Beginning Perl

Mark Senn

September 11, 2007

Overview

Perl is a popular programming language used to write systen software, text processing tools, World Wide Web CGI programs, etc. It was written by Larry Wall in 1986. He and others continue to support and make improvements to it.

Overview

Perl is a popular programming language used to write systen software, text processing tools, World Wide Web CGI programs, etc. It was written by Larry Wall in 1986. He and others continue to support and make improvements to it.

If you know how to use a text editor and have experience in at least one programming language this short course is designed to try and teach you enough to write simple programs in Perl.

Overview

Perl is a popular programming language used to write systen software, text processing tools, World Wide Web CGI programs, etc. It was written by Larry Wall in 1986. He and others continue to support and make improvements to it.

If you know how to use a text editor and have experience in at least one programming language this short course is designed to try and teach you enough to write simple programs in Perl.

Perl is a *very* rich language. This short course won't cover lots of stuff: complicated data structures, object oriented features, packages available to do specialized tasks, subroutines, etc. See "References" section for how to get information about these topics.

Choosing a Language

For projects that involve	I usually use
all stuff I used to use sh or csh for	Perl
almost all stuff I used to use C for	Perl
character string manipulation	Perl
programs for one-time use	Perl
"system tools"	Perl
mathematics	*Mathematica*
low-level software to control hardware	C
working with others	whatever we agree upon

Development and Run Times

This real-world example is from Diego Zamboni. Three different people each wrote a program for generating decision trees from a large dataset in a different lanaguage.

Development and Run Times

This real-world example is from Diego Zamboni. Three different people each wrote a program for generating decision trees from a large dataset in a different lanaguage.

Language	Development Time	Run Time
Perl	1 day	$\approx$ 4 hours
Java	3–4 days	$\approx$ 10 minutes
C	$\approx$ 1 week	$\approx$ 2 minutes

References

Type `perldoc perldoc` to see information on `perldoc`. `Perldoc` is used to look at Perl documentation based on manual page name, module name, or program name. Type `perldoc perl` to see information regarding Perl itself.

References

Type `perldoc perldoc` to see information on `perldoc`. `Perldoc` is used to look at Perl documentation based on manual page name, module name, or program name. Type `perldoc perl` to see information regarding Perl itself.

You may find *Learning Perl*, 2nd Edition, ISBN 1-56592-284-0, by Randal L. Schwartz and Tom Christiansen helpful to learn Perl. Chances are good that if you have a little programming experience you may be able to get by with `perldoc` and/or *Programming Perl*.

References

Type `perldoc perldoc` to see information on `perldoc`. `Perldoc` is used to look at Perl documentation based on manual page name, module name, or program name. Type `perldoc perl` to see information regarding Perl itself.

You may find *Learning Perl*, 2nd Edition, ISBN 1-56592-284-0, by Randal L. Schwartz and Tom Christiansen helpful to learn Perl. Chances are good that if you have a little programming experience you may be able to get by with `perldoc` and/or *Programming Perl*.

I consider *Programming Perl*, 3nd Edition, ISBN 0-596-00027-8, by Larry Wall, Tom Christiansen, and Jon Orwant essential for *serious* Perl programming. Advice: see if `perldoc` suits your needs before getting any books.

References (2)

See `http://www.perl.com` for pointers to all things Perl. Perl book information is available at `http://reference.perl.com/query.cgi?books`.

References (2)

See `http://www.perl.com` for pointers to all things Perl. Perl book information is available at `http://reference.perl.com/query.cgi?books`.

See `http://www.cpan.org` for CPAN, the Comprehensive Perl Archive Network, containing 3,874 Mbytes of code as of September 2007.

Starting Perl

Use `#!/usr/local/bin/perl -w` as the first line of Perl programs. The # must be the first character in the file. The `-w` option will *w*arn you about variables being used before being set, etc.

