Introduction to Perl

- Perl (uppercase)
- High-level language - very high level
- Glue language - excellent for uniting different systems
- Scripting language - easy to write functional code and CGI programs
- Text-processing language - powerful regular expressions
- Object-oriented language - OOP is simple but not required
- Open-source and free language - supported by a helpful international community

Introduction to Perl

- perl (lowercase)
- The language compiler/interpreter program
- Compiles and interprets source code in single step
- Accepts many useful command-line arguments for simple "one-line" scripts

```
% perl -ne 'print' filename
% perl -ne 'print if /match/' filename
% perl -pi -e 's/19100/2000/g' files
% perldoc perlrun
```

Perl History

- History
- Perl has evolved over time to meet new needs
- First version (1) released in 1987
- Modern version (5) released in 1994
- Current version (5.26.0) released in 2017
- Perl 6 is available, but not widely used.
- Some of the language has been changed and extended

Perl forms

- Obfuscated:
  ($_=q|cevag"znkvzhz:";@_=(2..<>);juvyr($a=fuvsg@_){cevag"
  $a ";@_=terc{$_%$a}@_;}|)=~tr|a-z|n-za-m|;eval"$_";
- For compactness, or obscurity, or fun.
- Don't be afraid; most Perl is not written this way.

Perl forms

- In more readable Perl:

```
 # Accept a number from the user
        print "maximum:";
        $maximum = <STDIN>;

        # Make an array of numbers
        @numbers = (2..$maximum);

        # Iterate through that array
        while ($prime = shift @numbers) {
            # Print the next prime
            print "$prime\n";
            # Remove multiples of that prime
            @numbers = grep {$_ % $prime}
@numbers;
         }
```

The Perl way

- Perl's motto is:
  - There's more than one way to do it.
- Why use Perl?
  - Well-suited to many tasks
  - Fast development time
  - Enjoyment

Perl overview

- Perl code is written in plaintext
- Use your favorite text editor (emacs, vi, etc.) to enter code
- emacs has a perl mode (M-x perl-mode) to assist with formatting
- Running Perl programs
- Programs may be run by the perl interpreter
- Programs may be made executable

Perl overview

- Interpreter line
- Also called the "shebang" line
- Similar to shell scripts and other interpreted languages
- Specifies the interpreter to use for a program
- /usr/bin/perl is the canonical location

Perl pragmas

- Pragma
- Compiler directives that affect program compilation
- Beginners should consider the following mandatory!
- use strict
- Restricts unsafe constructs, requires declaration of variables, helps prevent common errors, feels more formal and less casual.
- use warnings / use diagnostics
- Provides helpful diagnostics. Older perl versions used a -w flag on the interpreter line.

General perl

- Whitespace is ignored, so format for readability
- Comments begin with a # character
- All statements end with semicolons
- Statements may be combined into blocks with curly braces ({})

```
#!/usr/bin/perl
        # The Perl version of "Hello, world."

        use strict;
        use warnings;

        print "Just another Perl hacker.\n";
```

Perl Data Types

- Scalars - single values
- Arrays - multiple ordered values
- Hashes - mulitple unordered values

Perl Scalars

- Store any single value
- Have a name preceded by a $ character
- May be declared in a local scope with a my qualifier
- Dynamically assume whatever value is assigned

Perl Scalars

- Store any single value
  - number
    - integer (12, 1E+100)
    - real (3.14159, 2.71828182845905)
    - decimal (15), octal (017), or hexadecimal (0xF)
  - string
    - a single character ("a")
    - many characters ("A quick brown...")
  - Unicode ("\x{263A}", UTF-8 format)
  - reference

Perl Scalars

- Have a name preceded by a $ character
- Up to 251 alphabetic, numeric, and underscore characters
- Must not begin with a digit
- Case-sensitive
- Some special names are reserved ($_, $1, $/)
- Good style recommends descriptive words separated by underscores for readability

Perl Scalars

- May be declared in a local scope with a my qualifier
- Required under use strict
- Helps avoid common mistakes
- Limits the variable to its enclosing block
- Declaring a variable without assigning a value leaves its value undefined
- May declare a variable and assign a value in the same statement

```
use strict;

        my $apple_variety;                  # Value undefined
        $apple_variety = "Red Delicious"; # Defined
        $apple_vareity = "Granny Smith";  # Error

        my $apple_color = "red";            # Declare and define
```

