

Lecture 5.1

Introduction to Programming



Sohrab Shah and Sanja Rogic

Computer Science Department

University of British Columbia

`sshah@cs.ubc.ca` `rogic@cs.ubc.ca`



creativecommons
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:

 **BY:** **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](#) 

Goals for today

- Introduce major concepts in programming
- Understand the fundamental parts of a Perl program
- Discover the powerful attributes of Perl programming

Outline

- Important concepts to know before learning to program
- Live ‘ensemble’ programming session
 - Hands-on instruction to learn the fundamentals
 - Learn by doing
 - Preparation for tomorrow
- Question and answer

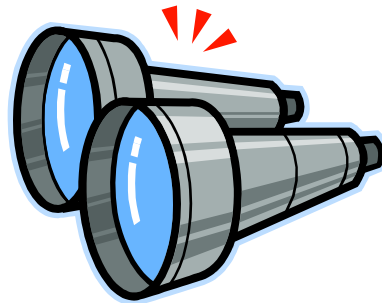


Why program?

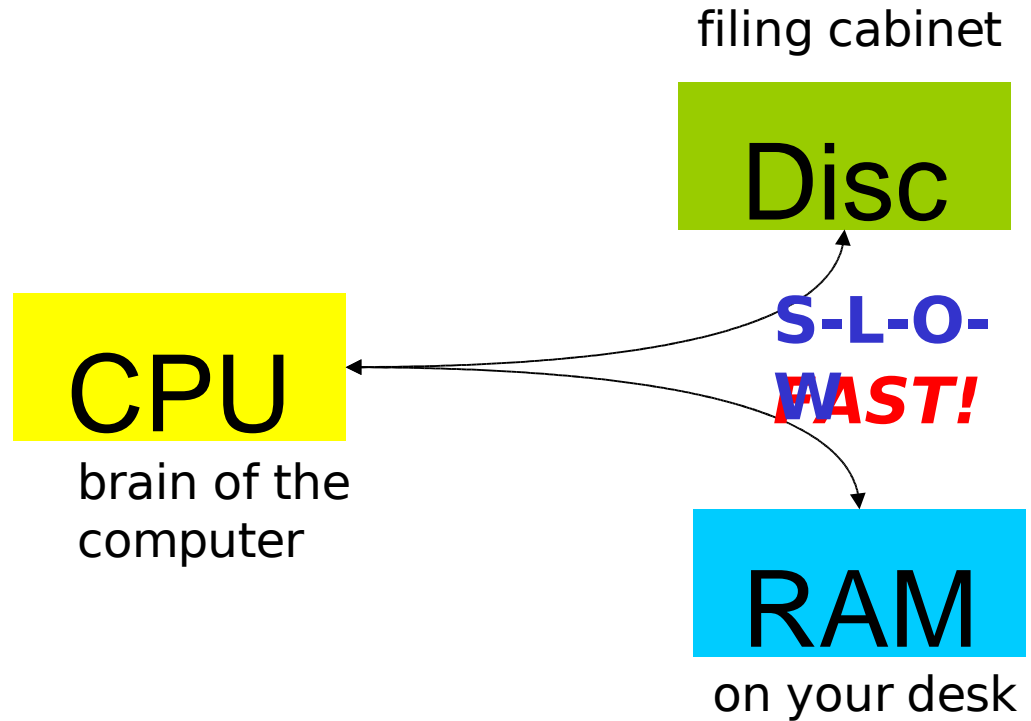
- Programming a computer provides:
 - Accuracy
 - Reduce human error
 - Speed and efficiency
 - Processing the human genome manually is impossible
 - Adding all numbers from 1..1000 is a waste of our time
 - Repetition
 - Same program can be run many times using different input data
 - Automation

When not to program

- There is already a program available that will perform your task
- Look before you code



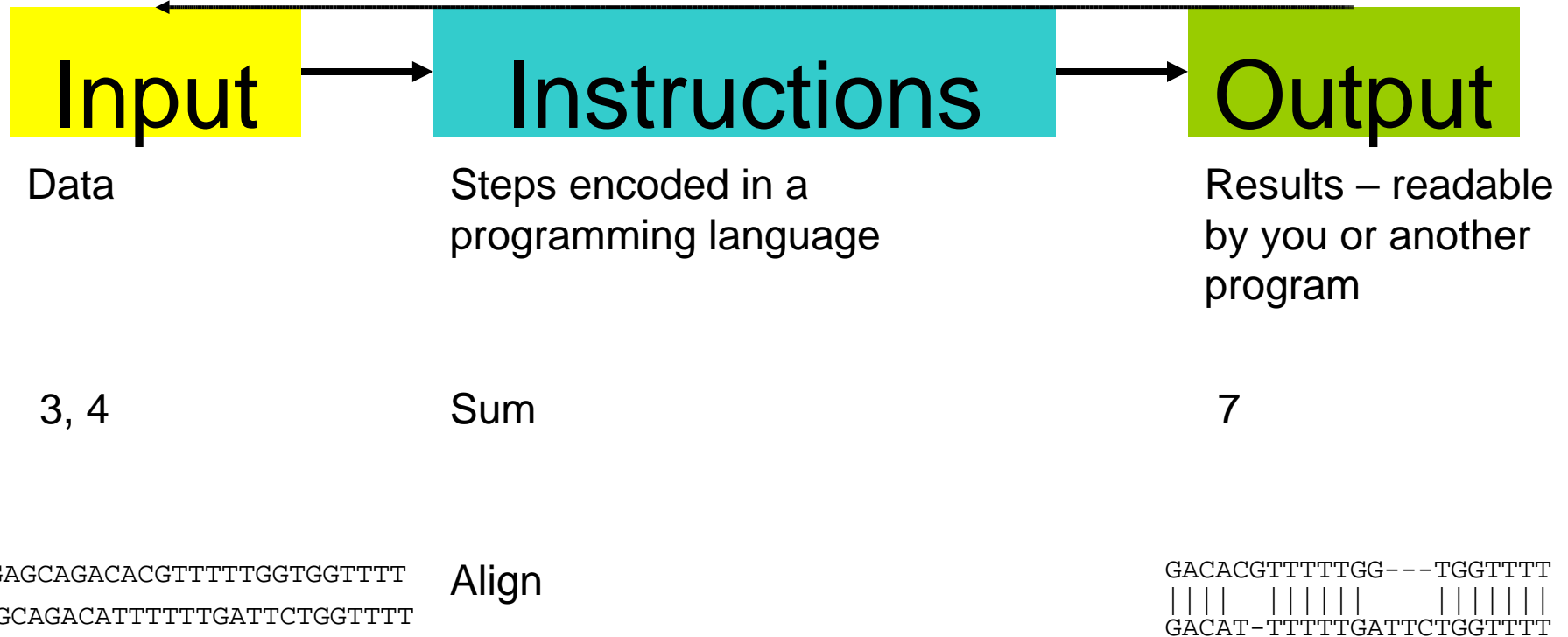
Anatomy of a computer



Before you start...

- First need a task
 - One or more steps that process input to produce some desired output
 - Adding 2 numbers
 - Aligning 2 sequences
- Break down steps of the task into a set of instructions the computer can understand

Anatomy of a program



Programming language

Add 3 and 4!

??



Programming language

Add 3 and 4!

??



```
$ans = 3 + 4;
```

PERL



Programming language

Add 3 and 4!

7



```
$ans = 3 + 4;
```

Perl language



Machine
code

Perl interpreter



Programming language

- Computers only understand machine language
- Need a way to translate our instructions into machine language
- Mediated by programming languages
 - Computers do not understand programming languages either!
- Need compilers & interpreters (translators)
- Commonly used programming languages
 - C/C++, Java, Perl

Programming nuts & bolts

- A computable task
 - Does it make sense to create a program to solve my problem?
- Text Editor
 - A program that allows you to save text files
 - Emacs (unix), vi (unix), gedit (unix), notepad (Windows)
- Compiler/Interpreter
 - gcc (C), javac (Java), perl (Perl)

What is Perl?

