Overview

Perl is a popular programming language used to write systems 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 systems 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 systems 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
all stuff I used to use sh or csh for
almost all stuff I used to use C for
character string manipulation
programs for one-time use
“system tools”
mathematics
low-level software to control hardware
working with others

I usually use
Perl
Perl
Perl
Perl
Perl
Mathematica
C
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 language.
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 language.

<table>
<thead>
<tr>
<th>Language</th>
<th>Development Time</th>
<th>Run Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perl</td>
<td>1 day</td>
<td>≈ 4 hours</td>
</tr>
<tr>
<td>Java</td>
<td>3–4 days</td>
<td>≈ 10 minutes</td>
</tr>
<tr>
<td>C</td>
<td>≈ 1 week</td>
<td>≈ 2 minutes</td>
</tr>
</tbody>
</table>
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.
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*. 
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 (2)

References (2)


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 warn 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 warn you about variables being used before being set, etc.

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

print "hello, world\n";
```
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 warn you about variables being used before being set, etc.

```perl
#!/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 warn 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'` lets one evaluate Perl expressions from the command line. Small Perl programs can be run this way.
Command Line Perl

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

```
perl -e 'print "hello\n";
```
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
```
Typing `perl -e 'expression'` lets one evaluate Perl expressions from the command line. Small Perl programs can be run this way.

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

Type `perldoc perlrunt` or see a Perl book for more information.
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 $.
A scalar contains a single number or string.

Scalar names start with $.

Here are some examples of setting scalars to values:

```perl
$_ = $line;
```
Scalars

A scalar contains a single number or string.

Scalar names start with $.

Here are some examples of setting scalars to values:

```perl
$_ = $line;
$n = 1000;
```
A scalar contains a single number or string.

Scalar names start with $.

Here are some examples of setting scalars to values:

```perl
$_ = $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:

```perl
$_   = $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  = 3.1415926535;

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:

```perl
$_   = $line;
$n   = 1000;
$name = 'Mark Senn';
$pi   = 3.1415926535;
```

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:

```perl
$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:

```perl
@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:

```perl
@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:

```perl
@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:

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

```perl
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:

```perl
%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:

```perl
%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:

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

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

$office{’Mark’}  =  'MSEE 130A';
$office{’George’} =  'White House';
```
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:

```perl
%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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>addition</td>
<td>$17 + 3 \rightarrow 20$</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
<td>$17 - 3 \rightarrow 14$</td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
<td>$17 \times 3 \rightarrow 51$</td>
</tr>
<tr>
<td>/</td>
<td>division</td>
<td>$17/3 \rightarrow 5.66\ldots$</td>
</tr>
<tr>
<td>%</td>
<td>modulo division (remainder)</td>
<td>$17 % 3 \rightarrow 2$</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
<td>$17 &lt; 3 \rightarrow 0$</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal</td>
<td>$17 \leq 3 \rightarrow 0$</td>
</tr>
<tr>
<td>!=</td>
<td>not equal</td>
<td>$17 \neq 3 \rightarrow 1$</td>
</tr>
<tr>
<td>==</td>
<td>equal</td>
<td>$17 == 3 \rightarrow 0$</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal</td>
<td>$17 \geq 3 \rightarrow 1$</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td>$17 &gt; 3 \rightarrow 1$</td>
</tr>
<tr>
<td>&lt;=&gt;</td>
<td>compare</td>
<td>$17 \leq&gt; 3 \rightarrow 1$</td>
</tr>
</tbody>
</table>

$a \leq> b$ returns
- $-1$ if $a < b$
- $0$ if $a == b$
- $1$ if $a > b$
Quoting and Interpolation

