement modifiers	• `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 "__END__" on it will **not** work since it reads all of STDIN:
		`foreach (<STDIN>) {`
		`  last if ?__END__/;`
		`  …;`
		`  }`
			The while statement **imposes a scalar context;** it takes one line at a time from <STDIN> and the following code works properly:
			`while (<STDIN>) {`
			`  last if /__END__/;`
			`  …;`
			`}`

	• do block		
Conditional statements			

Perl 5 Subroutines 🚧

Perl subroutines				
subroutine &	• Why we teach the subroutine ampersand • Why should I use the & to call a Perl subroutine? @ StackOverflow		Another point of view: Subroutines and Ampersands	
Subroutine Prototypes	An older Perl feature. Clashes with subroutine signatures as of Perl v5.20. In *Perl >= v5.20* put the **`:prototype`** attribute before subroutine prototype parenthesis.			
Subroutine signatures • *Perl >=5.36:* Stable • *Perl >= 5.20*: Experimental See: **Use v5.20 subroutine signatures**	Exactly zero arguments	**`()`**	Zero or 1 argument, no default, unnamed:	**`($=)`**
	Zero or 1 argument, no default, named	**`($val=)`**	Zero or 1 argument, named, with default	**`($val=1)`**
	exactly 1 named argument:	**`($val)`**	Exactly 2 arguments	**`($v1, $v2)`**
	2, 3 or 4 arguments no defaults:	**`($v1, $v2, $=, $=)`**	2,3 or 4 arguments, 1 default:	**`($v1, $v2, $v3='a', $=)`**
	Two or more, any number of arguments.	**`($v1, $v2, @)`**	Two or more arguments, remainders into a named array:	**`($v1, $v2, @rest)`**
	Two or more arguments: an even number	**`($v1, $v2, %)`**	Two or more arguments, remainders into a named hash:	**`($v1, $v2, %rest)`**
	Class method	**`($class, …)`**	**Object method**	**`( $self, …)`**
Variables in subroutines	global by default			
	my	local, lexical scope, non persistent		
	state	Local, lexical scope, persistent *Perl >= v5.10*	Restriction: in *Perl < v5.28*: array and hashes state cannot be initialized in list context.	
	`our`	creates a lexical scoped alias to a package variable		
	`local`			
Returned value	• 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. • Inside the subroutine, use the **wantarray** function to determine the context of the subroutine call.			

Perl 5 Built-in Functions 🚧

Perl Functions **Perl syntax**	☝To get information about a Perl function from the command line use the **perldoc -f** command. • To get information about **print** use: **`perldoc -f print`**
⚠️**Cautionary notes**	
• **each** keyword is broken • Use **Var::Pairs** instead.	Do NOT use the built-in **each**. It is broken, as described by Damian Conway in his Modern Perl Best Practice O'Reilly course, section control structure. • **each** is not re-entrant: • nested loops of each over the same hash does not work as expected and will create infinite loop since the nested loop each juts iterates from where the first loop each left it. • Exiting the loop leaves the state of the each internal pointer at the current location. • If you use each on the same hash later it will resume from where it left, it will not start form the beginning.

Perl 5 Modules 🚧

Perl Modules		
Perl core modules	• How to detect where a module is installed : `perldoc -l` *`Module`*	
Modules @perltutorial **Modules** Using simple modules ♂	**do**	Looks for the module file by searching the `@INC` path. Performed at run time (and therefore can be done conditionally). • 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.
	`require`	Loads the module file once, also teaching the `@INC` path. Performed at run time (and therefore can be done conditionally). • 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**)
The normal way to access Perl modules ➨	**`use`**	Similar to `require` except that Perl applies it before the program starts: it's done at compile time. • Therefore the `use` statement cannot be invoked inside conditional statements such as if-else. Used often to include a module in a program.

PerlTidy formatting control 🚧

perltidy option	**Option**	**Impact**
indentation style	• **-bl**, • **--opening-brace-on-new-line** • **--brace-left**	• Without this option (the default) the code indentation style selected is **K&R style**. • With this option, the indentation style is **Allman/BSD style**.