Teacher: **Matteo Re**

UNIVERSITA' DEGLI STUDI DI MILANO

Metodi e linguaggi per il trattamento dei dati

PERL scripting

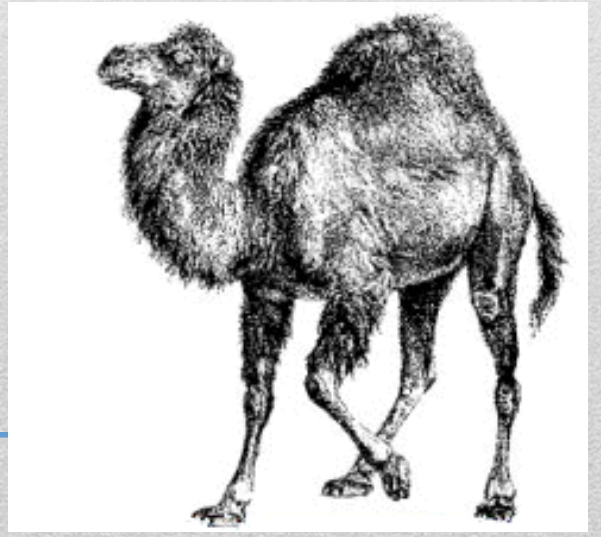
S

introduction

Objectives

- **PERL programming**
  - Problem solving and Debugging
  - To read and write documentation
  - Data manipulation: filtering and transformation
  - Pattern matching and data mining (examples)

- **Example application: Computational Biology**
  - Analysis and manipulation of biological sequences
  - Interaction with biological batabases (NCBI, EnsEMBL, UCSC)
  - BioPERL



Guidelines

- **Operating system**
  - During class appointments we will use windows

- **PERL installation**

  WIN: http://www.activestate.com/activeperl/downloads
  UNIX, MacOS: available by default

- **Text editor**

PERL are saved as text files. Many options available…

Vim (UNIX like OS)
Notepad (Windows)

Sequence file – FASTA format

```
>gi|40457238|HIV-1 isolate 97KE128 from Kenya gag gene, partial cds
CTTTTGAATGCATGGGTAAAAGTAATAGAAGAAAGAGGTTTCAGTCCAGAAGTAATACCCATGTTCTCAG
CATTATCAGAAGGAGCCACCCCACAAGATTTAAATACGATGCTGAACATAGTGGGGGGACACCAGGCAGC
TATGCAAATGCTAAAGGATACCATCAATGAGGAAGCTGCAGAATGGGACAGGTTACATCCAGTACATGCA
GGGCCTATTCCGCCAGGCCAGATGAGAGAACCAAGGGGAAGTGACATAGCAGGAACTACTAGTACCCCTC
AAGAACAAGTAGGATGGATGACAAACAATCCACCTATCCCAGTGGGAGACATCTATAAAAGATGGATCAT
CCTGGGCTTAAATAAAATAGTAAGAATGTATAGCCCTGTTAGCATTTTGGACATAAAACAAGGGCCAAAA
GAACCCTTTAGAGACTATGTAGATAGGTTCTTTAAAACTCTCAGAGCCGAACAAGCTT

>gi|40457236| HIV-1 isolate 97KE127 from Kenya gag gene, partial cds
TTGAATGCATGGGTGAAAGTAATAGAAGAAAAGGCTTTCAGCCCAGAAGTAATACCCATGTTCTCAGCAT
TATCAGAAGGAGCCACCCCACAAGATTTAAATATGATGCTGAATATAGTGGGGGGACACCAGGCAGCTAT
GCAAATGTTAAAAGATACCATCAATGAGGAAGCTGCAGAATGGGACAGGTTACATCCAATACATGCAGGG
CCTATTCCACCAGGCCAAATGAGAGAACCAAGGGGAAGTGACATAGCAGGAACTACTAGTACCCCTCAAG
AGCAAATAGGATGGATGACAAGCAACCCACCTATCCCAGTGGGAGACATCTATAAAAGATGGATAATCCT
GGGATTAAATAAAATAGTAAGAATGTATAGCCCTGTTAGCATTTTGGACATAAAACAAGGGCCAAAAGAA
CCTTTCAGAGACTATGTAGATAGGTTTTTTAAAACTCTCAGAGCCGAACAAGCTT

>gi|40457234| HIV-1 isolate 97KE126 from Kenya gag gene, partial cds
CCTTTGAATGCATGGGTGAAAGTAATAGAAGAAAAGGCTTTCAGCCCAGAAGTAATACCCATGTTTTCAG
CATTATCAGAAGGAGCCACCCCACAAGATTTAAATATGATGCTGAACATAGTGGGGGGGCACCAGGCAGC
TATGCAAATGTTAAAAGATACCATCAATGAGGAAGCTGCAGAATGGGACAGGCTACATCCAGCACAGGCA
GGGCCTATTGCACCAGGCCAGATAAGAGAACCAAGGGGAAGTGATATAGCAGGAACTACTAGTACCCCTC
AAGAACAAATAGCATGGATGACAGGCAACCCGCCTATCCCAGTGGGAGACATCTATAAAAGATGGATAAT
CCTGGGATTAAATAAAATAGTAAGAATGTATAGCCCTGTTAGCATTTTGGATATAAAACAAGGGCCAAAA
GAACCATTCAGAGACTATGTAGACAGGTTCTTTAAAACTCTCAGAGCCGAACAAGCTT
```

GenBank Record

```
LOCUS       AK091721                2234 bp    mRNA    linear   PRI 20-JAN-2006
DEFINITION  Homo sapiens cDNA FLJ34402 fis, clone HCHON2001505.
ACCESSION   AK091721
VERSION     AK091721.1  GI:21750158
KEYWORDS    oligo capping; fis (full insert sequence).
SOURCE      Homo sapiens (human)
  ORGANISM  Homo sapiens
            Eukaryota; Metazoa; Chordata; Craniata; Vertebrata; Euteleostomi;
            Mammalia; Eutheria; Euarchontoglires; Primates; Catarrhini;
            Hominidae; Homo.
  TITLE     Complete sequencing and characterization of 21,243 full-length
            human cDNAs
  JOURNAL   Nat. Genet. 36 (1), 40-45 (2004)
FEATURES             Location/Qualifiers
     source          1..2234
                     /organism="Homo sapiens"
                     /mol_type="mRNA"
     CDS             529..1995
                     /note="unnamed protein product"
                     /codon_start=1
                     /protein_id="BAC03731.1"
                     /db_xref="GI:21750159"
                     /translation="MVAERSPARSPGSWLFPGLWLLVLSGPGGLLRAQEQPSCRRAFD
               ...
                     RLDALWALLRRQYDRVSLMRPQEGDEGRCINFSRVPSQ"
ORIGIN
        1 gttttcggag tgcggaggga gttggggccg ccggaggaga agagtctcca ctcctagttt
       61 gttctgccgt cgccgcgtcc cagggacccc ttgtcccgaa gcgcacggca gcgggggggaa
   ...
```

Why Perl?

