Perl 5 🚧

See also: 𝔓ℓ - Perl • **Perl @ Wikipedia** • **perl.org** • **perl @ GitHub** • **PerlMonks.org** **ơ** : **O'Reilly Books** • **Perl mailing lists** • **Perl Weekly**	• *Quick Intros to Perl*: 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*) • Perl Cookbook **ơ** (PLEAC Perl: *list of Perl code solutions*) • Learning Perl **LPơ**, Intermediate Perl **IntPơ** , Mastering Perl **ơ** , Effective Perl Programming **ơ** • Object Oriented Perl, Higher-order Perl **HơP**. Some others are not recommended for various reasons.	perl , Perl command line options , **perlrun** , perlivp , perldoc , perlbug / perlthanks perlsec	• Online Perl Interpreter **perl-live-coding** out & in Emacs • Online PerlTidy option info.
	Perl Guidelines and tools: **Perl Style Guide, 10 Essential Development Practices**. Books: **Perl Best Practices ơ**, **Modern Perl Best Practices (course) ơ** • **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 **ơ**	
CPAN (@ Wikipedia) • **Search: meta::cpan** • **CPAN Testers** • **CPANdeps**	• **The Zen of Comprehensive Archive Networks** • **PAUSE - Perl Authors Upload Server** • **Installing Local Perl Modules with CPAN** • CPAN Issue tracker: **CPAN RT** See Also: **IntPơ**	**Command line tools** interacting with CPAN to install Perl modules **ơ**. (see also this StackOverflow Q/A): • **cpan**: (requires config, but has defaults). Use **local::lib**; cpan will be able to install into your ~/perl5 tree. • Type **cpan** to open the cpan shell, then type `install The::Module` to install packages. • **cpanplus**, or cpanminus : **cpanm** :(no config required). **cpanm**: cpanm -S *The::Module*	

Last updated on: 2025-02-17

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 version history • **at perldoc**	• **Perl Versions Guide** • **Perl versions @ perldoc**	• 5.*even*: maintenance track version • 5.*odd* : development track version • decimal: 1.02. # *old way* • dot-decimal: v5.38.2 • **$]** : current Perl version as a decimal number • **$^V** : current Perl version as a version object
M: minor, P: patch level	Equivalence between decimal and dot-decimal versions: AAA.*MMM*PP ⇔ **v**AAA.*MMM*.PP . Note that 3 *Minor* digits are used in the decimal versions. Patch use 2 or 3.	

Perl 5 Operators

Perl 5 Operators Note:

Perl 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

Note: ơ The **Bitwise String Operators** are :
`~.  &.  |.  ^.`
`&.=  |.=  ^.=`
- Stable: Perl >= 5.28
- Experimental: Perl >= 5.22

left	**terms and list operators (leftward)**	`( )`	**Note:** print, sort, reverse, chmod, are list operators
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 Or, Xor, 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 `- -` with a `+` to put them together. The `0+` is the same, but `-+-` 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
⚠ The strings '0' and '' mean false. The output of glob() may return a file named '0' !
⚠ The 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.

These scalar values are **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 are **true**, such as:
- 1 and 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 '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 **ơ**:
☝ 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 **ơ**
- File test operators @ perl tutorial

See also:
- **localtime**
- **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 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: 1st char: underscore or letter. Never use ALLCAPS
Case sensitive. ASCII by default, **UTF-8** if the **utf8 pragma** is used.	• All variables: words_with_underscores • Local variables: `$lowercase` • Global variables: `$Title_Case` • Constants: `$UPPER_CASE`	Same. 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

Scope of variables
• **Declarations**
Scope of variables in Perl @Perl Maven

`local` can be used to locally change the value of Perl special 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.

