Perl One-liner

Philippe Bardou
Cédric Cabau

Table of Content

- Introduction
- Send data to my Perl one-liner
- Store data
- Manipulate data
- Conditional constructs
- Looping constructs
- Perl command options
- Regex

Quiz

Exercices

Practice

Organisation

All the course material (slides, input data, practical exercises, corrections) is available online:

https://web-genobioinfo.toulouse.inrae.fr/~formation/21_Perl

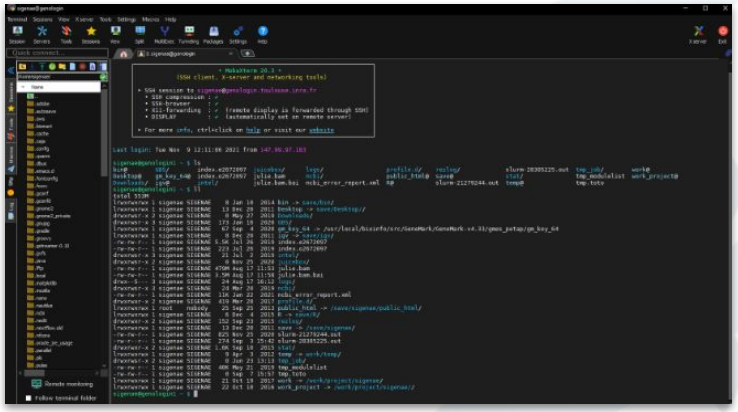
https://bioinfo.genotoul.fr/index.php/events/onelineperl

What are you going to learn?

Your command lines on steroids…

What you should already know?

- How to connect to a remote unix server (mobaXterm)?
- What a unix command looks like?
- How to move around the unix environment?
- How to edit a file?



What will you need during the training?

- An unix terminal (mobaXterm)
- The set of files available at this URL :

  https://web-genobioinfo.toulouse.inrae.fr/~formation/21_Perl
- Set up the working environment for the practical sessions :
  - ➢ Create a working directory and move into it

```
mkdir Formation_Perl ; cd Formation_Perl
```

  - ➢ Add the files required for the practical sessions

```
wget https://web-genobioinfo.toulouse.inrae.fr/~formation/21_Perl/input.tgz
```

Perl's name

- **P**ractical **E**xtraction and **R**eporting **L**anguage
- **Perl** is the language
  perl is the compiler
- Created by Larry Wall in 1987
  Inspired by the C language and the scripting languages sed, awk and shell (sh)
- Perl is an interpreted, versatile language, that is well-suited to processing textual data. Perl is a general-purpose glue for almost anything.
- Official documentation https://perldoc.perl.org

Perl: script or cmd line

Script

```
more hello.pl
print "Hello, World!\n";
perl hello.pl
```

Command line

```
perl -e 'print "Hello, World!\n";'
```

Creating a Perl one-liner

Basic syntax

- all statements should end with `;`
- Perl is case sensitive

our first Perl one-liner

```
perl -e 'print "Hello world!\n";'
```

does the same as

```
perl -le 'print "Hello world!";'
```

Creating a Perl one-liner

```
$ perl -h
Usage: perl [switches] [--] [programfile] [arguments]
  -0[octal]         specify record separator (\0, if no argument)
  -a                autosplit mode with -n or -p (splits $_ into @F)
  -C[number/list]   enables the listed Unicode features
  -c                check syntax only (runs BEGIN and CHECK blocks)
  -d[:debugger]     run program under debugger
  -D[number/list]   set debugging flags (argument is a bit mask or alphabets)
  -e program        one line of program (several -e's allowed, omit programfile)
  -E program        like -e, but enables all optional features
  -f                don't do $sitelib/sitecustomize.pl at startup
  -F/pattern/       split() pattern for -a switch (//'s are optional)
  -i[extension]     edit <> files in place (makes backup if extension supplied)
  -Idirectory       specify @INC/#include directory (several -I's allowed)
  -l[octal]         enable line ending processing, specifies line terminator
  -[mM][-]module    execute "use/no module..." before executing program
  -n                assume "while (<>) { ... }" loop around program
  -p                assume loop like -n but print line also, like sed
  -s                enable rudimentary parsing for switches after programfile
  -S                look for programfile using PATH environment variable
  -t                enable tainting warnings
  -T                enable tainting checks
  -u                dump core after parsing program
  -U                allow unsafe operations
  -v                print version, patchlevel and license
  -V[:variable]     print configuration summary (or a single Config.pm variable)
  -w                enable many useful warnings
  -W                enable all warnings
  -x[directory]     ignore text before #!perl line (optionally cd to directory)
  -X                disable all warnings

Run 'perldoc perl' for more help with Perl.
```

