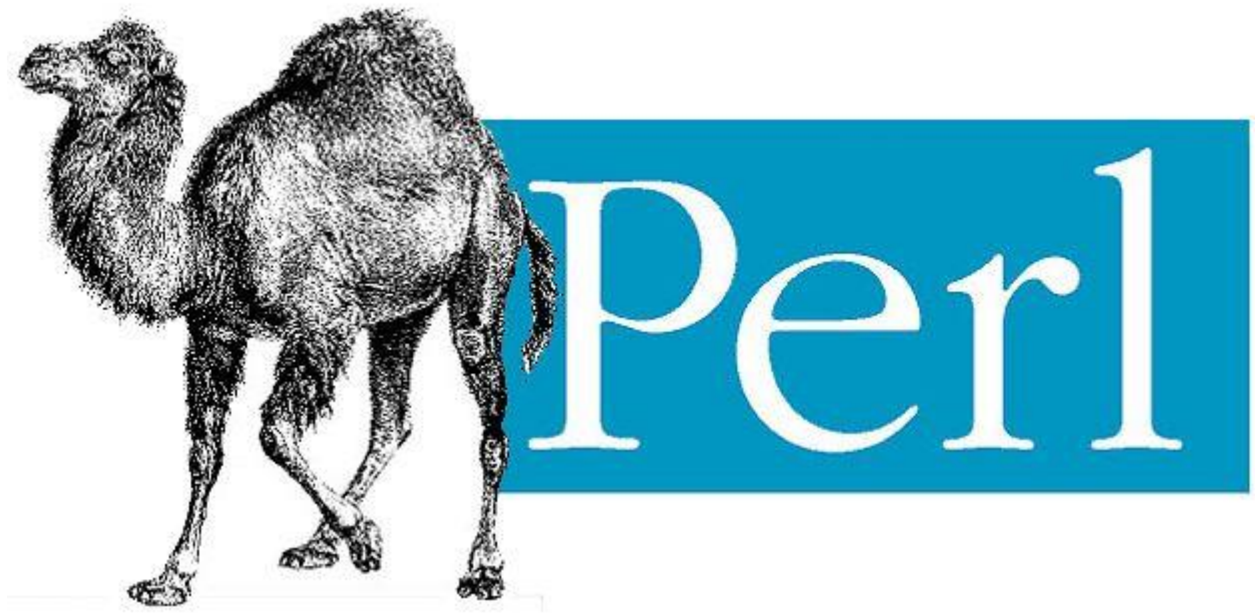
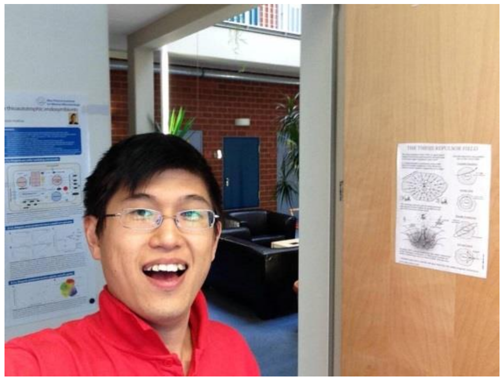
Introduction to Perl



Your friendly tutors ;)

Lizbeth Sayavedra
lsayaved@mpi-bremen.de



Brandon Kwee Boon Seah
kbseah@mpi-bremen.de

Juliane Wippler
jwippler@mpi-bremen.de

Course Material

- Course material will be shared via owncloud and github
  - slides
  - ipython notebooks
  - exercises
  - solutions to exercises
  - cheat sheets
  - scripts

- you’ll get homework exercises each day
  - we will share the solutions to the exercises the next day

What you will learn in this course

Day 1
- Intro & write first Perl program (Juliane)
- Data structures: scalars (Juliane)
- Control structures: if, else, elsif (Juliane)
- Intro to regular expressions (Juliane)
- More regular expressions (Liz)

Day2
- Data structures: arrays (Liz)
- Data structures: hashes (Liz)
- Looping over arrays (Liz)
- More control structures: while, for, foreach (Brandon)
- Reading and writing files (Brandon)

Day 3
- Subroutines and modularization (Brandon)
- Executing external programs within Perl (Brandon)
- Review of material, exercises, homework

What you will learn in this course

Day 1
- Intro & write first Perl program (Juliane)
- Data structures: scalars (Juliane)
- Control structures: if, else, elsif (Juliane)
- Intro to regular expressions (Juliane)
- More regular expressions (Liz)

might be shifted Day1 -> Day2

Day2
- Data structures: arrays (Liz)
- Data structures: hashes (Liz)
- Looping over arrays (Liz)
- More control structures: while, for, foreach (Brandon)
- Reading and writing files (Brandon)

Day 3
- Subroutines and modularization (Brandon)
- Executing external programs within Perl (Brandon)
- Review of material, exercises, homework

The objectives of this course

Write simple (and later more sophisticated) Perl programs to perform tasks, like:

- Analyze data in text form (fasta, fastq, blast output -> all text files!)

Most often:
Program 1 output -> Perl script to change format of output -> input for Program 2

The objectives of this course

Write simple (and later more sophisticated) Perl programs to perform tasks, like:

- Analyze data in text form (fasta, fastq, blast output -> all text files!)

Most often:
Program 1 output -> Perl script to change format of output -> input for Program 2

- Be able to pick up more Perl and other programming languages more easily

Why use Perl – Pros and Cons

Advantages:

- code is kind of human-readable (at least in simpler scripts)

Why use Perl – Pros and Cons

Advantages:

- code is kind of human-readable (at least in simpler scripts)
- manipulating/analyzing data in text-form (e.g. fasta, sam, GenBank etc.) is easy

Why use Perl – Pros and Cons

Advantages:

- code is kind of human-readable (at least in simpler scripts)
- manipulating/analyzing data in text-form (e.g. fasta, sam, GenBank etc.) is easy
- converting between/fixing file formats (=parsing) is easy

Why use Perl – Pros and Cons

Advantages:

- code is kind of human-readable (at least in simpler scripts)
- manipulating/analyzing data in text-form (e.g. fasta, sam, GenBank etc.) is easy
- converting between/fixing file formats (=parsing) is easy
- more powerful than shell tools like grep, sed, awk, …, but not as complicated and hard to learn as a low-level programming language such as C, C++ etc.

