Perl's `system()` function runs an external command from inside a script. The `system` function returns an error code value, with zero normally meaning that the command executed without error. Run `perldoc –f system` for more info.

**Example:**

```perl
# Run blastn command for 2 sequences
#(no blast database!)
$command = "blastn -query seqA.fa -subject seqB.fa -out AB.bln";
$return_code = system($command);
if ($return_code) {
    print "blastn returned $return_code \n";
}
More about **if**/**while** conditions

- In the last **blastn** example, the return code was checked as follows:

```php
# Run blastn command
$cmd = "blastn -query seqA.fa -subject seqB.fa -out AB.bln";
$return_code = system($cmd);
if ($return_code) {
    print "blastn returned $return_code \n";
}
```

- Conditions in **if**/**while** tests evaluate to 0 or 1
  - (0 = False, 1 = True)

- **if**($return_code) is the same as **if**($return_code != 0)
  - But NOT the same as **if** (!defined $return_code)
- $return_code was defined by the assignment:
  - $return_code = system($command);
More about `defined` vs `!defined`

- Under what circumstances are variables not defined?
  - `@ARGV` is undefined if no command line arguments are specified when the script is run
  - `$ARGV[1]` is undefined if only ONE command line argument is provided
    - Why?

- `$count` is undefined after a "my" statement with no initialization:
  - `my $count;`
- `$count` IS defined after it gets set to any value, including zero:
  - `$count = 0;`
  - Or
    - `$count = 1;`
  - Or
    - `$count++;`
• If you need your Perl script to capture the output of an external program, use backtics ```` instead of the system function. The backtics surround a command string, and the output of the command is returned as an array of lines.

  • **Backtics are distinct from single quotes!**

• Examples:

  # Run blastn and capture results in array
  @result = `blastn -query seqA.fa -subject seqB.fa`;
  $hit_count = 0;
  foreach $lin (@result) {
    $hit_count++;
    print "$hit_count\t$lin"
  }

  # Run EMBOSS msbar program to mutate $seqfile sequences
  @result = `msbar $seqfile -count 10 -point 4 -stdout -auto`;
  print OUTFILE "@result\n";
More backtics examples

# Run "wc -l" on a group of files, one at a time
$snpdata = $ARGV[0];
@files = glob("$snpdata/*SNP*.csv");
foreach $fil (@files) {
    print "$file has `, `wc -l $fil`, ` lines\n";
}

- If the glob() line is replaced with the following, will the @lsfiles array contain exactly the same items as @files?
  @lsfiles = `ls $snpdata/*SNP*.csv`;

- What line of code could be added to make @lsfiles identical to @files?
The `substr` function

- Recall that the function `substr` can be used to extract a substring from a given string.

- `substr` takes a string argument, starting position, and desired number of characters and returns a string.
  - **Example:**
    - `$dna = "ATGCAGTCATCAGTAGTCA";
    - `$first4 = substr($dna, 0, 4);
    - print "First 4 bases are $first4 \n";

- **What would the following code do?**
  - `$dna = "ATGCAGTCATCAGTAGTCA";
  - `$dna_substr = substr($dna, 2, 6);"
More about the `substr` function

• The `substr` function is very flexible. You can get substrings from a given position to the end of the string, get substrings at the end of a string, and even replace substrings when given a 4th argument.

• Examples:
  • `$dna = "ATGC_GTCA_TCGTAGTCAAAAAAAA";`
  • `$dna_3_to_end = substr($dna, 3);`
  • `$dna_last5 = substr($dna, -5);`
  • `$change_startc = substr($dna, 0, 3, "AUG");`
The index function

- The index function is used to determine the position of a letter or a substring in a string. Arguments are string, substring, starting position. Value returned is position where substring is found, or -1 if substring is not found.

- Examples:

```php
$dna = "CGCAGCAGTAGCTACAGCAGCAGAAA";
$motif = "CAG";
$pos = index($dna, $motif, 0);
print "First occurrence of $motif found at position $pos\n";

# scan entire string
$pos = 0;
while (($pos = index($seq, $motif, $pos)) >= 0) {
    print "Found $motif at $pos\n";
    $pos++;
} #what happens without this?
```
The **split** function

- The function `split` can help parse Tab-delimited, comma-separated, or any data with a separator that can be represented by a pattern.
- `split` takes a split pattern and a string argument and returns an array of pieces of the string.
  - The pattern is delimited by a pair of forward slashes `//`.
  - The parts of the string that matched the pattern are not returned.
  - The original string is not modified.

- Example:
  ```perl
  $seqID = "gi|259144736|emb|FN393060.1|";
  # To split apart on | need to escape with \n
  @parts = split(/\|/, $seqID);
  print "Accession number is $parts[3] \n";
  ```

- How many elements are in the `@parts` array?
The join() function

• Recall that the split function takes a string, splits it on a pattern, and returns an array. The join function takes an array, joins it together with a connector string and returns a new string:

• For the code:

```perl
@items = ("A", "B", "C", "D");
$new_str = join("+", @items);
print "Joined string is: $new_str \n";
```

• The output is:

```
Joined string is: A+B+C+D
```
More about the join function

- The perl `join()` function is complementary to `split()`.

- Example: To build a comma separated string from an array:

  ```perl
  $cstring = join ("","", @array);
  ```

- Note that `join` does not use patterns like `split` does!

- However, you can use more than one character to join elements:

  ```perl
  $word_list = join (" and ", @words);
  $link_str = join (‘…’, @links);
  ```
The Perl `grep` function

- The `grep` function in Perl is similar to Unix `grep`,
  - except that it searches an array and returns a sub-array containing references to elements that matched the specified pattern.
  - Note that the pattern match does not have to occur at the beginning of an element. (We will learn how to do that later.)