TP - first steps

- Create a working directory and move into it
- Add the files required for the practical sessions
- First `perl` commands:
  - display the `perl` command help
  - write your first `perl` command to display `"Hello world!"`

How to send data to my Perl one-liner?

STDOUT / STDIN

- From STDIN

```
echo Hello World | perl -lne 'print'
cat myfile.txt | perl -lne 'print'
```

```
for i in Hello World
 do echo $i | perl -lne 'print'
done
```

From file

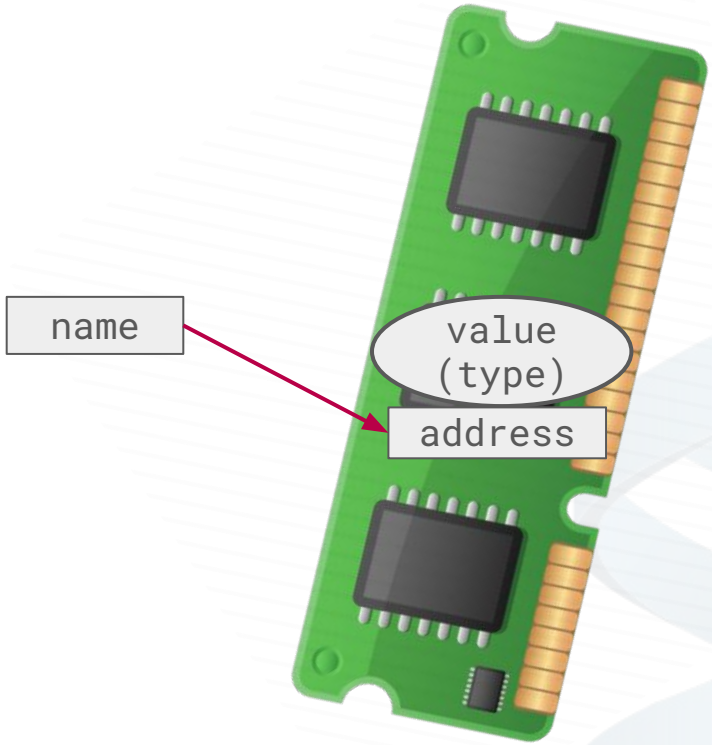
- By giving a file as a parameter

```
perl -lne 'print' myfile.tsv
```

How to store data?

Data types - Glossary

- Variable
  - name
  - address
  - value
  - (type)
- (Declaration)
- Assignment
- Accession
- Test