`my`	local, lexical scope, non persistent	Examples: `my @values = (42, 36, 99);` `my ($v1, $v2) = (42, 36);`
`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 (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. • 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**)
   • how to format output in Perl?, Perl-Formats
6. **I/O: file, directory, other handles**

Perl types `$`
Scalar

Archaic use of single quote: `$Dog'days`

`$foo`	Simple scalar value	`$#days`	Last index of array `@days`.
`$days[28]`	29th element of array @days	`$days->[28]`	29th element of array pointed to by reference $days.
`$days{'Feb'}`	Value associated with the *Feb* key of hash %days	`$days[0][2]`	Multi-dimensional array
`${days}`	Same as `$days`, use before alphanumumerics.	`$d{99}{'Feb'}`	Multi-dimensional hash
`$Dog::days`	The `$days` variable inside the Dog package.	`$d{99, 'Feb'}`	Multi-dimensional hash emulation

list and Array `@`
• 0-based indexed (first index is 0).
• Last index of array *@name* is **$#***name*

• Arrays are initialized by literal lists.
• Lists are always flattened in Perl:
  • This means that (1, 2, (10, 20, (100, 200), 30, 40), 4) is exactly the same is (1, 2, 10, 20, 100, 200, 30, 40, 4) . Use references to create nested data structures.

• You can assign a list of values to a list of variables. Useful to swap: `($val1, $val2) = ($val2, $val1);`
• If there are more variables than values: the extra variables are set to **undef**. Extra values are ignored.

`@days`	Array containing ($days[0], $days[1], … #days[$#days])
`@days[3,4,5]`	Array slices containing ($days[3], $days[4], $days[5])
`@days[3..5]`	Array slices containing ($days[3], $days[4], $days[5])

• 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**

• *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.

• **array slices** LPo
Simple explanation

• 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.
• Assign to array slice to update several values. ➨

```
my @extracted = (6, 2, 8, 4);
my @choices = @digits[@extracted]
my $mod_time = (state $filename)[9];
@extracted[1, 3] = (7, 9);
```

```
my @digits = (0..9);
my @one2five = @digits[1..5];
my @premiers  = @digit[1, 2, 3, 5, 7 ];
```

• **Anonymous arrays**

• What are the advantages of anonymous array? @ StackOverflow
• Perlref @ Perldoc, Perl reference tutorial @ Perldoc

• 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.
hash slice LPo ➨
key-value slices LPo ➨

`%`	`%days`	Associative array (hash): keys-value pairs. Can be initialized as: • `my %days = (Jan => 31, Feb => $leap? 29 : 28, …)` • `my %days = ("Jan", 31, 'Feb', $leap? 29 : 28, … )` Multiple values of a hash can be changed with the following construct:
	`@days{'J','F'}`	Hash slice returning a list containing `($days{'J'}, $days{'F'})`.
	extract/write values:	`my scores = @rating{ @names };  @rating { @names } = ( 45, 55);`

Initialize a hash slice with array context:
`@char_to_num{'A' .. 'Z'} = 1 .. 26;`
`my %rating = (ron => 20, al => 50, steve => 80);`
`# use fat comma to quote word left of it.`
`my @names = ('ron', 'al');`
`@rating{ @names } = (25, 35); # update ron & al's ratings`

Subroutine

`&` `&foo` — **&** is needed to create reference to subroutine with **\&**subroutine_name

I/O

Format

Typeglob

A typeglob 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.

`*` `*symbol` — 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
• brace around refs:
circumfix dereferencing:
• simplify with `->`
• simplify more
☝ Disambiguate hash references with `+{ …}`

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 `\` . 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.

```
my @array   = qw( a, b, c);
print $array[1]. # b
```
You can create complex data with references: ⇓⇓⇓

```
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
```

```
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.
```

• Creale a lexical reference: `my $hash_ref = \%hash;`
• Store a ref to an array or hash into an array: `push @array \%hash;`
• Pass array or hash to subroutine: fct(\@a, \%h); Return from sub: return (\@a, \%h);

⬅ Arrows *between* subscript are optional.

Symbolic References
With a simple string it refers to the symbols table of the *main* package. The string can also be fully qualified name, then it uses the specified symbol table.

⚠ 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 *symbolic* 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

(Perl >= v5.20.0) 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:
```
• &{ $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 autovifification

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?

• Checking if a nested data struct element exist will create it: it will always exist!! See BUG section here.
• Prevent that by checking each level data in step.

• It's also possible to lexically disable it, with the pragma: `no autovivification;`

`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.
• In the example at right, a greeter function is built and returned, remembering how to greet. It is used like this:
```
my $fr = make_greeting("Bonjour");
my $it = make_greeting("Buongiorno");
$fr->('Brigitte');  # prints: "Bonjour, Brigitte!\n"
$it->('Madonna');   # prints: "Buongiorno, Madonna!\n"
```

```
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
```

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!

