Perl for Biologists

Session 3

March 18, 2015

Control flow statements

Jaroslaw Pillardy

Session 2 Exercises Review

1. In a Perl program create a string representing a 54 bp DNA strand consisting of 6 repeats, store it in a variable. Create another variable containing the above DNA reversed. Create the third variable storing a subsequence of the original sequence from position 31 to position 47. Print all three.
   Hint: Use string functions and operators to create strings from a repeat.

   /home/jarekp/perl_02/exercise1.pl

2. Use perldoc to find out how rand() and srand() functions work. Write a Perl program that produces a 17 character string composed of random lower case letters, store it in a variable and print it out. Run the program several times and compare the results.
   Hint: use chr(), int() functions and ASCII table.

   /home/jarekp/perl_02/exercise2.pl

Simple Line Input

Each program has three default input/output objects associated with it

- *input stream* – usually keyboard input: STDIN
- *output stream* – usually screen: STDOUT
- *error stream* – usually screen: STDERR

script1.pl

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

$svar = <STDIN>;          #get one line of std input

print STDOUT "1. [$svar]\n";

chomp($svar);

print STDERR "2. [$svar]\n";

print "3. [$svar]\n";
```

All scripts for this session can be copied from

/home/jarekp/perl_03

in this case /home/jarekp/perl_03/script1.pl

>cp /home/jarekp/perl_03/script1.pl .

copies this script to your current directory

script1.pl

```
#!/usr/local/bin/perl
$svar = <STDIN>;          #get one line of std input
print STDOUT "1. [$svar]\n";
chomp($svar);
print STDERR "2. [$svar]\n";
print "3. [$svar]\n";
```

Linux help: redirecting input and output

```
./script1.pl
```

input from keyboard, output and error to screen

```
./script1.pl  1> out 2> err
```

input from keyboard, output to file out, error to file err, files overwritten

```
./script1.pl >& out.all
```

input from keyboard, output and error to file out.all, file overwritten

```
cat input.txt | ./script1.pl 1>> out  2>> err
```

input from file input.txt, output <u>appended</u> to file out, error <u>appended</u> to file err

symbol `|` is used to connect output from one program (`cat` in the example above) and input of another program (`./script1.pl`), it is called a *pipe*

script1.pl

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

$svar = <STDIN>;

print STDOUT "1. [$svar]\n";

chomp($svar);

print STDERR "2. [$svar]\n";

print "3. [$svar]\n";
```

```
[jarekp@cbsum1c2b014 perl_03]$ perl script1.pl
one line input
1. [one line input
]
2. [one line input]
3. [one line input]
[jarekp@cbsum1c2b014 perl_03]$ perl script1.pl 1> out 2> err
another line input
[jarekp@cbsum1c2b014 perl_03]$ cat out
1. [another line input
]
3. [another line input]
[jarekp@cbsum1c2b014 perl_03]$ cat err
2. [another line input]
[jarekp@cbsum1c2b014 perl_03]$ perl script1.pl >& out.all
yet another one
[jarekp@cbsum1c2b014 perl_03]$ cat out.all
2. [yet another one]
1. [yet another one
]
3. [yet another one]
[jarekp@cbsum1c2b014 perl_03]$ cat input.txt
stored input line
[jarekp@cbsum1c2b014 perl_03]$ cat input.txt | perl script1.pl
1. [stored input line
]
2. [stored input line]
3. [stored input line]
[jarekp@cbsum1c2b014 perl_03]$
```

Control flow statements

Statements to control the sequence of statements executed in the program.

