A (very) brief introduction to Perl

Fabien Burki
GDM workshop
Mayaguez, Puerto Rico 2010

Perl ???

- Perl is a programming language
- Invented by Larry Wall in 1987
- Originally developped for text manipulation -> First choice of early bioinformaticians
- Nowadays there are other widespread languages in bioinformatics (Python, RUBY), but Perl remains a standard
- Very versatile, "There is more than one way to do it"
- Bioperl

- Perl is usually pre-installed on unix/linux and mac OS systems
- Windows users typically install "Strawberry Perl" or "ActivePerl"

Why bother?

- More and more in biology, and this is even truer in genomics, the massive amount of data that is generated ever day makes it unrealistic to work by hand
- We want automated procedures, so that we can drink beers whilst the computer is working for you
- A very simple example:
  - We have 20000 sequences in a fasta file with various names format
  - `>GD4HFQN02C7RN7 length=66 xy=1201_2117 region=2 run=R_2010_03_17_16_00_52_`
  - `>contig00007 length=1179 numreads=189 gene=isogroup00001`
  - `>NODE_21_length_255_cov_191.129410`
  - We want to reformat the names in a more meaningful manner
  - `>Species_name|ID`
- By hand: open the file -> manually replace the 20000 names
  - 5 sec / seq = 100'000 s =~28 hours of a REALLY boring job
- With a simple script:
  - 5 min to write the script
  - 15 sec to run the script

Why bother

- You could just ask the computer geek in your department
- You could hire a computer scientist
- Or you could invest a small amount of time to learn the basics of coding

- Today, we won't have time to learn even the basics…
- How it looks like, and a practical example of why it is very helpful

- A limited knowledge is useful because often you can just adapt to your needs the tons of scripts that you can find on the web

A Perl script

- A Perl script is just a text file
- It will be interpreted by the Perl interpreter
- Use any text editor, but one that has a "Perl mode" will make your life easier (TextWrangler, Emacs, Crimson Editor)
- By convention, Perl script files end with .pl
- A silly example: hello_world.pl

hello_world.pl

- #!/usr/bin/perl: a very special comment. What follows is the name of the program that actally executes the rest of the file
- #: comment line
- print: a Perl function
- \n: new line
- ; ends lines

```
#!/usr/bin/perl
# hello_world.pl
#a silly perl example
print "hello, world\n";
```

- Common mistakes:
  - `#!/usr/bin/perl` OR `#!/usr/local/bin/perl`
  - `#!C:\Perl64\bin\perl.exe` OR `#!C:\Perl\bin\perl.exe`
  - chmod a+x

hello_world_1.pl

```
# Variables are what makes programming powerful

#!/usr/bin/perl
# hello_world_1.pl

my $your_first_variable = "Hello, world !";
print "$your_first_variable\n";
```

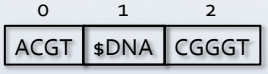
Use "my" to declare your variables

Perl variables

- Variables make programming powerful
- A variable is a bit of memory in the computer that is ready to be assigned a value

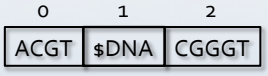
Perl variables

- Variables make programming powerful
- A variable is a bit of memory in the computer that is ready to be assigned a value
- Perl has 3 main built-in variable types
  - Scalars: $scalar_name (a single string)
    - my $DNA = 'ATGGGTCACGTA';
  - Arrays: @array_name (an ordered collection of scalar values)
    - my @dna_fragment = ('ACGT', $DNA, 'CGGGT');

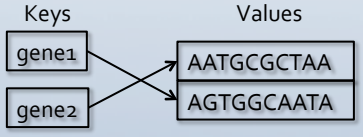


Perl variables

- Variables make programming powerful
- A variable is a bit of memory in the computer that is ready to be assigned a value
- Perl has 3 main built-in variable types
  - Scalars: $scalar_name (a single string)
    - my $DNA = 'ATGGGTCACGTA';
  - Arrays: @array_name (an ordered collection of scalar values)
    - my @dna_fragment = ('ACGT', $DNA, 'CGGGT');



  - Hashes: %hash_name (a collection of pairs of scalar values, called keys and values)
    - my %genes = ('gene1' => 'AATGCGCTAA', 'gene2' => AGTGGCAATA')



Perl functions

- A Perl function is a human-readable reserved word that will perform an action
- It takes one or more arguments passed as a list ("")

- print: print to terminal or to a file
- chomp: remove terminal new line from string variable
- close: close a filehandle
- exit: exit from the script
- my: create a local variable
- open: open a file for reading or writing
- system: execute an operating system command
- split: split a string into pieces according to a pattern

transcription.pl

- The binding operator =~
  - Apply the operation on the right to the string in the variable on the left
- The substitution operator s/1/2/g
  - s indicates this is a substitution
  - 1 is the element that will be substituted
  - 2 is the element that will replace 1
  - g is a modifier that stands for global, i.e. make the substitution throughout the entire string

```
#!/usr/bin/perl
#transcription.pl

# transcription: DNA to RNA
my $DNA = 'ATGCGGTGC';
print "here is the starting DNA:\n";
print "$DNA\n";
my $RNA = $DNA;
$RNA =~ s/T/U/g;
print "here is now the RNA:\n";
print "$RNA\n";

exit;
```