Why use Perl – Pros and Cons

Advantages:

- code is kind of human-readable (at least in simpler scripts)
- manipulating/analyzing data in text-form (e.g. fasta, sam, GenBank etc.) is easy
- converting between/fixing file formats (=parsing) is easy
- more powerful than shell tools like grep, sed, awk, …, but not as complicated and hard to learn as a low-level programming language such as C, C++ etc.
- easy to fit Perl programs into typical command pipelines, e.g.:

  <output from some program>|parse with Perl script| <input for some other program>

Why use Perl – Pros and Cons

Advantages:

- code is kind of human-readable (at least in simpler scripts)
- manipulating/analyzing data in text-form (e.g. fasta, sam, GenBank etc.) is easy
- converting between/fixing file formats (=parsing) is easy
- more powerful than shell tools like grep, sed, awk, …, but not as complicated and hard to learn as a low-level programming language such as C, C++ etc.
- easy to fit Perl programs into typical command pipelines, e.g.:
  <output from some program>|parse with Perl script| <input for some other program>
- your code will run platform-independent

Why use Perl – Pros and Cons

Advantages:

- code is kind of human-readable (at least in simpler scripts)
- manipulating/analyzing data in text-form (e.g. fasta, sam, GenBank etc.) is easy
- converting between/fixing file formats (=parsing) is easy
- more powerful than shell tools like grep, sed, awk, …, but not as complicated and hard to learn as a low-level programming language such as C, C++ etc.
- easy to fit Perl programs into typical command pipelines, e.g.:
  <output from some program>|parse with Perl script| <input for some other program>
- your code will run platform-independent
- if you have Linux or Mac, Perl is already installed

Why use Perl – Pros and Cons

Advantages:

- code is kind of human-readable (at least in simpler scripts)
- manipulating/analyzing data in text-form (e.g. fasta, sam, GenBank etc.) is easy
- converting between/fixing file formats (=parsing) is easy
- more powerful than shell tools like grep, sed, awk, …, but not as complicated and hard to learn as a low-level programming language such as C, C++ etc.
- easy to fit Perl programs into typical command pipelines, e.g.:
  <output from some program>|parse with Perl script| <input for some other program>
- your code will run platform-independent
- if you have Linux or Mac, Perl is already installed
- “There is more than one way to do it”

Why use Perl – Pros and Cons

Advantages:

- code is kind of human-readable (at least in simpler scripts)
- manipulating/analyzing data in text-form (e.g. fasta, sam, GenBank etc.) is easy
- converting between/fixing file formats (=parsing) is easy
- more powerful than shell tools like grep, sed, awk, …, but not as complicated and hard to learn as a low-level programming language such as C, C++ etc.
- easy to fit Perl programs into typical command pipelines, e.g.:
  <output from some program>|parse with Perl script| <input for some other program>
- your code will run platform-independent
- if you have Linux or Mac, Perl is already installed
- “There is more than one way to do it”

Drawbacks:

Why use Perl – Pros and Cons

Advantages:

- code is kind of human-readable (at least in simpler scripts)
- manipulating/analyzing data in text-form (e.g. fasta, sam, GenBank etc.) is easy
- converting between/fixing file formats (=parsing) is easy
- more powerful than shell tools like grep, sed, awk, …, but not as complicated and hard to learn as a low-level programming language such as C, C++ etc.
- easy to fit Perl programs into typical command pipelines, e.g.:
  <output from some program>|parse with Perl script| <input for some other program>
- your code will run platform-independent
- if you have Linux or Mac, Perl is already installed
- "There is more than one way to do it"

Drawbacks:

- "There is more than one way to do it"

Why use Perl – Pros and Cons

Advantages:

- code is kind of human-readable (at least in simpler scripts)
- manipulating/analyzing data in text-form (e.g. fasta, sam, GenBank etc.) is easy
- converting between/fixing file formats (=parsing) is easy
- more powerful than shell tools like grep, sed, awk, …, but not as complicated and hard to learn as a low-level programming language such as C, C++ etc.
- easy to fit Perl programs into typical command pipelines, e.g.:
  <output from some program>|parse with Perl script| <input for some other program>
- your code will run platform-independent
- if you have Linux or Mac, Perl is already installed
- “There is more than one way to do it”

Drawbacks:

- “There is more than one way to do it”
- processing files that aren’t text (images, audio files etc.) requires more advanced skills

Why use Perl – Pros and Cons

Advantages:

- code is kind of human-readable (at least in simpler scripts)
- manipulating/analyzing data in text-form (e.g. fasta, sam, GenBank etc.) is easy
- converting between/fixing file formats (=parsing) is easy
- more powerful than shell tools like grep, sed, awk, …, but not as complicated and hard to learn as a low-level programming language such as C, C++ etc.
- easy to fit Perl programs into typical command pipelines, e.g.:
  <output from some program>|parse with Perl script| <input for some other program>
- your code will run platform-independent
- if you have Linux or Mac, Perl is already installed
- "There is more than one way to do it"

Drawbacks:

- "There is more than one way to do it"
- processing files that aren't text (images, audio files etc.) requires more advanced skills
- more people are starting to use Python – but don't worry, if you know Perl, learning Python is much easier!

Perl resources

There are many, many online resources that help you learn and use Perl:

- Google your question, the top hit usually is literally the answer to your Perl problem

Perl resources

There are many, many online resources that help you learn and use Perl:

- Google your question, the top hit usually is literally the answer to your Perl problem
- http://proquest.tech.safaribooksonline.de/ (login within MPI network)
  Free (!) widely-used and definitive books on virtually any computer topic (for Perl: *Learning Perl, 7th edition, Randal L. Schwartz, Brian D. Fox, Tom Phoenix*)

Perl resources

There are many, many online resources that help you learn and use Perl:

- Google your question, the top hit usually is literally the answer to your Perl problem
- http://proquest.tech.safaribooksonline.de/ (login within MPI network)
  Free (!) widely-used and definitive books on virtually any computer topic (for Perl: *Learning Perl, 7th edition, Randal L. Schwartz, Brian D. Fox, Tom Phoenix*)
- comprehensive Perl Archive Database CPAN http://www.cpan.org/ Perl modules

Perl resources

There are many, many online resources that help you learn and use Perl:

- Google your question, the top hit usually is literally the answer to your Perl problem
- http://proquest.tech.safaribooksonline.de/ (login within MPI network)
  Free (!) widely-used and definitive books on virtually any computer topic (for Perl: *Learning Perl, 7th edition, Randal L. Schwartz, Brian D. Fox, Tom Phoenix*)
- comprehensive Perl Archive Database CPAN http://www.cpan.org/ Perl modules
- lots of great tutorials and forums for Perl:

  http://perldoc.perl.org

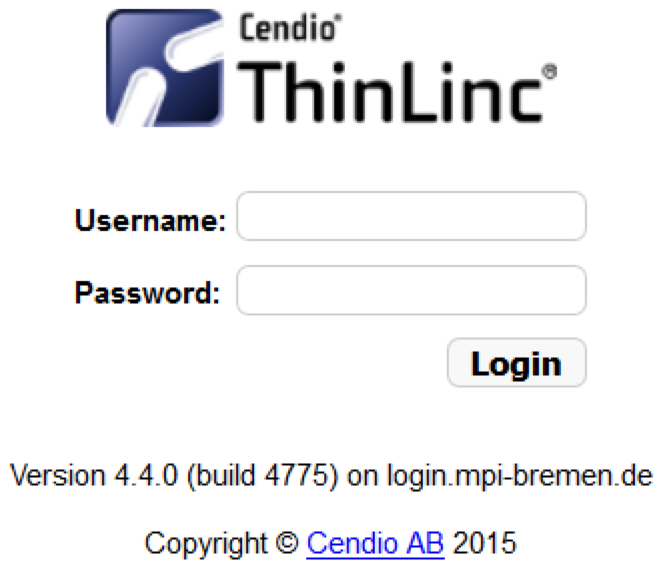
  https://perlmaven.com/perl-tutorial

  http://www.perlmonks.org

  https://www.tutorialspoint.com/perl/

Let‘s learn some Perl!

