

# Perl for Biologists

## Session 4

March 25, 2015

### *Arrays and lists*

Jaroslav Pillardy

# if statement

```
if(condition1)
{
    statement;
}
elseif(condition2)
{
    statement;
}
else
{
    statement;
}
```

```
if($n>6)
{
    print "n>6\n";
}
elseif($n==5)
{
    print "n=5\n";
}
elseif($n==6)
{
    print "n=6\n";
}
else
{
    print "n<5\n";
}
```

# while loop

```
while (condition)
{
    statement;
    optional → if (condition1) { next; }
               statement;
               → if (condition2) { last; }
               statement;
}
```

**next;**                      #moves to the next iteration

**last;**                      #exits the loop

example1.pl : generate random numbers 0..9 until a number is  $\geq 5$  (problem in script!)

## while loop

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

$n=rand(10);
print "start $n\n";

while ($n<9)
{
    if ($n<5)
    {
        print " less than 5 $n\n";
        next;
    }
    print "main loop $n\n";
    $n=rand(10);
}
```

example2.pl : generate random numbers 0..9 until a number is  $\geq 5$  (correct)

## while loop

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

$n=rand(10);
print "start $n\n";

while ($n<9)
{
    if ($n<5)
    {
        print " less than 5 $n\n";
        $n=rand(10);
        next;
    }
    print "main loop $n\n";
    $n=rand(10);
}
```

# logical operators

**and**

& &

`$n > 5 & & $n < 10`

**or**

| |

`$n < 5 || $n > 10`

**not**

!

`! ($n > 5 & & $n < 10)`

# for loop

```
for (init_statement; test_statement; increment;)
{
    statement;
    if (condition1) { next; }
    statement;
    if (condition2) { last; }
    statement;
}
```

optional

**next**;            #moves to the next iteration

**last**;            #exits the loop

## example3.pl : sum all odd or even numbers less then predefined value

# for loop

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

print "odd or even? ";
$choice = <STDIN>;
chomp($choice);
if(lc($choice) ne "odd" && lc($choice) ne "even")
{
    print "ERROR: wrong choice '$choice'\n";
    exit;
}
print "sum up to what number (int)? ";
$nnn = <STDIN>;
chomp($nnn);
if(int(1*$nnn) != $nnn)
{
    print "ERROR: wrong int number $nnn\n";
    exit;
}

$sum = 0;
$rem = 0;
if(lc($choice) eq "odd"){$rem = 1;}
for($i=1; $i<=$nnn; $i++)
{
    if($i % 2 == $rem){$sum += $i;}
}
print "Sum of all $choice int up to $nnn is $sum\n";
```



## example3.pl : sum all odd or even numbers less then predefined value

# for loop

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

print "odd or even? ";
$choice = <STDIN>;
chomp($choice);
if (lc($choice) ne "odd" && lc($choice) ne "even")
{
    print "ERROR: wrong choice '$choice'\n";
    exit;
}
print "sum up to what number (int)? ";
$nnn = <STDIN>;
chomp($nnn);
if (int(1*$nnn) != $nnn)
{
    print "ERROR: wrong int number $nnn\n";
    exit;
}

$sum = 0;
$rem = 0;
if (lc($choice) eq "odd") { $rem = 1; }
for ($i=1; $i<=$nnn; $i++)
{
    if ($i % 2 == $rem) { $sum += $i; }
}
print "Sum of all $choice int up to $nnn is $sum\n";
```

### Challenge

Add option "all" to the script

## printf/sprintf formats

%17.15f	floating point number, total 17 digits, 15 after dot
%17.10e	floating point number with exponent, 17 digits total 10 after dot
%10d	integer, total length 10 digits
%010d	integer, total length 10 digits, pad with zeros on the left
%s	string
%-10s	string, total length 10 chars, align left

```
$svar = sprintf("full length number %17.15f while short is %d", 2, 3);  
print "$svar\n";  
  
printf "full length number %17.15f while short is %d", 2, 3 ;
```

## Session 3 Exercises Review

1. Modify the program from script6a.pl to run it longer (more iterations). Try to run for several different numbers of iterations (increase each time by at least an order of magnitude). Is our  $\pi$  number converging to the real  $\pi$ ? If yes, what does it say about our computer? If no, what is the problem?

