



Welcome to Research Computing Services training week!

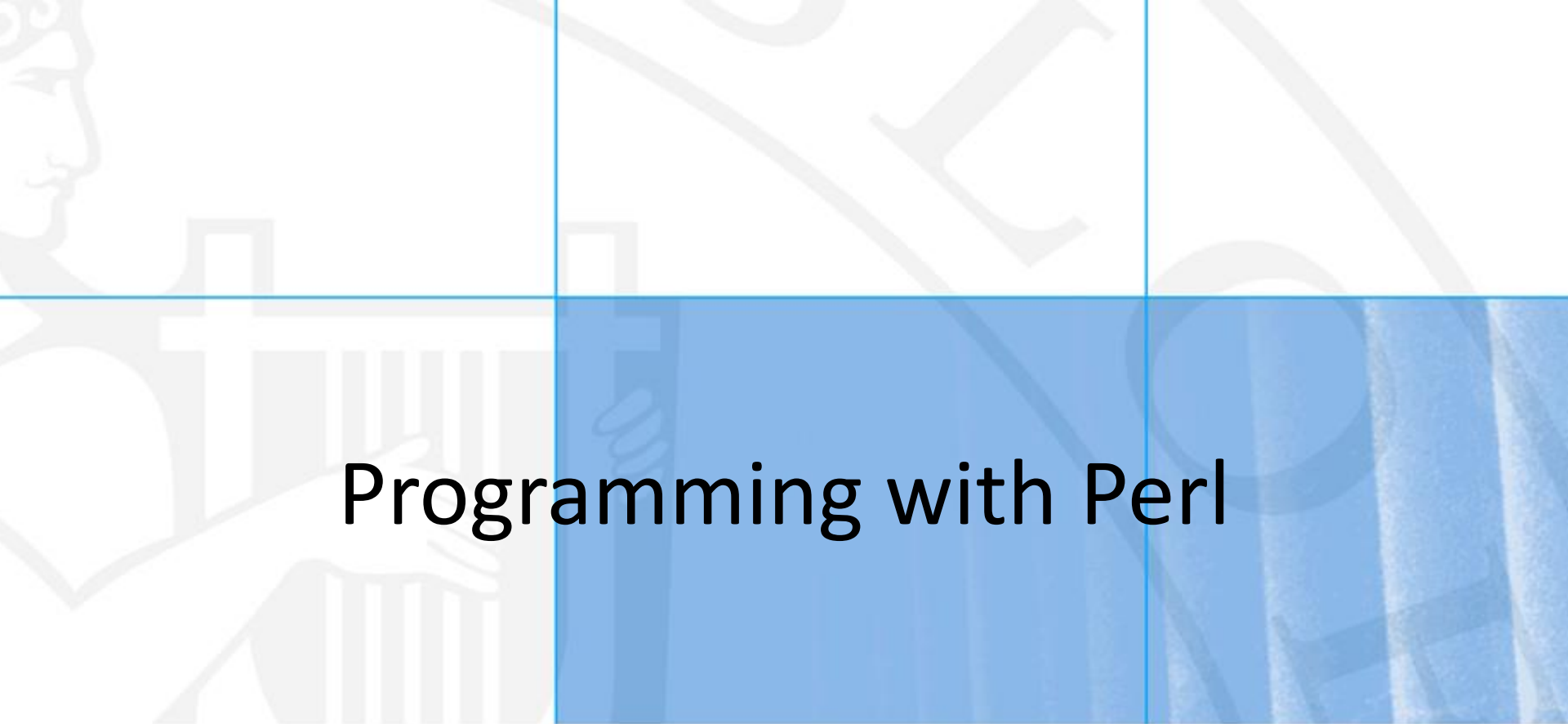
November 14-17, 2011

Monday – intro to Perl, Python and R

Tuesday – learn to use Titan

Wednesday – GPU, MPI and profiling

Thursday – about RCS and services

The background features a large, semi-transparent blue rectangle on the right side. To the left of this rectangle, there are faint, light-gray icons: a profile of a person's head, a cross, and a shield. The title 'Programming with Perl' is centered over the blue rectangle in a large, black, sans-serif font.

