Lecture 2:
Programming in Perl: Introduction 1

Torgeir R. Hvidsten

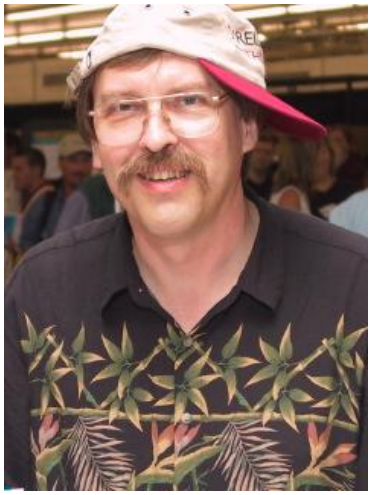
Professor
Norwegian University of Life Sciences

Guest lecturer
Umeå Plant Science Centre
Computational Life Science Cluster (CLiC)

This lecture

- Introduction to Perl 1
  - basic expressions
  - scalars
  - arrays
  - loops
  - conditions
  - file handling

What is Perl ?



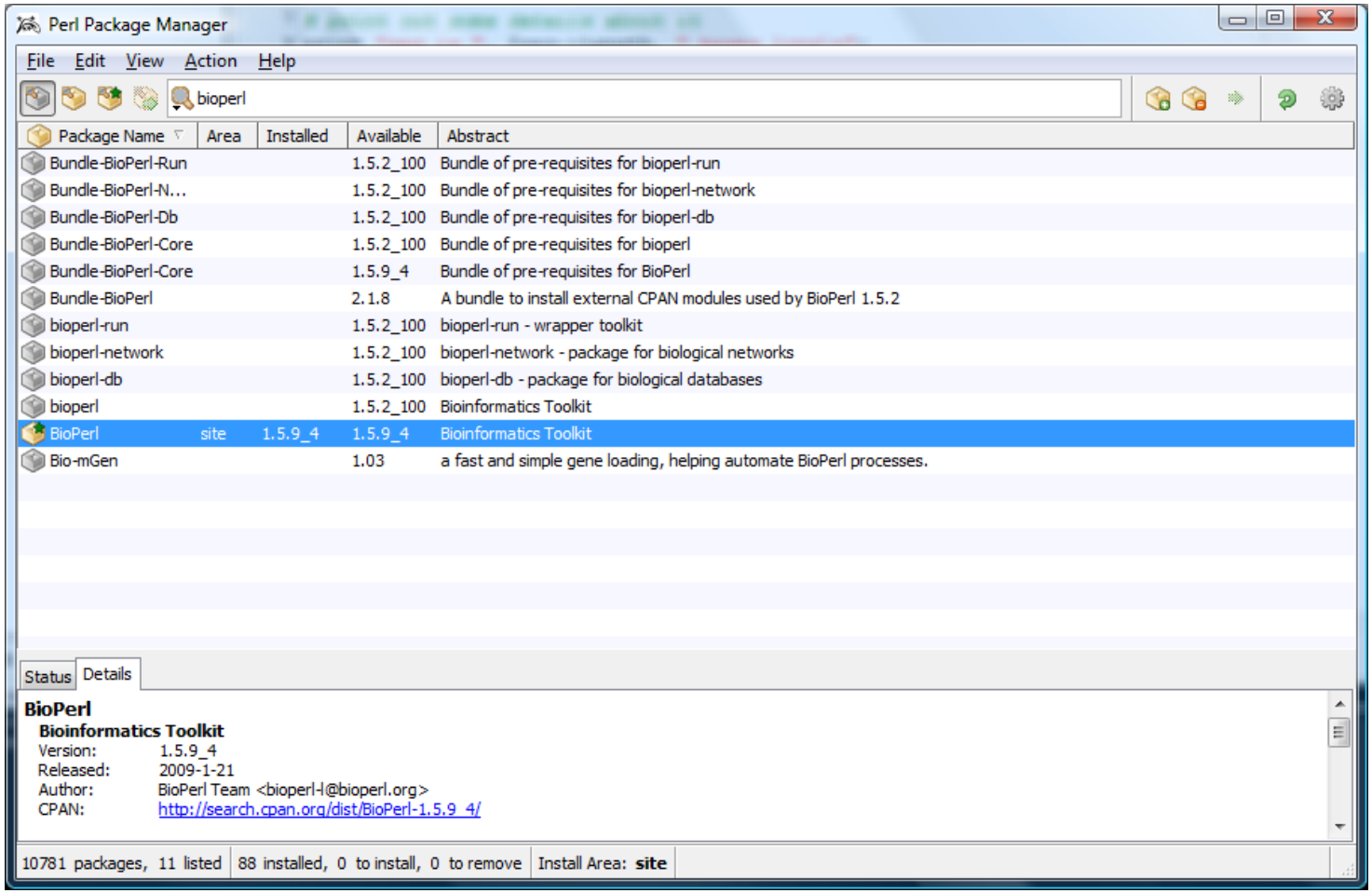
- Perl was created by Larry Wall
- Perl = Practical Extraction and Report Language
- Perl is an Open Source project
- Perl is a cross-platform programming language