[/home/jarekp/perl\\_03/exercise1.pl](/home/jarekp/perl_03/exercise1.pl)

After 1_000_000_000 iterations pi is	3.1416316200000001	1.000012403393600
--------------------------------------	--------------------	-------------------

After 10_000_000_000 iterations pi is	3.1415767500000000	0.999994937730144
---------------------------------------	--------------------	-------------------

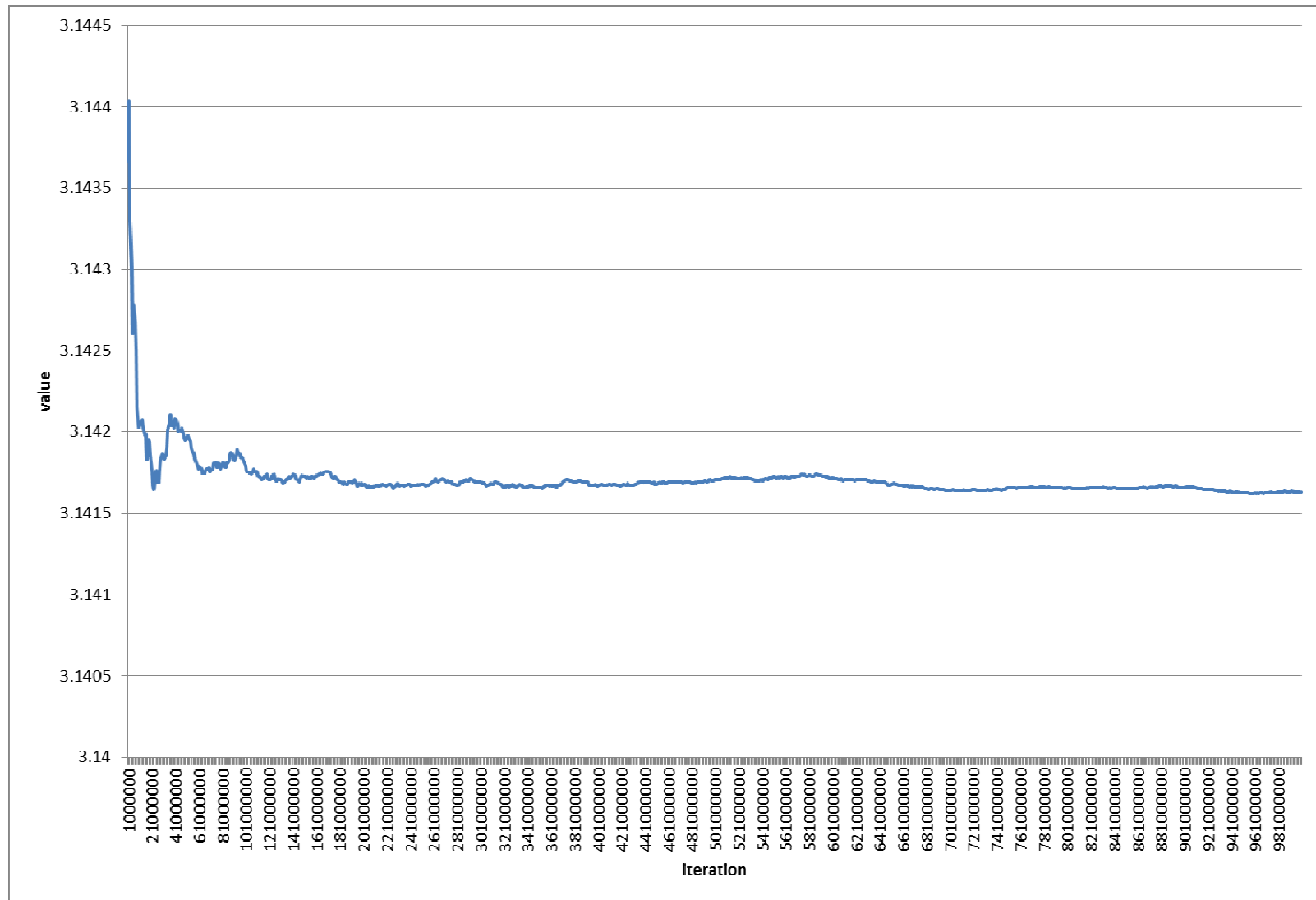
After 100_000_000_000 iterations pi is	3.1415895820399999	0.999999022295333
--	--------------------	-------------------