- **Broadly used in many application areas (i.e. computational Biology)**
  - Bioperl
    - http://www.bioperl.org/wiki/Main_Page
- **Relatively easy programming language**
  - Strong pattern matching capabilities
  - Easy construction of pipelines
  - Relatively easy to learn
- **Rapid prototyping**
  - Lot of problems can be solved with a few lines of code
- **Portability**
  - Available on *Unix, Windows, Macs*
- **Open source**
  - Good documentation (try: %perldoc perldoc)
    - %perldoc –f print
  - http://perldoc.perl.org/index-tutorials.html
  - Many modules (libraries) available (http://www.cpan.org/)

The PERL interpreter

PERL is an **interpreted** language.

The PERL interpreter:

1) translates each high level instruction (here high-level means that it is interpretable by a human being) into a instruction written in machine code (only composed by 0s and 1s) and that is specific for the considered machine architecture.

2) send the instruction to the CPU (that execute it)

This cycle continues until the source code (a text file containing the PERL instructions) contains at least one row of code.

It is possible to compile the script source. This produce a single file composed entirely by machine code instruction. The compiled file is called executable (the interpreter is no longer required) .

Teacher: **Matteo Re**

UNIVERSITA' DEGLI STUDI DI MILANO

Metodi e linguaggi per il trattamento dei dati

PERL scripting

p1

PERL basics (reference)

Originally developed by Larry Wall in 1987

Borrows features from

C: imperative language with variables, expressions, assignment statements, blocks of statements, control structures, and procedures / functions

Lisp: lists, list operations, functions as first-class citizens

AWK: (pattern scanning and processing language) hashes / associative arrays, regular expressions

sed: (stream editor for filtering and transforming text) regular expressions and substitution s///

Shell: **use of sigils to indicate type** ($ – scalar, @ – array, % – hash, & – procedure)

Object-oriented programming languages: classes/packages

PERL USES AD APPLICATIONS

- Main application areas of Perl
  - **text processing**
    → easier and more powerful than sed or awk
  - system administration
    → easier and more powerful than shell scripts

- Other application areas
  - web programming
  - code generation
  - **bioinformatics**
  - linguistics
  - testing and quality assurance

PERL: APPLICATIONS

Applications written in Perl

- Movable Type – web publishing platform
  http://www.movabletype.org/

- Request Tracker – issue tracking system
  http://bestpractical.com/rt/

- Slash – database-driven web application server
  http://sourceforge.net/projects/slashcode/

- EnsEMBL - genomic browser (originally written in PERL, now available in other programming languages)
  https://www.ensembl.org/index.html

PERL: APPLICATIONS

Organisations using Perl

- Amazon – online retailer
  http://www.amazon.co.uk
- BBC – TV/Radio/Online entertainment and journalism
  http://www.bbc.co.uk
- Booking.com – hotel bookings
  http://www.booking.com
- craigslist – classified ads
  http://www.craigslist.org
- IMDb – movie database
  http://www.imdb.com
- Monsanto – agriculture/biotech
  http://www.monsanto.co.uk/

JAVA vs PERL: JAVA

```
1  /* Author : java programmer
2  * The HelloWorld class implements an application
3  * that prints out " Hello World ".
4  */
5  public class HelloWorld {
6    // --------------- METHODS -------------
7    /* Main Method */
8      public static void main( String[] args ) {
9      System.out.println( " Hello World " );
10     }
11   }
```

Edit-compile-run cycle:

1) Edit and save as `HelloWorld.java`
2) Compile using `javac HelloWorld.java`
3) Run using `java HelloWorld`

JAVA vs PERL: PERL

PERL code on green background

```
1  #!/usr/bin/perl
2  # Author : Matteo Re
3  # The HelloWorld script implements an application
4  # that prints out " Hello World ".
5
6  print " Hello World \n" ;
```

Edit-compile-run cycle:

1) Edit and save as `HelloWorld`
2) Run using `perl HelloWorld`

OR

1) Edit and save as `HelloWorld`
2) Make it **executable** `chmod u+x HelloWorld`

This only needs to be done once!

3) Run using `./HelloWorld`

PERL

Perl borrows features from a wide range of programming languages including **imperative**, **object-oriented** and **functional** languages

- **Advantage:** Programmers have a choice of programming styles
- **Disadvantage:** Programmers have a choice of programming styles

- Perl makes it easy to write completely incomprehensible code
  → **Documenting and commenting** Perl code is very important

PERL

```
1  #!/usr/bin/perl
2  # Authors : Matteo Re
3  # Text manipulation using regular expressions
4  #
5  # Retrieve the Perl documentation of function ' atan2 '
6   @lines = `perldoc -u -f atan2`;
7  # Go through the lines of the documentation,turn all text
   # between angled brackets to uppercase and remove the
   # character in front of the opening angled bracket , then
8  # print the result
9   foreach( @lines ){
10    s/\w<([^\ >]+)>/\U$1/g ;
11    print ;
12  }
```

Perl makes it easy to write completely incomprehensible code.

In the example, there are more lines of comments than there are lines of code.

PERL

In the following we will consider various constructs of the Perl programming language

- numbers, strings
- variables, constants
- assignments
- control structures

If you are already able to program using other scripting languages (i.e. Java) you will notice some similarities. Remember that Perl predates Java and thus common constructs are almost always inherited by both languages from the C programming language

PERL SCRIPTS

- A Perl script consists of **one or more** statements and comments
  there is no need for a main function (or classes)

- Statements end in a semi-colon **;**

- Whitespace before and in between statements is irrelevant
(This does not mean its irrelevant to someone reading your code)

- Comments start with a hash symbol **#** and run to the end of the line

- Comments **should precede the code** they are referring to

PERL SCRIPTS

Perl statements include

- Assignments
- Control structures

NB: Every statement returns a value

Perl data types include

- Scalars
- Arrays / Lists
- Hashes / Associative arrays

Perl **expressions** are constructed from values and variables using operators and subroutines.
Perl expressions can have side-effects (evaluation of an expression can change the program state).
Every expression can be turned into a statement by adding a semi-colon.

A **scalar** is the simplest type of data in Perl

- **A scalar is either**
  - **an integer number**
    0 2012 -40 1_263_978
  - **a floating-point number**
    1.25 256.0 -12e19 2.4e-10
  - **a string**
    'hello world' "hello world\n"

Note:

- **There is no 'integer type', 'string type' etc**
- **There are no boolean constants (true / false)**

Data types: scalar $

Integers and floating point numbers

Perl provides a wide range of pre-defined mathematical functions

abs(number) absolute value
log(number) natural logarithm
random(number) random number between 0 and number
sqrt(number) square root

Additional functions are available via the POSIX module

ceil(number) round fractions up
floor(number) round fractions down

Note: There is **no** pre-defined **round** function

PSEUDOCODE on yellow background

```
use POSIX;
print ceil (4.3); // prints '5 '
print floor (4.3); // prints '4 '
```

Data type: scalar $

Error handling

Mathematical functions and Error handling

Perl, PHP and JavaScript differ in the way they deal with applications of mathematical functions that do not produce a number

In Perl we have

- log(0) produces an error message: Can't take log of 0
- sqrt(-1) produces an error message: Can't take sqrt of -1
- 1/0 produces an error message: Illegal division by zero
- 0/0 produces an error message: Illegal division by zero

the **execution of a script terminates** when an error occurs

Error handling

A possible way to perform error handling in Perl is as follows:

```
eval {
  ... run the code here .... # try
     1;
} or do {
  ...handle the error here using $@... # catch
};
```