Why Perl

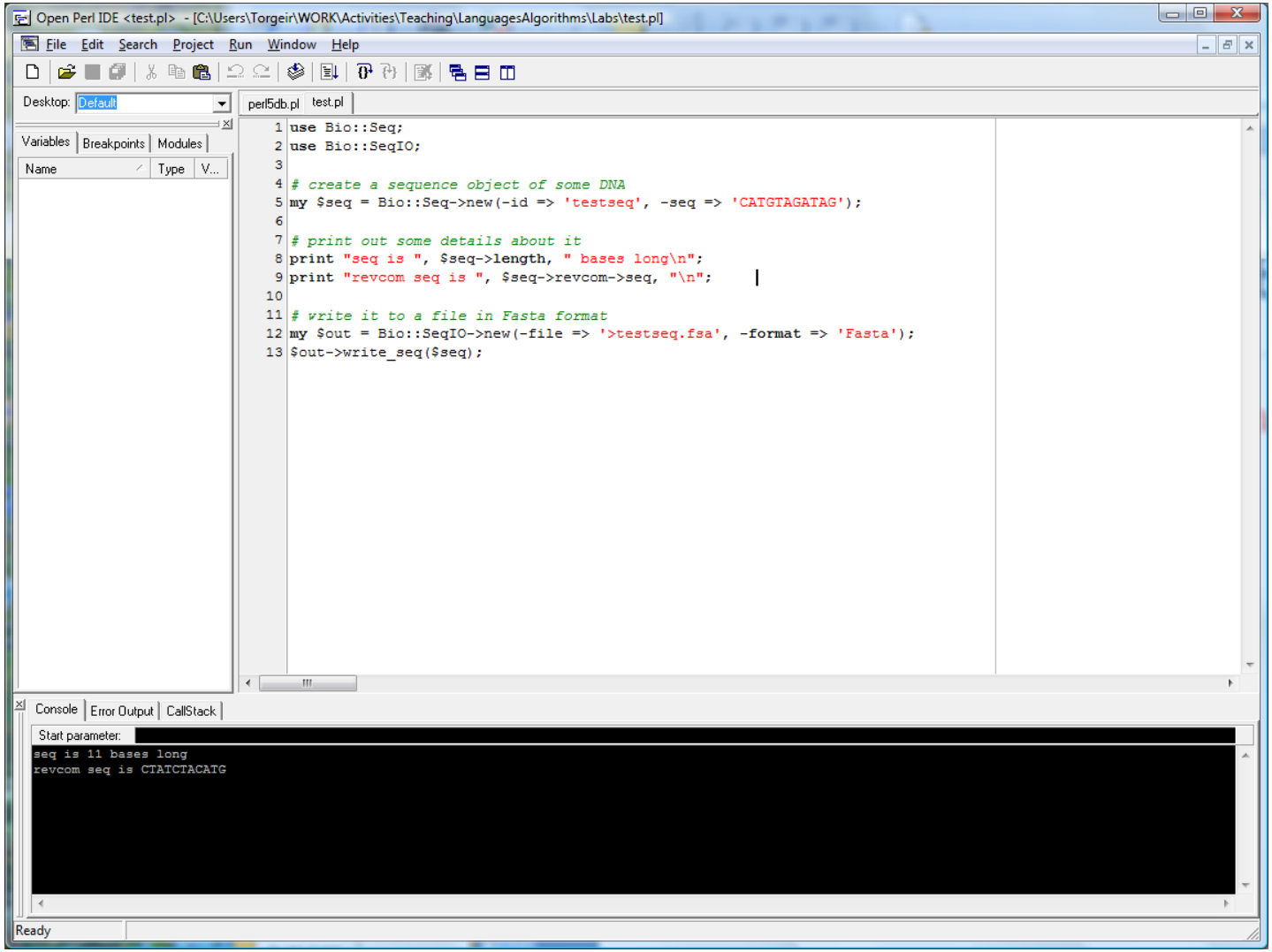
- Perl is a very popular programming language
- Perl allows a rapid development cycle
- Perl has strong text manipulation capabilities
- Perl can easily call other programs

- Existing Perl modules exists for nearly everything
  - http://www.bioperl.org
  - http://www.cpan.org/ (Comprehensive Perl Archive Network)

