Introduction to Perl programming
Session I

Ernesto Lowy

CRG Bioinformatics core

Basic Unix

- ✓ During the course all exercises are done using the terminal
- ✓ Terminal – an interface that allows users to run commands through the command line interface.
- ✓ Prompts for commands and execute them after pressing of Enter
- ✓ All commands are case-sensitive
- ✓ Windows terminal commands are not exactly the same as in UNIX

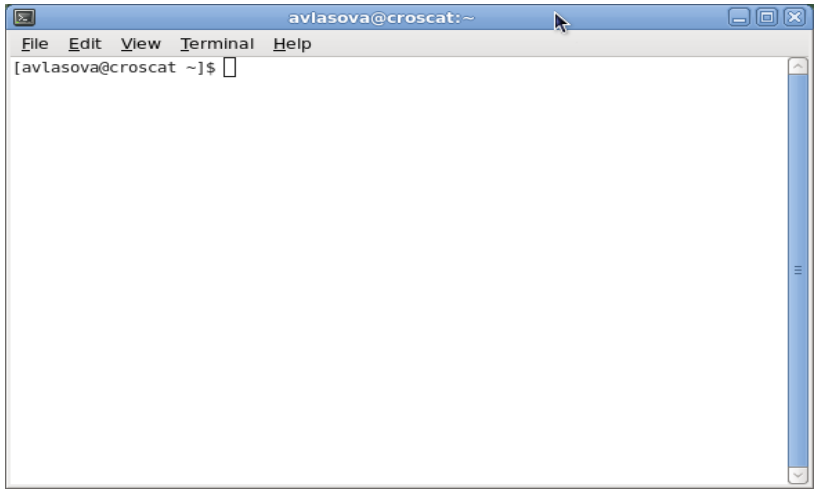
Exercise 1: *Where am I?*

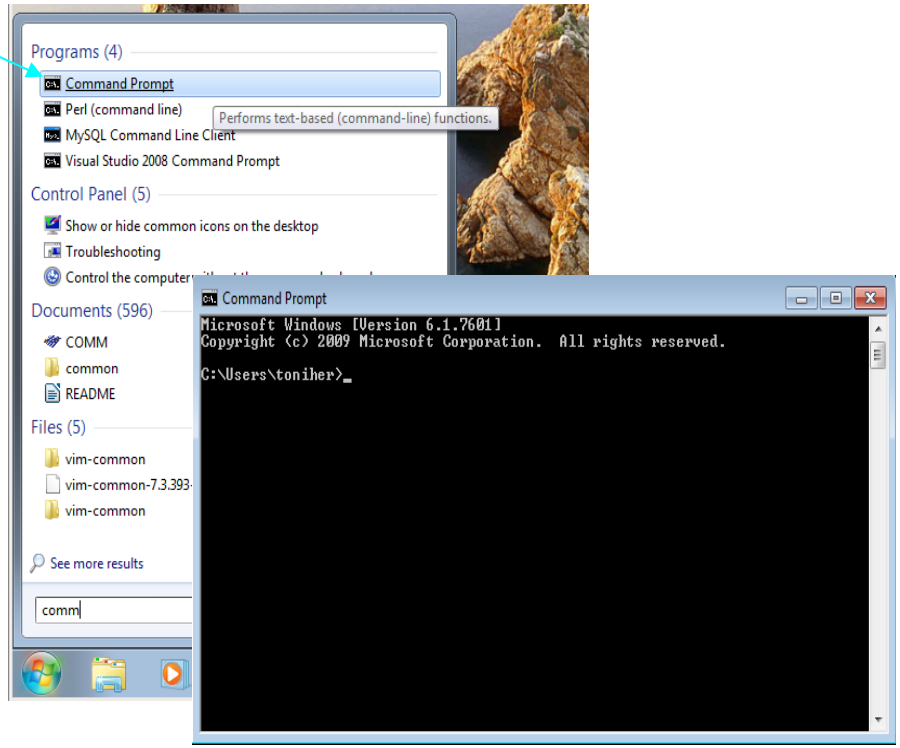
Launch terminal

Mac OS

WINDOWS

click here





Path

pwd	*current location*	chdir
ls	*folder content*	dir

Basic Unix: commands

Path

pwd ← get current path

ls ← list folder content

ls -l ← list folder content in long format

cd ← change to home folder

cd ../../relative/path/

cd /absolute/path/

Files

touch <file_name> ← change timestamp

less <file_name> ← show file content

cp <file1> <file2> ← copy file1 to file2

mv <file_name> <new_file> ← move file

rm <file_name> <new_file> ← delete file

cat <file1> <file2>← concatenate files

Folders

mkdir <dir_name> ← make

rmdir <dir_name> ← delete

rm -rf <dir_name> ← delete

cp -rf <dir1> <dir2> ← copy

mv -rf <dir1> <dir2> ← move

Other

<command> -h ← command help

man <command>← manual pages

ps alh ← list process in human readable format

kill ← stop program by process ID

zip <file_name> ← compress file

unzip <file_name> ← uncompress file

Exercise 2: *First file.*

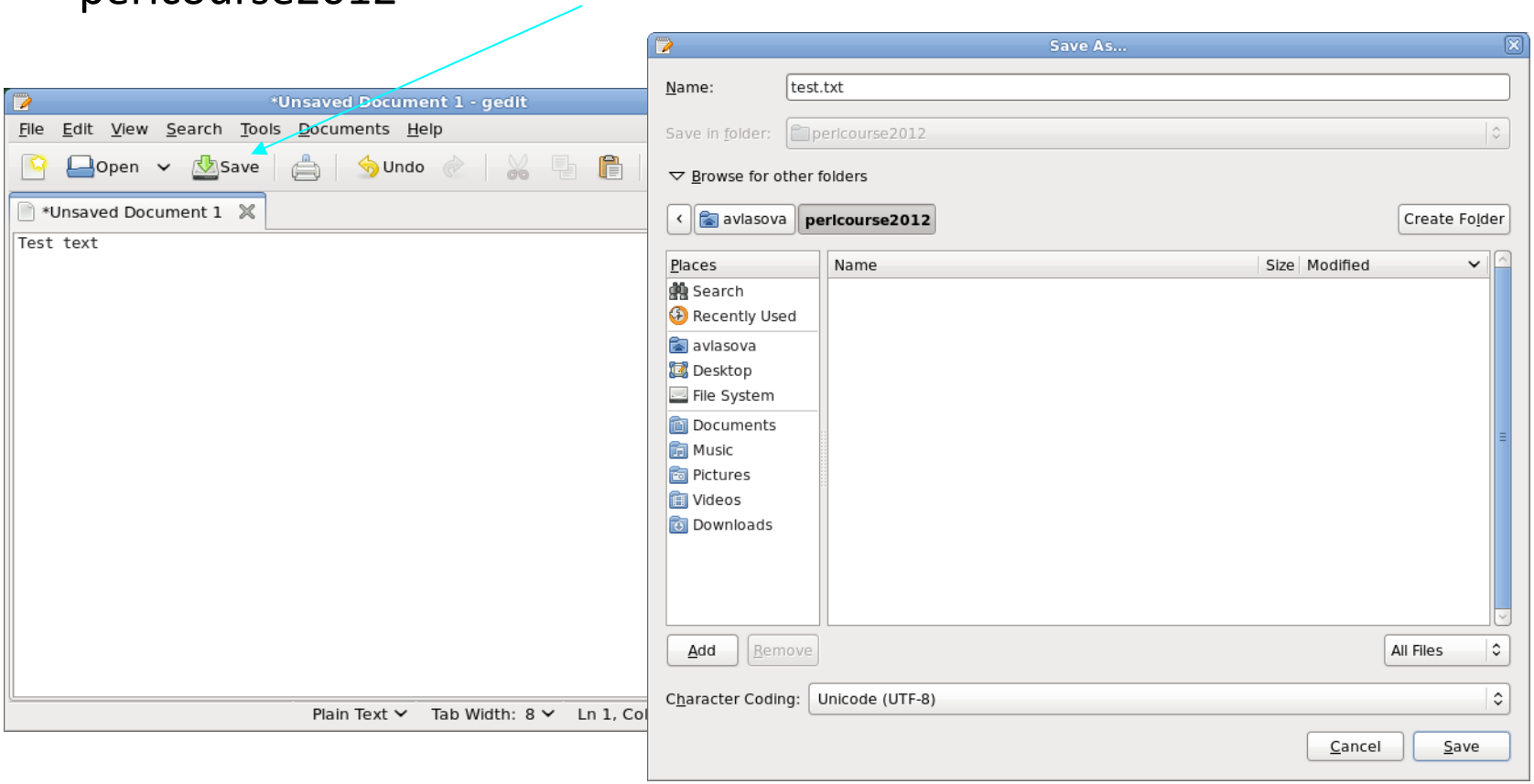
- ✓ Create folder for course exercises 'perlcourse2012'

  $ mkdir perlcourse2012

- ✓ Launch gedit

  $ gedit

- ✓ Type a random text and save file with name 'test.txt' into folder 'perlcourse2012'



Exercise 3: *Basic operations.*

Check that the working directory is 'perlcourse2012'

```
$ pwd
```

Get the directory content

```
$ ls
```

Copy 'test.txt' into 'test2.txt'

```
$ cp test.txt test2.txt
```

Get content of 'test2.txt'

```
$ more test2.txt
```

Get directory content with full information

```
$ ls -la
```

Delete 'test.txt'

```
$ rm test.txt
```

What is Perl?

