



FBW

10-11-2010



*Bi*INFORMATICS

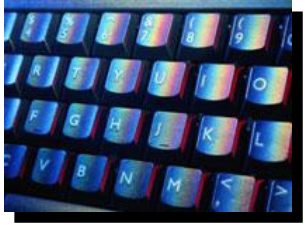


Wim Van Crieking

- Variables
- Flow control (if, regex ...)
- Loops

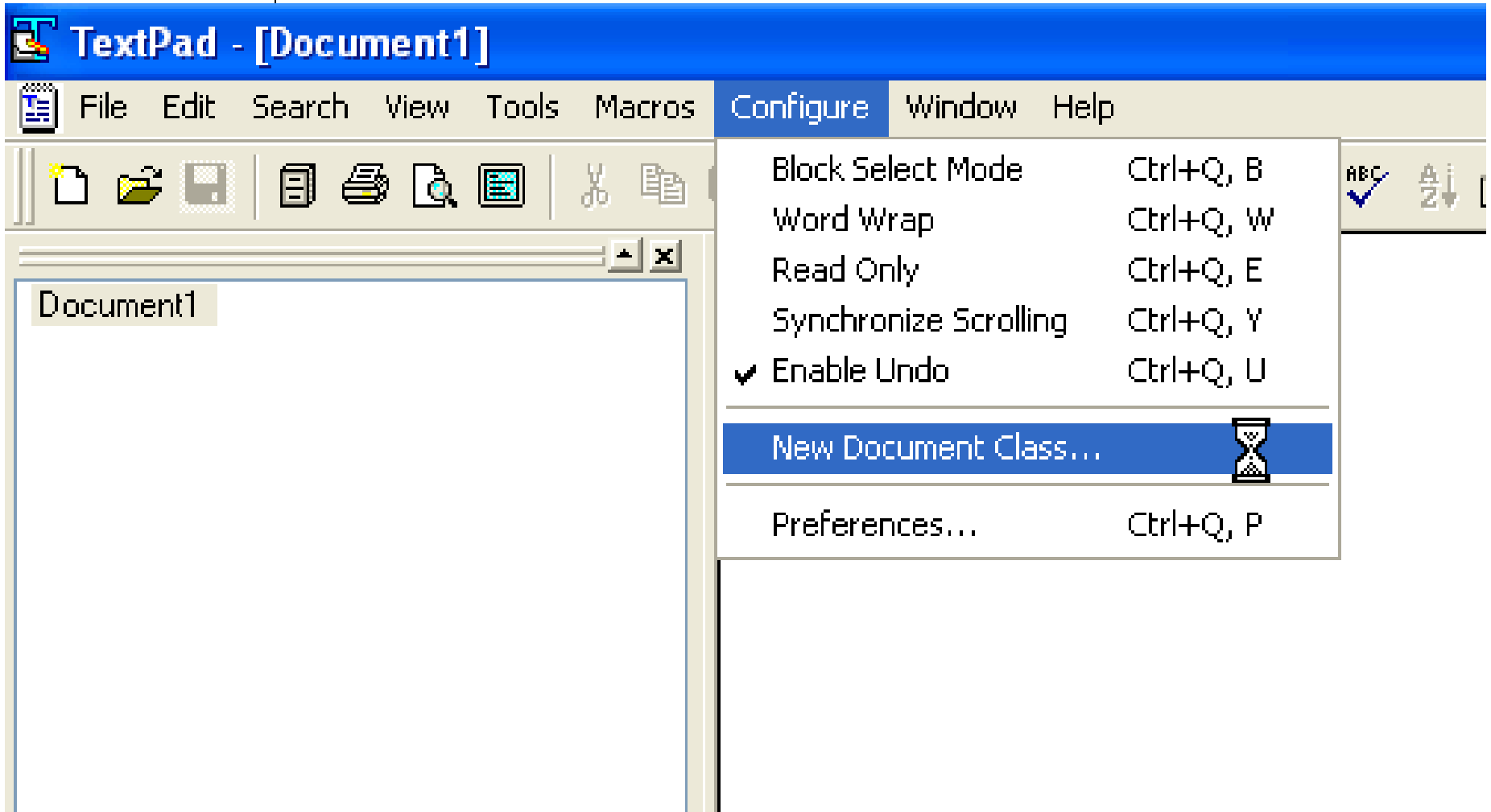
- input/output
- Subroutines/object

Three Basic Data Types



- Scalars - \$
- Arrays of scalars - @
- Associative arrays of scalars or Hashes - %

Customize textpad part 1: Create Document Class



Preferences

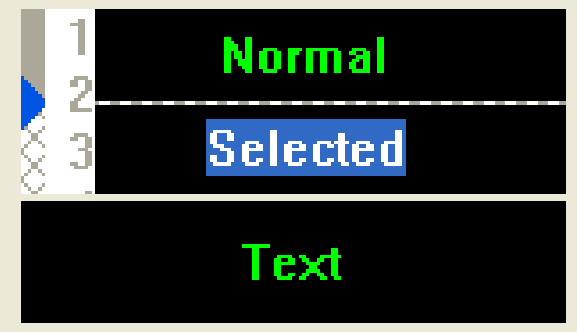


- [-] Document Classes
 - [+] Default
 - [+] Binary
 - [-] Command Rest
 - Colors**
 - Font
 - Printing
 - Syntax
 - Tabulation
 - [+] Search Results
 - [+] C/C++
 - [+] HTML
 - [+] Java
 - [+] perl
 - [+] Text
- Associated Files
- Backup
- File Name Filters

Item:

- Text**
- Selected text
- Selected text (no focus)
- Bookmarks
- Left margin
- Page breaks
- Keywords 1
- Keywords 2
- Keywords 3
- Keywords 4
- Keywords 5
- Keywords 6
- Preprocessor keywords
- Comments
- Numbers

Sample



Foreground

Background

Set Defaults

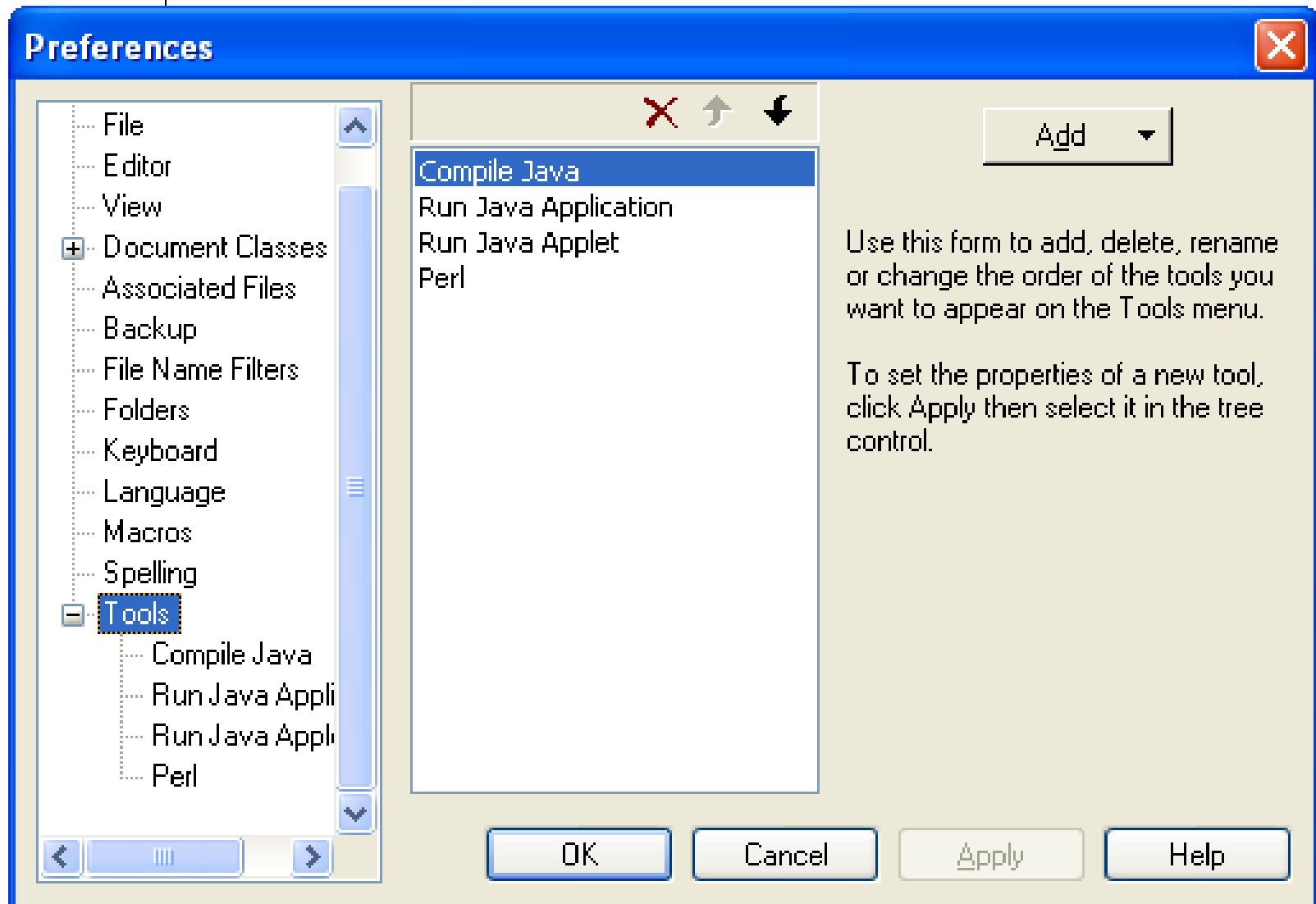
OK

Cancel

Apply

Help

Customize textpad part 2: Add Perl to “Tools Menu”



- Syntax highlighting
- Run program (prompt for parameters)
- Show line numbers
- Clip-ons for web with perl syntax
-



[What's New](#)

[Products](#)

[Support](#)

[Download](#)

[Buy](#)