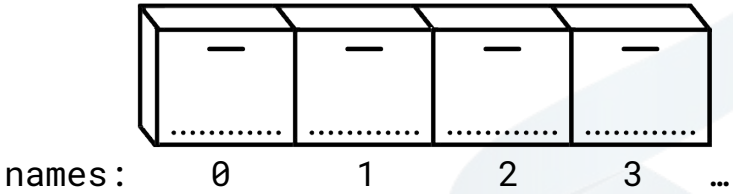


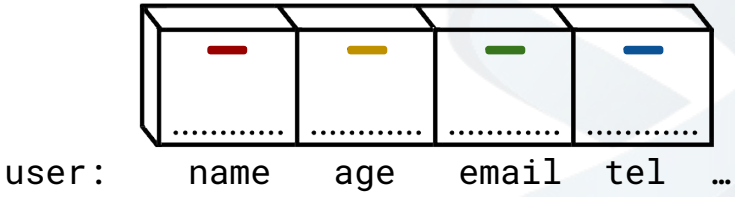
Data types

$scalar

@array

%hash

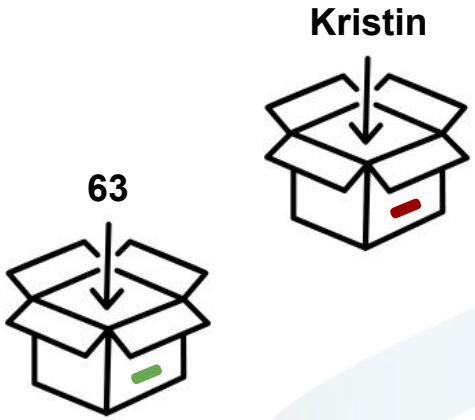




Scalar

`$scalar` variable

- a place where we can store data
- store a single item of data
- need a name
- starts with `$`




```
$age = 63; # assigning
$firstname = 'Kristin';
$login = "$firstname-$age"; # accessing
```

Escaped sequences and quoting

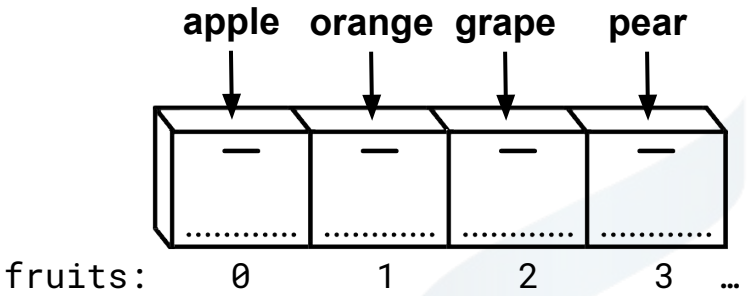
Escaped sequences

- consist of a backslash followed by one character
- special espace sequences
  - `\n` inserts a new line
  - `\t` inserts a tabulation

```
print "$195"; # prints “”
print "\$195"; # prints “$195”
```

`@array` variable

- an ordered list of scalar values
- each value is stored at a specific index
- could be sorted
- starts with `@`



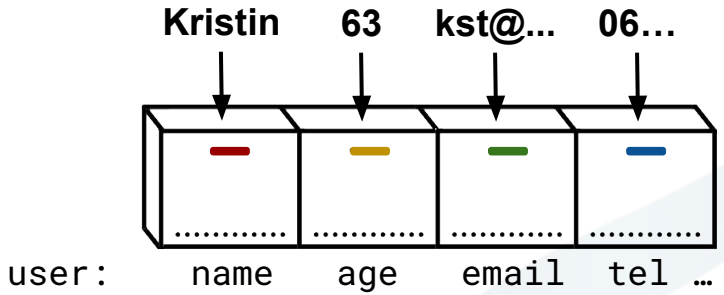

```
@fruits = ("apple","orange","grape","pear"); # assigning
@numbers = (23,42,78);
@random_scalars = ("Kristin",63,"Scott Thomas","$login");
```

`@array` variable

```
@fruits = ("apples","oranges","grapes"); # assigning
print $fruits[0];  # accessing - prints “apples”
$fruits[2] = "lemons";
print $fruits[2];  # prints “lemons”
print $fruits[-1]; # last element - prints “lemons”
```

%hash variable

- a set of key-value pairs
- keys are unique strings
  the values can be any scalars
- hashes are not sorted
- starts with %




```
%user = ("name"=>"Kristin",
         "age"=> 63,
         "email"=>"kst@gmail.com"); # assigning
```

%hash variable

```
$name = $user{"name"};      # accessing
print $user{"age"};         # prints “63”
$user{"tel"} = "01020304"; # assigning
$user{"age"} = 64;
%user=("name","Kristin","age",63,"email","kst@gmail.com");
```

Magic variables

- `$_` - is the “default variable” - input space
- `@ARGV` - the command line arguments to your script
- `$/` - the input record separator, newline `\n` by default
- `$.` - line number for the last filehandle accessed

TP - variables

- Write a one-liner Perl command that initializes a scalar variable with `"Hello world!"` and then display it

- Write a one-liner Perl command that initializes an array variable with three values : `"Hello"`, `"world"`, `"!"` and then display `"world"`

- Write a one-liner Perl command that initializes a hash variable with three keys/values: `"name":"YOUR NAME"`, `"age":"YOUR AGE"` and then display `"You are truly in the prime of life,YOUR NAME!"`

How to manipulate data?