```
{ 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**

• **numeric**:
 	
Note: underline separators can be used inside decimal, hexadecimal and binary literals.
	• 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.
	
	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).
		`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 = 0b1100_0010;  # binary with underlines`
		`my $x = 0xff55;  # hexadecimal`
			• oct - for: 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.
- **\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**

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

Usual	Generic	Meaning	Interpolates?	Notes
`''`	**q//**	Literal string	No	• Not all characters can be used as the **/** separator. **{ }, ( )** and **< >** can also be used.
`""`	**qq//**	Literal string	Yes	
`` `` ``	**qx//**	Command execution	Yes	• You can use whitespace between the quote specifier and its initial bracketing character:

`()`	**qw//**	World list	No	`my $chuck_of_code = q {`
				`  if ($condition) {`
				`  print "Bonjour!";`
				`  }`
				`};`

`//`	**m//**	Pattern match	Yes	
s///	**s///**	Pattern substitution	Yes	
tr///	**y///**	Character translation	No	
`""`	**qr//**	Regular expression	Yes	

- 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)	**\t**	Horizontal tab	**\x{263a}**	Character number 0x263A
\b	Backspace	**\e**	ESC character		
\e	ESC character	**\033**	ESC in octal	Any Unicode code point, by name:	
\f	Form feed	**\o{33}**	ESC in octal	**\N{LATIN SMALL LETTER E WITH ACUTE}**	é
\n	Newline (usually LF)	**\x7f**	DEL in hexadecimal	**\N{ U+E9 }**	é
\r	Carriage return (Usually CR)	**\cC**	Control-C		

• **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: **Note:** They can also be stacked and text can be transformed. See the documentation.
- 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`**

• **Perl Regexp**

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

• **index/substr**

`$pos = index($page, $line);` `$last_slash = rindex("/usr/bin/ls", "/");` `$part = substr($text, $pos, $len)` A value of **-1** in pos identifies last character.

• **Replacement**
• manipulate strings with **substr** LPo

```
my $pref = "I like awk and erlang";
substr($pref, index($pref, "awk"), length("awk")) = "Perl";
substr($pref, 0, 0) = "Sally and ";    # insert text anywhere
```

`substr($pref, -15) =~ s/Perl/Perl5/g;` # replace text inside a restricted portion of the string.

Perl 5 *Special* Literal and Variables

Special Literals

- __FILE__ : current file name
- __LINE__ : current line number
- __PACKAGE__ : current package name
- __SUB__ : reference to current subroutine
- __END__ : use to indicate logical end of script
- __DATA__ : same, but supports reading text

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**

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	• $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 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: `@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 started	• $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	• $^R	• $LAST_REGEXP_CODE_RESULT	
regexp debug flag	${^RE_DEBUG_FLAG}	regexp internal optimization/memory		${^RE_TRIE_MAXBUF}	
• **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 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)	• $RS • IO::Handle->input_record_separator(EXPR) • $/ • $INPUT_RECORD_SEPARATOR		Output record separator	• $ORS • $\ • IO::Handle->output_record_separator(EXPR) • $OUTPUT_RECORD_SEPARATOR	
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

Perl I/O

- 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**, 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!)

say

`use feature qw(say);` or `use v5.10;` (or higher). Like print, but implicitly appends a newline at the end of the list.

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.
- The **<>** operator supports shell redirection and pipe operations which **<<>>** does not allow (for security reasons).
  - The **<<>>** operator is always empty.
- With **<>** 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 **<<>>** operator.
- The **<>** 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 **<<>>** operator.):
  - If **<>** contains only a bareword or a simple scalar variable, it compiles to **readline**, otherwise it compiles to **glob**.

`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.
- **open**
- **open::layers**

Also see process and filehandles inside the Topic: Process Control below.

