Introduction to perl programming

Fredj Tekaia
Institut Pasteur
tekaia@pasteur.fr
Objective

• In this course we will introduce the most useful basics of *perl* programming.

• You are assumed to have a first experience with *perl* programming.

• In the end participants should be able to write simple scripts using *perl*.
Objective

Manipulate huge amount of:

• genome data;
• results;
References

• There are several books and sites that can help in the task to develop and improve your knowledge about perl, some examples are:

  • http://www.well.ox.ac.uk/~johnb/comp/perl/intro.html#perlbuiltin
  • http://www.perl.com/pub/q/faq

• You are encouraged to refer to these sites as often as needed
Course Outline

- Introduction
- Data types, comparison operators,
- Main commands
- Examples
What is *perl*?

- *perl* (the ‘Practical Extraction And Reporting Language’, originally called ‘Pearl’);
- A programming language written by and for working programmers;
- It aims to be practical (easy to use, efficient, complete) rather than beautiful (tiny, elegant, minimal);
- Easy to write (when you learn it), but sometimes hard to read.
What makes *perl* so powerful?

Characteristics which make *perl* so powerful and adaptable include:

- Cross-compatible implementations on all major platforms;
- A comprehensive suite of tools for creating, using, managing and extending features;
- Particularly adapted for the manipulation of huge text files (case of sequences, genomes and analyses results)
A simple perl script

**Hello.pl:**

```perl
#!/bin/perl -w
print "Hello, world!\n";
```

```bash
$ chmod a+x hello.pl
$ ./hello.pl
Hello, world!
$ perl -e 'print "Hello, world!\n";'  
Hello, world!
```
Another perl script

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

use strict;
use warnings;

my $filename = 'input.txt';
my @lines = <FILE>

for my $line (@lines) {
    print $line;
}
```

Another perl script

We will often combine Unix and perl commands and/or scripts.

Readseq.pl:

```perl
#!/bin/perl
while(<>){
    s#>#>MYTU_##go;
    print $_[0];
}
```

Chmod a+x readseq.pl
more sequence.prt | readseq.pl > seq.fa
perl script

• Beware *perl* is case sensitive

$val is different from $Val
Perl is different from perl
More is different from more
Bin is different from bin
Print is different from print
While is different from while

…..
Notations

• perl scripts : xx.pl
• Unix scripts: yy.scr

Scripts identifications should be as explicit as possible: readseq.pl; identseq.pl; codons.pl; etc…

• No space should be used in scripts identifications.
Notations

• We will generally consider sequences and databases in “fasta” format and use the following extensions:
  
  • DB.pep (extension “.pep” for protein databases);
  • DB.dna (extension “.dna” for dna databases);
  • seq.prt (extension “.prt” for protein sequences);
  • seq.dna (extension “.dna” for dna sequences);
  • MYTU.seq (extension ".seq" for genome sequences);
Data Types

- Basic types: scalar, arrays, hashes
What Type?

- Type of variable is determined by special leading character

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>$foo</td>
<td>scalar</td>
</tr>
<tr>
<td>@foo</td>
<td>list</td>
</tr>
<tr>
<td>%foo</td>
<td>hash</td>
</tr>
<tr>
<td>&amp;foo</td>
<td>function</td>
</tr>
</tbody>
</table>

- Data types have distinct name spaces
What Type?

- **Scalars - Start with a $**
  Strings, Integers, Floating Point Numbers, References to other variables.

- **Arrays - Start with a @**
  Zero based index;
  Contain an ordered list of Scalars.

- **Hashes - Start with %**
  Associative Arrays without order
  Key => Value
Scalars

- Can be numbers
  
  ```
  $num = 100;
  $num = 223.45;
  $num = -1.3e38;
  ```

- Can be strings
  
  ```
  $str = 'unix tools';
  $str = 'Who\’s there?';
  $str = "good evening\n";
  $str = "one\ttwo";
  ```

- Backslash (\) escapes and variable names are interpreted inside double quotes
# Special scalar variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>$0</code></td>
<td>Name of script</td>
</tr>
<tr>
<td><code>$_</code></td>
<td>Default variable</td>
</tr>
<tr>
<td><code>$$</code></td>
<td>Current PID</td>
</tr>
<tr>
<td><code>$?</code></td>
<td>Status of last pipe or system call</td>
</tr>
<tr>
<td><code>$!</code></td>
<td>System error message</td>
</tr>
<tr>
<td><code>$/</code></td>
<td>Input record separator</td>
</tr>
<tr>
<td><code>$.</code></td>
<td>Input record number</td>
</tr>
<tr>
<td><code>undef</code></td>
<td>Acts like 0 or empty string</td>
</tr>
</tbody>
</table>
Operators

- Numeric: +, -, *, /, %, **;
- String concatenation: .
  
  ```
  $state = "New" . "York";  # "NewYork"
  ```
- String repetition: x
  
  ```
  print "AT" x 3;  # ATATAT
  ```
- Binary assignments:
  
  ```
  $val = 2; $val *= 3;  # $val is 6
  $state .= "City";  # "NewYorkCity"
  ```
## Comparison operators

<table>
<thead>
<tr>
<th>Comparison</th>
<th>Numeric</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equal</td>
<td><code>==</code></td>
<td><code>eq</code></td>
</tr>
<tr>
<td>Not Equal</td>
<td><code>!=</code></td>
<td><code>ne</code></td>
</tr>
<tr>
<td>Greater than</td>
<td><code>&gt;</code></td>
<td><code>gt</code></td>
</tr>
<tr>
<td>Less than</td>
<td><code>&lt;</code></td>
<td><code>lt</code></td>
</tr>
<tr>
<td>Less than or equal to</td>
<td><code>&lt;=</code></td>
<td><code>le</code></td>
</tr>
<tr>
<td>Greater than or equal to</td>
<td><code>&gt;=</code></td>
<td><code>ge</code></td>
</tr>
</tbody>
</table>
Boolean “Values”

