

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 #3 Comments:

- Where to create from total?
- Each time when displaying list & total

Homework Review

Theory

Procedural Programming

- group statements into named units
- *aka* procedures, functions, subroutines, methods
- executed from anywhere, including self (recursion)
- usually supports inputs and output(s)
- like a script within a script

But why?

E. W. Dijkstra (1968)

Go to statement considered harmful

Communications of the ACM, 11, 147–148

Don't Repeat Yourself (DRY)

- Code
- Data
- Configuration
- Documentation

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

OK, OK, but... when?

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**, ...

~~Theory~~ Code

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();
```

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 \rightarrow foot). But I claim you need only 9–10...