Logical: (if)

Repetitive: (loops)

script2.pl

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

$var = <STDIN>;
chomp($var);

if($var > 5)
{
        print "input is greater than 5\n";
}
elsif($var == 5)
{
        print "input is equal to 5\n";
}
else
{
        print "input is less than 5\n";
}
print "input is $var";
```

script2.pl

beginning **if** statement with condition

subsequent **else if** statement(s) with condition (optional)

ending **else** statement (optional)

code blocks

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

$var = <STDIN>;
chomp($var);

if($var > 5)
{
        print "input is greater than 5\n";
}
elsif($var == 5)
{
        print "input is equal to 5\n";
}
else
{
        print "input is less than 5\n";
}
print "input is $var";
```

Code block and its scope

Code block is a separate part of program enclosed in { }

It acts as if it is a single statement

It is a way to group statements into one entity

Comparison operators

Numerical

==	equal
>	greater than
<	less then
>=	greater or equal
<=	less or equal
!=	not equal

String

eq	equal
lt	less than
gt	greater than
le	less then or equal
ge	greater than or equal
ne	not equal

compares ASCII code of a first different character:
"abd" gt "abc"   is true

Boolean values

The result of comparison is a Boolean value (true or false)

```
$res = "abd" gt "abc";
```

In fact `$res` is not storing anything special – it is just a 0 or 1 number.

In general, in any logical statement:

number 0 means *false*, any other number means *true*

empty string means *false*, any other string means *true*

`undef` is always *false*

ABOUT == AND =

```
$var1 = 5;
$var2 = 15;

if($var1 == $var2)       #obviously FALSE, will NOT print TRUE
{
   print "TRUE";
}

if($var1 = $var2)        #in LOGICAL context it is TRUE, will
{                        #print TRUE
   print "TRUE";
}
```

assign $var1 the value of $var2, the result of which is 15 (new value of $var1), number 15 means TRUE

script3.pl

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

print "type value 1: ";
$val1 = <STDIN>;
chomp($val1);
print "type value 2: ";
$val2 = <STDIN>;
chomp($val2);

if($val1>$val2){print "NUM: $val1 > $val2\n";}
elsif($val1==$val2){print "NUM: $val1 == $val2\n";}
else {print "NUM: $val1 < $val2\n";}

if($val1 gt $val2){print "STR: $val1 gt $val2\n";}
elsif($val1 eq $val2){print "STR: $val1 eq $val2\n";}
else {print "STR: $val1 lt $val2\n";}
```

while loop

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

optional

```
next;          #moves to the next iteration

last;          #exits the loop
```

script4.pl

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

#finding out the accuracy in Perl

$n1 = 1;
$n2 = 1;

while(1)
{
        $n2 = $n2 / 10;
        if($n1 + $n2 == $n1)
        {
                print "$n1 + $n2 SAME as $n1\n";
                print "Perl accuracy reached\n";
                last;
        }
        else
        {
                print "$n1 + $n2 DIFFERENT than $n1\n";
        }
}
```

for loop

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

```
next;          #moves to the next iteration

last;          #exits the loop
```

script5.pl

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

#compute factorial

print "type factorial input: ";
$n0 = <STDIN>;
chomp($n0);