```plaintext
if ($codon eq "ATG") { ... }
if ($val) { ... }
```

- No boolean data type;
- 0 is false; Non-zero numbers are true;
- The unary not (!) negates the boolean value;
undef and defined

$f = 1;
while ($n < 10)
{
    # $n is undef at 1st iteration
    $f *= ++$n;
}

• Use defined to check if a value is undef
  if (defined($val)) { ... }
Lists and Arrays

- **List**: ordered collection of scalars;
- **Array**: Variable containing a list;
- Each element is a scalar variable;
- Indices are integers starting at 0;
Array/List Assignment

@teams=("Knicks","Nets","Lakers");
print $teams[0];  # print Knicks
$teams[3]="Celtics";  # add a new element
@foo = ();          # empty list
@nums = (1..100);   # list of 1 to 100
@arr = ($x, $y*6);
($a, $b) = ("apple", "orange");
($a, $b) = ($b, $a);  # swap $a $b
@arr1 = @arr2;
Array/List Assignment


Examples:

$CODONS[0] = "TTT";
$CODONS[1]="TTC";
$CODONS[$#CODONS]="GGG";

$AA[0]="A";
$AA[1]="R";
$AA[$#AA]="V";
More about Arrays and Lists

• Quoted words : `qw`

```perl
@planets = qw/ earth mars jupiter /;
@planets = qw{ earth mars jupiter };
```

• Last element’s index: `@#planets`
  – Not the same as number of elements in array!

• Last element: `@planets[-1]`
Scalar and List Context

@colors = qw< red green blue >;

• **Array interpolated as string:**
  
  print “My favorite colors are @colors\n”;
  
  • Prints My favorite colors are red green blue

• **Array in scalar context returns the number of elements in the list**
  
  $num = @colors + 5; # $num gets 8

• **Scalar expression in list context**
  
  @num = 88; # a one-element list (88)
pop and push

• **push** and **pop**: arrays used as stacks

• **push** adds elements to end of array

```perl
@colors = qw# red green blue #;
push(@colors, "yellow");    # same as
@colors = (@colors, "yellow");
push @colors, @more_colors;
```

• **pop** removes last element of array and returns it

```perl
$lastcolor = pop(@colors);
```
shift and unshift

• **shift** and **unshift**: similar to push and pop on the “left” side of an array
• **unshift** adds elements to the beginning

```perl
@colors = qw# red green blue #;
unshift @colors, ”orange”;
```
• First element is now “orange”

• **shift** removes element from beginning

```perl
$c = shift(@colors);  # $c gets ”orange”
```
Example: shift

#!/bin/perl
@SEQ=`ls *.prt`;
while($file=shift@SEQ)
{
  print "$file\n";
}
sort and reverse

- **reverse** returns a list with elements in reverse order

  ```perl
  @list1 = qw# NY NJ CT #;
  @list2 = reverse(@list1); # (CT, NJ, NY)
  ```

- **sort** returns list with elements in ASCII order

