http://creativecommons.org/licenses/by-sa/2.0/

COMMONS DEED

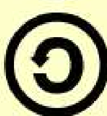
Attribution-ShareAlike 2.0

You are free:

- to copy, distribute, display, and perform the work
- to make derivative works
- to make commercial use of the work

Under the following conditions:

Attribution. You must give the original author credit.



Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

This is a human-readable summary of the Legal Code (the full license).

Disclaimer

An Introduction to Perl for Bioinformatics – Part 1

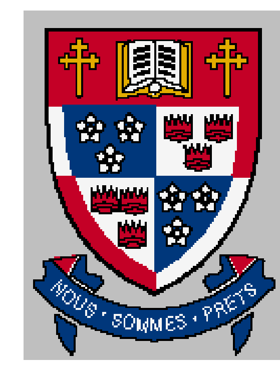
Will Hsiao

Simon Fraser University

Department of Molecular Biology and Biochemistry

wwhsiao@sfu.ca

www.pathogenomics.sfu.ca/brinkman



Outline

- Session 1
  - Review of the previous day
  - Perl – historical perspective
  - Expand on Regular Expression
  - General Use of Perl
  - Expand on Perl Functions and introduce Modules
  - Interactive demo on Modules
- Break
- Session 2
  - Use of Perl in Bioinformatics
  - Object Oriented Perl
  - Bioperl Overview
  - Interactive demo on Bioperl
  - Introduction to the Perl assignment

Today's Goals

- Will have become familiar with a few more advanced programming concepts
  - Regular Expression
  - Functions and Modules
  - Object Oriented Perl
- Will have heard a few common uses of Perl
- Will have learned how Perl can be used in bioinformatics
- Will have discovered Bioperl

Recap from Yesterday

- Which ones below are variables?
  – $sequence, @sequences, 74, ‘I knew this’, %seq_id, “exciting stuff”
- What are functions?
- Which part of the statement below is a function:
  – @sequences = split (/\t/, $genome);
- Other issues?

What does this program do?

```
#!/usr/bin/perl -w

#a mystery subroutine that does something
sub mystery_function{
   my ($seq1, $seq2)=@_;
   my $rDNA = reverse $seq1;
   $seq2 =~ tr/T/U/;
   my $hybrid = $rDNA.$seq2;
   return $hybrid;
}
#body of the main program
$DNA1 = “GATACAATAC”;
$DNA2 = “ATCGTAATCC”;
$answer = mystery_function($DNA1, $DNA2);
print “$answer\n”;
```

"use strict"

```
#!/usr/bin/perl -w
use strict;
#a mystery subroutine that does something
sub mystery_function{
   my ($seq1, $seq2)=@_;
   my $rDNA = reverse $seq1;
   $seq2 =~ tr/T/U/;
   my $hybrid = $rDNA.$seq2;
   return $hybrid;
}
#body of the main program
my $DNA1 = "GATACAATAC";
my $DNA2 = "ATCGTAATCC";
my $answer = mystery_function($DNA1, $DNA2);
print "$answer\n";
```

Effects of “use strict”

- Requires you to declare variables

Correct	Incorrect

`my $DNA;`
`$DNA = “ATCG”;`
or
`my $DNA = “ATCG”;`
	`$DNA = “ATCG”;`

- Warns you about possible typos in variables

No warning	Warning
`my $DNA = “ATCG”;` `$DNA =~tr/ATCG/TAGC/`	`my $DNA = “ATCG”;` `$DAN =~tr/ATCG/TAGC`

Why bother “use strict”

- Enforces some good programming rules
- Helps to prevent silly errors
- Makes trouble shooting your program easier
- Becomes essential as your code becomes longer
- We will use strict in all the code you see today and in your assignment
- **Bottom line: ALWAYS use strict**

Perl – a brief history

- Purpose: "… for scanning arbitrary text files, extracting information from those text files, and printing reports based on that information…"