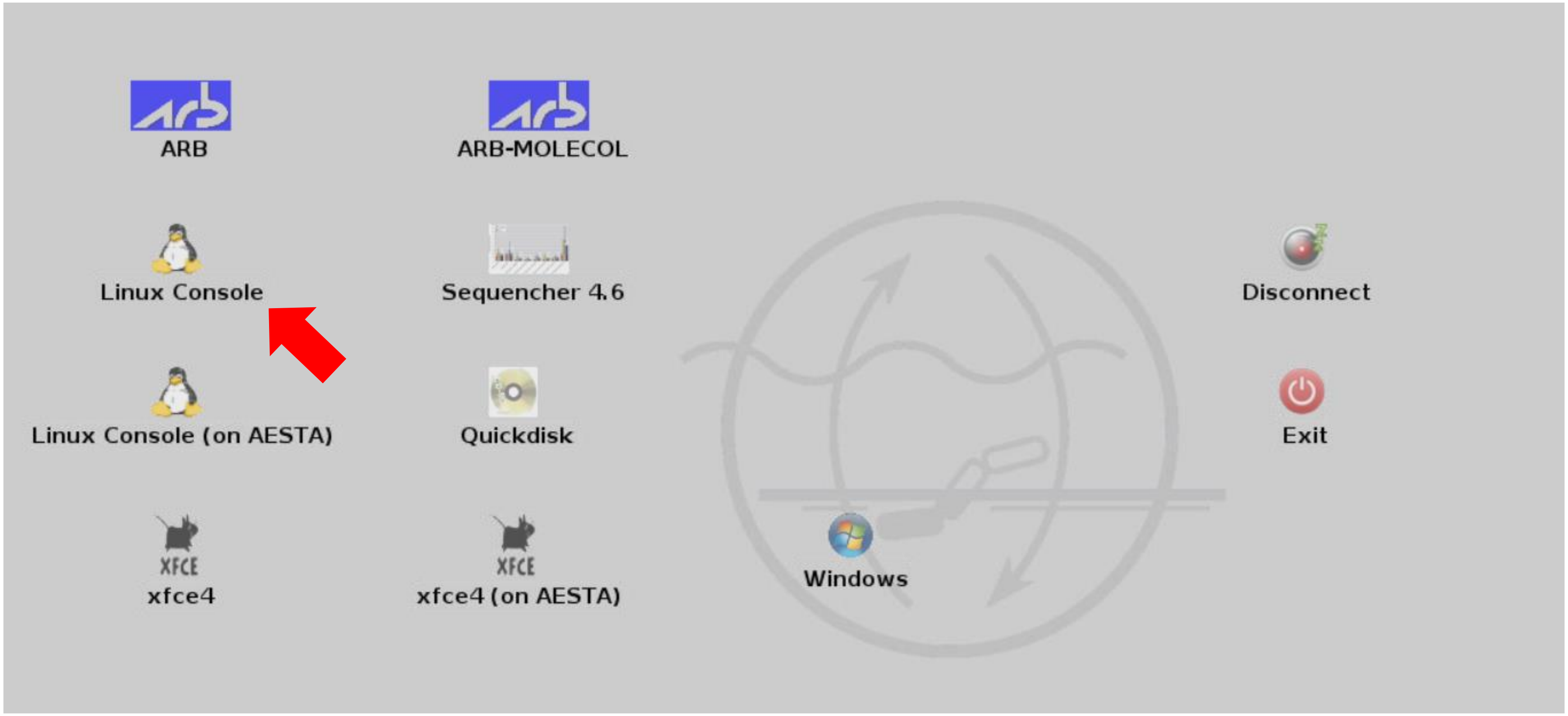
https://login.mpi-bremen.de



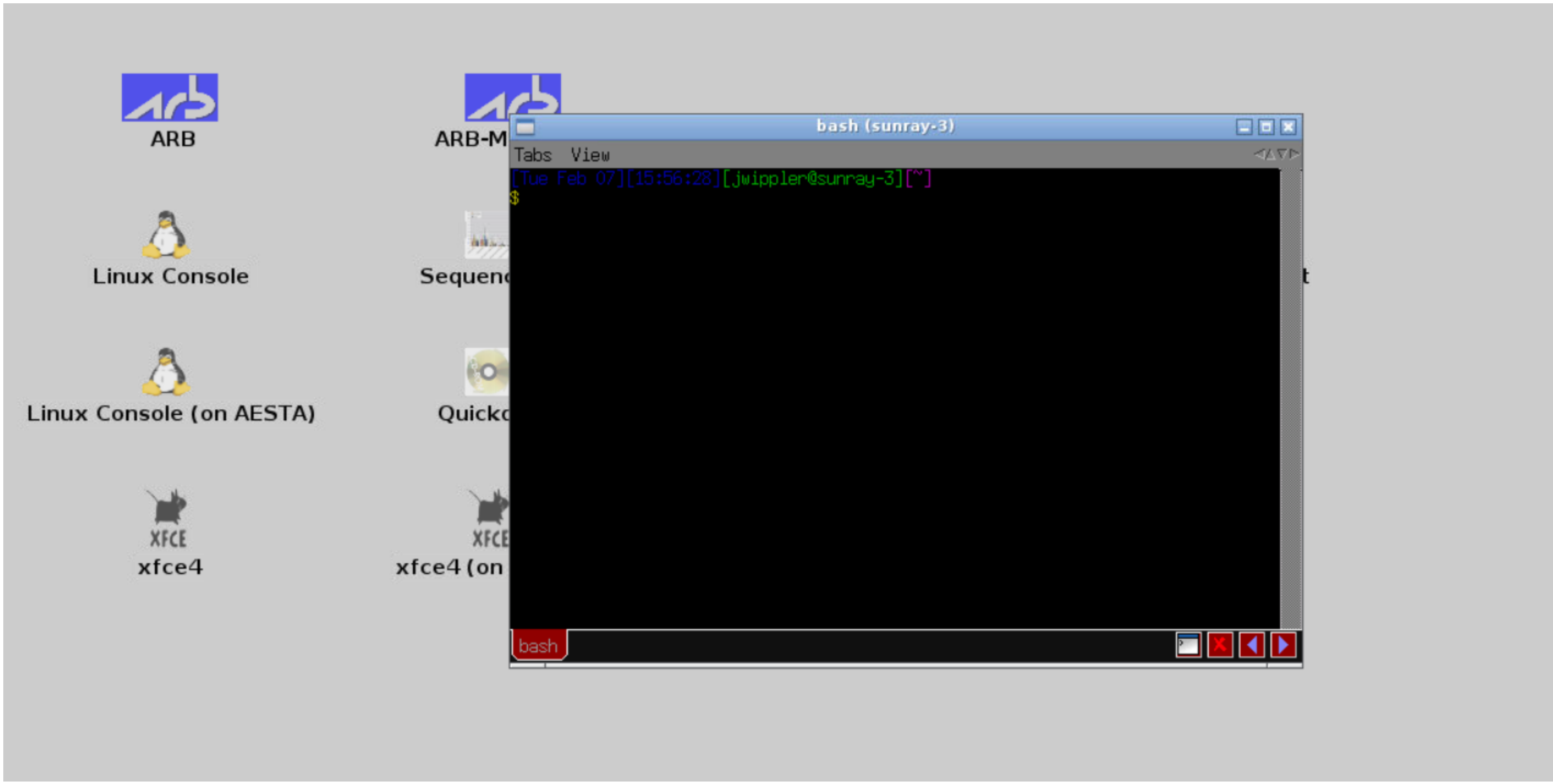
Version 4.4.0 (build 4775) on login.mpi-bremen.de