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	

Using lexical scalar filehandles

open also supports the use of lexical scalar filehandles, a more versatile and safer mechanism.
- 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 **undef** when the **open** statement executes, otherwise open uses it as a file handle value.

Example from Grinnz:

```
• open my $in_fh,  '<', $filename                or die "Failed to open $filename for reading: $!";
• open my $out_fh, '>>:encoding(UTF-8)', $outfile or die "Failed to open $outfile for appending: $!";
```

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`**
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.

- **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 Statements

Perl Syntax

`perldoc perlsyn` :Perl syntax is free-form. It borrowed concepts from many languages. See **`perldoc perltrap`** for comparisons and differences.

Comments

Comments start with a # 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 `eval {}`, `sub {}`, and `do {}` need explicit termination because these are not compound statements but just terms inside an expression.

Statement modifiers

⚠ Do not use with a **my state** and **our**.

A simple statement may be followed by a *single* modifier just before the terminating semicolon:

```
if EXPR                    while EXPR
unless EXPR                until EXPR
```

```
for LIST
foreach LIST
when EXPR     (Perl >= 5.14) Used in switch statement
```

Compound statements

A sequence of statements inside a file, a **{}** delimited block, or an **eval** string constitute a scope.
- Because hash references are also identified by **{}** , it may be necessary to put a semicolon after the opening brace to identify a block. As in: {; .... }

Basic Blocks

Defer blocks

Try Catch

Loop blocks

Specially Named Blocks

5 specially named blocks are run at the beginning or end 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.

Loop control

See **perlsyn** for more information on Perl syntax which includes declarations, blocks, loops, labels, subroutines, etc…

☝ Use the **last** and **redo** inside a naked block of code to control looping.

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	• The do block is ***very useful*** to set a value based on several conditions, just as the ? : conditional operator 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 *not* return from the block.
	• The last, next and redo cannot be used inside do blocks.
		`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' }`
		`}`	

if, elsif, else			
unless			
? : conditional operator			

Perl 5 Subroutines

Perl subroutines	See Object Oriented Perl, section 2.1.4 : Subroutines. • Note that 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.	
• Declaring subroutine In all cases, it's less ambiguous to define the subroutine before use and use parentheses in calls.	• Declare a subroutine to use as a list operator. • use **or** or not **||** because it binds too tightly.	`sub seed_for;` `$val = seed_for $0 or die 'seed_for failed';`
	• Declare a subroutine to use as a unary operator:	`sub seed_for($);` # use subroutine prototype to declare it as unary operator. `$val = seef_for $0 || die 'seed_for failed';`
• Defining subroutine	• Defined with the **sub** keyword followed by a block.	**sub** greet { print "hello!\n"; }
• Calling a subroutine	• If the subroutine definition follows its invocation, parentheses after the subroutine name are required, as in: greet();	• But if the definition was above the call, the parentheses are optional; as in: `greet;` • Subroutine sigil is &. It can optionally be used in a call; as in `&greet;` or `&greet();`
• pass current @_array	• Call with & prefix without args, as in &sub_function; to pass current **@_** array. Used to call a helper subroutine with in the primary one, providing all its arguments.	
• **goto**	• From a subroutine use goto &sub_function; to transfer control to that subroutine instead of calling it. It also passes the current **@_** array to it.	
• calling a method	• Parentheses are required if arguments are passed to method, but optional if there is no arguments.	`$obj->method_with_args($val1, $valb);` `$obj->method_without_arg; $obj->method_without_args();`
• 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 • Note it must be used to make a reference to a subroutine: $greeter = **\&**greet;
• subroutine arguments • **passed by list** • always variable by nature • **named arguments** Note: The **@_** is an **alias** to the passed values; changing them inside the subroutine affects the caller's values.	• The arguments passed to a subroutine are available to its code via the special **@_** array. • The caller code supplies a list of values. Remember that: • nested lists lists are flattened in Perl.	@sorted = alpha_order('Nice', 'Québec', 'Montréal'); @sorted = number_order @unsorted_numbers; @sorted = alpha_order('Trois-Rivières', @sorted, 'Gaspé', 'Rimouski');

