

CS306: Introduction to Perl

Functions and Scoping

U. of Alabama at Birmingham
Dept. of Computer & Information Sciences

Slide 1

Creating a Subroutine / Function

- ```
sub sayhello {
 print "Hello World!\n";
}
```
- You can define this subroutine anywhere in the code, doesn't have to be before you use it.
- The terms subroutine and function are interchangeable in Perl

Slide 2

## Calling A Subroutine

- Use the ()...  

```
sayhello();
```
- Or, if a function has arguments:  

```
print("Hello","World");
```
- Parens are optional around argument lists:  

```
print "Hello","World";
```

Slide 3

## Subroutine Style Tips

- Do the reverse of Java...
- Put your main block of code at the top, and the subroutines below it

Slide 4

## Passing Variables to Functions

- ```
my_sub(2,4);  
sub my_sub {  
    foreach $val (@_) {  
        $total += $val;  
    }  
    print "The total is $total\n";  
}
```
- The argument list goes to @_

Slide 5

Variable-length Argument Lists

- Perl allows you to have any number of items in the argument list – if the function doesn't use all of them, so be it.
- ```
sub sum {
 print "Sum is " . $_[0] + $_[1] . "\n";
}
```
- ```
sum(2,4) --> 6      sum(2,4,6) ---> 6  
sum(2) ---> 2
```

Slide 6

Returning Values From Functions

- ```
print "The total is " . my_sub(2,4,6);
sub my_sub {
 foreach $val (@_) {
 $total += $val;
 }
 return $total;
}
```
- ```
$test1{total} = my_sub(2,4,6);
```

Slide 7

Implicit Return Values

- ```
sub average {
 $avg = ($n1 + $n2) / 2;
 return $avg;
}
```
- If the return statement is omitted, the return value is the return value of the last operation.
- ```
sub average {  
    ($n1 + $n2) / 2;  
}
```

Slide 8

Understanding Variable Scope

- By default, all Perl variables are *global variables*
- Once they are created, they can be accessed anywhere else in the program, including inside subroutines.
- Even if you create the variable in the subroutine, it's still global, and accessible everywhere else in the program now (like \$total two slides back)

Slide 9

Global Variables in Subroutines

- `$n1 = 3; $n2 = 5;`
`sub add {`
 `$sum = $n1 + $n2;`
 `print $sum;`
`}`
`print "$n1 + $n2 = " . add() . "\n";`
- `$n1` and `$n2` created outside the subroutine but are also usable inside the subroutine

Slide 10

A Bit On Packages

- Every piece of code (subroutines, variables, etc...) in Perl is assigned to a particular package. This of a package as roughly equivalent to a class in Java.
- The default package is called "main"
- All variables in a given package are visible from all other areas of that package, including inside subroutines

Slide 11

Accessing Package Variables

- `$color = "red";`
- This can also be referred to by its full name, `$main::color = "blue";`
- use `strict`; will -force- us to use the full name
- `#!/usr/bin/perl -w`
`use strict;`
`$color = red; #ERROR, must say $main::color`

Slide 12

Lexical Variables With my

- Using `$_[0]` and `$_[1]` is still awkward, is there a better way?
- Yes, create variables that are private to the subroutine using the *my* operator
- ```
sub average {
 my ($n1, $n2) = @_;
 my $avg = ($n1 + $n2) / 2;
 return $avg;
} # $n1, $n2, $avg no longer exist
```

Slide 13

## Lexical Variables

- When you use *my*, you create *lexical variables* instead of globals
- Lexical variables are private to the enclosing block. A variable of the same name elsewhere in the program is unaffected.
  - Thus, a global `$n1` and lexical `$n2` can co-exist
- Lexical variables cannot be accessed or modified from outside their enclosing block.

Slide 14

## Scoping Example

- ```
$n1 = 5;  
print "n1 is $n1\n"; # prints 5  
foo(); # prints 2 and 4  
print "n1 is $n1 and n2 is $n2\n"; # 5 and undef  
  
sub foo {  
  my ($n1,$n2) = (2,4);  
  print "n1 is $n1 and n2 is $n2\n";  
}
```

Slide 15

More on my

- *my* can be used anywhere, not just within a subroutine...
- ```
my $string = "foo";
```
- ```
foreach my $num (1..10) {  
  my $squared = $num**2;  
  print "$num squared is $squared\n";  
}
```

Slide 16

When To Use my

- Always
- use strict; will help you remember to do so

Slide 17

Pass By Reference

- Perl always passes variables by reference

```
my $foo = 10;
print $foo;      # 10
my_sub($foo);
print $foo;      # 11
sub my_sub {
    $_[0]++;
}
```

Slide 18

Emulating Pass-By-Value

- ```
my $foo = 10;
print $foo; # 10
my_sub($foo);
print $foo; # 10
sub my_sub {
 my $foo = $_[0]; # local var to hold value
 $foo++; # 11
}
```

Slide 19

## Limitations of Argument Lists

- What's wrong with this code?

```
my(@arr1, @arr2);
@arr1 = (1,2,3);
@arr2 = (4,5,6);
sub process_lists {
 my ($a1, $a2) = @_;
 ...
}
```

Slide 20

## average() for list of unknown length

- It would be better if our average() function could take a variable number of arguments
- sub average {  
    my @nums = @\_;  
    if (@nums == 0) { return undef; } # no nums!  
    my \$total;  
    for my \$num (@nums) { \$total += \$num; }  
    \$total / @nums;  
}

Slide 21

## Order-Insensitive Argument Lists

- sub bake {  
    my (\$name, \$type) = @\_;  
    print "Here's your \$type cake, \$name.\n";  
}
- The order in which you pass the arguments matters here – name first, and then type of cake.
- You can eliminate this requirement with hashes

Slide 22

## Order-Insensitive Argument Lists 2

- sub bake {  
    my %args = @\_;  
    my \$name = \$args{name};  
    my \$type = \$args{type};  
    print "Here's your \$type cake, \$name.\n";  
}

Slide 23

## Order-Insensitive Argument Lists 2

- sub bake {  
    my %args = @\_;  
    my \$name = \$args{name};  
    my \$type = \$args{type};  
    print "Here's your \$type cake, \$name.\n";  
}
- bake(name => 'Fran', type => 'chocolate');  
  bake(type => 'chocolate', name => 'Fran');

Slide 24

## Default Arguments

- sub bake {  
  my %args = @\_;  
  my \$name = \$args{name}  
          || "Valued Customer";  
  my \$type = \$args{type} || "house special";  
  print "Here's your \$type cake, \$name.\n";  
}

Slide 25

## The End

- Any questions?

Slide 26