Context

Context

- Every expression in Perl is evaluated in a specific “context”.
  - mode, manner, meaning
- return value of expression can change depending on its context
- Perl variables and functions are evaluated in whatever context Perl is expecting for that situation
- Two *major* contexts – Scalar & List

Scalar Context

- **`$x = EXPR;`**
- **`if (EXPR < 5) { … }`**
- Perl is “expecting” a scalar, EXPR is evaluated in scalar context
  - assign to a scalar variable, or use an operator or function that takes a scalar argument
- force scalar context by **`scalar`** keyword
  - **`@array = scalar EXPR;`**

Scalar Sub-contexts

- Scalar values can be evaluated in Boolean, String, or Numeric contexts
- Boolean:
  - **`0, '0', '',`** and **`undef`** are all false
    - (zero, string containing zero, empty string, undefined)
  - anything else is true
- String: **`'hello world', 'I have 4'`**
- Numeric: **`5, 3.4, -5`**
- Perl will *automatically* convert to and from each of these contexts for you. Almost never need to concern yourself with them.

Automatic Conversions

- If a number is used as a string, the conversion is straight forward.
  - **`853`** becomes **`'853'`**
  - **`-4.7`** becomes **`'-4.7'`**
- If a string is used as a number, Perl will convert the string based on the first character(s)
  - If first non-space character is 'numeric' (ie, number, period (decimal), or negative (hyphen)), converted number reads from start to first non-numeric character.
    - **`'-534.4ab32'`** ➔ **`-534.4`**
  - If first character is non-numeric, converted number is 0.
    - **`'a4332.5'`** ➔ **`0`**
- If a scalar is used in a conditional (if, while), it is treated as a boolean value

When does this happen?

```
my $foo = 4;
print "Enter a number\n";
my $bar = <STDIN>; #note no chomp
my $sum = $foo + $bar;
```

- Note that **`$bar`** is unaffected. It's just used as a number in that one statement
- Method for checking input for numeric data involves Regular Expressions
  - ie, don't worry about it for now

List Context

- **`@x = EXPR;`**
- **`$str = join (' ', EXPR);`**
- Assign to a list/array, or use in a function or operator that is expecting a list
- There is no analogy to the **`scalar`** keyword for lists. If you use a scalar in any kind of list context, it is "promoted" to a list.
  - **`@array = 5;`**
  - @array gets value: (5)

Context Fun

- arrays evaluated in scalar context produce the size of that array
  - `my @x = (4, 8, 12);`
  - `my $size = @x;`
  - $size gets value 3.
- `print "@x has " . @x . " values.\n";`
  - The `.` operator expects a string on either side
    - strings are scalars, so `@x` is evaluated in scalar context
  - "4 8 12 has 3 values."
  - Contrast: `print "@x has ", @x, " values.\n";`
    - Comma operator expects a list on either side. `@x` evaluated in list context
    - Just printing a list of five values, so separated by `$,` (space)
    - "4 8 12 has 4812 values"
- `my @x = ('a', 'b', 'c');`
  `my $y = @x;`
  `my ($z) = @x;`
  - same as `my ($z) = ('a', 'b', 'c');`
- $y → 3, $z → 'a'

undef

- Any scalar variable which exists but is not defined has default value **`undef`**
  - `my ($a,$b,$c)=(15,20); # $c -> undef`
- In string context, undef → `''`
- In numeric context, undef → `0`
- In boolean context, undef → false
- **`use warnings;`** will warn you about using an undefined value!
- Array variables not given a value get the empty list
  - `my @bar; # @bar contains ()`
- **`my @foo = undef; #probably wrong!`**
  - @foo contains one element: the undefined value
  - To clear an existing array: `@foo = ();`
  - If you need to do this, you probably declared your variable in too large a scope