Perl as a Programming Language

CMSC 331: Principles of Programming Languages
Spring 2012

Background of Perl

- "Perl" <= Practical Extraction and Report Language, created in 1987 by Larry Wall
- Hybrid model: source is compiled to intermediate form at start of each execution (not pre-compiled, like Java).
- Sits at intersection between general interpreted programming languages like Python, special-purpose text-processing languages like ***sed*** and ***awk***, and scripting languages like shell scripts.

Background of Perl

- Main (original) purpose is to apply power of programming to text manipulation/processing
- Has grown to be a general-purpose language; but, at its heart: "Perl is above all a text processing language" (Wall et al.)
- Much more powerful and flexible than stream-based editors (sed) or token-based text reprocessing (awk)
- Extremely powerful pattern-matching operators, combined with full C-like control flow primitives, and extensibility (via Perl or C)
- It fills a huge gap between C/Java/… and awk/sed/grep

Background of Perl

- Dynamic language
  - Runtime can change parts of language behavior
- Portable
  - Versions for various flavors of Unix/Linux, MacOS, Windows, etc.
- VERY popular
  - E.g.: Web servers (mod_perl), BioPerl
- Extensive set of add-on modules:
  CPAN: Comprehensive Perl Archive Network
  > 24,000 modules

Motivating Example: Parsing XML

```
<?xml version="1.0"?>
<grade-db>
  <student id="umbc10001">
    <name>Doe, John</name>
    <major>Computer Science</major>
    <title>Mr.</title>
    <dissertation>
      <type>Masters</type>
      <title>Getting an MA Doing Nothing</title>
      <advising-ta>umbc10002</advising-ta>
    </dissertation>
  </student>
  <student id="umbc10002">
    <name>Smith, Mary</name>
....
```

Motivating Example: Parsing XML

- To parse XML files (like HTML, but much more structured): Complex pattern-based text processing
  - Difficult for Java, etc.
  - Need to recognize tags:
    - Pattern-matching tag syntax ("<…>")
  - Need to parse tags:
    - Matching and partitioning tag internals
  - Need to handle special tags:
    - Functions/subroutines
  - Need to handle nested tags:
    - Recursion!
  - Process content:
    - Mostly input/output, but with special context-based pattern-matching
    - Might also need to scan for specific user-requested content

Learning Perl

- Claim is that it is easy to learn
- Bible of sorts: "Programming Perl", by Wall, Christiansen, and Schwartz ("the Camel book")
- Most primitive program is:

```
print "Howdy, world!\n";
```

(note more informal nature of even "Hello, world")

Basic Perl Syntax

- Script, can be run from shell via "#!" header:

```
#!/usr/bin/perl
print "Hello, world\n";
```

- Simplest program consists of sequence of statements (like most imperative PLs), separated by semicolons ';'
  (Note: last semicolon is optional)
- There are built-in functions/subroutines/ commands, like "`print`"

Basic Perl Syntax

- Comments are indicated by a '#' – anything following in the line is ignored
- In most(!) cases, whitespace is ignored, so you should apply it judiciously to improve readability
- In most cases, syntax is very similar to C and Java, including most operators

Primitive Data Types

- Perl has basic standard data types: (numbers and strings), plus more complex aggregates (arrays, hashtables, etc.)
- Type set is simplified: all numbers are effectively floats, all characters and strings are represented as sequences of characters.
- String literals implied by quote marks (single or double)
- Numbers implied by digits (and some special chars, e.g. "e' for exponentiation) w/o quotes (0.0 == 0)
- Boolean is implied by interpretation:
  - Strings: "0", "" or null are false, all else is true
  - Numbers: 0 is false, all else is true (beware, tho')

Scalars, Arrays and Hashes

- Perl is a "loosely typed" or "dynamically typed" language: no declarations
- So, when we talk about data types in Perl, we bundle all primitive types into one: "scalar"
- There are also "array" and "hash" types to round out the trio.
  - An "array" (or "list") is a collection of things indexed by numerical position, starting at 0
  - A "hash" is also a collection, but indexed by a string value: the "key". Underlying representation is a hash table—thus the name.