- Perl is a programming language extensively used in bioinformatics
- Created by Larry Wall in 1987
- Provides powerful text processing facilities, facilitating easy manipulation of text files
- Perl is an interpreted language (no compiling is needed)
- Perl is quite portable
- Programs can be written in many different ways (advantage?)
  - Perl slogan is "There's more than one way to do it”
- Rapid prototyping (solve a problem with fewer lines of code than Java or C)

Installing Perl

- Perl comes by default on Linux and MacOSX
- On windows you have to install it:

  http://strawberryperl.com/ (100% open source)

  http://www.activestate.com/ (commercial distribution-but free!)
- Latest version is Perl 5.14.2

  To check if Perl is working and version

  $perl –v

Perl resources

- Web sites
  - www.perl.com
  - http://perldoc.perl.org/
  - https://www.socialtext.net/perl5/index.cgi
  - http://www.perlmonks.org/

- Books
  - Learning Perl (good for beginners)
  - Beginning Perl for Bioinformatics
  - Programming Perl (Camel book)
  - Perl cookbook

Ex1. First program…

1) Open a terminal
2) Enter `which perl`
3) Open `gedit` and enter

```
#!/../path/to/perl –w
#prints Hello world in the screen
print “Hello world!\n”;
```

4) Save it as `hello.pl`
5) Execute it with

```
perl hello.pl
```

Perl basic data types
Numbers

```
1000 #integer
1.25 #floating-point
1.2e30 #1.2 times 10 to the 30th power
-1
-1.2
```

Only important thing to remember is that you never insert commas or spaces into numbers in Perl. So in a Perl program you never will find:

```
10 000
10,000
```

Perl basic data types
Strings

- A string is a collection of characters in either single or double quotes:

```
“This is the CRG.”
‘CRG is in Barcelona!’
```

Difference between single and double quotes is:

```
print “Hello!\nMy name is Ernesto\n”; #Interprete contents
```

Will display:

```
>Hello!
>My name is Ernesto
```

```
print ‘Hello!\nMy name is Ernesto\n’; #contents should be
   taken literally
```

Will display:

```
>Hello!\nMy name is Ernesto\n
```

Scalar variables

- Variable is a name for a container that holds one or more values.
- Scalar variable (contains a single number or string):

```
$a=1;
$codon="ATG";
$a_single_peptide="GMLLKKKI";
```

(valid Perl identifiers are letter,words,underscore,digits)
Important! Scalar variables cannot start with a digit
Important! Uppercase and Lowercase letters are distinct ($Maria and $maria)

Example (Assignment operator):

```
$codon="ATG";
print "$codon codes for Methionine\n";
```

Will display:

```
ATG codes for Methionine
```

Ex 2. A program to store a DNA sequence

1) Open a terminal
2) Enter `which perl`
3) Open `gedit` and enter

```
#!/../path/to/perl –w
#Storing DNA in a variable, and printing it out
#First we store the DNA in a variable called $DNA
$DNA='ACGTGGTTAAATGTGTTGGTGTGTGG';
#Next, we print the DNA onto the screen
print $DNA;
```

4) Save it as `dna.pl`
5) Execute it with

```
perl dna.pl
```

Numerical operators

- Perl provides the typical operators. For example:

```
5+3 #5 plus 3, or 5
3.1-1.2 #3.1 minus 1.2, or 1.9
4*4 # 4 times 4 = 16
6/2 # 6 divided by 2, or 3
```

- Using variables

```
$a=1;
$b=2;
$c=$a+$b;
print “$c\n”;
```

Will print:

```
3
```

Special numerical operators

- `$a++; #same than`
  `$a=$a+1;`
- `$b--; #same than`
  `$b=$b-1;`
- `$c +=10; #same than`
  `$c=$c+10;`

String manipulation

- Concatenate strings with the dot operator

```
"ATG"."TCA" # same as "ATGTCA"
```

- String repetition operator (x)

```
"ATC" x 3   # same as "ATCATCATC"
```

- Length() get the length of a string

```
$dna="acgtggggtttttt";
print "This sequence has ".length($dna)."
nucleotides\n";
```

Will print:

```
This sequence has 10 nucleotides
```

- convert to upper case

```
$aa=uc($aa);
```

- convert to lower case

```
$aa=lc($aa);
```

Ex 3. Concatenating DNA fragments

1) Open a terminal
2) Enter `which perl`
3) Open `gedit` and enter

```
#!/../path/to/perl –w
#Store two DNA fragments into two variables called $DNA1 and $DNA2
$DNA1="AGGGGGTTTGCGTGTGGGCGGG";
$DNA2="GGGTGGGTGAGGTGCTGCTGCT";
#print the DNA onto the screen
print "Here are the original two DNA fragments:\n";
print $DNA1,"\n";
print $DNA2,"\n";
#Concatenate the DNA fragments into a third variable and print them
$DNA3=$DNA1.$DNA2
print "Here is the concatenation of the first two fragments:\n";
print $DNA3,"\n";
```

4) Save it as `concatenate.pl`
5) Execute it with

```
perl concatenate.pl
```

Conditional statements (if/else)

- Determine a particular course of action in the program.
- Conditional statements make use of the comparison operators to compare numbers or strings. These operators always return true/ false as a result of the comparison

Comparison operators (Numbers)

Comparison	Numeric
Equal	==
Not equal	!=
Less than	<
Greater than	>
Less than or equal to	<=
Greater than or equal to	>=

Examples:

35 == 35 # true
35 != 35 # false
35 != 32 # ????
35 == 32+3 # ????

Comparison operators (Strings)

Comparison	Numeric
Equal	eq
Not equal	ne
Less than	lt
Greater than	gt
Less than or equal to	le
Greater than or equal to	ge

Examples:

‘hello’ eq ‘hello’ # true
‘hello’ ne ‘bye’ # true
‘35’ eq ‘35.0’ # ????

If/else statement

- Allows to control the execution of the program

```
Example:
$a=4;
$b=10;
if ($a>$b) {
   print “$a is greater than $b\n”;
} else {
   print “$b is greater then $a\n”;
}
```

Ex 4.

a) Open gedit, write the code above and save it with the name `compare.pl`. Finally execute it. What do you obtain?

b) Change the variables values to $a=6 and $b=3 and rerun `compare.pl`. What do you obtain?

c) Change the variables values to $a=3 and $b=3 and rerun `compare.pl`. What do you obtain?

elsif clause

- To check a number of conditional expressions, one after another to see which one is true
- Game of rolling a dice. Player wins if it gets an even number

```
$outcome=6; #enter here the result from rolling a dice
if ($outcome==6)  {
   print “Congrats! You win!\n”;
} elsif ($outcome==4) {
   print “Congrats! You win!\n”;
} elsif ($outcome==2) {
   print “Congrats! You win!\n”;
} else {
   print “Sorry, try again!\n”;
}
```

Ex5. Correct `compare.pl` from Ex4. to cope with equal values for `$a` and `$b`

Answer

Ex 5. Correct `compare.pl` from Ex4. to cope with equal values for `$a` and `$b`

`compare.pl`

```
$a=4;
$b=10;
if ($a>$b) {
   print “$a is greater than $b
\n”;
} elsif ($a<$b) {
   print “$b is greater then $a
\n”;
} else {
   print “$b is equal to $a\n”;
}
```

Logical operators

- Used to combine conditional expressions
- || (OR)

1st expression outcome	2nd expression outcome	Combined outcome
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
TRUE	TRUE	TRUE
FALSE	FALSE	FALSE

Logical operators

Example:

```
$day=“Saturday”;
if ($day eq “Saturday” || $day eq “Sunday”)
{
   print “Hooray! It’s weekend!\n”;
}
```

Will print:

```
>Hooray! It’s weekend!
```

Logical operators

- && (AND)

1st expression outcome	2nd expression outcome	Combined outcome
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
TRUE	TRUE	TRUE
FALSE	FALSE	FALSE

Example:

```
$hour=12;
if ($hour >=9 && $hour <=18)  {
    “You are supposed to be at work!\n”;
}
```

Will print:

```
>You are supposed to be at work!
```

Boolean values

- Perl does not have the Boolean data type. So how Perl knows if a given variable is true or false?
- If the value is a number then 0 means false; all other numbers mean true
- Example:

```
$a=15;
$is_bigger=$a>10; # $is_bigger will be 1
if ($is_bigger) {….}; # this block will be
  executed
```

Boolean values

- If a certain value is a string. Then the empty string (‘’) means false; all other strings mean true

```
$day=“”;
#evaluates to false, so this block will not be
   executed
if($day) {
   print $day contains a string
}
```

Boolean values

- Get the opposite of a boolean value (! Operator)

```
Example (A program that expects a filename from the user):
print “Enter file name, please\n”;
$file=<>;
chomp($file); #remove \n from input
if (!$file) { #if $file is false (empty string)
   print “I need an input file to proceed\n”;
}
#try to process the file
```

die() function

- Raises an exception, which means that throws an error message and stops the execution of the program.
- So previous example revisited:

```
print “Enter file name, please\n”;
$file=<>;
chomp($file); #remove \n from input
if (!$file) { #if $file is false (empty string)
  die(“I need an input file to proceed\n”);
}
#process the file only if $file is defined
```

Ex 6. Using conditional expressions

- TODO: Write a program to get an exam score from the keyboard and prints out a message to the student.

