MCB 5472 Assignment #2: Introduction to Perl part 2
January 29, 2014

This week's goals

1. To show you some more useful data structures and functions in Perl
2. To write some simple scripts to do useful bioinformatics functions
3. To work as groups to design the logic of our software

Arrays

- Recall that strings can be thought of as words
- Extending this analogy, arrays are sentences

```
$string1 = “The”;
$string2 = “brown”;
$string3 = “fox”;
@array = ($string1, $string2, $string3);
# One method to load an array using commas
print “@array”; # returns: “The brown fox”
# One method to output an entire array with values
separated by a single spaces
```

Arrays

- Arrays are not just lists, but ordered lists
  - i.e., everything has its own place (“index”)
  - That place can be specified

```
print $array[0]; # returns “The”
print $array[1]; # returns “brown”
print $array[2]; # returns “fox”
```

- Note: arrays are counted from position zero, not one
- Question: why is `$array[2]` written as a string?

Fun with arrays

```
push (@array, “genus Vulpes”);
  print “@array”; # returns “The brown fox genus Vulpes”
pop (@array);
  print “@array”; # returns “The brown fox”
  print $array[0]; # returns “The”
shift (@array);
  print “@array”; # returns “brown fox”
  print $array[0]; # returns “brown”
unshift (@array, “The”);
  print “@array”; # returns “The brown fox”
  print $array[0]; # returns “The”
```

More fun with arrays

```
scalar @array; # returns “3”
push (@array, “genus Vulpes”);
scalar @array; # returns “4”
pop (@array);
scalar @array; # returns “3”
shift (@array);
scalar @array; # returns “2”
unshift (@array, “The”);
scalar @array; # returns “3”
```

Note: the output of the `scalar @array` will always be one greater than the last index of that array, because indices start at “0”

Assigning data to arrays using `split`

Strings can be converted to arrays using split

```
$string = “The brown fox”;
@array = split “ “, $string;
```

Logically: split at this pattern, this string

```
print $array[0]; # returns “The”
print $array[1]; # returns “brown”
print $array[2]; # returns “fox”
```

Split at every character:

```
@array = split “”, $string;
```

You can split using all sorts of complicated patterns, but save that thought…

Joining multiple arrays and strings

Strings:

```
$string1 = $string2.$string3;
$string1 = $string2.“something”;
```

Arrays:

```
@array1 = (@array2, @array3);
```

Fancy conversion of arrays to strings:

```
$string = join “ ** ”, @array;
print $string; # returns “The ** brown ** fox”
```

`foreach` loops

- Recall `while` used when reading files line by line
- `foreach` is the analogous control structure for arrays

```
foreach $word (@array){
     print $word, “ “;
     # joins text and variables
} # returns “The brown fox “
```

Finer control – for loops

Literal logic:

(1) start with this condition
(2) so long as this qualification is true
(3) change the initial condition each time in this way

```
for ($count = 0; $count <= 2; $count = $count + 1){
     print $array[$count], “ “;
} # returns “The brown fox “
```

Finer control – for loops

```
for ($count = 0; $count < scalar @array; $count = $count + 2){
       print $array[$count], “ “;
} # returns “The fox “ i.e. $array[0] and $array[2]
```

```
for ($count = scalar @array - 1; $count >= 0; $count = $count - 2){
       print $array[$count], “ “;
} # returns “fox The “ i.e. $array[2] and $array[0]
```

Questions

- What is the difference between the values given by `for` and `foreach`?
- When would you use `for` and `foreach`?

@ARGV, the input array

- @ARGV is a special array that contains input from the command line

```
[jlklassen@bbcsrv3 ~]$ perl test.pl input.in output.out
      print "$ARGV[0] $ARGV[1]";
            # returns "input.in output.out"
      open (INFILE, $ARGV[0]) or
            die "Cannot open $ARGV[0]";
      open (OUTFILE, ">$ARGV[1]") or
            die "Cannot open $ARGV[1]";
```

This is the most basic way to pass commands to your scripts

Questions about arrays?