Variables

- Variables are all prefixed by one or another special symbol, so will not conflict with keywords
- Scalars: variables begin with '$', e.g.:

```
$name = "Mary";
$age = 21;
```

- Arrays: variables begin with '@', e.g.:

```
@ages = (19, 19, 20, 17);
```

- Hashes: variables begin with '%', e.g.:

```
%grades = ("Jim" => 2.5, "Mary" => 4.0,
           "Rob" => 4);
# Can also just use ',':
%profs = ("Jim", "J. Park", "Mary", "J. Park",
          "Rob", "T. Finin");
```

Accessing Array Members

- Access elements in an array with '[ ]'

```
@ages = (19, 19, 20, 17); # defines an array
$my_age = $ages[2];  # returns 20
$other_age = $ages[-1] # negative index: from end
```

- Access element in a hashtable with '{ }':

```
%grades = ("Jim" => 2.5,"Mary" => 4.0,"Rob" => 4);
$rob = $grades{"Rob"};
```

NB: Whether you use $, @, or % depends on type of value you want to access, ***not*** type of collection it is coming from! This is true of all collection types. Study examples above.

- So, in above examples, we would not do:

```
$my_age = @ages[2];
```

or:

```
$rob = %grades{"Rob"};
```

Some Useful Array Functions

- Getting the length of an array:

```
$size = @some_array;
```

  - But why does this work? See "Contexts" later.
- Getting the index of the last element of an array:

```
$last = $#some_array;
```

- Using an array as a push-down stack:

```
push(@some_array, "last thing");
$last = pop(@some_array);
```

  - Note that the second line above should set $last to the string "last thing", as well as restoring @some_array to its original state.
- Reversing an array:

```
@rev_array = reverse(@some_array);
```

Assignment & Arithmetic Operators

- Assignment operators (almost identical to C and Java):

```
$name = "Susan";
$age += 1;
$num_left -= 10;
```

- Arithmetic operators (again, much like C/Java):

```
$budget = 100 – 3 * $num_bought;
$remainder = 17 % 3;
$count++;
```

  - Even has modulus and auto-increment/decrement operators
  - Precedence and associativity rules similar to C/Java

Comparison and Logical Operators

- Standard comparison operators for numbers:
  - Numerical: ==, !=, <, >, <=, >=, e.g.:

```
if ($val < $limit) …
```

- Logical operators: two versions:
  - Symbolic (as in C/Java): &&, ||, !

```
if ($age < 21 || $status eq "4F") …
```

  - …And text versions (different from C/Java): "and", "or", "not"

```
if ($age < 21 or $status eq "4F") …
```

  - The symbolic and text logical operators have the same semantics, but are of different precedence ("and", "or", and "not" are much lower precedence)
  - Logical operators don't return boolean—return last thing evaluated:

```
$a = ($x_is_defined and $x) or $default_value;
```

String Operators

- String literals are created with single or double quotes:

```
$first_name = 'John';
$last_name = "Doe";
```

- Inside single quotes:
  - can use double quotes
  - Escapes like \n are treated literally
- Inside double quotes only:
  - Can use single quotes
  - Special escapes like \n, \t are interpreted
  - Can *interpolate* ("substitute in") variables:

```
# "$name" in literal replaced w/its value:
$output = "Hello $name, now are you";
```

  - (Will see this interpolation again later…)

String Operators

- Separate set of comparison operators for strings:
  eq, ne, lt, gt, le, ge

  E.g.:

```
if ($name lt "zzzzz") …
```

  - Most languages would use functions for these
- Special concatenation operator: '.' (dot)

```
$output = "Hello " . $name . ", how are you?";
$output .= " I am fine";  # Can even append
```

- Also have some special string functions:
  - `length($str)`: returns length of string arg
  - `chomp($var)`: removes trailing newline from end of the variable; Note that this is in-place (i.e., $var is changed)
  - `split(' ', $str)`: returns pieces of string, in a list
  - `reverse($str)`: reverses the chars in the string (also lists)

String Operators

```
$line = "hello there everyone\n";
chomp($line);  # remove trailing '\n'
@pieces = split(' ' , $line);
foreach $str (@pieces) {
    $rev = reverse($str);
    print $rev, "\n";
}
```

Will output:

```
olleh
ereht
enoyreve
```

Conditionals

- Perl's primarily conditional form is similar, but slightly different, from C/Java:

```
if ($class eq "cmsc331") {
    $prof = "Park";
}
elsif ($class eq "cmsc202") {
    $prof = "Mitchell";
}
else {
    $prof = "Unknown";
}
```

  - Note special spelling of "elsif"; some languages are even stranger: PHP has both "elseif" and "else if", with slightly different semantics!