```perl
# How many names will be in @capB? In @smallB?
@names = ("Bill K", "Barb C", "Jen A", "Jen B", "Robyn Z");
@capB = grep (/B/, @names);
@smallB = grep (/b/, @names);

# Check for Minus/Plus in BLAST output Strand= lines
@strand = `grep Strand $file.bln`;
@minus = grep (/Minus/, @strand);
@plus = grep (!/Minus/, @strand);
# Would it work to use @plus = grep (/Plus/, @strand)?
```
More about the `grep` function

- The perl `grep()` function searches array elements for matches to a pattern.
  - We will learn more about patterns soon, but for now we'll use simple strings as patterns.
- Examples:
  - To search for array elements containing an upper case A:
    ```perl
    @bigA = grep (/A/, @array);
    ```
  - To search for array elements containing upper/lower case A:
    ```perl
    @anyA = grep (/A/i, @array);
    ```
  - Do you notice any similarities with Unix grep?
  - You can use variables inside a pattern:
    ```perl
    @found = grep (/${search}/, @array);
    ```
Even more about the grep function

• You can also use grep in scalar context to test whether there are any matches. In this case grep returns True or False:

```perl
if (grep (/\$search\/,  \@array)) {
    print "Found \$search in array\n";
} else {
    print "\$search not found in array\n";
}
```

• The `grep()` function can be used with a conditional (if) expression instead of a pattern.

• Example:
  • To find all filenames in an array that are directories:

```perl
@dirs = grep (-d,   \@files);
```
Numeric sorting

• Earlier we learned about the sort function:

```perl
@items = ("Greg Bear", 42, "X", 3.5e-107);
@sorted = sort (@items);
print "The sorted items are @sorted \n";
```

• By default, sort does an alphabetical sort. If we want to sort numerically we have to use the numeric comparison operator `<=>`:

```perl
@nums = (42, 17, 3.5e-107, 4e20, 6.6);
@sorted = sort \$a <=> $b \} @nums;   # sort ascending
print "The numbers low to high are @sorted \n";
```

• For descending numeric sort, exchange `$a` and `$b`:

```perl
@nums = (42, 17, 3.5e-107, 4e20, 6.6);
@sorted = sort \$b <=> $a \} @nums;
print "The numbers high to low are @sorted \n";
```
Perl provides a great deal of flexibility in sorting. The sort function can use a code block or a subroutine as instructions on how to sort items in a list. We've seen code block examples:

```perl
@sorted = sort sub { $a <=> $b } @nums; #sort ascending
print "The numbers low to high are @sorted \n";
```

To sort BLAST results primarily by length of match, then secondarily by e-values, then thirdly by percent identity, we can write a subroutine for sort:

```perl
@sorted = sort bsort @lines;
print "The sorted BLAST results are @sorted \n";
```

How do we think we would use this to “sort” a hash by the values?

```perl
my %hash
my @sortedKeys = sort { $hash{$a} <=> $hash{$b} } keys %hash;
foreach my $k (@sortedKeys) { print "\$hash{\$k} \n"; }
```
• Up to this point we have learned about individual concepts in Perl. We now have enough knowledge to begin putting more complex scripts together, from start to finish.

• We've talked about incremental development and testing, but there are other techniques worth exploring. Two of these are Concept Maps and Pseudocode.

• Concept Maps can help organize data flow and concepts/relationships

• Pseudocode is used to write the logical flow of a script, without worrying about precise syntax
Concept Maps and CmapTools

• From wikipedia.org:
  • Concept mapping is a technique for visualizing the relationships between different concepts. A concept map is a diagram showing the relationships between concepts. Concepts are connected with labeled arrows, in a downward-branching hierarchical structure.

• CmapTools is a freely available package that is very easy to use to build Concept maps
  • Download from http://cmap.ihmc.us/
  • This page uses a Concept Map to explain Cmap Tools!
  • You can create detailed maps that represent complex relationships in a very short time
Concept Map Example: tandem repeats

- **extras**: finds tandem repeats in genomic DNA.
- **verify_repeat**: reads and verifies, writes .ver file.
- **filter_extras**: reads and filters, writes .reps file.
- **find_singles**: reads, writes .sngl files.
- **motif_coords**: calculates avg distances below threshold.
- **avgdist_coords**: writes avg distances in overlapping windows.
- **avgdist_thresh**: writes avg distances below threshold.
- **coords2xml**: translates to XML.
- **extras_summary**: writes summary info.
- **.ext file**
- **.ver file**
- **.reps file**
- **.sngl files**
- **summary info**
• You know the importance of Experimental Design… The same holds true for software design.

• Before you begin writing a script, flesh out the logic using flowcharts or concept maps.

• Before writing your Perl code, write the script logic using pseudocode, a blend of natural language and programming language.
  – Pseudocode focuses on logic without worrying about programming language syntax.

• Pseudocode Example:

  
  ```
  if sequence is valid
    compute GC content and output
    search for promoter sequence
  else
    display error message
  end if
  ```
  

Homework

• Study for Perl Quiz on Thursday

• For more help with Perl functions, see
  • http://perlmeme.org/howtos/perlfunc/
  • You can also check http://stackoverflow.com

• Read Chapter 5 in "Beginning Perl". You may SKIP or SKIM sections:
  • Posix and Unicode Classes
  • Changing Delimiters
  • Inline Comments
  • Inline Modifiers
  • Lookaheads and Lookbehinds