Perl Scalars

- Dynamically assume whatever value is assigned
- Scalar is a basic type in Perl
- Any scalar value can be freely assigned to any scalar variable
- No need to "cast" or redeclare values

```
my $quantity = 6;          # Declare & define
$quantity = "half dozen"; # Now a string
$quantity = 0.5 * 12;     # Numeric again
```

Perl Arrays

- Store any number of ordered scalars
- Have a name preceded by an @ character
- May be declared in a local scope with a my qualifier
- Indexed by number
- May be assigned in several ways
- Dynamically assume whatever values or size needed
- May be sliced
- Easy to iterate

Perl Arrays

- Store any number of ordered scalars
- numbers
- strings
- references
- any combination of above
- Have a name preceded by an @ character
- May be declared in a local scope with a my qualifier

```
my @fibonacci = (1, 1, 2, 3, 5, 8, 11);        # Numbers
my @fruits = ("apples", "bananas", "cherries"); # Strings
my @grade_synonyms = (100, "A++", "Perfect");   # Both
```

Perl Arrays

- Indexed by number
  - An index in square brackets refers to an array item
  - Each item is a scalar, so use scalar syntax $array[$index]
  - By default, first item is index 0
  - Last item is index $#array
  - Negative numbers count from end of list
  - Can directly assign an array item

```
my @fruits = ("apples", "bananas", "cherries");

    print "Fruit flies like $fruits[1].\n";
    print "Life is like a bowl of $fruits[$#fruits].\n";
    print "We need more $fruits[-3] to make the pie.\n";

    $fruits[0] = "oranges"; # Replace apples with oranges
```

Perl Arrays

- May be assigned in several ways
  - Items bounded by parentheses and separated by commas
  - Numeric value ranges denoted by .. operator
  - Quoted word lists using qw operator
  - sublists are "flattened" into a single array

```
@prime_numbers  = (2, 3, 5, 7, 11, 13);        # Comma-separated
 @composite_numbers  = (4, 6, 8..10, 12, 14..16); # Numeric ranges

 @fruits = ("apples", "bananas",  "cherries");
 @fruits = qw(apples bananas cherries);         # Same as above
 @veggies = qw(radishes spinach);
 @grocery_list  = (@fruits, @veggies, "milk");
  print "@grocery_list\n";
```

Perl Arrays

- May be sliced
  - A slice (or sub-array) is itself an array
  - Take an array slice with @array[@indices]
  - $array[0] is a scalar, that is, a single value
  - @array[0] is an array, containing a single scalar
  - Scalars always begin with $
  - Arrays always begin with @

```
my @fruits = qw(apples bananas cherries oranges);

    my @yummy = @fruits[1,3];
    print "My favorite fruits are: @yummy\n";

    my @berries = @fruits[2];
    push @berries, "cranberries";
    print "These fruits are berries: @berries\n";
```

Perl Arrays

- Dynamically assume whatever values or size needed
  - Can dynamically lengthen or shorten arrays
  - May be defined, but empty
  - No predefined size or "out of bounds" error
  - unshift and shift add to and remove from the front
  - push and pop add to and remove from the end

```
my @fruits;                          # Undefined
@fruits = qw(apples bananas cherries); # Assigned
@fruits = (@fruits, "dates");        # Lengthen
@fruits = ();                        # Empty
unshift @fruits, "acorn";            # Add an item to the front
my $nut = shift @fruits;             # Remove from the front
print "Well, a squirrel would think an $nut was a fruit.\n";
push @fruits, "mango";               # Add an item to the end
my $food = pop @fruits;              # Remove from the end
print "My, that was a yummy $food!\n";
```

Perl Arrays

- Easy to iterate
- foreach loop iterates over entire array
- Good to localize the scalar to the loop

```
my @fruits = qw(apple orange grape cranberry);

    foreach my $fruit (@fruits) {
            print "We have $fruit juice in the refrigerator.\n";
    }
```