Conditionals

- There is also another form:

```
unless ($age >= 21) { …
```

  But that form has no "unlesif" or any such

- For all Perl control structures (conditionals, loops, etc.)*:*
  - The statements part of the structure *must* be a program block, inside "{ … }"
  - You cannot just have a single statement w/o braces

Loops

- Standard "while" and "for" loops:

```
while ($count < $end) {
    # Do your work here
}

for ($i = 0; $i < 100; $i++) {
    # Again, do stuff here
}
```

- Perl uses keywords `next` and `last` in place of C/Java's "continue" and "break";
- Perl also has "named loops", and you can use this to specify which level of nested loop "next" or "last" should continue/break out of (will not describe here)

More Loops

- Two additional loop forms:

```
until ($count >= $last) {
  # Note that "until" is like "while (not (…))"
}
```

… and:

```
foreach $elem (@some_list) {
    # Like Java's "for (MyClass elem: myList)"
}
```

Input/Output

- Perl uses "filehandles", which are special "things" created by opening files:

```
open(MYFILEHANDLE, "myfile.txt");
# From here on, can do I/O operations on
# MYFILEHANDLE
```

- Perl has a default input and output filehandle already open (much like C's stdin/stdout, or Java's System.in/System.out) called "STDIN" and "STDOUT"
- You read from a filehandle by enclosing it in "<…>", and assigning that to a variable:

```
$next_line = <MYFILEHANDLE>;
$next_cmd = <STDIN>;
```

Input/Output

- For output, functions like "print" default to STDOUT, but you can change that:

```
print $next_line;      # outputs to STDOUT
                       #   (the screen)
print MYFILEHANDLE $cmd; # write $cmd into
                         #   "myfile.txt"
```

Curiosity: what do you think the following does? (Note the '@')

```
@buffer = <MYFILEHANDLE>;
```

(See "Contexts" slides later)

Regular Expressions—Perl Rules^2

("Rules^2" == because these are Perl's regexp rules, and it is also where Perl truly rules!)

- Perl has an extremely powerful set of regular expression operations
- Perl uses an augmented form of the regular expression syntax we've already learned
- It allows us to specify actions as a result of matches, in either a transformational or procedural, syntax (i.e., "turn X into Y", or "when X, do Y")—whichever is more convenient.

Regular Expressions—Perl Rules^2

- Perl supports pattern-***matching***, ***substitution***, and ***translation*** operations as primitives
- To perform regular expression-based functions, use the binding operator: **=~**
- The left-hand side (LHS) of the '=~' operator is the target of the match/substitution/translation, the RHS is the action to perform on the RHS (note: no quotes needed here).
- An example: searching for substrings inside $a:

```
if ($a =~ m/hello/)
    { print "Found \"hello\""; }
if ($a !~ m/bye/)
    { print "Didn't find 'bye'"; }
```

Regular Expressions—Perl Rules^2

- To do pattern-matching string search:
  - syntax: "m/pattern/", where *pattern* can be a regexp:

```
$status = "error: bad name or bad mode";
if ($a =~ m/err/)
    { print "There was an error."; }
```

  - Can use any other delimiter, too, including matching left/right pairs; if you use '/…/' delimiters, the 'm' is optional. E.g.'s:

```
$a =~ m<foo>;
if ($a =~ /err/) …  # This is most common form
```

Regular Expressions—Perl Rules^2

- To substitute for the matched string: syntax: "s/old/new/"

```
$status = "error: bad name or bad mode";
$status =~ s/or/and/;
# $status becomes "errand: bad name or bad mode"
# Note: second "or" unchanged
```

Add a 'g' (for "global") at end to replace all instances); e.g.:

```
$status = "error: bad name or bad mode";
$status =~ s/bad/good/g;  # 'g': global replace
# Now, $status is "error: good name or good mode"
```

Regular Expressions—Perl Rules^2

- To translate characters from one set to another: syntax: tr/abc/xyz/

```
$a = "All Bugs Crunch";
$a =~ tr/ABC/abc/;
print $a;
```

would output:

```
all bugs crunch
```

Regular Expressions—Perl Rules²

- The pattern to be matched can be a full regular expression, and may contain:
  - character classes:
    - One of a specific set, e.g.: [XYZ]
    - A range: [a-z]
    - All *but* some set, e.g. non-digits: [^0-9]
    - One of a set of predefined character classes, which you can reference with the "*x*"-style form:
      - \d – digit
      - \w – word (i.e., alphanumeric chars)
      - \s – whitespace (space, tab, newline, etc.)