Programming with Perl

Katerina Michalickova

The Research Computing Services Group

SUF/USIT

November 14, 2011

Perl (from perltutorial.org)

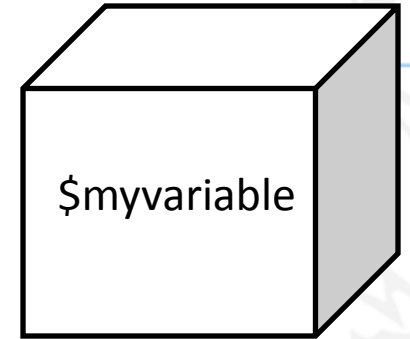
- Perl stands for Practical Extraction and Reporting Language. Perl is a general-purpose, interpreted programming language with a vast number of uses.
- Perl was invented by Larry Wall, a linguist working as a systems administrator at NASA in 1987.
- From the beginning, Perl was used to help processing reports faster and easier.
- Perl is very good and optimized for problems that handle 90% of text and 10% of other.
- Perl is best for short and small programs that can be entered and run on a single command line.

Topics

- Variables and operators
- Control structures
- Functions
- Input and output
- Regular expressions
- Hints and resources



What is a variable?

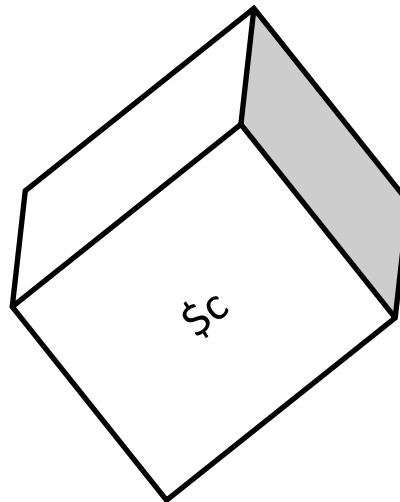
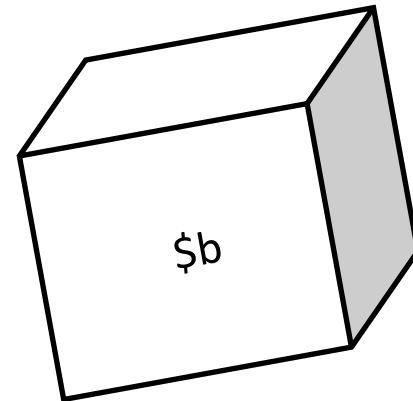
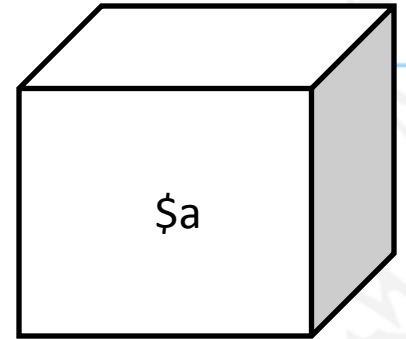


- Variable is a place to store your data
- It is a label that represents your data and you can recall its value during execution of your program
- You can change the value of the variable (hence it is a variable)

```
$myvariable = 3;    ..it is 3  
$myvariable = $myvariable + 2;    ..it is 5  
$myvariable = 1;    ..it is 1
```

Variables in Perl

- Scalars – numbers and strings
- Arrays – lists
- Hashes – associative lists
- (Objects)



\$.. Scalar variables - numbers

- Integers – 1,2,3...
- Floats – 1.1, 1.2...
- Non-decimal – 0xff, 0377, 0b11111111

`$mynumber = 2;`

`$mynumber = 5*6;`

`$mynumber = $a + $b;`

Numerical mathematical operators

+	Addition	$\$a + \$b$
-	Subtraction	$\$a - \$b$
*	Multiplication	$\$a * \$b$
/	Division	$\$a / \$b$
%	Modulus	$\$a \% \$b$
**	Exponentiation	$\$a ** \$b$
()	Grouping	$(\$a + \$b) * \$c$
++	Increment	$\$a++$
--	Decrement	$\$b--$

\$.. Scalar variables - strings

Any set of characters enclosed in quotes

```
$mystring="I like summer.";
```

```
$mystring="3or5_nonsense";
```

!Single or double quotes make a difference.