Score	Message
Greater than or equal to 90	Excellent Performance!
Greater than or equal to 70 and less than 90	Good Performance!
Greater than or equal to 50 and less than 70	Uuff! That was close!
Less than 50	Sorry, try harder!

Hint: To read input from keyboard enter in your program

```
print "Enter the score of a student: ";
$score = <>;
```

Solution

```
#! /usr/bin/perl
print "Enter the score of a student: ";
$score = <>;
if($score>=90) {
  print "Excellent Performance!\n";
} elsif ($score>=70 && $score<90) {
  print "Good Performance!\n”;
} elsif ($score>=50 && $score<70) {
  print "Uuff! That was close!\n”;
} else {
  print "Sorry, try harder!\n";
}
```

Introduction to Perl programming
Session II

Antonio Hermoso

CRG Bioinformatics Core

Overview

- Loops
- Arrays
- Reading/Writing files

Statements and Blocks

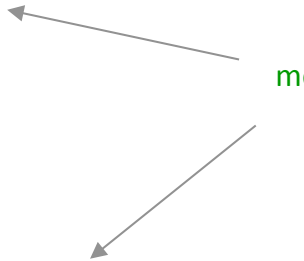
- Programs are composed of statements often grouped together into blocks
- A statement ends with a semicolon (;), which is optional for the last statement in a block
- A block is one or more statements usually surrounded by curly braces:

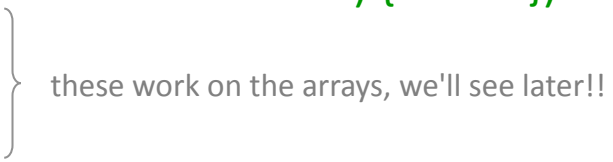
```
{
    $thousand = 1000;
    print $thousand;
}
```

Loops

- A *loop* allows you to repeatedly execute a block of statements

- There are several ways to loop in Perl:
  - while (CONDITION) {BLOCK}
  - do {BLOCK} while (CONDITION)
  - until (CONDITION) {BLOCK}
  - do {BLOCK} until (CONDITION)
  - for (INITIALIZATION; CONDITION; RE-INITIALIZATION) {BLOCK})
  - for VAR (LIST) {BLOCK})
  - foreach VAR (LIST) {BLOCK})





While Loop

- The while loop first tests the condition:
  - if true, it executes the block and then returns to the conditional to repeat the process
  - if false, it does nothing, and the loop is over

- Example:

```
$i = 1;
while ($i <= 1000) {
    print "$i\n";
    $i++;
}
```

?

IMP: do not forget to increment the variable

Code Layout

- Format A


```
while ($i) {
        if ($i) {
            print "$i\n";
          }
 }
```

- Format B


```
while ($i)
{
        if ($i)
         {
            print "$i\n";
          }
 }
```

- Format C

```
        while ($i)
        {
         if ($i)
 {
                print "$i\n";
 }
 }
```

- Format D

```
while($i){if($i){print "$i\n";}}
```

Do-while Loop

- In the *do-while* loop, the block is executed before the conditional test, and the test succeeds while the condition is true

- Example:

```
$i = 1000;
do {
    print "$i\n";
      $i--;
 } while ($i);
```

Until Loop

- Until loop is used to loop through a designated block of code until a specific condition is met (evaluated as true)
- It is the logical opposite of the *while* loop

- Example:

```
$i = 3;
until ($i) {
        print "$i\n";
        $i--;
}
```

?

Do-Until Loop

- In the *do-until* loop, the block is executed before the conditional test, and the test succeeds until the condition is true

- Example:

```
$i = 3;
do {
      print "$i\n";
      $i--;
} until ($i);
```

for (INITIALIZATION; CONDITION; RE-INITIALIZATION) {BLOCK}

For Loops

- The *for* loop makes it easy by including the variable initialization and the variable change in the loop statement

- Example:

```
for ($i = 1; $i <= 1000; $i++) {
        print "$i\n";
}
```

Moving around in a Loop

- next
  - ignore the current iteration
- last
  - terminates the loop
- What is the output for the following code snippet?

```
for ( $i = 0; $i < 20; $i++) {
        if ($i == 1 || $i == 5) { next; }
        elsif ($i == 7) { last; }
         else {print "$i\n";}
 }
```

?

Answer

0

2

3

4

6

Exercise

- Use a while loop to print the integer values from 1 to 10 on the screen:

12345678910

while (CONDITION) {BLOCK}

Answer

```
#!/path/to/perl -w
$i=1;
while ($i <= 10) {
    print $i;
    $i++;
}
```

Exercise

- Use a while loop to reproduce the following output:

1
22
333
4444
55555

TIP: you need to use a nested loop

Answer

```
#!/path/to/perl -w

$i = 1;
while ($i <= 5) {
    $j = 1;
    while ($j <= $i) {
            print $i;
            $j++;
    }
    print "\n";
    $i++;
}
```

Exercise

- Count the frequency of base G in the following DNA sequence:

  GATTAGCAGGGCAGT

TIP: you need to use a while loop for the length of the string, extract each base with substr, and use an if to check if the base is a G

substr EXPR,OFFSET,LENGTH

Examples:

```
my $dna="AAAATGG";
my $letter1=substr($dna,1,1);
print "$letter1\n";
>A
my $letter2=substr($dna,2,4);
print "$letter2\n";
>AATG
```

Answer

```
#!/path/to/perl -w

$DNA = "GATTAGCAGGGCAGT";

$countG = 0;  # initialize $countG and $currentPos
$currentPos = 0;

$DNAlength = length($DNA); # calculate the length of $DNA

while ($currentPos < $DNAlength) {
	$base = substr($DNA,$currentPos,1);
	if ($base eq "G") { # for each letter in the sequence check if it is the base G
		$countG++; # if 'yes' increment $countG
	}
	$currentPos++;
} # end of while loop

print "There are $countG G bases\n"; # print out the number of Gs
```

Arrays

Arrays

- Arrays are ordered lists of scalars
- Array variable is denoted by the @ symbol

```
@bases = ( "A", "C", "G","T");
```

- To access the whole array:

```
print @bases;  # prints : A C G T
```

Notice that you do not need to loop through the whole array to print it – Perl does this for you

Arrays cont.

- Array indexes start at 0
- To access one element of the array: use $
  - Why? Because every element in the array is a scalar

```
@molecules = ('DNA','RNA','Protein');
print "Here are the array elements:";
print "\nFirst element: ";
print $molecules[0];
print "\nSecond element: ";
print $molecules[1];
print "\nThird element: ";
print $molecules[2];
```

Positions:	*0*	*1*	*2*
Scalar values:	DNA	RNA	Protein

Schematic view of the array @molecules

Output

First element: DNA

Second element: RNA

Third element: Protein

Arrays cont.

- To find the index of the last element in the array

```
print $#bases;  #prints 3 in the previous example
```

- Other ways to find the number of elements in the array are:

```
$array_size = @bases; or $array_size = scalar(@bases);
```

Note: in our example, $array_size is 4 because there are 4 elements in the array @bases

Example: Numerical Sorting

```
#!/path/to/perl -w

@unsortedArray = (16, 12, 20, 10, 1, 77);
@sortedArray = sort {$a <=> $b} @unsortedArray;

print "@unsortedArray\n"; # prints 16 12 20 10 1 77
print "@sortedArray\n";     # prints 1 10 12 16 20 77
```

Sorting Arrays

- Perl has a built in function to *sort:*
  - In alphabetical order (default) with uppercase first

    ```
    @sortedArray = sort @unsortedArray;
    ```

    [equivalent to `@sortedArray = sort {$a cmp $b} @unsortedArray;`]

  - In a reverse alphabetical order

    ```
    @sortedArray = sort {$b cmp $a} @unsortedArray;
    ```

  - Numerically in ascending order

    ```
    @sortedArray = sort {$a <=> $b} @unsortedArray;
    ```

  - Numerically in descending order

    ```
    @sortedArray = sort {$b <=> $a} @unsortedArray;
    ```

Example: String Sorting

```
#!/path/to/perl -w

@unsortedArray = ("UAA", "UGA", "UAG");
@sortedArray = sort {$a cmp $b} @unsortedArray;

print "@unsortedArray\n"; # prints UAA UGA UAG
print "@sortedArray\n";     # prints UAA UAG UGA
```

Reversing an Array

- The *reverse* function reverses the order of the elements stored in an array:

```
@array = reverse (@array);
```

