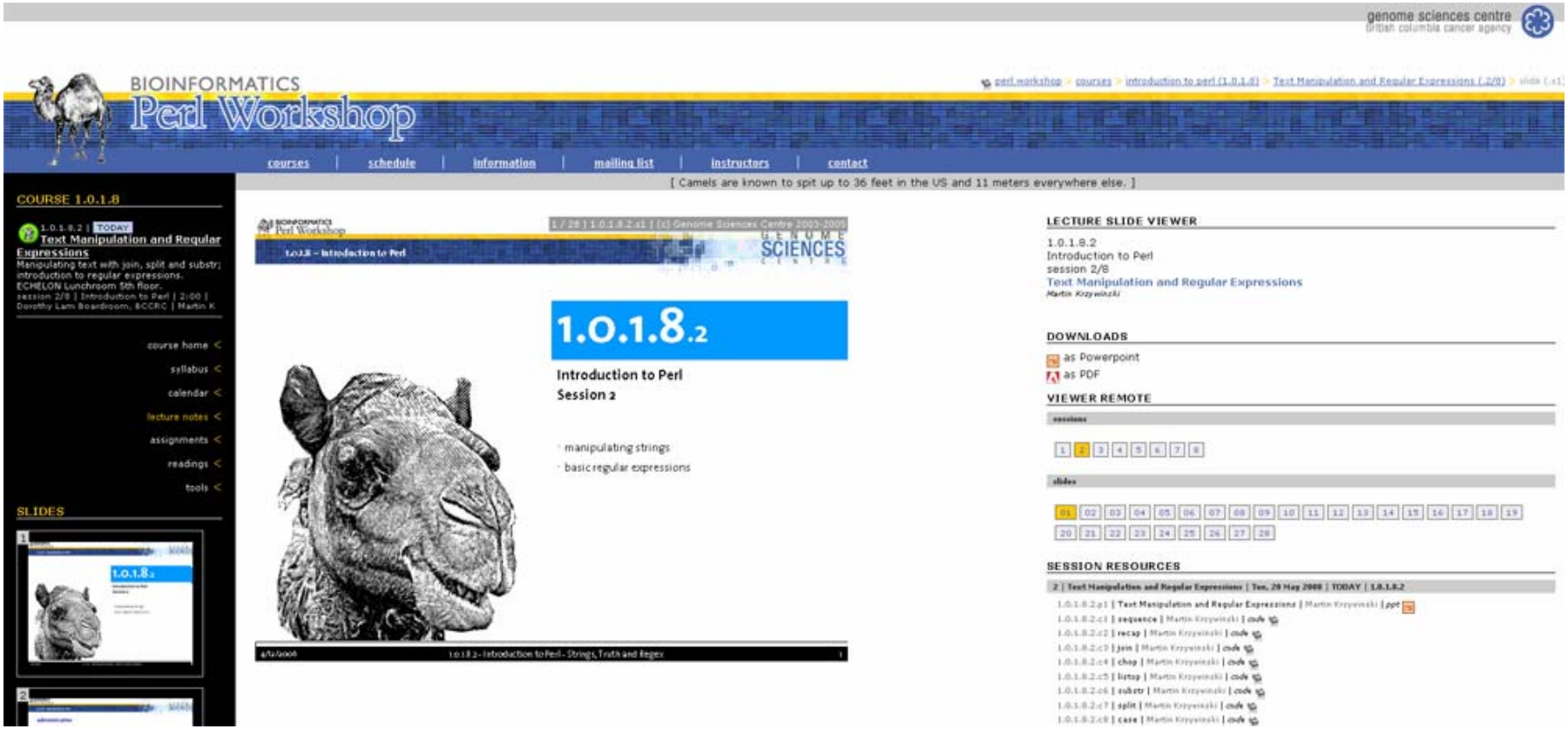
1.0.1.8.2

Introduction to Perl
Session 2

- manipulating strings
- basic regular expressions

administrative

- workshop slides are available at mkweb.bcgsc.ca/perlworkshop



Recap

- scalar variables are prefixed by $ sigil and can contain characters or numbers
- Perl interpolates variables in double quotes " " but not in single quotes ' '

double quote operator qq()		single quote operator q()	
$var = 1			
qq(var)	var	q(var)	var
qq("var")	"var"	q("var")	"var"
qq($var)	1	q($var)	$var
qq(${var}2)	12	q(${var}2)	${var}2
qq{\$var}	$var	q{\$var}	\$var
qq/'$var'/	'1'	q/'$var'/	'$var'

- == and eq are the equality test operators for numbers and strings
- *undef* is a special keyword used to undefine a variable

