Subroutines

Deb Triant
CSHL 2015

Subroutines

The same cleanup statements are run for $seq1 and $seq2.

We want to avoid duplication of code.

```
#!/usr/bin/perl

use strict;
use warnings;

my $seq1 = "ac ggTtAa";
my $seq2 = "tTcC aaA tgg";

# clean up $seq1
$seq1 = lc $seq1;  # 1) make it all lower case
$seq1 =~ s/\s//g;  # 2) remove white space

# clean up $seq2
$seq2 = lc $seq2; # 1) make it all lower case
$seq2 =~ s/\s//g; # 2) remove white space

# print cleaned up sequences
print "seq1: $seq1\n";
print "seq2: $seq2\n";
```

Subroutines

- Way to define your own functions
- Blocks of code that you can run from anywhere in your script
- Code resides in one place - only write the code once
- Makes code easier to maintain - useful function rather than many redundant lines of code.
- Reduces chance of introducing errors - just make changes in one place
- Make code easier to read.

How to get data in and out of a subroutine

- When you call a subroutines, you pass it a list of arguments and receive a list of results
- Subroutines usually located below the place where it is used, often at the bottom of your script
- Uses the function ***sub*** to label your subroutine
    - `sub do_calculations`
- Use the function ***return*** to return values
    - `return $calculations`

Building a subroutine

1. Turn the code of interest into a block.
2. Name the block with `sub subroutine_name`
3. Add statements to read the subroutine argument(s) and `return` the subroutine result(s).

```
sub cleanup_sequence {
    # get the sequence argument to the
    # subroutine - note that just like shift gets
    # an argument for your program, shift gets an
    # argument to your subroutine
    my $seq = shift;

    # clean up $seq
    $seq = lc $seq;    # 1) make it all lower case
    $seq =~ s/\s//g;   # 2) remove white space

    # return cleaned up sequence
    return $seq;
}
```

Passing arguments to a subroutine

Arguments can be passed:

1. one at a time with **`shift`**
2. as an array with `@_`

- `@_` a special array created by Perl
- similar to `@ARGV` for program arguments.

Passing arguments with shift:

```
my $sum = do_calculations(2,3);

sub do_calculations {
    # take the first argument
    my $arg1 = shift;
    # take the second argument
    my $arg2 = shift;
    # do some calculations
}
```

Passing arguments with @_:

```
my $sum = do_calculations(2,3);

sub do_calculations {
   #take list of numbers
    my @numbers = @_;
    #do some calculations
}
```

Returning Results

Use **`return`** operator to return results.

Usually you will call return at the end of the subroutine but can call it earlier it to exit the subroutine.

Return a single value.

```
return $single_value; #scalar
```

Return a list

```
return ($variable, “string”, 3); #list
return @array_of_values; #array
```

Location of Subroutines

- Usually at the bottom of the script.
- Allows you to visually separate main program from the subroutines.

Subroutine code using shift

```
#!/usr/bin/perl
use strict;
use warnings;

my $seq1 = "ac ggTtAa";
my $seq2 = "tTcC aaA tgg";

# call cleanup_sequence subroutine for each sequence
$seq1 = cleanup_sequence($seq1);
$seq2 = cleanup_sequence($seq2);
# print cleaned up sequences
print "seq1: $seq1\n";
print "seq2: $seq2\n";

#####################################
#begin subroutine
sub cleanup_sequence {
    my $seq = shift; # get the sequence argument
    $seq = lc $seq;
    $seq =~ s/\s//g;
    return $seq;  # return cleaned up sequence
}

output:seq1:acggttaa
       seq2:ttccaaatgg
```

Subroutine code using @_

```
#!/usr/bin/perl
use strict;
use warnings;

#calling subroutine
my $sum = do_calculations (3, 8, 10);
print "The sum of numbers is $sum\n";

#########################################
#begin subroutine
sub do_calculations {
    my @numbers = @_; #getting list of numbers
    my $sum;
    foreach my $number(@numbers){
    $sum += $number;
return $sum;  #returns sum of numbers
}

output: The sum of numbers is 21
```

Scope

Scope compilation error

```
#!/usr/bin/perl

use strict;
use warnings;

my $x = 100;

if ($x > 10) {
     my $z = 10;
     $x = 30;
     print "x (inside if block): $x\n";
     print "z (inside if block): $z\n";
}

print "x (outside if block): $x\n";
print "z (outside if block): $z\n”;  #line 19
```

```
Global symbol "$z" requires explicit
package name at ./scope.pl line 19.

Execution of ./scope.pl aborted due
to compilation errors.

Variable only recognized within if
loop.
```

Scope compilation errors: declaring variable with "my" within block

- Variables declared inside of a block using "my" only exist inside the block – once the block is finished, they will be destroyed. Block is saying "my variable".

- That's because `$z` was declared inside the if loop (block), so it is only accessible inside that block.

- To fix that error, we need to declare `$z` outside the if block.

Blocks - effects of declaring variable with “my”

```
#!/usr/bin/perl

use strict;
use warnings;

my $x = 100;
my $z = 5;

if ($x > 10) {
     my $z = 10;
     $x = 30;
     print "x (inside if block): $x\n";
     print "z (inside if block): $z\n";
}

print "x (outside if block): $x\n";
print "z (outside if block): $z\n”;
```

Output:
$x (inside of block):30
$z (inside of block):10
$x (outside if block): 30
$z (outside if block): 5

Scope

- Does the program give the expected behavior?
- By declaring "`my $z =10;`" inside the if block, we're creating a new variable called `$z` only accessible within the block.
- This new variable will not modify the outside variable!
- Note that we can create a new `$z` variable inside the block with no problems – if we do it outside, we'll get a warning.

```perl
#!/usr/bin/perl

use strict;
use warnings;

my $x = 100;
my $z = 5;

if ($x > 10) {
    $z = 10;
    $x = 30;
    print "x (inside if block): $x\n";
    print "z (inside if block): $z\n";
}

print "x (outside if block): $x\n";
print "z (outside if block): $z\n";
```

Output:

```
$x (inside if block): 30
$z (inside if block): 10
$x (outside if block): 30
$z (outside if block): 10
```

If we remove "`my`" from that line, the modification to `$z` will show outside the block.

Scope within fasta parser

```
#!/usr/bin/perl

use strict;
use warnings;

my %fasta;

while (my $line = <>){
  chomp $line;
  if ($line =~ /^>/){
    my $header = $line;          #$header declared in if loop
  }
  else {
  $fasta{$header}.= $line;    #$header not recognized
    }                           #within else loop
  }

foreach my $key….
```

Scope within fasta parser

```
#!/usr/bin/perl

use strict;
use warnings;

my %fasta;
my $header; #need to declare $header outside of while loop

while (my $line = <>){
  chomp $line;
  if ($line =~ /^>/){
    $header = $line;
  }
  else {
  $fasta{$header)  .= $line;
    }
  }

foreach my $key….
```