Hashes

- If strings are words and arrays are sentences, hashes can be thought of as tables

Keys	Values
key1	value1
key2	value2
key3	value3

- i.e., if you specify the hash (table) name and the name of a key you will return the value corresponding to that key

An example hash

```
%lotr = (
    hero     => "Frodo",
    villain  => "Gollum",
    sidekick => "Sam",
);
```

Keys: `hero`, `villain`, `sidekick`

Corresponding values: `Frodo`, `Gollum`, `Sam`

Getting your values back out

```
print $lotr{hero}; # returns "Frodo"
print $lotr{villain}; # returns "Gollum"
print $lotr{sidekick}; # returns "Sam"
```

Can assign values similarly

```
$lotr{good_wizard} = "Gandalf";
$string = "bad_wizard";
$lotr{$string} = "Saruman";
# keys can be variables
```

Where are my keys?

- An unordered array of hash keys can be generated using the `keys` function

```
@characters = keys %lotr;
print "@characters";
# returns something like "hero sidekick villain bad_wizard
good_wizard"
sort @characters;
print "@characters";
# returns "bad_wizard good_wizard hero sidekick villain"
scalar @characters; # returns 5
scalar keys %lotr; # returns 5
```

Questions about hashes?

Regular expressions

- Regular expressions are possibly what sets perl apart from other programming languages
- Perl regular expression syntax has been mirrored widely by other programming languages
- Unfortunately, regular expressions are somewhat of a language unto themselves
  - I will only show you the basics!

What is a regular expression?

- Regular expressions find some pattern in a string
  - These patterns can be as complicated as you want
- Once that pattern is found, you can do things to that pattern or the string in which you found that pattern
  - e.g., if a line starts with the character “>”, skip it
  - e.g., if the string has the character “A”, replace it with “a”

Regular expression syntax

- The pattern to be matched is bordered by forward slashes, e.g., `/fox/`
- recall split:

```
$string = “The brown fox”;
@array = split “ “, $string;
@array = split / /, $string; # same thing
```

Match a pattern in a string

- use the assignment operator “`=~`”

```
if ($string =~ /fox/){
     print “found the fox”;
}
```

Match a pattern in a string

```
foreach $word (@array){
     if ($word =~ /fox/){
          print “found the fox”;
     }
}
# will print found the fox only at the
third array element
```

Find and replace strings

- Modified matching syntax:
  `s/[something]/[something_else]/`

```
foreach $word (@array){
     $word =~ s/fox/bear/;
}
print "@array"; # returns "The brown bear"
```

- Matching with nothing (i.e., delete)

```
foreach $word (@array){
     $word =~ s/f//;
}
print "@array"; # returns "The brown ox"
```

Find and replace characters

- Modified matching syntax:
  `tr/[some_characters]/[other_characters]/`

```
foreach $word (@array){
     $word =~ tr/fx/FX/;
}
print "@array"; # returns "The brown FoX"
```

Special characters (metasymbols)

- Allows you to match spaces
  - `\t` – tab
  - `\n` – UNIX newline character
- Allows you to match character classes
  - `\s` – any whitespace character
  - `\d` – any digit
  - `\w` – any digit, letter, or "_"
- Add "+" to match one or more than instances
  - `\s+` – matches any amount of whitespace
- These work for other commands too

```
$string = join "\t", @array;
# returns "The    brown fox"
```

Global vs. local matching

- By default, replacement matching occurs left to right along a string and stops at the first value found
- You can replace all values by adding `g` to the end of your regular expression

```
$string = "@array"
print $string; # returns "The brown fox
$string =~ s/\s/\t/g;
print $string;
# returns "The     brown     fox";
```

Positional matching

- You can match a value only if it occurs in a specific place
  - "^" – only at beginning of string
  - "$" – only at end of string

```
$string =~ s/^\s+//;
# remove all leading whitespace
$string =~ s/\s+$//;
# remove all trailing whitespace
```

Questions about regular expressions?

Assignment

- There are three questions (on the website)
- Step 1: in groups of 4-5, design the logic of your scripts using pseudocode
  - i.e., develop the logic without worrying about how to actually do it
- Step 2: individually code your scripts
- Step 3: email Jonathan your input files, output files, and scripts as a .zip before next Wednesday's class

Pseudocode example

- pseudocode of last week's question 3c might read:

```
open jonathanklassen_3c.input1
open jonathanklassen_3c.input2
open jonathanklassen_3c.output
for each line of jonathanklassen_3c.input1 {
      print this line to jonathanklassen_3c.output
}
for each line of jonathanklassen_3c.input2 {
      print this line to jonathanklassen_3c.output
}
close jonathanklassen_3c.output
```