

Subroutines

```
#!/usr/bin/perl

use strict;
use warnings;

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

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

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

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

Problems With This Code

- The same cleanup statements are run for `$seq1` and `$seq2`.
- Duplication of code; remember DRY (Don't Repeat Yourself)
- Subroutines to the rescue.

Subroutines

- Blocks of code that you can call in different places.
- Code resides in one place.
- Only need to write the code once.
- Easier to maintain.
- Take arguments and return results.
- Make code easier to read.
- Like a mini-program within your program.

Creating a Subroutine

I. Turn the code of interest into a block.

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

Creating a subroutine

2. Label the block with: `sub subroutine_name`

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

Creating a Subroutine

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  
  
    # 1) make it all lower case  
    $seq = lc $seq;  
    # 2) remove white space  
    $seq =~ s/\s//g;  
  
    # return cleaned up sequence  
    return $seq;  
  
}
```


Passing Arguments to a Subroutine

Arguments are passed in `@_` a special array created by Perl.

- Analogous to `@ARGV` for program arguments.

Can use `shift` to take one argument at a time.

```
# take the first argument
my $arg1 = shift;
# take the second argument
my $arg2 = shift;
```

Passing Arguments to a Subroutine

Can copy the contents of `@_` into a list of named variables.

```
my ($arg1, $arg2) = @_;
```

Returning Subroutine Results

Use return operator to return results.

Usually return at the end of the subroutine but can use it to exit the subroutine earlier.

Return a single value.

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

Return a list.

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

Returning Subroutine Results

Return an empty list or undef depending on context.

```
return; #empty list or undef
```

Calling a Subroutine

Calling our subroutine is just like calling an existing built-in Perl function.

```
my $result = my_sub($arg1, $arg2, $arg3, ...);
```

Location of Subroutines

Usually at the bottom of the script.

Allows you to visually separate main program from the subroutines.

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

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

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

sub cleanup_sequence {
    # get the sequence argument
    my $seq = shift;
    # cleanup $seq
    # 1) make it all lower case
    $seq = lc $seq;
    # 2) remove white space
    $seq =~ s/\s//g;
    # return cleaned up sequence
    return $seq;
}
```

Scope


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

```
use strict;  
use warnings;
```

```
my $x = 100;  
my $y = 20;
```

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

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

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

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

Blocks

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

Any time we see `{ }`, we're creating a block.

Blocks are like boxes that have one way mirrors – you can see outside the box from inside, but not inside the box from the outside.

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

Blocks

Variables declared inside of a block only exist inside the block – once the block is finished, they will be destroyed.

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

```
use strict;  
use warnings;
```

```
my $x = 100;  
my $y = 20;  
my $z = 5;
```

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

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

Output:

```
$x (inside of block):30  
$y (inside of block): 20  
$z (inside of block):10  
$x (outside if block): 30  
$y (outside if block): 20  
$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.

Scope

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

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

```
use strict;  
use warnings;
```

```
my $x = 100;  
my $y = 20;  
my $z = 5;
```

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

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

Output:

```
$x (inside if block): 30  
$y (inside if block): 20  
$z (inside if block): 10  
$x (outside if block): 30  
$y (outside if block): 20  
$z (outside if block): 10
```

Positional vs Named Args

```
say 'pos1  = ', decorate_pos(['', 'foo', '']);
say 'pos2  = ', decorate_pos('', 'foo', '');
say 'name2 = ', decorate_named(str => 'foo', left => '(', right => ')');
say 'name1 = ', decorate_named(str => 'foo');
```

```
sub decorate_pos {
  my ($left, $str, $right) = @_;
  $left  ||= '>>>';
  $right ||= '<<<';
  return join(' ', $left, $str, $right);
}
```

```
sub decorate_named {
  my %args = @_;
  $args{'left'}  ||= '>>>';
  $args{'right'} ||= '<<<';
  return join(' ', $args{'left'}, $args{'str'}, $args{'right'});
}
```

```
pos1  = [ foo ]
pos2  = >>> foo <<<
name1 = >>> foo <<<
name2 = ( foo )
```


Positional vs Named Args

```
$ 04-kmer-count.pl AACATAG
```

```
$ 04-kmer-count.pl AACATAG 4
```

```
$ 04-kmer-count.pl mouse.fa
```

```
$ 04-kmer-count.pl mouse.fa 8
```

```
$ 04-kmer-count.pl -s AACATAG
```

```
$ 04-kmer-count.pl -s AACATAG -k 4
```

```
$ 04-kmer-count.pl -f mouse.fa
```

```
$ 04-kmer-count.pl -f mouse.fa -k 8 \  
-o outfile -m 5
```