  ```perl
  @day = qw/ tues wed thurs /;
  @sorted = sort(@day); # (thurs, tues, wed)
  @nums = sort 1..10; # 1 10 2 3 ... 8 9
  ```

- **reverse** and **sort** do not modify their arguments
Iterate over a list

• **foreach** loops through a list of values
  ```perl
  @codons = qw# TTT TTC TTA TTG #;
  foreach $codon (@codons)
  {
    print "Codon= $codon\n";
  }
  ```
• Value of *control variable* restored at end of loop
• Synonym for the **for** keyword
• **$_** is the default
  ```perl
  foreach (@codons)
  {
    $_ .= " \n";
    print;  # print $_
  }
  ```
Hashes

• Associative arrays - indexed by strings (keys)

```perl
$cap{"Hawaii"} = "Honolulu";
%cap = ( "New York", "Albany", "New Jersey", "Trenton", "Delaware", "Dover" );

$besthit{YAL001} = YBL101;

• Can use => (big arrow or comma arrow) in place of , (comma)

%cap = ( "New York" => "Albany",    
"New Jersey" => "Trenton",    
Delaware     => "Dover" );
```
Hash Element Access

- `$hash{$key}`
  
  ```perl
  print $cap{"New York"};
  print $cap{"New " . "York"};
  ```

- Unwinding the hash
  
  ```perl
  @cap_arr = %cap;
  - Gets unordered list of key-value pairs
  ```

- Assigning one hash to another
  
  ```perl
  %cap2 = %cap;
  %cap_of = reverse %cap;
  print $cap_of{"Trenton"};  # New Jersey
  ```
Hash Functions

- **keys** returns a list of keys
  
  ```perl
  @state = keys %cap;
  ```

- **values** returns a list of values
  
  ```perl
  @city = values %cap;
  ```

- **Use each to iterate over all (key, value) pairs**
  
  ```perl
  while ( ( $state, $city ) = each %cap )
  {
    print "Capital of $state is $city\n";
  }
  ```
Hash Element Interpolation

• Unlike a list, entire hash cannot be interpolated

  print “%cap\n”;
  – Prints %cap followed by a newline

• Individual elements can

  foreach $state (sort keys %cap)
  {
    print “Capital of $state is $cap{$state}\n”;  
  }
More Hash Functions

- **exists** checks if a hash element has ever been initialized
  ```perl
  print "Exists\n" if exists $cap{"Utah"};
  ```
  - Can be used for array elements
  - A hash or array element can only be defined if it exists

- **delete** removes a key from the hash
  ```perl
  delete $cap{"New York"};
  ```
Merging Hashes

• Method 1: Treat them as lists
  \%h3 = (%h1, %h2);

• Method 2 (save memory): Build a new hash by looping over all elements
  \%h3 = ();
  while ((%k, $v) = each(%h1)) {
    $h3{$k} = $v;
  }
  while ((%k, $v) = each(%h2)) {
    $h3{$k} = $v;
  }
Subroutines

- **sub myfunc { ... }**
  
  $name=“Jane”;

  ...

  sub print_hello {
    print “Hello $name\n”; # global $name
  }

  &print_hello; # print “Hello Jane”

  print_hello; # print “Hello Jane”

  print_hello(); # print “Hello Jane”
Arguments

• Parameters are assigned to the special array `@_`
• Individual parameter can be accessed as `$_[0]`, `$_[1]`, ...

```perl
sub sum {
    my $x;                # private variable $x
    foreach (@_) {        # iterate over params
        $x += $_;
    }
    return $x;
}

$n = &sum(3, 10, 22);  # n gets 35
```
More on Parameter Passing

- Any number of scalars, lists, and hashes can be passed to a subroutine
- Lists and hashes are “flattened”
  \[
  \text{func}(\$x, @y, \%z);
  \]
  - Inside \text{func}:
    - \$_[0] is \$x
    - \$_[1] is \$y[0]
    - \$_[2] is \$y[1], etc.

- Scalars in @ are implicit aliases (not copies) of the ones passed — changing values of \$_[0], etc. changes the original variables
Return Values

• The return value of a subroutine is the last expression evaluated, or the value returned by the return operator
  
  sub myfunc {
    my $x = 1;
    return $x + 2; #returns 3
  }

• Can also return a list: return @somelist;

• If return is used without an expression (failure), undefined or () is returned depending on context
Lexical Variables

• Variables can be scoped (scoped) to the enclosing block with the **my** operator
  
  ```perl
  sub myfunc {
    my $x;
    my($a, $b) = @_;    # copy params
    ...
  }
  ```

• Can be used in any block, such as **if** block or **while** block
  – Without enclosing block, the scope is the source file
use strict

• The **use strict** *pragma* enforces some good programming rules
  – All new variables need to be declared with *my*