Copyright © Cendio AB 2015

log in & open console

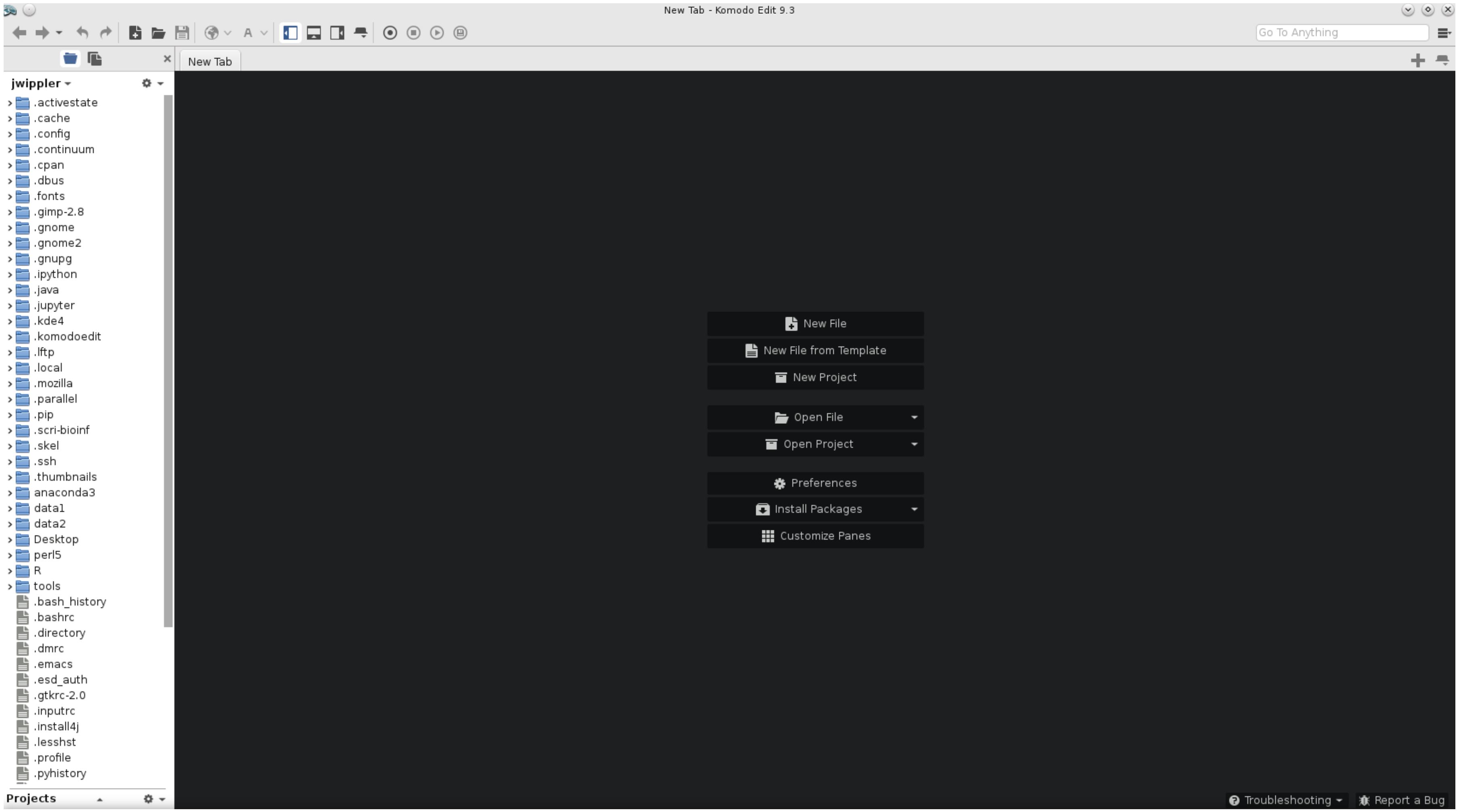


Open up a console



```
[Tue Feb 07][15:56:28][jwippler@sunray-3][~]
$
```

Open a text editor e.g. Komodo



Let’s write our very first Perl program!

Perl programs are simple text files, so open up your text editor (not word processor!) and type:

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

Let’s write our very first Perl program!

Perl programs are simple text files, so open up your text editor (not word processor!) and type:

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

- This will ALWAYS be the very first line of any code you write
- It tells the computer where to find the interpreter (perl) that actually executes your code

If you have a typo in this line, or perl is located in an unusual location, you will get the error message:

```
bad interpreter: No such file or directory
```

Say Hello! to the World

Now, let’s add some instructions:

```
#!/usr/bin/perl

print "Hello, World!\n";
```

- Save the text file (e.g. helloworld.pl)
- Make it executable:

```
chmod u+x helloworld.pl
```

- Run the program helloworld.pl:

```
./helloworld.pl
```

The print Statement

What does each element in our script do?

\n is a newline character

(special symbol that denotes the beginning of a new line)

```
#!/usr/bin/perl

print "Hello, World!\n";
```

function that let's you print stuff to screen

double quotes " " enclose the text you want to print

semicolons mark the end of each statement!

Perl Pragmas

Add these three lines to your code:

```
#!/usr/bin/perl
use strict;
use warnings;
use diagnostics;

print "Hello, World!\n";
```

Perl Pragmas

Add these three lines to your code:

```
#!/usr/bin/perl
use strict;
use warnings;
use diagnostics;

print "Hello, World!\n";
```

`use strict`	= Perl pragma to restrict unsafe constructs -> this will help you avoid “unsafe” code
`use warnings`	= Perl pragma will show optional warnings that are otherwise disabled -> this will help you de-bug your code, by giving out more info
`use diagnostics`	= gives longer description of warnings

Commenting

Comment your code! Better comment too much than too little!