Perl Hashes

- Store any number of unordered scalars organized in key/value pairs
- Have a name preceded by an % character
- May be declared in a local scope with a my qualifier
- Indexed by key name
- May be assigned in several ways
- Dynamically assume whatever values or size needed
- May be sliced
- Easy to iterate

Perl Hashes

- Store any number of unordered scalars
- numbers
- strings
- references
- any combination of above
- Organized in key/value pairs
- Have a name preceded by an % character
- May be declared in a local scope with a my qualifier

```
my %wheels = (unicycle => 1,
                        bike      => 2,
                        tricycle => 3,
                        car       => 4,
                        semi      => 18);
```

Perl Hashes

- Indexed by key name
- An index in curly braces refers to a hash item
- Each item is a scalar, so use scalar syntax $hash{$key}
- Items are stored in random order
- Looking up items is faster than searching through an array
- Can directly assign a hash item

```
print "A bicycle has $wheels{bike} wheels.\n";
$wheels{bike} = 4; # Adds training wheels
print "A bicycle with training wheels has $wheels{bike}
wheels.\n";
```

Perl Hashes

- May be assigned in several ways
- Items bounded by parentheses and separated by commas
- Special => operator quotes the key
- Sublists are "flattened" into a single hash

```
my %dessert = ("pie", "apple", "cake", "carrot", "sorbet",
"orange");
%dessert = (pie    => "apple",
            cake   => "carrot",
            sorbet => "orange"); # Same, but easier to read

my %ice_cream = (bowl => "chocolate", float => "root beer");
my %choices = (%dessert, %ice_cream);
print "I would like $choices{sorbet} sorbet.\n";
```

Perl Hashes

- Dynamically assume whatever values or size needed
- Can dynamically lengthen or shorten hashes
- May be defined, but empty
- No predefined size or "out of bounds" error
- New keys are auto-instantiated when used
- Old keys may be deleted

```
my %wheels = (unicycle => 1,
              bike     => 2,
              tricycle => 3);

$wheels{car} = 4; # Creates a new key/value pair
$wheels{van} = 4; # Creates another new key/value pair
delete $wheels{unicycle};
```

Perl Hashes

- May be sliced
- A slice is an array of the hash's values
- Take a hash slice with @hash{@keys}
- $hash{cow} is a scalar, that is, a single value
- @hash{cow} is an array, containing a single scalar
- Hashes always begin with %

```
 my %sounds = (cow   => "moooo",
               duck  => "quack",
               horse => "whinny",
               sheep => "baa",
               hen   => "cluck",
               pig   => "oink");

my @barnyard_sounds = @sounds{"horse", "hen", "pig"};
print "I heard the following in the barnyard:
@barnyard_sounds\n";
```

Perl Hashes

- Easy to iterate
- keys returns a list of the keys
- values returns a list of the values
- each returns key/value pairs in random order
- while loop can iterate over entire hash

```
my @animals = keys %sounds;
my @noises = values %sounds;

while (my ($animal, $noise) = each %sounds) {
    print "Old MacDonald had a $animal.";
    print "  With a $noise! $noise! here...\n";
}
```

Perl Contexts

- Context determines how variables and values are evaluated
- In Perl, is "1" a number, a string, or a boolean value?
- In English, is "fly" a noun, a verb, or an adjective?
- A variable or value can have different meanings in different contexts
- Context is more important than "type"
- Several types of scalar context
- numeric
- string
- boolean
- Scalar vs. list context
- Scalar context require a single value
- List context allows multiple values

Numeric Context

- Numeric operations apply a numeric context
- Undefined values are treated as zero
- Scalars are evaluated as numbers
- Strings are converted to their numeric value automatically
- Non-number strings are converted to zero
- Arrays are evaluated as their length

```
my $number;
my $string;                    # Zero in numeric context
$number = $string + 17;
print "Number is $number.\n";

$string = "5.2";               # 5.2 in numeric context
$number = $string + 17;
print "Number is $number.\n";

$string = "five";              # Zero in numeric context
$number = $string + 17;
print "Number is $number.\n";
```

String Contexts

- String operations apply a string context
- Period (.) performs string concatenation
- Double quotes ("") interpolate variable in string context
- Undefined values are treated as empty strings
- Scalars are evaluated as strings
- Numbers are converted to their string value automatically
- Special formatting can be done with sprintf
- Arrays in double quotes are evaluated as strings containing their items separated by spaces

