Topic 7

Functions

Learning Perl 2nd edition
chapter 8, pages 92-100
Programming Perl 3rd edition
pages 217-225
Programming Perl 2nd edition
pages 111-121
`perlsub` manpage

1

Last time

Covered in Topic 6

- Regular expressions
- Pattern matching
  - `/pattern/`
- Substitution
  - `s/pattern/replace/`
- Functions that use regular expressions
  - `split`, `join`

2

To be covered today

- Defining functions (subroutines)
- Calling functions
- Returning values from functions
- Passing arguments
  - the `@_` array
- Local variables
  - `my` and `local` keywords
- Sorting arbitrarily

3

Subroutines

Notes and terminology

- In Perl, called functions or subroutines
  - no difference in meaning
- All subroutines can take parameters
  - called arguments or actual parameters by caller
  - called (formal) parameters by function
- No type checking
  - of parameters or return value
  - optional prototypes allow rudimentary type checking
- No formal naming of parameters
  - programmer can do this if desired

4

Defining a function

Also called a subroutine

subroutines are defined with `sub` keyword

name of subroutine goes here

```
sub greet {
  print "Hello there.\n";
}
```

body of subroutine can contain anything

braces are required

Llama2 pages 92-93; *Camel3* pages 217-218
Camel2 pages 111-112

5

Subroutines

Scope

- Subroutines may be declared anywhere in the program
  - definitions are skipped on execution
  - by convention, definitions go first or last in code
- Subroutines can access all global variables
  - can declare localized variables with `my` keyword

Llama2 pages 93, 96; *Camel3* page 218, 223
Camel2 page 189

6

Subroutines

Naming conventions

- Technically, subroutine names begin with special character `&`
  - variables `$days`, `@days`, `%days` and function `&days` are all separate
- In practice, no leading character is needed
  - when declaring, not used in `sub` definition
  - when calling, parentheses after the subroutine name identify it as a function call
    - `days(2000, 1, 1) # or could say &days(2000, 1, 1)`

Llama2 pages 92-93; *Camel3* page 218
Camel2 pages 111-112

Subroutines

Returning values

- Subroutines return to their caller the last expression evaluated
  - `sub pi { 3.1415926535898; }`
- Can use return keyword to return sooner
  - `sub abs_a { if ($a >= 0) { return $a; } else { return -$a; } }`
- Return value can be scalar or list
  - `sub first_two_args { return @ARGV[0,1]; }`
  - return value is interpreted according to context subroutine was called in
  - can use `wantarray` function to determine context

Llama2 page 94; *Camel3* pages 219, 228
Camel2 pages 112, 241

Parameter passing

Passing values to a function

- Caller names arguments in parentheses after function name
  - `$dayname = weekday($year, $month, $day);`
  - as with built-in functions, parentheses can be omitted if subroutine is pre-declared
- Arguments are formed into list and placed in special local array variable `@_`
- Subroutine can access `@_` or individual members
  - `sub weekday { ($y, $m, $d) = @_; ... }`
  - `sub weekday { $y = $_[0]; ... }`

Llama2 pages 94-96; *Camel3* pages 219-221
Camel2 page 112

Example

Calculating the hypotenuse

```
# Declare the hypotenuse function.
sub hypotenuse {
  # Assign parameters meaningful names.
  # Could also have done:
  # ($x, $y) = @_;
  $x = $_[0]; $y = $_[1];
  return sqrt($x * $x + $y * $y);
}

# Read two numbers on one line.
print "Enter two numbers: ";
($a, $b) = split /\s+/, <STDIN>;

print "Hypotenuse is: ",
  hypotenuse($a, $b), "\n";
```

Example

Summing a list

```
# Read some numbers into @nums.
while (<>)
{
  chomp; push @nums, $_;
}
print "Sum is ", sum(@nums), "\n";

sub sum {
  $sum = 0;
  # Iterate $_ over parameter list @_.
  foreach (@_)   # $_ is the default iterator.
  {
    $sum += $_;  # Add this list element to @sum
  }
  return $sum;
}
```

Local variables with `my`

Protecting variables from accidental modification

- By default, all variables are global
- Variables can be declared local (lexical scoping) with `my` keyword
  - `my ($sum); # Protects old value of $sum.`
- Old value is restored at end of enclosing block (often end of subroutine)
- Can localize and assign in one step
  - `my ($x, $y) = @_;`
  - parentheses needed because of precedence