- **P** *Practical* **E** *Extraction* and **R** *Report* **L** *Language*
 - “*PERL saved the human genome project*”
- An interpreted programming language optimized for scanning text files and extracting information from them
- Combines the best features of *sed*, *awk*, *C*, *sh*, *csh*, *grep* and more

Introduction to Perl

- Perl
 - TMTOWTDI – There's more than one way to do it
 - The Perl mantra
 - Flexible, 'forgiving' language
 - Easy to learn for non-programmers
 - Sophisticated Regular Expression tools
 - Pattern matching text
 - Powerful for manipulation of text files
 - Object Oriented (sort of)
 - Can do almost anything, anywhere

Bioinformatics & Perl

- Some advantages for bioinformatics
 - it is easy to process and manipulate biological sequences
 - sequence data = text!
 - easy to run a program that controls and runs other programs
 - many problems can be solved in far fewer lines of Perl code than C or Java
 - Good for rapid prototyping in a research environment
 - dynamic web
- Bioperl, Ensembl, GMod & GBrowse

First Perl program

```
#!/usr/bin/perl
```

Path to the Perl interpreter

```
#This is a very simple program that prints  
#"Perl is my friend" to the screen
```

Comments get ignored by the interpreter

```
#Print "Perl is my friend" followed by a new  
#line to the screen
```

```
print "Perl is my friend\n";
```

Statements are the instructions of your program

All statements must be syntactically correct (eg semi-colon)

Parts of a Perl program

- **Interpreter line**
 - This tells the operating system where the Perl interpreter is sitting
 - Every Perl program must have this line
- **Comments**
 - Lines that begin with '#' that are ignored by the interpreter
 - Used to document the program
 - Extremely important in deciphering what the statements in the program are supposed to do
- **Statements**
 - Instructions of the program
 - Follow a strict syntax that must be learned by the programmer
 - Ill-formatted statements will prevent your program from running

Interactive Presentation

- Hands-on learning
- Concepts introduced first through definitions and examples with Sohrab
- Learn through real-time programming with Sanja
- Files have been created for you with comments
- You will fill in the code
- Ask questions

Print statement

- Print statements let you output information to the screen
- Can give print a list of items to print, separated by commas

```
print "this", " and that", "\n";
```

Steps to create and run a Perl program

- Change into “perl_one” directory
 - `cd perl_one`
- Open file “program1.pl” in text editor
 - `gedit`
- Write and save your program
- Change the permissions
 - `chmod u+x program1.pl`
- Run it!
 - `./program1.pl`

Program1

- Task
 - print something to the screen
- Concepts
 - structure of a program
 - perl interpreter line
 - comments
 - statements

Common errors

- ‘Command not found’
 - Did you include ‘./’ ?
- ‘... bad interpreter: No such file or directory’
 - Is the Perl interpreter line correctly formatted?
- ‘Can't find string terminator "'" anywhere before EOF’
 - Did you close the quotation marks?
- Did you remember the semi-colon?

How does Perl represent data?

- Most data we will encounter is the form of numbers or strings (text)
 - These are known as *scalars*
- Numbers
 - Integers, real, binary...
- Strings
 - Any text including **special characters** enclosed in single or double quotes

Numbers

7

3.14159

-0.00004

3.04E15

Perl supports
scientific
notation

Strings

“This is a string”

‘This is also a string’

“ACGACTACGACTAGCATCAGCATCAG”

Special characters for strings

- Strings can contain special characters for things like tab and newline
- These characters need to be escaped with a backslash
 - \t means tab
 - \n means newline
- These characters do not get interpreted inside single quotes

```
"Put a tab\t here"           "There is a newline \n here"
```

```
'Is there a \t tab here?'
```

No tab because \t will not be interpreted inside single quotes

Program1 cont'd

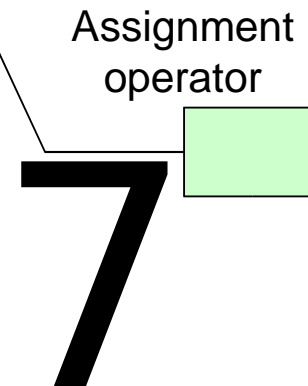
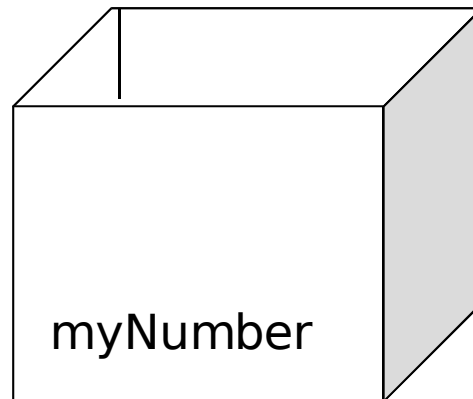
- Task
 - print out different strings to the screen
- Concepts
 - strings, special characters

Variables: how to store data in memory

- A *variable* is a ‘container’ that lets you store data
- Every variable has a name that acts as a label on the data it is storing
 - Variable names begin with a dollar sign followed by an alphanumeric name that may contain underscores
 - First character after the dollar sign must be a letter
 - \$x, \$this_variable, \$DNA, \$DNA2
- To store information in a variable, use the *assignment operator* ‘=’

Variables in action

```
$myNumber = 7;
```



Arithmetic operators – how Perl does math

+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
**	Exponentiation
()	Grouping

Using arithmetic operators

```
$sum = 3 + 4;
```

```
$difference = 4 - 3;
```

```
$product = 4 * 3;
```

```
$quotient = 4 / 3;
```

```
$mod = 4 % 3;
```

```
$power = 4**3;
```

```
$total = (4+3)*4-8;
```

Precedence follows rules of arithmetic

Increment/Decrement

- Other handy operators on integer variables

```
$var = 1;
```

```
$var++; # $var now equals 2
```

```
$var--; # $var now equals 1
```

Variable interpolation

- Substituting the name of a variable for its value in a double-quoted string
- Useful for printing out the value of a variable

```
$myVariable = 7;  
print "The value of my variable is: $myVariable\n";
```

```
The value of my variable is: 7
```

Variables only get interpolated inside double-quoted strings

Variable interpolation

- What about variables inside single-quoted strings?

```
$myVariable = 7;  
print `The value of my variable is: $myVariable\n`;
```

```
The value of my variable is: $myVariable
```

Variables do not
get interpolated
inside single-
quoted strings

Program2

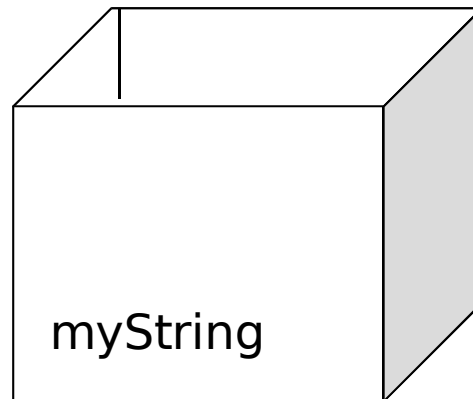
- Task
 - use numerical operators
 - print interpolated variables to the screen
- Concepts
 - using numerical operators
 - variable interpolation

Program2 - Functions

- print *ITEM1, ITEM2, ITEM3, ...*
 - Prints a list of variables, numbers, strings to the screen
 - The list might only contain one item

Variables can also store strings

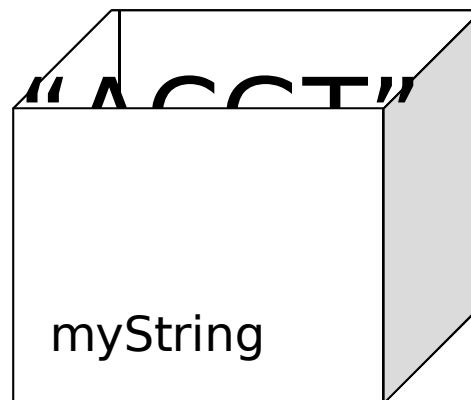
```
$myString = "ACGT";
```



"ACGT"

The value of a variable can be changed

```
$myString = "ACGT";  
$myString = "ACGU";
```



"ACGU"

Variables are so named
because they can take on
different values.