Arithmetic operators

- standard
  - add `+`, subtract `-`, multiply `*`, divide `/`
- less standard
  - mod `%`, exp `**`

```
$speed = $distance / $time;
$vol = $length * $breadth * $height;
$area = $pi * ($radius ** 2);
$odd = $number % 2;
```

Arithmetic operators

Shortcut operators : increment, decrement

```
$a = $b = 3;
$a = $a + $b;
# can be abbreviated to
$a += $b;
$a++; # same as $a += 1 or $a = $a + 1
$a--;
```

String operators

- concatenation

```
$name = $firstname." ".$name;
```

- shortcut

```
$page .= $line; # same as $page = $page.$line
```

- funny (repetition)

```
$knocking = "toc" x 3; # $knocking is “toctoctoc”
$line = "-" x 10;      # $line is “----------”
```

File test operators

- does the file exist

```
-e $file
```

- is the file a normal file

```
-f $file
```

- is the file a directory

```
-d $file
```

Some other operators

- Range operator `..`

```
@int = (1..6); # @int is (1,2,3,4,5,6)
```

- Conditional operator `COND ? THEN : ELSE`

```
$value = ($value <= $max) ? $value : $max;
```

- I/O operator `<>`

```
$first_line = <STDIN>;
```

TP - operators

- Calculate the price of an item during the sales. Complete the following line to return the price remaining to be paid after applying a discount of 10, and 40%

```
echo 275 | perl ...
```

- Count the number of lines in the fastq file (ERR.fastq) using `wc -l` and then use Perl to determine whether the file is valid (correct number of lines).

Builtin functions

- can take more arguments than operators
- arguments follow the function name
- used either with or without parentheses around its arguments
- see Perlfunc for a complete list

Functions can return scalar or list (or nothing)

```
$age = 29.75;
$years = int($age);
@list = ("a","random","collection","of","words");
@sorted = sort(@list);
# @sorted is ("a","collection","of","random","words")
```

String functions

- `chomp` removes the last character if it is a newline (`\n`)
- `length` returns the length of a string
- `uc`, `lc`, `ucfirst`, `lcfirst` allow playing with case

```
$fname = <STDIN>;       # echo 'kristin'|perl -e '… '
$len = length($line);
print $len              # prints “8”
chomp($fname);
print length($line);    # prints “7”
print uc($fname);       # prints “KRISTIN”
print ucfirst($fname); # prints “Kristin”
```

String functions

- `substr EXPR,OFFSET,LENGTH` returns substring from a string
- `split /PATTERN/,EXPR` cut the string on a pattern

```
$str = "Kristin Scott Thomas";
print substr($str,0,7); # prints “Kristin”
print substr($str,8,5); # prints “Scott”
@x = split(/ /, $str);
# @x is ("Kristin","Scott","Thomas")
```

Scalar Context Sensitivity

- context refers to the kind of value that is expected
- influence how an expression or a function behaves
- two main types of context in Perl
  - **list** context - Perl gives the list of elements
  - **scalar** context - it returns the number of elements in the array

```
$str = "Kristin Scott Thomas";
@x = split(/ /, $str);
$x = split(/ /, $str);
print join(" ", @x); # prints “Kristin Scott Thomas”
print $x; # prints “3”
```

Array manipulation

- `push LIST,EXPR` add a new element to the end of an array
- `pop LIST` removes and returns the last element in an array
- `unshift`, `shift` do the same for the start of an array
- `sort LIST` returns a sorted list
- `join EXPR,LIST` takes an array and returns a string
- `scalar LIST` returns the array length

```
push(@array, $newlast);

$last = pop(@array);

unshift(@array, $newfirst);

$first = shift(@array);

@sorted = sort(@random);

print join(" ", @line);

$len = scalar(@array);
```