Starting Perl

Use `#!/usr/local/bin/perl -w` as the first line of Perl programs. The # must be the first character in the file. The `-w` option will *w*arn you about variables being used before being set, etc.

```
#!/usr/local/bin/perl -w

print "hello, world\n";
```

Starting Perl

Use `#!/usr/local/bin/perl -w` as the first line of Perl programs. The # must be the first character in the file. The `-w` option will *w*arn you about variables being used before being set, etc.

```
#!/usr/local/bin/perl -w

print "hello, world\n";
```

prints

Starting Perl

Use `#!/usr/local/bin/perl -w` as the first line of Perl programs. The # must be the first character in the file. The `-w` option will *w*arn you about variables being used before being set, etc.

```
#!/usr/local/bin/perl -w

print "hello, world\n";
```

prints

```
hello, world
```

Using Perl Interactively

Type `perl -de 0` (that's a zero) to start Perl in an interactive mode. This is an easy way to get familiar with Perl. Type Control-D to get back to Unix. For example:

Using Perl Interactively

Type `perl -de 0` (that's a zero) to start Perl in an interactive mode. This is an easy way to get familiar with Perl. Type Control-D to get back to Unix. For example:

```
% perl -de 0
Default die handler restored.
Loading DB routines from perl5db.pl version 1.07
Editor support available.
Enter h or ‘h h’ for help, or ‘man perldebug’ for more
help.
main::(-e:1):  0
DB<1> $a = 3
DB<2> $b = $a + 4
DB<3> print "b is $b"
b is 7
DB<4>  [Type Control-D here.]
%
```

Command Line Perl

Typing `perl -e '`*expression*`'` let's one evaluate Perl expressions from the command line. Small Perl programs can be run this way.

Command Line Perl

Typing `perl -e '`*expression*`'` let's one evaluate Perl expressions from the command line. Small Perl programs can be run this way.

```
perl -e 'print "hello\n";'
```

Command Line Perl

Typing `perl -e '`*expression*`'` let's one evaluate Perl expressions from the command line. Small Perl programs can be run this way.

```
perl -e 'print "hello\n";'          hello
```

Command Line Perl

Typing `perl -e '`*expression*`'` let's one evaluate Perl expressions from the command line. Small Perl programs can be run this way.

```
perl -e 'print "hello\n";'          hello
```

Type `perldoc perlrun` or see a Perl book for more information.

General Syntax

Comments start with # and go to the end of the line.

General Syntax

Comments start with # and go to the end of the line.

Every statement should be followed by a semicolon (;).

General Syntax

Comments start with # and go to the end of the line.

Every statement should be followed by a semicolon (;).

The if and else parts of an if must always use curly braces, even if they are just one statement.

Variables

Variable names must start with a letter (`A`–`Z`, `a`–`z`) or underscore (`_`).
Any characters after the first must be letters, underscores, or digits.

Variables

Variable names must start with a letter (`A`–`Z`, `a`–`z`) or underscore (`_`). Any characters after the first must be letters, underscores, or digits.

There are three main variable types:

Variables

Variable names must start with a letter (`A`–`Z`, `a`–`z`) or underscore (`_`). Any characters after the first must be letters, underscores, or digits.

There are three main variable types:

- scalars

Variables

Variable names must start with a letter (`A`–`Z`, `a`–`z`) or underscore (`_`). Any characters after the first must be letters, underscores, or digits.

There are three main variable types:

- scalars
- arrays

Variables

Variable names must start with a letter (`A`–`Z`, `a`–`z`) or underscore (`_`). Any characters after the first must be letters, underscores, or digits.

There are three main variable types:

- scalars
- arrays
- hashes

Scalars

A scalar contains a single number or string.

Scalars

A scalar contains a single number or string.

Scalar names start with $.

Scalars

A scalar contains a single number or string.

Scalar names start with $.

Here are some examples of setting scalars to values:

```
$_     = $line;
```

Scalars

A scalar contains a single number or string.

Scalar names start with $.

Here are some examples of setting scalars to values:

```
$_    = $line;
$n    = 1000;
```

Scalars

A scalar contains a single number or string.

Scalar names start with $.

Here are some examples of setting scalars to values:

```
$_    = $line;
$n    = 1000;
$name = 'Mark Senn';
```

Scalars

A scalar contains a single number or string.

Scalar names start with $.

Here are some examples of setting scalars to values:

```
$_    = $line;
$n    = 1000;
$name = 'Mark Senn';
$pi   = 3.14115926535;
```

Scalars

A scalar contains a single number or string.

Scalar names start with $.