Real pi is	3.1415926535897932	1.0000000000000000
------------	--------------------	--------------------

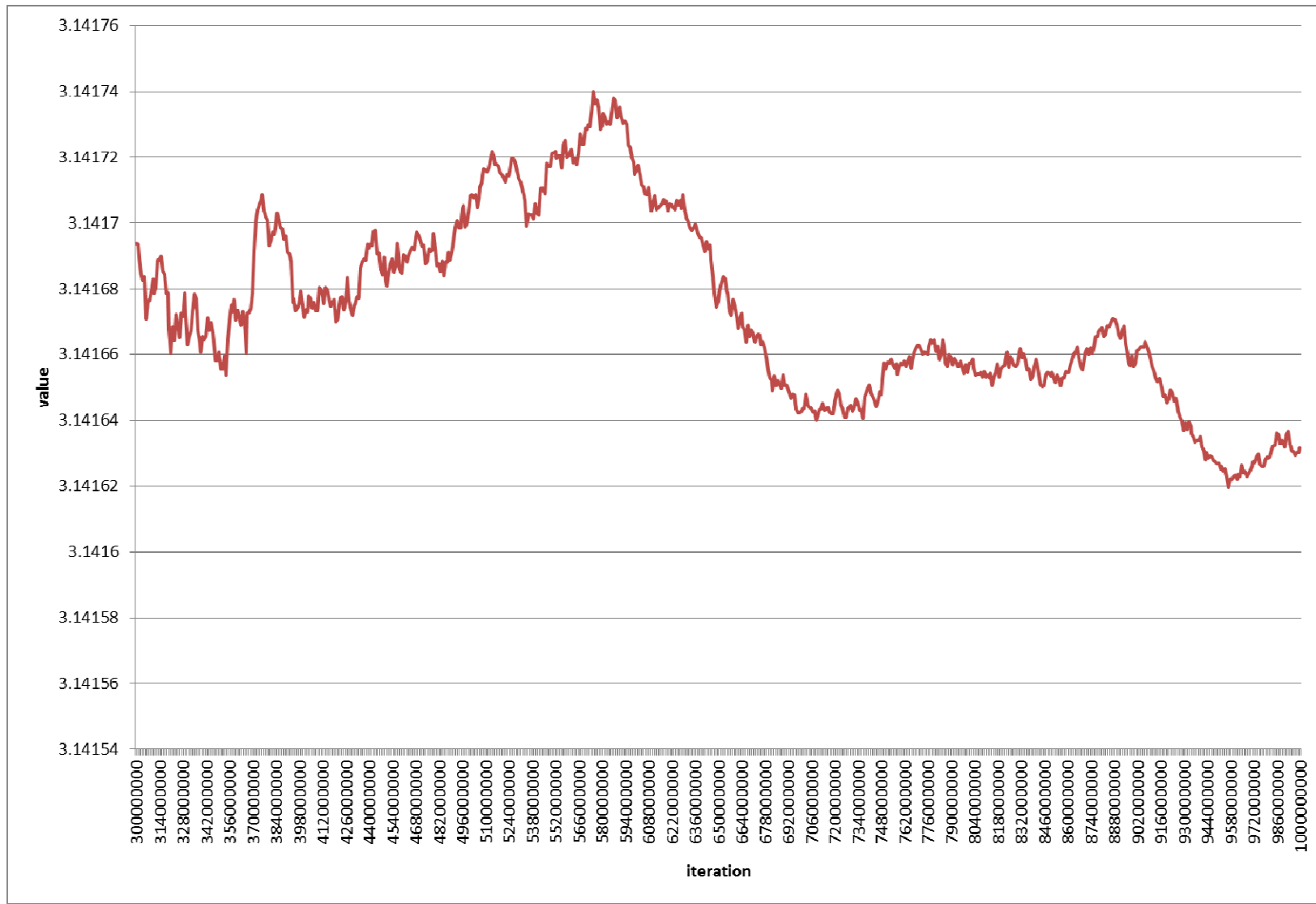
uniform distribution:

After 999_950_884 iterations pi is	3.1415931904911440	1.000000170901007
------------------------------------	--------------------	-------------------

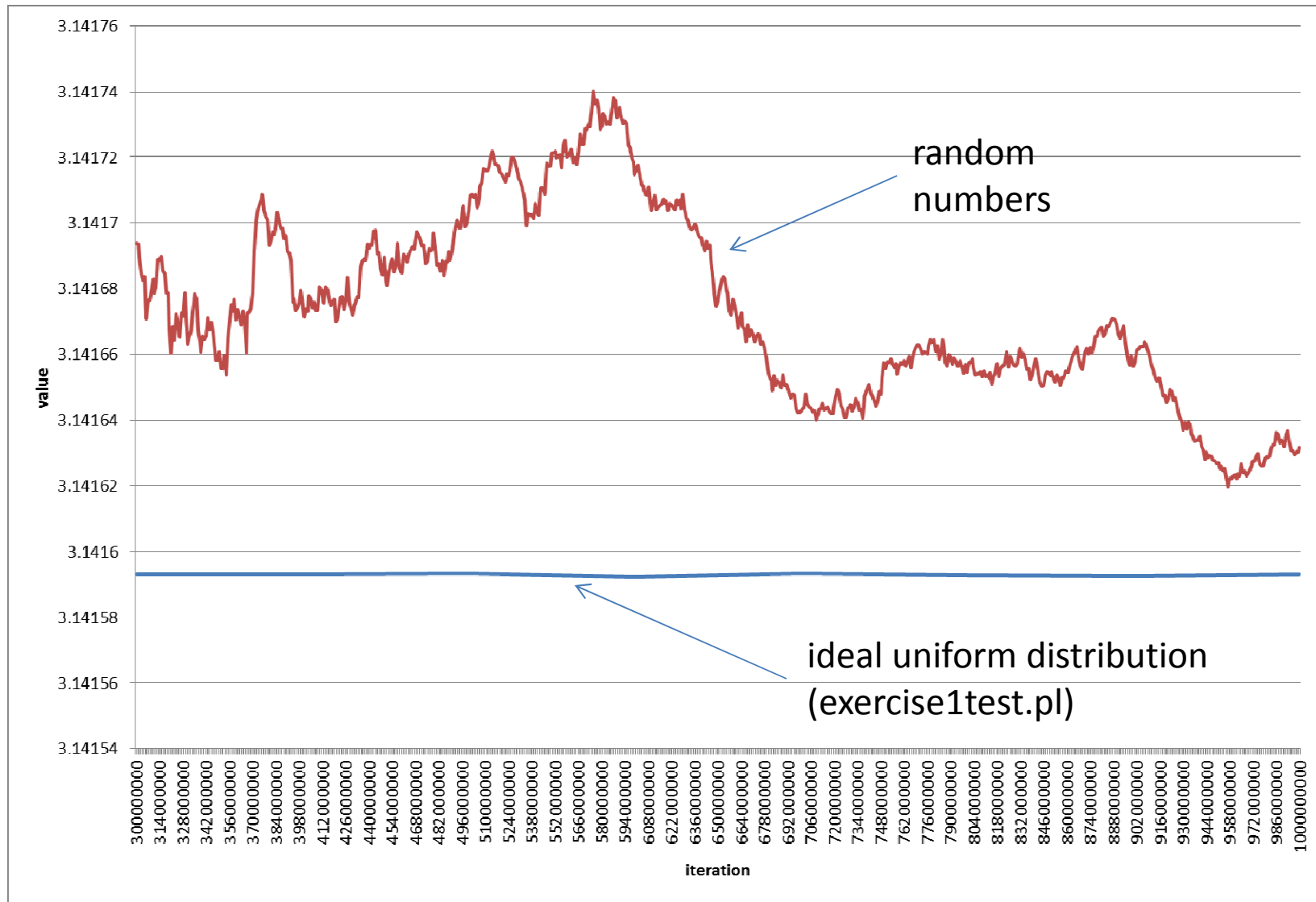
## Session 3 Exercises Review : Exercise 1 : 1,000,000,000 iterations



