ECOL 553L

Glob, Substr, split, parsing, wrapping

The Filename Matching function glob

- The `glob` function matches a collection of files based on a filename pattern (like * wildcards in Unix). Notice that `glob` returns an array result:

```
$max_hits = 2;
@files = glob("*.bln_m9");
if (scalar(@files) == 0) {
   die "No files match *.bln_m9 \n";
}

foreach $fil  (@files) {
   open (BFIL, $fil) or die "Cannot open $fil\n";
   print "Top $max_hits Hits from $fil:\n";
   $hits_seen = 0;
   while (++$hits_seen < $max_hits) {
      $lin = <BFIL>;
      print  "\t $lin";
   }
   next;
}
```

Better scripting practices...

- Using fixed strings for filenames, numbers, or patterns inside a script is known as hard-coding, and it is not very flexible! In the code below if we want to change `nhits`, we need to edit the script. To work on all files having names ending in `".bln"`, we need to change the script in two places:

```
$nhits = 2;
@files = glob("*.bln_m9 ");
if (scalar(@files) == 0) {
  die "No files match *.bln_m9 \n";
}
foreach $fil  (@files) {
  …
}
```

- We could use a variable to keep the file extension change to one line, but we still have to edit the script each time we want to use a different pattern.

Perl File Tests

- Perl can test file attributes: Does a file exist? Is it a directory? Is it readable? Writable? Empty?

```
if (-e $filename) {
   print  "$filename exists \n";
}
if (-d $filename) {
   print  "$filename is a directory \n";
}
if (-w $filename) {
   print  "$filename is writable \n";
}
if (-z $filename) {  # Zero size
   print  "$filename is empty \n";
}
```

Tips for Testing/Debugging Perl Scripts

- If your script does not work as expected:
  - Be sure that you are running the script that you have saved from the editor, and not another version of it.
  - Resolve all errors reported by the Perl interpreter

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

  - Write and test small segments of code at a time
  - Print the contents of variables, and check for non-printable characters within strings, e.g.:

    ```
    print “str is !$str!\n”;
    ```

  - Check for the existence of array or hash elements, e.g.:

    ```
    if (exists $arr[$i] && exists $hash{‘anopheles’}) {
      print “population size: $arr[$i]”;
      print “hazard: $hash{‘anopheles’}”;
    }
    ```

The `substr()` function

- The function `substr` is very useful and flexible. It extracts a substring of a given string.
- `substr` takes a string argument, starting position, and desired number of characters and returns a string
  - Example:
    - `$dna = "ATGCGTCAGTCGTAGTCA";`
    - `$dna_first10 = substr($dna, 0, 10);`
    - `print "First 10 bases are $dna_first10 \n";`
- How would you change the code above to print only the first 3 bases starting at base 5?

The `split()` function

- The function `split` is often used in parsing Tab-delimited or comma-separated data.
- `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:
    - `$aa = "ATG,CGT,CAG,TCG,TAG,TCA";`

      `# Split apart on commas`
    - `@codons = split(/,/, $aa);`
    - `print "Individual codons are @codons \n";`

- How many elements are in the `@codons` array?

Parsing a Tab-delimited file

- Tab-delimited files are quite common. We can use a File Handle to read lines from these files, and use the split function to split the lines into an array of columns.

```
my $blastfile = $ARGV[0];
open (BFILE, $blastfile)  or
                  die  "Cannot open $blastfile \n";
# Print the first two columns only
while ($line = <BFILE>) {
   @col = split(/\t/, $line);
   print   "$col[0],$col[1]\n";
}
```

- How would you print the last 2 columns?

The special variable $!

- In "Beginning Perl", Simon Cozens uses `$!` with the die function. The `$!` variable contains the error string generated by the operating system.

```
my $blastfile = $ARGV[0];
open (BFILE, $blastfile)  or  die  $!;
open (RFILE, ">$blastfile.summary") or  die  $!;
$nmatch = 0;
while ($line = <BFILE>) {
   $nmatch++;
}
print  RFILE "File $blastfile contains $nmatch matches\n";
close  RFILE;   # Make sure output gets flushed to disk!!
```

- Some people think that Perl code is more readable when these sorts of special variables (`$_`, `$!`) are avoided. If you run across others and want to know what they are, see Appendix B in "Beginning Perl".

More about the diamond operator

- Typically the diamond operator `<>` is used to read one line at a time. However, it is possible to read an entire file into an array of lines:

  -
    ```
    open (IDFIL, $idfile)
              or  die  "Could not open $idfile \n";
    ```

  -
    ```
    @ids = <IDFIL>;
      # each line becomes an element in the @ids array
    ```

- This is one example of Perl behavior that is dependent on **<u>context</u>**: *scalar* context vs *array* context.
- Do not read huge files in all at once. This could cause your script to run out of memory and die!

The Perl `system()` function

- 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:

```
# 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:

```
#  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++;`

Perl backtics ` `

- 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.
    - **<u>Backtics are distinct from single quotes!</u>**
- 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`?

Homework

- SKIM "Beginning Perl", Appendix C: Function Reference
- Continue with homework 5, due October 15th