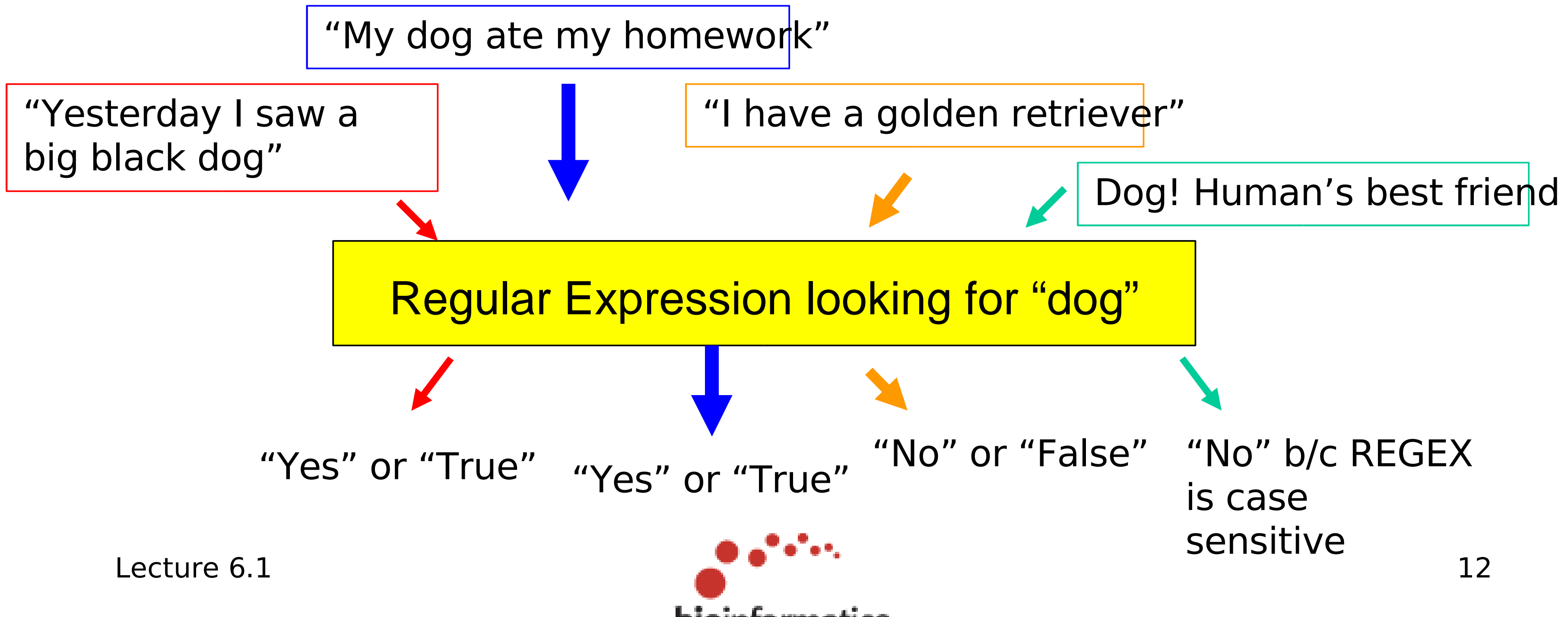
  - from perl manpage
- 1987-Perl 1.0 released
- 1993 – CPAN conceived
- 1995 – Perl 5.000 released
  - Object oriented perl
  - Modules for creating interactive web pages (CGI)
  - Modules for connection to databases (DBI)
- Current stable version of Perl is 5.8.5

How do we manipulate text?

Regular Expression

What is Regular Expression

- REGEX provides pattern matching ability
- Tells you whether a string contains a pattern or not (Note: it's a yes or no question!)