String operators

- Strings, like numbers also have operators
- *Concatenation* is done with '.'

```
$exon1 = "ATGTGG";
```

```
$exon2 = "TTGTGA";
```

```
$mRNA = $exon1 . $exon2;
```

```
print "The 2 exon transcript is: $mRNA \n";
```

```
The 2 exon transcript is: ATGTGGTTGTGA
```

The dot operator stitches strings to the left and right of it together into a new string

String operators

- How do I find the *length* of my string?
- Perl provides a useful function called...*length*

```
$DNA = "ACGTG";
```

```
$seqLength = length ($DNA);
```

```
print "my sequence is $seqLength residues long\n";
```

```
my sequence is 5 residues long
```

We'll go into
functions in more
detail later on

String operators

- How do I extract a *substring* (part of) of a string?
- Use the *substr* function

```
$longDNA = "ACGACTAGCATGCATCGACTACGACTACGATCAGCATCGACTACGTTTTAGCATCA"  
$shortDNA = substr ($longDNA, 0, 10);  
print "The first 10 residues of my sequence are: $shortDNA\n";
```

String from which
to extract the
substring

Start from this
position – Perl
starts counting
from 0

Length of the
substring

The first 10 residues of my sequence are: ACGACTAGCA

String operators

- How do you find the *position* of a given substring in a string?
 - Find the position of the first ambiguous “N” residue in a DNA sequence
- This can be done with the *index* function

```
$DNA = "AGGNCCT";
```

```
$position = index ($DNA, "N");
```

```
print "N found at position: $position\n";
```

```
N found at position: 3
```

The string to search

The substring to search for

Perl starts counting from 0

index returns -1 if substring is not found

Program 3

- Task
 - concatenate two strings together and calculate the new string's length
 - extract a substring from the new string
- Concepts
 - string operators and functions

Functions in Program3

- *STRING1 . STRING2*
 - concatenate two strings
- *length (VARIABLENAME)*
 - returns the length of a string
- *substr (VARIABLENAME, START, LENGTH)*
 - returns a substring from *VARIABLENAME* starting at position *START* with length *LENGTH*

Variables - recap

- Variables are containers to store data in the memory of your program
- Variables have a name and a value
- The value of a variable can change
- In Perl, variables starts with a '\$' and values are given to it with the '=' operator

I/O 2: Inputting data into variables

- So far we have ‘hard coded’ our data in our programs
- Sometimes we want to get input from the user
- This is done with the ‘<>’ operator
 - <STDIN> can be used interchangeably
- This allows your program to interact with the user and store data typed in on the keyboard

```
$input = <>;  
print "You typed $input\n";
```

I/O 2: Inputting data into variables

- Extra newline character?
- When using `<>`, the new line character is included in the input
- You can remove it by calling *chomp*

```
$input = <>;  
chomp ($input);  
print "$input was typed\n";
```

Program4

- Task
 - get a users name and age and print their age in days to the screen
- Concept
 - getting input from the user

Program4

- `<>`
 - get input from the keyboard
- `chomp (VARIABLENAME)`
 - removes the newline character from the end of a string

Arrays

- Arrays are designed to store more than one item in a single variable

- An ordered list of data

`(1, 2, 3, 4, 5)`

`("Mon", "Tues", "Wed", "Thurs", "Fri")`

`($gene1, $gene2, ...)`

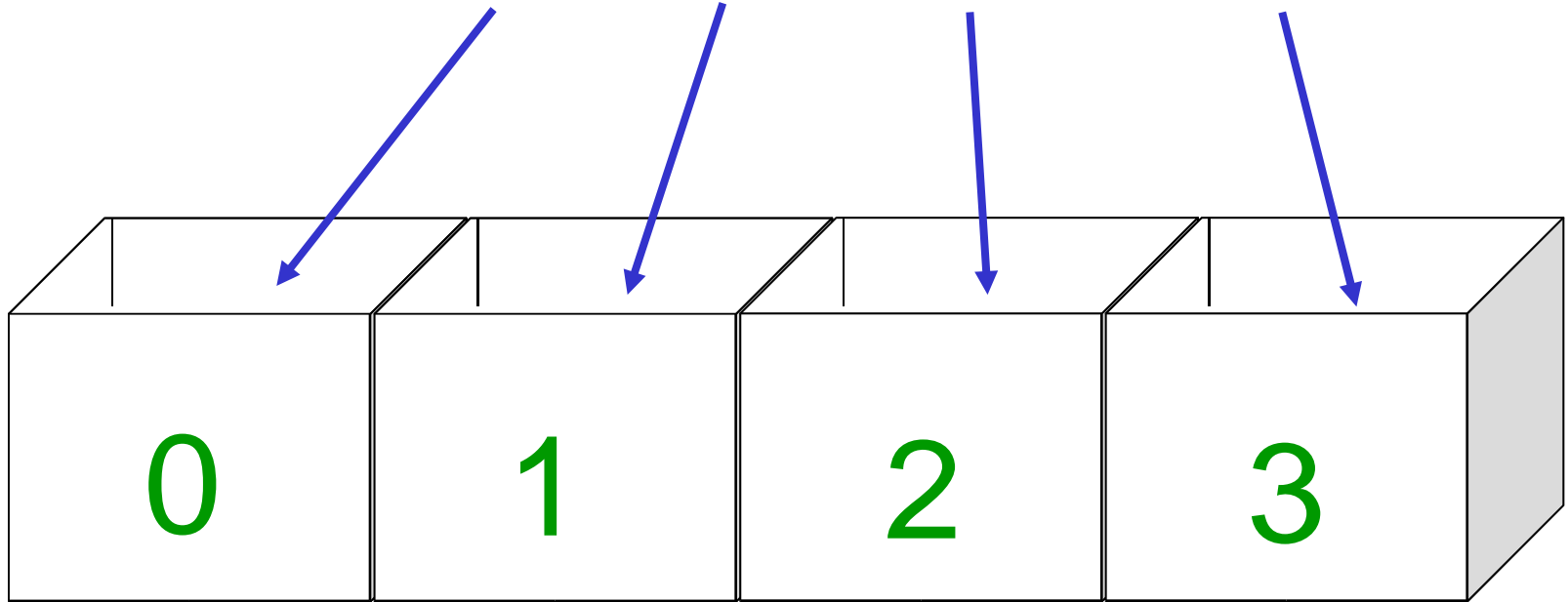
- Array variables start with '@'

`@myNumbers = (1, 2, 3, 4, 5);`

`@myGenes = ("gene1", "gene2", "gene3");`

Arrays

```
@myNumbers = ("one", "two", "three", "four");
```



Individual items in the array are indexed starting from 0

Array operators – []

- Accessing the individual elements of an array is done with the square brackets operator – []

```
@myNumbers = ("one","two","three","four");  
  
$element = $myNumbers[0];  
  
print "The first element is: $element\n";
```