Hash functions

- `delete EXPR` removes a key/value pair from a hash
- `exists EXPR` tells you if a key exists in a hash
- `keys HASH` returns a list of all the keys in a hash
- `values HASH` returns a list of all the values in a hash

```
%user = (
 "name"=>"Kristin",
 "age"=> 63,
 "email"=>"kst@gmail.com"
);
delete($user{$email});
exists $user{"age"} ? … : …
@keys = keys(%user);
# @keys is ("name","age")
@values = values(%user);
# @values is ("Kristin",63)
```

Some other functions

- `int(EXPR)` returns the integer portion of `EXPR` - ~~rounding~~
- `q()` same as `''`
- `qq()` same as `""`
- `grep EXPR,LIST` returns elements for which `EXPR` is true
- `system(EXPR)` or `` `EXPR` `` executes `EXPR` as a system cmd

```
print 17/3," ",int(17/3)
# prints 5.66666666666667 5


print q(I said, "You said,
'She said it.'");
# prints
I said, "You said, 'She said it.'"


my @f = grep(/^a/, @fruits);


print `ps -l $$`;
# prints
    PID TTY    STAT TIME COMMAND
2189865 pts/43 S+   0:00 perl -le print `ps $$`
```

TP - functions

- Create an array that contains the first five natural numbers. Print the array. Create an new array shifting the elements left by one position (element 1 goes to 0) and setting the first element in the last position. Print the new array.

- Use the 3 tables below to print the favorite shoe color and size per each family member. Output lines should be in the format: `"Homer wears brown shoes size 12"`.

```
@family = ("Homer","Marge","Bart");
@shoe_color = ("Marge","blue","Bart","yellow", "Homer","brown");
@shoe_size = ("Bart",8,"Homer",12,"Marge",10)
```

Conditional constructs

For what purpose?

- Conditional constructs allows us to choose different routes of execution through the program
- This makes for far more interesting programs
- The unit of program is a block of code
- Blocks are delimited with braces `{ . . . }`
- Blocks are controlled by the evaluation of an expression which could be true or false
- But what is truth?

What is truth?

- In Perl, it's easier to answer the question "what is false?"
  - `0`, `'0'` (the number or string 0)
  - `''` (the empty string)
  - `undef` (an undefined value)
  - `()` (an empty list)
- Everything else is true
  - even the string `'false'`

```
$x = undef; undef($x);
```

Comparison operators

Compare two values

- equal?
  - numeric `==`, `!=`
  - string `eq`, `ne`
- greater than?
  - numeric `>`, `<`, `>=`, `<=`
  - string `gt`, `lt`, `ge`, `le`

Comparison operators

Combine two or more conditional expressions

- `EXPR_1 and EXPR_2` - true if both `EXPR_1` and `EXPR_2` are true
- `EXPR_1 or EXPR_2` - true if either `EXPR_1` and `EXPR_2` are true
- `not EXPR` - true if `EXPR` is false
- alternative syntaxes
  - `&&` means `and`
  - `||` means `or`
  - `!` means `not`

First conditional statement

- if -

```
if (EXPR) { BLOCK }
```

Only executes `BLOCK` if `EXPR` is true

```
if (exists $user{"age"}) {
    print "$user{'name'} is $user{'age'} years old\n";
}
```

First conditional statement

- **unless - negative version of if**

```
unless (EXPR) { BLOCK }
```

Only executes `BLOCK` if `EXPR` is false

More readable version of

```
if (not (EXPR)) { BLOCK }
```

First conditional statement

Clever way of making one line block code more readable

```
if (EXPR) { BLOCK }
```

could be written

```
BLOCK if (EXPR)
```

```
print "Hello you\n" if ($user{"name"} eq "Kristin");
print "My contact list is empty\n" unless %user;
```

Extended if statement

- if - else -

```
if (EXPR) { BLOCK1 } else { BLOCK2 }
```

If `EXPR` is true, execute `BLOCK1` otherwise execute `BLOCK2`

```
if (exists $user{"age"}) {
    print "$user{'name'} is $user{'age'} years old\n";
} else {
    print "The age of $user{'name'} is not known\n";
}
```

TP - conditional statement

- Using the `samples.tsv` file and knowing the size of the genome (2,922,600,443 bp), display the number of samples with coverage <10X, between 10 and 50X and >50X

- How was the header taken into account and why?