ActivePerl


Perl Package Manager
File Edit View Action Help
bioperl
Package Name | Area | Installed | Available | Abstract
Bundle-BioPerl-Run 1.5.2_100 Bundle of pre-requisites for bioperl-run
Bundle-BioPerl-N... 1.5.2_100 Bundle of pre-requisites for bioperl-network
Bundle-BioPerl-Db 1.5.2_100 Bundle of pre-requisites for bioperl-db
Bundle-BioPerl-Core 1.5.2_100 Bundle of pre-requisites for bioperl
Bundle-BioPerl-Core 1.5.9_4 Bundle of pre-requisites for BioPerl
Bundle-BioPerl 2.1.8 A bundle to install external CPAN modules used by BioPerl 1.5.2
bioperl-run 1.5.2_100 bioperl-run - wrapper toolkit
bioperl-network 1.5.2_100 bioperl-network - package for biological networks
bioperl-db 1.5.2_100 bioperl-db - package for biological databases
bioperl 1.5.2_100 Bioinformatics Toolkit
BioPerl site 1.5.9_4 1.5.9_4 Bioinformatics Toolkit
Bio-mGen 1.03 a fast and simple gene loading, helping automate BioPerl processes.
Status Details
BioPerl
Bioinformatics Toolkit
Version: 1.5.9_4
Released: 2009-1-21
Author: BioPerl Team <bioperl-l@bioperl.org>
CPAN: http://search.cpan.org/dist/BioPerl-1.5.9_4/
10781 packages, 11 listed 88 installed, 0 to install, 0 to remove Install Area: site

Open Perl IDE



Our first Perl program

```
use strict;
use warnings;

print "Hello world!\n";

------------------------------

Hello world!
```

“use strict" makes it harder to write bad software

”use warnings” makes Perl complain at a huge variety of things that are almost always sources of bugs in your programs

”\n” prints a new line

Perl scalars

- Perl variables that hold single values are called *Scalars*.
- Scalars hold values of many different types such as *strings, characters, floats, integers,* and *references*
- Scalars are written with a leading $, like: $sum
- Scalars, as all variables, are declared with my, like my $sum
- Perl is not a typed language: scalars can be strings, numbers, etc.

- You can reassign values of different types to a scalar:

```
my $b = 42; $b = "forty-two"; print "$b\n";
forty-two
```

- Perl will convert between strings and numbers for you:

```
my $a = "42" + 8; print "$a\n";
50
my $a = "Perl" + 8; print "$a\n";
Argument "Perl" isn't numeric in addition (+) at test.pl line 4.
8
```