Special characters for strings

- `\n` ..newline “one line\nsecond line” is

```
one line
second line
```

- `\t` ..tab “one\ttwo” is

```
one      two
```

- BUT ‘one line\nsecond line’ is

```
one line\nsecond line
```

Strings operators

- Concatenation

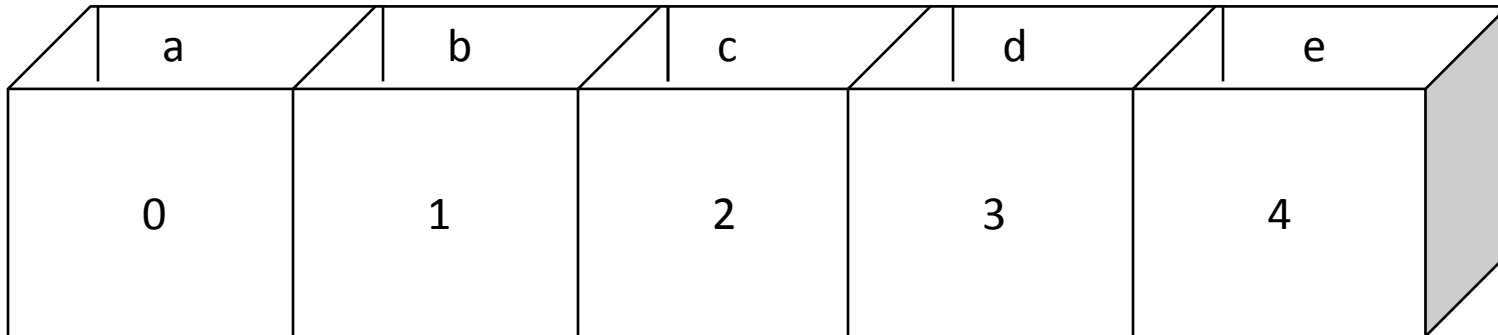
\$a = "honey"; \$b = "bee";

`$a.$b` is honeybee

- Length - `length($c)` is 8
- Substring - `substr($c,5,3)` is bee
- Index - `index($c,"b")` is 5

@.. Arrays

Arrays are ordered lists -
elements within arrays can be scalars



```
@myarray = ("a", "b", "c", "d", "e");  
$myarray[1] = "b";
```

! Note: when accessing a single element use a "\$" sign.

Array operators

- **qw** – easy way to declare array

@myarray = qw(a b c d e);

- **scalar** – length of an array

scalar(@myarray); returns 5

- **pop** – returns the last element of array

pop(@myarray); returns “e” and @myarray is now (“a”, “b”, “c”, “d”)

- **push** – adds an element on the end of the array

push(@myarray, “f”); @myarray is now (“a”, “b”, “c”, “d”, “f”)

- **shift** – returns the first element of the array

shift(@myarray); returns “a” and @myarray is now (“b”, “c”, “d”, “f”)

- **unshift** – adds an element to the beginning of the array

unshift (@myarray, “z”); @myarray is now (“z”, “b”, “c”, “d”, “f”)

- **reverse** – reverse order of elements

reverse(@myarray); @myarray is now (“f”, “d”, “c”, “b”, “z”)

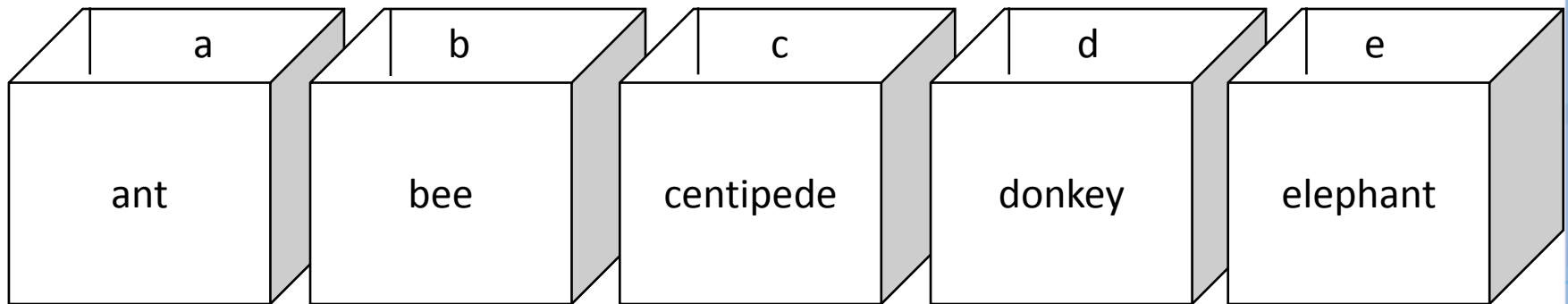
- **sort** – sort elements of the array

sort(@myarray); @myarray is now (“b”, “c”, “d”, “f”, “z”)

%.. Hashes

Hash is collection of pairs of keys and values.