- Example:

```
@bases = ( "A", "C", "G","T");
print @bases;  # prints : A C G T

@bases = reverse (@bases);
print @bases;  # prints : T G C A
```

Example: playing a bit with your names

```
#!/path/to/perl -w

@names = ("elisa", "Laura", "angela", "astrid", "Maria", "andreas", "Federico",
"Susana","Alessandro");
print "1-names: @names\n\n";

@names = reverse(@names);
print "2-reversed: @names\n\n";

@names = sort (@names);
print "3-sorted: @names\n\n";

@names = sort {$b cmp $a} @names;
print "4-sorted desc: @names\n\n";
```

Output:

1-names: elisa Laura angela astrid Maria andreas Federico Susana Alessandro

2-reversed: Alessandro Susana Federico andreas Maria astrid angela Laura elisa

3-sorted: Alessandro Federico Laura Maria Susana andreas angela astrid elisa

4-sorted desc: elisa astrid angela andreas Susana Maria Laura Federico Alessandro

Foreach

- Foreach allows you to iterate over an array
- Example:

```
foreach $element (@array) {
    print "$element\n";
}
```

- This is similar to:

```
for ($i = 0; $i <= $#array; $i++) {
    print "$array[$i]\n";
}
```

Sorting with Foreach

- The sort function sorts the array and returns the list in sorted order

- Example:

```
@family = ("father","mother","son","daughter");

foreach $element (sort @family) {
    print "$element ";
}
```

- Prints the elements in sorted order:

```
daughter father mother son
```

For Loop - on the arrays

- The *for* loop allows you to iterate also the arrays

- Example:

```
@family = ("father","mother","son","daughter");

for $element (sort @family) {
  print "$element ";
}
```

Manipulating Arrays

String to Array: *split*

- Split a string into words and put into an array

```
@bases = split(";", "A;C;G;T");

#creates the same array as we saw previously @bases = ("A", "C",
    "G", "T");
```

- Split into characters

```
@bases = split("", "ACGT" );

# array @bases has 4 elements: A, C, G, T
```

  - NB: Split functions can be also used to prepare a list:

```
($first,$second,$third,$fourth) = split(";", "A;C;G;T");
```

Array to String: *join*

- Array of characters to string:

```
@aa = ("M", "N", "I", "D","K","L");
$pep_fragment = join("", @aa);

# pep_fragment = "MNIDKL"
```

- Array to space separated string:

```
@array = ("one", "two", "three");
$string = join(" ", @array);

# string = "one two three"
```

More examples...

- Join with any character you want:

```
@array = ("D", "v", "lop", "r");
$string = join("e", @array);

#  string = "Developer"
```

- Join with multiple characters:

```
@array = ("1", "2", "3", "4", "5");
$string = join("->", @array);

#  string = "1->2->3->4->5"
```

Add/remove elements *(at the end of the array)*

- To append to the end of an array:

```
@bases = ("A", "C", "G");
push (@bases, "T" );
print @bases;          #  prints A C G T
```

- To remove the last element of the array:

```
@bases = ("A", "C", "G", "T");
$base = pop (@bases);
print $base;           #  prints "T"
print @bases;          #  prints A C G
```

Add/remove elements (*at the beginning of the array)*

- To add an element to the beginning of an array:

```
@bases = ("A", "C", "T");
unshift (@array, "G");
print @bases;                    #  prints  G A C T
```

- To remove the first element of the array:

```
$base = shift @bases;
print $base;                     #  prints "G"
print @bases;                    #  prints  A C T
```

Reading/Writing Files

File Handlers

- Opening a File:

```
open (FH, "file.txt");
```

- Reading from a File

```
$line = <FH>;          # reads up to a newline character
```

- Closing a File

```
close (FH);
```

File Handlers

- Program to read the whole file content:

```
#!/path/to/perl -w

open (FH, "file.txt");

while ($line = <FH>) {
    print $line."\n";
}

close (FH);
```

Exercise: Write a program to print out a file

1) Download ENSG00000139618.fasta from

`http://nin.crg.es/perlCourse2012/ENSG00000139618.fasta`

2) Write a program called `readfile.pl` to print out the sequence of ENSG00000139618

3) Run `readfile.pl` (will print output into the screen [STDOUT]

4) Finally, type in the terminal (redirection usage):

```
perl readfile.pl > ouputname.txt
```

Solution

```
#!/path/to/perl -w

open (FH, ”ENSG00000139618.fasta");

while ($line = <FH>) {
    print $line."\n";
}

close (FH);
```

File Handlers cont.

- Opening a file for output:

```
open (FH, ">file.txt");
```

- Opening a file for appending:

```
open (FH, ">>file.txt");
```

- Exiting if opening a non-existing file:

```
open (FH, ">file.txt") || die "Could not open file.\n";
```

- Writing to a file:

```
print FH "Printing my first line.\n";
```

File Test Operators

- Another check to see if a file exists:

```
if (-e "file.txt") {
    #  The file exists!
}
```

- Other file test operators:

-r	readable
-x	executable
-d	is a directory
-T	is a text file

A program with File Handles

- Program to copy a file to a destination file:

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

open(FH1, "file.txt") || die "Could not open source file.\n";
open(FH2, ">newfile.txt");

while ($line = <FH1>) {
        print FH2 $line;
}

close FH1;
close FH2;
```

Some Default File Handles

- STDIN : Standard Input

```
$line = <STDIN>;          #  takes input from stdin
```

- STDOUT : Standard output

```
print STDOUT ”This prints out something\n";
```

- STDERR : Standard Error

```
print STDERR "Error!!\n";
```

Chomp and Chop

- Chomp: function that deletes a trailing newline from the end of a string

```
$line  = "this is the first line of text\n";
chomp $line;       #  removes the new line character
print $line;       # prints "this is the first line of
                   # text" without returning
```

- Chop: function that chops off the last character of a string

```
$line = "this is the first line of text";
chop $line;
print $line;       #prints "this is the first line of tex"
```

Exercise

- Download the file human_genes.txt containing the coordinates of all the human genes (take a look at it)
- Write a program to print all the genes longer than 1Mb (1000000 bp)

- Steps:
  1. Download file from `http://nin.crg.es/perlCourse2012/human_genes.txt`
  1. Read all the lines of file human_genes.txt, and skip the header
  2. Compute the gene length and assess whether the gene is longer than 1Mb
  3. If yes, print the gene name and the length

Answer

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

open(FH, “/path_to_the_file/human_genes.txt") || die "Could not open source file.\n";

$i = 0;
while ($line = <FH>) {
	if ($i==0) {
		$i++;
		next;
	}
	($gene_name,$ensembl_id,$chr,$gene_start,$gene_end,$gene_strand,$gene_band,$transcript_num,
	$gene_biotype,$gene_status)= split("\t", $line);
	 $gene_length = ($gene_end - $gene_start) + 1;

	if ($gene_length > 1000000) {
		print "Gene $ensembl_id ($gene_name) has length $gene_length\n";
	}
}
close FH;
```

Exercise

- Using the same file human_genes.txt
- Write a program to print the number of genes with more than 20 transcripts

- Steps:
  1. Read all the lines of file human_genes.txt, and skip the header
  2. Increment a variable $gene_count if the gene has more than 20 transcript
  3. Print the count

Answer

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

open(FH, “/path_to_the_file/human_genes.txt") || die "Could not open source file.\n";
$i = 0;
$gene_count = 0;

while ($line = <FH>) {
        if ($i==0) {
                $i++;
                next;
        }

        @columns = split("\t", $line);
        $transcript_num = $columns[7];

        if ($transcript_num > 20) {
                $gene_count++;
        }
}

print "$gene_count genes have more than 20 transcripts\n";

close FH;
```

Exercise

- Write a program named `count_nucleotides1.pl` to determine the frequency of nucleotides in a DNA sequence provided by file

- Steps:

1)Download file `sequence.txt` by typing:

`http://nin.crg.es/perlCourse2012/sequence.txt`

2)Read in DNA from `sequence.txt`

3)Remove white spaces in the sequence and then creates an arrays of nucleotides

4)Look at each base in a loop to count the different nucleotides

Adapted from example 5-4 of the book "Beginning Perl for Bioinformatics", J. Tisdall

Example Program

Step 1- Read DNA from `sequence.txt`:

```
#!/path/to/perl -w

open (FH, $file) || die "Could not open file.\n";

@DNA = <FH>;
print "working on DNA:\n@DNA\n";

close (FH);
```

Example Program cont.

Step 2- Remove white spaces in the sequence and then creates an arrays of nucleotides

```
$DNA = join('', @DNA); # put the DNA sequence into a string

$DNA =~ s/\s//g; # remove whitespace

@DNA = split('', $DNA); # create an array of nucleotides

print "now DNA is:\n@DNA\n";
```

This is a regular expression! We'll talk about this next time!!

Example Program cont.

Step 3- Look at each base in a loop to count the different nucleotides

```
($A,$C,$G,$T) = (0,0,0,0);
foreach $base (@DNA) {
        if ($base eq 'A') {
                $A++;
        } elsif ($base eq 'C') {
                $C++;
        } elsif ($base eq 'G') {
                $G++;
        } elsif ($base eq 'T') {
                $T++;
        } else {
                print "Error - I do not recognize this base: $base\n";
        }
}
print "A = $A\tC = $C\tG = $G\tT = $T\n\n";
```

Introduction to Perl programming
Session III

Ernesto Lowy

CRG Bioinformatics core

REGULAR EXPRESSIONS

REGEX

- Fast, flexible and reliable method to look for patterns in strings
- Strong support in Perl
- Also in other programming languages and in awk,sed,emacs...

What is a REGEX?

- A pattern/template that match/not match a given string
- Almost always used in a conditional that returns True/False

Ex.