When using the [] operator to access an element, the '@' changes to a '\$' because elements are scalars

When using the [] operator, the index of the element you want goes in the brackets

```
The first element is: one
```

Array functions - scalar

- How do I know how many elements are in my array?
- use the *scalar* function

```
@myNumbers = ("one","two","three","four");  
$numElements = scalar (@myNumbers);  
print "There are $numElements elements in my array\n";
```

There are 4 elements in my array

Array functions - reverse

- You can reverse the order of the elements in an array with *reverse*

```
@myNumbers = ("one","two","three","four");
```

```
@backwards = reverse(@myNumbers);
```

```
print "@backwards\n";
```

```
four three two one
```

Program5

- Task
 - create an array and access its elements
- Concepts
 - declaring an array
 - square bracket operator
 - array functions
 - array variable interpolation

Program5

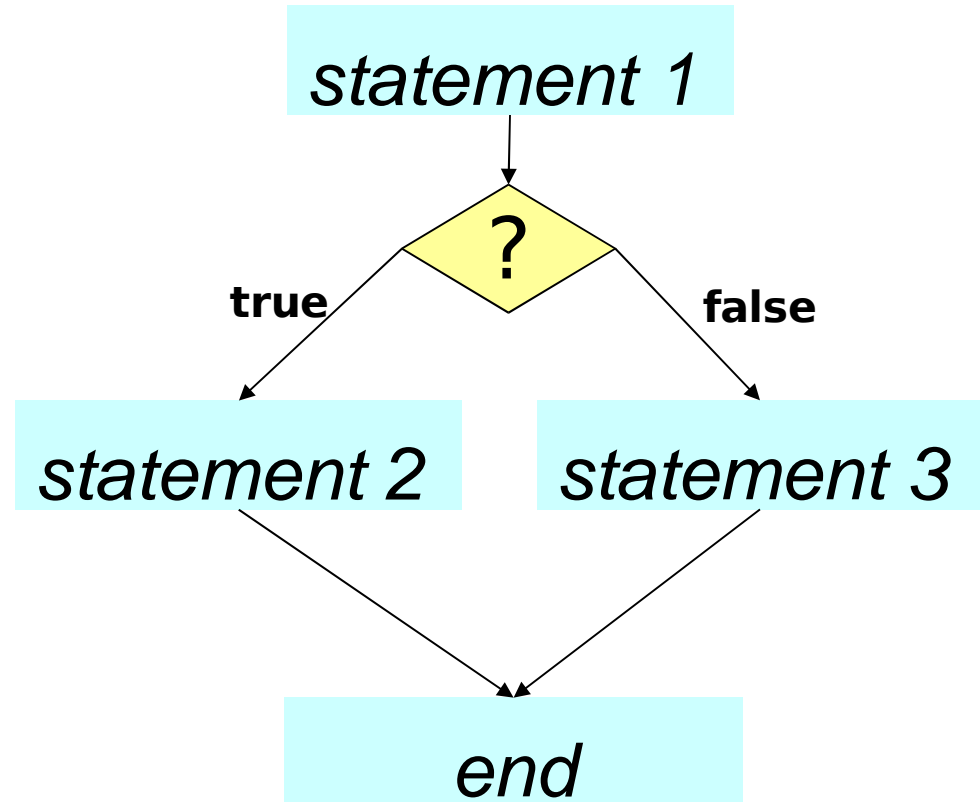
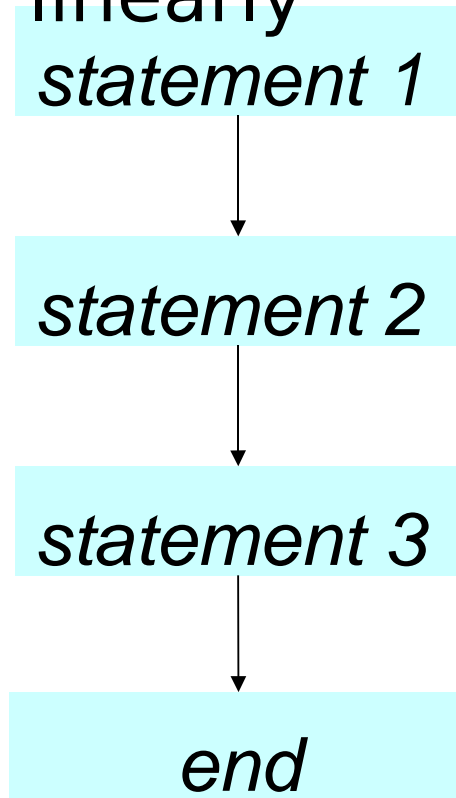
- $\$ARRAYNAME[INDEX]$
 - accesses individual element of the array indexed by *INDEX*
 - remember indexes start from 0
 - precede *ARRAYNAME* with '\$'
- scalar (*ARRAYNAME*)
 - returns the number of elements in the array

Arrays - recap

- Arrays are used to store lists of data
- Array names begin with '@'
- Individual elements are indexed and can be accessed with the square bracket operator
 - use '\$' for the variable name!
- Numerous functions exist to manipulate data in arrays

Control structures

Programs do not necessarily run linearly



Conditional expressions

- Perl allows us to program conditional paths through the program
 - Based on whether a condition is true, do this or that
- Is condition true? \Rightarrow *conditional expression*
- Most often conditional expressions are mediated by comparing two numbers or two strings with *comparison operators*

Comparison operators evaluate to true or false

Equals operator is different from assignment operator!

- Numerical comparison operators allow us to create *conditional expressions* with numbers
- In other words we can ask true/false questions about two numbers and Perl will give us an answer

==	equal	<code>\$a == \$b</code>
!=	not equal	<code>\$a != \$b</code>
>	greater than	<code>\$a > \$b</code>
<	less than	<code>\$a < \$b</code>
>=	greater than or equal	<code>\$a >= \$b</code>
<=	less than or equal	<code>\$a <= \$b</code>

Comparison operators evaluate to true or false

eq is different from == and cannot be used with numbers

- String comparison operators allow us to create *conditional expressions* with strings
- Strings are compared alphabetically except all uppercase letters are less than lowercase letters
- "z" lt "a" is **true**

Special operator that we will revisit later

eq	equal	\$a eq \$b
ne	not equal	\$a ne \$b
gt	greater than	\$a gt \$b
lt	less than	\$a lt \$b
ge	greater than or equal	\$a ge \$b
le	less than or equal	\$a le \$b
=~	pattern matching	\$a =~ /ATG/

What is true?

- Everything except
 - 0
 - ""
 - empty string
- Comparison operators evaluate to 0 or 1

Conditional expressions - if



- In English:
 - “If the light is red, then apply the brake.”
- In Perl:

```
if ($light eq "red") {  
    $brake = "yes";  
}
```

This is known as a *conditional block*. If the condition is true, execute all statements enclosed by the curly braces