Keys are unique strings that are used to index the hash.



key .. "ant" and value .. "a"

```
%myhash =
```

```
("ant"=>"a", "bee"=>"b", "centipede"=>"c", "donkey"=>"d", "elephant"=>"e");
```

```
$myhash{"bee"} returns "b"
```

```
$myhash{"elephant"} returns "e"
```

Hash functions

- **keys**

`keys(%myhash)`

returns ("ant", "bee", "centipede", "donkey", "elephant")

- **values**

`values(%myhash)`

returns ("a", "b", "c", "d", "e")

- **exists**

`exists $myhash{"ant"}` is true

`exists $myhash{"sloth"}` is false

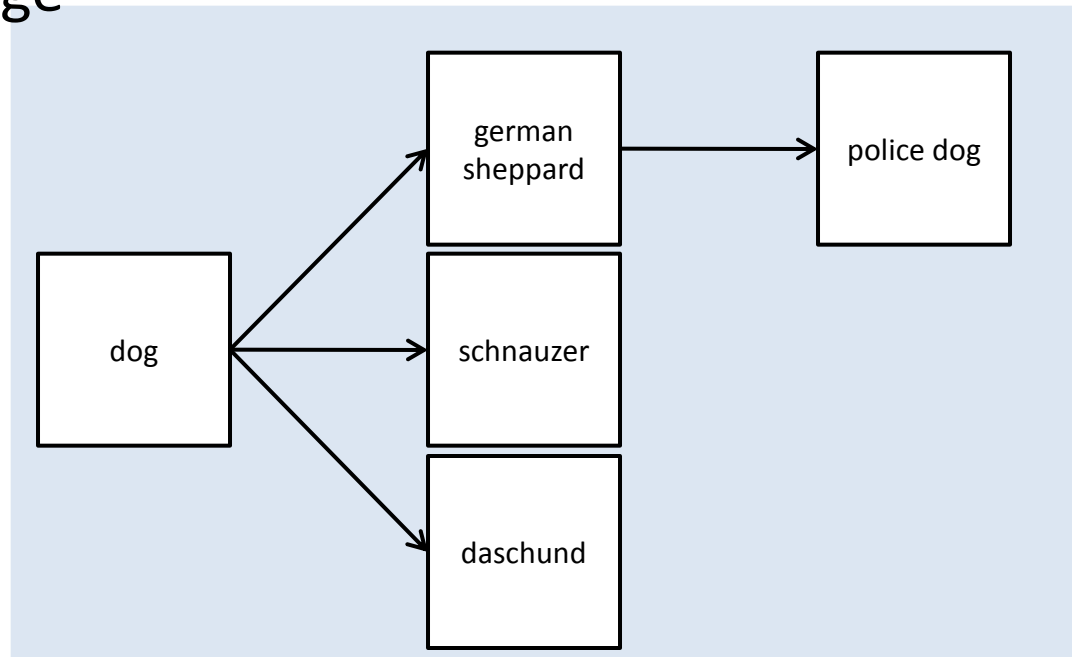
- **delete**

`delete $myhash{"elephant"}`

removes the pair "elephant"=>"e" from the hash

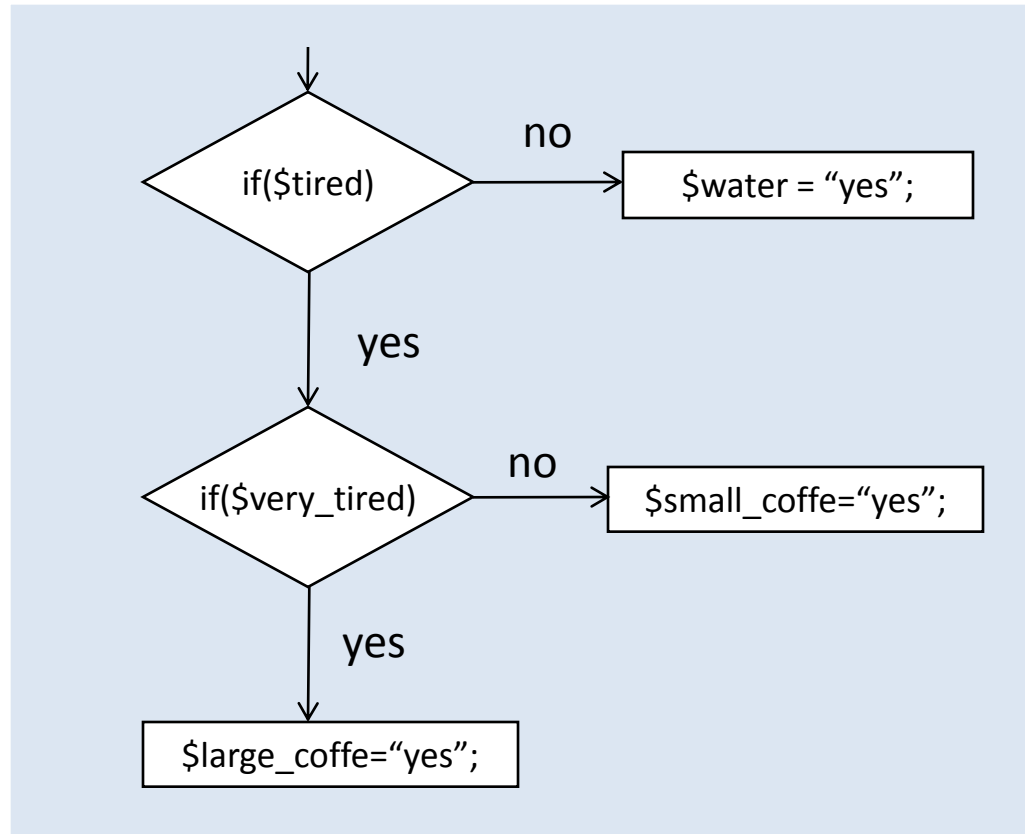
Objects

- you create an object to match your requirements
- objects have attached variables and methods
- objects can be abstracts or more specific
- if you have need for objects, consider other programming language



Control structures

- determine the order of operations within a program



True or false?

- To make decision, the program evaluates a conditional expression

```
$a = 4; $b = 5;  
$a == $b  is false
```

```
$a++;  
$a == $b  is true
```

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

! Make sure you use the right operators for numbers or strings, using an incorrect one might have unpredictable results.

Types of control structures

- **if** – makes one time decision
- **while** – repeats part of the program until some condition is not true any more
- **for** – repeats part of the program fixed number of times

while

```
my $time = 9;  
while($time < 17)  
{  
    $work = "yes";  
    $time = current_hour();  
}
```

! Beaware of infinite loops.

```
while($time < 17)  
{  
