IT441

Network Services Administration

Subroutines

(a.k.a., Functions, Methods, etc.)

DRAFT

Organizing Code

- We have recently discussed the topic of organizing data (i.e., arrays and hashes) in order to make it more manageable
- Similarly, you can also organize your *code* into logical, related units
- As you write code, you may find yourself frequently repeating a set of statements in order to accomplish a task
- In such cases, you will likely want to group those statements into a function, or ***subroutine***.

Why Subroutines?

- With simpler scripts, separating groups of statements by white space may be enough
- However, as scripts become more complex, numerous lines will be increasingly difficult to read, understand, and maintain.
- Also, it may become tedious to repeatedly type the same several lines of code.
- Creating subroutines allows you to make your code more ***organized*** and ***concise***.

What Is a Subroutine?

- At the most basic level, a **subroutine** is a *named block of code* that *accomplishes a task*
- When a subroutine is *invoked*, the flow of control jumps to the subroutine and executes its code
- When complete, the flow *returns* to the place where the subroutine was called and continues
- The invocation may or may not *return a value*, depending on how the subroutine is defined

How do we make a subroutine?

- Subroutines in Perl have three parts:
  1. The *declaration* **`sub`**
  2. The *name* of the subroutine
     - The name may contain a list of parameters
     - Make the name mean something to you
  3. A *block of code* enclosed in curly braces **`{ actions }`**
- The subroutine can contain any code that the main routine can contain. It can even call other subroutines.

Example Subroutines

```
sub getTimestampEpoch {
    # body...
}
```

```
sub getIPAddress {
    # body...
}
```

Components:

1. Keyword "sub"
2. Name of subroutine
3. Code body

- The first example might be used to get a timestamp, in Unix time, for an **auth.log** entry line
- The second could extract an IP address from a log entry line

How do we invoke a subroutine?

- The most common way is to just refer to it by its *name* followed by a *set of parentheses* ***()***

```
$tStamp = getTimestampEpoch($line);
```

- This can call a subroutine defined anywhere in the file.
- If the subroutine is defined prior to its invocation, then the parenthesis can be omitted.

```
$tStamp = getTimestampEpoch $line ;
```

Other ways to call a subroutine

- We can also invoke the subroutine *before* it is defined, if we let the code know it is a subroutine. We can do the by:
  - By including the statement `sub exampleSubroutine;` prior to invoking it
  - Or calling it by `&exampleSubroutine;`
  - You can think of the **&** as a *type declaration* sort of like the `$` , `@` , and `%` symbols
- The *first* method is the more common
- See following…

```
sub getTimestampEpoch;

# lines of code...

@stamps = ();
while (<INFILE>){
     $st = getTimestampEpoch $_ ;
     push @stamps, $st;
}

# more lines of code...

sub getTimestampEpoch {
     # body...
}
```

Subroutine Argument List

- When calling a subroutine, arguments (parameters) can be passed in -- via an array, in parentheses, following the subroutine name.
- Arguments are passed by reference, not by name or value
- Arguments are passed in the array `@_`
- You should not use the array `@_` directly
  - Assign it to another array: `@args = @_ ;`
  - … or to a variable list: `($name, $age, $major) = @_ ;`
- The latter makes assumptions about the contents of `@_`

Return values

- Subroutines return a value. It is the result of the last assignment completed.

```
$value = exampleSubroutine();
```

- You can use a `return` statement in a subroutine
  - As soon as the first return statement is reached, control is returned to the calling program.
  - In other words, subroutine execution finishes *immediately*.
  - A subroutine may have *more than one* return statement
- See following...

Use of a *return* statement

```
sub getIPAddresses {
  @lines = @_ ;
  @ipAddrs = ();
  foreach $line (@lines){
    if ($line =~ /(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})/){
      push @ipAddrs, $1;
    }
  }
  return @ipAddrs;
}
```

Use of a *return* statement

```
sub verifyValidIPAddress {
  @args = @_ ;
  @octets = split (/./, $args[0]) ;
  foreach $octet (@octets){
    if ($octet > 255){
      return 0;
    }
  }
  return 1;
}
```

Variable Scope

- There are two main types of variables.
  - ***Global*** or package variables:
    - Global variables are accessible anywhere in the program
    - The notion of a "package" is a more advanced topic...
  - ***Lexical*** or local variables
    - Only accessible within the block of code where they are defined
    - Defined with a ***my*** statement
- Why do we have two types of variables?
- All variables are *global* by default!
- If using ***strict*** , you can make a variable global with ***our***

Complex Data Structures

- For more information, consult this link:
  http://perldoc.perl.org/perldsc.html
- Sometimes, you want to organize complex data.
  - For example, an "array of arrays" might be handy; however, code such as the following...

```
@arr1 = (1,2,3,4); @arr2 = (5,6,7,8); @arr3 = (@arr1, @arr2);
@arr4 = ((1,2,3,4), (5,6,7,8));
```

  - ...will merely "flatten" the two separate arrays into a single array.
- Fortunately, there is special syntax you may use for this purpose...

Complex Data Structures

- Either of the following gives you an "array of arrays" :

```
@arr1 = (1,2,3,4); @arr2 = (5,6,7,8);
@arr3 = ( [ @arr1 ] , [ @arr2 ] );
```

OR

```
@arr4 = ( [ 1,2,3,4 ] , [ 5,6,7,8 ] );
```

- To access the contents:
  - Single element: `print $arr4[1][2];  # prints 7`
  - One of the inner arrays: `@inner1st = @{ $arr4[0] }; # (1,2,3,4)`

- *Warning:* The syntax can get very complex!

Complex Data Structures

- You can also have a "*hash* of arrays" :

```
%myHash = ( "foo", [ @arr1 ] , "bar", [ @arr2 ] );
```

OR

```
%myHash2 = ( hello => [ @arr1 ] , world => [ @arr2 ] );
```

- To access the contents:
  - Single element:
  - One of the inner arrays:

```
print $myHash2{world}[2];   # prints 7
@myArr = @{ $myHash{foo} }; # (1,2,3,4)
```

- How about "arrays of hashes" and "hashes of hashes"?
- Absolutely! Consult the aforementioned link...