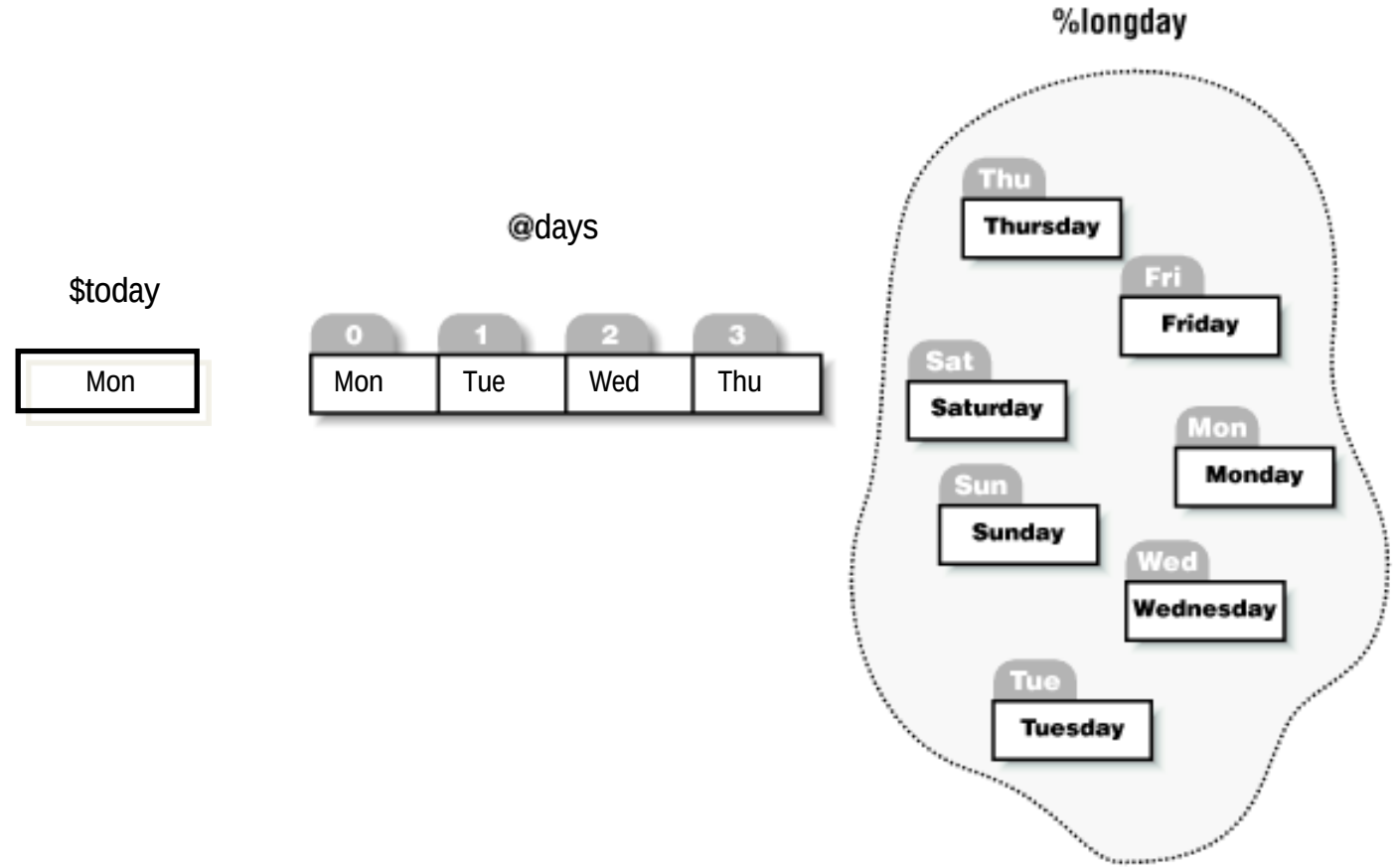
Perl scalars: some numerical operators

- $v = 1+4; # addition
- $v = 5-4; # subtraction
- $v = 3*4; # multiplication
- $v = 7/8; # division
- $v = 2**8; # power
- $v = sqrt(4); # square root

- $i++; # $i = $i + 1;
- $i--; # $i = $i - 1;
- $i+= 5; # $i = $i + 5;
- $i/=5; # $i = $i / 5;

- Everything after "#" is not executed

The three fundamental datatypes in Perl



Scalar Array Hash

- The *sigills* $,@,% must always be used.
- You can use different datatypes with the same name in the same program.

Perl Arrays

- *Arrays* hold multiple ordered values.
- Arrays are written with a leading @, like: @shopping_list
- Arrays can be initialized by lists.

  my @s = ("milk","eggs","butter"); print "@s\n";

  milk eggs butter
- Arrays are indexed by integer. The first scalar in an array has index 0 and no matter its size, the last scalar has index -1:

  my @s = ("milk","eggs","butter"); print "$s[0] - $s[-1]\n";

  milk - butter
- The sizes of arrays are not declared; they grow and shrink as necessary.

  my @s = ("milk","eggs","butter"); $s[4] = "beer"; print "@s\n";

  Use of uninitialized value in join or string at test.pl line 4.

  milk eggs butter beer

Perl Arrays

- Arrays can be iterated over in foreach loops. You don't need to know their size:

```
my @s = ("milk","eggs","butter");
foreach (@s) {
    print "$_\n";
}
```

```
milk
eggs
butter
```

$_ is known as the "default input and pattern matching variable".

This is all equivalent:

```
my @s = ("milk","eggs","butter");
foreach (@s) {
    print;
    print "\n";
}
```

```
my @s = ("milk","eggs","butter");
foreach my $item (@s) {
    print "$item\n";
}
```

```
my @s = ("milk","eggs","butter");
foreach (@s) {
    print "$_\n";
}
```

Perl Arrays

An array in scalar context evaluates to its size. You can easily get the index of the last item in an array.

```
my @s = ("milk","eggs","butter");
my $length = @s;
print "$length\n";
3

my @s = ("milk","eggs","butter");
my $last_index = $#s;
print "$last_index\n";
2

my @s = ("milk","eggs","butter");
print "$s[$#s]\n";
butter
```

Perl Arrays

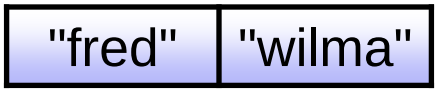
Special commands add or remove items to the front or back of arrays.
push and pop add to the back, making a stack.

```
my @s = ("milk","eggs","butter");
push @s, "beer";
print "@s\n";
milk eggs butter beer
```

```
my @s = ("milk","eggs","butter");
pop @s;
print "@s\n";
milk eggs
```

```
my @s = ("milk","eggs","butter");
my $last_item = pop @s;
print "$last_item\n";
butter
```

Perl arrays
grow or shrink as needed



@data

```
my @data = ("fred","wilma");
```

Perl arrays
grow or shrink as needed

@data

```
my @data = ("fred","wilma");
push @data, 42;
```

Perl arrays
grow or shrink as needed

@data

```
my @data = ("fred","wilma");
push @data, 42;
$data[5] = "dino";
```

undef

- The value of all uninitialized scalars (and scalar elements of arrays and hashes) has the special scalar value undef.
- undef evaluates as 0 when used as a number and "" when used as a string, which is why you most often don't have to initialize variables explicitly before you use them.

```
my $a; $a++; print "$a\n";
1
my @a = (1,2);
$a[3] = 23; print "@a\n";
Use of uninitialized value in join or string at test.pl line 4.
1 2  23
```

- Even after a scalar has been assigned, you can undefine them using the undef operator.

```
$a = undef;
undef @a;
```