Reading in files

- To read a file in your program, you need a filehandle
- It's not important to understand what they really are, they're just things you use when dealing with files
- A filehandle enclosed in angle brackets is how you bring in data from some source outside the program
- In this example, we called the filehandle IN

The hard-coded way

```
#!/usr/bin/perl
# read_in.pl

my $file = 'sequence.txt';

open (IN, "$file"); #open the file, and associate a filehandle with it
my $sequence = <IN>; #this is the actual reading of the data from the file
                     #we get the sequence that was being temporarily handled by IN
                     #we store the data into the variable $sequence
close IN; #we've got the data, we can close the filehandle

print "$sequence\n";
exit;
```

Reading in files

The interactive way

```
#!/usr/bin/perl
# read_in_1.pl

print "Please enter the file name:\n";
my $file = <STDIN>;
chomp $file;

open (IN, "$file");
my $sequence = <IN>;
close IN;

print "$sequence\n";
exit;
```

The fast way (from the command line)

```
#!/usr/bin/perl
# read_in_2.pl

my $file = shift;

open (IN, "$file");
my $sequence = <IN>;
close IN;

print "$sequence\n";
exit;
```

Control structures - conditions and loops

- Testing for a conditional are among the most powerful features of computer languages
- They allow the program to take alternative directions depending on the condition
- Example: *if-else* or *unless* conditional statements

- A loop allows to repeatedly execute a block of statements until a condition if no longer valid
- Example: *while*, *foreach* or *for* loops

Control structures - conditions and loops

if - elsif -else

```
#!/usr/bin/perl
# if_else.pl

my $i = 3;
my $j = 4;

print "\nThis is another silly example\n\n";

if ($i == $j) {
     print "$i equals $j\n\n";
}

elsif ($i < $j) {
     print "$i is smaller than $j\n\n";
}

else {
     print "$i is bigger than $j\n\n";
}

exit;
```

while

```
#!/usr/bin/perl
# while.pl

print "\nLast silly example\n\n";

my $number = 0;

while ($number < 5) {
     print "$number is below 5\n";
     $number = $number + 1; # what happens
     # if you forget this line?
}
```

Regular expressions

- A great thing that will make you feel you're the king of the world
- Reg exp are ways of matching one or more strings using special operators
- They can be as simple as a word, which matches the word itself, ar they can be very complex and made to match a large set of different words

```
#!/usr/bin/perl
# RegExp.pl

# ok it wasn't the last silly example, here is another one

my $h = "who is afraid of JF?";

if ($h =~ /JF/) {
     print "I am\n";
}

exit;
```

Regular expressions

```
#!/usr/bin/perl
# RegExp_1.pl

# here is better example

my $h = "who is afraid of JF?";

if ($h =~ /\s\w{2}\?/) {
     print "I am\n";
}

exit;
```

```
#!/usr/bin/perl
# RegExp_2.pl

# here is an even better example

my $h = "who is afraid of JF?";

if ($h =~ /(\S+)\s(\w{2})\s(\S+)\s(\of)\s(\w{2})\?/) {
     print "I am $3 $4 $5\n";
}

exit;
```

- Regular Expression Atoms:
  - \d: any digit 0-9
  - \D: a non-digit
  - \w: any letter a-z A-Z and the underscore _
  - \W: a non-letter
  - \s: a white space
  - \S: a non-whitespace
  - \t: a tab
  - . : everything
- Anchors:
  - ^: matches beginning of a string
  - $: matches end of a string

- Quantifiers:
  - ?: matches zero or one
  - *: matches zero or more
  - +: one or more
  - {3}: exactly 3 times
  - {2,4}: between 2 and 4 times
  - {2,}: at least 2 times

Regular expressions

- Useful tools:
  - Reggy mac (http://reggyapp.com/)
  - RegEx tester windows (http://download.cnet.com/RegEx-Tester/3000-2229_4-10818282.html)
  - regexpal online (http://regexpal.com/)

A real-world example: blast parsing

- Batch blast searches generate huge outputs, unfriendly to look at
- Often we are only interested in a couple of things from these outputs
- SOLUTIONS: use a Perl script to parse the blast outputs and pick only the information relevant for your project
- Example:
  - We want to parse a blast output and are interested in the name, the score and e-value of the best hit for each query

Bioperl - bioperl.org

- Bioperl is an open source bioinformatics toolkit for:
  - format conversion
  - report parsing
  - data manipulation
  - sequence analysis
  - batch processing
  - much much more…
- It's unlikely that you will find a script built to fit your exact needs
- But you will find a large collection of Perl modules that you can customize and build your scripts around
- It's so large that all "basic" genomic stuff have their bioperl modules
- "Don't reinvent the wheel"
- A very good start: http://www.bioperl.org/wiki/HOWTOs

Further readings