Looping constructs

While looping constructs

- while - repeat the same code

```
while (EXPR) { BLOCK }
```

Repeat `BLOCK` while `EXPR` is true

```
while ($contacts) {
    print "I've got a friend\n";
    $contacts--;
}
```

For looping constructs

- for - more complex loop

```
for (INIT; EXPR; INCR) { BLOCK }
```

Execute INIT

if `EXPR` is false, exit loop

otherwise, execute `BLOCK`, execute `INCR` and retest `EXPR`

```
for ($i=0; $i<=$max; $i++) {
    print $fruits[$i]."\n";
}
```

Foreach looping constructs

- foreach - more friendly loop over list

```
foreach VAR (LIST) { BLOCK }
```

For each element of `LIST`

set `VAR` to equal the element

execute `BLOCK`

```
foreach $fruit (@fruits) {
    print "$fruit\n";
}
```

Breaking out of loops

- `next` - jump to next iteration of loop
- `last` - jump out of the loop
- `redo` - jump to start of the same iteration of loop

Special code blocks

- `BEGIN` - code block executes immediately

```
BEGIN { BLOCK }
```

- `END` - code block is executed as late as possible

```
END { BLOCK }
```

Magic variables

- `$_` - is the “default variable” - input and pattern-searching space
- `@F` - autosplit mode `-a` (splits `$_` into `@F`)
- `@ARGV` - the command line arguments to your script
- `$/` - the input record separator, newline `\n` by default
- `$.` - line number for the last filehandle accessed
- `$a`, `$b` - special package variables when using `sort()`

TP - functions and structures

- Read the `samples.tsv` file, calculate the read length per line with `perl` and use the `sort` and `uniq` shell commands to get the number of samples with the same read length.

- Same exercise but without using `sort` and `uniq`.

- Read the `samples.tsv` file, calculate the average number of reads and the average number of bases between all samples.

The perl command options

The perl command options

The perl command has relevant options to write one-liners

- `-e` to enter one line of program
  Perl does not look for a file name to execute
- `-l` to
  - remove the record separator on input
  - add the record separator to all `print` instructions on output
- `-n` to enclose your program in a loop of the following type

```
while (<STDIN>) { my_program }
```

The perl command options

The perl command has relevant options to write one-liners

- `-p` to enclose your program in a `while` loop, like the `-n` option, and display the lines automatically

```
while (<STDIN>) { my_program } continue { print }
```

- `-a` to enable the auto-split mode when used with `-n` or `-p` and thus an implicit `split` command to the `@F` array is done at the start of the `while` loop (an alternate delimiter may be specified using `-F`)

```
while (<STDIN>) { @F=split(' '); my_program }
```

Quiz perl command options

https://digistorm.app/p/8191099

Regex - enter the wonderful world of regular expressions

Regular expressions

Allow you to search patterns that match strings

The simplest regex is a simply

```
"Hello world" =~ /World/;
```

Expression like this are useful in conditionals

```
print "Matches" if $string =~ /World/;
```

The sense of the match can be reverse by using the `!~` operator

```
print "No match" if $string !~ /World/;
```

Regular expressions

To specify where the regex should match we would use the anchor metacharacters `^` and `$`

The `^` means match at the beginning of the string

```
print "Start with Hello" if $string =~ /^Hello/;
```

The `$` means match at the end of the string

```
print "Ends with World" if /World$/;
```

Character classes

A character class allows a set of possible characters, rather than just a single character. Character classes are denoted by brackets `[ ]`

```
/[bcr]at/;       # match “bat”, “cat” or “rat”
/[yY][eE][sS]/; # match “yes” case-insensitive
/yes/i;          # match “yes” case-insensitive (modifier)
```

The special character `-` acts as a range operator

```
/item[0-9]/;     # match “item0” or … or “item9”
/[0-9a-f]/i;     # match a hexadecimal digit
```

Character classes

The special character `^` in the first position of a class denotes a negated character class - matches any character but those in the brackets

Both `[ ]` and `[^]` must match one character or the match fails

```
/[^a]at/; # ~~“aat”~~ or ~~“at”~~ but “bat”,“cat”,“1at”,“ at”,…
/[^0-9]/; # match non-numeric character
/[a^]at/; # match “aat” or “^at”; “^” is ordinary
```