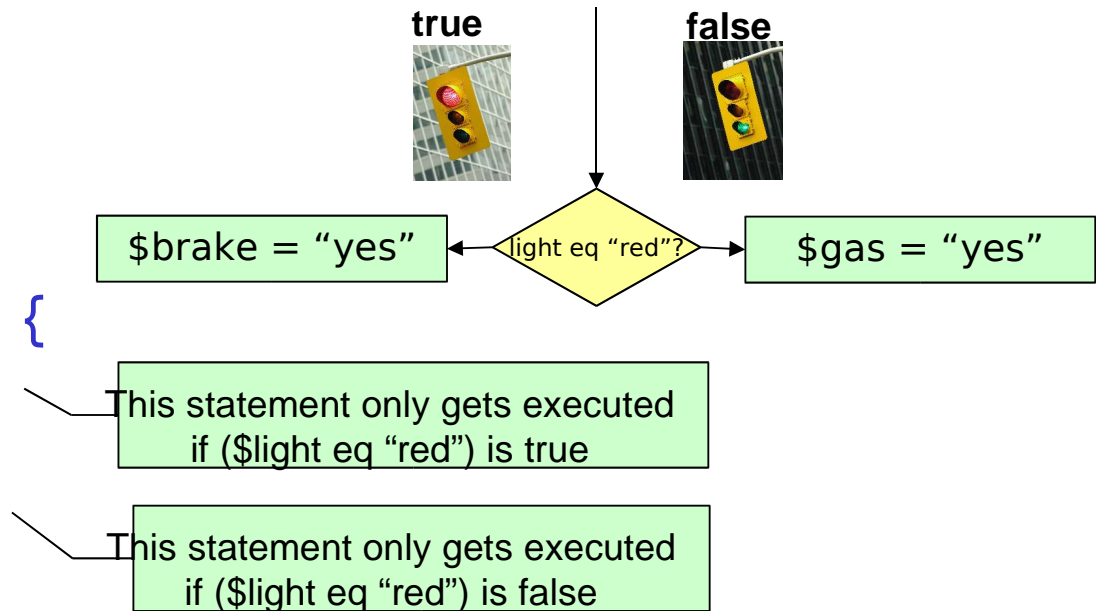
```
if (CONDITION) {  
    STATEMENT1;  
    STATEMENT2;  
    STATEMENT3;  
    ...  
}
```

Conditional expressions – if, else

- In English:
 - “If the light is red, then apply the brake. Otherwise hit the gas”

- In Perl:

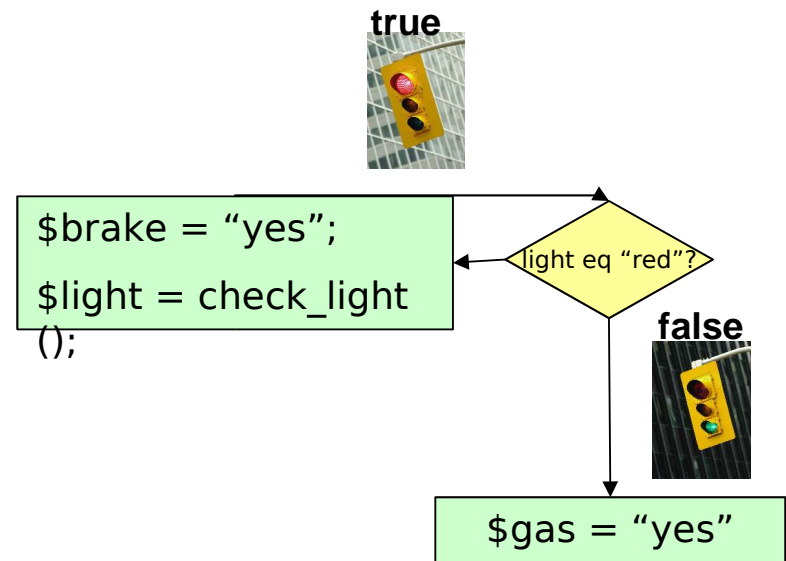
```
if ($light eq "red") {  
    $brake = "yes";  
} else {  
    $gas = "yes";  
}
```



Conditional expressions - while

- What if we need to repeat a block of statements ?
- This can be done with a *while* loop

```
$light = check_light();  
while ($light eq "red") {  
    $brake = "yes";  
    $light = check_light();  
}  
$gas = "yes";
```



Conditional expressions - while

- *while loop* structure

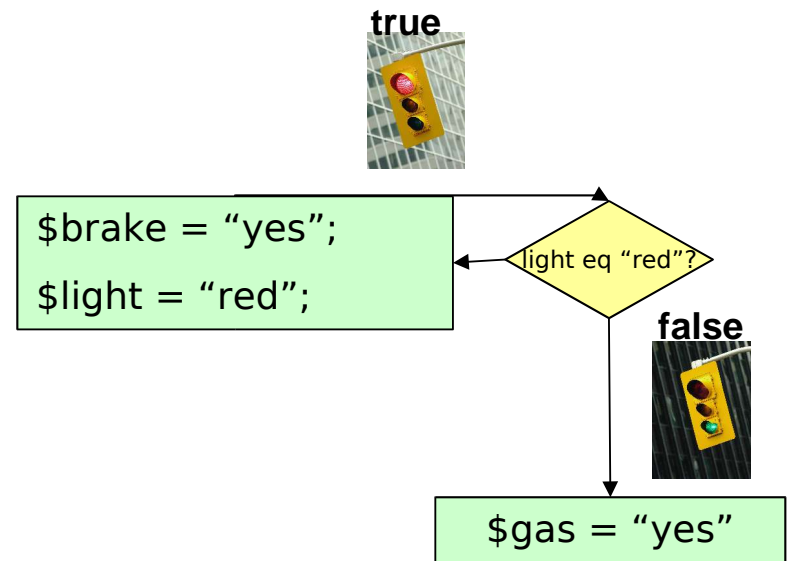
This is known as a *conditional block*. While the condition is true, execute all statements enclosed by the curly braces

```
while (CONDITION) {  
    STATEMENT1;  
    STATEMENT2;  
    STATEMENT3;  
    ...  
}
```

Conditional expressions - while

- Watch out for infinite loops!

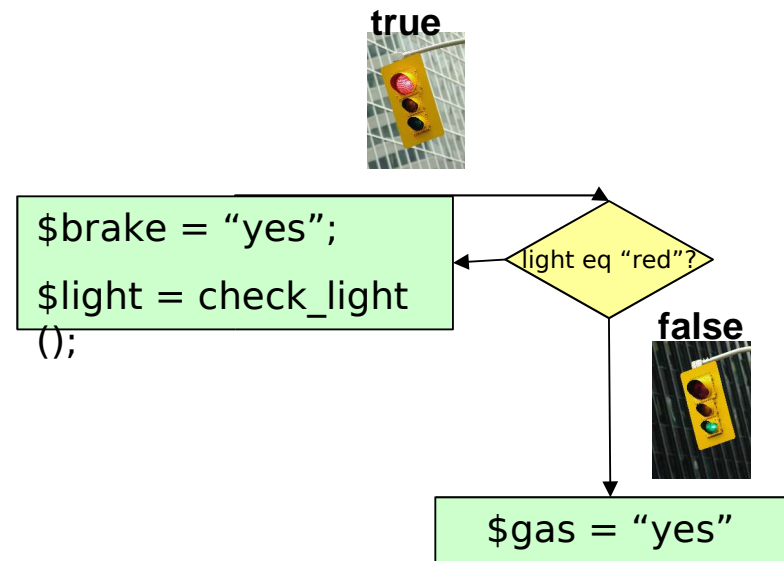
```
$light = check_light();  
while ($light eq "red") {  
    $brake = "yes";  
    $light = "red";  
}  
$gas = "yes";
```