String Context

```
my $string;
my $number;                    # Empty string in string context
$string = $number . 17;          # Concatenate
print "String is '$string'\n";

$number = 5.2;                 # "5.2" in string context
$string = $number . 17;
print "String is '$string'\n";

$number = 5.2;
$string = sprintf("%.2f", $number); # Formatting
print "String is '$string'\n";
```

Boolean Context

- Boolean operations apply a boolean context
- Undefined values are treated as false
- Scalars are evaluated as booleans
- Numbers are evaluated as true if non-zero
- Strings are evaluated as true if non-empty
- Lists are evaluated as booleans
- Lists, arrays, and hashes are evaluated as true if non-empty

```
my $string;
        if ($string) {
            print "A: Hello, $string.\n";
        }

        $string = "world";
        if ($string) {
            print "B: Hello, $string.\n";
        }
```

List Context

- Lists
  - In scalar context, returns the last item of the list
  - In list context, returns the entire list
- Arrays
  - In scalar context, returns the length of the array
  - In list context, returns the entire array
- Hashes
  - In scalar context, returns a measure of the space used
  - In list context, returns the entire hash

List Context

```
$last_item = qw(goldfish cat dog); # $last_item = "dog";
@pets = qw(goldfish cat dog); # entire list

$count = @pets;             # $count = 3;
@new_pets = @pets;          # entire array

%pets = (goldfish => "glub",
         cat => "meow",
         dog => "woof");
$boolean = %pets;           # true
@mix = %pets;               # ("goldfish", ..., "woof")
%new_pets = %pets;          # entire hash
```

Perl operators

- Perl has many types of operators
  - Numeric
  - String
  - Quoting
  - Boolean
  - List
- Operators provide context
- perldoc perlop gives complete descriptions

Numeric operators

- Numeric operators provide numeric context
- All common operators are provided
- Increment and decrement (++, --)
- Arithmetic (+, *)
- Assignment (+=, *=)
- Bitwise (<<, >>)

```
my $i = 17;
$i = ($i + 3) * 2;  # Parentheses for order of operation
$i++;               # $i = $i + 1;
$i *= 3;            # $i = $i * 3;
print "$i\n";
```

String operators

- String operators provide string context
- Common operators are provided
- Concatenation (.)
- Repetition (x)
- Assignment (.=, x=)

```
$bark = "Woof!";
$bark .= " ";        # Append a space
$bark x= 5;          # Repeat 5 times
print "The dog barked: $bark\n";
```

Quoting operators

- Quoting operators construct strings
- Many types of quoting are provided
  - Literal, Interpolating (", qq)
  - Literal, Non-interpolating (', q)
  - Command execution (`, qx)
  - Word list (qw)
  - Pattern matching
- Choose your own delimiters (qq(), qq{}, qq**)

```
$my $cat = "meow";
        my $sound = "$cat";                 # $sound = "meow"
        my $variable = '$cat';              # $variable =
"\$cat"
        print "$variable says $sound\n";

        $sound = qq{"meow"};                # If you want to
quote quotes
        $sound = qq("meow");                # Same
        print "$variable says $sound\n";
```

Quoting operators

```
my $cat = "meow";
my $sound = "$cat";                 # $sound = "meow"
my $variable = '$cat';              # $variable = "\$cat"
print "$variable says $sound\n";

$sound = qq{"meow"};                # If you want to quote
quotes
$sound = qq("meow");                # Same
print "$variable says $sound\n";

$contents = `cat $sound`;           # contents of file "meow"
```

Boolean operators

- Boolean operators provide boolean context
- Many types of operators are provided
- Relational (<, >, lt, gt)
- Equality (==, !=, eq, ne)
- Logical (high precedence) (&&, ||, !)
- Logical (low precedence) (and, or, not)
- Conditional (?:)

```
my ($x, $y) = (12, 100);
my $smaller = $x < $y ? $x : $y;
print "The smaller number is $smaller.\n";
```