Arrays and lists in assignments

@data

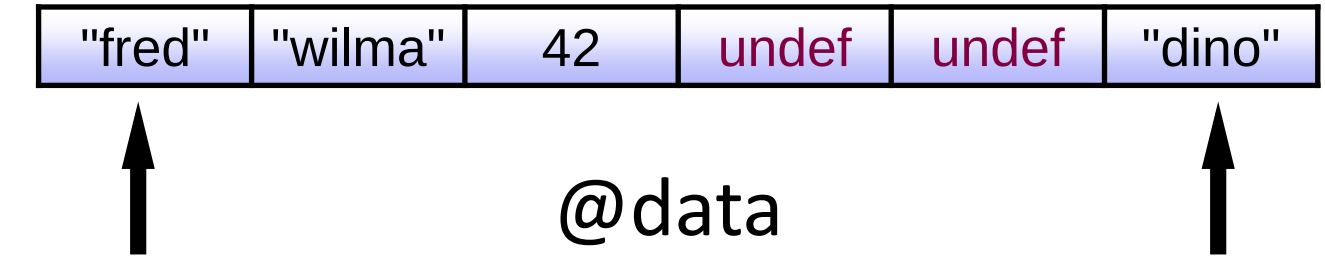
You can initialize or set arrays or lists by arrays or lists:

```
my ($man,$wmn) = ($data[0],$data[1]); print "$man $wmn\n";
fred wilma
my ($man,$wmn) = @data; print "$man $wmn\n";
fred wilma
@data = ("barney", "bambam"); print "@data\n";
barney bambam
my @mydata = @data; print "@data | @mydata\n";
barney bambam | barney bambam
```

You can swap elements without a temporary:

```
($data[1],$data[0]) = ($data[0],$data[1]); print "$data[0] $data[1]\n";
 bambam barney
```

Array indexing



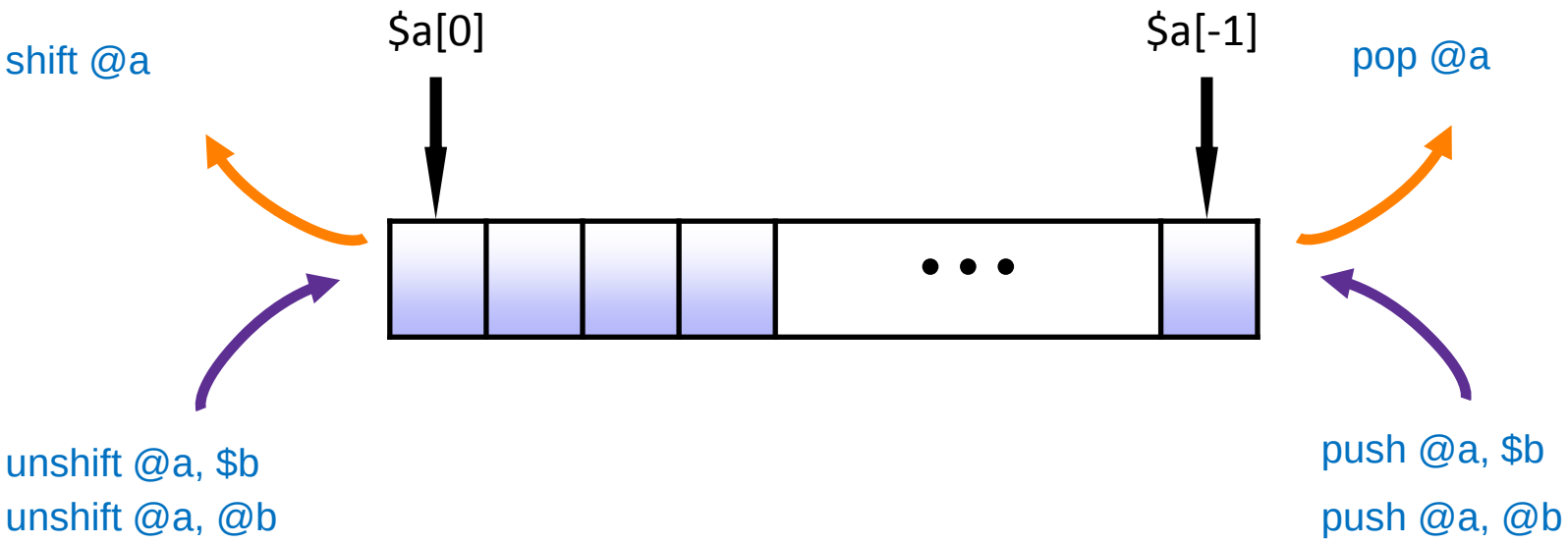
$data[0]

$data[5]

$data[$#data]

$data[-1]

Adding elements to array ends



Loops: Iterating over Arrays

When we need the index:

```
for ($i = 0; $i  < @data ; $i++) { # c-style
    print "$data[$i]\n";
}
```

When we only need the element:

```
foreach (@data) { # perl-style
    print "$_\n";
}
```

conditions

- if – else statements are used to test whether an expression is true or false

```
if ($a < 0) {
        print "$a is a negative number\n";
} elsif ($a == 0) {
        print "$a is zero\n";
} else {
        print "$a is a positive number\n";
}
```

- Use the function defined to test if a scalar has the value undef

```
if (defined $a) {
        $a++;
}
```

equivalent to

```
$a++ if defined $a;
```

The rules of truth in Perl

- Only Scalars can be True or False
- undef is False
- "" is False
- 0 is False
- 0.0 is False
- "0" is False
- Everything else is True (including "0.0" !)