The **special variable $@** contains the Perl syntax or routine error message from the last eval, do-FILE, or require command

Double quoted strings backslash escapes

- In a single-quoted string \t is simply a string consisting of \ and t
- In a double-quoted string \t and other backslash escapes have the following meaning

\n	Logical Newline (actual character is platform dependent)
\f	Formfeed
\r	Return
\t	Tab
\l	Lower case next letter
\L	Lower case all following letters until \E
\u	Upper case next letter
\U	Upper case all following letters until \E
\Q	Quote non-word characters by adding a backslash until \E
\E	End \L, \U, \Q

UTF-8

Perl supports UTF-8 character encodings which give you access to non-ASCII characters

- **The pragma**

```
use utf8;
```

allows you to use UTF-8 encoded characters in Perl scripts

- **The function call**

```
binmode ( STDIN , " : encoding ( UTF -8) " );
binmode ( STDOUT , " : encoding ( UTF -8) " );
```

ensures that UFT-8 characters are read correctly from STDIN and printed correctly to STDOUT

- **The Unicode::Normalize module enables correct decomposition of strings containing UTF-8 encoded characters**

```
use Unicode :: Normalize ;
```

UTF-8

Example…

```
binmode( STDOUT , " : utf8 " );
print "\x{4f60}\x{597d}\x{4e16}\x{754c}\n";    # chinese
print "\x{062d}\x{fef0}\n";                    # arabic
```

String operators and automatic conversion:

Two basic operation on strings are:

- **String concatenation**

```
"hello" . "world"          ⇝  "helloworld"
"hello" . '␣' . "world"    ⇝  'hello␣world'
"\Uhello" . '␣\LWORLD'     ⇝  'HELLO␣\LWORLD'
```

- **String repetition**

```
"hello␣" x 3   ⇝  "hello␣hello␣hello␣"
```

These operation can be **combined**:

```
"hello␣" . "world␣" x 2   ⇝  "hello␣world␣world␣"
```

Perl **automatically converts** between strings and numbers:

```
2 . "␣worlds"   ⇝  "2␣worlds"
"2" * 3         ⇝  6
2e-1 x 3        ⇝  "0.20.20.2" ("0.2" repeated three times)
"hello"* 3      ⇝  0
```

‘Booleans’

- Perl does **not** have a boolean datatype
- Instead the values

```
0      # zero and all floating-point numbers equal to 0
''     # empty string
'0 '   # string consisting of zero , but not '0.0'
undef  # undefined
()     # empty list
```

all represent **false** with all other values represent **true** .

‘Boolean operators’

- Perl offers the same short-circuit boolean operators as Java: **&&**, **||**, **!**
- Alternatively, **and**, **or**, **not** can be used

A	B	(A && B)
true	*true*	B (*true*)
true	*false*	B (*false*)
false	*true*	A (*false*)
false	*false*	A (*false*)

A	B	(A || B)
true	*true*	A (*true*)
true	*false*	A (*true*)
false	*true*	B (*true*)
false	*false*	B (*false*)

A	(! A)
true	'' (*false*)
false	1 (*true*)

Note that this means that && and || **are not commutative**, that is, `(A && B)` **is not** the same as `(B && A)`

```
( $denom != 0 ) && ( $num / $denom > 10 )
```

Comparison operators

- Perl distinguishes between **numeric comparison** and **string comparison**

Comparison	Numeric	String
Equal	==	eq
Not equal	!=	ne
Less than	<	lt
Greater than	>	gt
Less than or equal to	<=	le
Greater than or equal to	>=	ge

Examples

```
  35        ==       35.0     # true
' 35 '      eq     ' 35.0 '   # false
' 35 '      ==     ' 35.0 '   # true
  35        <        35.0     # false
' 35 '      lt     ' 35.0 '   # true
' ABC '     eq     "\Uabc"    # true
```

Scalar variables

A variable also **does not have** to be initialised before it can be used, although initialisation **is a good idea**

- Uninitialised variables have the special value **undef**

However, undef acts

```
like 0 for numeric variables and
like '' for string variables
```

if an uninitialised variable is used in an arithmetic or string operation

- To test whether a variable has value undef use the routine **defined**

```
$s1 = "";
print '$s1 eq undef : ' ,($s1 eq undef)?' TRUE':' FALSE' ," \n";
print '$s1 defined : ' ,(defined($s1))?' TRUE ':' FALSE',"\n" ;
print '$s2 defined : ' ,(defined($s2))?' TRUE ':' FALSE',"\ n";
```

```
$s1 eq undef : TRUE
$s1 defined : TRUE
$s2 defined : FALSE
```

OUTPUT on light blue background

special variables

Perl has a lot of ‘pre-defined’ variables that have a particular meaning and serve a particular purpose

Variable	Meaning
`$_`	The default or implicit variable
`@_`	Subroutine parameters
`$a, $b`	sort comparison routine variables
`$&`	the string matched by the last successful pattern match
`$/`	input record separator, newline by default
`$\`	output record separator, `undef` by default
`$]`	version of Perl used

For a full list see:
`https://perldoc.perl.org/perlvar.html#SPECIAL-VARIABLES`

Constants

Perl offers three different ways to declare constants

- Using the **constant** pragma:

```
use constant PI = > 3.14159265359;
```

(A **pragma** is a module which influences some aspect of the compile time or run time behaviour of Perl)

- Using the **Readonly module**:

```
use Readonly ;
Readonly $PI = > 3.14159265359;
```

- Using the **Const::Fast module**:

```
use Const :: Fast ;
const $PI = > 3.14159265359;
```

variable interpolation with constants does not work

- Just like Java, Perl uses the **equality sign =** for assignments:

```
$student_id = 200846369;
$name = " Mario Rossi " ;
$student_id = " E00481370 " ;
```

But no type declaration is required and the same variable can hold a number at one point and a string at another

- An assignment also returns a value, namely (the **final** value of) the variable **on the left** → enables us to use an assignment as an expressions

Example:

```
$b = ( $a = 0) + 1;
# $a has value 0
# $b has value 1
```

Constants

Variable declaration

In Perl, variables can be declared using the **my** function (Remember: This is not a requirement)

- The pragma

```
use strict;
```

enforces that all variables **must be declared before their use**, otherwise a compile time error is raised

Example:

```
use strict;
$studentsOnPERLmodule = 133;
```

```
Global symbol " $studentOnPERLmodule " requires explicit
  package name at ./ script line 2.
Execution of ./ script aborted due to compilation errors.
```

```
use strict;
my $studentsOnPERLmodule;
$studentsOnPERLmodule = 154;
my $studentsOnPERLmodule2 = 53;
```

Variable interpolation

Any scalar variable name **in a double quoted string** is (automatically) replaced by its current value at the time the string is 'created'

Example:

```
$actor = " Jeff Bridges " ;
$prize = " Academy Award for Best Actor " ;
$year  = 2010;

print " 1: " , $actor , " won the ",$prize," in ",$year,"\n";
print " 2: $actor won the $prize in $year\n";
```