Boolean operators

- Separate operators for numeric vs. string comparison
- Numeric (<=, >=)
- String (le, ge)

```
my ($a, $b) = ("apple", "orange");
print "1: apples are oranges\n" if ($a eq $b);  # False
print "2: apples are oranges\n" if ($a == $b);  # True!

my ($x, $y) = (12, 100);
print "3: $x is more than $y\n" if ($x gt $y);  # True!
print "4: $x is more than $y\n" if ($x > $y);   # False
```

List operators

- List operators provide list context
- Many useful operations or functions are built-in
- sort
- reverse
- push/pop
- unshift/shift
- split/join
- grep
- map

List operators

- sort
  - Sorts the list, alphabetically by default
  - Many other sorting methods available
  - perldoc -f sort gives function details
- reverse
  - In scalar context, concatenates the list and reverses the resulting string
  - In list context, reverses the list

```
my @animals = qw(dog cat fish parrot hamster);
my @sorted = reverse sort @animals;
print "I have the following pets: @sorted\n";
my $word = "backwards";
my $mirror = reverse $word;
print qq("$word" reversed is "$mirror"\n);
%by_address = reverse %by_name;        # Beware of lost duplicate
values
```

List operators

- split
  - Splits a string into a list of substrings
  - Removes a delimiting string or regular expression match
- join
  - Joins a list of substrings into a single string
  - Adds a delimiting string

```
my @animals = qw(dog cat fish parrot hamster);
my $string = join(" and a ", @animals);
print "I have a $string.\n";

my $sentence = "The quick brown fox...";
my @words = split(" ", $sentence);
```

List operators

- grep
  - Similar to the Unix command grep
  - Finds matching items in the list
  - Matches usually based on a regular expression or a comparison

```
my @juices = qw(apple cranapple orange grape apple-cider);
        my @apple = grep(/apple/, @juices);
        print "These juices contain apple: @apple\n";

        my @primes = (2, 3, 5, 7, 11, 13, 17, 19);
        my @small = grep {$_ < 10} @primes; # $_ is each
element of @primes
        print "The primes smaller than 10 are: @small\n";
```

List operators

- map
  - Maps an input list to an output list
  - Powerful, but mapping can be complex
  - $_ is the "fill in the blank" scalar
  - grep is a special case of map

```
my @primes = (2, 3, 5, 7, 11, 13, 17, 19);
my @doubles = map {$_ * 2} @primes;
print "The doubles of the primes are: @doubles\n";

my @small = map {$_ < 10 ? $_ : ()} @primes;  # grep {$_ <
10} @primes
print "The primes smaller than 10 are: @small\n";
```

Flow Control

- Flow control is more expressive in Perl
- Conditional statements
- if
- unless
- Loop statements
- while
- until
- for
- foreach
- Modifiers
- Simple statements may be modified
- See perldoc perlsyn for complete syntax

Conditional Statements

- Conditional statements provide boolean context
- if statement controls the following block
- if, elsif, else
- Yes, that does say elsif. Oh well.
- unless is opposite of if
- Equivalent to if (not $boolean)
- unless, elsif, else
- There is no elsunless, thankfully.

```
my ($a, $b) = (0, 1);
if (!$a && !$b)          {print "Neither\n";} # Conventional
if (not $a and not $b)   {print "Neither\n";} # Same, but in English
if (not ($a or $b))      {print "Neither\n";} # Same, but parentheses
unless ($a or $b)        {print "Neither\n";} # Same, but clearest
```

Loop Statements

- Loop statements provide a boolean context
- while
  - Loops while boolean is true
- until
  - Loops until boolean is true
  - Opposite of while
- do
  - At least one loop, then depends on while or until

```
while ($hungry) {
    $hungry = eat($banana);
}

do {
    $answer = get_answer();
    $correct = check($answer);
} until ($correct);
```

Loop Statements

- for
- Like C: for (initialization; condition; increment)
- foreach
- Iterates over a list or array
- Good to localize loop variables with my

```
for (my $i = 10; $i >= 0; $i--) {
    print "$i...\n";                  # Countdown
}

foreach my $i (reverse 0..10) {
    print "$i...\n";                  # Same
}
%hash = (dog => "lazy", fox => "quick");
foreach my $key (keys %hash) {
    print "The $key is $hash{$key}.\n";  # Print out hash pairs
}
```

