Day 5: Subroutines

Suggested reading: *Learning Perl* (4th Ed.)

Chapter 4, Subroutines

Turn In Homework

Homework #3 Comments:
Where to Define Variables?

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

my %groceries;
my $total = 0;
my $item;
my $count;
my $average;

while (1) {

    # ...

}
```

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

my %groceries;                  # no initial value

while (1) {

    my $total = 0;
    foreach my $item (keys %groceries) {
        $total += $groceries{$item};
    }

    my $count = scalar(keys %groceries);
    my $average = $total / $count;
}
```

Homework Review

Background

The Problem

```
my $sum = 0;
my $count = 0;
while (my $temp_c = <INPUT_FILE>) {
    my $temp_f = ($temp_c * 9 / 5) + 32;
    print "Temp: $temp_f F\n";
    $sum += $temp_c;
    $count += 1;
}

my $avg_c = $sum / $count;
my $avg_f = ($avg_c * 9 / 5) + 32;
```

Don't Repeat Yourself (DRY)

- Code
- Data
- Configuration
- Documentation

Hunt & Thomas (1999), *The Pragmatic Programmer*

E. W. Dijkstra (1968)
Go to statement considered harmful

Communications of the ACM, 11, 147–148

Solution: Procedures

- Also called: subroutines, functions, methods, …
- Organize (some) code into "chunks"
  - Maximize code reuse / minimize repetition
  - Organize code clearly (decomposition)
  - Make testable units of code
- Like a script within a script
  - Consists of Perl statements
  - Accepts input
  - Can give back output
  - Has its own state (i.e., variables)

Subroutines

Defining a Subroutine

```
sub calculate_total {
    $total = 0;
    foreach my $item (@array) {
        $total += $item;
    }
}
```

- Put anywhere, but ***not*** in a statement or **{}**
- Namespace is distinct (but don't abuse this!)

Using a Subroutine

```
&subroutine_name;
 subroutine_name();
&subroutine_name();
```

- **&**: almost always OK, often optional
- **()**: often optional, sometimes helpful

- A subroutine call is an expression:

```
&halt_cpu if temperature_too_high();
```

When Subroutines Are Run

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

my $target = 'world';

sub say_hello {
    print "Hello, $target!\n";
}

print "Hello, everyone!\n";
$target = 'Tim';
say_hello();
```

Subroutine code not run when defined, only when called

Subroutine Input: Arguments

We want to provide input to a subroutine

```
square_root(81);

show_greeting('Tim');

average(@list_of_numbers);

print_num_with_precision($pi, 10);
```

Using Arguments I — The Bad Way

Remember automatic variables?

Within a subroutine, arguments are in `@_`

```
sub examples_of_using_arguments {

    foreach my $argument (@_) {
        print "Argument: '$argument'\n";
    }

    if ($_[0] > $_[1]) { ... }

    my $named_argument = $_[2];
}
```

Using Arguments II — Better Options

```
sub option_1 {
    my ($foo, $bar) = @_;
    # ...
}
```

```
sub option_2 {
    my $foo = shift;    # @_ is implied
    my $bar = shift;    # @_ is implied
    # ...
}
```

Subroutine Output: Return Values

Default: Return the last expression ***evaluated***

```
sub bigger {
  my ($a, $b) = @_;
  if ($a > $b) { $a } else { $b }
}
```

An explicit **`return`** is usually clearer:

- Stops executing the subroutine immediately
- Returns the given value

```
return;
return 42;
return "Hello, $name\n";
```

Returning Lists

Perl lets you return lists, too:

```
sub how_do_i_love_thee {
    # ...
    return @the_ways;
}

my @array = how_do_i_love_thee();
print "Let me count... "
print scalar(@array);
print "\n";
```

Scoping

```
sub subroutine {
   my $answer = shift;
   $answer /= 6;
   print "Subroutine answer: $answer\n";
}

my $answer = 42;
print "Main answer: $answer\n";
subroutine($answer);
print "Main answer: $answer\n";
```

Make a Subroutine…

- For repeated code (DRY)
- For logical organization
  - capture main flow vs. parts
  - break up excessively long parts
  - scope control
- For testing

Already been using: **`print`**, **`chomp`**, **`open`**, …

```
ask_questions()
```

Other Scripting Languages

- All have procedures
- Syntax varies widely
- Look for:
  - explicit declaration of argument signature
    [Ruby] **`def foo(arg1, arg2, blah)`**
  - default arguments
    [Ruby] **`def bar(arg1, arg2 = 42)`**
  - different scoping rules (PHP)
  - different requirements (Python requires return)
  - anonymous functions / closures (Ruby)

Homework

- Unit conversions!
- Subroutine usage is a bit forced…
- Extra cool bonus challenge: Can you avoid defining the conversion factor between every ***pair*** of units? Imagine you have 10 different length units (inch, foot, mile, meter, etc.); there are $10 \times 9 = 90$ unique conversion pairs (e.g., inch –> foot). But I claim you need only 9–10…