```perl
#!/usr/bin/perl -w
use strict;
$n = 1;
# <-- perl will complain
```
Another Subroutine Example

@nums = (1, 2, 3);
$num = 4;
@res = dec_by_one(@nums, $num);

minus_one(@nums, $num);

sub dec_by_one {
    my @ret = @_;  # make a copy
    for my $n (@ret) { $n-- ;}
    return @ret;
}

sub minus_one {
    for (@_) { $_[-- ];}
}
Reading from STDIN

• **STDIN** is the builtin filehandle to the std input
• Use the line input operator around a file handle to read from it
  
  ```perl
  $line = <STDIN>;              # read next line
  chomp($line);
  ```
• **chomp** removes trailing string that corresponds to the value of `$/` (usually the newline character)
while (<STDIN>)
{
    chomp;
    print "Line $. ==> 
$_ \n";
}
# $. = line number
Line 1 ==> [Contents of line 1]
Line 2 ==> [Contents of line 2]
...
• *Diamond operator* `< >` helps *perl* programs behave like standard Unix utilities (cut, sed, …)

• Lines are read from list of files given as command line arguments (@ARGV), otherwise from stdin
  ```perl
  while (<>) {
    chomp;
    print "Line $. from $ARGV is $_\n";
  }
  ```

• `./myprog file1 file2` –
  – Read from *file1*, then *file2*, then standard input

• `$ARGV` is the current filename
Filehandles

- **Use** `open` **to** open a file for reading/writing
  
  ```
  open (IN, "syslog");  # read
  open (IN1, "<syslog");  # read
  open (OUT, ">syslog");  # write
  open (OUT, ">>syslog");  # append
  ```

- **When you’re done with a filehandle, close it**
  
  ```
  close IN; close IN1, close OUT;
  ```
Filehandles

- **Use** `open` **to open a file for reading/writing**

```perl
script.pl file_input1 file_input2 file_output
$IN1=@ARGV[0]; $IN2=@ARGV[1]; $OUT=@ARGV[2];
open (IN1, "$IN1"); # read
open (IN2, "$IN2"); # read
open (OUT, ">$OUT"); # write
```

- **When you’re done with a filehandle, close it**