Here are some examples of setting scalars to values:

```
$_    = $line;
$n    = 1000;
$name = 'Mark Senn';
$pi   = 3.14115926535;
```

Strings get converted to numbers automatically when doing arithmetic operations.

Scalars

A scalar contains a single number or string.

Scalar names start with $.

Here are some examples of setting scalars to values:

```
$_    = $line;
$n    = 1000;
$name = 'Mark Senn';
$pi   = 3.14115926535;
```

Strings get converted to numbers automatically when doing arithmetic operations.

The default variable is $_. If you don't explicitly say what variable to operate on, $_ is often used. To specify another variable use =~ like this:

```
$var =~ /abc/;
```

Arrays

An array contains a list of values.

Arrays

An array contains a list of values.
Array names start with `@`.

Arrays

An array contains a list of values.
Array names start with `@`.

Here are some examples of setting arrays to values:

```
@prime = (2, 3, 5);
```

Arrays

An array contains a list of values.
Array names start with `@`.

Here are some examples of setting arrays to values:

```
@prime = (2, 3, 5);
@mixed = (1, 2.3, 'word', 'more than one word');
```

Arrays

An array contains a list of values.
Array names start with `@`.

Here are some examples of setting arrays to values:

```
@prime = (2, 3, 5);
@mixed = (1, 2.3, 'word', 'more than one word');
@month = ('Jan', 'Feb', 'Mar');
```

Arrays

An array contains a list of values.
Array names start with `@`.

Here are some examples of setting arrays to values:

```
@prime = (2, 3, 5);
@mixed = (1, 2.3, 'word', 'more than one word');
@month = ('Jan', 'Feb', 'Mar');
# "Quote words" operator makes this easier to type.
@month = qw(Jan Feb Mar);
```

Arrays

An array contains a list of values.
Array names start with `@`.

Here are some examples of setting arrays to values:

```
@prime = (2, 3, 5);
@mixed = (1, 2.3, 'word', 'more than one word');
@month = ('Jan', 'Feb', 'Mar');
# "Quote words" operator makes this easier to type.
@month = qw(Jan Feb Mar);
# "Quote words" works on separate lines.
@month = qw(
    Jan
    Feb
    Mar
);
```

Arrays (2)

```
@month = qw(
    Jan
    Feb
    Mar
    Apr
);
```

Arrays (2)

```
@month = qw(
    Jan
    Feb
    Mar
    Apr
);
```

Array indices start at 0.

Arrays (2)

```
@month = qw(
    Jan
    Feb
    Mar
    Apr
);
```

Array indices start at 0.

`$month[0]` is Jan.

Arrays (2)

```
@month = qw(
    Jan
    Feb
    Mar
    Apr
);
```

Array indices start at 0.

`$month[0]` is Jan.

`$month[1]` is Feb.

Arrays (2)

```
@month = qw(
    Jan
    Feb
    Mar
    Apr
);
```

Array indices start at 0.

`$month[0]` is Jan.

`$month[1]` is Feb.

Negative indices count from the end of the array.

Arrays (2)

```
@month = qw(
    Jan
    Feb
    Mar
    Apr
);
```

Array indices start at 0.

`$month[0]` is Jan.

`$month[1]` is Feb.

Negative indices count from the end of the array.

`$month[-1]` is Apr.

Arrays (2)

```
@month = qw(
    Jan
    Feb
    Mar
    Apr
);
```

Array indices start at 0.

`$month[0]` is Jan.

`$month[1]` is Feb.

Negative indices count from the end of the array.

`$month[-1]` is Apr.

`$month[-2]` is Mar.

Arrays (2)

```
@month = qw(
    Jan
    Feb
    Mar
    Apr
);
```

Array indices start at 0.
`$month[0]` is Jan.
`$month[1]` is Feb.

Negative indices count from the end of the array.
`$month[-1]` is Apr.
`$month[-2]` is Mar.

`$#`*array* is the index of the last element of *var*.

Arrays (2)

```
@month = qw(
    Jan
    Feb
    Mar
    Apr
);
```

Array indices start at 0.
`$month[0]` is Jan.
`$month[1]` is Feb.

Negative indices count from the end of the array.
`$month[-1]` is Apr.
`$month[-2]` is Mar.