	• 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.
		Implementation: `sub move { my (%directions) = @_; ... }`
		Caller: `move(up=>3, left=>4); move('down', 2);` # it's by convention!
		To set a default:
		`sub move {`
		`%default = (up=>0, down=0, left=>0, right=>0);`
		`my (%directions) = (%default, @_);`
		`... }`

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, …)**
Returned value. Detecting calling context with **wantarray**	• 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 calling context of the subroutine call and why it should return:	
Identify caller	The **caller** 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 $AUTOLOAD subroutine it calls that.	Also see: **AutoLoader**.
Continuation with goto	The **goto** 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 • **Perl OO Tutorial** • Perl Module Library • **Module creation guideline** Other: • Corinna Class Tutorial • Object Oriented Perl by Damian Conway	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.	

	`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.');`
	
	# 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.
	`$empl->set_office('L1-100');`
	
	Note the that calling Employee::new directory, no object reference is passed; therefore the arrow nation is required.
		`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;`
		`}`

	• By convention, something a name that starts with an underscore is *internal*, not meant to be used directly. • 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. • 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 **@ISA array**. In code set them by identifying them via the **use parent pragma**.	

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	• How to detect where a module is installed : `perldoc -l Module` • How to check if a module is part of Perl core : `corelist Module` *(Perl >= v5.9.2)*

Access to Modules		Provide access to modules in your code with one of the following: **do** , **require** or **use**
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. • Skip the `@INC` path lookup if given a file path starting with `./`, `../`, or `/`
	require	Loads the module file once, also searching 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**). • Skip the **@INC** path lookup if given a file path starting with `./`, `../`, or `/`
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**. Modify it dynamically in a **BEGIN** block. See IntPo. • 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. Select what to import with one of the two equivalent forms: (See IntPo): • `use Module::Name ('function_a', 'function_b');` • `use Module::Name qw( function_a function_b );` • `use Module::Name ();` # import nothing. All accesses to the module must be done with `Module::Name::something`
Error handling for: **Can't locate in @INC** • **How to fix that** See Also: IntPo • See: show-perl-inc @ USRHOME		For the above statements to work Perl must be able to identify the location of the requested module(s). • Perl looks for a module code inside the directories identified by the **@INC** array. if you have. `use The::Module;` inside your code, Perl looks for a sub-directory named 'The' containing a file named 'Module.pm' inside each **@INC** directory. If Perl does not find it, there are multiple ways to solve the problem: • 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 `use lib` 'path/to/the/directory'; 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. To List the directories used by Perl from one of the following equivalent command lines: • `perl -e 'print join("\n", @INC), "\n";'` • `perl -le 'print for INC';'` You can also get more information with `perl -V`
Declare **packages**		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. • 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					
Using **Perl Debugger** • **Debugger Tutorial**	Debug a program:	**perl -d** program_name program_args			
	Debug interactive session:	**perl -d -e 0**			
Debugger commands	**q**	Quit debugger	**s**	single step	
	h	help. List all available commands.	**x**	evaluate expression	

Modules for Data introspection			
	Data::Dumper *(Perl >= 5.005)* It provides the Dumper function that prints strings that can be used by **eval** to rebuild the data.	• 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.	• `print Dumper(\@array);` • `print Dumper \%hash;`
	Data::Dump *(Requires Perl >= v5.6.0)*	Provides a dump function that has nicer output, but is not **eval** compatible. • dump() prints on the stdout. No need to use print.	use **Data::Dump qw(dump)**; **dump**(\@array); **dump**(\%hash);
	Data::Printer A nicer data dumper, not **eval** compatible.	• 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.	use **Data::Printer**; **p**(@array); **p**(%hash);
Data Marshalling • **Data Serialization**	There are several modules, either part of Perl core or outside, that provides mechanism to marshall/serialize and unmarshall/de-serialize data. • See the links at left for more info.		