Output:

```
1: Jeff Bridges won the Academy Award for Best Actor in 2010
2: Jeff Bridges won the Academy Award for Best Actor in 2010
```

- Conditional statements
- Switch statements
- While and Until loops
- For loops

Control structures

Conditional statements

The general format of conditional statements is very similar to that in Java and other scriptng langiages:

```
if( condition ){
                    statements
}elsif( condition ){
                    statements
}else{
                    statements
}
```

- **condition** is an arbitrary expression
- the **elsif-clause** is optional and there can be more than one
- the **else-clause** is optional but there can be at most one
- in contrast to Java, the curly brackets must be present

even if **statements** consist only of a single statement

Control structures

Conditional statements

Perl also offers two shorter conditional statements:

```
statement if( condition );
```

and

```
statement unless( condition );
```

- Perl also offers **conditional expressions**:

```
condition ? if_true_expr : if_false_expr
```

Examples:

```
$descr = ( $distance < 50) ? " near " : " far " ;

$size = ( $width < 10) ? " small " :
        ( $width < 20) ? " medium " :
                         " large " ;
```

Control structures

Blocks

A sequence of statements in curly brackets is a block --> an alternative definition of conditional statements is

```
if( condition ) block
elsif( condition ) block
else block
```

In

```
statement if ( condition );
statement unless ( condition );
```

only a single statement is allowed,
but **do block** counts as a single statement, so we can write

```
do block if ( condition );
do block unless ( condition );
```

Control structures

Switch statement / expression

Starting with Perl 5.10 (released Dec 2007), the language includes a switch statement and corresponding switch expression. But these are considered experimental and need to be enabled explicitly.

Example:

```
use feature "switch";

given ( $month ){
when ([1 ,3 ,5 ,7 ,8 ,10 ,12]){ $days=31 }
when ([4 ,6 ,9 ,11]){ $days=30 }
when (2){ $days=28 }
default{ $days=0 }
}
```

Note: no explicit break statement is needed

Control structures

while and until loops

Perl offers while-loops and until-loops

```
while ( condition ) {
                statements
}
```

```
until ( condition ) {
                statements
}
```

A ‘proper’ until-loop where the loop is executed at least once can be obtained as follows

```
do{ statements }until( condition );
```

The same construct also works for **if**, **unless** and **while**
In case there is only a single statement it is also possible to write

```
statement until ( condition );
```

Again this also works for **if**, **unless** and **while**

Control structures

Control structures

For loops

for-loops in Perl take the form

```
for( initialisation ; test ; increment ){
      statements
}
```

Again, the curly brackets are required even if the body of the loop only consists of a single statement

Such a **for**-loop is **equivalent** to the following **while**-loop:

```
initialisation ;

while ( test ) {
      statements;
      increment;
}
```

A list is an **ordered collection of scalars**

An **array** (array variable) is a variable that **contains a list**

Array variables **start with @** followed by a Perl identifier

```
@identifier
```

An array variable denotes the entire list stored in that variable

Perl uses

```
$identifier[ index ]
```

to denote the element stored at position index in @identifier
The first array element **has index 0**

- Note that

```
$identifier
@identifier
```

are two unrelated variables (but this situation should be avoided)

Data type: lists , arrays

A list can be specified by a **list literal**, a **comma-separated list of values** enclosed by parentheses

```
(1 , 2 , 3)
( " adam " , " ben " , " colin " , " david " )
( " adam " , 1 , " ben " , 3)
( )
(1..10 , 15 , 20..30)
( $start .. $end )
```

List literals can be assigned to an array:

```
@numbers = (1..10 , 15 , 20..30);
@names = ( " adam " , " ben " , " colin " , " david " );
```

Examples of more complex assignments, involving array concatenation:

```
@numbers = (1..10 , undef , @numbers , ( ));
@names = ( @names , @numbers );
```

Note that arrays do not have a pre-defined size/length

Data type: lists , arrays

Size of an array:

There are three different ways to determine the size of an array

```
$arraySize = scalar( @array );
$arraySize = @array ;
$arraySize = $#array + 1;
```

One can access all elements of an array using indices in the range **0** to **$#array**

But Perl also allows **negative** array indices:
The expression **$**array[**-index**]
is equivalent to **$**array[**scalar**(**@array**)**-index**]

Example:

```
$array[-1]      is the same as    $array[scalar(@array)-1]
```
is the same as `$array[$#array]`
that is the **last element** in `@array`

Data type: lists , arrays

Perl allows you to access array indices that are **out** of bounds

```
@array = (0 , undef , 22 , 33);

print '$array[4] = ',$array[4],' ,which ',(defined($array[4]) ? ' IS NOT ' : ' IS
  ',"undef\n";
```

```
$array[4] = , which IS undef
```

```
print '$array[1] = ',$array[1],' ,which ',(defined($array[1]) ? ' IS NOT ' : ' IS
  ',"undef\n";
```

```
$array[1] = , which IS undef
```

The function **exists** can be used to determine whether an array index is within bounds and has a value (including undef) associated with it

```
print ' $array[4] exists : ' , exists($array[4]) ? 'T ': 'F ' ,"\n" ;
```

```
$array[4] exists : F
```

```
print ' $array[1] exists : ' , exists($array[1]) ? 'T ': 'F ' ,"\n" ;
```

```
$array[1] exists : T
```

Data type: lists , arrays

Scalar context vs List context:

when an expression is used as an argument of an operation that requires a scalar value, the expression will be evaluated in a **scalar context**

Example of scalar context:

```
$arraySize = @array;
```

@array stores a list , but returns the number of elements of @array in a scalar context.

when an expression is used as an argument of an operation that requires a list value, the expression will be evaluated in a list context

Example of list context:

```
@sorted = sort 5;
```

A single scalar value is treated as a list with one element in a list context

Data type: lists , arrays

Scalar context vs List context:

Expressions behave differently in different contexts following these rules:

- Some operators and functions automatically return different values in different contexts

```
$line = <IN>;       # return one line from IN
@lines = <IN>;      # return a list of all lines from IN
```

- If an expression returns a scalar value **in a list context**, then by default Perl will convert it into a list value with the returned scalar value being the one and only element

- If an expression returns a list value **in a scalar context**, then by default Perl will convert it into a scalar value by take **the last element** of the returned list

Data type: lists , arrays

Array functions: push, pop, shift, unshift

Perl has no stack or queue data structures, but has stack and queue **functions** for arrays:

Function	Semantics
`push(@array1,value)` `push(@array1,list)`	appends an element or an entire list to the end of an array variable; returns the number of elements in the resulting array
`pop(@array1)`	extracts the last element from an array and returns it
`shift(@array1)`	shift extracts the first element of an array and returns it
`unshift(@array1,value)` `unshift(@array1,list)`	insert an element or an entire list at the start of an array variable; returns the number of elements in the resulting array

Data type: lists , arrays

Array operators: push, pop, shift, unshift

```
@planets = ( "earth" );
push( @planets , "mars" ,"jupiter" ,"saturn" );
unshift( @planets , "mercury" ,"venus" );
print "Array\@1 : " , join ( " " , @planets ) , "\n";
```

```
Array@1 : mercury venus earth mars jupiter saturn
```