`$#`*array* is the index of the last element of *var*.
`$#month` is 3.

Arrays (3)

@*array* used in a scalar context represents the number of elements in the array.

Arrays (3)

@*array* used in a scalar context represents the number of elements in the array.

```
for ($i = 0;  $i < @array;  $i++)  {
    print "element $i is $array[$i]\n";
}
```

Hashes

A hash is like an array but instead of being indexed by integers it is indexed by arbitrary strings. The index is called the key, the value is called the value.

Hashes

A hash is like an array but instead of being indexed by integers it is indexed by arbitrary strings. The index is called the key, the value is called the value. Hash names begin with %.

Hashes

A hash is like an array but instead of being indexed by integers it is indexed by arbitrary strings. The index is called the key, the value is called the value. Hash names begin with %.

Here are some examples of setting hash elements to values:

```
%office = (
    'Mark',   'MSEE 130A',
    'George', 'White House'
);
```

Hashes

A hash is like an array but instead of being indexed by integers it is indexed by arbitrary strings. The index is called the key, the value is called the value. Hash names begin with %.

Here are some examples of setting hash elements to values:

```
%office = (
    'Mark',   'MSEE 130A',
    'George', 'White House'
);
%office = (
    Mark   => 'MSEE 130A',
    George => 'White House'
);
```

Hashes

A hash is like an array but instead of being indexed by integers it is indexed by arbitrary strings. The index is called the key, the value is called the value. Hash names begin with %.

Here are some examples of setting hash elements to values:

```
%office = (
    'Mark',   'MSEE 130A',
    'George', 'White House'
);
%office = (
    Mark   => 'MSEE 130A',
    George => 'White House'
);
$office{'Mark'}   = 'MSEE 130A';
$office{'George'} = 'White House';
```

Hashes

A hash is like an array but instead of being indexed by integers it is indexed by arbitrary strings. The index is called the key, the value is called the value. Hash names begin with %.

Here are some examples of setting hash elements to values:

```
%office = (
    'Mark',   'MSEE 130A',
    'George', 'White House'
);
%office = (
    Mark   => 'MSEE 130A',
    George => 'White House'
);
$office{'Mark'}   = 'MSEE 130A';
$office{'George'} = 'White House';
$office{Mark}   = 'MSEE 130A';
$office{George} = 'White House';
```

Hashes (2)

Type `perldoc perldata` or see a Perl book for more information.

Arithmetic Operators

Operator	Operation	Example
+	addition	17 + 3 → 20
-	subtraction	17 - 3 → 14
*	multiplication	17 * 3 → 51
/	division	17/3 → 5.66...
%	modulo division (remainder)	17 % 3 → 2
<	less than	17 < 3 → 0
<=	less than or equal	17 <= 3 → 0
!=	not equal	17 != 3 → 1
==	equal	17 == 3 → 0
>=	greater than or equal	17 >= 3 → 1
>	greater than	17 > 3 → 1
<=>	compare	17 <=> 3 → 1

$a <=> $b returns

-1 if $a < $b

0 if $a == $b

1 if $a > $b

Quoting and Interpolation

Input	Result
`$x = 'abc'`	`abc`
`$y = '$x'`	`$x`
`$y = q/$x/`	`$x`
`$y = q#$x#`	`$x`
`$y = q($x)`	`$x`

Input	Result
`$x = "abc"`	`abc`
`$y = "$x"`	`abc`
`$y = qq/$x/`	`abc`
`$y = qq#$x#`	`abc`
`$y = qq($x)`	`abc`

Bit-wise Operators

Operator	Operation	Example
`~`	not	`~1` → `all bits 1 except last bit 0`
`|`	or	`1 | 2` → 3
`&`	and	`1 & 3` → 1
`^`	exclusive or	`1 ^ 3` → 2

Logical Operators

Operator	Operation	Example
`&&` *or* `and`	and	`0 and 1` $\longrightarrow$ `0`
`||` *or* `or`	or	`0 or 1` $\longrightarrow$ `1`
`!`	not	`!4` $\longrightarrow$ `0`, `!0` $\longrightarrow$ `1`
`xor`	exclusive or	`0 xor 1` $\longrightarrow$ `1`, `1 xor 1` $\longrightarrow$ `0`