Topic: Directory Operations

Directory Operations	In Books: LPo	
Opening Files	All file open operations are relative to the *current working directory* (for relative file names)	`open my $filehandle, '<:utf8', 'a_relative/path.txt'`
Creating temporary files	**File::Temp** (Perl >= v5.6.1). Using File::Temp • Also see **IO::File**	

Built-in Functions	**Related Functions/Packages / Descriptions**	**Notes**	
Getting file names by: • Globbing : • with **glob** • with the glob operator **<>**	**File::Glob** *(Perl >= v5.6.0)* - provides more control.	Example:	`my @all_files = glob '*';` `my @perl_files = glob '*.pm *.pl'; # 2 globs, space-separated`

	The <> operator is identifying:
	• a **filehandle**, when: the item inside <> is a Perl identifier or an indirect file handle read scalar,
	• a **glob expression** otherwise.
		Glob examples:	`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`
	See: **readline**	Filehandle examples:	`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;`

• with a directory handle
LPo
	• **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.	`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;`

Creating directory	• **mkdir**	Example:	`mkdir $dir_name, oct($permissions); # octal for permissions` `mkdir $dir_name, 0700; # do not use "0700", it's 700 decimal!`
Removing directory	• **rmdir** Removes an **empty** directory. • **File::Path remove_tree** , **rmtree** remove dir & files (Perl >= v5.0.1)		

Removing files	• **unlink** a list or **$_**		`unlink 'file1.txt', 'file2.txt';`
			`unlink qw( file1.txt file2.txt);`
			`unlink glob 'file?.txt'`

Renaming files	• **rename** an old file name to a new one. • The fat comma operator is sometimes used to highlight what is the old and the new name.	As in here:	`rename 'old_name' , 'new_name';` `rename  old_name => 'new_name';  # use fat comma to quote word left of it.`
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	• File::chdir • File::HomeDir	• Change the current working directory. • **chdir** without argument attempt to change to user home directory using the `$ENV{HOME}` and `$ENV{LOGDIR}` 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** **Legend: Exported by default,** exported on request, *Win32 specific*	**Extra Information**

Cwd	• **getcwd**, **cwd**, **fastcwd**, **fastgetcwd**, *getdcwd*
	• abs_path, realpath, fast_abs_path
		`use Cwd;`
		`my $curdir = getcwd;`
		`print "cwd is $curdir\n";`

File::Basename	• **fileparse**, **basename**, **dirname**,	
File::Spec **File::Spec::Functions**	• functional interface to methods: **canonpath**, **catdir**, **catfile**, **curdir**, **rootdir**, **updir**, **no_upwards**, **file_name_is_absolute**, **path**. devnul, tmpdir, case_tolerant, splitpath, splitdir, catpath, abs2rel, rel2abs. All can be imported by using the `:ALL` tag.	

File::Find : Traverse a directory tree.
See: File::Find::Closures
	find, **finddepth**, **%options**. In **wanted**: **File::Find::dir**, **File::Find::name**
	Note that **$_** gets the base name of the file (no path). It is used to perform filetest operations in the example here (as explicit argument to -s, and implicit argument to -d and -f). This traverses the **entire** tree.
		`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

Topic: List Operations 🚧

List Operators					
Sorting lists	**sort**	Sort a list	`my @sorted = sort @unsorted_list;`	in place:	`my @data = sort @data;`
	reverse	Sort a list in reverse order	`my @rsorted = reverse @unsorted_list;`	in place:	`my @data = reverse @data;`
Filtering list with **grep**	`my @adult_ages = grep $_ > 18, @ages;`		`my @lucky_ages = grep /7$/, @ages;  # all that end with 7`		`my @read_ages = grep { $_ >= 7 && $_ <= 77 } @ages;`
Counting matches	`my $count = grep $_ > 18, @ages;`				
	An expression, subroutine or block with trailing boolean can be used as the grep criteria. Each item in the list is identified inside grep by **$_** • 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**					

Topic: Process control 🚧

Process Control	In Books: LPo	Important security information: **perldoc perlsec**
Environment Variables	Inside the **%ENV** hash.	Perl **%Config** hash: Perl configuration information. For example, whether it support threads, what are path separators, etc… • To use it: `use Config;`