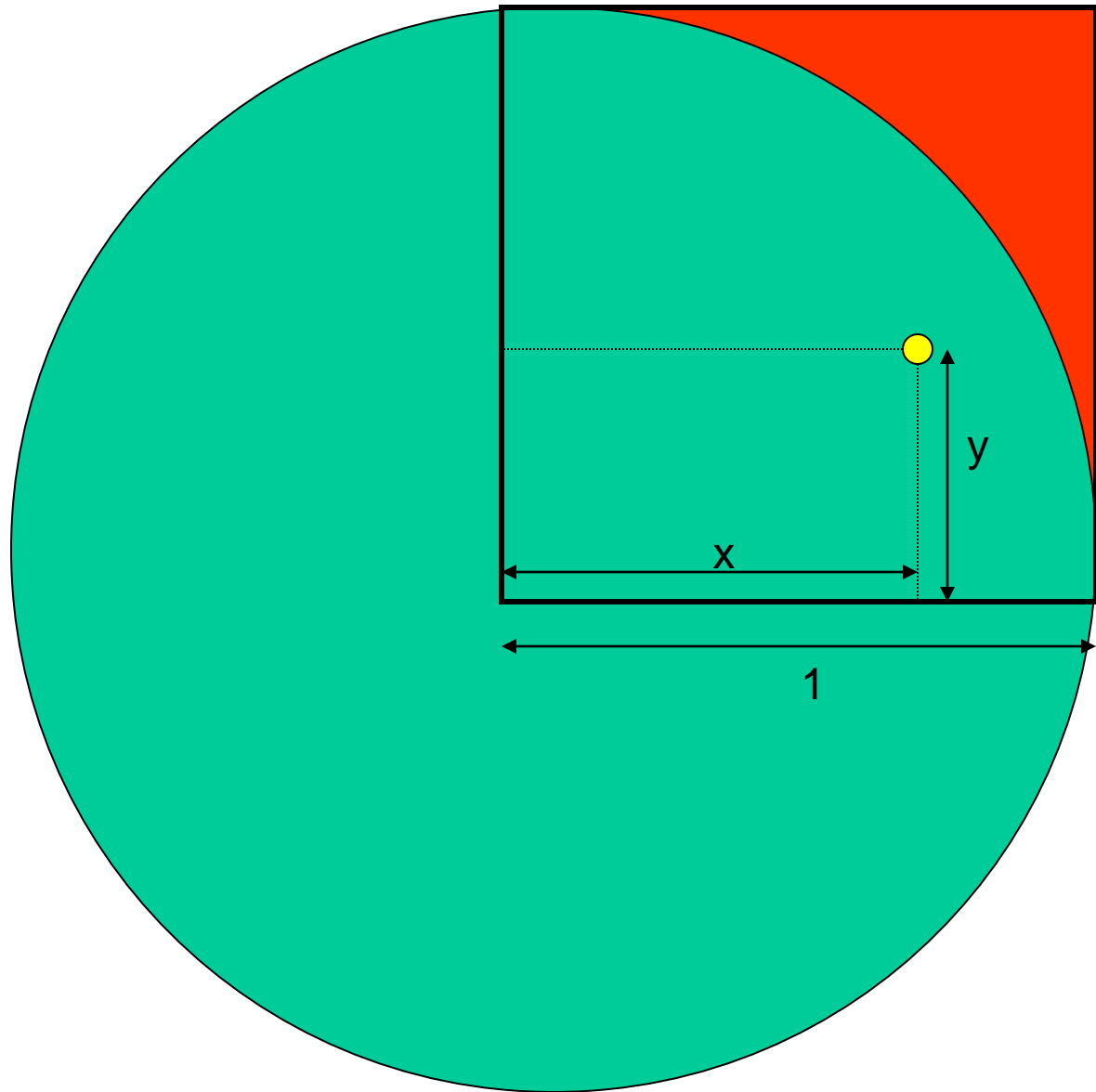
[Forums](#)

add-ons

[Utilities](#) | [Clip Libraries](#) | [Dictionaries](#) | [Macros](#) | [Syntax Definitions](#)

Unzip to textpad samples directory

Bereken Pi aan de hand van twee random getallen




```
Terminal: File Edit Shell View Window Help
7:22 PM
# /usr/local/bin/EXECVTE

adnota Ecce Cribrum Eratotherenis!

maximum tum val inquentum
tum biguttam tum stadium egresso scribe.
estibulo perlegementum da.

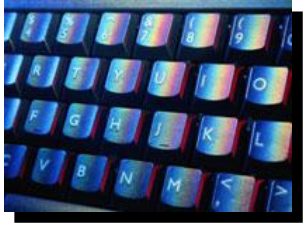
maximum comentum tum novumversum egresso scribe.
meis listis
conscribementa II tum maximum da.
dum damentum nexto listis
decapitamentum fac

sic lista sic hoc tum nextum recidementum
cis vannementa listis da.
dictum sic deinde cis tum biguttam tum
stadium tum cum nextum
mentum tum novumversum scribe egresso.
cis

7,22
```



Two brief diversions (warnings & strict)



- Use warnings;
- Use strict;
- `strict` – forces you to ‘declare’ a variable the first time you use it.
 - usage: `use strict;` (somewhere near the top of your script)
- declare variables with ‘*my*’
 - usage: `my $variable;`
 - or: `my $variable = ‘value’;`
- *my* sets the ‘scope’ of the variable. Variable exists only within the current block of code
- *use strict* and *my* both help you to debug errors, and help prevent mistakes.

Practicum Bioinformatica

- Practicum PC Zaal D
 - 8u30 – 11u30
- Recap
- Installeren en werken met TextPad
- **Regex**
- Arrays/hashtables

What is a regular expression?

- A regular expression (*regex*) is simply a way of describing text.
- Regular expressions are built up of small units (atoms) which can represent the type and number of characters in the text
- Regular expressions can be very broad (describing everything), or very narrow (describing only one pattern).

Why would you use a regex?

- Often you wish to test a string for the presence of a specific character, word, or phrase
 - Examples
 - “Are there any letter characters in my string?”
 - “Is this a valid accession number?”
 - “Does my sequence contain a start codon (ATG)?”

Regular Expressions

Match to a sequence of characters

The EcoRI restriction enzyme cuts at the consensus sequence GAATTC.

To find out whether a sequence contains a restriction site for EcoR1, write;

```
if ($sequence =~ /GAATTC/) {  
    ...  
};
```

- [m]/PATTERN/[g][i][o]
- s/PATTERN/PATTERN/[g][i][e][o]
- tr/PATTERNLIST/PATTERNLIST/[c][d][s]

Regular Expressions

Match to a character class

- **Example**
- The BstYI restriction enzyme cuts at the consensus sequence rGATCy, namely A or G in the first position, then GATC, and then T or C. To find out whether a sequence contains a restriction site for BstYI, write;
- if (**\$sequence =~ /[AG]GATC[TC]/**) {...}; # This will match all of AGATCT, GGATCT, AGATCC, GGATCC.

Definition

- When a list of characters is enclosed in square brackets [], one and only one of these characters must be present at the corresponding position of the string in order for the pattern to match. You may specify a range of characters using a hyphen -.
- A caret ^ at the front of the list negates the character class.

Examples

- if (**\$string =~ /[AGTC]/**) {...}; # matches any nucleotide
- if (**\$string =~ /[a-z]/**) {...}; # matches any lowercase letter
- if (**\$string =~ /chromosome[1-6]/**) {...}; # matches chromosome1, chromosome2 ... chromosome6
- if (**\$string =~ /^[^xyzXYZ]/**) {...}; # matches any character except x, X, y, Y, z, Z

Constructing a Regex

- Pattern starts and ends with a / */pattern/*
 - if you want to match a /, you need to escape it
 - \ (backslash, forward slash)
 - you can change the delimiter to some other character, but you probably won't need to
 - m|*pattern*|
- any 'modifiers' to the pattern go after the last /
 - *i* : case insensitive */[a-z]/i*
 - *o* : compile once
 - *g* : match in list context (global)
 - *m* or *s* : match over multiple lines

Looking for a pattern

- By default, a regular expression is applied to `$_` (the default variable)
 - if `(/a+/) {die}`
 - looks for one or more 'a' in `$_`
- If you want to look for the pattern in any other variable, you must use the *bind* operator
 - if `($value =~ /a+/) {die}`
 - looks for one or more 'a' in `$value`
- The bind operator is in no way similar to the '=' sign!! = is assignment, =~ is bind.
 - if `($value = /[a-z]/) {die}`
 - Looks for one or more 'a' in `$_`, **not** `$value`!!!

Regular Expression Atoms

- An 'atom' is the smallest unit of a regular expression.
- Character atoms
 - 0-9, a-Z match themselves
 - . (dot) matches everything
 - [atgcATGC] : A character class (group)
 - [a-z] : another character class, a through z

More atoms

- `\d` - All Digits
- `\D` - Any non-Digit
- `\s` - Any Whitespace (`\s`, `\t`, `\n`)
- `\S` - Any non-Whitespace
- `\w` - Any Word character [`a-zA-Z_0-9`]
- `\W` - Any non-Word character

An example

- if your pattern is $\wedge d\d\d-\d\d\d\d/$
 - You could match
 - 555-1212
 - 5512-12222
 - 555-5155-55
 - But not:
 - 55-1212
 - 555-121
 - 555j-5555

Quantifiers

- You can specify the number of times you want to see an atom. Examples
 - $\backslash d^*$: Zero or more times
 - $\backslash d^+$: One or more times
 - $\backslash d\{3\}$: Exactly three times
 - $\backslash d\{4,7\}$: At least four, and not more than seven
 - $\backslash d\{3,}\}$: Three or more times
 - We could rewrite $\wedge d\backslash d\backslash d-\backslash d\backslash d\backslash d/$ as:
– $\wedge d\{3\}-\backslash d\{4\}/$

Anchors

- Anchors force a pattern match to a certain location
 - `^` : start matching at beginning of string
 - `$` : start matching at end of string
 - `\b` : match at word boundary (between `\w` and `\W`)
- Example:
 - `/^\d\d\d-\d\d\d\d$/` : matches only valid phone numbers

Grouping

- You can group atoms together with parentheses
 - `/cat+/
matches cat, catt, cattt`
 - `/(cat)+/
matches cat, catcat, catcatcat`
- Use as many sets of parentheses as you need

Alternation

- You can specify patterns which match either one thing *or* another.
 - `/cat|dog/` matches either 'cat' or 'dog'
 - `/ca(t|d)og/` matches either 'catog' or 'cadog'

Variable interpolation

- You can put variables into your pattern.
 - if \$string = 'cat'
 - /\$string/ matches 'cat'
 - /\$string+/ matches 'cat', 'catcat', etc.
 - $\wedge d\{2\}$string+$ matches '12cat', '24catcat', etc.

Regular Expression Review

- A regular expression (*regex*) is a way of describing text.
- Regular expressions are built up of small units (*atoms*) which can represent the type and number of characters in the text
- You can *group* or *quantify* atoms to describe your pattern
- Always use the bind operator (`=~`) to apply your regular expression to a variable

Remembering Stuff

- Being able to match patterns is good, but limited.
- We want to be able to keep portions of the regular expression for later.
 - Example: `$string = 'phone: 353-7236'`
 - We want to keep the phone number only
 - Just figuring out that the string contains a phone number is insufficient, we need to keep the number as well.

Memory Parentheses (pattern memory)

- Since we almost always want to keep portions of the string we have matched, there is a mechanism built into perl.
- Anything in parentheses within the regular expression is kept in memory.
 - ‘phone:353-7236’ =~ /^phone\:(.+)\$/;
 - Perl knows we want to keep everything that matches ‘.+’ in the above pattern

Getting at pattern memory

- Perl stores the matches in a series of default variables. The first parentheses set goes into \$1, second into \$2, etc.
 - This is why we can't name variables \${digit}
 - Memory variables are created only in the amounts needed. If you have three sets of parentheses, you have (\$1,\$2,\$3).
 - Memory variables are created for each matched set of parentheses. If you have one set contained within another set, you get two variables (inner set gets lowest number)
 - Memory variables are only valid in the current *scope*

An example of pattern memory

```
my $string = shift;
if ($string =~ /^phone\:(\d{3}-\d{4})$/){
    $phone_number = $1;
}
else {
    print "Enter a phone number!\n"
}
```

Finding all instances of a match

- Use the 'g' modifier to the regular expression
 - @sites = \$sequence =~ /(TATTA)/g;
 - think **g** for **g**lobal
 - Returns a list of all the matches (in order), and stores them in the array
 - If you have more than one pair of parentheses, your array gets values in sets
 - (\$1,\$2,\$3,\$1,\$2,\$3...)