Conditional expressions - while

- This loop is better

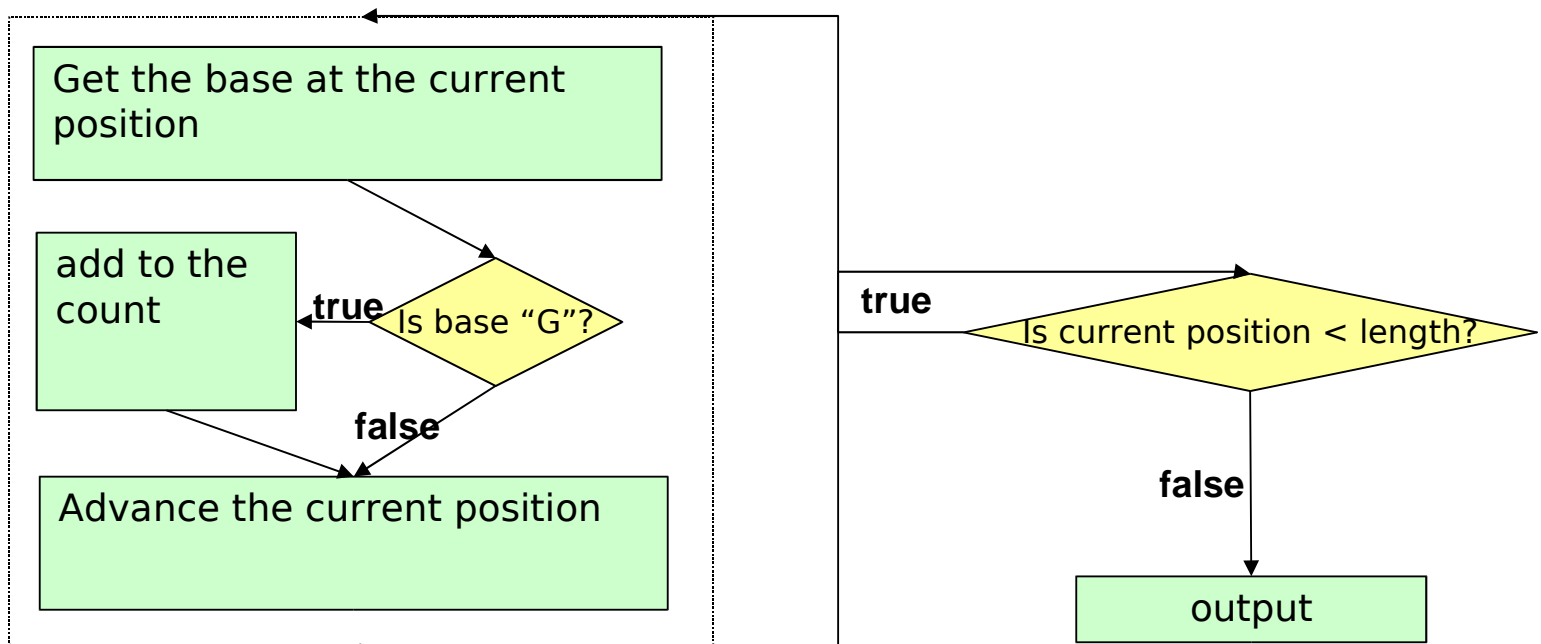
```
$light = check_light();  
while ($light eq "red") {  
    $brake = "yes";  
    $light = check_light();  
}  
$gas = "yes";
```



Program6

- Task
 - count the number of G's in a DNA sequence
- Concepts
 - Conditional expressions
 - If statement
 - While loop

Program6



Loops - for

- To repeat statements a fixed number of times, use a *for loop*

```
for (INITIAL ASSIGNMENT; CONDITION; INCREMENT) {  
    STATEMENT1;  
    STATEMENT2;  
    STATEMENT3;  
    ...  
}
```

```
for ($i = 1; $i < 11; $i++) {  
    print $i, "\n";  
}
```

This code prints the numbers 1-10 with each number on a new line

Loops - foreach

- *foreach* loops are used to *iterate* over an array
 - access each element of the array once

```
@array = (2,4,6,8);  
foreach $element (@array) {  
    $element++;  
    print $element, "\n";  
}
```

3

5

7

9

Program7

- Task
 - Initialise an array and print out its elements
- Concepts
 - for loops
 - foreach loops

Control structures - recap

- Control structures help your program make decisions about what to do based on *conditional expressions*
- Can construct conditional expressions using comparison operators for numbers and strings
- If – do something if the condition is true (otherwise do something else)
- While – do something while the condition is true
- For – do something a fixed number of times
- Foreach – do something for every element of an array

Regular Expressions

- Regular expressions are used to define patterns you wish to search for in strings
- Use a syntax with rules and operators
 - Can create extremely sophisticated patterns
 - Numbers, letters, case insensitivity, repetition, anchoring, zero or one, white space, tabs, newlines, etc....
 - Patterns are deterministic but can be made extremely specific or extremely general
 - Test for match, replace, select
- Lots on REGEX tomorrow!

Using REGEX

- `=~` is the operator we use with REGEX
- `=~` is combined with *utility operators* to match, replace

```
$DNA = "AGATGATAT";  
if ($DNA =~ /ATG/) {  
    print "Match!";  
}
```

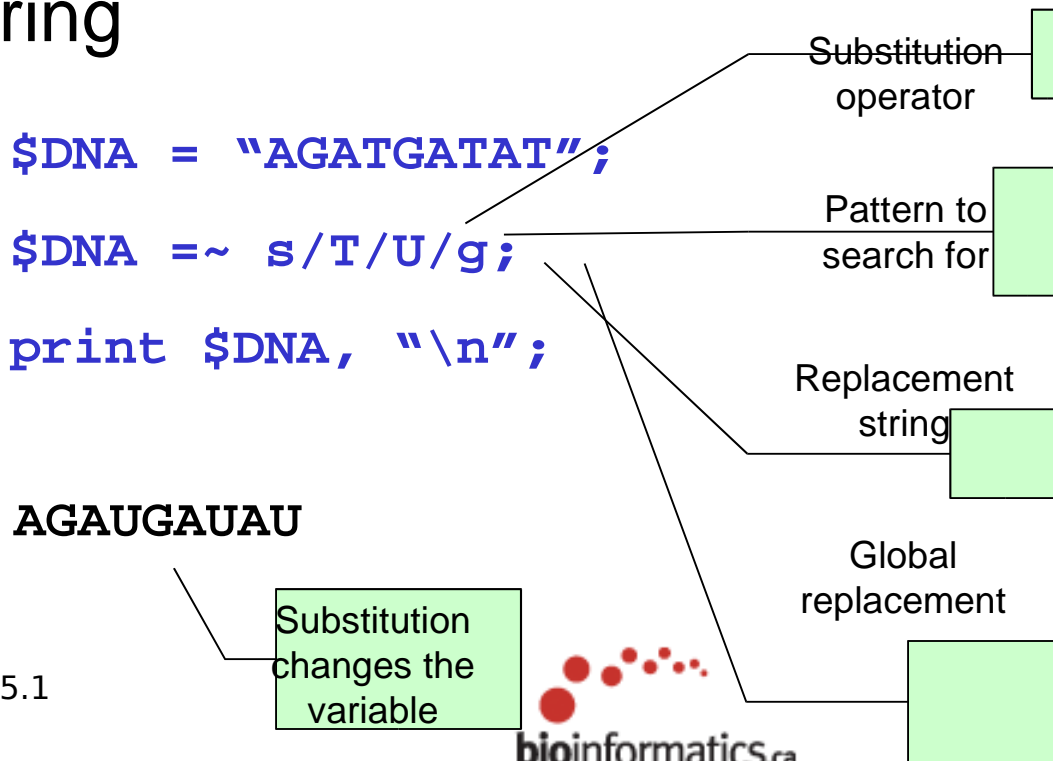
`=~` pattern match
comparison operator

Matching leaves the
string unchanged

The pattern is a set
of characters
between //

REGEX - Substitution

- You can substitute the parts of a string that match a regular expression with another string



REGEX - Substitution

```
$DNA = "AGATGATAT";
```

```
$DNA =~ s/T/U/;
```

```
print $DNA, "\n";
```

AGAUGATAT

REGEX - Translation

- You can translate a string by exchanging one set of characters for another set of characters

```
$DNA = "AGATGATAT";
```

```
$DNA =~ tr/ACGT/TGCA/;
```

```
print $DNA, "\n";
```