## Session 3 Exercises Review : Exercise 1 : 1,000,000,000 iterations (tail)



## Session 3 Exercises Review : Exercise 1 : 1,000,000,000 iterations (tail)



## Session 3 Exercises Review : Exercise 1 : 10,000,000,000 iterations



## Session 3 Exercises Review

2. Change script4.pl so it doesn't use *last* statement at all.

`/home/jarekp/perl_03/exercise2.pl`

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

```
#finding out the accuracy in Perl
```

```
$n1 = 1;
```

```
$n2 = 1;
```

```
while ($n1 + $n2 != $n1)
{
```

```
    print "$n1 + $n2 DIFFERENT than $n1\n";
```

```
    $n2 = $n2 / 10;
```

```
}
```

```
print "$n1 + $n2 SAME as $n1\n";
```

```
print "Perl accuracy reached\n";
```



## Session 3 Exercises Review

3. Using `rand()` and `srand()` functions produce 4.1 kb long random DNA sequence with AT content propensity of 75%, store it in a variable, then print it out to STDERR stream in fasta format. Run the program and redirect STDERR to a file `randomdna.fa`.

Hint 1: For each bp use `rand()` twice, first deciding if it will be GC or AT with 75% probability, then choosing G/C or A/T with 50% probability (two *if*).

Hint 2: Generate the sequence by adding 1 bp to the string variable in a *for* loop.

`/home/jarekp/perl_03/exercise3.pl` (minimum version)

`/home/jarekp/perl_03/exercise3a.pl` (nice version)

List: ordered collection of scalar values

A list:

(1, 5, 8, 33, 23, 11, 1, 44)

each element has assigned *index* starting from 0

(1, 5, "a", 77, "abcd", 99)

lists can contain mixed types

## Lists can be declared in various ways

explicit: `(1, 5, 8, 33, 23, 11, 1, 44)`

range: `1..9;`  
same as `(1, 2, 3, 4, 5, 6, 7, 8, 9)`

quoted word: `qw(jarek pillardy perl 2013)`  
same as `('jarek', 'pillardy', 'perl', '2013')`

quoted word 1: `qw*jarek pillardy perl 2013*`  
same as `('jarek', 'pillardy', 'perl', '2013')`

list delimiter (any character)

note SINGLE QUOTATIONS

In **qw()** words are delimited by space, multiple spaces are compressed to one

Array: a variable that contains a list

A variable:

```
@arvar = (1, 5, 8, 33, 23, 11, 1, 44);
```

each element has assigned *index* starting from 0

```
@arvar = (1, 5, "a", 77, "abcd", 99);
```

arrays can contain mixed types

```
@arvar = 1..55;
```

any valid list declaration is OK to assign to an array

Array: a variable that contains a list


```
@arvar = (1, 5, "a", 77, "abcd", 99);
```



@ character means we reference to an array variable as a whole

array elements are scalar variables and can be accessed with index

\$ character means we reference to a scalar variable – a single element of @arvar, index starts from 0



```
print $arvar[0];           #will print      1
print $arvar[4];           #will print      abcd

print $arvar[5];           #will print      99

$i=3;
print $arvar[$i];          #will print      77
```

## script1.pl

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

@var = (1, 2, 3);
for ($i=3; $i<=10; $i++)
{
    $var[$i] = rand(10);
}
for ($i=0; $i<=10; $i++)
{
    printf "%5.3f ", $var[$i];
}
print "\n";
```


All scripts for this session can be copied from  
/home/jarekp/perl\_04  
in this case /home/jarekp/perl\_04/script1.pl  
>cp /home/jarekp/perl\_04/script1.pl .  
copies this script to your current directory

## script1.pl

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