Note: unshift **does not** proceed argument by argument

```
$last = pop( @planets );
print "Array\@2 : " , join ( " " , @planets ) , "\n";
```

```
Array@2 : mercury venus earth mars jupiter
```

```
$first = shift( @planets );
print "Array\@3 : " , join ( " " , @planets ) , "\n";
print "     @4 : " , $first , " " , $last , "\n";
```

```
Array@3 : venus earth mars jupiter
     @4 : mercury saturn
```

Data type: lists , arrays

Array operators: delete

```
delete($array[index])
```

– removes the value stored at index in @array **and returns it**
– only if index equals $#array will the array's size shrink to the position of the highest element that returns true for exists()

Example

```
@array = (0 , 11 , 22 , 33);
delete($array[2]);
print '$array[2] exists : ', exists($array[2])? "T" : "F","\n";
print 'Size of $array : ' ,$#array+1, " \ n " ;
```

```
$array[2] exists : F
Size of $array : 4
```

```
delete($array[3]);
print '$array[3] exists : ',exists($array[3])? "T" : "F" ,
  "\n";
print 'Size of $array : ' ,$#array+1, "\n" ;
```

```
$array[3] exists : F
Size of $array : 2
```

Data type: lists , arrays

Foreach loop

Changing the value of the foreach-variable changes the element of the list that it currently stores

A foreach-variable reverts to its previous value after the end of a loop

Example:

```
@my_list = (1..5 ,20 ,11..18);
print "Before : " . join("," , @my_list ). "\n";
```

```
Before : 1,2,3,4,5,20,11,12,13,14,15,16,17,18
```

```
foreach $number ( @my_list ){
$number++;
}
print "After : ". join("," , @my_list). "\n";
```

```
After : 2,3,4,5,6,21,12,13,14,15,16,17,18,19
```

```
print '$number = ' ,defined($number)?$number:"undef " ,"\n";
```

```
$number = undef
```

If no variable is specified, then the special variable **$_** will be used to store the array elements

Control structures

Foreach loop

An alternative way to traverse an array is

```
foreach $index (0..$#array ){
  statements
}
```

where an element of the array is then accessed using $array[$index] in statements

Example:

```
@my_list = (1..5 ,20 ,11..18);
foreach $index (0..$# my_list ) {
  $max = $my_list[$index] if( $my_list[$index]>$max);
}
print("Maximum number in ",join(',',@my_list)," is $max\n");
```

Control structures

Foreach loop variants

In analogy to while- and until-loops, there are the following variants of foreach-loops:

```
do{ statements } foreach list;
statement foreach list;
```

In the execution of the statements within the loop, the special variable **$_** will be set to **consecutive elements of list**

Instead of foreach we can also use for:

```
do{ statements } for list;
statement for list;
```

Example:

```
# Instead of
foreach( @my_list ){ $_++ }
# we can write
$_++ foreach(@my_list);
```

Control structures

last and next

The **last** command can be used in while-, until-, and foreach-loops and discontinues the execution of a loop

```
while( $value = shift($data)){
  $written = print(FILE $value);
  if(!$written){ last; }
}
# Execution of last takes us here
```

• The **next** command stops the execution of the current iteration of a loop and moves the execution to the next iteration

```
foreach $x (-2..2){
  if($x == 0){ next; }
  printf(" 10/%2d = %3d\n",$x,(10/$x));
}
```

```
10 / -2 = -5
10 / -1 = -10
10 / 1 = 10
10 / 2 = 5
```

Control structures

Data type: hashes

- A hash is a data structure similar to an array but it **associates scalars with a string instead of a number**
- Alternatively, a hash can be seen as a partial function mapping strings to scalars
- Remember that Perl can auto-magically convert any scalar into a string
- Hash variables start with a percent sign followed by a Perl identifier

```
%identifier
```

A hash variable denotes the entirety of the hash

- Perl uses

```
$identifier{ key }
```

where key is a string, to refer to the value associated with key

Data type: hashes

Note that

```
$identifier
%identifier
```

are two unrelated variables (but this situation should be avoided)

An easy way to print all key-value pairs of a hash %hash is the following

```
use Data::Dumper;
$Data::Dumper::Terse = 1;
print Dumper \%hash;
```

Note the use of \%hash instead of %hash
(\%hash is a reference to %hash)

`Data::Dumper` can produce string representations for arbitrary Perl data structureswith key

Basic hash operations

- Initialise a hash using a list of key-value pairs

```
%hash = (key1, value1, key2, value2, ...);
```

- Initialise a hash using a list in big arrow notation

```
%hash = ( key1=>value1 ,key2=>value2, ...);
```

- Associate a single value with a key

```
$hash{ key } = value;
```

- Remember that undef is a **scalar** value

```
$hash{ key } = undef;
```

extends a hash with another key but unknown value

Data type: hashes

Basic hash operations

It is also possible to assign one hash to another

```
%hash1 = %hash2;
```

In contrast to **C** or **Java** this operation creates a copy of %hash2 that is then assigned to %hash1

Example:

```
%hash1 = ( 'a ' = > 1 , 'b ' = > 2);
%hash2 = % hash1;
$hash1 { 'b '} = 4;
print "\$hash1{ ' b '} = $hash1{ ' b '}\n";
print "\$hash2{ ' b '} = $hash2{ ' b '}\n";
```

```
$hash1 { 'b '} = 4  # value has changed
$hash2 { 'b '} = 2  # value has not changed
```

Data type: hashes

The each, keys and values functions

each %hash	returns a 2-element list consisting of the key and value for the next element of %hash, so that one can iterate over it
values %hash	returns a list consisting of all the values of %hash, resets the internal iterator for %hash
keys %hash	returns a list consisting of all keys of %hash, resets the internal iterator for %hash

Examples:

```
while (($key,$value) = each %hash){
  statements
}
```

```
foreach $key ( sort keys %hash ){
  $\value = $hash{ $key };
}
```

Data type: hashes

Example: Two-dimensional hash as a ‘database’

```
use List :: Util " sum " ;
$name { ' 200846369 '} = ' Jan Olsen ';
$marks { ' 200846369 ' }{ ' INGM_master_class_1 '} = 61;
$marks { ' 200846369 ' }{ ' INGM_master_class_2 '} = 57;
$marks { ' 200846369 ' }{ ' INGM_master_class_3 '} = 43;
$marks { ' 200846369 ' }{ ' INGM_master_class_4 '} = 79;
$average = sum(values($marks{'200846369
  '}))/scalar(values($marks{' 200846369 '});
print( " avg : $average\n" );
```

Output:

```
avg : 60
```

Data type: hashes

Example: Frequency of words

```
# Establish the frequency of words in a string
$string = " peter paul mary paul jim mary paul " ;
# Split the string into words and use a hash
# to accumulate the word count for each word
++$count{ $_ } foreach split (/\s+/ , $string);
# Print the frequency of each word found in the
# string
while ( ($key,$value ) = each %count){
print ( "$key = > $value ;");
}
```

Output:

```
jim = > 1; peter = > 1; mary = > 2; paul = > 3
```

Data type: hashes

Teacher: **Matteo Re**

UNIVERSITA' DEGLI STUDI DI MILANO

Metodi e linguaggi per il trattamento dei dati

PERL scripting

p2

The PERL debugger

1 Debugger basics

Starting the debugger

Usually you will start the debugger with a script/application you want to explore in greater detail.

