

CS306: INTRODUCTION TO PERL

Lists and Arrays (Chapter 4)

U. of Alabama at Birmingham
Dept. of Computer & Information Sciences

REVIEW - SCALARS

- A scalar is one single piece of data in Perl.
- A scalar could be a number or a string.
- A scalar variable is a container for one single piece of data.
- Scalar variables are indicated with a \$ prefix, such as \$name.

LISTS

- A *list* is an ordered collection of scalars. We call each individual scalar an *element* of the list.
- The elements of a list are *indexed* starting at 0.
- Each element is independent scalar which means that they can contain anything a scalar can
- The elements don't have to hold the same kind of data, one could be a string, another a number, another undef, etc...

CREATING LIST LITERALS

- Use the () notation
- () # The empty list
- ("fran", "ying", "tony", "\n")
- (1, 5, 3, 4)
- ("fran", 2, undef, "seven", 4.5)
- (1, 4, 2)

LISTS AS FUNCTION ARGUMENTS

- `print("this","function","expects","a","list");`
- `print "this", "function", "expects", "a", "list";`
 - The parens are optional to functions that take lists
- `print "a one-element list";`
 - Even when just one parameter, Perl promotes the item to a one-element list

OTHER WAYS TO CREATE LISTS

- The *qw* shortcut - "quoted words"
 - `qw /george jane judy elroy/`
 - Other delimiters like `!!` `{}` `[]` `<>` `()` also work
- The *..range* operator
 - `(1..10)` # Creates a 10 element list
 - `('A'..'Z')` # The alphabet in capital letters

ACCESSING LISTS

- `print (qw(perl, java, ruby, c)[2]);`
 - prints "ruby"
- Use the `[]` notation to select the value at that index from the list.
- `$num = 1;`
`my $scalar = (4,2,7)[$num];`
 - What value does `$scalar` get?

SLICES

- `print (qw(fee, fi, fo, fum)[1,3]);`
- Has the effect of selecting a new sublist. Essentially, this statement becomes:
`print ("fi", "fum");`
- Ranges + Slices...
`print (qw(eeny, meeny, miney, moe)[1..3]);`
prints "meeny miney moe"

ARRAYS

- Lists aren't very useful unless we can store the data somewhere, so we have *arrays*.
- `@oddnums = (1,3,5,7,9);`
- `@words = qw/blazers tigers tide/;`
 - Note that `$words` and `@words` are two completely separate variables!
- `@morewords = ("gators", @words, "vols");`

SCALAR VS. LIST CONTEXT

- “What type of value does this piece of code want?”
- `@array2 = @array1;`
 - This is an array on the left side. Arrays hold lists. So, we want to end up with a list on the left side. Thus, this is list context.
- `$scalar = @array1;`
 - There's a scalar on the left. Scalars can only hold one value, and this is scalar context.

SCALAR VS. LIST CONTEXT

- **WHAT HAPPENS WHEN YOU TRY TO SQUEEZE A LIST INTO SCALAR CONTEXT?**
- `MY @WORDS = qw/HOMER MARGE BART LISA MAGGIE/;`
`MY $SCALAR = @WORDS;`
`PRINT $SCALAR;`
 - **WHAT DO YOU THINK THIS WILL PRINT?**

ACCESSING ARRAYS

- Assign to a scalar:
`my @names = qw/joe fran stacy/;`
`my $name = $names[1];`
- Didn't we just say `$names` and `@names` are two separate variables? Why `$names[1]` here?
- The `@` and `$` signify *what we want*, not what it is. The thing at `[1]` of `@names` is a single element, a scalar. So we use `$` to indicate that the thing at index 1 of `names` array is a scalar value.

ACCESSING ARRAYS

- Grabbing multiple elements at once
 - This implies we want to get a list out, so assign to a list. A list of what? Scalar variables.
- ```
@nums = (2,4,6);
($num1, $num2, $num3) = @nums;
($num1, $num2) = @nums[0,3];
```

## GETTING THE LAST ELEMENT

- Grab the index of the last element with \$#  

```
@nums = (1,2,3,4);
$last = $#nums;
```
- Remember, this is the index of the element. To get the value:  

```
print $nums[$#nums]; or print $nums[$last];
```
- Use -1 subscript to get to last element
- ```
print $nums[-1]
```

ITERATING OVER ARRAYS

- Java/C style:

```
for ($i = 0; $i <= $#array; $i++) {  
    # each time through, $array[$i] is  
    # the next element in @array  
}
```

ITERATING OVER ARRAYS

- Perl style:

```
foreach my $elem (@array) {  
    # each time through, $elem is the  
    # next element in @array  
}
```
- ```
for (@array) { # shorthand for same thing
 # each time through $_ is the next element
}
```

## FUNCTIONS FOR ARRAYS

- *pop()* and *push()* – operate on right side of array
  - `$lastelement = pop @array; # @array is now shorter`
  - `push @array, $new_element_on_right;`
- *shift()* and *unshift()* – operate on left side of array
  - `$firstelement = shift @array;`
  - `unshift @array, $new_element_on_left;`

## FUNCTIONS FOR LISTS

- *reverse()* – reverses the elements of a list
  - `@ascending = reverse @descending;`
- *sort()* – sorts the elements of a list
  - `@sorted = sort @names;`
  - sort is an extremely powerful/general/flexible operator – it can sort any kind of data. In this most simple form, it sorts via string comparison.

## LISTS <--> STRINGS

- *join()* – join a list into a string using a separator
  - `@names = ("fran", "tony", "ying");`  
`$string = join '|', @names; # "fran|tony|ying"`  
`$string = join @names; # "fran tony ying"`
- *split()* – splits a string into a list on a separator
  - `$string = "fran tony ying";`  
`@names = split / /, $string;`

## INTERPOLATING INTO STRINGS

- An entire array
  - `@teams = qw / blazers tide tigers /;`  
`print "gators @teams dawgz";`  
`# produces "gators blazers tide tigers dawgz"`
- Just one element
  - `print "The $teams[0] made the NCAA Tournament last year."`

## MORE ON CONTEXT

- Context simply means “What is Perl expecting here?”
- `8 - ???` # Here Perl expects a scalar
- `sort ???` # Here Perl expects a list
- `???` could be the same exact thing in both instances, but lead to a different answer, because it was used in a different context. IMPORTANT.

## CONTEXT EXAMPLES

- `@heros = qw/ bond luke indy lara/;`
- List context
  - `sort @heros` # produces “bond indy lara luke”
- Scalar context
  - `5 + @heros` # produces 9
  - Since you used an array in scalar context, Perl assumed you meant the length of the array

## CONTEXT EXAMPLES 2

- Assignment
  - `@array = “foo”;` # creates one element list
  - `$size = @array;` # \$size gets the size of the array
- Some functions do different things depending on the context in which you use them
  - `@ascending = reverse @descending;`
  - `$name = reverse “fran”;` # produces “narf”

## CONTEXT EXAMPLES 3

What's the context of each of these?

`$result = ???`

`@array = ???`

`($name, $address) = ???`

`($result) = ???`

`$colors[3] = ???`

## A SPECIAL CONTEXT CASE - <STDIN>

- <STDIN> in scalar context
  - `$line = <STDIN>;`
  - `chomp $line;`
- <STDIN> in list context
  - `@lines = <STDIN>; # grabs all lines until EOF`
  - `chomp @lines;` or `chomp(@lines = <STDIN>);`