String Operators

Operator	Operation	Example
`.`	catenation	`'abc' . 'def'` → `'abcdef'`
`x`	replication	`'abc' x 4` → `'abcabcabcabc'`
`lt`	less than	`'abc' lt 'def'` → `1`
`le`	less than or equal	`'abc' le 'def'` → `1`
`eq`	equals	`'abc' eq 'def'` → `0`
`ne`	not equals	`'abc' ne 'def'` → `1`
`ge`	greater than or equal	`'abc' ge 'def'` → `0`
`gt`	greater than	`'abc' gt 'def'` → `0`
`cmp`	compare	`'abc' cmp 'def'` → `-1`

	`$a cmp $b` returns
	`-1` if `$a lt $b`
	`0` if `$a eq $b`
	`1` if `$a gt $b`	

If

```
if (expression)  {
    statement1;
    statement2;
}
```

If

```
if (expression)  {
    statement1;
    statement2;
}
```

```
if ($temp > 100)  {
    print "it's hot!\n";
}
```

If

```
if (expression)  {
    statement1;
    statement2;
}
```

```
if ($temp > 100)  {
    print "it's hot!\n";
}
```

```
if (expression)  {
    statement1;
    statement2;
}  else  {
    statement3;
    statement4;
}
```

If

```
if (expression) {
    statement1;
    statement2;
}
```

```
if ($temp > 100) {
    print "it's hot!\n";
}
```

```
if (expression) {
    statement1;
    statement2;
} else {
    statement3;
    statement4;
}
```

```
if ($day eq 'Sat' || $day eq 'Sun') {
    print "it's a weekend\n";
} else {
    print "it's a weekday\n";
}
```

Weird If Statements

The if modifier puts the condition after the statement.

Weird If Statements

The if modifier puts the condition after the statement.

statement `if (`*expression*`);`

Weird If Statements

The if modifier puts the condition after the statement.

statement `if (`*expression*`);`

```
print "bad\n"  if  (!$ok);
```

Weird If Statements

The if modifier puts the condition after the statement.

statement `if (`*expression*`);` `print "bad\n" if (!$ok);`

The shorthand if tests a condition and does the statement if condition is true (for `and`) or false (for `or`).

Weird If Statements

The if modifier puts the condition after the statement.

statement `if (`*expression*`);`

```
print "bad\n"  if  (!$ok);
```

The shorthand if tests a condition and does the statement if condition is true (for `and`) or false (for `or`).

condition `and/or` *statement*;

Weird If Statements

The if modifier puts the condition after the statement.

statement `if (`*expression*`);` `print "bad\n" if (!$ok);`

The shorthand if tests a condition and does the statement if condition is true (for `and`) or false (for `or`).

condition `and/or` *statement*; `$ok or print "bad\n";`

Weird If Statements

The if modifier puts the condition after the statement.

statement `if (`*expression*`);`

```
print "bad\n"  if  (!$ok);
```

The shorthand if tests a condition and does the statement if condition is true (for `and`) or false (for `or`).

condition `and/or` *statement*;

```
$ok  or  print "bad\n";
```

Type `perldoc perlsyn` or see a Perl book for more information.

Loops

Three common types of loops are described here: for, foreach, and while. Perl also has continue, do, goto, labels, unless, and until loop-related statements.

Loops

Three common types of loops are described here: for, foreach, and while. Perl also has continue, do, goto, labels, unless, and until loop-related statements.

For each type of loop, `redo` goes back to the top of the loop, `next` goes to the bottom of the loop, and `last` breaks out of the loop.

Loops

Three common types of loops are described here: for, foreach, and while. Perl also has continue, do, goto, labels, unless, and until loop-related statements.

For each type of loop, `redo` goes back to the top of the loop, `next` goes to the bottom of the loop, and `last` breaks out of the loop.

Type `perldoc perlsyn` or see a Perl book for more information.