```
@var = (1, 2, 3);  
for($i=3; $i<=10; $i++)  
{  
    $var[$i] = rand(10);  
}  
for($i=0; $i<=10; $i++)  
{  
    printf "%5.3f ", $var[$i];  
}  
print "\n";
```

an array can be expanded just  
by adding elements to it

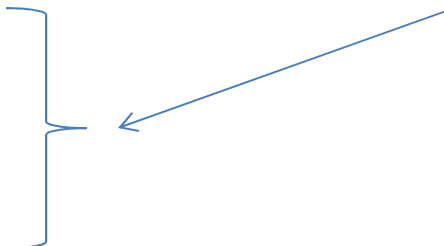


```
[jarekp@cbsum1c2b014 perl_04]$ perl script1.pl  
1.000 2.000 3.000 1.331 5.585 7.717 4.804 5.715 2.986 2.731 3.388  
[jarekp@cbsum1c2b014 perl_04]$
```

## script2.pl

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

```
$var[0] = 1;  
$var[1] = 2;  
$var[4] = 5;  
$var[8] = 9;
```



it is possible to create an array  
just by creating its elements

```
print "Array length is " . ($#var + 1) . "\n";
```



```
for($i=0; $i<=$#var; $i++)  
{  
    printf "%5.3f\n", $var[$i];  
}
```

index of the last  
element in an array



script2.pl

```
[jarekp@cbsum1c2b014 perl_04]$ perl script2.pl
```

```
Array length is 9
```

```
1.000
```

```
2.000
```

```
0.000
```

```
0.000
```

```
5.000
```

```
0.000
```

```
0.000
```

```
0.000
```

```
9.000
```

```
[jarekp@cbsum1c2b014 perl_04]$
```

## script2a.pl

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

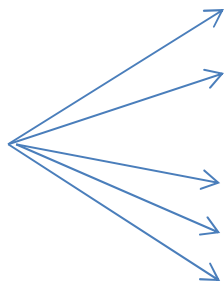
$var[0] = 1;
$var[1] = 2;
$var[4] = 5;
$var[8] = 9;

print "Array length is " . ($#var + 1) . "\n";

for ($i=0; $i<=$#var; $i++)
{
    printf "%5.3f '%s'\n", $var[$i], $var[$i];
}
```

script2.pl

undef



```
[jarekp@cbsum1c2b014 perl_04]$ perl script2a.pl
Array length is 9
1.000 '1'
2.000 '2'
0.000 "
0.000 "
5.000 '5'
0.000 "
0.000 "
0.000 "
9.000 '9'
[jarekp@cbsum1c2b014 perl_04]$
```

All the omitted array elements are assigned “undef” value, but they do exist

script3.pl

## ARGV array

Command line arguments are passed into a Perl script with a special array **ARGV**

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

print "You have entered " . $#ARGV+1 . " parameters\n";

print "here they are:\n";

for($i=0; $i<=$#ARGV; $i++)
{
    print $i . " " . $ARGV[$i] . "\n";
}
```

## script3.pl

```
[jarekp@cbsum1c2b014 perl_04]$ perl script3.pl p1 22 -p3 abc def
```

```
You have entered 5 parameters
```

```
here they are:
```

```
0 p1
```

```
1 22
```

```
2 -p3
```

```
3 abc
```

```
4 def
```

```
[jarekp@cbsum1c2b014 perl_04]$ perl script3.pl p1 22 -p3 abc\ def
```

```
You have entered 4 parameters
```

```
here they are:
```

```
0 p1
```

```
1 22
```

```
2 -p3
```

```
3 abc def
```

```
[jarekp@cbsum1c2b014 perl_04]$ perl script3.pl
```

```
You have entered 0 parameters
```

```
here they are:
```

```
[jarekp@cbsum1c2b014 perl_04]$
```

## Array and list assignment

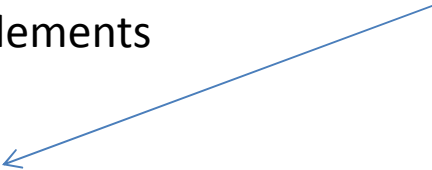
It is possible to assign entire arrays and lists

```
@arr = (1..22, 55, 66..77);  
@arr1 = @arr;
```

it is a 4 element list, the list elements are variables, assigning array to list assigns its elements to variables (extra elements of @arr2 are ignored)

or you can assign arrays to elements