Llama2 pages 96-97; *Camel3* page 132-133
Camel2 page 189

Example

Finding all elements in a list that match a pattern

```
# Almost same as the Perl builtin function grep.
sub filter {
  # Declare meaningful names for parameters.
  my ($pattern, @values);
  # Declare a temporary array for return value.
  my (@result);
  # shift in a function defaults to using @_.
  $pattern = shift;
  @values = @_;

  foreach (@values) {
    if /$pattern/ {  # Test $_ against $pattern.
      push @result, $_;  # Save this string.
    }
  }
  @result; # Return value.
}
```

CSE2395/CSE3395

13

`local` versus `my`

Use this only if you have to

- Perl has another kind of localizing of variables, using `local` keyword
- Use `local` where `my` does not work
  - `local $_;  # my $_ isn't allowed.`
- Otherwise, use `my`
  - `local` causes dynamic scoping of variables (they are visible inside all called functions); this is unfamiliar to C programmers
  - `my` causes lexical scoping, which behaves like local variables in C programs

Llama2 pages 98-99; *Camel3* pages 135-136
Camel2 pages 184-185

CSE2395/CSE3395

14

Global variables: `our`

Explicitly declaring globals

- Perl 5.6 has `our` keyword
  - declares global variable to be visible in current scope
  - `our $house;`
- Not needed unless programming under `use strict 'vars'`
  - undeclared variable is normally automatically global
  - `use strict` is recommended for modules or large programs

Camel3 pages 133-135

CSE2395/CSE3395

15

Parameter passing

Passing arrays and hashes

- Arrays and hashes are unwound into single list before being stored in `@_`
  - sizes of arrays are lost in unwinding
- If passing one array, make it the last argument
- Passing more than one array can't usually be done
  - `diff(@a, @b)` will pass one list containing all elements in both `@a` and `@b` to the `diff` function
  - if `diff` does `(@x,@y) = @_` then `@x` gets all elements, and `@y` is empty
  - can be solved with references (Topic 11)

Camel3 pages 221, 224-225; *Camel2* page 114

CSE2395/CSE3395

16

Sorting

Sorting a list numerically

- `sort` function normally sorts items alphabetically (lexicographically)
- Can sort by other criteria by providing comparison function
  - does not use normal parameter-passing mechanism
  - inside comparison function, `$a` and `$b` are aliases of two list elements
  - function must return
    - less than zero if `$a` precedes `$b`
    - zero if `$a` and `$b` are the same
    - greater than zero if `$a` follows `$b`

Llama2 pages 156-159; *Camel3* pages 789-793
Camel2 pages 217-219; `perlfunc` manpage

CSE2395/CSE3395

17

Example

Sorting a list numerically

```
# For code readability, use adverbs for names.
sub numerically
{
  # $a and $b are automatically localized
  # in this function.
  # Could also have said: return $a <=> $b;
  if ($a < $b) { return -1 }
  elsif ($a > $b) { return 1 }
  else { return 0 }
}

@list = (1, 128, 16, 2, 32, 4, 64, 8);

# Note name of function between keyword
# and list; also no comma after function name.
@newlist = sort numerically @list;
```

CSE2395/CSE3395

18

Covered today

- Defining functions (subroutines)
  - `sub` keyword
- Calling subroutines
- Returning values from subroutines
  - last expression evaluated in function
  - `return` keyword
- Passing arguments
  - the `@_` array
- Local variables
  - `my` and `local` keywords
- Sorting lists arbitrarily
  - comparison functions

Going further

More things related to today's topic

- Prototypes
  - making user-defined subroutines behave more like builtins
  - *Camel3* pages 225-228; *Camel2* pages 118-121
- Code generation
  - building Perl code on-the-fly with `eval`
  - *Camel3* pages 705-707; *Camel2* pages 161-163
- Built-in functions
  - a myriad of standard subroutines provided in Perl
  - *Camel3* pages 677-830; *Camel2* pages 141-242
- **`BEGIN`** and **`END`**
  - special functions that run before or after other code
  - *Camel3* pages 480-485; *Camel2* pages 283-284

Next time

To be covered in Topic 8

- File operations
  - `open`, `close`
- Reading from and writing to files
- File tests
- Scanning directories

Reading:
Learning Perl 2nd edition chapters 10, 12, 13
pages 108-115, 129- 133, 134-141
Programming Perl 3rd edition pages 20-22, 28-29, 97-100, 747-755, 770
Programming Perl 2nd edition pages 12-14, 19-20, 85-87, 191-195
`perlfunc`, `perlopentut` manpages