Perl is Greedy

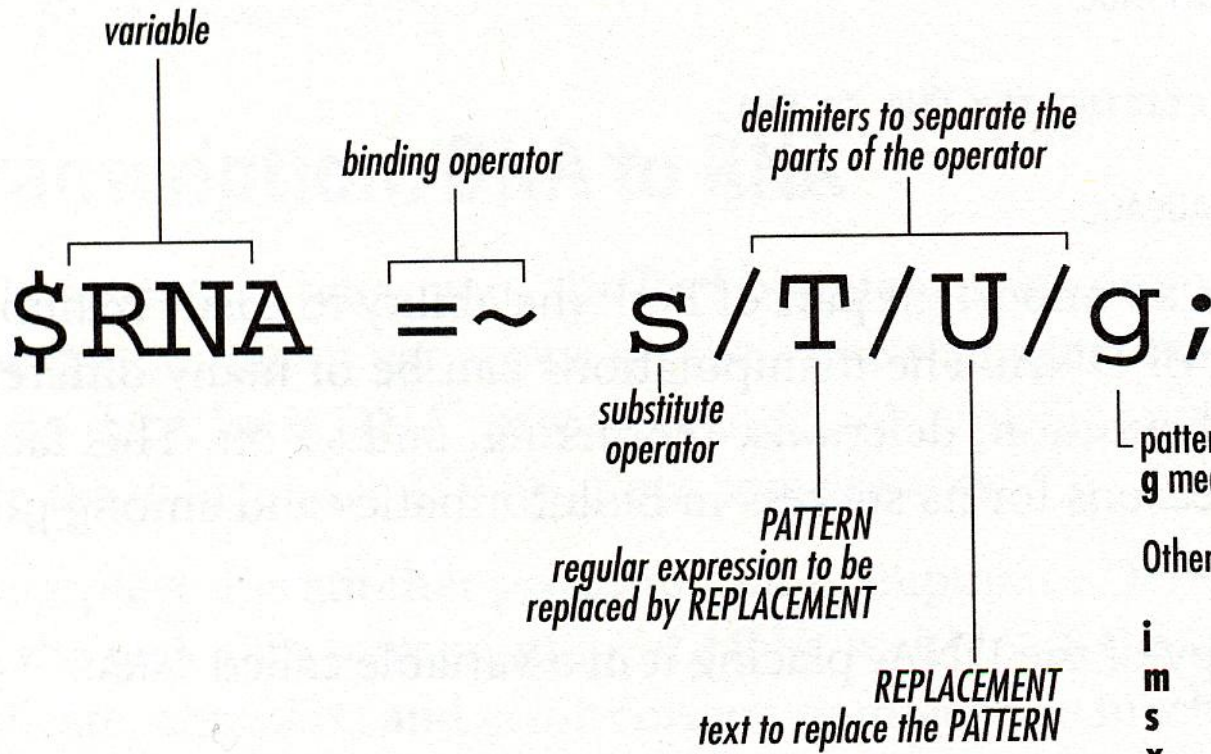
- In addition to taking all your time, perl regular expressions also try to match the largest possible string which fits your pattern
 - `/ga+t/` matches `gat`, `gaat`, `gaaat`
 - ‘Doh! No doughnuts left!’ `=~ /(d.+t)/`
 - `$1` contains ‘doughnuts left’
- If this is not what you wanted to do, use the ‘?’ modifier
 - `/(d.+?t)/` # match as few ‘.’s as you can and still make the pattern work

Substitute function

- *s/pattern 1/pattern 2/;*
- Looks kind of like a regular expression
 - Patterns constructed the same way
- Inherited from previous languages, so it can be a bit different.
 - Changes the variable it is bound to!

- Substituting one word for another
 - `$string =~ s/dogs/cats/;`
 - If `$string` was “I love dogs”, it is now “I love cats”
- Removing trailing white space
 - `$string =~ s/\s+$/;`
 - If `$string` was ‘ATG ’, it is now ‘ATG’
- Adding 10 to every number in a string
 - `$string =~ /(\d+)/$1+10/ge;`
 - If string was “I bought 5 dogs at 2 bucks each”, it is now:
 - “I bought 15 dogs at 12 bucks each”
 - Note pattern memory!!
 - **g** means **g**lobal (just like a regex)
 - **e** is special to `s`, **e**valuate the expression on the right

Substitutions



pattern modifier:
g means "globally, throughout the string".

Other common options are:

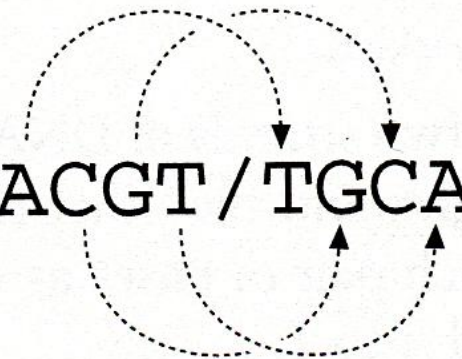
- i** case insensitive
- m** multiline (let `^` and `$` match embedded newline)
- s** single line (let `.` match newline)
- x** permit comments within PATTERN
- o** compile only once, for speed
- e** treat REPLACEMENT as Perl code

tr function

- *tr* translate or *tr* transliterate
- *tr/characterlist1/characterlist2/;*
- Even less like a regular expression than *s*
- substitutes characters in the first list with characters in the second list
 - \$string =~ *tr/a/A/;* # changes every 'a' to an 'A'
 - No need for the ***g*** modifier when using *tr*.

Translations

```
$revcom =~ tr/ACGT/TGCA/;
```



<i>base</i>	<i>maps to</i>	<i>base</i>
A	→	T
C	→	G
G	→	C
T	→	A

Using *tr*

- Creating complimentary DNA sequence
 - `$sequence =~ tr/atgc/TACG/;`
- Sneaky Perl trick for the day
 - *tr* does two things.
 - 1. changes characters in the bound variable
 - 2. Counts the number of times it does this
 - Super-fast character counter™
 - `$a_count = $sequence =~ tr/a/a/;`
 - replaces an 'a' with an 'a' (no net change), and assigns the result (number of substitutions) to `$a_count`

- **Regex-Related Special Variables**
- Perl has a host of special variables that get filled after every `m//` or `s///` regex match. `$1`, `$2`, `$3`, etc. hold the [backreferences](#). `$+` holds the last (highest-numbered) backreference. `$&` (dollar ampersand) holds the entire regex match.
- `@-` is an array of match-start indices into the string. `$-[0]` holds the start of the entire regex match, `$-[1]` the start of the first backreference, etc. Likewise, `@+` holds match-end indices (ends, not lengths).
- `$'` (dollar followed by an apostrophe or single quote) holds the part of the string after (to the right of) the regex match. `$`` (dollar backtick) holds the part of the string before (to the left of) the regex match. Using these variables is not recommended in scripts when performance matters, as it causes Perl to slow down *all* regex matches in your entire script.
- All these variables are read-only, and persist until the next regex match is attempted. They are dynamically scoped, as if they had an implicit 'local' at the start of the enclosing scope. Thus if you do a regex match, and call a sub that does a regex match, when that sub returns, your variables are still set as they were for the first match.

- **Finding All Matches In a String**
- The "/g" modifier can be used to process all regex matches in a string. The first m/regex/g will find the first match, the second m/regex/g the second match, etc. The location in the string where the next match attempt will begin is automatically remembered by Perl, separately for each string. Here is an example:
- `while ($string =~ m/regex/g) { print "Found '$&'. Next attempt at character " . pos($string)+1 . "\n"; }` The `pos()` function retrieves the position where the next attempt begins. The first character in the string has position zero. You can modify this position by using the function as the left side of an assignment, like in `pos($string) = 123;`.

Regex

- O'Reilly book: Mastering regular expressions (2nd edition)
- [Regular Expressions Tutorial](#)
- <http://www.regular-expressions.info/examples.html>

Oefeningen practicum 2

1. Which of following 4 sequences (seq1/2/3/4)
 - a) contains a “Galactokinase signature”



<http://us.expasy.org/prosite/>

- b) How many of them?
- c) *Where (hints:pos and \$&) ?*

>SEQ1

MGNLFENCTHRYSFYIYENCTNTTNCQGLIRNVASSIDVFHWLWVYISTTIFVISGILNFYCLFIALYT
YYFLDNETRKHVYFVLSRFLSSILVIISLLVLESTLFSELSPTFAYYAVAFSIYDFSMDFLFFSYIMIS
LITYFGVVHYNFYRRHVSLRSLYIILISMWTFSLAIAIPLGLYEASNSQGPICDLSYCGKVVEWITCS
LQGCDSFYANELLVQSISSVETLVGSLVFLTDPLINIFFDKNISKMVKLQTLGKWFIALYRFLFQMT
NIFENCSTHYSFEKNLQKCVNASNPCQLLQKMNTAHSLMIWWMGFYIPSAMCFLAVLVDTYCLLVTSILK
SLKKQSRKQYIFGRANIIGEHN DYVVVRLSAAILIALCIIIQSTYFIDIPFRDTFAFFAVLFIYDFSI LLSLLGSFTGVAM
MTYFGVMRPLVYRDKFTLTKTIYIIAFAIVLFSVCVAIPFGLFQAAD EIDGPIKCDSESC ELIVKWLLFCI
ACLILMGCTGTL L FVTVSLHWHSYKSKKMGNVSSSAFNHGKSRLTWTTTILVILCCVELIPTGLLA AFGK
SESISDDCYDFYNANSLIFPAIVSSLETFLGSITFLLDPIINF SFDKRISKVFSSQVSMFSIFFCGKR

>SEQ2

MLDDRARMEA AKKEKVEQIL AEFQLQEEDL KKVMMRMQKE MDRGLRLETH EASVKMLPT YVRSTPEGSE VGDFLSLDLG GTNFRVMLVK
VGEGEEGQWS VKTKHQMYSI PEDAMTGTA E MLFDYISECI SDFLDKHQMK HKKPLPGFTF SFPVRHEDID KGILLNWTKG
FKASGAEGNN VVGLLRDAIK RRGDFEMDVV AMVNDTVATM ISCYEDHQC EVGMIVGTGC NACYMEEMQN VELVEGDEGR
MCVNTQWGA F GDSGELDEF L LEYDRLVDES SANPGQLYE KLIGGKYMGE LVRLVLLRLV DENLLFHGEA SEQLRTRGAF
ETRFVSEQVES DTGDRKQIYN ILSTLGLRPS TTDCDIVRRA CESVSTRAAH MCSAGLAGVI NRMRESRSED VMRITVGVDG SVYKLHPSFK
ERFHASVRR L TPSCEITFIE SEEGSGRGAA LVS AVACKKA CMLGQ

>SEQ3

MESDSFEDFLKGEDFSNYSYSSDLPPFL LDAAPCEPESLEINKYFVVIIYVLVFLLSLLGNSLVMLVILY
SRVGRSGRDNVIGDHVDYVTDVYLLNLALADLLFALTLP IWAASKVTGWIFGTFLCKVV SLLKEVNFYSGILLLACISVDRY
LAIVHATR TLTKRYLVKFCISIWGLSLLLALPVLIFRKT IYPPYVSPVCYEDMGNNTANWRMLLRILP
QSFGFIVPLLIMLFCYGF LRTLRFKAHMGQKHRAMRVIFAVVLIFLLCWLPYNLVLLADTL MRTWVIQET
CERRNDIDRALEATEILGILGRVNLIGEHW DYHSCLNPLIYAFIGQKFRHG LLLKILAIHGLISKDSL PKDSRPSFVGSSSGH TSTTL