```
@arr2 = 11..22;  
($var1, $var2, $var3, $var4) = @arr2;  
print "$var1 $var2 $var3 $var4"           #will print  11 12 13 14
```



## Array and list assignment

or you can assign ranges of arrays to elements

```
@arr2 = 11..22;
```

```
($var1, $var2, $var3, $var4) = (@arr2[3..5], $arr2[7]);
```

```
print "$var1 $var2 $var3 $var4"           #will print 14 15 16 18
```

## List operators

<b>push</b> (@arr, \$val)	adds value of \$val as a new element at the end of array @arr
\$val = <b>pop</b> (@arr)	removes last element of @arr and returns it to \$val
\$val= <b>shift</b> (@arr)	removes the first element of @arr and returns it to \$val
<b>unshift</b> (@arr, \$val)	adds value of \$val as the first element of array @arr (the previous first element will become the second)
@arr1= <b>reverse</b> (@arr)	reverses order of elements of @arr and returns it to @arr1
@arr1= <b>sort</b> @arr	sorts elements of @arr and returns sorted array to @arr1 (the sort is based on ASCII codes)
@arr1= <b>sort</b> {\$a<=>\$b} @arr	sorts elements of @arr and returns sorted array to @arr1 (the sort is based on numerical values)



## List operators

`@arr1=splice (@arr, $n1)`

removes everything after index \$n1 from @arr,  
returns it to @arr1

`@arr1=splice (@arr, $n1, $n2)`

removes everything between indexes \$n1 and \$n2  
from @arr, returns it to @arr1

`@arr1=splice (@arr, $n1, $n2, @replacement)`

removes everything between indexes \$n1 and \$n2 from  
@arr, returns it to @arr1, then inserts @replacement as a  
replacement of the removed part (may be different length)

Everything that can be done using list operators can be also done explicitly using indexes, assignments and loops

```
@arr = 11..22;
```

```
push(@arr, 33);
```

```
$arr[$#arr+1] = 33;
```

} same thing

```
$var = pop(@arr);
```

```
$var = $arr[$#arr] ;
```

```
$arr[$#arr] = undef;
```

```
$#arr--;
```

} same thing

... but **push** and **pop** are **MUCH FASTER**

converting string to array

```
@arr = split /pattern/, $str
```

string is split into array elements wherever *pattern* is found

script3a.pl

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

@arr = split / /, "jarek pillardy perl 2013";
for($i=0; $i<=$#arr; $i++)
{
    print "$i '$arr[$i]'\n";
}
print "\n";

@arr = split / +/, "jarek pillardy perl 2013";
for($i=0; $i<=$#arr; $i++)
{
    print "$i '$arr[$i]'\n";
}
print "\n";

@arr = split / p/, "jarek pillardy perl 2013";
for($i=0; $i<=$#arr; $i++)
{
    print "$i '$arr[$i]'\n";
}
print "\n";
```

## script3a.pl

```
[jarekp@cbsum1c2b014 perl_04]$ perl script3a.pl
```

```
0 'jarek'
```

```
1 ''
```

```
2 'pillardy'
```

```
3 'perl'
```

```
4 ''
```

```
5 '2013'
```

```
0 'jarek'
```

```
1 'pillardy'
```

```
2 'perl'
```

```
3 '2013'
```

```
0 'jarek '
```

```
1 'illardy'
```

```
2 'erl 2013'
```

```
[jarekp@cbsum1c2b014 perl_04]$
```

## foreach loop

A special version of for loop going over ALL elements of an array

```
foreach $var (@arr)
{
    print "$var\n";
}
```

The code above will print out each element of an array @arr

## default variable \$\_

When a variable is not specified in Perl code, the default variable \$\_ is used.

```
foreach (@arr)
{
    print "$_\n";
}
```

The code above will print out each element of an array @arr

## default variable \$\_

When a variable is not specified in Perl code, the default variable \$\_ is used.

```
foreach (@arr)
{
    print;
}
```

The code above will print out each element of an array @arr



default variable \$\_

Finally built-in array to string conversion can be used. Last two examples print the array in ONE line.

```
print "@arr";
```