Logical expression

- $a == $b # compare numbers, true if $a equal to $b
- $a != $b # compare numbers, true if $a is not equal to $b
- $a eq $b # compare strings, true if $a is equal to $b
- $a ne $b # compare strings, true if $a is not equal to $b
- !$a # boolean, true if $a is 0, false if $a is 1

Controlling loops: next and last

next skip to the next iteration

```
my @a = (1,2,5,6,7,0);

my @filtered;
foreach (@a) {
    next if $_ < 5;
    push @filtered, $_;
}
print "@filtered\n";
```

```
5 6 7
```

last ends the loop

```
my @a = (1,2,5,6,7,0);

my $found_zero = 0;
foreach (@a) {
    if ($_ == 0) {
        $found_zero = 1;
        last;
    }
}
print "$found_zero \n";
```

```
1
```

Sorting arrays

- Use the built in function sort

- The results may surprise you!

```
my @words = ("c","b","a","B");
@words = sort @words;
print "@words\n";
```

B a b c

```
my @numbers = (10,3,1,2,100);
@numbers = sort @numbers;
print "@numbers\n";
```

1 10 100 2 3

sort

- sort uses a default sorting operator cmp that sorts "ASCIIbetically", with capital letters ranking over lower-case letters, and then numbers.

  sort @words;
  is equivalent to:

  sort {$a cmp $b} @words;

- cmp is a function that returns three values:
  - -1 if $a le $b
  - 0 if $a eq $b
  - +1 if $a ge $b
- where le, eq, and ge are string comparison operators.
- $a and $b are special scalars that only have meaning inside the subroutine block argument of sort. They are aliases to the members of the list being sorted.

sort {$a <=> $b} @numbers

- <=> (the "spaceship operator") is the numerical equivalent to the cmp operator:
  - -1 if $a < $b
  - 0 if $a == $b
  - +1 if $a > $b

- You can provide your own named or anonymous comparison subroutine to sort:

```
my @numbers = (10,3,1,2,100);
@numbers = sort {$a <=> $b} @numbers;
print "@numbers\n";
1 2 3 10 100
@numbers = sort {$b <=> $a} @numbers;
print "@numbers\n";
100 10 3 2 1
```

Syntax summary: scalars

- Declare: my $age;
- Set: $age = 29; $age = “twenty-nine”;
- Access: print “$age\n”; twenty-nine

Syntax summary: arrays

- Declare: my @children;
- Set all: @children = (“Troy”,”Anea”);
- Set element: $children[0] = “Troy Alexander”;
- Access all: print “@children\n”; Troy Alexander Anea
- Access element: print “$children[1]\n”; Anea

Syntax summary: loops

```
foreach my $child (@children) {
    print "$child\n";
}
```

```
Troy Alexander
Anea
```

```
for (my $i = 0; $i < @children; $i++) {
     print "$i: $children[$i]\n";
}
```

```
0: Troy Alexander
1: Anea
```

Syntax summary: conditions

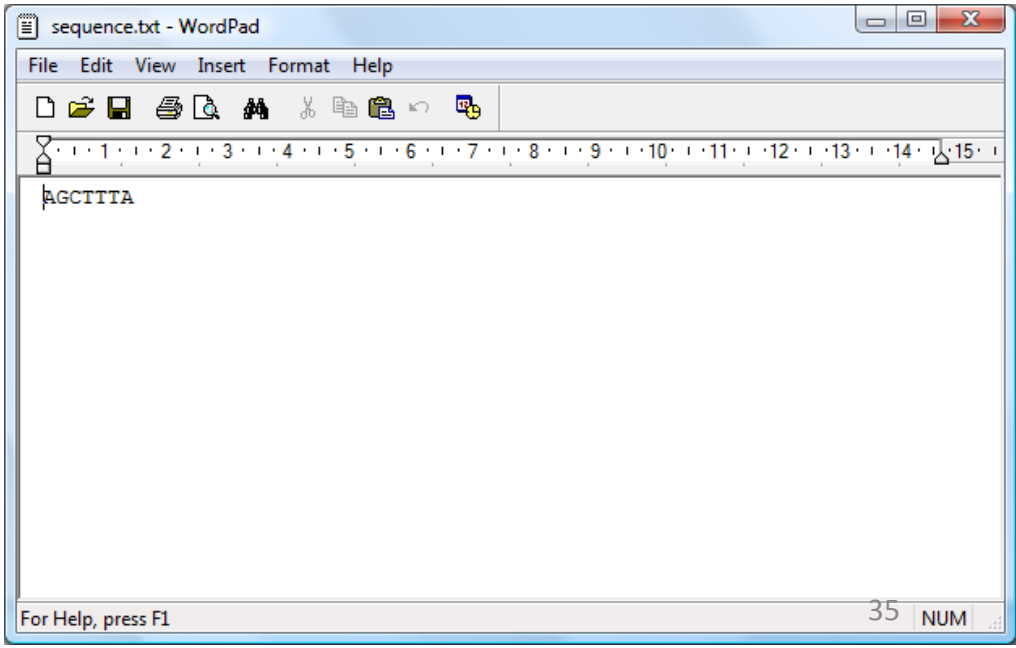
```
foreach my $child (@children) {
    if (length($child) > 4) {
        print "$child\n";
    }
}
```