The easiest way to do this is to use the -d switch with perl:

```
perl -d ./simplest.pl
```

If you run that you will see something like:

```
$perl -d simplest.pl

Loading DB routines from perl5db.pl version 1.39
Editor support available.

Enter h or 'h h' for help, or 'man perldebug' for more help.

main::(simplest.pl:2):  my $nothing = 0;
  DB
```

At the prompt simply type q and press ENTER to quit the debugger.

1 Debugger basics

Stepping through a script

Use the debugger with step.pl:

```
perl -d ./step.pl
```

Each time you see the debugger prompt, type **n** and press **ENTER**.

When you see:

```
Debugged program terminated.  Use q to quit or R to restart
```

press q then ENTER to quit the debugger.

1 Debugger basics

Example output

```
$ perl -d ./step.pl

Loading DB routines from perl5db.pl version 1.39
Editor support available.

Enter h or 'h h' for help, or 'man perldebug' for more help.

main::(step.pl:2):  my $nothing = 0;
  DB n
main::(step.pl:3):  my $something = 1;
  DB n
main::(step.pl:4):  $nothing++;
  DB n
main::(step.pl:5):  if ($nothing == $something) {
  DB n
main::(step.pl:6):      print "How can nothing be something?\n";
  DB n
How can nothing be something?
Debugged program terminated.  Use q to quit or R to restart,
```

1 Debugger basics

Stepping through a script with fewer key-presses

Use the debugger with step.pl:

```
perl -d ./step.pl
```

The **FIRST** time you see the prompt, **type n and press ENTER**.
All other times **only press ENTER**.

When you see:

```
Debugged program terminated.  Use q to quit or R to restart,
```

press q then ENTER to quit the debugger.

1 Debugger basics

Restarting your script

Use the debugger with step.pl:

```
perl -d ./step.pl
```

Step one or two lines through the script **using n**.

Restart the script by typing **R** and pressing **ENTER**.

Step through and restart a few times more.

Once you are happy stepping through and restarting the script press q then ENTER to quit the debugger.

1 Debugger basics

Summary

perl -d **./**scriptfilename	start debugging
n	step through the script (‘next’)
q	quit the debugger (‘quit’)
R	start the script from the beginning (‘restart’)

2 Examining variables

Examining variables

Start by running the relevant script in the debugger:

```
perl -d ./variables.pl
```

Step through the script (with n) until you see:

```
string is Mary had a little lamb.
main::(variables.pl:8): print "stop pressing 'n' or 'ENTER' now\n";
  DB
```

2 Examining variables

Examining variables

You can now examine the variables that have been **declared** in the script. Try these commands in the debugger:

```
p $string
x $string
p @things
x @things
p %hashof
x %hashof
```

What is the difference between the p and the x command (use the h command to answer this question)

2 Examining variables

Examining variables

You may have noticed that **x %hashof** isn't very easy to parse. Once you realise it's a hash you can examine the reference for a nicer output:

```
x  \%hashof
```

Slightly less useful in this simple example is:

```
x  \@things
```

In your day-to-day debugging you will usually use **x** for examining the value of variables.

2 Examining variables

Examining variables

The perl debugger doesn't run in **'strict mode'** so you can examine variables that have not been defined.

```
  DB x $my_made_up_thing
0  undef
```

2 Examining variables

Summary

X	examine a variable
p	print the value of a variable

Teacher: **Matteo Re**

UNIVERSITA' DEGLI STUDI DI MILANO

Metodi e linguaggi per il trattamento dei dati

PERL scripting

p3

Regular expressions (1)

Regular expressions

Introduction

Characters

Character classes

Quantifiers

Regular expressions: Motivation

Suppose you are testing the performance of a new sorting algorithm by measuring its runtime on randomly generated arrays of numbers of a given length:

```
Generating an unsorted array with 10000 elements took 1.250 seconds
Sorting took 7.220 seconds
Generating an unsorted array with 10000 elements took 1.243 seconds
Sorting took 10.486 seconds
Generating an unsorted array with 10000 elements took 1.216 seconds
Sorting took 8.951 seconds
```

Your task is to write a program that determines the average runtime of the sorting algorithm:

```
Average runtime for 10000 elements is 8.886 seconds
```

Solution: The regular expression **/^Sorting took (\d+\.\d+) seconds/**

allows us to get the required information

Regular expressions are useful for **information extraction**

Regular expressions: Motivation

Suppose you have recently taken over responsibility for a company's website. You note that their HTML files contain a large number of URLs containing superfluous occurrences of '..', e.g.

http://www.myorg.co.uk/info/refund/../vat.html

Your task is to write a program that replaces URLs like these with equivalent ones without occurrences of '..':

http://www.myorg.co.uk/info/vat.html

while making sure that relative URLs like

../video/disk.html

are preserved

Solution: **s!/[^\/]+/\.\.!!;** removes a superfluous dot-segment

Substitution of regular expressions is useful for **text manipulation**

Regular expressions: Motivation

```
\Ahttps?:\/\/[^\/]+\/.\w.\/(cat|dog)\/\1
```

- \A is an assertion or anchor
- h, t, p, s, :, \/, c, a, t, d, o, g are characters
- ? and + are quantifiers
- [^\/] is a character class
- . is a metacharacter and \w is a special escape
- (cat|dog) is alternation within a capture group
- \1 is a backreference to a capture group

Pattern match operation

To match a regular expession regexpr against the **special variable $_** simply use one of the expressions **/regexpr/** or **m/regexpr/**

- This is called a **pattern match**
- $_ is the **target string** of the pattern match

In a scalar context a pattern match returns true (1) or false (") depending on whether regexpr matches the target string

```
if (/\Ahttps?:\/\/[^\/]+\/.\w.\/(cat|dog)\/\1/) {
... }
if ( m/\Ahttps?:\/\/[^\/]+\/.\w.\/(cat|dog)\/\1/) {
... }
```

Regular expressions: characters

The simplest regular expression just consists of a sequence of
- **alphanumeric** characters and
- **non-alphanumeric** characters escaped by a backslash:

that matches exactly this sequence of characters occurring as a substring in the target string

```
$_="ababcbcdcde" ;
if (/cbc/){print "Match\n"}else{print "No match\n"}
```

Output:

```
Match
```

```
$_="ababcbcdcde" ;
if (/dbd/){print "Match\n"}else{print "No match\n"}
```

Output:

```
No match
```

Regular expressions: special escapes

There are various special escapes and metacharacters that match more than one character:

`.`	Matches any character except `\n`
`\w`	Matches a 'word' character (alphanumeric plus '_', plus other connector punctuation characters plus Unicode characters
`\W`	Matches a non-'word' character
`\s`	Match a whitespace character
`\S`	Match a non-whitespace character
`\d`	Match a decimal digit character
`\D`	Match a non-digit character
`\p{`*UnicodeProperty*`}`	Match *UnicodeProperty* characters
`\P{`*UnicodeProperty*`}`	Match non-*UnicodeProperty* characters

