ECOL 553

big scripts, REGEX!!!! (yay!)

Writing more powerful scripts

- 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

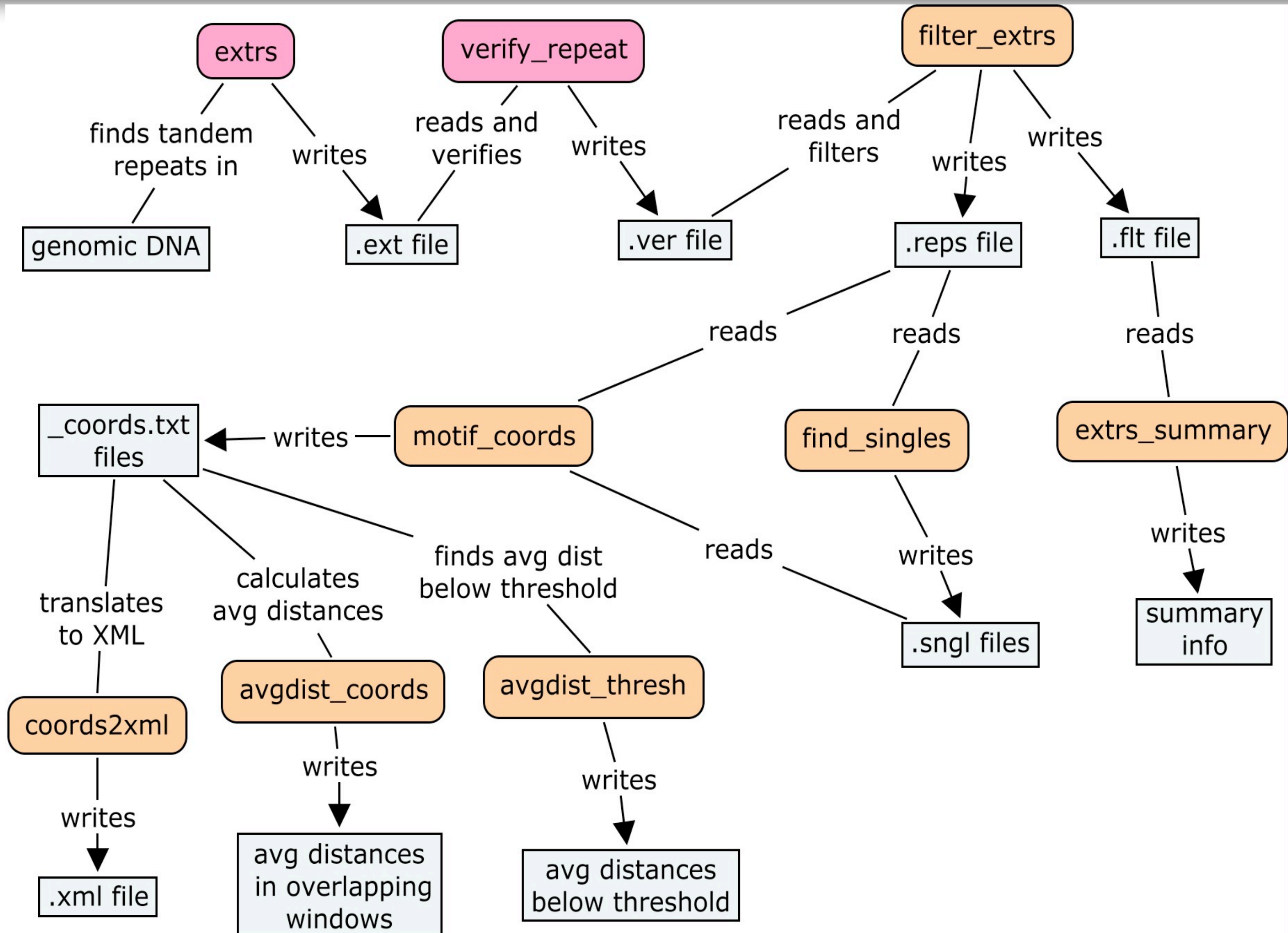
- Pseudocode is used to write the logical flow of a script, without worrying about precise syntax

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 project



Pseudocode: Basic Script Design

- 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
```

The Power of Pattern

- In a homework exercise, we used substr to extract GI numbers from sequence identifiers such as:
  - >gi|123456789|ref|XP_001316127.1| surface antigen BspA-like
  - >gi|234567890|gb|BF978463.1|BF978463 602148868F2 NIH_MGC_62 Homo sapiens cDNA clone IMAGE:4307753 5', mRNA sequence
  - >gi|345678901|emb|Z53267.1| H.sapiens (D9S1833) DNA segment containing (CA) repeat; clone AFMb072zc1; single read, sequence tagged site

- Using substr is not a robust means of getting a GI number from a sequence identifier:
  - GI numbers are not composed of a fixed number of digits
  - Pattern matching can easily recognize any number of digits

- A large number of CAG repeats in a sequence can predict Huntington's disease. Using Perl patterns, it is easy to find these sorts of patterns in a sequence. Can you think of other patterns you might want to look for in DNA or protein sequences?

Pattern Matching and Regular Expressions

- Forward slash characters `//` are commonly used to delimit patterns. The binding operator `=~` applies a pattern search to a string. Modifiers may be specified after the pattern. The simplest patterns are made up of a series of characters:

```
if ($name =~ /James/) { … }
if ($seq  =~  /CAG/i) { … }
```

- Regular expressions include symbols that allow more powerful pattern searches:

```
$name =~  /James|Jim/;   # James OR Jim
# Square brackets [ ] specify character classes
$letters =~ /[A-Z]/;     # match any letter A thru Z
$seq =~ /[^ACGT]/i;      # Not A,C,G or T
(i modifier =>  case insensitive match)
```

More Perl Pattern Matching Details

- pattern matching `($str =~ m/pattern/):`

```
$seq =~ m/CAG/;     # match upper case CAG
$seq =~ m/CAG/i;    # match mixed case CAG
$seq =~ m/[ACGT]/;  # match any one of A, C, G, or T
```

- With the binding operator `=~` the m before the pattern is often omitted, because "match" is the default behavior:

```
$seq =~ /CAG/;      # match upper case CAG
```

- Some character classes:

```
[0-9]               # match one digit
[a-z]               # match one lower case letter
[^0-9]              # match one non-digit.
```

  - Notice that the negation symbol ^ is inside the character class

Pattern Matching Shortcuts and Quantifiers

- Some shorthands for character classes:
  - `\d` A digit. Same as `[0-9]`
  - `\s` A whitespace character.
    - Same as `[ \t\n\r\f]`

- Matching a Pattern zero or more times:
  - `/a?c/` **# Zero or one a's followed by c**
  - `/a*c/` **# Zero or more a's followed by c**
  - `/a+c/` **# One or more a's followed by c**
  - `/a{10,20}c/` **# Ten to twenty a's followed by c**
  - Quantifiers are greedy: they try to match as far to the right as possible.
    - `$str = "caaaaaag"; $str =~ /ca*/` **#matches caaaaaa**

in VIM

- before we found out how to use / to find stuff in a file
- this can take a regex!!

in VIM

- you can also use regex for replacement!
- :%s/<search>/<replace>/g
  - uses \(...\) for groups
  - \1, \2 for references
  - \0 is whole match