String Manipulation

- manipulating strings in Perl is very easy
- large number of functions help you massage, cut, and glue strings together
- today we will explore how to
  - concatenate strings
  - replace parts of a string
  - determine the length of a string
  - change the case of a string
- I will also introduce regular expressions, which can be used to
  - split a string on a boundary
  - search a string for patterns

Concatenating Strings

- we've already seen one way to concatenate values of scalar variables
  - concatenation operator .
  - create a new variable and use interpolation to place strings in appropriate spot

```
$x = "baby";
$y = "is";
$z = "crying";
$s = " ";

$phrase = $x . " " . $y . " " . $z;
$phrase = $x . $s . $y . $s . $z;
$phrase = qq($x $y $z);
$phrase = qq($x$s$y$s$z);
```

Concatenating with join

- use perldoc –f FUNCTION to learn about a built-in Perl function

```
> perldoc -f join

join EXPR,LIST

Joins the separate strings of LIST into a single string with fields
separated by the value of EXPR, and returns that new string.  Example:

    $rec = join(':', $login,$passwd,$uid,$gid,$gcos,$home,$shell);

See split.
```

- given a list of strings, you can glue them together with a given string using join

```
($x,$y,$z,$s) = (“baby”,”is”,”crying”,” “);

$phrase = join(“ “,$x,$y,$z);
$phrase = join($s,$x,$y,$z);
```

Concatenating with join

- join takes a list as an argument
  - first element is the glue
  - all other elements are the things to be glued

```
($x,$y,$z,$s) = (“baby”,”is”,”crying”,” “);

$phrase = join(“ “,$x,$y,$z,”-”,”make”,”it”,”stop”);  baby is crying - make is stop

$phrase = join(“ “,1,”+”,1,”=“,2);                    1 + 1 = 2
```

- we’re drowning in double quotes here
  - we’re creating a list of strings and need to delimit each string with “ “ or qq( )

```
(“babies”,”cry”,”a”,”lot”);   # noisy syntax
(babies cry a lot);           # ERROR - barewords
```

Word List Operator qw( )

- qw(STRING) splits the STRING into words along whitespace characters and evaluates to a list of these words

```
$x = “camels”;
$y = “spit”;
$z = “far”;

... or

($x,$y,$z) = qw(camels spit far);
```

- no quotes are necessary
- qw( ) does not interpolate

```
$num = 3;
($w,$x,$y,$z) = qw($num camels spit far);
print “$w $x $y $z”;                          $num camels spit far
```

Use qw( ) for Concise Assignment

- assigning values to multiple variables on one line is a good idea
  - terse
  - easy to read
  - even better if the variables are semantically related

```
($w,$x,$y,$z) = qw(blue 1 10$10 5.5);          $y → 10$10
```

- we haven't seen lists formally yet, but we are using them here
  - a list is an ordered set of things (e.g. the soup)
  - an array is a variable which holds a list (e.g. the can)
  - the distinction is important because we can use lists without creating array variables

```
evaluates to a list ← qw(blue 1 10$10 5.5);

($w,$x,$y,$z) ← expects a list
```

Extracting Parts of a String

- substr is both an interesting and useful function
  - it demonstrates Perl's flexibility because it can be used as an l-value
  - l-value → you can assign the output of substr values
  - substr takes 2-4 arguments and behaves differently in each case
- strings are o-indexed – first character in the string is indexed by 0 (zero)

```
 $string = "soggy vegetables in the crisper";
           ||||||||||||||||||||||||||||||
+'ve index→012345678901234567890123456789
           109876543210987654321098765432←  -'ve index
```

- substr(STRING,OFFSET) returns the part of the STRING starting at OFFSET

```
$substring = substr($string,6);    soggy vegetables in the crisper
$substring = substr($string,0);    soggy vegetables in the crisper
$substring = substr($string,-1);   soggy vegetables in the crisper
$substring = substr($string,-7);   soggy vegetables in the crisper
```

Extracting Parts of a String

- substr(STRING,OFFSET,LEN) extracts LEN characters from the string, starting at OFFSET

```
 $string = "soggy vegetables in the crisper";
           ||||||||||||||||||||||||||||
+'ve index→01234567890123456789012345678 90
           10987654321098765432109876543 21← -'ve index
```

```
$substring = substr($string,6,10);  soggy vegetables in the crisper
$substring = substr($string,6,100); soggy vegetables in the crisper
$substring = substr($string,-3);    soggy vegetables in the crisper
$substring = substr($string,-3,1);  soggy vegetables in the crisper
$substring = substr($string,-3,2);  soggy vegetables in the crisper
$substring = substr($string,-3,3);  soggy vegetables in the crisper

$substring = substr($string,6,5);   soggy vegetables in the crisper
$substring = substr($string,6,-5);  soggy vegetables in the crisper
$substring = substr($string,1,-1);  soggy vegetables in the crisper
```