    $work = "yes";  
}
```

for

```
for ($time = 9, $time < 17, $time++)  
{  
    $work = "yes";  
}
```

Control structures for arrays and hashes

- **foreach** – iterates through an array

```
foreach $i (@myarray)
{
    $sum = $sum + $i;
} # sums up an array
```

- **each** – iterates through a hash

```
while (($key, $value) = each %myhash)
{
    $sum = $sum + $value;
} # sums up values in a hash
```

Functions/Subroutines

- separate logical units of a program
- make coding more manageable
- reusable
- each function contains a set of instructions that operate on pre-defined input and produce pre-defined output
- Perl contains hundreds of pre-defined functions readily available for use

Functions syntax

```
sub array_max {  
  my (@array) = @_  
  my $i = 0;  
  my $max = $array[0];  
  
  foreach $i (@array) {  
    if ($i > $max) {  
      $max = $i;  
    }  
  }  
  return $max;  
}
```

Functions syntax

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

```
# this program select maximum  
  from an array  
my @array = (2,4,3,6,8,9,1);  
my $mx = 0;
```

```
$mx = array_max(@array);
```

Input and output

- programs usually read and produce data
- program can be interactive or can read data from a file
- files are read or written to using a special kind of variables – filehandles
 - named in CAPITAL LETTERS
 - default for input is STDIN and output is STDOUT
 - open, read, close operations

Interactive input

```
print "Please enter your name.\n";  
$name = <STDIN>;  
chomp($name);
```

```
print "Please enter your age.\n";  
$age = <>;  
chomp($age);
```

Reading a file

```
open (IN, "myfile.txt") ;  
while (<IN>)  
{  
    $line = $_;  
    chomp ($line) ;  
}  
close (IN) ;
```