```
$dna='AAAAATGAAAAA';

if ($dna =~ /ATG/) {
     print "it matched!\n";

}

>it matched!
>
```

Binding operator

What is a REGEX?

Ex.

```
$dna='ATGAAAATGAAAAA';


if ($dna =~ /ATG/) {
    print “it matched!\n”;
}

>it matched!
>
```

What is a REGEX?

- \t or \n also can be matched in REGEX

Ex.

```
$names=”peter\tmaria”;

if ($names =~ /peter\tmaria/)
{
    print “$names\n”;
}

>peter  maria
>
```

EXERCISE

- Download textdemo.txt from:

`http://nin.crg.es/perlCourse2012/textdemo.txt`

- Write a Perl script that read this file line per line and only prints out the lines that contain the word Darwin

ANSWER

```
$file="textdemo.txt";

open FH,”$file"; #open filehandle

while($line=<FH>) {
    chomp($line);
    #regex
    if ($line=~/Darwin/) {
        print "$line\n";
    }
}

close FH; #close filehandle
```

Metacharacter (dot operator)

- Allow to use a simple pattern to match more than one string
- the dot (.) matches any single character except “\n”

Ex.

```
$name=”betty”;
if ($names =~ /bet.y/) {
        print “it matched!\n”;
}
```

It will not match:

```
betsey
betseey
```

It will match:

```
betsy
bet=y
bet-y
....
```

Simple quantifiers

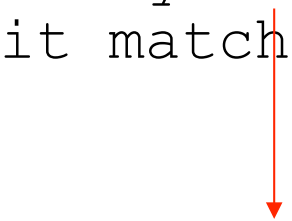
- When one needs to repeat something in the pattern
- * (asterisk) means match preceding item 0 or more times
- + (plus) means match preceding item 1 or more times

```
if ($name=~/frey\t*barney/) {
     print "it matched!\n";
}

$name="fred\tbarney";
$name="fred\t\tbarney";
$name="fred\t\t\tbarney\tand\tjohn";
$name="fredbarney";
```

Simple quantifiers

```
if ($name=~/frey\t+barney/) {
    print "it matched!\n";
}
```




```
$name="fredbarney";
????????
```

Simple quantifiers

- Match exactly at least n times with { }

- Ex:

```
$dna_string=”TTTTAAAAAA”; #has this string at
least five As?

if ($dna_string=~/A{5}/) {
     print “this string has at least five As\n”;
}
```

Grouping things in REGEX

- Parentheses (**( )**) are used for this

Ex:

/fred+/ will match freddddddddd

/(fred)+/ will match fredfred or fred or and so on but will not match freafrea

Character classes

- List of possible characters inside brackets ([ ])
- Important: It matches only a single character but this can be any of the characters within brackets

```
$a=2;
if ($a=~/[0123456789]/) {
     print “Scalar variable is a digit!\n”;
}
```

- Same example but with less typing:

```
$a=2;
if ($a=~/[0-9]/) {
     print “Scalar variable is a digit!\n”;
}
```

Character classes

- Some character classes appear so frequently that have shortcuts

Class	Shortcut
[0-9]	\d
[A-Za-z0-9]	\w
[\f\t\n\r]	\s

Character classes

- All character classes can be negated using the caret (^) symbol or using the corresponding capital letter

Negated class	Shortcut	Capital-letter
[^0-9]	[^\d]	\D
[^A-Za-z0-9]	[^\w]	\W
[^\f\t\n\r]	[^\s]	\S

```
$a="a";
if ($a=~/\D/) {
    print "It is not a digit!\n";
}
```

Will print:

```
>It is not a digit!
>
```

Anchors

- Allow to match a pattern but only at the beginning or end of a string
- Caret (^) symbol match a pattern at the beginning of the string
- Dollar ($) symbol match a pattern at the end of the string

```
$string=”fred is 23 years old”;

if ($string=~/^fred/) {
     print “we are talking about fred!\n”;
}
```

Will print:

>we are talking about fred!
>

Anchors

```
$string=”is fred 23 years old”;

if ($string=~/^fred/) {
    print “we are talking about fred!\n”;
}
```

Will not match!

Anchors

- Match at the end of the string with $

```
$string=”they are 3”;

if ($string=~/\d$/) {
    print “\$string ends in a number\n”;
}


>$string ends in a number
>
```

Anchors

```
$string=”3 they are”;

if ($string=~/\d$/) {
    print “\$string ends in a number\n”;
}
```

Will not match!

EXERCISE

- Download demo.fasta (multifasta file with DNA sequences) by typing:
  `http://nin.crg.es/perlCourse2012/demo.fasta`
- Write a Perl script to parse demo.fasta and print out the lines that contain the IDs for the different sequences

Tip. Remember that the Fasta format has always the following format:

```
>seq1
ACGTGGGTGTGATG
```

ANSWER

```
$file="demo.fasta";

open FH,”$file";

while($line=<FH>) {
    chomp($line);
    #match only lines starting with >
    if ($line=~/^>/) {
        print "$line\n";
    }
}

close FH;
```

Extracting the matches

- Parentheses () allow to recover the parts of a string that matched
- Matches will be kept in special variables called $1 , $2 , etc
- For example:

```
$a=”Hello there, neighbor”;

if ($a=~/\s(\w+),/) {
        print “the word was $1\n”;
}
```

Will print:

```
>there
>
```

Extracting the matches

```
$a=”Hello there, neighbor”;

if ($a=~/(\w+) (\w+), (\w+)/) {
    print “words were $1 $2 $3\n”;
}
```

Will print:

```
>words were Hello there neighbor
>
```

EXERCISE

- Download demo.fasta (multifasta file with DNA sequences) by typing:

http://nin.crg.es/perlCourse2012/demo.fasta

- Write a Perl script to parse demo.fasta and print out the part of the ID that differentiates one sequence from the other. For example:

```
>seq1
>seq2
>seq3
...
```

Our script will print:

```
1
2
3
...
```

Tip. Remember that the Fasta format has always the following format:

```
>seq1
ACGTGGGTGTGATG
```

ANSWER

```
$file="demo.fasta";

open FH,”$file";
while($line=<FH>) {
    chomp($line);
    #capture the digits after
    #the word seq
    if ($line=~/^>seq(\d+)/) {
        print "$1\n";
    }
}
close FH;
```

Processing text with REGEX

- So far REGEX were used to check if a given string has a given pattern inside, but we did not modify the original string
- Substitution operator:

```
$string=”Homer Simpson”;

$string=~s/Homer/Bart/;

print “Now we have $string\n”;
```

Will print:

```
>Now we have Bart Simpson
>
```

Processing text with REGEX

- Substituting globally

Example (Removing extra tabspaces in a string):

```
$string=”Hello,\tI am attending\t\t a Perl course\n”;
print $string; #print $string before removing
tabspaces

$string=s/\t+/ /g;
print $string; #print $string after removing
tabspaces
```

Will print:

```
>Hello,  I am attending    a
Perl course
>Hello, I am attending a Perl
course
```

EXERCISE

1. Open `gedit` and create a file called `substituteTs.pl`
2. Create a variable called `$seq` containing the following sequence:

`AACCCttttGGGTTTTTGTCGTAGAAAAAAAA`

3. Subsitute all Ts or ts in `$seq` by Us
4. Print the contents of `$seq`
5. Execute `substituteTs.pl`

ANSWER

```
$seq="AACCCttttGGGTTTTTGTCGTAGAAAAAAAA";

$seq=~ s/Tt/U/g;

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

Processing text with REGEX

- Transliterator operator

```
tr/SEARCHLIST/REPLACEMENTLIST/
```

- Definition:

it replaces all occurrences of the characters in SEARCHLIST with the characters in REPLACEMENTLIST

- Example I:

```
$string = 'the cat sat on the mat.';
$string =~ tr/a/o/;

print "$string\n";
```

Will print:

>the cot sot on the mot.
>

Processing text with REGEX

- Transliterator operator

- Example II:

```
$string = 'the cat sat on the mat.';
$string =~ tr/at/ol/;

print "$string\n";
```

Will print:

>lhe col sol on lhe mol
>

Exercise

- Calculate the reverse complementary of a DNA sequence using the tr/// operator
- Answer:

```
#!/usr/bin/perl

$dna="ACGGTTGGAAAACGTTTGCGCGCGCGATGGCCCCGAACG";
print "the original sequence is:\n$dna\n";

#reverse string
$revcom=reverse $dna;
print "Reversed sequence is:\n$revcom\n";

#calculate the complementary for each nucleotide
$revcom=~tr/ACGT/TGCA/;
print "Reverse complement is:\n$revcom\n";
```