Determining the Length of a String

- length(STRING) returns the number of characters in the string
  - this includes any special characters like newline
  - escaped characters like \$ count for +1

```
 $string = “soggy vegetables in the crisper”;
           |||||||||||||||||||||||||||||
+’ve index→01234567890123456789012345678 90
           10987654321098765432109876543 21← -’ve index
```

```
$len = length($string);     31
```

Replacing Parts of a String

- substr( ) returns a part of a string

```
$substring = substr($string,0,5);   soggy vegetables in the crisper
```

- substr( ) is also used to replace parts of a string

```
substr($string,0,5) = “very tasty”;   very tasty vegetables in the crisper
```

- substr(STRING,OFFSET,LEN) = VALUE replaces the characters that would normally be returned by substr(STRING,OFFSET,LEN) with VALUE
  - VALUE can be shorter or longer than LEN – the string shrinks as required

```
substr($string,0,5) = “no”;      no vegetables in the crisper
substr($string,0,5) = “tasty”;   tasty vegetables in the crisper
```

More on substr( )

- instead of assigning a value to substr( ), use the replacement string as 4th arg

```
substr($string,0,5) = “no”;       no vegetables in the crisper
substr($string,0,5,”no”);         no vegetables in the crisper

$prev = substr($string,0,5,”no”);    no vegetables in the crisper
                                     $prev = “soggy”
```

- the 4 arg version of substr( ) returns the string that was replaced

```
$x = “i have no food in my fridge”;
$y = substr($x,0,length($x),”take out!”);

$x → ?
$y → ?
```

Changing Case

- there are four basic case operators in Perl
  - lc – convert all characters to lower case
  - uc – convert all characters to upper case
  - lcfirst – convert first character to lower case
  - ucfirst – convert first character to upper case

```
$x = “federal case”;

$y = uc $x;           FEDERAL CASE
$y = ucfirst $x;      Federal case
$y = lcfirst uc $x    fEDERAL CASE
```

Converting Case Inline

- convert case inline with \U \L \u \l
  - \L ~ lc \U ~ uc
  - \l ~ lcfirst \u ~ ucfirst
  - \E terminates effect of \U \L \u \l

```
$x = "\Ufederal case";      FEDERAL CASE
$x = "\Ufederal\E case";    FEDERAL case
$x = "\ufederal \ucase";    Federal Case


$y = qq(\U$error\E $message);
```

Regular Expressions

- a regular expression is a string that describes or matches a set of strings according to syntax rules
- Perl's match operator is m/ / (c.f. qq/ / or q/ /)
  - the m is frequently dropped, and / / is used
- to bind a regular expression to a string =~ is used
  - we will later see that m/ / may be used without accompanying =~

```
$string =~ m/REGEX/          $string =~ /REGEX/
```

- you must think of =~ as a binary operator, like + or -, which returns a value

```
see this                          think this

$string =~ m/REGEX/       →       =~($string,REGEX)
```

Regular Expressions

- regular expressions are made up of
  - characters – literals like the letter "a" or number "1"
  - metacharacters – special characters that have complex meaning
    - character classes – a single character that can match a variety of characters
    - modifiers – determine plurality (how many) characters can be matched (e.g. one, more than one)
    - *and others*
- we'll start slow and build up a basic vocabulary of regular expressions
- commonly the following paradigm is seen with regular expressions

```
if ($string =~ /REGEX/) {
  # do this if REGEX matches the $string
} else {
  # do this, otherwise
}
```

- remember that =~ is a binary operator – it will return true if a match is successful

Regular Expressions

- the most basic regular expression is one which contains the string you want to match, as literals

```
$string = "Hello world";
if ($string =~ /Hello/) {
  print "string matched";     ← a match is made in this case
} else {
  print "no match";
}
```

- regular expressions are case sensitive, unless / /i is used
  - i is one of many flags that control how the REGEX is applied

```
$string = "Hello world";
if ($string =~ /hello/i) {
  print "string matched";     ← a match is made in this case
} else {
  print "no match";
}
```

