Introduction to Perl

Info 427

Announcements

- Schedule posted on OnCourse
  - Click the Wiki link
  - Readings, resources, assignments posted here
- Find a Perl book!

Our Plan

- We want to understand how search engines work
  - So we’re going to build one!
- But first we need to catch up on Unix and Perl, which will make up our toolbox...

Perl

- Practical Extraction and Report Language
  - Written by Larry Wall (1987)
  - Designed for writing programs quickly (not necessarily elegantly)
- Perl is a high-level, interpreted language
- Popular versions: 4 (mostly obsolete), 5 (current), 6 (upcoming)
  - We’ll use Perl 5
- Ubiquitous, especially for web services
  - "The duct tape that holds the Internet together"
  - Major users include Amazon, Craigslist, IMDb, Livejournal, Slashdot, ...

Perl: Pros and cons

- **Pro:** Very powerful and flexible. You can accomplish a lot in a short program.
  - Perl's semi-official motto: "There's more than one way to do it."
- **Con:** Makes it easy to write unreadable, unmaintainable programs. Perl's syntax can be hard to understand.
  - Programming Languages (PL) researchers often dislike Perl for this reason

  "Perl is the swiss army chainsaw of languages." – Henry Spencer

A first program

Winner of 5th Obfuscated Perl contest
By G. Taylor
A first program

```perl
#!/usr/bin/perl
use warnings;
use strict;

# endless print loop
while (1) {
    print "hello world!\n";
    sleep 1;
}
```

These 3 lines not required, but are good programming practice. Required for assignments in this class!

Running the script

- Run your programs with the "Perl" command
  ```
  [djcran@capricorn ~]$ perl my_program.pl
  ```
- Or make sure you have the `#!/usr/bin/perl` as the first line of your program, change the permissions of the program file to be executable, and then:
  ```
  [djcran@capricorn ~]$ chmod u+x ./my_program.pl
  [djcran@capricorn ~]$ ./my_program.pl
  ```

Data types in Perl

- Scalar types
  - Numbers, strings
  - Variable names must start with $  
- Lists (aka arrays)
  - Store a collection of scalars
  - Variable names must start with @
- Hashes (aka associative arrays)
  - Store a mapping from scalars to scalars
  - Variable names must start with %

Scalar variables in Perl

- Scalar variables store numbers and/or strings
  - Scalar variable names must begin with $  
- Please use descriptive names
  - Bad:  $a, $x1, or $xx2new
  - Better:  $employee_name
- Use keyword my to declare new variables
  - Makes the variable local in scope.
  ```
  my $page_count = 20;
  ```

Variables in Perl

- Perl is dynamically typed
  - The type of a variable is determined based on context, and can change throughout the program
  ```
  my $name = 7;
  $name = 'Dennis';
  ```
- Q. What value do new variables have?
  - A. A special value called undef

Numeric Operators

- Assignment
  ```
  $var = 7;
  $newvar = $var;
  ```
- Basic math
  ```
  $var = 3*5.2;
  $var = 3+5 / 2;
  $var = $var + 7;  # shortcut: $var +=7;
  $var += 1;  # shortcut: $var++;
  $var = 2**5;
  ```
String literals

- Strings can be written in one of 2 ways
  - Strings in single quotes are interpreted literally
    ```perl
    my $a = 12;
    print 'Hello! \n $a \n';
    ```
  - Strings in double quotes are modified: variable names are replaced with values, escape codes are resolved
    ```perl
    my $a = 12;
    print "Hello! \n $a \n";
    ```

Useful escape codes:
- newline \n
String Operators

- Use the . (dot) operator to concatenate strings
  ```perl
  $first_name = "Barack";
  $last_name = "Obama";
  $name = $first_name . " " . $last_name;
  print $name;
  ```

  Or equivalently:
  ```perl
  $name = "$first_name $last_name";
  print $name;
  ```

Type conversion

- If you use a numeric operator on a string, Perl will try to convert the string to a number
  - If it can't, Perl will print a warning and use a value of zero
    ```perl
    $test = "3.1415926" * 2;
    print "$test\n";
    ```
    ```plaintext
    6.2831852
    ```

Getting input

```perl
#!/usr/bin/perl
use strict;
use warnings;

print "What is your name? ";
my $a = <STDIN>; # stands for "Standard Input"
print "Hello $a, how are you?\n";
```
“if” expressions

• if (condition) {  
  # this is executed if condition is true  
}

• if (condition) {  
  # this is executed if condition is true  
} else {  
  # this is executed if condition is false  
}

• if (condition1) {  
  # this is executed if condition1 is true  
} elsif (condition2) {  
  # this is executed if condition1 is false but  
  condition2 is true  
} else {  
  # default -- if both conditions are false  
}

