

CS306: Introduction to Perl

Section 6: Functions and Scoping

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

Slide 1

Section 6: Functions and Scoping

Introduction
Return values
Parameters
Scoping
Private Variables

Slide 2

Creating a Subroutine / Function

- ```
sub sayhello {
 print "Hello World!\n";
}
```
- You'll see the names *subroutine* and *function* used interchangeably for user-defined functions
- You can define this subroutine anywhere in the code, doesn't have to be above where you use it.

Slide 3

## Calling A Subroutine

- Use the & symbol to indicate a subroutine call
- `&sayhello;` # prints "Hello, World!"
- The & is optional if its obvious you are calling a function...  
`&sayhello;` # works  
`&sayhello();` # works  
`sayhello();` # works  
`sayhello;` # may fail – ambiguous \*

Slide 4

## Variables in Subroutines

- `$a = 3; $b = 5;`  
`sub add {`  
    `$sum = $a + $b;`  
    `print $sum;`  
`}`  
`print "$a + $b = " . add() . "\n";`
- Note that the variables `$a` and `$b` were created outside the subroutine but are also usable inside the subroutine – more on that later

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:  
`sub add {`  
    `$sum = $a + $b;`  
    `return $sum;`  
`}`
- `$a = 3; $b = 5;`  
`$result = add();`

Slide 7

## Return Values 2

- `sub average {`  
    `$avg = ($a + $b) / 2; # $a, $b exist before call`  
    `return $avg;`  
`}`
- If the return statement is omitted, the return value is the return value of the last operation.
- `sub average {`  
    `$avg = ($a + $b) / 2; # or just ($a + $b) / 2;`  
`}`

Slide 8

## Arguments

- Using the average() function requires that we put the two numbers into \$a and \$b 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?
- Create variables that are private to the subroutine using the *my* operator
- ```
sub average {  
    my ($a, $b) = @_  
    my $avg = ($a + $b) / 2;  
    return $avg;  
} # $a, $b, $avg no longer exist
```

Slide 11

Lexical Variables

- In contrast to global variables, when you use *my*, you create *lexical variables*
- Lexical variables are private to the enclosing block. Any other variable of the same name elsewhere in the program is unaffected.
 - Thus, a global \$a and lexical \$a can co-exist
- Lexical variables cannot be accessed or modified from outside their enclosing block.

Slide 12

Scoping Example

- `$a = 5;`
`print "a is $a\n"; # prints 5`
`foo(); # prints 2`
`print "a is $a\n"; # prints 5`

```
sub foo {  
    my $a = 2;  
    print "a is $a\n";  
}
```

Slide 13

A Trickier Example

- `$store = "open";`
`state();`
`closestore();`
`sub state { print "$store\n"; }`
`sub closestore { my $store = "closed"; state(); }`
- prints:
open
open

Slide 14

More on my

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

Slide 15

When To Use my

- Just about always
- You were probably taught globals were bad – they still are.
- Rule of thumb: 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 16

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;
 foreach \$num (@nums) { \$total += \$num; }
 \$total / @nums;
}

Slide 17

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 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?