Anything behind a # will become a comment and is ignored by the interpreter, e.g.:

```
#!/usr/bin/perl
# written by Juliane Wippler 2016-02-08
use strict;
use warnings;

print "Hello, World!\n"; # print “Hello, World!” to screen
```

Small Exercises

- Leave out the \n and run the script again. What changed and why?
- Modify the script to output this text instead:

```
Hello

World
!
```

Scalar Data & Variables

Scalar data = single data values, like numbers and character strings,
e.g.: 5, 134, 1e-10, hello, scalar

Scalar variable = variable that stores a scalar value

The scalar – a popular pet fish

Scalar Variables

Variable = “container” that holds one or more values

Scalar variable: holds exactly one value

Non-scalar variables (arrays and hashes): can hold many values

The name of a variable is permanent

The value of a variable can change indefinitely:

```
$variable1 = 3;
$variable1 = 5;
```

Notation of a Scalar Variable

This is a scalar variable: $name

dollar sign
(sigil)

denotes a
scalar variable

variable name =
Perl identifier

Naming Scalar Variables

<u>Mandatory rules for variable naming:</u>

- may consist of alphanumeric characters and underscores
- can‘t start with a number (you will later see why)
- don’t start with underscore (you will later see why)

`$sequence`	valid
`$seq_ID`	valid
`$sample3`	valid
`$3rd_sample`	not valid
`$_`	bad idea!

Naming Scalar Variables

<u>Recommendations for variable naming:</u>

- names should be descriptive and meaningful: `$r` and `$var1` are BAD names
- names shouldn't be endlessly long
- Avoid ALLCAPS (these can have special meaning, like $ARGV)
- Choose either underscores OR CamelCase, be consistent in style:

```
$fasta_sequence      $fastaSequence
$seq_id              $seqID
```

Initializing Variables

The **first time** you use a variable, it should be declared using “my”:

```
my $sequence;
my $GC_content;
```

Assign Values to Variables

Perl assignment operator is the equals sign:

```
my $sequence = "ATCGATGG";
my $Seq_ID = "contig_1";
my $GC_content = 54;
```

Assign Values to Variables

Perl assignment operator is the equals sign:

```
my $sequence = "ATCGATGG";
my $Seq_ID = "contig_1";
my $GC_content = 54;
```

Don‘t forget the semicolon to mark the end of a statement!

Scalar Data: Numbers

Numbers can be specified as:

integers (1, 2, 3, -5024) or

floating-point numbers = decimal numbers (1.35, 1.00, 7.5e4, -6.5e57, 1E-10)

Perl internally treats everything as double-precision floating-point values
(precision up to the 16th decimal)

Numeric Operators

Basic numeric operators work exactly as you would expect:

`2 + 2;`	addition
`7.5 – 3;`	subtraction
`5 * 4;`	multiplication
`9 / 3;`	division

Auto-increment/Auto-decrement

Perl can automatically increase/decrease the value of a variable by 1.

This is extremely useful, e.g. for variables that keep count of something (flags)!

Autoincrement operator:

```
my $count = 3;
$count++;
print "The value of \$count is $count\n";
```

Will print: The value of $count is 4

Auto-increment/Auto-decrement

Perl can automatically increase/decrease the value of a variable by 1.

This is extremely useful, e.g. for variables that keep count of something (flags)!

Autoincrement operator:

```
my $count = 3;
$count++;
print "The value of \$count is $count\n";
```

Will print: The value of $count is 4

the value of the variable is **interpolated** by Perl

Auto-increment/Auto-decrement

Perl can automatically increase/decrease the value of a variable by 1.

This is extremely useful, e.g. for variables that keep count of something

Autoincrement operator:

```
my $count = 3;
$count++;
print "The value of \$count is $count\n";
```

the backslash escape allows literal printing of $

Will print: The value of $count is 4

Auto-increment/Auto-decrement

Autodecrement operator:

```
my $count = 3;
$count--;
print "The value of \$count is $count\n";
```

Will print: The value of $count is 2

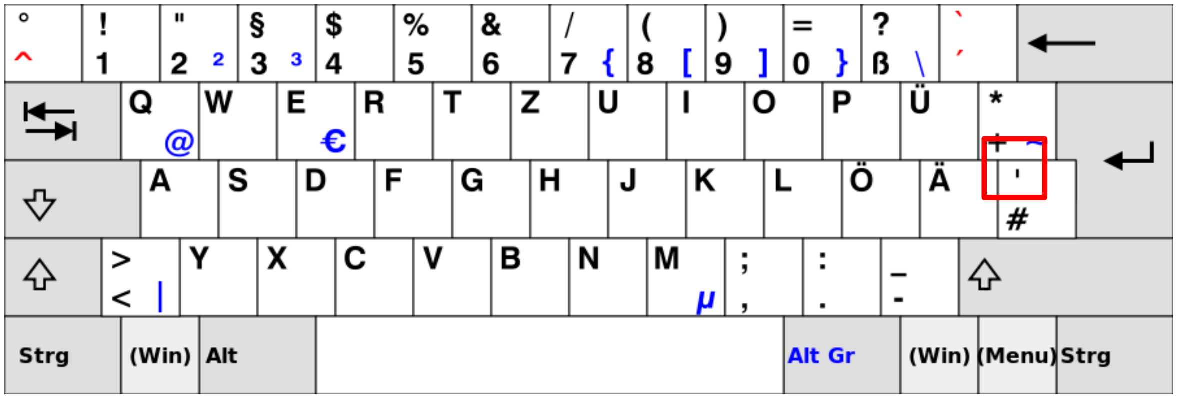
Scalar Data: Strings

Strings = sequences of characters ("hello", "R2D2", "I like trains!", "ACTGGTAAGG")

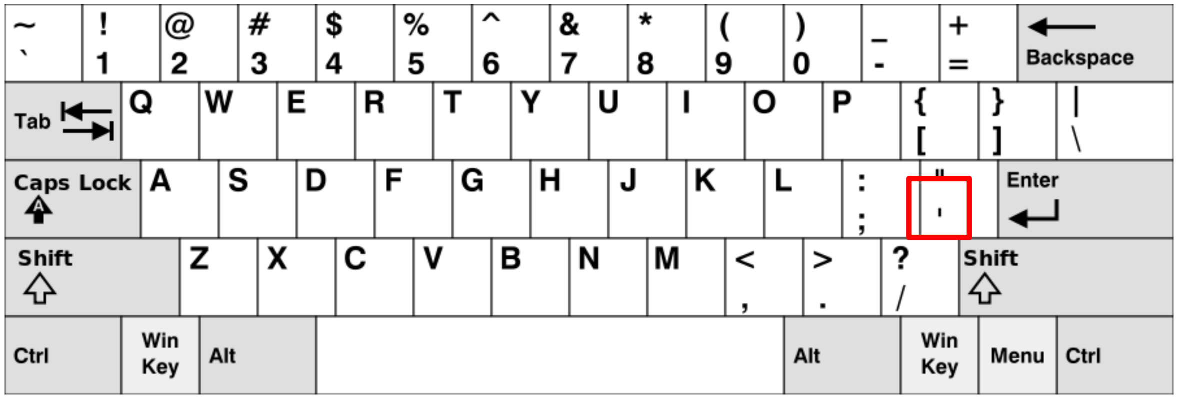
- Characters can be letters, digits, punctuation, whitespaces
- Can be any combination of any characters and of any length
- The shortest string has zero characters (empty string, null string)
- The longest string fills all of your available memory!