For Loops

```
for (init; expression; incr)  {
    # redo comes here
    statement1;
    statement2;
    # next comes here, incr done
}
# last comes here
```

For Loops

```
for (init; expression; incr)  {
    # redo comes here
    statement1;
    statement2;
    # next comes here, incr done
}
# last comes here
```

```
# Print numbers from 1 to 10.
for ($i=1; $i<=10; $i++)  {
    print "$i\n";
}

# Even numbers only.
for ($i=2; $i<=10; $i=$i+2)  {
    print "$i\n";
}

# Even numbers in reverse.
for ($i=10; $i>=2; $i=$i-2)  {
    print "$i\n";
}
```

Foreach Loop

```
foreach (array)  {
    # redo comes here
    statement1;
    statement2;
    # next comes here
}
# last comes here
```

Foreach Loop

```
foreach (array)  {
    # redo comes here
    statement1;
    statement2;
    # next comes here
}
# last comes here
```

```
foreach (@array)  {
    print "$_\n";
}

foreach $element (@array)  {
    print "$element\n";
}

foreach (sort(keys(%hash)))  {
    print "\$hash{$_} is $hash{$_}\n";
}

# works same as previous
foreach (sort keys %hash)  {
    print "\$hash{$_} is $hash{$_}\n";
}
```

While Loop

```
while (expression)  {
    # redo comes here
    statement1;
    statement2;
    # next comes here
}
# last comes here
```

While Loop

```
while (expression)  {
    # redo comes here
    statement1;
    statement2;
    # next comes here
}
# last comes here
```

```
# Read from standard input.
while (<>)  {
    chomp; # delete line-ending
    print "line read was \"$_\"\n";
}

# Read from file handle FH.
while (<FH>)  {
    chomp; # delete line-ending
    print "line read was \"$_\"\n";
}

# Read from file handle FH.
while (<FH>)  {
    chomp; # delete line-ending
    print qq/line read was "$_"\n/;
}
```

Functions

Perl has *lots* of functions. Type `perldoc perlfuns` or see a Perl book for more information.

Functions can be called using

function_name(*argument1*, *argument2*, ...)

Functions

Perl has *lots* of functions. Type `perldoc perlfuns` or see a Perl book for more information.

Functions can be called using

```
function_name(argument1, argument2, ...)
```

or

```
function_name argument1, argument2, ...
```

I prefer the later.

Functions

Perl has *lots* of functions. Type `perldoc perlfuns` or see a Perl book for more information.

Functions can be called using

```
function_name(argument1, argument2, ...)
```

or

```
function_name argument1, argument2, ...
```

I prefer the later.

Here are a few of Perl's built-in functions:

Functions

Perl has *lots* of functions. Type `perldoc perlfuns` or see a Perl book for more information.

Functions can be called using

function_name(*argument1*, *argument2*, ...)

or

function_name argument1, *argument2*, ...

I prefer the later.

Here are a few of Perl's built-in functions:

`chomp` *string*

Delete line-ending characters from *string*.

Functions (2)

`die` *message*

Print "*message* at *program* line *line*" and abort program.

For example, if `$fn` is set to "f" in program "p"

```
die qq/Can't open "$fn": $!, stopped/;
```

might print

```
Can't open "f": permission denied, stopped at p
line 4.
```

Functions (2)

`die` *message*

Print "*message* at *program* line *line*" and abort program.
For example, if `$fn` is set to "f" in program "p"

```
die qq/Can't open "$fn": $!, stopped/;
```

might print

```
Can't open "f": permission denied, stopped at p
line 4.
```

`grep` *expression*`,` *array*

For each element of *array*, set `$_` to that element, and evaluate *expression*. If value is used in an array context return all elements of *array* for which *expression* was true. If value is used in a scalar context return number of times *expression* was true.

Functions (3)

`keys` *hash*

Returns an array consisting of all of *hash*'s keys in no particular order. In `$name{$k} = $v`, `$k` is a key and `$v` is a value. (See values.)

Functions (3)

`keys` *hash*

Returns an array consisting of all of *hash*'s keys in no particular order. In `$name{$k} = $v`, `$k` is a key and `$v` is a value. (See values.)

`length` *string*

Returns number of characters in *string*.

Functions (3)

`keys` *hash*

Returns an array consisting of all of *hash*'s keys in no particular order. In `$name{$k} = $v`, `$k` is a key and `$v` is a value. (See values.)

`length` *string*

Returns number of characters in *string*.

`pop` *array*

Delete last element from *array* and return deleted element as value of function. (See push.)