Modifiers

- Simple statements can take single modifiers
- Places emphasis on the statement, not the control
- Can make programs more legible
- Parentheses usually not needed
- Good for setting default values
- Valid modifiers are if, unless, while, until, foreach

```
$a = $default unless defined $a;
$b = $default unless defined $b;
$c = $default unless defined $c;

$balance += $deposit if $deposit;
$balance -= $withdrawal
if $withdrawal and $withdrawal <= $balance;
```

Subroutines

- Subs group related statements into a single task
- Perl allows both declared and anonymous subs
- Perl allows various ways of handling arguments
- Perl allows various ways of calling subs
- perldoc perlsub gives complete description

Declaring a subroutine

- Subroutines are declared with the sub keyword
- Subroutines return values
- Explicitly with the return command
- Implicitly as the value of the last executed statement
- Return values can be a scalar or a flat list
- wantarray describes what context was used
- Unused values are just lost

```
sub ten {
    return wantarray() ? (1 .. 10) : 10;
}

@ten = ten();           # (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
$ten = ten();           # 10
($ten) = ten();         # (1)
($one, $two) = ten();   # (1, 2)
```

Passing values

- Two common means of passing arguments to subs
  - Pass by value
  - Pass by reference
  - Perl allows either
- Arguments are passed into the @_ array
  - @_ is the "fill in the blanks" array
  - Usually should copy @_ into local variables

Passing values

```
sub add_one {              # Like pass by value
    my ($n) = @_;          # Copy first argument
    return ($n + 1);       # Return 1 more than argument
}

sub plus_plus {            # Like pass by reference
    $_[0] = $_[0] + 1;     # Modify first argument
}

my ($a, $b) = (10, 0);

add_one($a);               # Return value is lost, nothing changes
$b = add_one($a);          # $a is 10, $b is 11
plus_plus($a);             # Return value lost, but a now is 11
$b = plus_plus($a);        # $a and $b are 12
```

Calling a subroutine

- Subroutine calls usually have arguments in parentheses
- Parentheses are not needed if sub is declared first
- But using parentheses is often good style
- Subroutine calls may be recursive
- Subroutines are another data type
- Name may be preceded by an & character
- & is not needed when calling subs

```
print factorial(5) . "\n";    # Parentheses required
sub factorial {
    my ($n) = @_;
    return $n if $n <= 2;
    $n * factorial($n - 1);
}

print ((factorial 5) . "\n");
# Parentheses around argument not required,
# but need to ensure there are no extra arguments
print &factorial(5) . "\n";  # Neither () nor & required
```

Subroutine example

- A more typical example
- Declare subroutine
- Copy arguments
- Check arguments
- Perform computation
- Return results

```
sub fibonacci {
    my ($n) = @_;
    die "Number must be positive" if $n <= 0;
    return 1 if $n <= 2;
    return (fibonacci($n-1) + fibonacci($n-2));
}

foreach my $i (1..5) {
    my $fib = fibonacci($i);
    print "fibonacci($i) is $fib\n";
}
```

File I/O

- Access to files is similar to shell redirection
- Standard files
- Reading from files
- Writing to files
- Pipes
- File checks

File I/O

- Access to files is similar to shell redirection
- open allows access to the file
- Redirect characters (<, >) define access type
- Can read, write, append, read & write, etc.
- Filehandle refers to opened file
- close stops access to the file
- $! contains IO error messages
- perldoc perlopentut has complete description

```
open INPUT,  "<  datafile" or die "Can't open input file:
$!";
open OUTPUT, ">  outfile " or die "Can't open output file:
$!";
open LOG,    ">> logfile " or die "Can't open log file: $!";
open RWFILE, "+< myfile  " or die "Can't open file: $!";
close INPUT;
```

File I/O

- Standard files are opened automatically
- STDIN is standard input
- STDOUT is standard output
- STDERR is standard error output
- Can re-open these for special handling
- print uses standard output by default
- die and warn use standard error by default

```
print STDOUT "Hello, world.\n";  # STDOUT not needed
open STDERR, ">> logfile" or die "Can't redirect errors to
log: $!";
print STDERR "Oh no, here's an error message.\n";
warn "Oh no, here's another error message.\n";
close STDERR;
```