>SEQ4

MEANFQQAVK KLVNDFEYPT ESLREAVKEF DELRQKGLQK NGEVLAMAPA FISTLPTGAE TGDFLALDFG GTNLRVCW IQ LLGDGKYEMK
HSKSVLPREC VRNESVKPII DFMSDHVELF IKEHFPSKFG CPEEEYLP MG FTFSYPANQV SITESYLLRW TKGLNIPEAI NKDFAQFLTE
GFKARNLPIR IEAVINDTVG TLVTRAYTSK ESDTFMGIIF GTGTNGAYVE QMNQIPKLAG KCTGDHMLIN MEWGATDFSC LHSTRYDLLL
DHDTPNAGRQ IFEKRVGGMY LGELFRRALF HLIKVYNFNE GIFPPSITDA WSLET SVLSR MMVERSAENV RNVLSTFKFR FRSD E EALYL
WDAAHAIGRR AARMSAVPIA SLYLSTGRAG KKS DVGVDGS LVEHYPHFVD MLREALRELI GDNEKLISIG IAKDGS GIGA ALCALQAVKE
KKGLA MEANFQQAVK KLVNDFEYPT ESLREAVKEF DELRQKGLQK NGEVLAMAPA FISTLPTGAE TGDFLALDFG GTNLRVCW IQ
LLGDGKYEMK HSKSVLPREC VRNESVKPII DFMSDHVELF IKEHFPSKFG CPEEEYLP MG FTFSYPANQV SITESYLLRW TKGLNIPEAI
NKDFAQFLTE GFKARNLPIR IEAVINDTVG TLVTRAYTSK ESDTFMGIIF GTGTNGAYVE QMNQIPKLAG KCTGDHMLIN MEWGATDFSC
LHSTRYDLLL DHDTPNAGRQ IFEKRVGGMY LGELFRRALF HLIKVYNFNE GIFPPSITDA WSLET SVLSR MMVERSAENV RNVLSTFKFR
FRSD E EALYL WDAAHAIGRR AARMSAVPIA SLYLSTGRAG KKS DVGVDGS LVEHYPHFVD MLREALRELI GDNEKLISIG IAKDGS GIGA
ALCALQAVKE KKGLA

Three Basic Data Types

- Scalars - \$
- Arrays of scalars - @
- Associative arrays of scalars or Hashes - %

Arrays

Definitions

- A scalar variable contains a scalar value: one number or one string. A string might contain many words, but Perl regards it as one unit.
- An array variable contains a list of scalar data: a list of numbers or a list of strings or a mixed list of numbers and strings. The order of elements in the list matters.

Syntax

- Array variable names start with an @ sign.
- You may use in the same program a variable named \$var and another variable named @var, and they will mean two different, unrelated things.

Example

- Assume we have a list of numbers which were obtained as a result of some measurement. We can store this list in an array variable as the following:
- @msr = (3, 2, 5, 9, 7, 13, 16);

The *foreach* construct

The `foreach` construct iterates over a list of scalar values (e.g. that are contained in an array) and executes a block of code for each of the values.

- Example:
 - `foreach $i (@some_array) {`
 - `statement_1;`
 - `statement_2;`
 - `statement_3; }`
 - Each element in `@some_array` is aliased to the variable `$i` in turn, and the block of code inside the curly brackets `{ }` is executed once for each element.
- The variable `$i` (or give it any other name you wish) is local to the `foreach` loop and regains its former value upon exiting of the loop.
- Remark `$_`

Examples for using the *foreach* construct - cont.

- Calculate sum of all array elements:

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

```
@msr = (3, 2, 5, 9, 7, 13, 16);
```

```
$sum = 0;
```

```
foreach $i (@msr) {
```

```
  $sum += $i; }
```

```
print "sum is: $sum\n";
```

Accessing individual array elements

Individual array elements may be accessed by indicating their position in the list (their index).

Example:

```
@msr = (3, 2, 5, 9, 7, 13, 16);
```

index value 0 3 1 2 2 5 3 9 4 7 5 13 6 16

First element: \$msr[0] (here has the value of 3),

Third element: \$msr[2] (here has the value of 5),

and so on.

The *sort* function

The `sort` function receives a list of variables (or an array) and returns the sorted list.

```
@array2 = sort (@array1);
```

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

```
@countries = ("Israel", "Norway", "France", "Argentina");
```

```
@sorted_countries = sort (@countries);
```

```
print "ORIG: @countries\n", "SORTED: @sorted_countries\n";
```

Output:

```
ORIG: Israel Norway France Argentina
```

```
SORTED: Argentina France Israel Norway
```

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

```
@numbers = (1 ,2, 4, 16, 18, 32, 64);
```

```
@sorted_num = sort (@numbers);
```

```
print "ORIG: @numbers \n", "SORTED: @sorted_num \n";
```

Output:

```
ORIG: 1 2 4 16 18 32 64
```

```
SORTED: 1 16 18 2 32 4 64
```

Note that sorting numbers does not happen numerically, but by the string values of each number.

The *push* and *shift* functions

The push function adds a variable or a list of variables to the end of a given array.

Example:

```
$a = 5;
```

```
$b = 7;
```

```
@array = ("David", "John", "Gadi");
```

```
push (@array, $a, $b);
```

```
# @array is now ("David", "John", "Gadi", 5, 7)
```

The shift function removes the first element of a given array and returns this element.

Example:

```
@array = ("David", "John", "Gadi");
```

```
$k = shift (@array);
```

```
# @array is now ("John", "Gadi"); # $k is now "David"
```

Note that after both the push and shift operations the given array @array is changed!

How can I know the length of a given array?

You have three options:

- Assigning the array variable into a scalar variable, as in the [previous slide](#). This is not recommended, because the code is confusing.
- Use the scalar function. Example:
 - `$x = scalar (@array); # $x now contains the number of elements in @array.`
- Use the special variable `$#array_name` to get the index value of the last element of `@array_name`. Example:
 - `@fruits = ("apple", "orange", "banana", "melon");`
 - `$a = $#fruits;`
 - `# $a is now 3;`
 - `$b = $#fruits + 1;`
 - `# $b is now 4, i.e. # the no. of elements in @fruits.`

Special array for command line arguments @ARGV

```
#!/usr/bin/perl -w
# print out user-entered command line
  arguments
foreach $arg (@ARGV) {

    # print each argument followed by <tab>
    print $arg . "\t";

}
# print hard return
print "\n";
```

Perl Array review

- **An array is designated with the '@' sign**
- **An array is a list of individual elements**
- **Arrays are ordered**
 - Your list stays in the same order that you created it, although you can add or subtract elements to the front or back of the list
- **You access array elements by number, using the special syntax:**
 - `$array[1]` returns the '1th' element of the array (remember perl starts counting at zero)
- **You can do anything with an array element that you can do with a scalar variable (addition, subtraction, printing ... whatever)**

Generate random string

```
for($n=1;$n<=50;$n++){  
  
    @a = ("A","C","G","T");  
  
    $b=$a[rand(@a)];  
    $r.=$b;  
}  
print $r;
```

Text Processing Functions

The *split* function

- The `split` function splits a string to a list of substrings according to the positions of a given delimiter. The delimiter is written as a pattern enclosed by slashes: `/PATTERN/`. **Examples:**
- `$string = "programming::course::for::bioinformatics";`
- `@list = split (/::/, $string);`
- `# @list` is now ("programming", "course", "for", "bioinformatics") `# $string` remains unchanged.
- `$string = "protein kinase C\t450 Kilodaltons\t120 Kilobases";`
- `@list = split (/^t/, $string);` `#\t` indicates tab `#`
- `@list` is now ("protein kinase C", "450 Kilodaltons", "120 Kilobases")

Text Processing Functions

The *join* function

- The join function does the opposite of split. It receives a delimiter and a list of strings, and joins the strings into a single string, such that they are separated by the delimiter.
- Note that the delimiter is written inside quotes.
- **Examples:**
- `@list = ("programming", "course", "for", "bioinformatics");`
- `$string = join ("::", @list);`
- `# $string is now "programming::course::for::bioinformatics"`
- `$name = "protein kinase C"; $mol_weight = "450 Kilodaltons";`
`$seq_length = "120 Kilobases";`
- `$string = join ("\t", $name, $mol_weight, $seq_length);`
- `# $string is now: # "protein kinase C\t450 Kilodaltons\t120 Kilobases"`

Three Basic Data Types

- Scalars - \$
- Arrays of scalars - @
- Associative arrays of scalars or Hashes - %

When is an array not good enough?

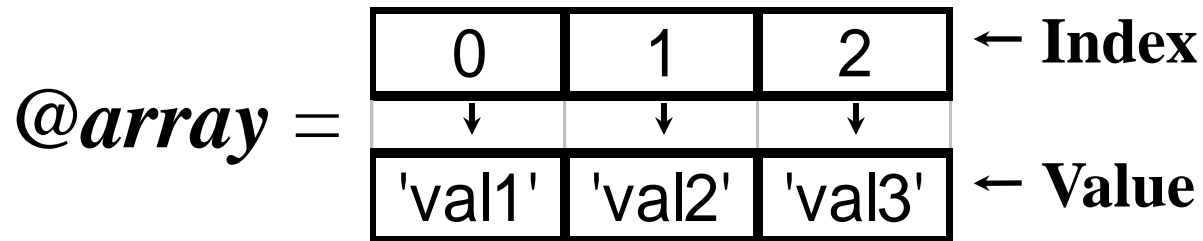
- Sometimes you want to associate a given value with another value. (name/value pairs)
(Rob => 353-7236, Matt => 353-7122,
Joe_anonymous => 555-1212)
(Acc#1 => sequence1, Acc#2 => sequence2, Acc#n
=> sequence-n)
- You could put this information into an array, but it would be difficult to keep your names and values together (what happens when you sort? Yuck)

Problem solved: The associative array

- As the name suggests, an associative array allows you to link a name with a value
- In perl-speak: associative array = hash
 - ‘hash’ is the preferred term, for various arcane reasons, including that it is easier to say.
- Consider an array: The elements (values) are each associated with a name – the index position. These index positions are numerical, sequential, and start at zero.
- A hash is similar to an array, but we get to name the index positions anything we want

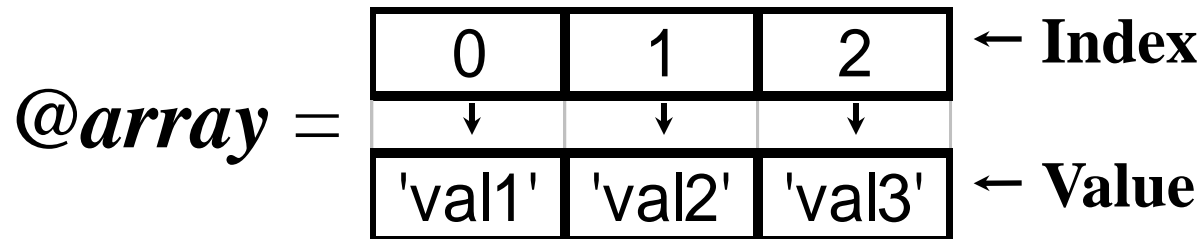
The 'structure' of a Hash

- An array looks something like this:

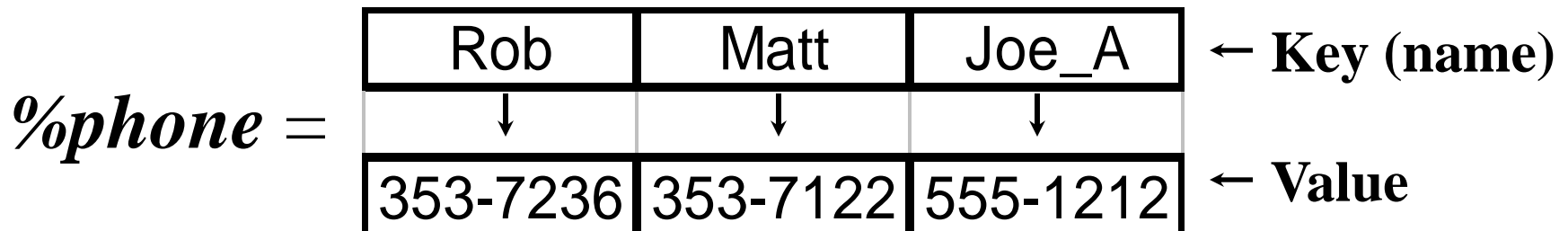


The 'structure' of a Hash

- An array looks something like this:



- A hash looks something like this:



Hash Rules:

- Names have the same rules as any other variables (no spaces, etc.)
- A hash is preceded by a ‘%’ sign
 - \$value => scalar variable
 - @array => array variable
 - %hash => hash variable
- A hash **key** can be any string
- Hash **keys** are unique!!
 - You may not have two keys in a hash with the same name.
 - **You may not have two keys in a hash with the same name. Ever. Really. I mean it this time.**

Creating a hash

- There are several methods for creating a hash. The most simple way – assign a list to a hash.
 - `%hash = ('rob', 56, 'joe', 17, 'jeff', 'green');`
- Perl is smart enough to know that since you are assigning a list to a hash, you meant to alternate keys and values.
 - `%hash = ('rob' => 56 , 'joe' => 17, 'jeff' => 'green');`
- The arrow (`'=>'`) notation helps some people, and clarifies which keys go with which values. The perl interpreter sees `'=>'` as a comma.

Getting at values

- You should expect by now that there is some way to get at a value, given a key.
- You access a hash key like this:
 - `$hash{'key'}`
- This should look somewhat familiar
 - `$array[21]` : refer to a value associated with a specific index position in an array
 - `$hash{key}` : refer to a value associated with a specific key in a hash

Getting at values, continued

- ***Magic incantation:*** Given a hash `%somehash`, you access the value in a specific key by this notation:
`$somehash{some_key}`
- Memorize this incantation!!!
 - If it helps, remember that you are getting a single element out of the hash, hence the `$` notation. To tell perl it is a hash, you use curly braces `{}`.

A phone book program

```
#!/usr/bin/perl -w

use strict;
my %phonenumber = ('Rob' => '353-7236',
                  'Matt' => '353-7122',
                  'Dave' => '353-5284',
                  'Jeff' => 'unlisted - go away');

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

print "${name}'s phone number is $phonenumber{$name}\n";

# note ${name} is a way to set off the variable name from any other
# text
# $name's may have been interpreted as the variable $name's
```

Count unique things with a hash

- Remember, keys must be unique. So,

```
while ($thing = <>){  
    chomp;  
    $hash{$thing}++  
}
```

- \$hash{key} is equal to “ the first time you see an item – add one to it
- \$hash{key} is equal to 1 the next time you see the same thing, add one to it.
- And so on...

Printing a Hash

- Of course there is a way to print a hash. It isn't as easy as printing an array:
 - *print @array*; **or** *print "@array"*;
 - There is no equivalent *print %hash*;
- We must visit each key and print its associated value. Sounds like a job for a loop...

Printing a hash (continued)

- First, create a list of keys. Fortunately, there is a function for that:
 - **keys** %hash (returns a list of keys)

- Next, visit each key and print its associated

value:

```
foreach (keys %hash){  
    print "The key $_ has the value $hash{$_}\n";  
}
```

- One complication. Hashes do not maintain any sort of order. In other words, if you put key/value pairs into a hash in a particular order, you will not get them out in that order!!

Programming in general and Perl in particular

- There is more than one right way to do it. Unfortunately, there are also many wrong ways.
 - 1. Always check and make sure the output is correct and logical
 - Consider what errors might occur, and take steps to ensure that you are accounting for them.
 - 2. Check to make sure you are using every variable you declare.
 - Use Strict !
 - 3. Always go back to a script once it is working and see if you can eliminate unnecessary steps.
 - Concise code is good code.
 - You will learn more if you optimize your code.
 - Concise does not mean comment free. Please use as many comments as you think are necessary.
 - Sometimes you want to leave easy to understand code in, rather than short but difficult to understand tricks. Use your judgment.
 - Remember that in the future, you may wish to use or alter the code you wrote today. If you don't understand it today, you won't tomorrow.

Programming in general and Perl in particular

Develop your program in stages. Once part of it works, save the working version to another file (or use a source code control system like RCS) before continuing to improve it.

When running interactively, show the user signs of activity.

There is no need to dump everything to the screen (unless requested to), but a few words or a number change every few minutes will show that your program is doing something.

Comment your script. Any information on what it is doing or why might be useful to you a few months later.

Decide on a coding convention and stick to it. For example,

- for variable names, begin globals with a capital letter and privates (my) with a lower case letter
- indent new control structures with (say) 2 spaces
- line up closing braces, as in: `if (....) { }`
- Add blank lines between sections to improve readability

Oefeningen practicum 2

2. Find the answer in ultimate-sequence.txt ?
(hint: use %AA1)

>ultimate-sequence

```

ACTCGTTATGATATTTTTTTTGAACGTGAAAATACTTTTTCGTGC
TATGGAAGGACTCGTTATCGTGAAGTTGAACGTTCTGAATG
TATGCCTCTTGAAATGGAAAATACTCATTGTTTATCTGAAAT
TTGAATGGGAATTTTATCTACAATGTTTTATTCTTACAGAAC
ATTAAATTGTGTTATGTTTCATTTCACATTTTAGTAGTTTTTT
CAGTGAAAGCTTGAAAACCAAGAAAGAAAAGCTGGTAT
GCGTAGCTATGTATATATAAAATTAGATTTTCCACAAAAAAT
GATCTGATAAACCTTCTCTGTTGGCTCCAAGTATAAGTACG
AAAAGAAATACGTTCCAAGAATTAGCTTCATGAGTAAGAA
GAAAAGCTGGTATGCGTAGCTATGTATATATAAAATTAGATT
TTCCACAAAAAATGATCTGATAA
  
```

```

my %AA1 = (
    'UUU','F',
    'UUC','F',
    'UUA','L',
    'UUG','L',
    'UCU','S',
    'UCC','S',
    'UCA','S',
    'UCG','S',
    'UAU','Y',
    'UAC','Y',
    'UAA','*',
    'UAG','*',
    'UGU','C',
    'UGC','C',
    'UGA','*',
    'UGG','W',

    'CUU','L',
    'CUC','L',
    'CUA','L',
    'CUG','L',
    'CCU','P',
    'CCC','P',
    'CCA','P',
    'CCG','P',
    'CAU','H',
    'CAC','H',
    'CAA','Q',
    'CAG','Q',
    'CGU','R',
    'CGC','R',
    'CGA','R',
    'CGG','R',

    'AUU','I',
    'AUC','I',
    'AUA','I',
    'AUG','M',
    'ACU','T',
    'ACC','T',
    'ACA','T',
    'ACG','T',
    'AAU','N',
    'AAC','N',
    'AAA','K',
    'AAG','K',
    'AGU','S',
    'AGC','S',
    'AGA','R',
    'AGG','R',

    'GUU','V',
    'GUC','V',
    'GUA','V',
    'GUG','V',
    'GCU','A',
    'GCC','A',
    'GCA','A',
    'GCG','A',
    'GAU','D',
    'GAC','D',
    'GAA','E',
    'GAG','E',
    'GGU','G',
    'GGC','G',
    'GGA','G',
    'GGG','G' );

```

3. Palindromes

What is the longest palindromic sequence in palin.fasta ?

Why are restriction sites palindromic ?

How long is the longest palindromic sequence in the genome ?

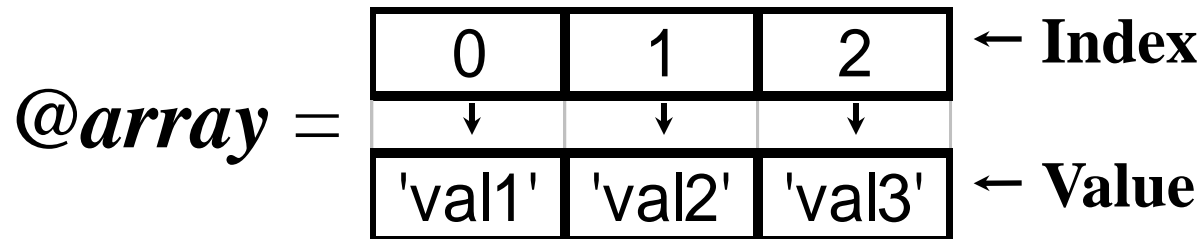
Hints:

http://www.man.poznan.pl/cmst/papers/5/art_2/vol5art2.html

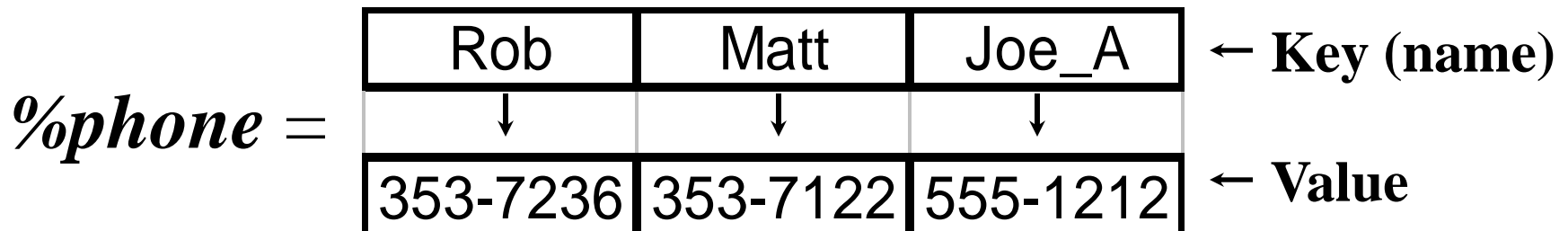
Palingram.pl

The 'structure' of a Hash

- An array looks something like this:



- A hash looks something like this:

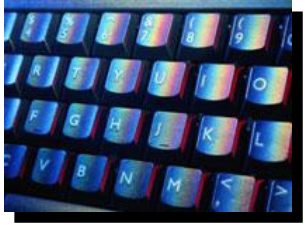


Sub routine

```
$a=5;  
$b=9;  
$sum=Optellen(5,9);  
print "The SUM is $sum\n";
```

```
sub Optellen()  
{  
    $d=@_[0];  
    $e=@_[1];  
    #alternatively we could do this:  
    my($a,$b)=@_  
    my($answer)=$d+$e;  
    return $answer;  
}
```

Overview



- Advanced data structures in Perl
- Object-oriented Programming in Perl
- Bioperl: is a large collection of Perl software for bioinformatics
- Motivation:
 - Simple extension: “Multiline parsing“ more difficult than expected
- Goal: to make software modular, easier to maintain, more reliable, and easier to reuse

Multi-line parsing



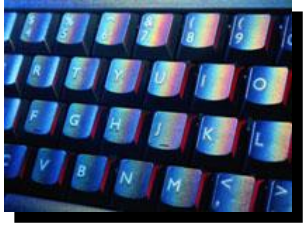
```
use strict;  
use Bio::SeqIO;
```

```
my $filename="sw.txt";  
my $sequence_object;
```

```
my $seqio = Bio::SeqIO -> new (   
    '-format' => 'swiss',  
    '-file' => $filename  
);
```

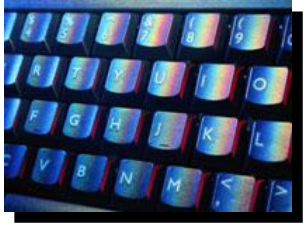
```
while ($sequence_object = $seqio -> next_seq) {  
    my $sequeintie = $sequence_object-> seq();  
    print $sequeintie."\n";  
}
```

Advanced data structures in Perl



- The scalar, array and hash data structures are builtin datatypes in Perl and cover most programming needs
- More complex data structures can be built from this basic ones

References: Scalars



- Advanced data structures depend on Perl references. A reference is a scalar variable that “points at” some other value (the “referent”)

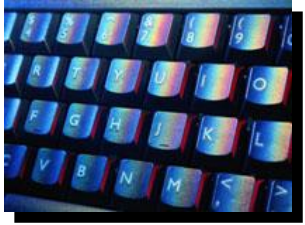
```
$peptide = “TGDTSGGT”;
```

```
$peptideref = \ $peptide;
```

```
Print $peptideref.”\n”;
```

```
Print $$peptideref.”\n”;
```

Reference to arrays



```
@tags=("acacac","tgtgtgt","gcgcgc");  
$tagref = \@tags;  
print "@$tagref\n";  
print $$tagref[1]."\n";  
print $tagref->[1]."\n";  
push (@$tagref,"aaaaa");  
print "@tags";
```


References to hashes



```
%geneticmarkers =  
    ('curly'=>'yes','hairy'=>'no','topiary'=>'yes');  
$hashref = \%geneticmarkers;  
print $hashref."\n";  
foreach $k (keys %$hashref) {  
    print "key\t$k\t\tvalue\t$$hashref{$k}\n";  
}  
foreach $k (keys %$hashref) {  
    print "key\t$k\t\tvalue\t$hashref->{$k}\n";  
}
```

Anonymous ...

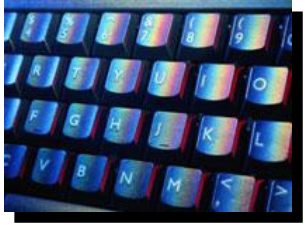
- A value need not be contained in a defined variable to create a reference
- To create an anonymous array reference, use square brackets instead of parentheses:
 - `$a_ref = [20, 30, 50, "hi!!"];`
 - `@a = @$a_ref;`
 - `@a → (20, 30, 50, “hi!!”);`
- For hash references, use curly brackets instead of parentheses:
 - `$h_ref={ "sky"=>'blue' , "grass"=>'green' }`
 - `%h = %$h_ref;`
 - `%h → (“sky” => ‘blue’, “grass” => ‘green’);`

Programming paradigms

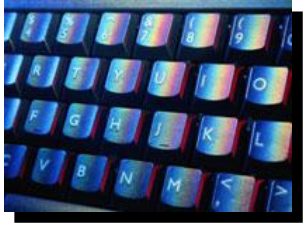


- Declarative (procedural) programming
 - Perl, C, Fortran, Pascal, Basic, ...
- Functional programming
 - Lisp, SML, ...
- Logic programming
 - Prolog
- Object-oriented programming
 - Perl, C++, Smalltalk, Java, ...

The key idea



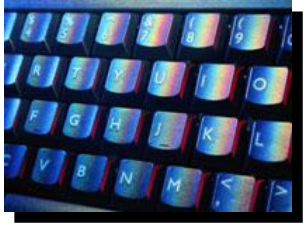
- All data is stored and used by means of **objects**, and each object can only be accessed by means of the defined subroutines call **methods**.
- Together, the objects and methods constitute a **class**
- Warning: a lot of conflicting terminology among OO practitioners



- **A class** is a package
- **An object** is a reference to a data structure (usually a hash) in a class
- **A method** is a subroutine in the class



- **Modules/Packages**
 - A Perl module is a file that uses a package declaration
 - Packages provide a separate namespace for different parts of program
 - A namespace protects the variable of one part of a program from unwanted modification by another part of the program
 - The module must always have a last line that evaluates to true, e.g. 1;
 - The module must be in “known” directory (environment variable)
 - Eg ... site/lib/bio/Sequentie.pm



```
use Bio::Sequentie;  
use Data::Dumper;
```

```
%sequence_object =  
  (_id=>"AF001",_name=>"Demo",_seq=>"ATGAT  
  G");
```

```
print Dumper(%sequence_object);
```

```
$seqRef = \%sequence_object;
```

```
Bio::Sequentie::RandomSeq($seqRef,100);
```

```
print Dumper($seqRef);
```

```
.. /site/lib/bio/Sequentie.pm
```



```
package Bio::Sequentie;
```

```
use strict;
```

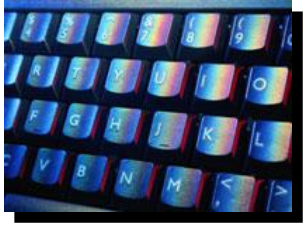
```
use Data::Dumper;
```

```
sub RandomSeq {  
    my @nucleotides = ('A','C','G','T');  
    my $self = shift;  
    #print $self;  
    print Dumper($self);  
    my $length = shift || 40;  
    print $length;  
    $self->{_id} = "0000";  
    $self->{_name} = "Random Sequence";  
    $self->{_seq} = "";  
    for (my $i = 0; $i < $length; $i++) {  
        my $base = $nucleotides[int(rand(4))];  
        $self->{_seq} .= $base;  
    }  
    #return 1;  
}
```


Bless



- Making objects ... **Bless** attaches a class name to a reference, making it possible to use references as objects in a class



```
use Bio::Sequentie;  
use Data::Dumper;
```

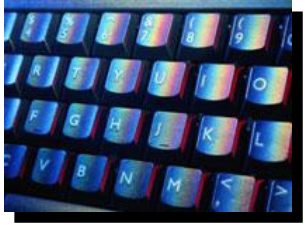
```
%sequence_object =  
  (_id=>"AF001",_name=>"Demo",_se  
   q=>"ATGATG");  
$seqRef = \%sequence_object;  
 bless ($seqRef,"Bio::Sequentie");  
 $seqRef->RandomSeq(100);  
print Dumper($seqRef);
```

Bless



- Making objects ... **Bless** attaches a class name to a reference, making it possible to use references as objects in a class
- **New** is the usual name given to the special method in the class that creates a new class object (CONSTRUCTOR)

Method: NEW (constructor)



```
sub new {  
  print "new";  
  my ($class, $id, $name, $seq) = @_;  
  my $ref =  
    {_id=>$id, _name=>$name, _seq=>$seq};  
  return bless $ref, $class;  
}
```

Random Sequence an OO Way



```
use Bio::Sequencie;  
use Data::Dumper;
```

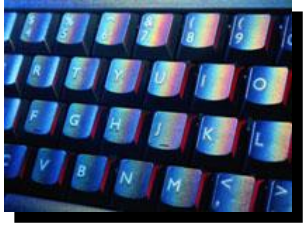
```
$seqRef=Bio::Sequencie-  
->new("007","Constructor","it works");
```

```
print Dumper($seqRef);
```

```
$seqRef->RandomSeq(100);
```

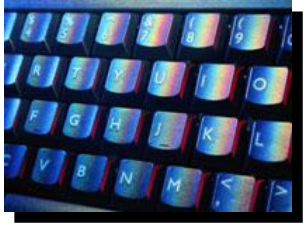
```
print Dumper($seqRef);
```

Review



- Three OO rules
 - Classes are packages
 - Methods are subroutines
 - Objects are blessed referents
 - Bless (\$seq, “Bio::Seq”);

CPAN



- CPAN: The Comprehensive Perl Archive Network is available at www.cpan.org and is a very large respository of Perl modules for all kind of taks (including bioperl)

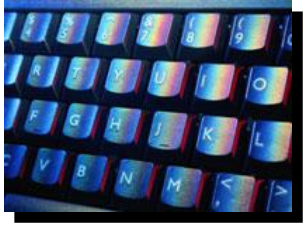
What is BioPerl?

- An 'open source' project
 - <http://bio.perl.org> or <http://www.cpan.org>
- A loose international collaboration of biologist/programmers
 - Nobody (that I know of) gets paid for this
- A collection of PERL modules and methods for doing a number of bioinformatics tasks
 - Think of it as subroutines to do biology
- Consider it a 'tool-box'
 - There are a lot of nice tools in there, and (usually) somebody else takes care of fixing parsers when they break
- BioPerl code is portable - if you give somebody a script, it will probably work on their system

What BioPerl isn't

- 'Out of the box' solutions to problems
 - You will have to know a little perl, and you will have to read documentation
- Particularly well documented
 - Yes, there is documentation, but it can be difficult to see the 'big picture' - or sometimes the small picture
- It can be a substantial investment in time to learn how to use bioperl properly
 - Once you get used to it, it is pretty simple to do some complicated things

Installing Modules



- Steps for installing modules
 - Uncompress the module
 - Gunzip file.tar.gz
 - Tar -xvf file.tar
 - perl Makefile.PL
 - make
 - make test
 - make install

Bioperl installation



- PPM
- Perl Package Manager

- Search Bioperl
- Install Bioperl

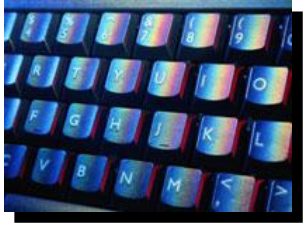
- **What is the PPM?**
- The Programmer's Package Manager (PPM), formerly known as the Perl Package Manager, provides a command line interface for managing your modules and extensions (packages). PPM is used to access package repositories (or collections of packages on other machines), install and remove packages from your system, as well as update previously installed packages with the latest versions.