Regular Expressions—Perl Rules²

- Can specify the number of occurrences:
  - Familiar "0 or more", "1 or more": *, +
    - E.g.: "/[a-z]+[0-9]*/" means "one or more letters, followed by 0 or more digits"
  - Even a range of times:
    - E.g.: /[a-zA-Z0-9]{1-8}\.[a-z]{1-3}/ means "1 to 8 alphanumeric chars, followed by a '.' (dot), followed by 1 to 3 letters. This pattern might be useful for a Windows 8.3 filename

Regular Expressions—Perl Rules²

- Some other things you can do:
  - Pick specific parts of the matched pattern, and…
    - and use it in the substitution part of the pattern
    - Access it procedurally in your code

    (Example on next slide)
  - Use variables for parts of the pattern
    - Again, part of *interpolation*
  - …

Regular Expressions—Perl Rules²

- Be careful when using matched pattern:
  - Regexps are greedy, but also match earliest instance, which takes precedence.

E.g.:

```
$str = "---XYZ:aaa:XYZXYZ:ccc:";
$str =~ /([XYZ]+:)/;
print $1; # Corresponds to first "(…)" subpattern
# Will print out just "XYZ:" due to early matching,
# even though second run is longer

$str =~ /([abc].*:);
print $1;
# Greedy: will match "aaa:XYZXYZ:ccc:"
```

Regular Expression Examples

- To match a simple HTML/XML tag:

```
$a =~ /<[\w]*>/;
```

- An attempt at being more liberal:

```
$a =~ /<.*>/;
# Will not work: for "this is a <a>link</a>.",
# will greedily match entire "<a>link</a>"
```

- Better version:

```
$a =~ /<[^>]*>/;
```

- A whitespace-bounded substring:

```
$a =~ /\s[^\s]+\s/;
```

  - Note: using predefined class "\s" here (== whitespace)

[END OF BASIC CONCEPTS]

Contexts

- Recall that $, @, and % indicate the type (scalar, array, and hash, respectively)
- Also recall that they occupy different namespaces (i.e., $foo is different from @foo and $foo[0] (which is part of @foo)
- Perl also does type-mapping based on *context* (list or scalar)

Contexts

- There are some contexts that assume lists, others that are clearly expecting scalars

```
$foo  = "Hello";  # scalar into scalar
@fum = $foo;  # list context: create 1-item list
@fie = @fum;  # list context: copy entire array
$faa = @fie;  # scalar context: copy size of @fie
Perl tries to infer context from code
```

- It is not always obvious what the context type is—just have to learn from experience ☹

Contexts

- More fun with contexts:

```
@fie = ("Hello", "bye");
$fie[2] = "adios";

# Print all elements in @fie:
print @fie

# Read in ALL the lines from a file at once
@all_lines = <FILEHANDLE>;

($l1, $l2, $l3) = @all_lines;
```

Contexts

- All variables treated as scalar in boolean context:

```
while (@argv) {
    process(shift @argv);
}
```

or:

```
if (%symbol_table) {
    # Things to do if @my_array is not empty
}
```

Namespaces

- In Perl, scalars, arrays, and hashtables all have distinct *namespaces* ("tables of variable names")
- So, $foo is a different variable from @foo, which is distinct from %foo:

```
$foo = 99;
@foo = ("x", "y", "z");
%foo = ("0" => "a", "1" => "b", "2" => "c");
print $foo;  # outputs "99"
print @foo;  # outputs "xyz"
print %foo;  # outputs "1b0a2c"; not pretty…
```

- Important to note: in above, $foo, $foo[0], and $foo{"0"} are all distinct, co-existing variables!