Translation operator

Set of characters to replace

Replacement characters

TCTACTATA

Translation changes the variable

Program8

- Task
 - transcription and reverse complement a DNA sequence
- Concepts
 - Simple regular expressions using s and tr

Prgram8 - Functions

- `reverse(STRING)`
 - Function that reverses a string
- `STRING =~ s/PATTERN/REPLACEMENT/modifiers`
 - This is the substitute operator
- `STRING =~ tr/CHARS/REPLACEMENT CHARS/`
 - This is the translation operator

REGEX - recap

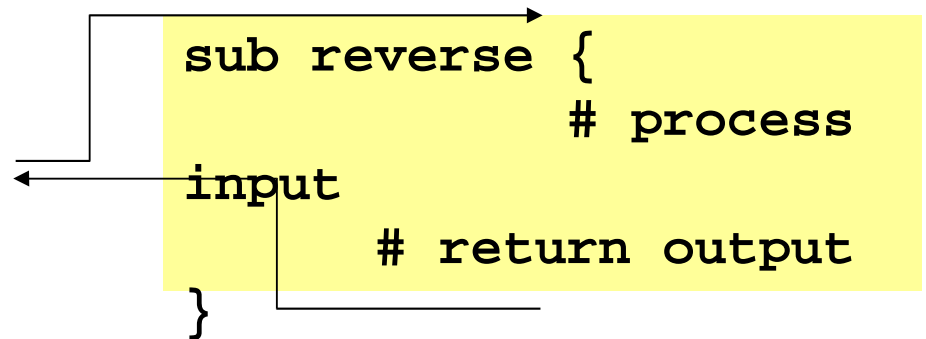
- REGEX are used to find patterns in text
- Use a syntax that must be learned in order to be exploited
- Extremely powerful for processing and manipulating text
- Will be examined more closely tomorrow

Functions

- Functions (sub-routines) are like small programs inside your program
- Like programs, functions execute a series of statements that process input to produce some desired output
- Functions help to organise your program
 - parcel it into named functional units that can be called repeatedly
- There are literally hundreds of functions built-in to Perl
- You can make your own functions

What happens when you call a function?

```
$DNA = "ACATAATCAT";  
$rcDNA = reverse ($DNA);  
$rcDNA =~ tr/ACGT/TGCA/;
```



Calling a function

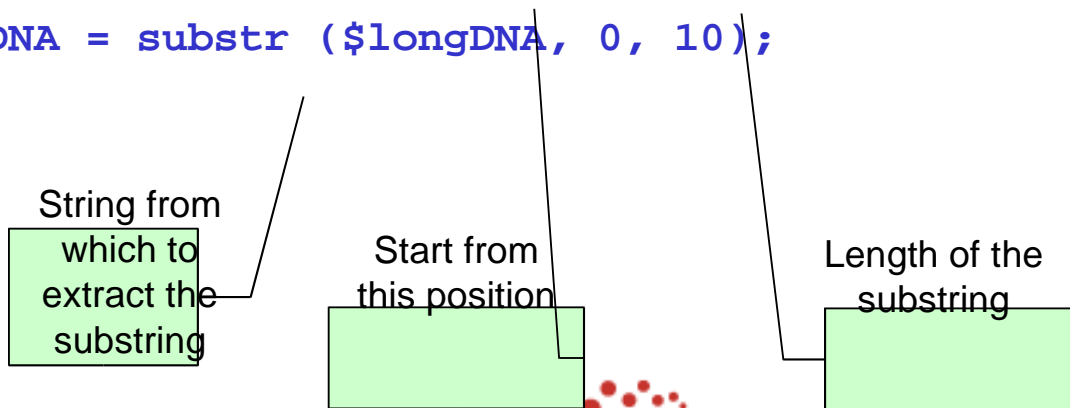
- Input is passed to a function by way of an ordered parameter list

Basic syntax of calling a function

```
$result = function_name (parameter list);
```

```
$longDNA = "ACGACTAGCATGCATCGACTACGACTACGATCAGCATCGACT"
```

```
$shortDNA = substr ($longDNA, 0, 10);
```



Useful string functions in Perl

- **chomp(STRING) OR chomp(ARRAY)** –
 - Uses the value of the \$/ special variable to remove endings from STRING or each element of ARRAY. The line ending is only removed if it matches the current value of \$/.
- **chop(STRING) OR chop(ARRAY)**
 - Removes the last character from a string or the last character from every element in an array. The last character chopped is returned.
- **index(STRING, SUBSTRING, POSITION)**
 - Returns the position of the first occurrence of SUBSTRING in STRING at or after POSITION. If you don't specify POSITION, the search starts at the beginning of STRING.
- **join(STRING, ARRAY)**
 - Returns a string that consists of all of the elements of ARRAY joined together by STRING. For instance, join(">>", ("AA", "BB", "cc")) returns "AA>>BB>>cc".
- **lc(STRING)**
 - Returns a string with every letter of STRING in lowercase. For instance, lc("ABCD") returns "abcd".
- **lcfirst(STRING)**
 - Returns a string with the first letter of STRING in lowercase. For instance, lcfirst("ABCD") returns "aBCD".
- **length(STRING)**
 - Returns the length of STRING.
- **split(PATTERN, STRING, LIMIT)**
 - Breaks up a string based on some delimiter. In an array context, it returns a list of the things that were found. In a scalar context, it returns the number of things found.
- **substr(STRING, OFFSET, LENGTH)**
 - Returns a portion of STRING as determined by the OFFSET and LENGTH parameters. If LENGTH is not specified, then everything from OFFSET to the end of STRING is returned. A negative OFFSET can be used to start from the right side of STRING.
- **uc(STRING)**
 - Returns a string with every letter of STRING in uppercase. For instance, uc("abcd") returns "ABCD".
- **ucfirst(STRING)**
 - Returns a string with the first letter of STRING in uppercase. For instance, ucfirst("abcd") returns "Abcd".

Useful array functions in Perl

- **pop(ARRAY)**
 - Returns the last value of an array. It also reduces the size of the array by one.
- **push(ARRAY1, ARRAY2)**
 - Appends the contents of ARRAY2 to ARRAY1. This increases the size of ARRAY1 as needed.
- **reverse(ARRAY)**
 - Reverses the elements of a given array when used in an array context. When used in a scalar context, the array is converted to a string, and the string is reversed.
- **scalar(ARRAY)**
 - Evaluates the array in a scalar context and returns the number of elements in the array.
- **shift(ARRAY)**
 - Returns the first value of an array. It also reduces the size of the array by one.
- **sort(ARRAY)**
 - Returns a list containing the elements of ARRAY in sorted order. See next Chapter 8 on References for more information.
- **split(PATTERN, STRING, LIMIT)**
 - Breaks up a string based on some delimiter. In an array context, it returns a list of the things that were found. In a scalar context, it returns the number of things found.

