

Section 1: Scalars

What are scalars?
Creating scalar variables
Operating on scalars

Slide 1

What are scalars?

- In Perl, a *scalar* is a singular piece of data, either a number or a string
- Some examples:
14
hello
5.64234
This sentence is a scalar!
<the entire text of the Programming Perl book>

Slide 2

What are scalars? 2

- Perl treats numbers and strings nearly interchangeably
- If you use a string where Perl expects a number, it will use the string as a number, and vice versa. Examples later.

Slide 3

Numbers

- Integers
5
745634
- Floating Point
5.2304
3.88e12
-56.3421

Slide 4

Numbers in Various Bases

- Decimal
46
- Hexadecimal
0x2e
- Binary
0b101110
- Octal
056 # Just a leading zero in this case

Slide 5

Strings

- String *literals** are represented in a Perl program within quotes
'hello!'
'Four score and seven years ago'
'What\'s for dinner?'
- Everything within the single quotes is interpreted literally. Use \ to represent ' in your strings.

* A literal is a piece of data represented directly in your program as opposed to being stored inside a variable. i.e. "Hard-coded" Slide 6

Double-Quoted Strings

- To represent *control characters* within strings, use double-quotes instead.
"Go Blazers!\n"
"Fran\Fabrizio"
"She said, \"Take me to your leader.\""
- There are lots of control characters and the \ takes on a lot of power within double quotes. See *Programming Perl* page 61.

Slide 7

Numbers <--> Strings

- Perl will convert strings to numbers when needed
'35' + 4 # Equals 39
"16fran34" / " 8" # Equals 2
"tony" + 3 # Equals 3, non-numeric string converts to 0
- Numbers to strings works as well
"Y" . 2 . "K" # Equals "Y2K"

Slide 8

Operating on Scalars

- For numbers, all the usual *operators*
+ - * / % **
- For strings, we have
 - The concatenation operator
"base" . "ball" # Equals "baseball"
 - The repetition operator
'ho' x 3 # 'hohoho'
3 x 3 # "333" Don't get fooled :-)

Slide 9

Scalar Variables

- Perl uses the \$ to indicate a scalar *variable*
\$name
\$return_value
\$aLengthyVariableName
- Variable names are case-sensitive, can contain alphanumeric and underscore characters, cannot start with a digit, and can be of any length

Slide 10

Scalar Assignment

- \$day = 'Wednesday';
- \$grade = '95';
- \$bonus = 5;
- \$total = \$grade + \$bonus; # 100
- \$extra_credit = 10;
- \$total += \$extra_credit; # 110
- \$day .= ' afternoon'; # "Wednesday afternoon"

Slide 11

String Interpolation

- When using double-quotes, not only does the \ become more powerful, but you can also perform *variable substitution*
\$day = "Wednesday";
\$phrase = "\$day night"; # Wednesday night
\$literal = '\$day night'; # \$day night

Slide 12

String Interpolation 2

- When perl sees a \$ inside double-quotes, it tries to form the longest variable name possible

```
$word = 'dog';
```

```
$string = "The $word barks."; # The dog barks
```

```
$string = "Cats and $words"; # Cats and
```

... because \$words is not defined. Instead:

```
$string = "Cats and ${word}s"; # Cats and dogs
```

Slide 13

String Interpolation 3

- Four ways to accomplish the same thing

```
$word = 'dog';
```

```
$string = "Cats and ${word}s\n";
```

```
$string = "Cats and $word" . "\n";
```

```
$string = "Cats and $word" . s . "\n";
```

```
$string = "Cats and " . $word . "\n";
```

Slide 14

Comparing Scalars

- For numbers, you can use:

```
== != > < >= <=
```

- For strings, you can use:

```
eq ne gt lt ge le
```

- A very common Perl mistake:

```
'cat' == 'dog' # True, Perl makes both sides 0
```

```
'cat' eq 'dog' # False, what you meant to do
```

Slide 15

undef

- If a scalar variable has not yet been assigned a value, it has the Perl special value *undef*
- undef behaves like 0 when used as a number and like an empty string when used as a string

```
5 + $sum # equals 5
```

```
$sum++;
```

```
5 + $sum # equals 6
```

```
$newstring .= "hello"; # $newstring is "hello"
```

Slide 16

Boolean Values

- Perl does not require or have a specific boolean data type
- The following evaluate to false
`undef 0 '0' ''`
- Everything else evaluates to true
- Use the `!` to get the opposite of a boolean
`(! $done)` # if `$done` is 1, evaluates to 0

Slide 17

Simple output

- `print "Hello World!\n";`
- `$weather = 'rainy';`
- `print "It will be $weather today.\n";`
- `print "My name is ";`
`print "Fran";`
`print "\n";`
- `print "My name is " . "Fran" . "\n";`

Slide 18

The if Construct

- `if (some condition) {`
 `# do this if condition is true`
`} else {`
 `# do this if condition is false`
`};`
- `if ($name eq 'Fran') {`
 `print "Hello Mr. Fabrizio\n";`
`}`

Slide 19

The if Construct 2

- `if (! $finished) { print "Keep working!\n"; }`
- `if ($pushups >= 30) {`
 `print "Great job!\n";`
`} elsif ($pushups >= 20) {`
 `print "Not too bad.\n";`
`} else {`
 `print "You'd better visit the new rec center.\n";`
`}`

Slide 20

Simple input

- Use <STDIN>
 - The full explanation of this will occur in a later lecture
- print “Your name please: “;
\$name = <STDIN>;
print “Hello \$name!”;

Slide 21

Simple input 2

- Note that <STDIN> grabs an entire line of input from standard input (typically the keyboard), including the newline character at the end
- The chomp() function is handy here
 - \$name = <STDIN>;
chomp \$name;
 - chomp() looks for and removes a trailing \n
 - Important to remember to remove it, otherwise it will mess up things like string comparisons

Slide 22

The End

- Grab the homework assignment off the web site tonight.
- Post questions to the web forum
- Review today's material for the quiz next class.

Slide 23