Single Quotes vs. Double Quotes

Single-quoted string literal:	'TCGGGTAATCGATTGCA'
Double-quoted string (interpolated)	"TCGGGTAATCGATTGCA"



German keyboard



US keyboard

Single Quotes vs. Double Quotes

Single-quoted string literal: $\texttt{'}\underbrace{\texttt{TCGGGTAATCGATTGCA}}_{\text{string}}\texttt{'}$

Null string: `''`

Single Quotes vs. Double Quotes

Single-quoted string literal: $\texttt{'}\underbrace{\texttt{TCGGGTAATCGATTGCA}}_{\text{string}}\texttt{'}$

Null string: `''`

- If `'` and \ need to be part of a string, they need to be **escaped**
- Escaping is done by **adding a backslash** \ in front:

Single Quotes vs. Double Quotes

Single-quoted string literal: $'\underbrace{\texttt{TCGGGTAATCGATTGCA}}_{\text{string}}'$

Null string: `''`

- If `'` and \ need to be part of a string, they need to be **escaped**
- Escaping is done by **adding a backslash** \ in front:

```
print 'Don\'t  forget to escape the apostrophe with a backslash: \\';
```

will print: Don't forget to escape the apostrophe with a backslash: \

```
'\'\\'
```

will print: ' \

Single Quotes vs. Double Quotes

Add the following line to your Perl script:

```
print 'Don\'t  forget to escape the apostrophe with a backslash: \\\n‘;
print "Don\'t  forget to escape the apostrophe with a backslash: \\\n“;
```

- single quotes in first line
- double quotes in second line

- Run the script
- Compare the output from the two print statements
- What’s the difference?

Single Quotes vs. Double Quotes

Add the following line to your Perl script:

```
print 'Don\'t  forget to escape the apostrophe with a backslash: \\\n'
print "Don\'t  forget to escape the apostrophe with a backslash: \\\"
```

- single quotes in first line
- double quotes in second line

- Run the script
- Compare the output from the two print statements
- What do you notice?

Backslash escapes like `\n` are ignored in single quotes!

Single Quotes vs. Double Quotes

Double quotes "" allow us to use special backslash escape characters:

Construct	**Meaning**
\n	Newline
\r	Return
\t	Tab
\f	Formfeed
\b	Backspace
\a	Bell
\e	Escape (ASCII escape character)
\007	Any octal ASCII value (here, 007 = bell)
\x7f	Any two-digit, hex ASCII value (here, 7f = delete)
\cC	A “control” character (here, Ctrl-C)
\\\\	Backslash
\\\\”	Double quote
\l	Lowercase next letter
\L	Lowercase all following letters until \E
\u	Uppercase next letter
\U	Uppercase all following letters until \E
\Q	Quote non-word characters by adding a backslash until \E
\E	End \L, \U, or \Q

String Operators

Concatenation of strings using the . operator

`"ATCG" . "CCCG"`	is the same as	`"ATCGCCCG"`
`"ATCG" . ' ' . "CCCG"`	is the same as	`"ATCG CCCG"`
`"ATCG" . "\n"`	is the same as	`"ATCG\n“`

String Operators

Repetition of strings using the x operator

`"ATCG" x 3`	is the same as	`"ATCGATCGATCG"`
`"ATCG" x (4+1)`	is the same as	`"ATCGATCGATCGATCGATCG"`
`5 x 4`	is the same as	`"5555"`
`4 x 5`	is the same as	`"44444"`

Numbers vs. Strings

Perl **automatically converts** between numbers and strings, depending on the operator:

`5 x 4`	is the same as	`"5555"`
`4 * 5`	is the same as	`20`
`"Z" . 4 * 5`	is the same as	`"Z20"`

Operators also work on Variables

```
my $GC = 36;
my $AT = 100 - $GC;
print "$AT\n";
```
will print: `64`

```
$sequence = $sequence . "GGGGTTTT";
print "$sequence\n";
```
will print: `ATCGATGGGGGGTTTT`

Compound Assignment Operators

Compound assignment operators allow you to do this more concisely:

```
$sequence = $sequence . "GGGGTTTT";
```

is the same as:

```
$sequence .= "GGGGTTTT";
```

The same can be done with other operators, e.g.:

```
$number = $number + 1;        is the same as       $number += 1;

$number = $number * 2;        is the same as       $number *= 2;
```

Printing Variables

Sometimes you want to print something directly following a variable.
In that case, you can tell Perl explicitly where the variable name starts and ends:

```
print "${sequence}GGGGCTC\n";
```
will print: `ATCGATGGGGGGTTTT`

```
print "The GC content is ${GC}%\n";
```
will print: The GC content is 64%

Operator Precedence

Perl follows the common mathematical order: first multiplication, then addition

```
5 + 4 * 3
```

= 17

Parentheses have the highest precedence:

```
(5 + 4) * 3
```

= 60

Precedence of string operators is documented here:
http://perldoc.perl.org/perlop.html#Operator-Precedence-and-Associativity

Operator Associativity

If operators with the same precedence level are resolved by associativity:

`*` and `/` have left associativity:
`36 / 6 * 3` is the same as `(36 / 6) * 3`

`**` (exponentiation) has right associativity:
`4 ** 3 ** 2` is the same as `4 ** (3 ** 2)`

If you really need it, look up associativity here:
http://perldoc.perl.org/perlop.html#Operator-Precedence-and-Associativity

Comparison Operators

Numbers and strings can be compared using comparison operators:

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`

Comparison Operators

Comparison operators return a **TRUE** or **FALSE** value:

`35 != 34;`	is TRUE
`35 != 30 + 5;`	is FALSE
`35 == 35.0;`	is TRUE
`'35' eq '35.0';`	is FALSE
`'ATCTCG' ne 'ACCCCG';`	is TRUE

Boolean Values

Comparison Operators

Comparison operators return a **TRUE** or **FALSE** value:

`35 != 34;`	is TRUE
`35 != 30 + 5;`	is FALSE
`35 == 35.0;`	is TRUE
`'35' eq '35.0';`	is FALSE
`'ATCTCG' ne 'ACCCCG';`	is TRUE

Boolean Values

note string context!