Using "and" and "or" w/o "if"

- A common pattern, that takes advantage of short-circuiting, is:

```
open(HANDLE, "myfile.txt") or die "bad file!";
```

- Also can do:

```
# This reads in entire file into array,
# but only if open succeeds
open(FILE, "foo.txt") and @lines = <FILE>;
```

Statement Modifiers

- Can modify statements with trailing conditionals:
  - *blahblahblah* `if (expr);`
  - *blahblahblah* `unless (expr);`
  - *blahblahblah* `while (expr);`
  - *blahblahblah* `until (expr);`
- Even though the modifier comes at the end, it is evaluated first, and with all four kinds, the statement might not be executed at all:

```
# Reattach STDIN only if filename is specified
open(STDIN, $fname) if $fname ne "";
# Then, dump all of the contents to the screen
print $line while ($line = <STDIN>);
```

  - In above, open() is not called if $fname is an empty string, and print() might never be called if we get EOF right away

Functions/Subroutines

- To define a function (a.k.a. subroutine) in Perl:

```
sub my_add    # parameters are in @_
{
    my $num1 = $_[0];
    my $num2 = $_[1];
    # Or, can use:
    ($num1, $num2) = @_;
    return $num1 + $num2;
}
```

- To call the defined function, use '&' prefix:

```
$val = &my_add(37, 10);
print $val;   # outputs "47"
```

- If no "return" statement, return value is last evaluated expression

Variable Scope

- Variables are by default global scope (any code or function can see anywhere)
- You can create a lexically scoped variable with the keyword "my":

  my $count = 1;
- You can create dynamically scoped variables with the keyword "local":

  local $special_var;

Executing External Programs

- Can run programs from the filesystem (executable binaries, scripts, etc.) and capture the output for processing:

```
# First, get listing of park's home directory:
my @ls_output = `ls -l ~park`
# Now, can post-process output here
foreach $file_desc (@ls_output) {
    …
}
```

  – Note: those are backquotes around the "ls …"!

[EXAMPLES]

Sample Perl Program

```
@DNA_T = ();  # Create an empty list
open(SEQFILE, "chromosome.db") or
      die "Couldn't open file";
while ($line = <SEQFILE>) {
    # splitting on empty pattern divides every char
    $subseq = split(//, $line);
    foreach $c (@subseq) {
        push(@DNA_T, $c);
    }
}
# @DNA_Q : from user; contains each nucleotide of query
sequence as an element
$length_Q = @DNA_Q;  # scalar context gets list length
$length_T = @DNA_T;
```

Sample Perl Program

```
@DNA_T = ();  # Create an empty list
open(SEQFILE, "chromosome.db") or
      die "Couldn't open file";
while ($line = <SEQFILE>) {
    # splitting on empty pattern divides every char
    $subseq = split(//, $line);
    foreach $c (@subseq) {
        push(@DNA_T, $c);
    }
}
# @DNA_Q : from user; contains each nucleotide of query
sequence as an element
$length_Q = @DNA_Q;  # scalar context gets list length
$length_T = @DNA_T;
```

Sample Perl Program

```
# (continued)
$num_matches = 0;
# Search for sequence match,
# and print position of match if found.
for ($i = 0; $i <= ($length_T - $length_Q); $i++) {
    for ($j = 0;$j < $length_Q; $j++) {
        if ($DNA_Q[$j] ne $DNA_T[($i + $j)]) {
            last;  # equiv. to "break"
        }
    }
    if ($j == $length_Q) {
        print ("Found match at $i in chromosome\n");
        $num_matches++;
    }
}
```

(Adapted from http://www.bioinformatics.wsu.edu/bioinfo_course/notes/lecture4.pdf)

Using (and Abusing) Defaults

- Many operators and functions have implied default arguments (@_, $_, STDIN, etc.)
- For example: "$_" is default for much I/O, regexp, etc.; so, could do:

```
while ($line = <STDIN>) {
    if ($line =~ /[a-z]+[0-9]/) {
        print ("Found an identifier: ");
        print $line);
```

- Or, could do:

```
while (<STDIN>) {          # implied "$_ = "
    if (/[a-z]+[0-9]/) {   # implied "$_ =~ "
        print ("Found an identifier: ");
        print;             # implied "print $_"
```

- STDIN, STDOUT
- "while ($in = <STDIN>)" works because "\n" is always left in the string, so != "" or "0", and therefore, true.