Why need Regular Expression

- Human does this quite well
- But….
- Imagine trying to find all ATG's in the human genome by hand
- Furthermore, imagine trying to find all EcoRI digestion sites (GAATTC) in the human genome

Perl REGEX example

```
my $text = “Bioinformatics Kicks Ass”;
if ($text=~/Kicks/){
  print “The text contains Kicks\n”;
}
```

- =~ is the **binding operator**
  - It says: “does the string on the left contain the pattern on the right?”
- /Kicks/ is my **pattern**
- The matching operation results in a true or false answer

More Regular Expression

- A pattern that match only one string is not very useful!
- We need symbols to represent classes of strings

- REGEX is its own little language inside Perl
  - Has different syntax and symbols!
  - Symbols which you have used in perl such as $. { } [ ] have totally different meanings in REGEX

REGEX Metacharacters

- Metacharacters allow a pattern to match different strings
  - Wildcards are examples of metacharacters
  - /.ick/ will match “kick”, “sick”, “tick”, “stick”, “kicks”, etc.
  - Perl REGEX has much more powerful metacharacters used to represent classes of characters

Types of Metacharacters

- . matches any one character or space except “\n”
- [ ] denotes a selection of characters and matches ONE of the characters in the selection.
  - What does [ATCG] match?
- \t, \s, \n match a tab, a space and a newline respectively
- \w matches any characters in [a-zA-Z0-9]
- \d matches [0-9]
- \D matches anything except [0-9]

An Example of Metacharacters

V1S 5A6?

/\w\d\D\s\d.[0-9]/

Is it a good pattern for postal code?

What else does it match?

REGEX Quantifiers

- What if you want to match a character more than once?

- What if you want to match an mRNA with a polyA tail that is at least 5 – 12 A's?

"ATG……AAAAAAAAAAAA"

REGEX Quantifiers

"ATG......AAAAAAAAAAA"

/ATG[ATCG]+A{5,12}/

- + matches one or more copies of the previous character
- * matches zero or more copies of the previous character
- ? matches zero or one copy of the previous character
- {min,max} matches a number of copies within the specified range

REGEX Anchors

- The previous pattern is not strictly correct because:
  - It'll match a string that doesn't start with ATG
  - It'll match a string that doesn't end with poly A's
- **Anchors** tell REGEX that a pattern must occur at the beginning or at the end of a string

REGEX Anchors

- ^ anchors the pattern to the start of a string
- $ anchors the pattern to the end of a string

- `/^ATG[ATCG]+A{5,12}$/`

REGEX is greedy!

- The revised pattern is still incorrect because
  - It'll match a string that has more than 12 A's at the end
- quantifiers will try to match as many copies of a sub-pattern as possible!

```
/^ATG[ATCG]+A{5,12}$/
```

"ATGGCCCGGCCTTTCCCAAAAAAAAAAAAA"

"ATGGCCCGGCCTTTCCCAAAAAAAAAAAAA"

Curb that Greed!

- ? after a quantifier prevensts REGEX from being greed
- note this is the second use of the question mark
- What is the other use of ? in REGEX?

```
/^ATG[ATCG]+?A{5,12}$/
```

"ATGGCCCGGCCTTTCCGAAAAAAAAAAAA"

"ATGGCCCGGCCTTTCCGAAAAAAAAAAAA"

bioinformatics.ca

REGEX Capture

- What if you want to keep the part of a string that matches to your pattern?
- Use ( ) “memory parentheses”

“ATGGCCCGGCCTTTCCGAAAAAAAAAAAA”

```
/^ATG([ATCG]+?)A{5,12}$/
```

REGEX Capture

/^ATG([ATCG]+?)(A{5,12})$/

$1 (first group: [ATCG]+?)

$2 (second group: A{5,12})

- What's inside the first ( ) is assigned to $1
- What's inside the Second ( ) is $2 and so on
- So $2 eq "AAAAAAAAAAAA"

REGEX Modifiers

- Modifiers come after a pattern and affect the entire pattern
- You have seen //g already which does global matching (/T/g) and global replacement(s/T/U/g)
- Other useful modifiers:

//i	make pattern case insensitive
//s	let . match newline
//m	let ^ and $ (anchors) match next to embedded newline
///e	allow the replacement string to be a perl statement

REGEX Demo

- Demonstrate quantifiers
- Demonstrate anchors
- Demonstrate //i
- Demonstrate capture
- Demonstrate the effect of greedy vs. non-greedy
- Demonstrate metacharacters