The if Clause

Perl can make decisions based on, e.g. the outcome of comparing two values:

```
if (condition) {
       do something;
}
```

The if Clause

Perl can make decisions based on, e.g. the outcome of comparing two values:

```
if (condition) {
      do something;
}
```

Perl will execute an if statement, if the condition is **TRUE**

```
my $sequence_1 = "ATCG";
my $sequence_2 = "ATCG";
if ($sequence_1 eq $sequence_2) {
      print "Sequences are identical\n";
}
```

Perl Style Conventions: Indentation

```
if ($sequence_1 eq $sequence_2) {
      print “Sequences are identical\n”;
}
```

close braces on a separate line
&
align vertically with the
beginning of the block

indent the contents of the block

The else Clause

```
if (condition) {
       do something;
} else {
       do something else;
}
```

The `else` keyword let’s you define what happens if the condition isn’t met or is FALSE

The else Clause

```
my $sequence_1 = "ATCG";
my $sequence_2 = "ATCg";

if ($sequence_1 eq $sequence_2) {
      print "Sequences are identical\n";
} else {
      print "Sequences are not identical\n";
}
```

The `else` keyword let’s you define what happens if the condition isn’t met or is FALSE

Boolean Values in Variables

Boolean values can also be stored in a scalar variable:

```
my $compare1 = "ATCG" eq "ATCG";
my $compare2 = "ATCG" eq "AAAA";
```

Do something if Boolean value = TRUE:

```
if ($compare1) {
      print "Sequences are identical\n";
}
```

will print: Sequences are identical

Boolean Values in Variables

Boolean values can also be stored in a scalar variable:

```
my $compare1 = "ATCG" eq "ATCG";
my $compare2 = "ATCG" eq "AAAA";
```

Do something if Boolean value = TRUE:

```
if ($compare1) {
      print "Sequences are identical\n";
}
```

will print: Sequences are identical

Do Something if Boolean value = FALSE:

```
if (! $compare2) {
      print "Sequences don’t match\n";
}
```

will print: Sequences don’t match

Logical Operators

Perl has logical operators to work with Boolean TRUE/FALSE values:

```
&& = AND
|| = OR
```

Logical Operators

Perl has logical operators to work with Boolean TRUE/FALSE values:

```
&& = AND
|| = OR
```

```
if (condition_1 && condition_2) {
      print "Both conditions are true! ";
} elsif (condition_1 || condition_2) {
      print "At least one or the other condition is
true";
}
```

Logical Operators

Perl has logical operators to work with Boolean TRUE/FALSE values:

```
&& = AND
|| = OR
```

Example:

```
if ( ($GC >= 40) && ($GC <=60) ) {
      print "GC content is between 40% and 60%\n";
}
```

The elsif Clause

This can be used to check a number of conditional expressions one by one:

```
if (condition 1) {
       do something;
} elsif (condition 2){
       do something else;
} elsif (condition 3){
       do another thing;
} else {
       do when all other conditions fail;
}
```

Variable Scope

my $var is valid within the enclosing block

If it‘s not enclosed in a block it is valid throughout the code

You can put your code into a “naked block” to limit the scope of the variable:

```
{
my $variable = 5;
print "$variable\n";
}
```

Regular Expressions

Regular expressions (**RegEx**) let us write patterns to match strings, so we can do things like:

- Match each line that begins with an A and end with a T
- Check if there is DNA sequence with non-standard characters
- Check if I'm looking at a DNA or a protein sequence
- Find any lines that contain the string "Bacillus subtilis" or "Bacillus anthracis"
- Count all lines that have the string "recA" in them

RegEx are used for pattern matching

Regular Expressions

- RegEx in Perl either match a string or they don‘t (no partial matches)

Regular Expressions

- RegEx in Perl either match a string or they don‘t (no partial matches)
- the leftmost, longest substring that satisfies the pattern is matched

Regular Expressions

- RegExes in Perl either match a string or they don‘t (no partial matches)
- the leftmost, longest substring that satisfies the pattern is matched

The easiest RegEx is one that literally matches a substring:

```
$_ = "AGGATAGGATATTA";

if (/GGA/) {
       print "It matched!\n";
}
```

Regular Expressions

- RegExes in Perl either match a string or they don‘t (no partial matches)
- the leftmost, longest substring that satisfies the pattern is matched

The easiest RegEx is one that literally matches a substring:

```
$_ = "AGGATAGGATATTA";

if (/GGA/) {
      print "It matched!\n";
}
```

What parts of the string will be matched?

Regular Expressions

- RegExes in Perl either match a string or they don‘t (no partial matches)
- the leftmost, longest substring that satisfies the pattern is matched

The easiest RegEx is one that literally matches a substring:

```
$_ = "AGGATAGGATATTA";

if (/GGA/) {
       print "It matched!\n";
}
```

What parts of the string will be matched?

```
AGGATAGGATATTA
```

We‘ll learn about
global matching later

Regular Expressions

RegEx pattern

special variable that holds input values and values for pattern matching

```
$_ = “AGGATAGGATATTA”;

if (/GGA/) {
        print “It matched!\n”;
}
```

match operator

Regular Expressions

Whitespaces matter!

`/GGA/  ≠ /G GA/`

Capitalization matters!

`/GGA/  ≠ /gga/`

Regular Expressions

The match operator // is similar to double quotes ""

Special backslash-escaped characters, like \n (newline), \t (tab), \s (whitespace) work:
`/Hello\tWorld/` will match `"Hello	World"`

Regular Expressions

The match operator // is similar to double quotes ""

Special backslash-escaped characters, like \n (newline), \t (tab), \s (whitespace) work:

`/Hello\tWorld/` will match `"Hello	World"`

Variables are interpolated:

```
my $word = "World";
/Hello\s${word}/
```

will match `"Hello World"`

Regular Expressions

Matching character types:

\w	matches any *single* character classified as a “word” character (alphanumeric or “_”)
\W	matches any non-“word” character
\s	matches any whitespace character (space, tab, newline)
\S	matches any non-whitespace character
\d	matches any digit character, equiv. to [0-9]
\D	matches any non-digit character

Regular Expressions

If you leave the match operator empty //, it will match any string!

Some RegExercises

First, write the following pattern matching program, save it as check_match.pl, and make it executable:

```
#!/usr/bin/perl
use strict;
use warnings;
use diagnostics;

while (<STDIN>) {
  chomp;
  if (/YOUR_PATTERN_HERE/) {
    print "Matched!\n";
  } else {
    print "No match :(\n";
  }
}
```

Some RegExercises

First, write the following pattern matching program, save it as check_match.pl, and make it executable:

```
#!/usr/bin/perl
use strict;
use warnings;
use diagnostics;

while (<STDIN>) {
  chomp;
  if (/YOUR_PATTERN_HERE/) {
    print "Matched!\n";
  } else {
    print "No match :(\n";
  }
}
```

This let‘s you read user input from the command line (this will be explained more later)

This will remove the invisible newline character \n

check_match.pl

You can use this script by piping strings to match into it with echo:

```
echo “blablabla"|./check_match.pl
```

Or by reading text from a file into it:

```
./check_match.pl < file.txt
```

Some RegExercises

Work with example GenBank file: example_genbank.gbk

- Modify your check_match.pl script to match the string "CDS"
- How many CDS does the GenBank file contain?
  (hint: use a count variable and autoincrement to count the number of matches)

RegEx Metacharacters

For building more sophisticated RegEx

If you need to use any of these as literals, use the backslash \ escape!