Issues with Perl

- Bends (possibly outright violates) principles of simplicity and orthogonality
  - Perl motto: "There's More Than One Way To Do It" (capitalization theirs; commonly known as "TMTOWTDI")
  - Footnote in Camel book specifically saying Perl is not orthogonal, but rather, "diagonal"
  - E.g.: Perl has both "if" vs. "unless", "while" vs. "until"

Issues with Perl

- Examples of multiple ways to do simple things (from book):

```
@days[3 .. 5]
  == ($days[3], $days[4], $days[5])
  == @days[3, 4, 5]

'xyz'
  ==  q/xyz/  ==  q#xyz#  ==  q{xyz}  ==  q<xyz>

"abc"  ==  qq/abc/  == …

s/abc/xyz/
  ==  s{abc}{xyz}  ==  s{abc}[xyz]  ==  s[abc][xyz]
  ==  s[abc]<CR>
      [xyz]
```

Issues with Perl

- This "multiple ways" philosophy can lead to semantic ambiguity
  - Note the following regexp pattern (in Perl, pattern matches are subject to *interpolation*; explained earlier):
    `/$foo[bar]/`

  So, should it be treated as an interpolated $foo, followed by a character class, as in:
    `/${foo}[bar]/`

  Or should it be parsed as a subscripted array:
    `/${foo[bar]}/`

  Book says if @foo doesn't exist, Perl will treat "[bar]" as a character class, but if @foo *does* exist, it will "try to guess"!

Issues with Perl

- Readabilty/Writeability:
  - Wall et al. claim (talking about how Perl borrowed elements from many other languages):

    In fact, just about any programmer can read a well written piece of Perl code and have some idea of what it does.
    ("Programming Perl", p.XI)
  - IMHO, Perl is high in writeability, but very low in readability, due to lack of simplicity and orthogonality
  - Poor readability leads in turn to issues with maintainence, reliability, and modification

Issues with Perl

- No declarative standard: the language is what the canonical (and only) interpreter does.
  - Perl6 has language spec, but Perl6 is not real

Random Confusing Examples

- Example 1: What is true??

  0.00 == 0 == "0", so: == false

  "0.00" != "0", so: == true

  But: "0.00" + 0 == 0.00 == 0 == "0", so == false
- Example 2: Regexp ambiguity
  - The regexp: "/last$/" will search for "last" occurring at end of string…
  - But "/last$a/" will search for "last" with value of $a interpolated, anywhere in string!

Random Confusing Examples

- Example 3: How do I access function parameters? ("Let me count the ways…")

```
sub my_add    # parameters are in @_
{
    # can access directly:
    $a = $_[0] + $_[1]
    # Or, can use:
    ($num1, $num2) = @_;
    # Or:
    $num1 = shift @_;
    $num2 = shift @_;
    # Or could have used (assuming more args):
    $num3 = shift;
    $num4 = shift;
}
```

Random Confusing Examples

- Example 4: Perl's "do {} while"
  - "do {…}" is not a compound statement: they are just terms in an expression; so…
  - Terminating ';' mandatory in "do {};"
  - "do {…} while (cond)" is not a control structure: it is a "while" modifier to a "do {}" term… but: the usual action of the "while" modifier is altered to execute the "{…}" part at least once.

Random Confusing Examples

- Example 5: What will following print out?

```
sub foo { # Recall: @_ contains the arguments
    print " in foo:"; print @_;
}

sub test {
    print "\n foo1:";
    foo(" Hello", " bye");
    print "\n foo2:";
    foo();
    print "\n foo3:";
    &foo;
    print "\n foo4:";
    foo;
}

&test("yes", "no");
```

- Output:

```
foo1: in foo: Hello bye
foo2: in foo:
foo3: in foo: yes no
foo4:
```

Random Confusing Examples

- Example 6: Barewords:
  - In some cases, Perl lets you omit quotation marks for string literals
  - For example, can do:

```
print STDOUT hello, ' ', bye, "\n";
# outputs "hello world"
```

  - BUT: if you leave out STDOUT because it is default:

```
print hello, ' ', bye, "\n";
```

    - Tries to parse *hello* as filehandle, gets syntax error
  - However, if you don't use bareword:

```
print "hello", ' ', bye, "\n";
# It works!  outputs "hello world" again
```

PGA (Perl Golf Apocalypse)

- Write a subroutine that accepts a list of words, and returns the list sorted by the first *vowel* that appears in each word. If two words have the same first vowel, it does not matter which is sorted first. Words will be in lowercase and will always contain at least one vowel. (Vowels are a, e, i, o, and u.)

  e.g.: hole('dog', 'cat', 'fish', 'duck', 'lemur')
  returns: ('cat', 'lemur', 'fish', 'dog', 'duck')

```
sub hole{sort{($a=~/([aeiou])/)[0]cmp($b=~//)[0]}@_}
```

Resources

- Another resource:
  http://www.perltutorial.org/introducing-to-perl.aspx