Regular expressions: Character class

A character class, a list of characters, special escapes, metacharacters and unicode properties **enclosed in square brackets**, matches **any single character from within the class**, for example, **[ad\t\n\-\\09]**

• One may specify a range of characters with a **hyphen -**, for example, **[b-u]**

• A **caret ^** at the **start of a character class** negates/complements it, that is, it matches any single character that **is not** from within the class, for example, [^01a-z]

```
$_ = "ababcbcdcde" ;
if(/[bc][b-e][^bcd]/){
print "Match at positions $-[0] to ", $+[0]-1 ,": $&\n"};
```

Output:

```
Match at positions 8 to 10: cde
```

Regular expressions: quantifiers

- The constructs for regular expressions that we have so far are not sufficient to match, for example, natural numbers of arbitrary size
- Also, writing a regular expressions for, say, a nine digit number would be tedious. This is made possible with the use of **quantifiers**

*regexpr**	Match *regexpr* 0 or more times
regexpr+	Match *regexpr* 1 or more times
regexpr?	Match *regexpr* 1 or 0 times
regexpr{*n*}	Match *regexpr* exactly n times
regexpr{*n*,}	Match *regexpr* at least n times
regexpr{*n*,*m*}	Match *regexpr* at least n but not more than m times

Quantifiers are greedy by default and match the longest leftmost sequence of characters possible

Regular expressions: quantifiers

*regexpr**	Match *regexpr* 0 or more times
regexpr+	Match *regexpr* 1 or more times
regexpr?	Match *regexpr* 1 or 0 times
regexpr{*n*}	Match *regexpr* exactly n times
regexpr{*n*,}	Match *regexpr* at least n times
regexpr{*n*,*m*}	Match *regexpr* at least n but not more than m times

Example:

```
$_ = "Sorting took 10.486 seconds" ;
if (/\d+\.\d+/){
print "Match at positions $-[0] to ",$+[0]-1,": $&\n"};
```

```
Match at positions 13 to 18: 10.486
```

```
$_ = "A sample staff id is E00481370" ;
if (/[A-Z]0{2}\d{6}/){
print "Match at positions $-[0] to ",$+[0]-1, ": $&\n"};
```

```
Match at positions 21 to 29: E00481370
```

Regular expressions: quantifiers

Example:

```
$_ = "E00481370";
if (/\d+/) {
print "Match at positions $-[0] to ",$+[0]-1, " : $&\n"};
```

Output:

```
Match at positions 1 to 8: 00481370
```

- The regular expression **\d+** matches **1 or more digits**
- As the example illustrates, the regular expression **\d+**
- matches as early as possible
- matches as many digits as possible → quantifiers are **greedy** by default

Teacher: **Matteo Re**

UNIVERSITA' DEGLI STUDI DI MILANO

Metodi e linguaggi per il trattamento dei dati

PERL scripting

p4 Regular expressions (2)

Regular expressions: capture groups and backreferences

- We often encounter situations where we want to identify the repetition of the same or similar text, for example, in HTML markup:

```
<strong> ... </strong>
<li> ... </li>
```

- We might also not just be interested in the repeating text itself, but the text between or outside the repetition

- We can characterise each individual example above using regular expressions:

```
<strong>.*<\/strong>
<li>.* <\/li>
```

but we cannot characterise both without losing fidelity, for example:

```
<\w+>.* <\/\w+>
```

does not capture the 'pairing' of HTML tags

Regular expressions: capture groups

The solution are capture groups and backreferences

(*regexpr*)	creates a capture group
(?<*name*>*regexpr*)	creates a named capture group
(?:*regexpr*)	creates a non-capturing group
N, \g*N*, \g{*N*}	backreference to capture group *N* (where *N* is a natural number)
\g{*name*}	backreference to a named capture group

```
/Sorting took (\d+\.\d+) seconds/
/<(\w+)>.*<\/\1>/
/([A-Z])0{2}(\d+)/
/(?<c1>\w)(?<c2>\w)\g{c2}\g{c1}/
/((?<c1>\w)(?<c2>\w)\g{c2}\g{c1})/
```

Regular expressions: capture groups

Via capture variables the strings matched by a capture group are also available outside the pattern in which they are contained

$*N*	string matched by capture group *N* (where *N* is a natural number)
$+{*name*}	string matched by a named capture group

The matched strings are available until the end of the enclosing code block or until the next successful match

```
$_ = "Yabba dabba doo" ;
if ((?<c1>\w)(?<c2>\w)\g{c2}\g{c1})){
print " Match found : $1 | $2 | $+{c1}\n"
}
```

```
Match found : abba | a | a
```

Regular expressions: alternations

- The regular expression **regexpr1 | regexpr2** matches if either regexpr1 **OR** regexpr2 matches. This type of regular expression is called an **alternation**

- Within a larger regular expression we need to enclose alternations in a capture group or non-capturing group: **(regexpr1|regexpr2)** or **(?:regexpr1|regexpr2)**

Examples:

```
1 /Mr|Ms|Mrs|Dr/
2 /cat|dog|bird/
3 /(?:Bill|Hillary) Clinton /
```

Regular expressions: alternations

- The order of expressions in an alternation only matters if one expression matches a sub-expression of another

Example:

```
$_ = " cats and dogs " ;
if (/(cat|dog|bird )/) { print " Match 1: $1\n" }
```

```
Match 1: cat
```

```
$_ = " cats and dogs " ;
if (/( dog | cat | bird )/) { print " Match 2: $1 \ n " }
```

```
Match 2: cat
```

```
if (/( dog | dogs )/) { print " Match 3: $1 \ n " }
```

```
Match 3: dog
```

→ Matching is greedy with respect to quantifiers not wrt alternations!

```
if (/( dogs | dog )/) { print " Match 4: $1 \ n " }
```

```
Match 4: dogs
```

Regular expressions: anchors

- Anchors allow us to fix where a match has to start or end

\A	Match only at string start
^	Match only at string start (default) Match only at a line start (in //m)
\Z	Match only at string end modulo a preceding \n
\z	Match only at string end
$	Match only at string end modulo a preceding \n Match only at a line end (in //m)
\b	Match word boundary (between \w and \W)
\B	Match except at word boundary

Example:

```
$_ = " The girl who \nplayed with fire\n";
if(/fire\z/){print "`fire' at string end\n"}
if(/fire\Z/){print "`fire' at string end modulo \\n \n"}
```

```
`fire' at string end modulo \n
```

Regular expressions: Modifiers

- Modifiers change the interpretation of certain characters in a regular expression or the way in which Perl finds a match for a regular expression

/ /	Default '.' matches any character except '\n' '^' matches only at string start '$' matches only at string end modulo preceding \n
/ /s	Treat string as a single long line '.' matches any character including '\n' '^' matches only at string start '$' matches only at string end modulo preceding \n
/ /m	Treat string as a set of multiple lines '.' matches any character except '\n' '^' matches at a line start '$' matches at a line end

Regular expressions: Modifiers

- Modifiers change the interpretation of certain characters in a regular expression or the way in which Perl finds a match for a regular expression