<table>
<thead>
<tr>
<th>Input</th>
<th>Result</th>
<th>Input</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = 'abc'</td>
<td>abc</td>
<td>$x = &quot;abc&quot;</td>
<td>abc</td>
</tr>
<tr>
<td>$y = '$x'</td>
<td>$x</td>
<td>$y = &quot;$x&quot;</td>
<td>abc</td>
</tr>
<tr>
<td>$y = q/$x/</td>
<td>$x</td>
<td>$y = qq/$x/</td>
<td>abc</td>
</tr>
<tr>
<td>$y = q#$x#</td>
<td>$x</td>
<td>$y = qq#$x#</td>
<td>abc</td>
</tr>
<tr>
<td>$y = q($x)</td>
<td>$x</td>
<td>$y = qq($x)</td>
<td>abc</td>
</tr>
</tbody>
</table>
## Bit-wise Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>not</td>
<td>~1 → all bits 1 except last bit 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>&amp;</td>
<td>and</td>
<td>1 &amp; 3 → 1</td>
</tr>
<tr>
<td>^</td>
<td>exclusive or</td>
<td>1 ^ 3 → 2</td>
</tr>
</tbody>
</table>
# Logical Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&amp;&amp;</code></td>
<td>and</td>
<td><code>0 and 1 → 0</code></td>
</tr>
<tr>
<td>`</td>
<td></td>
<td>`</td>
</tr>
<tr>
<td><code>!</code></td>
<td>not</td>
<td><code>!4 → 0, !0 → 1</code></td>
</tr>
<tr>
<td><code>xor</code></td>
<td>exclusive or</td>
<td><code>0 xor 1 → 1, 1 xor 1 → 0</code></td>
</tr>
</tbody>
</table>
# String Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>catenation</td>
<td>'abc' . 'def' → 'abcdef'</td>
</tr>
<tr>
<td>x</td>
<td>replication</td>
<td>'abc' x 4 → 'abcabcabcabc'</td>
</tr>
<tr>
<td>lt</td>
<td>less than</td>
<td>'abc' lt 'def' → 1</td>
</tr>
<tr>
<td>le</td>
<td>less than or equal</td>
<td>'abc' le 'def' → 1</td>
</tr>
<tr>
<td>eq</td>
<td>equals</td>
<td>'abc' eq 'def' → 0</td>
</tr>
<tr>
<td>ne</td>
<td>not equals</td>
<td>'abc' ne 'def' → 1</td>
</tr>
<tr>
<td>ge</td>
<td>greater than or equal</td>
<td>'abc' ge 'def' → 0</td>
</tr>
<tr>
<td>gt</td>
<td>greater than</td>
<td>'abc' gt 'def' → 0</td>
</tr>
<tr>
<td>cmp</td>
<td>compare</td>
<td>$a$ cmp $b$ returns</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-1 if $a$ lt $b$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 if $a$ eq $b$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 if $a$ gt $b$</td>
</tr>
</tbody>
</table>

$\text{cmp} \text{ returns } -1 \text{ if } a < b$
$0 \text{ if } a = b$
$1 \text{ if } a > b$
If

if (expression) {
    statement1;
    statement2;
}

Beginning Perl – slide 23
If

if (expression) {
    statement1;
    statement2;
}

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

if (expression) {
    statement1;
    statement2;
}

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

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

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

if (expression) {
    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.

```perl
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.

```perl
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`).

```perl
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.

```perl
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).

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

Type perldoc perlsyn or see a Perl book for more information.
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
";
}

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

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

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

```perl
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/;
}
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, ...)
```
Perl has *lots* of functions. Type `perldoc perlfuns` or see a Perl book for more information.

Functions can be called using

\[
\text{function\_name}(\text{argument1}, \text{argument2}, \ldots)
\]

or

\[
\text{function\_name\ argument1, argument2, \ldots}
\]

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

Functions can be called using

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

or

```perl
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.
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
Adding 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 perlr` 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 ə 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

<table>
<thead>
<tr>
<th>Use This</th>
<th>To Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>any character except newline (\n)</td>
</tr>
<tr>
<td>\d</td>
<td>a digit ([0-9])</td>
</tr>
<tr>
<td>\D</td>
<td>a non-digit ([^0-9])</td>
</tr>
<tr>
<td>\f</td>
<td>form feed (Control-L, ASCII decimal 12)</td>
</tr>
<tr>
<td>\n</td>
<td>newline (Control-J, ASCII decimal 10)</td>
</tr>
<tr>
<td>\r</td>
<td>carriage return (Control-M, ASCII decimal 13)</td>
</tr>
<tr>
<td>\s</td>
<td>a space character ([ \r\t\n\f])</td>
</tr>
<tr>
<td>\S</td>
<td>a non-space character ([^\r\t\n\f])</td>
</tr>
<tr>
<td>\t</td>
<td>tab (Control-I, ASCII decimal 9)</td>
</tr>
<tr>
<td>\w</td>
<td>a word character ([a-zA-Z0-9])</td>
</tr>
<tr>
<td>\W</td>
<td>a non-word character ([^a-zA-Z0-9])</td>
</tr>
<tr>
<td>\</td>
<td>it</td>
</tr>
</tbody>
</table>

*most* anything else
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:

<table>
<thead>
<tr>
<th>Repetition Count</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>0 or 1 times</td>
</tr>
<tr>
<td>*</td>
<td>0 or more times</td>
</tr>
<tr>
<td>+</td>
<td>1 or more times</td>
</tr>
<tr>
<td>{m, n}</td>
<td>at least (m) but no more than (n) times</td>
</tr>
<tr>
<td>{m, }</td>
<td>at least (m) times</td>
</tr>
<tr>
<td>{\ , n}</td>
<td>no more than (n) times</td>
</tr>
</tbody>
</table>

**Beginning and Ending of Lines** \(^\wedge\) 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.

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

# Same as previous example.
```perl
/ab/ and print qq/"ab" is in "$_"\n/;
```
if ($str =~ /yz$/) {
    print qq/"yz" at end of line in "$str".
/;
}

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

# Set $match to 1 if second character
# on line of $str is "X", 0 otherwise.
$match = ($str =~ /^.X/);
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:
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
```
Here are some examples of the substitution command:

<table>
<thead>
<tr>
<th>Command</th>
<th>$_ before</th>
<th>$_ after</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>s/abc/def/;</td>
<td>xyz</td>
<td>xyz</td>
<td>no change</td>
</tr>
<tr>
<td>s/abc/def/;</td>
<td>abc</td>
<td>def</td>
<td>found it</td>
</tr>
<tr>
<td>s/abc/def/;</td>
<td>abcabc</td>
<td>defabc</td>
<td>only first done</td>
</tr>
<tr>
<td>s/abc/def/g;</td>
<td>abcabc</td>
<td>defdef</td>
<td>/g does it globally</td>
</tr>
<tr>
<td>s/a*bc/bc/;</td>
<td>abcabc</td>
<td>bcabc</td>
<td>repeat count after a</td>
</tr>
<tr>
<td>s/a*bc/bc/g;</td>
<td>abcabc</td>
<td>bcbc</td>
<td>/g does it globally</td>
</tr>
<tr>
<td>s/^a/A/;</td>
<td>abcabc</td>
<td>Abcabc</td>
<td>^ matches beginning of line</td>
</tr>
<tr>
<td>s/c$/C/;</td>
<td>abcabc</td>
<td>abcabC</td>
<td>$ matches end of line</td>
</tr>
<tr>
<td>s/[ac]/x/;</td>
<td>abcabc</td>
<td>xbcabc</td>
<td>character class for a or c</td>
</tr>
<tr>
<td>s/[ac]/x/g;</td>
<td>abcabc</td>
<td>xbxxbx</td>
<td>/g does it globally</td>
</tr>
<tr>
<td>s/[a-c]/x/g;</td>
<td>abcabc</td>
<td>xxxxxxx</td>
<td>character class for a–c</td>
</tr>
<tr>
<td>s/a///;</td>
<td>abcabc</td>
<td>bcabc</td>
<td>substituting with nothing</td>
</tr>
<tr>
<td>s/.////;</td>
<td>abcabc</td>
<td>bcabc</td>
<td>. matches anything</td>
</tr>
</tbody>
</table>
## Substitution Operator (3)

<table>
<thead>
<tr>
<th>Command</th>
<th>$_ \text{ before}</th>
<th>$_ \text{ after}</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>s/^.(.*).$/\1/;</code></td>
<td><code>abcabc</code></td>
<td><code>bcab</code></td>
<td>delete first and last characters of line</td>
</tr>
<tr>
<td><code>s/\d*$/;</code></td>
<td><code>abc123</code></td>
<td><code>abc</code></td>
<td>delete trailing digits</td>
</tr>
<tr>
<td><code>s#/home/##;</code></td>
<td><code>/home/abc</code></td>
<td><code>abc</code></td>
<td>don’t use have to use /</td>
</tr>
</tbody>
</table>
## Substitution Operator (3)

<table>
<thead>
<tr>
<th>Command</th>
<th>$._$ before</th>
<th>$._$ after</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>s/^.<em>(.</em>).$/\1/;</td>
<td>abcabc</td>
<td>bcab</td>
<td>delete first and last characters of line</td>
</tr>
<tr>
<td>s/\d*$/;</td>
<td>abc123</td>
<td>abc</td>
<td>delete trailing digits</td>
</tr>
<tr>
<td>s#/home/###;</td>
<td>/home/abc</td>
<td>abc</td>
<td>don’t use have to use /</td>
</tr>
</tbody>
</table>

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 not preceded by $, @, or %.

Type perldoc perlfunc or see a Perl book for more information.
# 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/;
# The loop for a program that only reads the input
# file until an "END" line and only checks every 10th
# line for something 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";
}
# 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/;