Functions (3)

`keys` *hash*

Returns an array consisting of all of *hash*'s keys in no particular order. In `$name{$k} = $v`, `$k` is a key and `$v` is a value. (See values.)

`length` *string*

Returns number of characters in *string*.

`pop` *array*

Delete last element from *array* and return deleted element as value of function. (See push.)

`push` *array*, *scalar*

Adds *scalar* to the end of *array*.

Functions (3)

`keys` *hash*

Returns an array consisting of all of *hash*'s keys in no particular order. In `$name{$k} = $v`, `$k` is a key and `$v` is a value. (See values.)

`length` *string*

Returns number of characters in *string*.

`pop` *array*

Delete last element from *array* and return deleted element as value of function. (See push.)

`push` *array*, *scalar*

Adds *scalar* to the end of *array*.

`sort` *array*

Return sorted *array* as value of function.

Regular Expressions

Regular expressions are used to match strings. There are *lots* of things you can put in a regular expressions. This is a *simple* and *incomplete* introduction. Type `perldoc perlre` or see a Perl book for more information.

Regular Expressions

Regular expressions are used to match strings. There are *lots* of things you can put in a regular expressions. This is a *simple* and *incomplete* introduction. Type `perldoc perlre` or see a Perl book for more information.

Character Classes

A character class is a set of characters. It is used to match any of the characters in the character class. A character class starts with `[` and ends with `]`. In between put the characters you want matched. For example, `[abcn-z]` matches `a` or `b` or `c` or any of the characters `n` through `z`. If the first character after the `[` is `^` (uparrow) that means match all characters *not* in the character class.

Regular Expressions (2)

Matching a single character

Use This	To Match
`.`	any character except newline (`\n`)
`\d`	a digit (`[0-9]`)
`\D`	a non-digit (`[^0-9]`)
`\f`	form feed (Control-L, ASCII decimal 12)
`\n`	newline (Control-J, ASCII decimal 10)
`\r`	carriage return (Control-M, ASCII decimal 13)
`\s`	a space character (`[ \r\t\n\f]`)
`\S`	a non-space character (`[^\r\t\n\f]`)
`\t`	tab (Control-I, ASCII decimal 9)
`\w`	a word character (`[a-zA-Z0-9_]`)
`\W`	a non-word character (`[^a-zA-Z0-9_]`)
`\\`	`\`
most `anything else`	it

Regular Expressions (3)

Matching a sequence of characters To group something put it inside parentheses. This let's you specify a repeat count for what's in parentheses or let's you refer to what was matched within parentheses when doing a substitution. **Repetition** To match a single character or a sequence of characters put a repetition count after the character or group specification:

Repetition Count	Matches
?	0 or 1 times
*	0 or more times
+	1 or more times
$\{m,n\}$	at least m but no more than n times
$\{m,\}$	at least m times
$\{,n\}$	no more than n times

Beginning and Ending of Lines ^ matches the beginning of a line. $ matches the end of a line.

Match Operator

The match operator is used to test a string to see if it matches a regular expression.

```
if (/ab/)  {
    print "\"ab\" is in \"$_\"\n";
}

# Same as previous example except use qq
# operator to ‘‘double quote’’ " so we don’t
# have to use \" inside double quoted string.
if (/ab/)  {
    print qq/"ab" is in "$_".\n/;
}

# Same as previous example.
/ab/  and  print qq/"ab" is in "$_"\n/;
```

Match Operator (2)

```
if ($str =~ /yz$/)  {
    print qq/"yz" at end of line in "$str".\n/;
}

# Same as previous example.
($str =~ /yz$/)
    and  print qq/"yz" at end of line in "$str".\n/;

# Set $match to 1 if second character
# on line of $str is "X", 0 otherwise.
$match = ($str =~ /^.X/);
```

Substitution Operator

The general form of the substitution operator is s/*old*/*new*/;.

Substitution Operator

The general form of the substitution operator is s/*old*/*new*/;.

By default the substitution is done on $_.

Substitution Operator

The general form of the substitution operator is s/*old*/*new*/;.

By default the substitution is done on $_.

To do it on another scalar do, for example:

Substitution Operator

The general form of the substitution operator is s/*old*/*new*/;.