Troy Alexander

Reading and writing to files

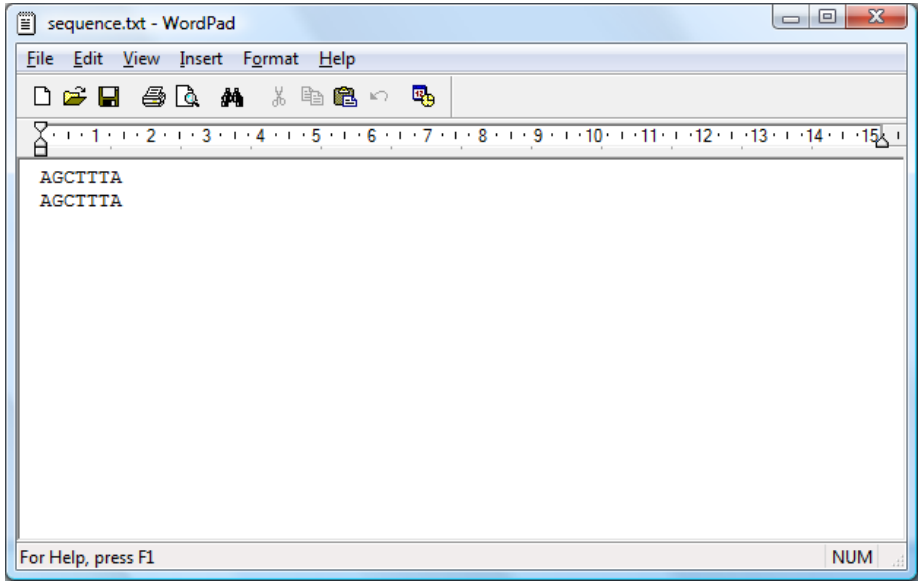
- open(A, ">sequence.txt") – creates a new file and opens it for writing
- open(A, ">>sequence.txt") – opens an existing file for writing
- open(A, "sequence.txt") – opens an allready existing file for reading

```
open(A , ">sequence.txt");
print A “AGCTTTA\n";
close(A);
```



Reading and writing to files

```
open(A , ">>sequence.txt");
print A "AGCTTTA\n";
close(A);
```




```
open(A , "sequence.txt");
my $line1 = readline *A;
my $line2 = readline *A;
close(A);
print "$line1 | $line2\n";
AGCTTTA
 | AGCTTTA
```

Reading files

```
my @seqs;
open(A , "sequence.txt");
while (<A>) {
    chomp;
    push @seqs, $_;
}
close(A);
print ”@seqs\n";
AGCTTTA AGCTTTA
```

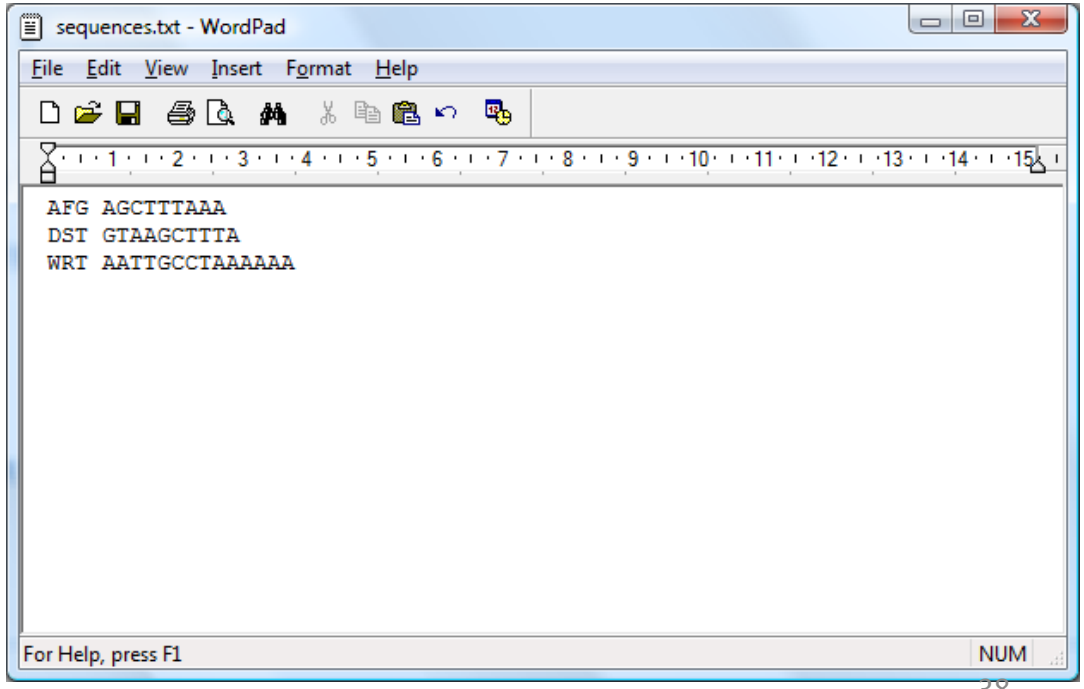
chomp removes ”\n” from the end of the line if it exists