Regular Expressions – Character Classes

- two commonly used character classes are . and [ ]
  - . means "any character"
  - [ ] means "any of these characters", e.g. [abc] will match either a **or** b **or** c, **not** ab or abc
- when used in isolation these classes match a single character in your string

```
$string = "hello world";
                              match?              matched by class
$string =~ /hello/            YES
$string =~ /HeLLo/i           YES
$string =~ /hell./            YES                 o
$string =~ /hell[abc]/        NO
$string =~ /hell[aeiou]/      YES                 o
$string =~ /hel/              YES
$string =~ /hel[lo]/          YES                 l
$string =~ /hel[lo]o/         YES                 l
$string =~ /he[ll]o/          NO
```

- [ ] works with a range
  - [a-z], [c-e], [0-9]

Three Ubiquitous Character Classes

- \d – any digit
  - equivalent to [0123456789] or [0-9]
- \w – any alphanumeric character or _
  - equivalent to [a-zA-Z0-9_]
- \s – any whitespace

```
regex              matches if string contains...

/\d\d\d/           three digits in succession
/1\d2/             1 followed by any digit followed by 2
/\d\s\d/           a digit followed by a whitespace followed by a digit
/[aeiou].[aeiou]/  a lowercase vowel followed by any character followed by lowercase vowel
/[aeiou][1-5].B/i  a vowel followed by any digit in the range 1-5 followed by any character
                          followed by B or b (case insensitive match)
```

```
$string = “hello”
$string =~ /[hello]/  → ?
```

Splitting a String

- split is used to create a list from a string, by splitting it along a boundary
  - reverse of join, which takes a list and glues elements together using a delimiter

```
join   qw(a b c) → “a b c”
split  “a b c”   → qw(a b c)
```

- split takes a regular expression to act as the boundary
  - split(/REGEX/,$string)

```
$string = “once upon a camel”;

($a,$b,$c,$d) = split(/\s/,$string) # split along a single white space

$string = “1-2-3-4”;

($a,$b,$c,$d) = split(/-/,$string)  # split along hyphen → (1,2,3,4)
```

Splitting Along Spaces

- because whitespace (tab, space) is such a common delimiter, split can be used with " " as a boundary to mean any (positive) amount of whitespace

```
$string = "a b  c    d"

split(" ",$string)  →  qw(a b c d)
```

- note that split(/ /,$string) would split between single spaces

```
$string = "a b  c    d"

split(/ /,$string)  →  "a","b","","c","","","d"

think this a_b_[]_c_[]_[]_d where [] is the empty string
```

Splitting a String

- split is perfect for separating content from delimiters

```
$string = “user:password:flag”;
($user,$password,$flag) = split(“:”,$string);          user password flag

$string = “2_5_100”
($x,$y,$z) = split(“_”,$string);                       2 5 100

$string = “a1b2c”;
($x,$y,$z) = split(/\d/,$string);                      a b c
```

- split creates output (a list) suitable for input to join

```
$string = “a b   c     d          e f g”;
join(“ “, split(“ “,$string) );                        a b c d e f g
join(“-“, split(“ “,$string) );                        a-b-c-d-e-f-g
join(“ and “, split(“ “,$string) );                    a and b and c and d and e and f and g
```

Chop and Chomp

- chomp is a boon and used everywhere
  - it removes a trailing newline (actually the current record separator) from a string
  - it's safe to use because it doesn't touch other characters
  - it returns the total number of characters chomped

```
# $string may have a newline at the end

chomp $string;

# now string has no newline at the end
```

- chop removes the last character (whatever it may be) and returns it

```
$string = "camels";
$x = chop $string;

$string → camel
$x → s
```

Short Script

```
$sequence = undef;
for (1..100) {
  $x = rand();
  if ( $x < 0.25 ) {
    $sequence = $sequence . q(a);
  } elsif ( $x < 0.5 ) {
    $sequence = $sequence . q(c);
  } elsif ( $x < 0.75 ) {
    $sequence = $sequence . q(g);
  } else {
    $sequence = $sequence . q(t);
  }
}

print $sequence;
print “saw poly-A” if $sequence =~ /aaaa/;
print “saw aantt” if $sequence =~ /aa.tt/;
print join(“ + ”, split(“ata”,$sequence));

### output

atcgccaagttggtgtagatatgaggcccgtccattgttcgtacttaacatgtctgtagggatctgcttatacttgtcggagataatacggtggcgcg
saw aantt
atcgccaagttggtgtag + tgaggcccgtccattgttcgtacttaacatgtctgt + gggatctgctt + cttgtcggag +  + cggtggcgcg
```

1.0.8.1.2

Introduction to Perl
Session 2

- you now know
  - all about string manipulation
  - a little about regular expressions
  - use of **split**, **join**, and **chomp**
- next time
  - lists and arrays