```perl
close IN2; close IN2; close OUT;
```
Errors

• When a fatal error is encountered, use `die` to print out error message and exit program
  
  ```
  die "Something bad happened\n" if ....;
  ```

• **Always check return value of `open`**
  
  ```
  open (LOG, ">>tempfile") || die "Cannot open log: $!";
  ```

• For non-fatal errors, use `warn` instead
  
  ```
  warn "Temperature is below 0!"
  if $temp < 0;
  ```
open (SEQ, "sequence_file.dna") || die "Cannot open sequence: $!
";
while (<SEQ>)
{
  chomp;
  # do something with $_
}
close SEQ;
Reading Whole File

- In scalar context, `<FH>` reads the next line
  
  ```perl
  $line = <LOG>;
  ```

- In list context, `<FH>` read all remaining lines
  
  ```perl
  @lines = <LOG>;
  ```

- Undefine `$/` to read the rest of file as a string
  
  ```perl
  undef $/;
  $all_lines = <LOG>;
  ```
Writing to a File

open (OUT, ">RESULT") || die "Cannot create file: $!");
print OUT "Some results...
"
printf "$num \\%d entries processed.\n", $num;
close OUT;
File Tests examples

die "The file $filename is not readable" if ! -r $filename;

warn "The file $filename is not owned by you" unless -o $filename;

print "This file is old" if -M $filename > 365;
## File Tests list

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-r</td>
<td>File or directory is readable</td>
</tr>
<tr>
<td>-w</td>
<td>File or directory is writable</td>
</tr>
<tr>
<td>-x</td>
<td>File or directory is executable</td>
</tr>
<tr>
<td>-o</td>
<td>File or directory is owned by this user</td>
</tr>
<tr>
<td>-e</td>
<td>File or directory exists</td>
</tr>
<tr>
<td>-z</td>
<td>File exists and has zero size</td>
</tr>
<tr>
<td>-s</td>
<td>File or directory exists and has nonzero size (value in bytes)</td>
</tr>
</tbody>
</table>
## File Tests

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-f</code></td>
<td>Entry is a plain file</td>
</tr>
<tr>
<td><code>-d</code></td>
<td>Entry is a directory</td>
</tr>
<tr>
<td><code>-l</code></td>
<td>Entry is a symbolic link</td>
</tr>
<tr>
<td><code>-M</code></td>
<td>Modification age (in days)</td>
</tr>
<tr>
<td><code>-A</code></td>
<td>Access age (in days)</td>
</tr>
</tbody>
</table>

- `$_` is the default operand
Manipulating Files and Dirs

- `unlink` removes files
  
  `unlink "file1", "file2"
  or warn "failed to remove file: $!";`

- `rename` renames a file
  
  `rename "file1", "file2";`

- `link` creates a new (hard) link
  
  `link "file1", "file2"
  or warn "can’t create link: $!";`

- `symlink` creates a soft link
  
  `link "file1", "file2" or warn " ... ";`
Manipulating Files and Dirs cont.

- `mkdir` creates directory
  
  `mkdir "mydir", 0755`
  
  or warn “Cannot create mydir: $!”;

- `rmdir` removes empty directories
  
  `rmdir "dir1", "dir2", "dir3";`

- `chmod` modifies permissions on file or directory
  
  `chmod 0600, "file1", "file2";`
if - elsif - else

• if … elsif … else …
  
  if ( $x > 0 ) {
      print "x is positive\n";
  }
  elsif ( $x < 0 ) {
      print "x is negative\n";
  }
  else {
      print "x is zero\n";
  }
unless

• Like the opposite of if

    unless ($x < 0) {
        print "$x is non-negative\n";
    }

    unlink $file unless -A $file < 100;
while and until

while ($x < 100)
{
    $y += $x++;  
}

• until is like the opposite of while

until ($x >= 100)
{
    $y += $x++;   
}

for

• for (init; test; incr) { ... }

# sum of squares of 1 to 5
for ($i = 1; $i <= 5; $i++) {
  $sum += $i*$i;
}
next

- **next** skips the remaining of the current iteration (like `continue` in C)

```perl
# only print non-blank lines
while (<>) {
    if ( $_ eq "\n") { next; }
    else { print; }
}
```
last

- **last** exits loop immediately (like `break` in C)

```perl
# print up to first blank line
while (<>) {
    if ( $_ eq "\n") { last; }
    else { print; }
}
```
Logical AND/OR

• Logical AND : 
  ```
  if ( ($x > 0) && ($x < 10) ) { ... }
  ```

• Logical OR : 
  ```
  if ( ($x < 0) || ($x > 0) ) { ... }
  ```

• Both are short-circuit — second expression evaluated only if necessary
Regular Expressions (RE)

• Plus the following character classes
  \w "word" characters: [A-Za-z0-9_]  
  \d digits: [0-9]  
  \s whitespaces: [\f\t\n\r ]  
  \b word boundaries  
  \W, \D, \S, \B are complements of the corresponding classes above  

• Can use \t to denote a tab
Backreferences

- Support backreferences
- Subexpressions are referred to using \1, \2, etc. in the RE and $1, $2, etc. outside RE