The code above will print out each element of an array @arr

script4.pl

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

@arr = (1, 2, 3);
for($i=3; $i<=10; $i++)
{
    $arr[$i] = rand(10);
}

foreach $var (@arr)
{
    printf "%5.3f ", $var;
}
print "\n";
```

## list and scalar context

As usual in Perl any variable can be treated differently based on the context – as previously seen with strings and numbers

Now any variable can be treated differently in an scalar context or a list (array) context

## script5.pl

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

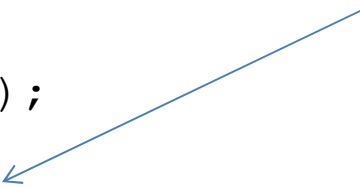
@arr = (1, 2, 3);
for ($i=3; $i<=5; $i++)
{
    $arr[$i] = rand(10);
}
print "Our array is:\n@arr\n";

$arr1 = sort(@arr); #array context
print "@arr1\n";

print @arr1 . "\n"; #scalar context for @arr1

$nnn = @arr + 2;    #scalar context
print "$nnn\n";
```

converts array  
into string, similar  
to as it was with  
numbers



script5.pl

```
[jarekp@cbsum1c2b014 perl_04]$ perl script5.pl
Our array is:
1 2 3 6.80320264612423 9.9025302016841 0.655239057179067
0.655239057179067 1 2 3 6.80320264612423 9.9025302016841
6
8
[jarekp@cbsum1c2b014 perl_04]$
```

## script6.pl

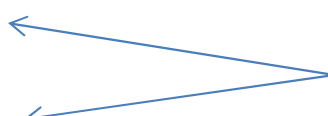
```
#!/usr/local/bin/perl
```

```
print "please enter input lines, end with CTRL+D\n\n";
```

```
$first_line = <STDIN>;
```

```
@other_lines = <STDIN>;
```

<STDIN> acts as a single string or an array of strings, depending on context



```
print "\nfirst line of input was:\n$first_line";
```

```
print "There were " . ($#other_lines + 1) . " more lines of input\n";
```

```
print "Here they are:\n";
```

```
foreach $line (@other_lines)
```

```
{
```

```
    print $line;
```

```
}
```

script6.pl

```
[jarekp@cbsum1c2b014 perl_04]$ perl script6.pl  
please enter input lines, end with CTRL+D
```

```
line 1
```

```
line 2
```

```
line 3
```

```
first line of input was:
```

```
line 1
```

```
There were 2 more lines of input
```

```
Here they are:
```

```
line 2
```

```
line 3
```

```
[jarekp@cbsum1c2b014 perl_04]$
```

## Exercise

1. Modify the program from session 3 exercise 3 (random DNA sequence) to produce a random DNA sequence of 5 Mb (originally 4.1kb), store the sequence string in a variable and discard the rest of the program (the part printing it to STDERR).
2. Take the random DNA string obtained in step 1 and apply *in silico* restriction enzyme by cutting the DNA at each occurrence of the pattern of "ATGCAT" . The easiest way to do it is to use `split` function with ATGCAT as the splitting pattern, store the DNA fragments in an array.
3. Create a new array containing lengths of the strings from the array obtained in step 2 (`length($str)` function returns the length of a string `$str`). Unlike the real restriction enzyme, `split` function removes ATGCAT pattern, to correct for this you need to add 6 to each middle fragment, 1 to first and 5 to the last (simulating cutting A{cut}TGCAT).
4. Sort the lengths array. Remember that `sort` function by default sorts in string context (in alphabetical order i.e. 123 comes before 99), you need to provide sorting function to sort numerically :  

```
sort {$a <=> $b} @array
```

  
Print out the sorted fragment lengths.

continued on the next page



## Exercise cont.

4. Run the program and redirect output to file “histogram.txt”. Transfer the file to your laptop, import to MS Excel and use histogram tool to plot it out, use 2kb as bin width.

Hint: You need to install Analysis ToolPak add-in to create the histogram in Excel, you can follow the step-by-step instructions from this website <http://support.microsoft.com/kb/214269>

5. Run the program for different GC content (originally 75%) and compare the results.