$result = 1;
for($i=2; $i<=$n0; $i+=1)
{
        $result *= $i;
}
print "$n0 factorial is $result\n";
```

The first real program: compute π number.

Steps

1. Decide how to do it – choose algorithm
2. Write a plan in *pseudocode* to have execution framework
3. Fill the framework with the actual code
4. Try to run and eliminate basic errors (syntax etc)
5. Run and verify the output – debug.

Compute π number: algorithm

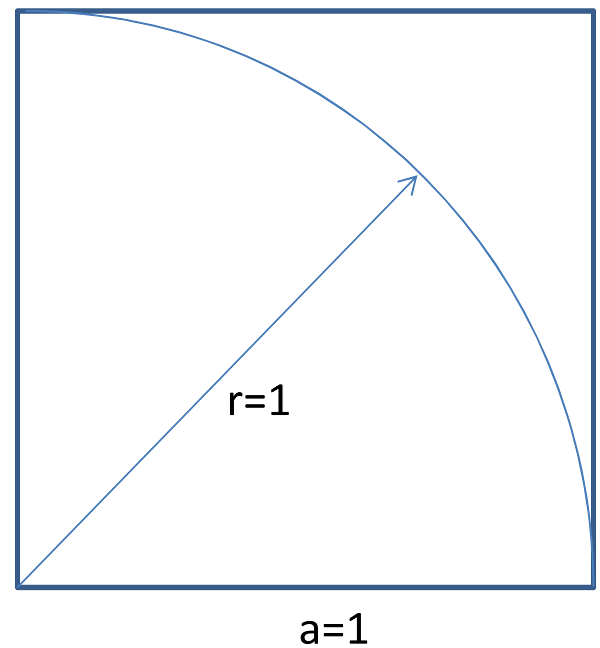
take a square of a side length of 1

put a quarter of a circle with radius of 1 inside

area of the square is $A_s = a*a = 1$

area of the quarter of this circle is
$A_c = 0.25*\pi*r^2 = 0.25*\pi$

$A_c / A_s = \pi / 4$



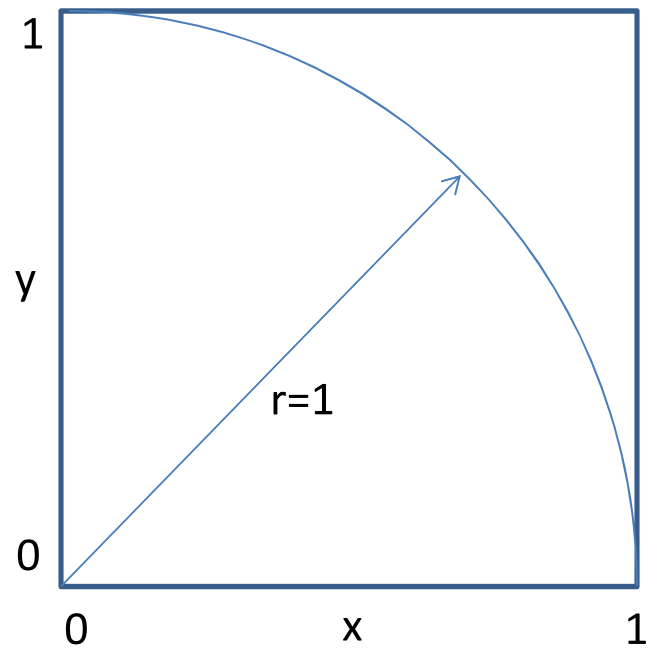
Compute π number: algorithm

$A_c / A_s = \pi / 4$

If we select random points inside the square the ratio of the number of those that are inside the circle to the total will approach the ratio of areas as accurately as we want, provided we select sufficient number of points and our random numbers are random

$N_c / N_{total} = \pi / 4$

$\pi = 4 * N_c / N_{total}$



Compute π number: algorithm

$\pi = 4 * N_c / N_{total}$

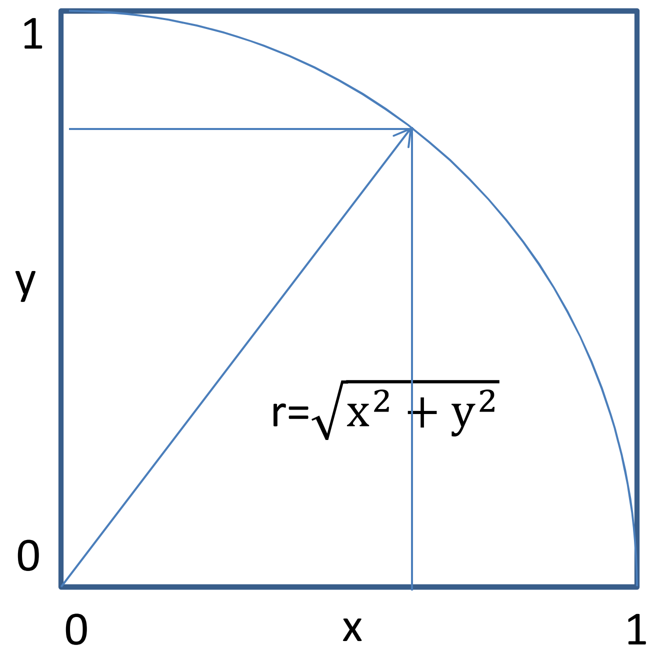
Algorithm

Select two random numbers x,y; each between 0 and 1

if $\sqrt{x^2 + y^2} < 1$ count it as inside the circle

repeat the above MANY times counting total number of pairs and the number of pairs inside circle

compute π



script6.pl

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

#initialize random number generator and counters

#do computations in a loop

        #get two random numbers

        #check if they are inside circle

        #update counters

        #compute current pi and print it

#end loop

#print final pi value
```

script6.pl

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

#initialize random number generator and counters
srand(1484638389);
$ntot = 0;
$nc = 0;