Splitting strings: split

- You can split a string on any substrings that match a regular-expression with:
  - @array = split /PATTERN/, $string;
  - split /\s/, "do the twist"; # gives ("do","the","twist")
  - split //, "dice me"; # gives ("d","i","c","e"," ","m","e");

- Extremly useful when parsing files:

```
my @genes;
open(A , "sequences.txt");
while (<A>) {
    chomp;
    my ($gene) = split /\s/;
    push @genes, $gene;
}
close(A);
print ”@genes\n";
AFG DST WRT
```



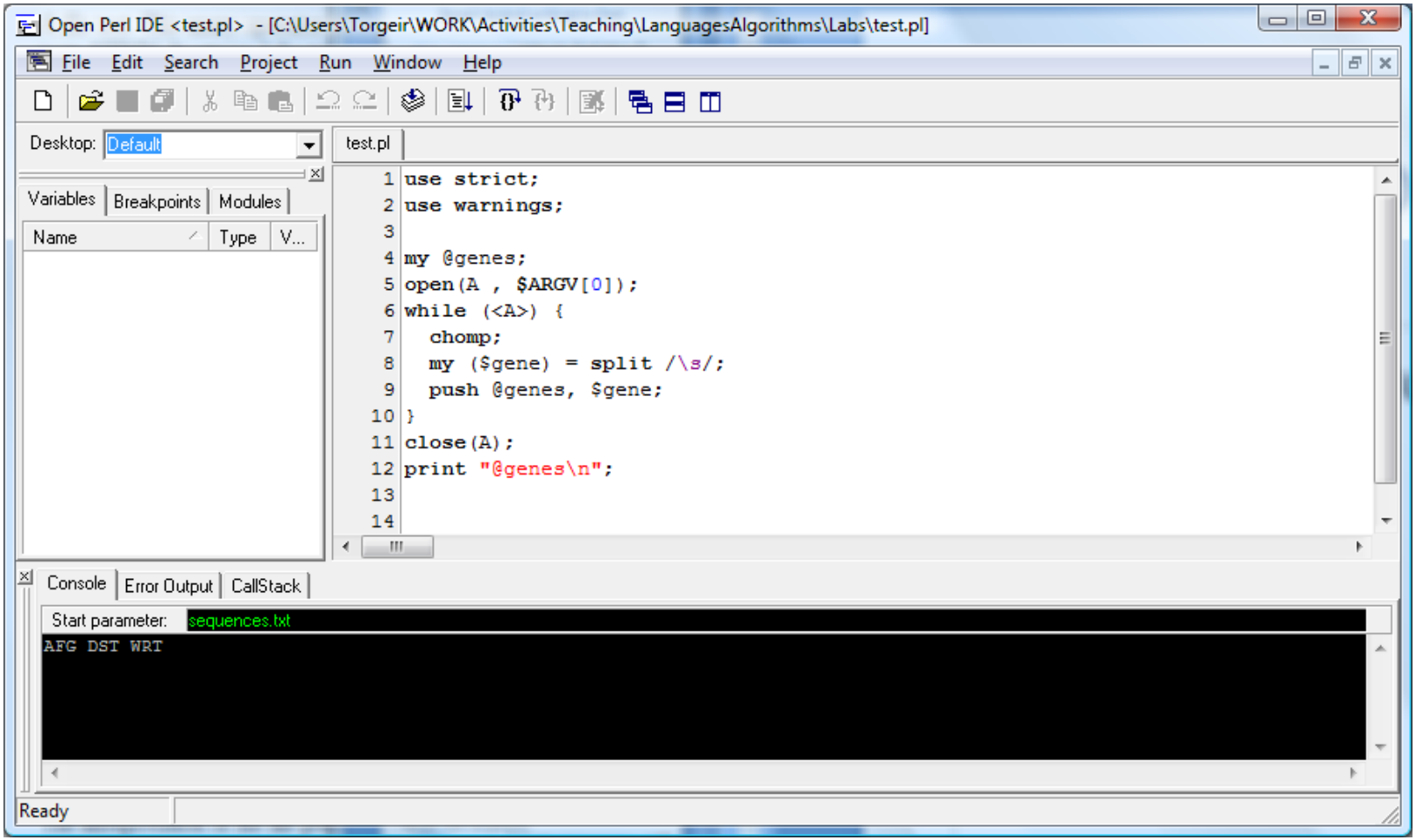
Extracting fragments: substr

```
my $string = "AC Milan";
my $fragment = substr $string, 3;
print "$fragment\n";
```
Milan

```
my $string = "F.C. Internazionale";
my $fragment = substr $string, 5, 5;
print "$fragment\n";
```
Inter

```
my $string = "F.C. Internazionale";
my $fragment = substr $string, -7, 4;
print "$fragment\n";
```
zion

@ARGV: command-line arguments



Acknowledgements

- Several slides were taken or re-worked from David Ardell and Yannick Pouliot.