Built-in Functions	**Example**	**Description/ Notes**
system (2 functions) • using the shell • **security risk?**	`system 'ls -l $HOME';`	Run child process asynchronously using parent's stdin, stdout and stderr, using the OS native command shell.
	`system "cd $project; make &";`	Use the Unix shell to execute a long running build asynchronously. ☝ However: avoid using the shell like this. • Using the shell to build commands from unvalidated user input data may lead to security issues.
• avoiding the shell	**system** 'tar', 'cvf', $tarfile, @directories;	No shell invoked when more than 1 argument is passed to system. No shell interpretation, piping, re-direction done.
• other syntax	**system**('tar', @arguments);	0 means success: `unless ( system 'tar', arguments) { print "tar command success\n"; }`
	system({ $prog }, $arg0, @args);	
	☝ Note that if the string contain **no** shell **metacharacters** it is executed directly (not through a shell).	

system return value:
• A value of 0 ***usually*** means all was OK.
	2 bytes: MSByte: child program exit code.
	LSByte: system-specific information bits:
	• 0x80 : set on core dump.
	• 0x7f : **signal** number
		`my $retval = system( … );`
		`my $childp_exitcode = $retval >> 8;` ⬅ shift most significant byte
		`my $had_core_dump  = ($retval & 0x80) == 0x80? 1 : 0;`
		`my signal_number  =  $retval & 0x7f;` ⬅ use least significant byte

exec	Unlike system, **exec** does not return to the parent Perl process. Use:	`exec 'the_program' or die "Could not run: $!"; #or warn or exit`
backquotes ``	Use backquotes to **capture the stdout** of a program. That's the main point of using it. • The trailing newline is not filtered out; it can be filter by **chomp**.	`chomp( my $current_date = `date` );`

	• 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.
	  • The following example builds a dictionary (hash) of topics with the text extracted from perldoc.
	• Note that `…` is also written as **qx/ … /**
	• backquote operation in scalar context returns 1 string. In list context it returns a list of strings (1 per line).
		`my @topics = qw( die warn exit );`
		`my %info;`
		`foreach (@topics) {`
		`  $info{$_} = `perldoc -t -f $_`;`
		`}`

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	`open DATE, 'date |'  or die "Cannot pipe from date: $!";`		Use a bare word to define the DATE file handle.
		`open my $date_fh, '-|', 'date' or die "Cannot pipe from date: $!";`		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.
		`open my $ps_fh, '-|', 'ps', 'aux' or die "Cannot pipe from ps: $!";`		
		`open my $find_fh, '-|', 'find', qw( . -name '*.p[lm]' -print ) or die "Cannot pipe from find: $!";`		
Perl ➡ program	Launching a process that the Perl process pipes into.	`open my $dispather_fh, '|-', 'dispatcher', qw ( '—to-perl-groups' 'Help!' ) or die "Cannot pipe to the dispatcher: $!";`		
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:
• Other IPC functions
• Perl IPC
	• fork the process into parent and child.
	• in the child process start the program with exec
	• In the parent process wait for the program termination with waitpid
		`defined(my $process_id = fork) or die "Fork failed: $!";`
		`unless ($process_id) {`
		`  # Inside the child process (created by fork)`
		`  exec 'long_running_process' or die "Failed starting long_running_process: $!";`
		`}`
		`# Inside the parent process, wait for completion of long_running_process.`
		`waitpid($process_id, 0);`

Signals	In Books: LPo	

kill	Sends a signal to a list of processes.
	• The signal may be identified by number or name (string), which is more portable.
	• The `%Config{sign_name}` provides the supported signal names.
	• Note that the *fat comma* operator (=>) can be used to automatically quote signal name:
		`kill 'INT', $pid or die "Can't signal $pid with SIGINT: $!";`
		
		`kill INT => $pid or die "Can't signal $pid with SIGINT: $!";`

	• 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.	`unless (kill 0, $process_id) {`
		`  warn "Process $process_id is no longer running!";`
		`}`

	• 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.	• `kill '-KILL', $process_group` • `kill -9, $process_group`
Signal handlers	• Set the signal handler by setting **%SIG** for the signal name (with no 'SIG' prefix) to a string holding the name of the subroutine.	`$SIG{'INT'} = 'dispatcher_int_handler';`
Error Logging and Reporting	• Perl supports the warn buil-in to generate warnings on stderr. • The **Carp::carp** from the **Carp** 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	• **-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**.