Basic comparison

```perl
$a = 1;
if ($a > 1) {
    print "$a is more than 1\n";
} else {
    print "$a is not more than 1\n";
}
```

Conditional operators

• Numbers and strings have different conditional operators

<table>
<thead>
<tr>
<th>Relation</th>
<th>Numeric</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equal</td>
<td>==</td>
<td>eq</td>
</tr>
<tr>
<td>Not equal</td>
<td>!=</td>
<td>ne</td>
</tr>
<tr>
<td>Less than</td>
<td>&lt;</td>
<td>lt</td>
</tr>
<tr>
<td>Greater than</td>
<td>&gt;</td>
<td>gt</td>
</tr>
<tr>
<td>Less than or equal</td>
<td>&lt;=</td>
<td>le</td>
</tr>
<tr>
<td>Greater than or equal</td>
<td>&gt;=</td>
<td>ge</td>
</tr>
</tbody>
</table>

Basic comparison

```perl
$a = “A”;
if ($a gt “A”) {
    print "$a is later in the dictionary than A\n";
} else {
    print "$a is not later in the dictionary than A\n";
}
```

String comparisons

• String comparisons test lexicographic order  
  – Uppercase letters first; i.e. “A” it “a” is true

• Some examples:
  “Dennis” < “dennis”
  “Dennis” lt “dennis”
  “2” gt “5”
  “20” gt “5”
  “20” > “5”

basic comparison

Basic comparison

```perl
$a = “A”;
if ($a gt “A”) {
    print "$a is later in the dictionary than A\n";
} else {
    print "$a is not later in the dictionary than A\n";
}
```

String comparisons

• String comparisons test lexicographic order  
  – Uppercase letters first; i.e. “A” it “a” is true

• Some examples:
  “Dennis” < “dennis”
  “Dennis” lt “dennis”
  “2” gt “5”
  “20” gt “5”
  “20” > “5”

unless

• The opposite of if ... else

unless (condition) {
  # executed if condition is false
} else {
  # executed if condition is true
}
Alternative syntax for if and unless

• Syntax: statement if condition;
  – Executes statement iff condition is true
  – Equivalent to: if(condition) { statement; }

• Syntax: statement unless condition;
  – Equivalent to: if(!condition) { statement; }

• Examples
  print "OK" if $a == 3;
die "undefined!" unless defined $x;

Arrays and lists

• An array is another type of variable that holds an ordered list of scalar values
  – Array variable names begin with @

• Lists are written as comma-separated values, surrounded by parentheses, e.g.:
  @names = ("Fil", "Dennis"); # 2 values
  @ages = (45, 57, 22, 18); # 4 values
  @userids = (); # 0 values
  @ints = (3..7); # shortcut for (3,4,5,6,7)

Accessing array elements

• The first element of array @names is referred to as $names[0]
  – Array indices are 0-based, so the n-th element is $names[n-1]

my @names = (’b’, ’obama’, ’g’, ’bush’, ’b’, ’clinton’);

Accessing array elements

• The first element of array @names is referred to as $names[0]
  – Array indices are 0-based, so the n-th element is $names[n-1]

my @names = (’b’, ’obama’, ’g’, ’bush’, ’b’, ’clinton’);

Accessing array elements

• The first element of array @names is referred to as $names[0]
  – Array indices are 0-based, so the n-th element is $names[n-1]

my @names = (’b’, ’obama’, ’g’, ’bush’, ’b’, ’clinton’);

Accessing array elements

• The first element of array @names is referred to as $names[0]
  – Array indices are 0-based, so the n-th element is $names[n-1]

my @names = (’b’, ’obama’, ’g’, ’bush’, ’b’, ’clinton’);

Accessing array elements

• The first element of array @names is referred to as $names[0]
  – Array indices are 0-based, so the n-th element is $names[n-1]

my @names = (’b’, ’obama’, ’g’, ’bush’, ’b’, ’clinton’);

Accessing array elements

• The first element of array @names is referred to as $names[0]
  – Array indices are 0-based, so the n-th element is $names[n-1]

my @names = (’b’, ’obama’, ’g’, ’bush’, ’b’, ’clinton’);
Accessing array elements

- The first element of array $\text{@names}$ is referred to as $\text{@names[0]}$
  
  Array indices are 0-based, so the n-th element is $\text{@names[n-1]}$

```perl
my @names = ('b', 'obama', 'q', 'bush', 'b', 'clinton');
print @names;
```

```
obama bush clinton
obama bush clinton
obama bush clinton
bobamagbushbclinton
```