Writing to a file

```
open (OUT, ">myoutfile.txt");
open (IN, "myfile.txt");
while (<IN>)
{
    $line = $_;
    chomp($line);
    $line = check_line($line);
    print OUT "$line\n";
}
close (IN);
close (OUT);
```

Regular expression

- Perl is powerful in manipulating text using regular expressions
- regular expressions are used to find matching patterns in text
- patterns can be made extremely general
- syntax of regular expressions can be studied at <http://www.perl.com/doc/manual/html/pod/perlre.html>
- the following examples are taken from <http://www.cs.tut.fi/~jkorpela/perl/regexp.html>

Simple matching

```
my $greeting = "Hello World" ;

if ($greeting =~ /Hello/)
{
    print "Hello found.\n";
}
else
{
    print "Hello not found.\n";
}
```

Metacharacters

<code>^</code>	beginning of string
<code>\$</code>	end of string
<code>.</code>	any character except newline
<code>*</code>	match 0 or more times
<code>+</code>	match 1 or more times
<code>?</code>	match 0 or 1 times
<code> </code>	alternative
<code>()</code>	grouping
<code>[]</code>	set of characters
<code>{ }</code>	repetition modifier

Repetition

a^*	zero or more
a^+	one or more
$a^?$	zero or one
$a\{m\}$	exactly m
$a\{m,\}$	at least m
$a\{m,n\}$	at least m but at most n

Matching with “\”

\w	matches any <i>single</i> 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
\b	“word” boundary
\B	not a “word” boundary

Examples...

abc	abc (that exact character sequence, but anywhere in the string – regular expressions are greedy)
^abc	abc at the <i>beginning</i> of the string
abc\$	abc at the <i>end</i> of the string
a b	either of a and b
^abc abc\$	the string abc at the beginning or at the end of the string
ab{2,4}c	an a followed by two, three or four b's followed by a c
ab{2,}c	an a followed by at least two b's followed by a c
ab*c	an a followed by any number (zero or more) of b's followed by a c
ab+c	an a followed by one or more b's followed by a c
ab?c	an a followed by an optional b followed by a c (abc or ac)
a.c	an a followed by any single character (but not a newline) followed by a c
a\\.c	a.c exactly (“\” is an escape character)

More examples...

<code>[abc]</code>	any one of a, b and c
<code>[Aa]bc</code>	either of Abc and abc
<code>[abc]+</code>	any (nonempty) string of a's, b's and c's (such as a, abba, acbabacaaa)
<code>[^abc]+</code>	any (nonempty) string which does <i>not</i> contain any of a, b and c (such as defg)
<code>\d\d</code>	any two decimal digits, such as 42; same as <code>\d{2}</code>
<code>\w+</code>	a “word”, a nonempty sequence of alphanumeric characters (and underscores), such as foo and 12bar8 and foo_1
<code>a\s*bc</code>	the strings a and bc optionally separated by any amount of white space (spaces, tabs, newlines)
<code>abc\b</code>	abc when followed by a word boundary (e.g. in abc! but not in abcd)
<code>perl\b</code>	perl when <i>not</i> followed by a word boundary (e.g. in perlert but not in perl stuff)

Substitution

\$string = "This apple is mine, this orange is yours
and this pear is his.";

s/this/that/	"This apple is mine, that orange is yours and this pear is his."
s/this/that/g	"This apple is mine, that orange is yours and that pear is his."
s/this/that/gi	" That apple is mine, that orange is yours and that pear is his."

Split function

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

my $data = "Oslo,Blindern,IFI2,Prolog";
my @values = split(/,/, $data);
foreach my $val (@values)
{
    print "$val\n";
}
```

Hints

- use warnings
- use strict
- undef and defined
 - before assigning a value or using “my”, a new variable has a status “undef” and Perl ignores it
 - to test the status use “defined(\$myvariable)”
- be careful about a numerical versus a string context

Resources

- Perl documents <http://perl.org> or <http://perldoc.perl.org>
- 22 500 modules – wealth of written code; search <http://cpan.org>
- this lecture was inspired by the Canadian bioinformatics workshops material, see the originals at <http://donaldson.uio.no/wiki/MBV3070>

Thank you

```
my $text = "It is time for Perl.";
if ($text =~ /Perl/)
{
    $text =~ s/Perl/a break/;
    print "$text\n";
}
```