Metacharacter	**Matches**
^	beginning of string
$	end of string
.	any character except newline
*	match 0 or more times
+	match at least once
?	match 0 or 1 times; *or*: shortest match
|	alternative
( )	grouping; “storing”
[ ]	set of characters

Examples of Metacharacters in use

/^Hello/	matches "Hello, World!" but not "World, I say Hello"
/Hello$/	matches "World, I say Hello" but not “Hello, World!"
/H.llo/	matches "Hello“, "Hallo", "H3llo", and "H\sllo"
/Hel+o/	matches "Hello" and "Helo", but also "Helllllllllllo"
/Hel?o/	matches "Hello", "Helo", Heo"
/He|allo/	matches "Hello" and "Hallo"

RegEx Repetition Operators

Define how many time something should be matched:

Repetition operator	**Matches**
`a*`	zero or more *a*'s
`a+`	one or more *a*'s
`a?`	zero or one *a*'s (i.e., optional *a*)
`a{m}`	exactly *m* *a*'s
`a{m,}`	at least *m* *a*'s
`a{m,n}`	at least *m* but at most *n* *a*'s

Examples of Repetition Operators

/Hel{2}o/	matches "Hello"
/Hel{2,}o/	matches "Hello", "Helllo", "Hellllo" etc.
/Hel{1,5}o/	matches "Helo", "Hello", "Helllo", "Hellllo", "Helllllo"

RegEx Grouping

`/Hello+/` matches also “`Helloooooo`”

Grouping helps to define what exactly should be matched one or more times:

$\texttt{/(}\underbrace{\texttt{Hello}}_{\text{capture group}}\texttt{)+/}$ matches “`HelloHelloHello`”

RegEx Grouping

`/Hello+/` matches also “`Helloooooooo`”

Grouping helps to define what exactly should be matched one or more times:

$/(\underbrace{\texttt{Hello}}_{\text{capture group}})+/$ matches “`HelloHelloHello`”

The string matched by a capture group will automatically be saved in special variables:

$1
$2
$3
…

according to the position in the RegEx pattern

RegExercise

Modify your check_match.pl script and read example_genbank.gbk file into it:

- Enclose your pattern in parentheses (), e.g. `/(C)DS/`, and add the line
  `print "$1\n";`
- What is printed?

- Add another capture group, e.g. /(C)D(S)/
- print the values of each capture group

Back Referencing

Back reference (`\1`):
You can match the pattern in parentheses again, e.g. this will match any character that appears again right next to itself:

`/(.)\1/`
This matches `"Hello", "deep sea"` (= any character twice)

Back Referencing

Back reference (`\1`):
You can match the pattern in parentheses again, e.g. this will match any character that appears again right next to itself:

`/(.)\1/`
This matches "`Hello`", "`deep sea`" (= any character twice)

The back reference does not have to immediately follow:
`/(ll)ow\s.{1,3}\1/`
This matches "`Yellow Mellow`" and "`fellow swallow`"

Back Referencing

Back reference (`\1`):
You can match the pattern in parentheses again, e.g. this will match any character that appears again right next to itself:

`/(.)\1/`
This matches "`Hello`", "`deep sea`" (= any character twice)

The back reference does not have to immediately follow:
`/(ll)ow\s.{1,3}\1/`
This matches "`Yellow Mellow`" `and` "`fellow swallow`"

If you use multiple capture groups, each group gets it's back reference (\1, \2, …) similar to the $1, $2 etc. match variables

RegEx Character Classes

Character class	**Matches**
[*characters*]	any of the characters in the brackets
[\-]	hyphen character “-”
\n	newline character (others like \s, \d, \t work too)
\b	match word boundary, e.g. /word\b/ matches “word” but not “wordblub” or “blubword”
[^*something*]	any character *except* those that [*something*] denotes; that is, immediately after the leading “[”, the circumflex “^” means “not” applied to all of the rest
[a-zA-Z]	any lower case or upper case letter of the alphabet
[0-9]	any digits from zero to nine

Match Modifiers

Modifiers can be used to control the matching behavior:

Case insensitive matching: `/RegEx/i`

"." now also matches newline character: `/RegEx/s`

Allow whitespaces in the pattern: `/RegEx/x`

Match modifiers can be combined, e.g. `/RegEx/six`

Match at beginning of line: `/RegEx/A` (same as ^)

Match at end of line: `/RegEx/Z` (same as $)

The Binding Operator

So far, we matched against the string contained in Perl's special variable $_

However, we can also **match pattern on the right to the string on the left**:

```
string =~ /pattern/
```

For example:

```
if ($string =~ /[^ACTG$]+/){
      print "String is a nucleotide sequence\n";
}
```

The Automatic Match Variables

We already know two types of special Perl variables for matching:

`$_`	Default storage of strings for matching
`$1, $2`	Storage of capture group values

The Automatic Match Variables

We already know two types of special Perl variables for matching:

`$_`	Default storage of strings for matching
`$1, $2`	Storage of capture group values

However, there are more special match variables:

`$&`	Stores that part of the string that actually matched the pattern
`` $` `` (back tick)	Stores the string **before** the matched portion
`$'` (single quote)	Stores the string **after** the matched portion

Substring Manipulation

Substitution with s/// operator:

```
s/RegEx/REPLACEMENT/
```

Substring Manipulation

Substitution with s/// operator:

s/RegEx/REPLACEMENT/

```
$_ = "CDS";
s/CDS/cds/;
print "$_\n";
```

will print “cds”

- **Note that the value of the variable that holds the string will be changed!**
- Can be combined with match modifiers: /s /i /x etc.

Substring Manipulation

Substitution with s/// operator:

s/RegEx/REPLACEMENT/

```
$_ = "CDS";
s/CDS/cds/;
print "$_\n";
```
will print “cds”

- **Note that the value of the variable that holds the string will be changed!**
- Can be combined with match modifiers: /s /i /x

To replace all matches within a string use global replacement with /g
```
s/CDS/cds/g;
```

Substring Manipulation

Binding operator also works with substitutions to act on a string instead of $_:

```
$string =~ s/RegEx/REPLACEMENT/g;
```

We can use this e.g. to convert DNA to RNA:

```
my $sequence = "ATTTGACTATA";
print "DNA: $sequence\n";
$sequence =~ s/T/U/g;
print "RNA: $sequence\n";
```

Will print:

```
DNA: ATTTGACTATA
RNA: AUUUGACUAUA
```

Substring Manipulation

Binding operator also works with substitutions to act on a string instead of $_:

```
$string =~ s/RegEx/REPLACEMENT/g;
```

Substring Manipulation

Binding operator also works with substitutions to act on a string instead of $_:

```
$string =~ s/RegEx/REPLACEMENT/g;
```

Note: the variable $string is changed by this!

Homework Exercises