/ /sm	Treat string as a single long line, but detect multiple lines '.' matches any character including '\n' '^' matches at a line start '$' matches at a line end
/ /i	perform a case-insensitive match

```
$_=" bill\nClinton " ;
if (/( Bill | Hillary ). Clinton/mi){ print " R1 mi : $&\n" }
if (/( Bill | Hillary ). Clinton/si){ print " R1 si : $&\n" }
if (/( Bill | Hillary ). Clinton/smi){ print " R1 smi : $&\n" }
if (/( Bill | Hillary ).^ Clinton/si){ print " R2 si : $&\n" }
if (/( Bill | Hillary ).^ Clinton/smi){ print " R2 smi : $&\n" }
```

```
R1 si : bill
Clinton
R1 smi : bill
Clinton
```

```
R2 smi : bill
Clinton
```

Regular expressions: Modifiers (/ /g and / /c)

• Often we want to process **all matches** for a regular expression, but the following code has not the desired effect

```
$_ = " 11 22 33 " ;
while(/\d+/){ print "Match starts at $-[0]: $&\n"}
```

The code above does not terminate and endlessly prints out the same text:

```
Match starts at 0: 11
```

To obtain the desired behaviour of the while-loop we have to use the **/ /g modifier**:

In **scalar context**, successive invocations against a string will move from match to match, keeping track of the position in the string

In **list context**, returns a list of matched capture groups, or if there are no capture groups, a list of matches to the whole regular expression

Regular expressions: Modifiers (/ /g and / /c)

• With the / /g modifier our code works as desired:

```
$_ = "11 22 33" ;
while(/\d+/g){print "Match starts at $-[0]: $&\n"}
```

Output:

```
Match starts at 0: 11
Match starts at 3: 22
Match starts at 6: 33
```

An example in a **list context** is the following:

```
$_ = "ab 11 cd 22 ef 33" ;
@numbers = (/\d+/g);
print "Numbers : ",join(" | ",@numbers), "\n";
```

Output:

```
Numbers : 11 | 22 | 33
```

Read / /g as: Start to look for a match from the position where the last match using / /g ended

Regular expressions: Modifiers (/ /g and / /c)

/ /g modifier in scalar context:

```
$_ = "11 22 33" ;
while(/\d+/g){print "Match starts at $-[0]: $&\n"}
```

/ /g modifier in a **list context**:

```
$_ = "ab 11 cd 22 ef 33" ;
@numbers = (/\d+/g);
print "Numbers : ",join(" | ",@numbers), "\n";
```

/ /g	In scalar context, successive invocations against a string will move from match to match, keeping track of the position in the string
	In list context, returns a list of matched capture groups, or if there are no capture groups, a list of matches to the whole regular expression

Generating regular expressions on-the-fly

The Perl parser will expand occurrences of **$variable** and **@variable** in regular expressions
→ regular expessions **can be constructed at runtime**

Example:

```
$_="Bart teases Lisa" ;
@keywords = ( "bart" ,"lisa" ,"marge" , 'L\w+' ,"t\\w+");
while($keyword = shift(@keywords)){
print "Match found for $keyword : $&\n" if/$keyword/i;
}
```

Output:

```
Match found for bart : Bart
Match found for lisa : Lisa
Match found for L\w+ : Lisa
Match found for t\w+ : teases
```

Binding operator

Perl offers two binding operators for regular expressions

string =~ /*regexpr*/	true iff *regexpr* matches *string*
string !~ /*regexpr*/	true iff *regexpr* does not match *string*

Note that these are similar to comparison operators not assignments

Most of the time we are not just interested whether these expressions return true or false, but in the side effect they have on the special variables $N that store the strings matched by capture groups

Examples:

```
$name = " Dr Mario Rossi " ;
if($name =~ /( Sig | Dr )?\s*(\w+)/ ){print "Ciao $2\n"}
Ciao Mario
$name = " Davide Neri " ;
if($name =~ /( Sig  | Dr )?\s*(\ w +)/ ){print "Ciao $2\n"}
Ciao Davide
```

Pattern matching in list context

When a pattern match /regexpr/ is used in a list context, then the return value is

- a **list of the strings matched** by the capture groups in regexpr if the match succeeds and regexpr contains capture groups, or
- (a list containing) the **value 1** if the match succeeds and regexpr contains no capture groups, or
- an **empty list** if the match fails

Examples:

```
$name = " Dr Matteo Rossi";
($t,$f,$l)=($name =~ /(Mr|Ms|Mrs|Dr)?\s*(\w+)\s+(\w+)/);
print " Name : $t , $f , $l\n";
Name : Dr , Matteo , Re
```

```
$name = "Mario Verdi" ;
($t,$f,$l)=($name = ~ /(Mr|Ms|Mrs|Dr)?\s*(\w+)\s+(\w +)/);
print " Name : $t , $f , $l\n";
Name : , Mario , Verdi
```

Pattern matching in list context

When a pattern match /regexpr/g is used in a list context, then the return value is

- a list of the strings matched by the capture groups in regexpr **each time regex matches** provided that regexpr contains capture groups, or
- a list containing the string matched by regexpr **each time regexpr matches** provided that regexpr contains no capture groups, or
- an **empty list** if the match fails

```
$string="firefox: 10.3 seconds ; chrome: 9.5 seconds";
%performance =($string =~ /(\w+)\:\s+(\d+\.\d+)/g);
foreach $system (keys %performance){
print " $system -> $performance {$system}\n"
}
```

```
firefox -> 10.3
chrome -> 9.5
```

Text manipulation examples (functions and RegExp)

```
join(":", "a", "b", "c") → "a:b:c"

split(/:/, "a:b:c") → "a", "b", "c"

reverse("ACTG") →  "GTCA" #NOT complement!

"ACCTTG" =~ s/T/U/g → "ACCUUG" # DNA->RNA

"ACCTTG" =~ tr/ACGT/UGCA/ →  "UGGAAC" #complement!

length("abc") → 3

index("ACT", "TTTACTGAA") →  3 # -1 if not found
```

Replace first occurrence of FOO in variable $x wit BAR

```
$x =~ s/FOO/BAR/;
```

"aaaFOObbbFOO" → "aaaBARbbbFOO"

Replace all occurrences

```
$x =~ s/FOO/BAR/g; # g stands for "global"
```

"aaaFOObbbFOO" → "aaaBARbbbBAR"

The thing to substitute can be a regular expression

```
$x =~ s/a+/x/;
```

"aaaFOObbbFOO" → "xFOObbbFOO"

Matches are “greedy”

```
$x =~ s/a.*F/x/;
```

"aaaFOObbbFOO" → "axOO"

If it can't find FOO, s/// does nothing

```
$x =~ s/FOO/BAR/;
```

"aaabbb" → "aaabbb"

Split a sequence in codons:

```
#!/usr/bin/perl
$seq="ATTCGATTCGATCTATATCGGCTAGCTGATCTCTCGAGATCGTCGATATAGC";
my @codons = $seq =~ /\w{3}/g;

print "@codons\n";
```

Exercise:

- Read a FASTA sequence from a file
- Compute EACH possible reading frame (F1,F2,F3, R1,R2,R3)
- Split each reading frame in codons

For each reading frame DO{
  - Print the reading frame
  - Print the codons
}