File I/O

- Reading from files
- Input operator <> reads one line from the file, including the newline character
- chomp will remove newline if you want
- Can modify input recorder separator $/ to read characters, words, paragraphs, records, etc.

```
print "What type of pet do you have? ";
my $pet = <STDIN>;          # Read a line from STDIN
chomp $pet;                 # Remove newline

print "Enter your pet's name: ";
my $name = <>;              # STDIN is optional
chomp $name;
print "Your pet $pet is named $name.\n";
```

File I/O

- Reading from files
- Easy to loop over entire file
- Loops will assign to $_ by default
- Be sure that the file is open for reading first

```
open CUSTOMERS, "< mailing_list" or die "Can't open input
file: $!";
while (my $line = <CUSTOMERS>) {
    my @fields = split(":", $line);       # Fields separated
by colons
    print "$fields[1] $fields[0]\n";      # Display selected
fields
    print "$fields[3], $fields[4]\n";
    print "$fields[5], $fields[6]  $fields[7]\n";
}
print while <>;                                  # cat
print STDOUT $_ while ($_ = <STDIN>);  # same, but more
verbose
```

File I/O

- Writing to files
- print writes to a file
- print writes to a STDOUT by default
- Be sure that the file is open for writing first

```
open CUSTOMERS, "< mailing_list" or die "Can't open input
file: $!";
open LABELS,"> labels" or die "Can't open output file: $!";
while (my $line = <CUSTOMERS>) {
  my @fields = split(":", $line);
  print LABELS "$fields[1] $fields[0]\n";
  print LABELS "$fields[3], $fields[4]\n";
  print LABELS "$fields[5], $fields[6]  $fields[7]\n";
}
```

File Tests

- File checks
- File test operators check if a file exists, is readable or writable, etc.
- -e tests if file is exists
- -r tests if file is readable
- -w tests if file is writable
- -x tests if file is executable
- -l tests if file is a symlink
- -T tests if file is a text file
- perldoc perlfunc lists more

```
my $filename = "pie_recipe";
if (-r $filename) {
    open INPUT, "> $filename" or die "Can't open $filename:
$!";
} else {
    print "The file $filename is not readable.\n";
}
```

Regular Expressions

- Regexes perform textual pattern matching
- Regexes ask:
- Does a string...
- ...contain the letters "dog" in order?
- ...not contain the letter "z"?
- ...begin with the letters "Y" or "y"?
- ...end with a question mark?
- ...contain only letters?
- ...contain only digits?

```
my $string = "Did the fox jump over the dogs?";
print "1: $string\n" if $string =~ /dog/; # matches
print "2: $string\n" if $string !~ /z/;   # matches
print "3: $string\n" if $string =~ /^[Yy]/;
print "4: $string\n" if $string =~ /\?$/; # matches
print "5: $string\n" if $string =~ /^[a-zA-Z]*$/;
print "6: $string\n" if $string =~ /^\d*$/;
```

Regular Expressions

- Regexes may be quoted in several ways
- Regex quotes are usually slashes (/regex/)
- May use other quotes with the match operator m
- Use other quotes when matching slashes (m[/])
- Comparison operators (=~, !~) test for matches

```
my $string = "Did the fox jump over the dogs?";
print "1: $string\n" if $string =~ /dog/; # matches
print "2: $string\n" if $string =~ m/dog/;# matches
print "3: $string\n" if $string =~ m(dog);# matches
print "4: $string\n" if $string =~ m|dog|;# matches
```

Regular Expressions

- Regexes are their own mini-language
- Match letters, numbers, other characters
- Exclude certain characters from matches
- Character classes (in []) denote a possible set of characters to match
- Negate a character classes with a carat ([^abc])
- Provided character classes include:
- \d, \D for digits, non-digits
- \w, \W for word and non-word characters (letters, digits, underscore)
- \s, \S for white-space and non-space characters
- . for any character except newline

```
my $string = "Did the fox jump over the dogs?";
print "1: $string\n" if $string =~ m/[bdl]og/; # bog, dog,
log
print "2: $string\n" if $string =~ m/dog[^s]/; # no match
print "3: $string\n" if $string =~ m/\s\w\w\wp\s/; # matches
```