Introduction to Perl programming
Session IV

Ernesto Lowy

CRG Bioinformatics core

HASHES

- **Very Useful**
- **Make Perl a very powerful language**
- **But... what is a Hash?**

  Is another data structure (like arrays) that holds any number

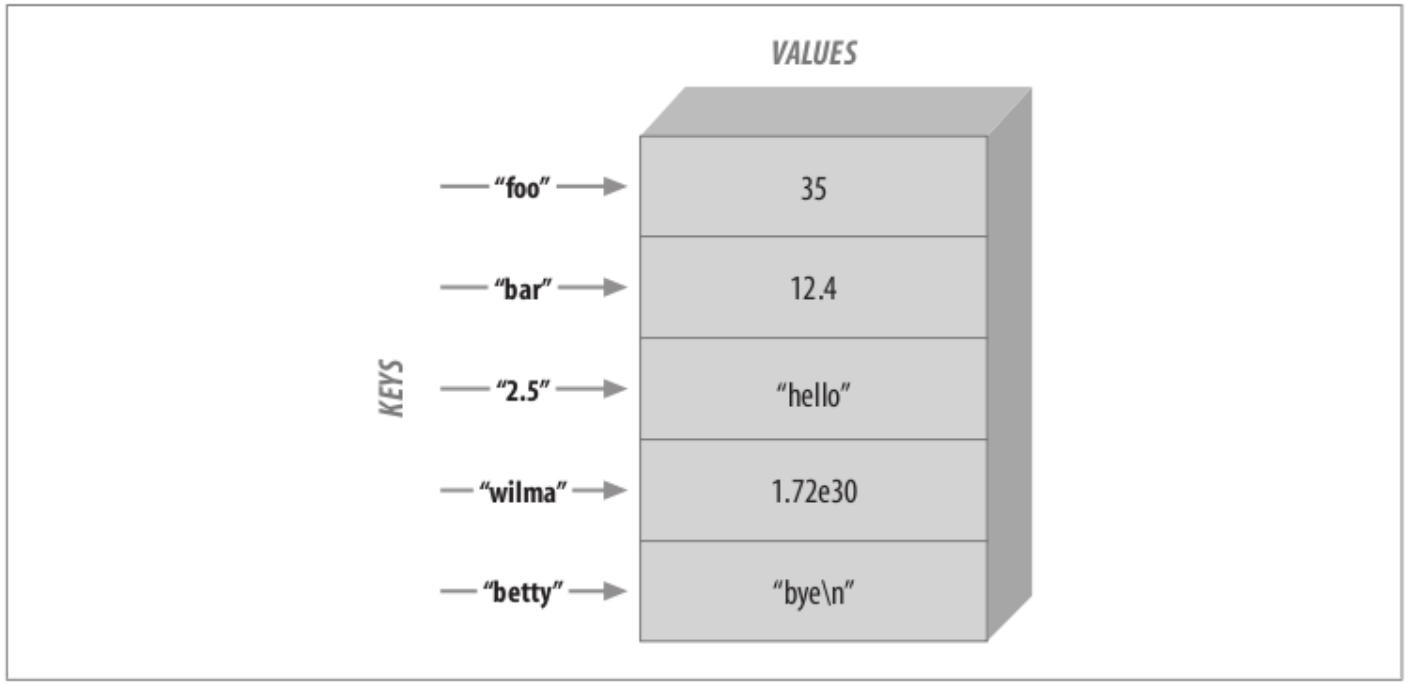
  (a collection) of values

  Unlike the arrays (where the values are indexed by numbers)

  In hashes we'll look up the data by name

HASHES

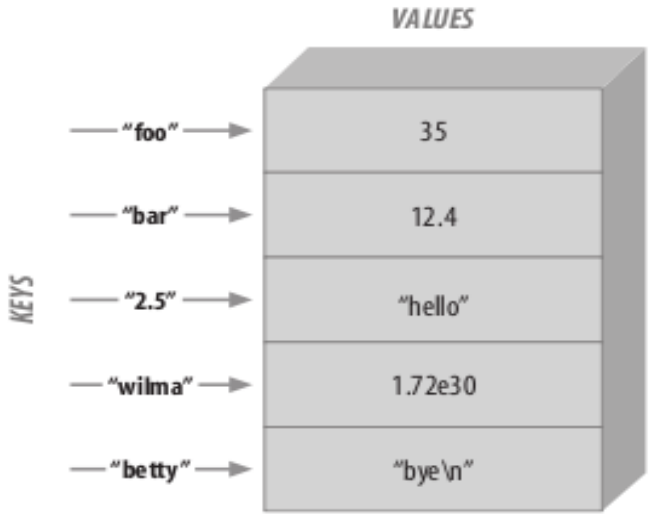
- We access the data through the association between a key and a value
- Keys are arbitrary strings
- They are unique (cannot exist the same key associated to different values)
- Values can be numbers,strings,undef values

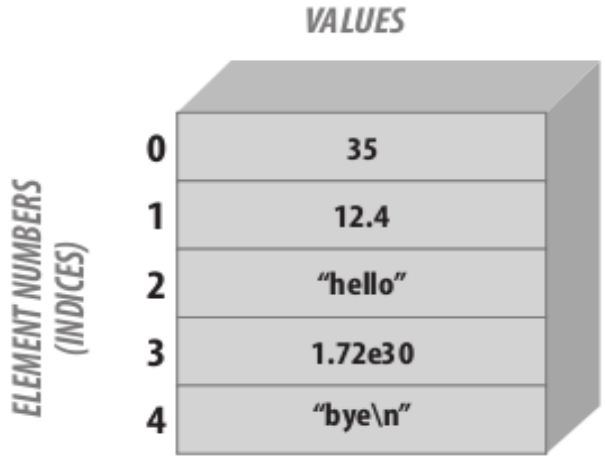


Extracted from Learning Perl (Tom Phoenix, Randal L. Schwartz)

HASHES vs ARRAYS

- Keys are unordered (so we can look up any item quickly)
- Indices of an array are ordered





Extracted from Learning Perl (Tom Phoenix, Randal L. Schwartz)

CREATING A HASH

```
%cities = (
    "Rome" => "Italy",
    "London" => "UK",
    "Paris" => "France",
    "New York" => "United States",
    "Lisbon" => "Portugal"
    );
```

KEYS

VALUES

CREATING A HASH

- Which is the same than (less visually clear):

```
my %cities= (“Rome” => “Italy”,“London” =>
“UK”,“Paris” => “France”,“New York” => “United
States”,“Lisbon” => “Portugal”);
```

HASH ELEMENT ACCESS

- Syntax is:

```
$hash{$some_key}
```

- Similar to arrays were we had (square brackets instead of curly brackets)

```
$array[0]
```

- Example:

```
print $cities{“Paris”},”\n”;
```

- Will print:

```
>France
```

ADD DATA INTO THE HASH

- Syntax is:

```
#add new key-value pair into %cities
$cities{“Madrid”}=”Spain”;
```

- Now %cities will be:

```
%cities= (
     “Rome” => “Italy”,
     “London” => “UK”,
     “Paris” => “France”,
     “New York” => “United States”,
     “Lisbon” => “Portugal”,
     “Madrid” => “Spain”
);
```

HASH FUNCTIONS

KEYS FUNCTION

- Returns an array with all the keys in the hash

Example I:

```
my @certain_cities=keys %cities;

foreach $this_city (@certain_cities) {
    print $this_city,"\n";
}
```

Will print:

```
>Paris
>Madrid
>London
>Lisbon
>Rome
>New York
```

Unsorted

HASH FUNCTIONS

KEYS FUNCTION

Example II:

```
my @certain_cities=sort keys %cities;

foreach $this_city (@certain_cities) {
    print $this_city,"\n";
}
```

Will print:

```
>Lisbon
>London
>Madrid
>New York
>Paris
>Rome
```

Sorted

HASH FUNCTIONS

KEYS FUNCTION

Example III:

- Same than previous example but less typing:

```
foreach $this_city (sort keys %cities) {
    print $this_city,"\n";
}
```

HASH FUNCTIONS

VALUES FUNCTION

- Returns an array with all the values in the hash

Example I:

```
@certain_countries=values %cities;

foreach $this_country (@certain_countries) {
     print $this_country,"\n";
}
```

Will print:
>France
>UK
>Portugal
>Spain
>Italy
>United States

Unsorted

HASH FUNCTIONS

VALUES FUNCTION

- Returns an array with all the values in the hash

Example II:

```
my @certain_countries=sort values %cities;

foreach $this_country (@certain_countries) {
     print $this_country,"\n";
}
```

Will print:

```
>France
>Italy
>Portugal
>Spain
>UK
>United States
```

Sorted

EXERCISE

1) Create a hash called %names with the following pairs (First Name/Last Name):