By default the substitution is done on $_.

To do it on another scalar do, for example:

```
$home =~s/White House/Texas/; # after his term is over
```

Substitution Operator (2)

Here are some examples of the substitution command:

Command	$_ before	$_ after	Comment
`s/abc/def/;`	`xyz`	`xyz`	no change
`s/abc/def/;`	`abc`	`def`	found it
`s/abc/def/;`	`abcabc`	`defabc`	only first done
`s/abc/def/g;`	`abcabc`	`defdef`	/g does it globally
`s/a*bc/bc/;`	`abcabc`	`bcabc`	repeat count after a
`s/a*bc/bc/g;`	`abcabc`	`bcbc`	/g does it globally
`s/^a/A/;`	`abcabc`	`Abcabc`	^ matches beginning of line
`s/c$/C/;`	`abcabc`	`abcabC`	$ matches end of line
`s/[ac]/x/;`	`abcabc`	`xbcabc`	character class for a or c
`s/[ac]/x/g;`	`abcabc`	`xbxxbx`	/g does it globally
`s/[a-c]/x/g;`	`abcabc`	`xxxxxx`	character class for a–c
`s/a//;`	`abcabc`	`bcabc`	substituting with nothing
`s/.//;`	`abcabc`	`bcabc`	. matches anything

Substitution Operator (3)

Command	$_ before	$_ after	Comment
`s/^.(.*).$/\1/;`	`abcabc`	`bcab`	delete first and last characters of line
`s/\d*$//;`	`abc123`	`abc`	delete trailing digits
`s#/home/##;`	`/home/abc`	`abc`	don't use have to use /

Substitution Operator (3)

Command	$_ before	$_ after	Comment
`s/^.(.*).$/\1/;`	`abcabc`	`bcab`	delete first and last characters of line
`s/\d*$//;`	`abc123`	`abc`	delete trailing digits
`s#/home/##;`	`/home/abc`	`abc`	don't use have to use /

For more information type `perldoc perlop` or see a Perl book.

File Manipulation

Here are the functions necessary to open, read, print on, and close files.

File Manipulation

Here are the functions necessary to open, read, print on, and close files.

One manipulates a file by specifying a file name and/or a file handle. I usually use `$fn` for the file name and `FH` for the file handle. Or, for example, `$ifn` and IFH for input files, if there is more than one file open at a time. Note that the file handle is is not preceded by $, @, or %.

File Manipulation

Here are the functions necessary to open, read, print on, and close files.

One manipulates a file by specifying a file name and/or a file handle. I usually use `$fn` for the file name and `FH` for the file handle. Or, for example, `$ifn` and IFH for input files, if there is more than one file open at a time. Note that the file handle is is not preceded by $, @, or %.

Type `perldoc perlfunc` or see a Perl book for more information.

File Manipulation (2)

```
#  Read lines from file "filename"
#  and print lines on standard output.

$fn = "filename";

# Use ‘<’ to open a file for input.
# Some people use    open FH, "<$fn" ...
open FH, ’<’, $fn
    or  die qq/Can’t open "$fn" for input: $!, stopped/;

while (<FH>)  {
    chomp;  # I like to chomp everything for consistency.
    print "$_\n";
}

close FH
    or  die qq/Can’t close input file "$fn": $!, stopped/;
```

File Manipulation (3)

```
#  The loop for a program that only reads the input
#  file until an "END" line and only checks every 10th
#  line for somethig that looks like a room number.
$n = 0;
while (<FH>)  {
    chomp;
    $n++;
    (/^END$/)  and  last;
    ($n % 10)  and  next;
    (/
        (^|\b)       # beginning of line or word break
        [A-Z]{2,4}   # building abbreviation
        \            # a single space
        \d+          # room number
        (\b|$)       # word break or end of line
    /x)  and  print "$_\n";
}
```

File Manipulation (4)

```
#  Print "Testing..." on file "filename".

$fn = "filename";

# Use ‘>’ to open a file for output.
# Some people use    open FH, ">$fn" ...
open FH, ’>’, $fn
    or  die qq/Can’t open "$fn" for output: $!, stopped/;

print FH "Testing...\n";

close FH
    or  die qq/Can’t close output file "$fn": $!, stopped/;
```