Character classes

Perl has several abbreviations for common character classes

- `\d` is a digit character `[0-9]`
- `\D` is a non-digit character `[^\d]`
- `\s` is a whitespace character `[ \t\r\n\f]`
- `\S` is a non-whitespace character `[^\s]`
- `\w` is a word character `[0-9a-zA-Z_]`
- `\W` is a non-word character `[^\w]`
- the dot `.` matches any character but `[^\n]`

Character classes

Abbreviations can be used both inside and outside of character classes

```
/\d\d:\d\d:\d\d/; # match a hh:mm:ss time format
/\d\s/;   # match any digit or whitespace character
/..rt/;   # match any two chars, followed by “rt”
/end\./;  # match “end.” (escape character)
/end[.]/; # again match “end.”
```

Quantifiers

Quantifiers can be used to specify the number of occurrences
Without quantifier, the number of times to match is exactly one

- `*` means zero or more
- `+` means one or more
- `?` means zero or one
- `{3}` means exactly 3
- `{3,6}` means between 3 and 6
- `{3,}` means at least 3 and `{,3}` means at most 3
- `/\d{4}:\d\d:\d\d/` - matches a YYYY:MM:DD date format

Quantifiers

If you want to match the minimum number of times possible, follow the quantifier with a `?`, this will change just the "greediness" not the meaning

```
"a,b,c,d,e" =~ /,.+,/;  # match “b,c,d”
"a,b,c,d,e" =~ /,.+?,/; # match “b”
```

Use parentheses for grouping

The grouping metacharacters `( )` allow a part of a regex to be treated as a single unit. Parts of a regex are grouped by enclosing them in `( )`

```
/[+-]?\d+\.\d+/;    # match a signed float e.g -1.75
```

But if we want to make the decimal part optional, we need to group the dot and the numbers that follow

```
/[+-]?\d+(\.\d+)?/ # match any rational number
```

Matching this or that

- We can match different strings using the alternation metacharacter `|`
- Again, we can use parentheses for grouping

```
/cat|dog|bear/;      # match “cat” and “cat and dog”
/^(cat|dog|bear)$/; # match “cat” not “cat and dog”
```

Extracting matches

- The grouping metacharacters `( )` also allow the extraction of the parts of the string that matched
- For each grouping, the part that matched inside goes into special variables `$1`, `$2`, `$3`, etc

```
# extract time in hh:mm:ss format
$time =~ /(\d\d):(\d\d):(\d\d)/;
($hour,$min,$sec) = ($1,$2,$3)
```

Search and replace

- Substitution is performed using `s/PATTERN/REPLACEMENT/`
- The replacement replaces whatever is matched with the regex

```
$x =~ "'Good cat!'";
$x =~ s/cat/dog/;   # $x is "'Good dog!'"
$x =~ s/'(.*)'/$1/; # $x is "Good dog!"
```

- Use the global modifier to replace all occurrences of the regex

```
$x = $y = "4 by 4";
$x = s/4/four/;     # $x is "four by 4"
$y = s/4/four/g;    # $y is "four by four" (modifier)
```

Magic variables

- `$_` - is the “default variable” - input and pattern-searching space
- `@F` - autosplit mode `-a` (splits `$_` into `@F`)
- `@ARGV` - the command line arguments to your script
- `$1`, `$2`, … - subpatterns from capturing parentheses of a pattern match
- `$/` - the input record separator, newline `\n` by default
- `$.` - line number for the last filehandle accessed
- `$a`, `$b` - special package variables when using `sort()`