- E:\Perl\Bin\ppm



- Using PPM shell to install BioPerl
 - Get the number of the BioPerl repository:
 - PPM>repository
 - Set the BioPerl repository, find BioPerl, install BioPerl:
 - PPM>repository set <BioPerl repository number>
 - PPM>search *
 - PPM>install <BioPerl package number>
- Download BioPerl in archive form from
 - <http://www.BioPerl.org/Core/Latest/index.shtml>
 - Use winzip to uncompress and install

Directory Structure



- BioPerl directory structure organization:
 - Bio/ BioPerl modules
 - models/ UML for BioPerl classes
 - t/ Perl built-in tests
 - t/data/ Data files used for the tests
 - scripts/ Reusable scripts that use BioPerl
 - scripts/contributed/ Contributed scripts not necessarily integrated into BioPerl.
 - doc/ "How To" files and the FAQ as XML

Folders	Name	Size	Type	Date Modified
html	Align		Folder	16/10/2003 14:40
lib	AlignIO		Folder	16/10/2003 14:40
site	Annotation		Folder	16/10/2003 14:40
lib	Assembly		Folder	16/10/2003 14:40
lib	Biblio		Folder	16/10/2003 14:40
lib	Cluster		Folder	16/10/2003 14:40
lib	ClusterIO		Folder	16/10/2003 14:40
lib	Coordinate		Folder	16/10/2003 14:40
lib	Das		Folder	16/10/2003 14:40
lib	DB		Folder	16/10/2003 14:40
lib	Event		Folder	16/10/2003 14:40
lib	Expression		Folder	16/10/2003 14:40
lib	Factory		Folder	16/10/2003 14:40
lib	Graphics		Folder	16/10/2003 14:40
lib	Index		Folder	16/10/2003 14:40
lib	LiveSeq		Folder	16/10/2003 14:40
lib	Location		Folder	16/10/2003 14:40
lib	Map		Folder	16/10/2003 14:40
lib	MapIO		Folder	16/10/2003 14:40
lib	Matrix		Folder	16/10/2003 14:40
lib	Ontology		Folder	16/10/2003 14:40
lib	OntologyIO		Folder	16/10/2003 14:40
lib	Phenotype		Folder	16/10/2003 14:40
lib	Root		Folder	16/10/2003 14:40
lib	Search		Folder	16/10/2003 14:40
lib	SearchIO		Folder	16/10/2003 14:40
lib	Seq		Folder	16/10/2003 14:40
lib	SeqFeature		Folder	16/10/2003 14:40
lib	SeqIO		Folder	16/10/2003 14:40
lib	Structure		Folder	16/10/2003 14:40
lib	Symbol		Folder	16/10/2003 14:40
lib	Taxonomy		Folder	16/10/2003 14:40
lib	Tools		Folder	16/10/2003 14:40
lib	Tree		Folder	16/10/2003 14:40
lib	TreeIO		Folder	16/10/2003 14:40
lib	Variation		Folder	16/10/2003 14:40
Bundle	Variation		Folder	16/10/2003 14:40
Compress	Variation		Folder	16/10/2003 14:40
Data	Variation		Folder	16/10/2003 14:40
DBD	Variation		Folder	16/10/2003 14:40
DBT	Variation		Folder	16/10/2003 14:40
	AlignIO.pm	14 KB	PM File	22/10/2002 9:38
	AnalysisI.pm	23 KB	PM File	4/07/2003 4:40
	AnalysisParserI.pm	6 KB	PM File	1/12/2002 0:05
	AnalysisResultI.pm	7 KB	PM File	1/10/2003 1:30
	AnnotatableI.pm	3 KB	PM File	31/12/2002 13:09
	AnnotationCollectionI.pm	5 KB	PM File	1/10/2003 1:30
	AnnotationI.pm	5 KB	PM File	22/10/2002 9:38
	Biblio.pm	11 KB	PM File	22/10/2002 9:45
	ClusterI.pm	5 KB	PM File	25/10/2002 3:29
	ClusterIO.pm	8 KB	PM File	21/01/2003 1:11
	DasI.pm	13 KB	PM File	11/11/2002 18:16
	DBLinkContainerI.pm	3 KB	PM File	1/10/2003 1:31
	DescribableI.pm	3 KB	PM File	25/10/2002 3:29
	FeatureHolderI.pm	6 KB	PM File	19/11/2002 7:04
	Graphics.pm	3 KB	PM File	12/09/2003 15:38

Sequence Object Creation

Sequence Creation :

```
$sequence = Bio::Seq->new( -seq => 'AATGCAA');
```

```
$sequence = Bio::Seq->new( -file => 'sequencefile.fasta');
```

```
$sequence = Bio::Seq->new( -file => 'sequencefile',  
                           -ffmt => 'gcg' );
```

Flat File Format Support :

Raw, FASTA, GCG, GenBank, EMBL, PIR

Via ReadSeq: IG, NBRF, DnaStrider, Fitch, Phylip, MSF, PAUP

Multi-line parsing

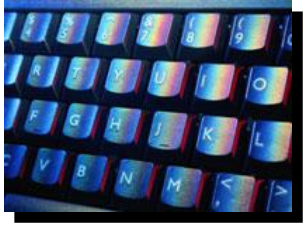


```
use strict;  
use Bio::SeqIO;
```

```
my $filename="sw.txt";  
my $sequence_object;
```

```
my $seqio = Bio::SeqIO -> new (  
    '-format' => 'swiss',  
    '-file' => $filename  
);
```

```
while ($sequence_object = $seqio -> next_seq) {  
    my $sequeintie = $sequence_object-> seq();  
    print $sequeintie."\n";  
}
```



```
#!/e:\Perl\bin\perl.exe -w
# script for looping over genbank entries, printing out name
use Bio::DB::Genbank;
use Data::Dumper;

$gb = new Bio::DB::GenBank();

$sequence_object = $gb->get_Seq_by_id('MUSIGHBA1');
print Dumper ($sequence_object);

$seq1_id = $sequence_object->display_id();
$seq1_s = $sequence_object->seq();
print "seq1 display id is $seq1_id \n";
print "seq1 sequence is $seq1_s \n";
```


File converter

```
#!/opt/perl/bin/perl -w
#genbank_to_fasta.pl
use Bio::SeqIO;
my $input = Bio::SeqIO::new->('-file' => $ARGV[0],
                              '-format' =>
                                'GenBank');
my $output = Bio::SeqIO::new->('-file' => '>output.fasta',
                              '-format' => 'Fasta');

while (my $seq = $input->next_seq()){
    $output->write_seq($seq)
}
}
```

Changing Formats (with Filehandles):

```
#!/opt/perl/bin/perl -w
#genbank_to_fasta_with_filehandles.pl
use Bio::SeqIO;
my $input = Bio::SeqIO::newFh->('-file' => $ARGV[0],
                                '-format' =>
                                'GenBank');
my $output = Bio::SeqIO::newFh->('-format' => 'Fasta');

while (<$input>){
    print $output $_
}
```



- Bptutorial.pl
- It includes the written tutorial as well as runnable scripts
- **2 ESSENTIAL TOOLS**
 - Data::Dumper to find out what class your in
 - Perl bptutorial (100 Bio::Seq) to find the available methods for that class

Oef 1



- Zoek het meest zure en het meest basische aminozuur in Swiss-Prot door het isoelectrisch punt te berekenen.
- Is er een biologische verklaring voor de gevonden resultaten ?

Uit AAIndex

H ZIMJ680104

D Isoelectric point (Zimmerman et al., 1968)

R LIT:2004109b PMID:5700434

A Zimmerman, J.M., Eliezer, N. and Simha, R.

T The characterization of amino acid sequences in proteins by statistical methods

J J. Theor. Biol. 21, 170-201 (1968)

C KLEP840101 0.941 FAUJ880111 0.813 FINA910103 0.805

I	A/L	R/K	N/M	D/F	C/P	Q/S	E/T	G/W	H/Y	I/V
	6.00	10.76	5.41	2.77	5.05	5.65	3.22	5.97	7.59	6.02
	5.98	9.74	5.74	5.48	6.30	5.68	5.66	5.89	5.66	5.96

- Database
 - (choose)
<http://www.ebi.ac.uk/swissprot/FTP/ftp.html>
 - (small)
 - <http://biobix.ugent.be/zip/swiss-prot.zip>



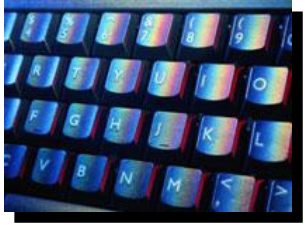
- CpG Islands
 - Download from **ENSEMBL** 100 (random) promoters (3000 bp)
 - How many times would you expect to observe CG if all nucleotides were equipropable
 - Count the number of times CG is observed for these 100 genes and make a histogram from these scores
 - CG repeats are often methylated. In order to study methylation patterns bisulfite treatment of DNA is used. Bisulfite changes every C which is not followed by G into T. Generate computationally the bisulfite treated version of DNA.
 - How would you find primers that discriminate between methylated and unmethylated DNA ? Given that the genome is $3 \cdot 10^9$ bp how long do you need to make a primer to avoid mispriming ?

So what is BioPerl? (continued...)



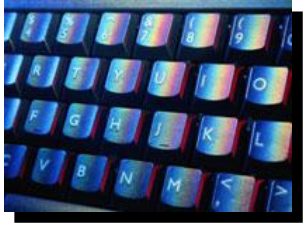
- 551 modules (incl. 82 interface modules)
- 37 module groups
- 79,582 lines of code (223,310 lines total)
- 144 lines of code per module
- For More info: [BioPerl Module Listing](#)

Searching for Sequence Similarity



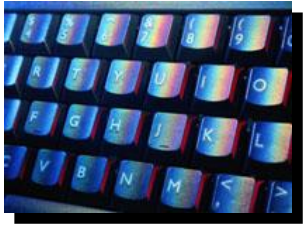
- BLAST with BioPerl
- Parsing Blast and FASTA Reports
 - Search and SearchIO
 - BPLite, BPpsilite, BPbl2seq
- Parsing HMM Reports
- Standalone BioPerl BLAST

Remote Execution of BLAST



- BioPerl has built in capability of running BLAST jobs remotely using RemoteBlast.pm
- Runs these jobs at NCBI automatically
 - NCBI has dynamic configurations (server side) to “always” be up and ready
 - Automatically updated for new BioPerl Releases
- Convenient for independent researchers who do not have access to huge computing resources
- Quick submission of Blast jobs without tying up local resources (especially if working from standalone workstation)
- Legal Restrictions!!!

Example of Remote Blast



A script to run a remote blast would be something like the following skeleton:

```
$remote_blast = Bio::Tools::Run::RemoteBlast->new(  
  '-prog' => 'blastp', '-data' => 'ecoli', '-expect' => '1e-  
  10' );  
$r = $remote_blast->submit_blast("t/data/ecolist.fa");  
while ( @rids = $remote_blast->each_rid ) { foreach  
  $rid ( @rids ) { $src = $remote_blast->  
    retrieve_blast($rid); } }
```

In this example we are running a blastp (pairwise comparison) using the ecoli database and a e-value threshold of 1e-10. The sequences that are being compared are located in the file “t/data/ecolist.fa”.

Example



It is important to note that all command line options that fall under the blastall umbrella are available under BlastRemote.pm.

For example you can change some parameters of the remote job.

Consider the following example:

```
$Bio::Tools::Run::RemoteBlast::HEADER{'MATRIX_NAME'} = 'BLOSUM25';
```

This basically allows you to change the matrix used to BLOSUM 25, rather than the default of BLOSUM 62.