Other binding operators

- =~ is called the binding operator which "binds" the a string on the left to a pattern on the right
  - E.g. $text =~/PATTERN/
- Two other binding operators: s/// and tr///
  - =~s/// (substitution) substitutes a matched pattern by a string (kind of like the "replace" function in MS Word)
  - =~tr/// (translation) translates a character to another

Summary on REGEX

- REGEX is its own little language!!!
- REGEX is used in some functions (e.g. split)
- Perl REGEX: extremely powerful and fast
- REGEX is one of the main strengths of Perl
- To learn more:
  - Learning Perl (3rd ed.) Chapters 7, 8, 9
  - Programming Perl (3rd ed.) Chapter 5
  - Mastering Regular Expression (2nd ed.)

Common Uses of Perl

- REGEX
  - Complete set of tools for pattern matching text
- System administration
  - Perl scripts can be written to automate many system administration tasks
- CGI.pm
  - Module for designing interactive web pages
- DBI.pm
  - Database Interface – allows communication between all major RDBMS systems (Oracle, MySQL, etc.)

Review on Functions

- How do we “call” a function?

```
my $sum = add (2, 3)
```

Return Value Function Name Parameter list

- Functions can take some input values (parameters) and can return some output values
- You need to assign the return values to a variable in order to use them

More Review on Functions

- Benefits of subroutines
  - Decompose a big problem into smaller, more manageable problems
  - Organize your code
  - Improve code reuse
  - Easier to test and debug your code

```
sub add {
#some code that adds numbers here
#return the sum
}
```

What are Modules

- a “logical” collection of functions
- Each collection (or module) has its own “name space”
  - Name space: a table containing the names of variables and functions used in your code

Why is name space important

Package: SEQanalysis_DNA

```
$DNA = "ATGAATACTACTAT..."

$polyAtail = "AAAAAAAAAA"

sub Revcom{}
#reverse complement sequence

sub concat{}
#concatenate two DNA sequences
```

Package: SEQanalysis_Prot

```
$exon1 = "MEDAVRSKNTMI"

$exon2 = "RSVADEGFLSMIRQH"

sub findmotif{}
#find a peptide motif

sub concat{}
#concatenate two exon
  sequences
```

SEQanalysis_DNA::concat

SEQanalysis_Prot::concat

Why Use Modules

- Modules allow you to use others' code to extend the functionality of your program
- But, use other people's modules is like going to other people's houses
  - Not everything will be the way you like it
    - Read the module documentation
  - Be nice
    - use a module as it is intended
- In Perl, each module is a file stored in some directory in your system
  - E.g. you can find cgi.pm in /usr/lib/… on your system (ask Graeme where it is)

Use Modules

- To use a module:
  - use <modulename>;
- Examples:
  - `use strict;`
  - `use Env;`
  - `use cgi qw(:standard);`
- To find out where modules are installed : `perl -V`
- To find out what standard modules are available: `perldoc perlmodlib`

Module Demo

- Demonstrate perldoc as a method to read module documentation
- Demonstrate the difference before and after using a module (use strict and use Env)
- Demonstrate the perl –V and an example of directory structure of modules

Where to find modules

- CPAN: Comprehensive Perl Archive Network
- Central repository for Perl modules and more
- "If it's written in Perl, and it's helpful and free, it's probably on CPAN"
- http://www.perl.com/CPAN/

- To install modules from CPAN
  - `perl -MCPAN -e "install 'Some::Module'"`
    - Module dependency is taken cared of automatically
    - You'll (usually) need to be root to install a module successfully
    - For details see your notes

CPAN Web Demo

- Demonstrate how to search for a module and how to access the online documentation
- We'll use Getopt::Long as an example

Interactive Demo on Getopt::Long

- Open your laptop!
- Open a terminal window
- Type `cd ~/perl_two`
- Type `emacs ./getopt_demo.pl&`
- Let's go over the example together

Summary for Session 1

- Always “use strict”
- Regular Expression is its own language inside Perl
- I encourage you to read the chapters on REGEX in Learning Perl
- A module is a logical collection of functions
- You can find module documentation by using perldoc (command line) or by going online to CPAN

Break