```perl
if (/^this (red|blue|green) (bat|ball) is \1/) {
    ($color, $object) = ($1, $2);
}
```
Matching

• Pattern match operator: /RE/ is shortcut of m/RE/
  – Returns true if there is a match
  – Match against $_
  – Can also use m(RE), m<RE>, m!RE!, etc.
    if (/^\//usr/local//) { ... }
    if (m%/usr/local/%) { ... }

• Case-insensitive match
  if (/new york/i) { ... };
Matching cont.

• To match an RE against something other than \$_, use the binding operator =~
  
  ```perl
  if ($s =~ /\bblah/i) {
    print "Found blah!"
  }
  ```

• !~ negates the match
  
  ```perl
  while (<STDIN> !~ /^#/) {
  ... 
  }
  ```

• Variables are interpolated inside REs
  
  ```perl
  if (/^$word/) {
  ... 
  }
  ```
Substitutions

- **sed-like search and replace with** `s///`
  `s/red/blue/;`
  `\$x =~ s/\w+$/$/;`
  - **m///** does not modify variable; **s///** does

- **Global replacement with** `/g`
  `s/(.)\1/$1/g;`
  `S#(.)\1#$1#g;`

- **Transliteration operator:** `tr///` or `y///`
  `tr/A-Z/a-z/;`
Strings extraction

Substr ($STRING, startPosition, length_substring2extract);

$x = substr($_, 0, 10);
RE Functions

• **split** string using RE *(whitespace by default)*
  ```perl
  @fields = split /:/, "::ab:cde:f";
  # gets ("", ",", "ab", "cde", "f")
  ```

• **join** strings into one
  ```perl
  $str = join "-", @fields;  # gets "--ab-cde-f"
  ```

• **grep** something from a list
  – Similar to UNIX grep, but not limited to using RE
    ```perl
    @selected = grep (!/^#//, @code);
    @matched = grep { $_[>]100 && $_[<]150 } @nums;
    – Modifying elements in returned list actually modifies the elements in the original list
  ```
Examples: split

```perl
#!/bin/perl
while(<>)
{
  @tab=spli(t(//\s+/,$_));
  @TAB=spli(t(//[\t]/,$_));
  @Tab=spli(t(//\s+/,$_,3));
}
```
Running Another program

• Use the `system` function to run an external program
• With one argument, the shell is used to run the command
  – Convenient when redirection is needed
    ```
    $status = system("cmd1 args > file");
    ```
• To avoid the shell, pass `system` a list
  ```
  $status = system($prog, @args);
  die "$prog exited abnormally: $?" unless $status == 0;
  ```
Capturing output

- If output from another program needs to be collected, use the backticks
  ```
  my $files = `ls *.*.prt`;
  ```
  - Collect all output lines into a single string
  ```
  my @files = `ls *.*.dna`;
  ```
  - Each element is an output line
- The shell is invoked to run the command
Environment Variables

• Environment variables are stored in the special hash %ENV

$ENV{’PATH’} = 
"/usr/local/bin:$ENV{’PATH’}";
Example: Word Frequency

#!/usr/bin/perl -w
# Read a list of words (one per line) and
# print the frequency of each word
use strict;
my(@words, %count, $word);
chomp(@words = <STDIN>);  # read and chomp all lines
for $word (@words) {
    $count{$word}++;
}
for $word (keys %count) {
    print "$word was seen $count{$word} times.\n";
}
Modules

• Perl modules are libraries of reusable code with specific functionalities
• Standard modules are distributed with perl, others can be obtained from specific servers:
  
  **CPAN:**
  http://www.cpan.org/modules/index.html

  **Bioperl:**
  http://www.bioperl.org/Core/Latest/bptutorial.html

• Each module has its own namespace
References

• Sites to consider and visit as often as needed:

  • [http://www.well.ox.ac.uk/~johnb/comp/perl/intro.html#perlbuiltin](http://www.well.ox.ac.uk/~johnb/comp/perl/intro.html#perlbuiltin)

  • [http://www.perl.com/pub/q/faqs](http://www.perl.com/pub/q/faqs)
Scripts: Examples

• Use \t (tab) as separator
  useful to read and to create tables (word, excel,...)

#!/bin/perl
while(<>){
  @tab = split(/\s+/,$_);
  print $tab[0];
  foreach $j ( 1 .. $#tab ) { print "\t$tab[$j]"; } 
  print "\n";
}

Scripts: Examples

Blocs of commands that will be often used in *perl scripts*:

**Example 1:**
```perl
#!/bin/perl
while(<>)
{
    @tab=split(/\s+/, $_);
}
```

**Example 2:**
```perl
#!/bin/perl
open(P,"$BESTHITS");
while(<P>)
{
    @tab=split(/\s+/, $_);
    $PNUM{$tab[0]}=$tab[1];
    $FREQP{$tab[1]}++;
}
close(P);
```
Overview of working directories

BCGA

DATAMANIP

RESMINING

GENOMEDB

GCOMP

DUPCONS

ORTH

MUL

data

genanal

allmtuprt.fasta

MTU

MTUseqnew
Notations

Sequence and genome files:

We consider sequences and databases in “fasta” format.
DB.pep (extension “.pep” for protein databases);
DB.dna (extension “.dna” for dna databases);
seq.prt (extension “.prt” for protein sequences);
seq.dna (extension “.dna” for dna sequences);
GSPEC.seq (extension “.seq” for genome database sequences);

Scripts:

script.pl (extension “.pl” for perl scripts);
script.scr (extension “.scr” for unix shell scripts);