Regular Expressions

- Regexes are their own mini-language
  - Match letters, numbers, other characters
  - Exclude certain chaSracters from matches
  - Character classes (in []) denote a possible set of characters to match
  - Negate a character classes with a carat ([^abc])
  - Provided character classes include:
    - \d, \D for digits, non-digits
    - \w, \W for word and non-word characters (letters, digits, underscore)
    - \s, \S for white-space and non-space characters
    - . for any character except newline

```
my $string = "Did the fox jump over the dogs?";
print "1: $string\n" if $string =~ m/[bdl]og/; # bog, dog,
log
print "2: $string\n" if $string =~ m/dog[^s]/; # no match
print "3: $string\n" if $string =~ m/\s\w\w\wp\s/; # matches
```

Regular Expressions

- Match boundaries between types of characters
- ^ matches the start of the string
- $ matches the end of the string
- \b matches a word boundary

```
my $string = "Did the fox jump over the dogs?";
print "1: $string\n" if $string =~ m/^[Yy]/;  # no match
print "2: $string\n" if $string =~ m/\?$/;    # match
print "3: $string\n" if $string =~ m/the\b/;  # match
```

Regular Expressions

- Quantify matches
  - * matches the preceding character 0 or more times
  - + matches the preceding character 1 or more times
  - ? matches the preceding character 0 or once
  - {4} matches exactly 4 times
  - {3,6} matches 3 to 6 times

```
my $string = "Did the fox jump over the dogs?";
print "1: $string\n" if $string =~ m/z*/;          # matches
print "2: $string\n" if $string =~ m/z+/;          # no match
print "3: $string\n" if $string =~ m/\b\w{4}\b/;   # matches
"jump"
```

Regular Expressions

- Quantify matches
  - * matches the preceding character 0 or more times
  - + matches the preceding character 1 or more times
  - ? matches the preceding character 0 or once
  - {4} matches exactly 4 times
  - {3,6} matches 3 to 6 times

```
my $string = "Did the fox jump over the dogs?";
print "1: $string\n" if $string =~ m/z*/;          # matches
print "2: $string\n" if $string =~ m/z+/;          # no match
print "3: $string\n" if $string =~ m/\b\w{4}\b/;   # matches
"jump"
```

Regular Expressions

- Group subpatterns
  - () group a subpattern
- Match repetitions
  - \1, \2 refer to the first and second groups

```
my $string = "Did the fox jump over the dogs?";
print "1: $string\n" if $string =~ m/(fox){2}/; # "foxfox"
print "2: $string\n" if $string =~ m/(the\s).*\1/; # matches
```

Regular Expressions

- Perl has several regex operators
  - m just matches, providing boolean context
  - s/match/replacement/ substitutes
  - tr/class/replacement/ transliterates

```
my $string = "Did the fox jump over the dogs?";
$string =~ s/dog/cat/;   # substitute "cat" for "dog"
$string =~ s(fox)(kangaroo); # substitute "kangaroo" for
"fox"
print "$string\n";

$string =~ tr/a-z/n-za-m/;          # Rot13
print "$string\n";
```

Regular Expressions

- Perl has several regex modifiers
  - g global: allows for multiple substitutions
  - i case insensitive matching
  - s treat string as one line
  - m treat string as multiple lines

```
my $breakfast = 'Lox Lox Lox lox and bagels';
$breakfast =~ s/Lox //g;
print "$breakfast\n";

my $paragraph = "First Line\nSecond Line\nThird Line\n";
my ($first, $second) = ($paragraph =~ /(^.*$)\n(^.*$)/m);
print "$first, $second\n";
```

Perl modules and CPAN

- Modules provide extra functionality
  - CGI provides a CGI scripting interface
  - Data::Dumper prints out data structures like hashes
  - DBI provides a database interface
  - Error provides simple error handling
- Using modules
  - use modules at the top of your program
  - Some modules take a parameter list
  - perldoc gives information about installed modules

```
use CGI qw(:standard);
print header,
start_html('A Simple Example'),
h1('A Simple Example'),
p("Hello, World"),
end_html;
```