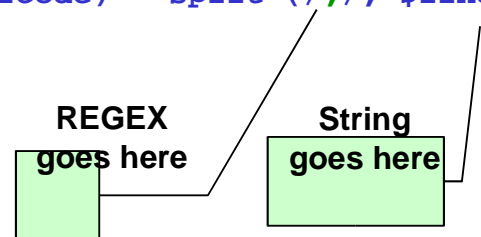
source: <http://www.cs.cf.ac.uk/Dave/PERL/>

String functions - split

- ‘splits’ a string into an array based on a delimiter
- excellent for processing tab or comma delimited files

```
$line = "MacDonald,Old,The farm,Some city,BC,E1E 101";  
($lastname, $firstname, $address, $city, $province, $postalcode) = split (/,/, $line);  
  
print ("LAST NAME: ", $lastname, "\n",  
      "FIRST NAME: ", $firstname, "\n",  
      "ADDRESS: ", $address, "\n",  
      "CITY: ", $city, "\n",  
      "PROVINCE: ", $province, "\n",  
      "POSTAL CODE: ", $postalcode, "\n");
```

```
LAST NAME: MacDonald  
FIRST NAME: Old  
ADDRESS: The Farm  
CITY: Some city  
PROVINCE: BC  
POSTAL CODE: E1E 101
```



Array functions - sort

- You can sort the elements in your array with 'sort'

```
@myNumbers = ("one", "two", "three", "four");  
@sorted = sort(@myNumbers);  
print "@sorted\n";
```

```
four one three two
```

sorts
alphabetically



Making your own function

'sub' tells the interpreter you are declaring a function

```
sub function_name {  
    (my $param1, my $param2, ...) = @_  
    # do something with the parameters  
    my $result = ...  
    return $result;  
}
```

This is the function name. Use this name to 'call' the function from within your program

What is this? This is an array that gets created automatically to hold the parameter list.

What is the word 'my' doing here? 'my' is a variable qualifier that makes it local to the function. Without it, the variable is available anywhere in the program. It is good practice to use 'my' throughout your programs – more on this tomorrow.

'return' tells the interpreter to go back to the place in the program that called this function. When followed by scalars or variables, these values are passed back to where the function was called. This is the output of the function

Making your own function - example

```
sub mean {  
  my @values = @_  
  my $numValues = scalar @values;  
  my $mean;  
  foreach my $element (@values) {  
    my $sum = $sum + $element;  
  }  
  $mean = $sum / $numValues;  
  return $mean;  
}
```

Function definition

local variables to be used
inside the function

do the work!

return the answer

```
$avg = mean(1,2,3,4,5);
```

Program9

- Task
 - Create a function to reverse complement a DNA sequence
- Concepts
 - creating and calling functions

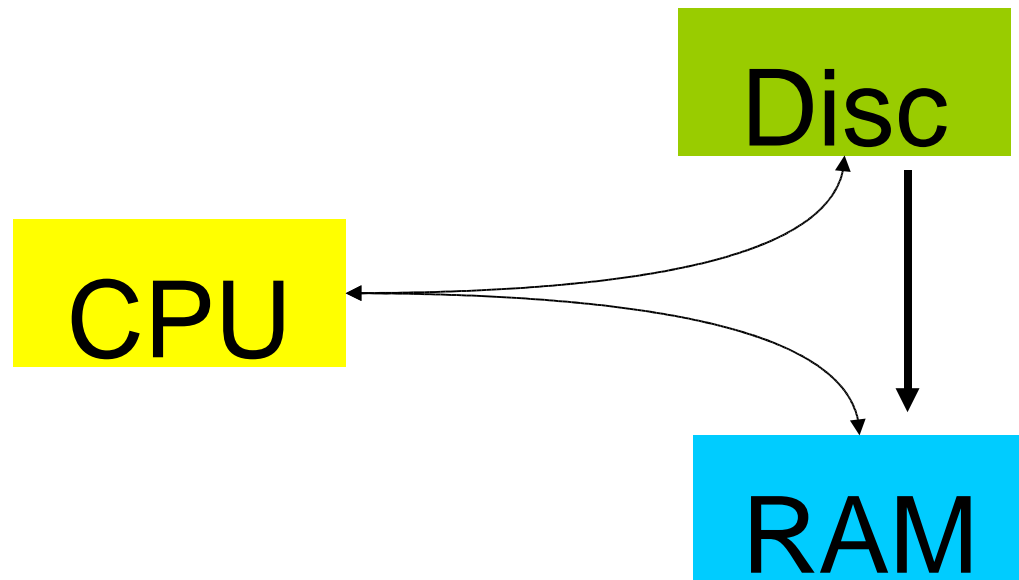
Functions - recap

- A function packages up a set of statements to perform a given task
- Functions take a parameter list as input and return some output
- Perl has hundreds of functions built-in that you should familiarise yourself with
 - Keep a good book, or URL handy at all times
- You can (and should!) make your own functions

I/O 3: Filehandles

- So far, we have only considered input data that we created inside our program, or captured from the user
- Most data of any size is stored on your computer as files
 - Fasta files for sequences, tab-delimited files for gene expression, General Feature Format for annotations, XML files for literature records, etc...
- Filehandles allow us to access data contained in files from our programs

Filehandles



- Filehandles allow you to read a file from disc and store the information in memory

Filehandles

- Filehandles have 3 major operations
 - open
 - read one line at a time with '<>'
 - close
- Filehandles are special variables that don't use '\$'
 - by convention Filehandle names are ALLCAPS
- The default filehandle for input is STDIN
- The default filehandle for output is STDOUT

Filehandles - example

gcfile.txt

seq1	ACGACTAGCATCAGCAT	47.0	
seq2	AAAAATGATCGACTATATAGCATA		25.0
seq3	AAAGGTGCATCAGCATGG	50.0	

Filehandles - example

```
open (FILE, "gcfile.txt");
```

opens the filehandle

```
while (<FILE>) {
```

reads the next line of the file into the automatic \$_ variable

```
    $line = $_;  
    chomp ($line);
```

removes the newline character from \$line

```
    ($id,$seq,$gc) = split (/\\t/, $line);
```

splits the line on \t

```
    print ("ID: " , $id, "\n",  
          "SEQ: ", $seq, "\n",  
          "GC: ", $gc, "\n");
```

prints out the data in a nice format

```
}
```

```
close (FILE);
```

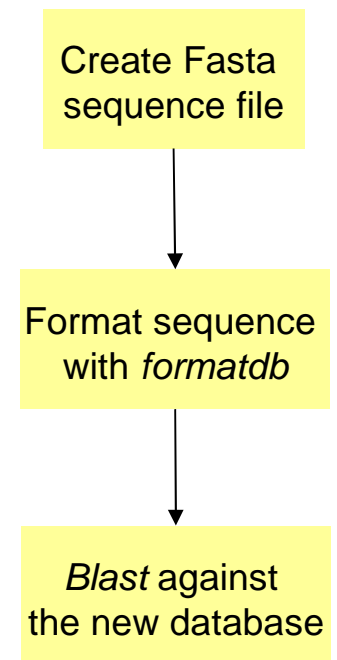
closes the filehandle

Program9, cont'd

- Task
 - read a DNA sequence from a file and reverse complement it
- Concepts
 - reading and processing data from a file

Process management

- Perl allows you to make 'system calls' inside your program
 - Executing Unix commands from your code
- Useful for 'wrapping' other programs inside a Perl program
 - Automated executions of a program with variable arguments
- Create pipelines of programs where output from one program is used as input to a subsequent program



system() and backticks

- There are two common ways of executing system calls
- Use the function system()
 - does not capture the output
- Use the backticks operators
 - captures the output

```
$command = "ls -l";  
system($command);  
$listing = ` $command `;
```

Executes 'ls -l' as though you typed it at the prompt. Prints the output to the screen

Executes 'ls -l' and stores the output in \$listing. Does not print the output to the screen

Program10

- Task
 - List the files in your working directory
- Concepts
 - Backticks operator to execute a Unix command and capture and process its output

Putting it all together

- Write a program that reads a DNA sequence from a file, calculates its GC content, finds its reverse complement and writes the reverse complement to a file
- Concepts
 - covers most of the topics from today
- prepare for next day

Programming tips

- Think hard and plan before even starting your program
- Save and run often!
- Include lots of comments
- Use meaningful variable and function names
 - self-documenting code
- Keep it simple
 - If its complex, break the problem into simple, manageable parts
- Use functions
- Avoid hard-coding values – use variables!
- Make your code modifiable, modular, legible

QUESTIONS?