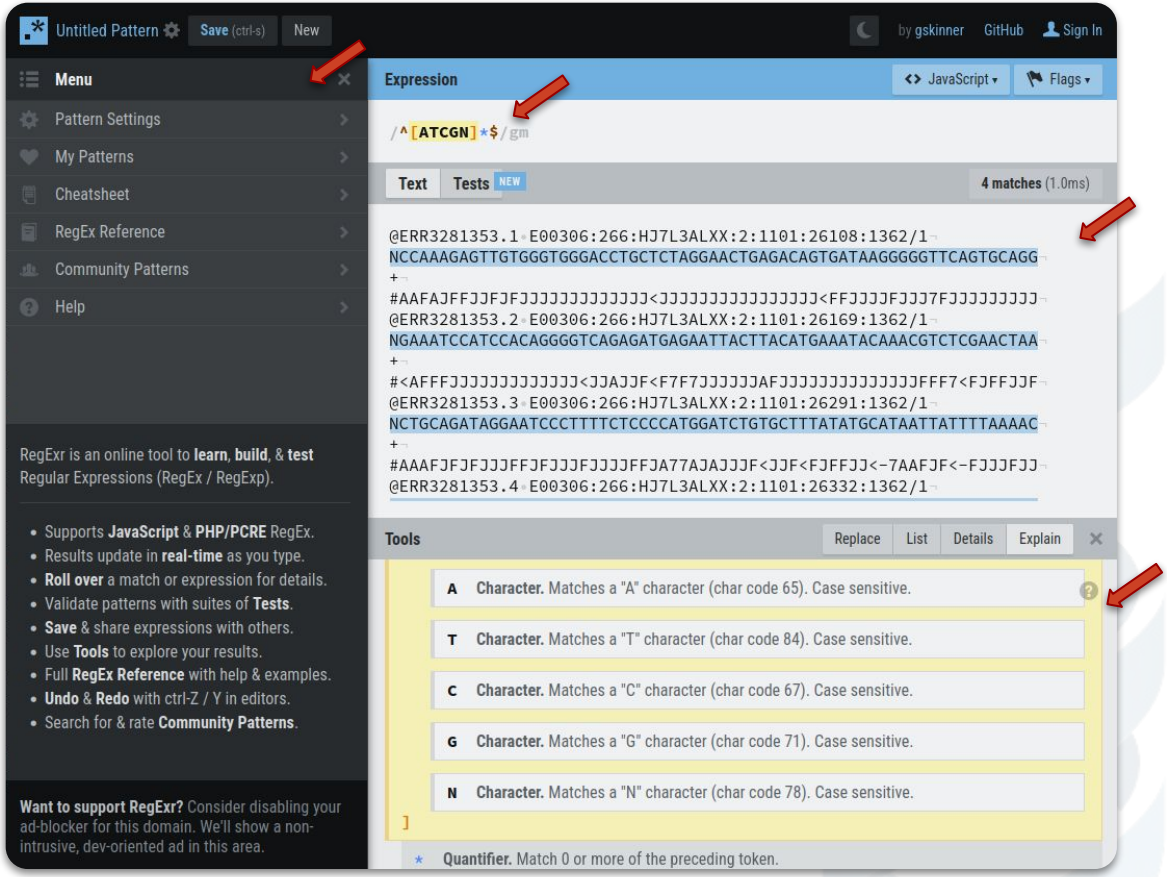
Quiz Perl regex

https://digistorm.app/p/8883214

Learn / test your regex

https://regexr.com/

- Documentation
- Build your regex
- Enter text to test
- Explain your regex
- List all matches



TP - regex

- Write a regular expression to check whether a DNA sequence begins with `ATG` and ends with `TAA`, `TAG` or `TGA`

- Write a regular expression to check the validity of an email address

TP - Build cmd files to run on a cluster

- Write a Perl one-liner that generates a bwa command file from the `samples.tsv` file (e.g. `bwa mem REF.fa ERR3281353_1.fastq.gz ERR3281353_2.fastq.gz | samtools sort - > ERR3281353.bam`)

TP - Stat on GFF

- Write a Perl one-liner that counts and displays the number of genes for each biotype in the GFF file

TP - Stat on GFF

- Write a Perl one-liner which, per chromosome, counts the total number of genes, the number of genes on each strand, and calculates the average length of these genes in the GFF file

TP - Correspondence table

- Write a Perl one-liner which adds the column `sample_alias` of the file `sample_names.tsv` to the file `samples.tsv`.

- Write a Perl one-liner that replaces the `sample_accesssion` column in the `samples.tsv` file with the `sample_alias` column in the `sample_names.tsv` file.