

CS306: Introduction to Perl

Functions and Scoping

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

Slide 1

Functions and Scoping

Introduction
Return values
Parameters
Scoping
Private Variables

Slide 2

Creating a Subroutine / Function

- ```
sub sayhello {
 print "Hello World!\n";
}
```
- Technically, a function is built-in to Perl and a subroutine is something you define. I will use the terms interchangeably.
- You can define this subroutine anywhere in the code, doesn't have to be above where you use it.

Slide 3

### Calling A Subroutine

- The & symbol indicates a subroutine call....  

```
&sayhello; # prints "Hello, World!"
```
- ...but is almost always optional and absent.

```
&sayhello; # works
&sayhello(); # works
sayhello(); # works
sayhello; # may fail – ambiguous
```

Slide 4

## 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 5

## Default 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 `$sum` on the last slide)

Slide 6

## Return Values

- The *return* operator specifies the return value.
- We could rewrite our `add()` function like this:  
`$n1 = 3; $n2 = 5;`  
`$result = add();`

```
sub add {
 $sum = $n1 + $n2;
 return $sum;
}
```

Slide 7

## Return Values 2

- `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

## Arguments

- Using the `average()` function requires that we put the numbers into `$n1` and `$n2` first. Awkward.
- Using arguments, we can pass in the two numbers directly.
- ```
sub average {  
    $avg = ($_[0] + $_[1]) / 2;  
}
```
- `print average (3, 5);`

Slide 9

Arguments 2

- The arguments passed to a subroutine are stored in the `@_` array; So, `$_[0]` is the first argument, `$_[1]` the second argument, etc...
- `average(2,4,8);`
 - This would work fine. The function doesn't examine `$_[2]`, so this would only average 2 and 4 together, but Perl is happy to pass on all three parameters.
- `average(2);`
 - This will work too – will average 2 and undef. Slide 10

Private 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 11

## 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 12

## 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 13

## 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 14

## When To Use my

- Always
- Ok, you can cheat in tiny scripts, but if your entire source code doesn't fit on one screen, you really should be using `my`
- `my` makes programs more maintainable because you are limiting the scope of the variable (and any problems that may be associated with it)

Slide 15

## Variable Length Argument Lists

- 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 16

## 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 17

## 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 18

## 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 19

## 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 20

The End

- Any questions?