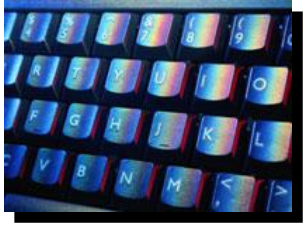
Parsing Blast Reports

- One of the strengths of BioPerl is its ability to parse complex data structures. Like a blast report.
- Unfortunately, there is a bit of arcane terminology.
- Also, you have to ‘think like bioperl’, in order to figure out the syntax.
- This next script might get you started

Sample Script to Read and Parse BLAST Report



```
# Get the report $searchio = new Bio::SearchIO (-format =>
    'blast', -file => $blast_report);
$result = $searchio->next_result; # Get info about the entire
    report $result->database_name;
$algorithm_type = $result->algorithm;
# get info about the first hit $hit = $result->next_hit;
$hit_name = $hit->name ;
# get info about the first hsp of the first hit $hsp =
    $hit->next_hsp;
$hsp_start = $hsp->query->start;
```



- Similar to Search and SearchIO in basic functionality
- However:
 - Older and will likely be phased out in the near future
 - Substantially limited advanced functionality compared to Search and SearchIO
 - Important to know about because many legacy scripts utilize these objects and either need to be converted

Parse BLAST output

```
#!/opt/perl/bin/perl -w
#bioperl_blast_parse.pl
# program prints out query, and all hits with scores for each blast result
use Bio::SearchIO;

my $record = Bio::SearchIO->new(-format => 'blast', -file => $ARGV[0]);

while (my $result = $record->next_result){
    print ">", $result->query_name, " ", $result->query_description, "\n";
    my $seen = 0;
    while (my $hit = $result->next_hit){
        print "\t", $hit->name, "\t", $hit->bits, "\t", $hit->significance, "\n";
        $seen++ }
    if ($seen == 0 ) { print "No Hits Found\n" }
}
```

Parse BLAST in a little more detail

```
#!/opt/perl/bin/perl -w
#bioperl_blast_parse_hsp.pl
# program prints out query, and all hsps with scores for each blast result
use Bio::SearchIO;
my $record = Bio::SearchIO->new(-format => 'blast', -file => $ARGV[0]);
while (my $result = $record->next_result){
    print ">", $result->query_name, " ", $result->query_description, "\n";
    my $seen = 0;
    while (my $hit = $result->next_hit{
        $seen++;
        while (my $hsp = $hit->next_hsp){
            print "\t", $hit->name, "has an HSP with an eval of: ",
            $hsp->evaluate, "\n";}
        if ($seen == 0 ) { print "No Hits Found\n" }
    }
}
```

SearchIO parsers:

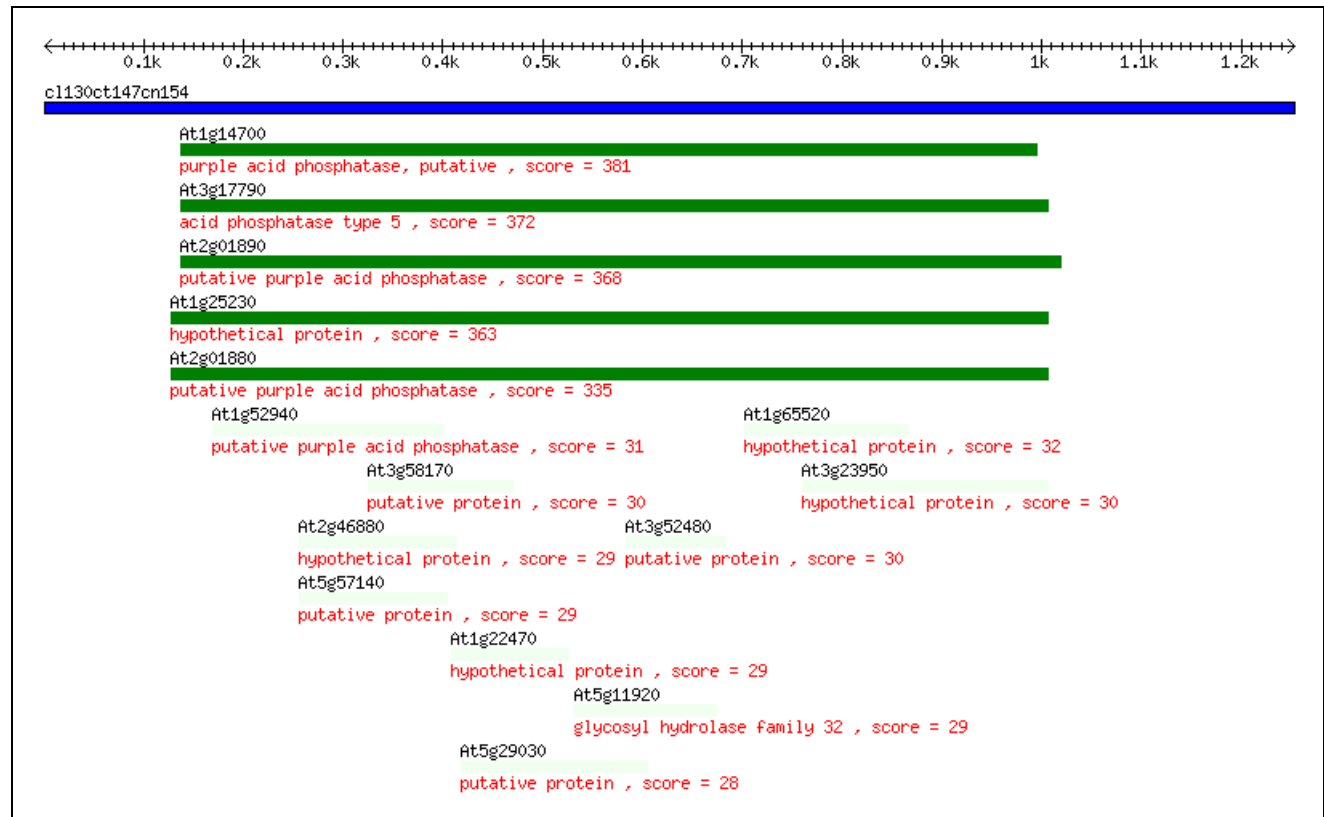
- SearchIO can parse (and reformat) several formats containing alignment or similarity data:
 - blast
 - xml formatted blast (still a little wonky)
 - psi-blast
 - exonerate
 - WABA
 - FASTA
 - HMMER
- The interface is the same for all of these, making your life a little easier.

What else is in BioPerl?

- Cluster or Assembly tools
- Protein structure tools
- Graphics
- Database functions
- Phylogeny tools
- Bibliography tools
- ...

Some advanced BioPerl

- What if you want to draw pretty images of blast reports?
- Bio::Graphics to the rescue:
- Input - A blast report (single query):
- Output:



Some really advanced BioPerl

- What if you want to display an entire genome with annotation? Grow your own genome project
 - We can do that all in BioPerl (and a mySQL or Oracle database):
- GBrowse - the Generic Genome Browser
 - Allows any feature to be displayed on a reference sequence
 - Many different styles of 'Glyphs', so all features can be drawn in a different style
 - Allows user to zoom in and out on reference sequence
 - User can select which features to display
 - User can upload their own features to be displayed on your reference sequence
 - And more!!!
- <http://www.bch.msu.edu/cgi-bin/gbrowse>
- <http://www.gmod.org/>

GBrowse (Generic Genome Browser)

Showing 10 Kbp from II, positions 401,570 to 411,569

Instructions [Hide]: Search using a sequence name, gene name, locus, or other landmark. The wildcard character * is allowed. **Examples:** II, II:80,000..120,000, Mit, NPY1, NAB2, Orf.YGL123W. [Help]
 To center on a location, click the ruler. Use the Scroll/Zoom buttons to change magnification and position. To save this view, bookmark this link.

Landmark or Region
 II:401570..411569

Scroll/Zoom: Show 10 Kbp

Data Source: S. cerevisiae (via SGD Nov 2001)

Tracks [Hide]

<input checked="" type="checkbox"/> Named gene	<input type="checkbox"/> Centromeres	<input checked="" type="checkbox"/> DNA/GC Content
(External tracks italicized)	<input type="checkbox"/> Transposons	<input type="checkbox"/> 3-frame translation (reverse)
<input checked="" type="checkbox"/> ORF	<input type="checkbox"/> Long Terminal Repeats	<input type="checkbox"/> Noncoding RNAs
<input type="checkbox"/> CDS	<input type="checkbox"/> 3-frame translation (forward)	<input type="checkbox"/> plugin:Restriction Sites
<input checked="" type="checkbox"/> tRNAs		

Image Width: 450 640 800 1024

Key position: Between Beneath

Upload your own annotations: [Help]

Add remote annotations: [Help]
 Enter Remote Annotation URL

Palin.fasta

- >palin.fasta
- ATGGCTTATTTATTTGCCCAACAAGAACTTAGGGTGCATTGAAATCTAAA
 GCTAATTGCTTATTTAGCTTTGCTTGGCCTTTTCACTTAAATAAAACA
 TAGCATCAACTTCAGCAGGAATGGGTGCACATGCTGATCGAGGTGG
 AAGAAGGGCACATATGGCATCGGCATCCTTATGGCTAATTTTAAATG
 GAGAACTTTCTAAAGTCACGTTTTTCACATGCAATATTCTTAACATTTT
 CAATTTTTTTTGTAACTAATTCTTCCCATCTACTATGTGTTTGCAAGAC
 AATCTCAGTAGCAAACCTCCTTATGCTTAGCCTCACCGTTAAAAGCAA
 ACTTATTTGGGGGATCTCCACCAGGCATTTTATATATTTTGAACCACT
 CTA CTGACGCGTTAGCTTCAAGTAAACCAGGCATCACTTCTTTTACG
 TCATCAATATCATTAAAGCTTTGAAGCTAGAGGATCATTTACATCAATT
 GCTATTACTTAGCTTAGCCCTTCAAGTACTTGAAGGGCTAAGCTTCC
 AATCTGTTTCACCATTGTCAATCATAGCTAAGACACCCAGCAACTTAA
 CTTGCAAACAGATCCTCTTTCTGCAACTTTGTAACCTATCTCTATTA
 CATCAACAGGATCACCATCACCAAATGCATTAGTGTGCTCATCAATA
 AGATTTGGATCCTCCAAGTCTGTGGCAAAGCTCCATAATTCCAAGG
 ATAACC

Palingram.pl

```

#!E:\perl\bin\perl -w
$line_input = "edellede parterretrap trap op sirenes en er is popart test";
$line_input =~ s/\s//g;
$l = length($line_input);
for ($m = 0; $m <= $l - 1; $m++)
{
    $line = substr($line_input, $m);
    print "length=$m:$l\t".$line."\n";
    for $n (8..25)
    {
        $re = qr /[a-z]{$n}/;
        print "pattern ($n) = $re\n";
        $regexes[$n-8] = $re;
    }
    foreach (@regexes)
    {
        while ($line =~ m/$_/g)
        {
            $endline = $';
            $match = $&;
            $all = $match.$endline;
            $revmatch = reverse($match);
            if ($all =~ /^($revmatch)/)
            {
                $palindrome = $revmatch . "*" . $1 ;
                $palhash{$palindrome}++;
            }
        }
    }
}

```

```

print "Set van palingram\n";
while(($key, $value) = each (%palhash))
{
    print "$key => $value\n";
}

```