First Name	Last Name
James	Taylor
Elisabeth	Bacon
Helen	Smith
Henry	Logan

2) Use a foreach to print all values in the screen with not particular order

3) Use a foreach to print all values, but this time print the values sorted alphabetically

ANSWER

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

#create hash
%names= (
    "James"=>"Taylor",
    "Elisabeth"=>"Bacon",
    "Helen"=>"Smith",
    "Henry"=>"Logan"
    );

print "Unsorted:\n";
#print each value in the screen unordered
foreach $last_name (values %names) {
    print "$last_name\n";
}

print "\nSorted:\n";
#print each value in the screen sorted alphabetically
foreach $last_name (sort values %names) {
    print "$last_name\n";
}
```

HASH FUNCTIONS

EACH FUNCTION

- To iterate over an entire hash (or examine each element of a hash)
- Returns a key-value pair as a two element list
- It has to be used in a while loop

Example:

```
while(@a=each %cities) {
     $key=$a[0];
     $value=$a[1];
     print "$key\t$value\n";
}
```

Will print:

```
>Paris    France
>London   UK
>Lisbon   Portugal
>Barcelona     Spain
>New York      United States
```

HASH FUNCTIONS

EACH FUNCTION

The same but with less typing

```
while(($key,$value)=each %cities) {
    print "$key\t$value\n";
}
```

EXERCISE

Use a hash to remove duplicated entries

1) http://nin.crg.es/perlCourse2012/human_data.txt
This files contain 2 tab separated columns
(1st column=gene_name; 2nd column=ensembl ID)
2) Open `human_data.txt` and check if there are duplicated entries
3) Create a program called `remove_duplicates.pl` containing a hash called `%hash` for which:
key=1st column or gene_name
value=2nd column or ensembl ID
Print the entire hash using the `each` function
Hint. Each line in the file must be split into the 2 columns using the tab separator (using the split function) and added into the hash.
4) Execute `remove_duplicates.pl` and redirect the output into a file called `human_data_nodupl.txt`
5) Check that all the duplicated entries were removed

ANSWER

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

%hash; #declare the hash

open(FH,"human_data.txt") || die "Could not open file.\n";

while($line=<FH>) {
    chomp($line);
    ($geneId,$ensId)=split/\t/,$line;
    # $geneId=key and $ensId=value
    $hash{$geneId}=$ensId;
}

close FH;

# print non duplicated key/value pairs
while(($key,$value)=each %hash) {
    print "$key\t$value\n";
}
```

HASH FUNCTIONS

EXISTS FUNCTION

- To see whether a key exists in the hash
- Returns a true value if the given key exists in the hash

Example:

```
#initialize %ages
my %ages= (
    "fred"=>10,
    "henry"=>35,
    "peter"=>40,
    );

#check if “fred” exists in %ages
if (exists($ages{"fred"})) {
    print "fred key EXISTS in this hash\n";
} else {
    print "fred does NOT EXIST in this hash\n";
}
```

EXERCISE

Use a hash to remove duplicated entries

1) Download `human_data.txt` from the web by typing:

http://nin.crg.es/perlCourse2012/human_data.txt

This files contain 2 tab separated columns

(1st column=gene_name; 2nd column=ensembl ID)

2) Create a hash called `%hash` for which:

key=1st column or gene_name

value=2nd column or ensembl ID

Hint. Each line in the file must be split into the 2 columns using the tab separator (using the split function) and added into the hash.

Important. You have to check with the exists function if there is a gene name associated to 2 different ensembl Ids. If this is the case then stop the execution of the program with die()

For example:

ZNF684 ENSG00000117010

ZNF684 ENSG00000117015

3) print the entire hash using the each function

ANSWER

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

%hash; #declare the hash

open(FH,"human_data.txt") || die "Could not open file.\n";

while($line=<FH>) {
    chomp($line);
    ($geneId,$ensId)=split/\t/,$line;
    #check if this $geneId already exists in %hash
    if (exists($hash{$geneId})) {
              $ens=$hash{$geneId};
              if ($ens ne $ensId) {
              die("Inconsistency!. This gene $geneId has 2 different ens IDs: $ensId
and $ens\n");
              }
    } else {
              #store $geneId/$ensId in the hash
              $hash{$geneId}=$ensId;
    }
}

close FH;

# print non duplicated key/value pairs
while(($key,$value)=each %hash) {
    print "$key\t$value\n";
}
```

HASH FUNCTIONS

DELETE FUNCTION

- Removes the given key (and its corresponding value)

from the hash

- Example:

```
#initialize %phone_numbers

my %phone_numbers= (
    "carol"=>687653720,
    "susan"=>66078665,
    "ramon"=>67898674,
    );

#delete “carol”=>687653720 pair
delete($phone_numbers{“carol”});
```

HASH FUNCTIONS

DELETE FUNCTION

- Check if the key/value pair was removed

```
foreach $key (keys %phone_numbers) {
    print "$key\t$phone_numbers{$key}\n";
}
```

Will print:

```
>ramon  67898674
>susan  66078665
```

EXERCISE

Write a second version of `count_nucleotides.pl` called `count_nucleotides2.pl` to determine the frequency of nucleotides in a DNA sequence but using a hash this time

Steps:

1) Download file `sequence.txt` by typing:

http://nin.crg.es/perlCourse2012/sequence.txt

2) Read in the sequence from the file using a while loop

3) split the sequence into its nucleotides using split

4) print all counts with the each function

ANSWER

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

open(FH,"sequence.txt") || die "Could not open file.\n";

while($line=<FH>) {
     chomp($line);
     @DNA=split('',$line);
     foreach $nt (@DNA) {
          $counts{$nt}++;
     }
}

close FH;

while(($nt,$count)=each %counts) {
    print "$nt\t$count\n";
}
```

SORT A HASH BY VALUES

- It is slightly trickier than sorting by keys

Example:

```
#hash with number of occurrences of the different words in a text
%hash=(
      “the”=>20,
      “a”=>10,
      “house”=>2,
      “car”=>3,
      “red”=>4
);

print “Unsorted hash:\n”;
while (($word,$count)=each %hash) {
      print “$word\t$count\n”;
}

#do the sorting
@sorted_count=sort {$hash{$b}<=>$hash{$a}} keys %hash;

print “Sorted by values:\n”;
foreach $word (@sorted_count) {
      print “$word\t$hash{$word}\n”;
}
```

SORT A HASH BY VALUES

Will print:

```
Unsorted hash:
house     2
the  20
a    10
red  4
car  3
Sorted by values:
house     2
car  3
red  4
a    10
the  20
```

EXERCISE

Sort a hash by Values

1) Download `positions.txt` (ensembl genes/starting positions) from the web:

`http://nin.crg.es/perlCourse2012/positions.txt`

This files contain 2 tab separated columns

(1st column=Ensembl ID; 2nd column=positions in chromosome 1)

File is not sorted by values

2) Create a hash called %chromosomal for which:

key=1st Ensembl ID

value=2nd positions

Hint. Each line in the file must be split into the 2 columns using the tab separator (using the split function) and added into the hash.

3) sort %chromosomal by positions (values)

4) print contents of %chromosomal with a foreach

ANSWER

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

#hash declaration
%chromosomal;

open(FH,"positions.txt") || die "Could not open file.\n";
#read file contents line per line
while($line=<FH>) {
    chomp($line);
    ($ensId,$position)=split/\t/,$line;
    #add key/value pair in %chromosomal
    $chromosomal{$ensId}=$position;
}
close FH;

#do the sorting
@sorted_positions=sort {$chromosomal{$a}<=>$chromosomal{$b}} keys\
%chromosomal;

#print %chromosomal contents
foreach $position (@sorted_positions) {
     print "$position\t$chromosomal{$position}\n";
}
```

Introduction to Perl programming
Session V

Antonio Hermoso

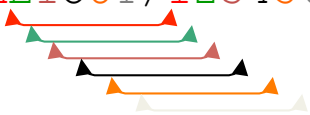
CRG Bioinformatics Core

Overview

- Transliteration operator `tr`
- Subroutines (Perl functions)
- Defining local variables with `my`
- `use strict;`

Transliteration operator: *tr*

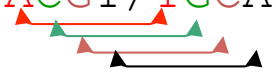
- Translations are like substitutions, but they happen only on a letter by letter basis
- Examples:
  - Change all vowels to upper case
    - `$string =~ tr/aeiouy/AEIOUY/;`
  - Change everything to upper case
    - `$string =~ tr/[a-z]/[A-Z]/;`
  - Change everything to lower case
    - `$string =~ tr/[A-Z]/[a-z]/;`
  - Change all vowels to numbers
    - `$string =~ tr/AEIOUY/123456/;`



Transliterator operator *tr*

- More examples:
  - Change bases to their complements:

```
$DNA = 'ACGTTTAA';
$DNA =~ tr/ACGT/TGCA/; #produces TGCAAATT
```



  - Count the number of a particular character in a string:

```
$DNA = 'ACGTTTAA';
$count_A = ($DNA =~ tr/Aa//);
$count_G = ($DNA =~ tr/Gg//);
print "A: $count_A - G: $count_G\n";
# prints: A: 3 - G:1
```

Subroutines

- A user-defined function or *subroutine* is defined in Perl as follows:

```
sub subname {
    statement1;
    statement2;
    statement3;
}
```

- Simple example:

```
sub hello {
    print "hello world!\n";
}
```

Subroutines cont.

- Subroutine can be anywhere in your program text they are skipped on execution), but it is most common to put them at the end of the file
- You can call a subroutine using its name followed by a parenthesized list of arguments
- Within the subroutine body, you may use any variable from the main program (variables in Perl are *global* by default)

```
#!/usr/local/bin/perl -w
$user = ”guglielmo";
hello();
print "goodbye $user!\n";
sub hello {
    print "hello $user!\n";
}
```

Calling a Subroutines

- You can also use variables from the subroutine back in the main program (it is the same global variable):

```
#!/usr/local/bin/perl -w
$a = 1; $b = 2;
$sum = 0;
sum_a_and_b();
print "sum of $a plus $b: $sum\n";
sub sum_a_and_b{
    $sum = $a + $b;
}
```

prints => `sum of 1 plus 2: 3`

Returning Values

- You can return a value from a function, and use it in any expression:

```
#!/usr/local/bin/perl -w
$a = 1;
$b = 2;
$c = sum_a_and_b() + 1;
print "value of c: $c\n";
sub sum_a_and_b {
    return $a + $b;
}
```

prints => `value of c: 4`

Returning Values

- A subroutine can also return a list of values:

```
#!/usr/local/bin/perl -w
$a = 1;
$b = 2;
@c = list_of_a_and_b();
print "list of c: @c\n";
sub list_of_a_and_b{
    return ($a,$b);
}
```

prints => `list of c: 1 2`

Returning Values

- Example: print the maximum of 2 numbers

```
#!/usr/local/bin/perl -w
$a = 1;
$b = 2;
$max = max_of_a_and_b();
print "max: $max\n";
sub max_of_a_and_b{
    if ($a > $b){
        return $a;
    } else {
        return $b;
    }
}
```

prints => `max: 2`

Arguments

- You can also pass arguments to a subroutine
- The arguments are assigned to a list in a special variable @_ for the duration of the subroutine

```
#!/usr/local/bin/perl -w
$a = 1;
$b = 2;
$max = max($a,$b);
print "max: $max\n";
sub max{
    if ($_[0] > $_[1]){
        return $_[0];
    } else {
        return $_[1];
    }
}
```

prints => max: 2

Arguments

- A more general way to write max() with no limit on the number of arguments:

```
#!/usr/local/bin/perl -w
$a = 1;
$b = 2;
$max = max($a,$b,5);
print "max: $max\n";
sub max{
    $max = 0;
    foreach $n (@_){
        if($n > $max){
            $max = $n;
        }
    }
    return $max;
}
```

prints => max: 5

Arguments

- Don’t confuse `$_` and `@_`
- Excess parameters are ignored if you don’t use them
- Insufficient parameters simply return `undef` if you look beyond the end of the `@_` array
- `@_` is local to the subroutine.

Local Variables

- You can create local versions of scalar, array and hash variables with the my() operator.

```
#!/usr/local/bin/perl -w
$a = 1; $b = 2;
$max = 0;
$max1 = max($a, $b, 5);
print "max1: $max1\n";
print "max : $max\n";
sub max{
    my($max,$n);  # local variables
    $max = 0;
    foreach $n (@_){
        if ($n > $max){
            $max = $n;
        }
    }
    return $max;
}
```

prints =>
```
max1: 5
max : 0
```

Local Variables

- You can initialize local variables:

```
#!/usr/local/bin/perl -w
$a = 1; $b = 2;
$max = 0;
$max1 = max($a, $b, 5);
print "max1: $max1\n";
print "max : $max\n";
sub max {
    my($max,$n) = (0,0);  # local
    foreach $n (@_){
        if ($n > $max){
            $max = $n;
        }
    }
    return $max;
}
```

prints =>
```
max1: 5
max : 0
```

Local Variables

- You can also load local variables directly from `@_`:

```
#!/usr/local/bin/perl -w
$a = 1;
$b = 2;
$max = max($a, $b);
print "max: $max\n";
sub max{
    my($n1, $n2) = @_;
    if ($n1 > $n2){
        return $n1;
    } else {
        return $n2;
    }
}
```

prints => `max: 2`

use strict

- You can force all variables to require declaration with `my()` by starting your program with: `use strict;`

```
#!/usr/local/bin/perl -w
use strict;
my $a = 1;    # declare and initialize $a
my $b = 2;    # declare and initialize $b
my $max = max($a, $b);  # declare and initialize
print "max: $max\n";
sub max{
     my($n1, $n2) = @_; # declare locals from @_
     if($n1 > $n2){
       return $n1;
     } else{
       return $n2;
     }
}
```

prints => `max: 2`

`use strict`

- `use strict` effectively makes all variables local
- Typing mistakes are easier to catch with `use strict`, because you can no longer accidentally reference `$billl` instead of `$bill`
- Programs also run a bit faster with `use strict`
- For these reasons, many programmers automatically begin every Perl program with `use strict`
- It is up to you which style you prefer

Exercise 1

- Write a function to concatenate 2 strings

```
sub concatenate {
    my($string1,$string2) = @_;
    my $concatenation = $string1.$string2;
    return $concatenation;
}

# example call:
my $dnastring = concatenate("atctg","ATC");
```

Exercise 2

- Write a function to compute reverse complement of a DNA string

```
sub revcom {
    my ($dna) = @_;
    my $revcom = reverse $dna;
    $revcom =~ tr/ACGTacgt/TGCAtgca/;
    return $revcom;
}

# example call:
my $revcomDNA = revcom("atctgATC");
```

Exercise 3

- Write a function to count the numbers of nucleotides in a given DNA sequence

```
sub countNs {
    my ($dna) = @_;
    my $As = ($dna =~ tr/Aa//);
    my $Gs = ($dna =~ tr/Gg//);
    my $Cs = ($dna =~ tr/Cc//);
    my $Ts = ($dna =~ tr/Tt//);
    return ($As,$Gs,$Cs,$Ts);
}

# example call:
my($As,$Gs,$Cs,$Ts) = countNs("atctgATC");
```

Exercise 4

- Create a file “functions.pm” and copy/paste the 3 functions you have just written in it.
- Note: When one creates a Perl module, it has to return a true value. For this you have to add:

  ```
  1;
  ```

  at the end of the file
- download exons from BRCA2-001 (ENSG00000139618) from:

`http://nin.crg.es/perlCourse2012/BRCA2-001.fasta`

Exercise 4

- Write a script to:
  - Use require “functions.pm”; to include functions
  - Open/read the file containing exon sequences
  - Join all exons together into $seq
  - Calculcate/print revcom of $seq
  - Calculate/count the numbers of Ns in $seq:
    - $As,$Ts,$Gs,$Cs

Exercise 4

```
#!/opt/local/bin/perl -w
use strict;
require ("functions.pm");

# open file containing exon sequences
open (FH, "ENST00000380152_exons.fa");

# join all exons together
my $seq;
while (my $line = <FH>) {
    if ($line =~ /^>/) {
     next;
    }
    chomp ($line);
    $seq = concatenate ($seq,$line);
}
close (FH);
print "Sequence is:\n$seq\n\n";

# calculate revcom
my $revcom_seq = revcom ($seq);
print "REVCOM sequence is:\n$revcom_seq\n\n";
```

```
# count the numbers of nucleotides
my ($As,$Gs,$Cs,$Ts) = countNs ($seq);
print "As: $As\tGs: $Gs\tCs: $Cs\tTs: $Ts\n";
```

The END!!!

Thanks all for your patience!

Congratulations!!!

We hope to see you
soon with many
impossible questions
on Perl
programming!!!

REFERENCE CHART

Basic Unix: commands

Path

pwd ← get current path

ls ← list folder content

ls -l ← list folder content in long format

cd ← change to home folder

cd ../../relative/path/

cd /absolute/path/

Files

touch <file_name> ← change timestamp

less <file_name> ← show file content

cp <file1> <file2> ← copy file1 to file2

mv <file_name> <new_file> ← move file

rm <file_name> <new_file> ← delete file

cat <file1> <file2>← concatenate files

Folders

mkdir <dir_name> ← make

rmdir <dir_name> ← delete

rm -rf <dir_name> ← delete

cp -rf <dir1> <dir2> ← copy

mv -rf <dir1> <dir2> ← move

Other

<command> -h ← command help

man <command>← manual pages

ps alh ← list process in human readable format

kill ← stop program by process ID

zip <file_name> ← compress file

unzip <file_name> ← uncompress file

Basic Unix: Redirection & Piping

Redirection:

- < ← Input from a file

perl program.pl < parameter.file

- > ← Output into file, overwrite if exists

cat file_1 file_2 file_3 > sum_file

- >> ← Output into file, append if exists

wc -l file >> number_lines

- 2> ← Output errors into file

perl program.pl > file.out 2> output.err

Piping:

- | ← Piping through programs

zcat file_1.zip | less

(allows to see content without de-compressing file)