#do computations in a loop

        #get two random numbers

        #check if they are inside circle

        #update counters

        #compute current pi and print it

#end loop

#print final pi value
```

script6.pl

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

#initialize random number generator and counters
srand(1484638389);
$ntot = 0;
$nc = 0;

while($ntot<1000)
{
        #get two random numbers

        #check if they are inside circle

        #update counters

        #compute current pi and print it
}

#print final pi value
```

script6.pl

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

#initialize random number generator and counters
srand(1484638389);
$ntot = 0;
$nc = 0;

while($ntot<1000)
{
        #get two random numbers
        $x = rand(1);
        $y = rand(1);

        #check if they are inside circle

        #update counters

        #compute current pi and print it
}

#print final pi value
```

script6.pl

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

#initialize random number generator and counters
srand(1484638389);
$ntot = 0;
$nc = 0;

while($ntot<1000)
{
        #get two random numbers
        $x = rand(1);
        $y = rand(1);
        #check if they are inside circle
        if(sqrt($x*$x + $y*$y) < 1)
        {
                $nc += 1;
        }
        $ntot += 1;
        #compute current pi and print it
}

#print final pi value
```

script6.pl

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

#initialize random number generator and counters
srand(1484638389);
$ntot = 0;
$nc = 0;

while($ntot<1000)
{
        #get two random numbers
        $x = rand(1);
        $y = rand(1);
        #check if they are inside circle
        if(sqrt($x*$x + $y*$y) < 1)
        {
                $nc++;
        }
        $ntot++;
        #compute current pi and print it
}

#print final pi value
```

shortcut for adding one to itself

script6.pl

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

#initialize random number generator and counters
srand(1484638389);
$ntot = 0;
$nc = 0;

while($ntot<1000)
{
        #get two random numbers
        $x = rand(1);
        $y = rand(1);
        #check if they are inside circle
        if(sqrt($x*$x + $y*$y) < 1)
        {
                $nc++;
        }
        $ntot++;
        #compute current pi and print it
        $pi = 4*$nc/$ntot;
        print "$ntot   $pi\n";
}

print "After $ntot iterations pi is  $pi\n";
```

RUN IT

Looks good, but:

1. π is displayed with varying accuracy
2. we don't need that many lines printed – way too fast
3. 1000 iterations is not enough

```
[jarekp@cbsum1c2b014 perl_03]$ perl script6.pl

980   3.01632653061225
981   3.01732925586137
982   3.0183299389002
983   3.01525940996948
984   3.01626016260163
985   3.01725888324873
986   3.01825557809331
987   3.01925025329281
988   3.02024291497976
989   3.02123356926188
990   3.02222222222222
991   3.01917255297679
992   3.02016129032258
993   3.02114803625378
994   3.02213279678068
995   3.02311557788945
996   3.02409638554217
997   3.02507522567703
998   3.02204408817635
999   3.02302302302302
1000   3.02
After 1000 iterations pi is  3.02
```

The function to produce a string with full control of its shape and form is

printf and **sprintf**

the first parameter is the format, expressed in C notation

the following parameters are values to be printed according to format

printf is like print, but formatted, **sprintf** prints to a string

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

will produce output

```
full length number 2.000000000000000 while short is 3
```

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

script7.pl

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

printf("%17.15f", 2);
print "\n";

printf("%17.10e", 2);
print "\n";

printf("%10d", 2);
print "\n";

printf("%010d", 2);
print "\n";

printf("*%s*", "a string");
print "\n";

printf("*%-20s*", "a string");
print "\n";

print sprintf("*%20s*", "a string") ."\n";
```

RUN IT

Looks good, but:

1. π is displayed with varying accuracy

2. we don't need that many lines printed – way too fast

3. 1000 iterations is not enough

```
[jarekp@cbsum1c2b014 perl_03]$ perl script6.pl

980   3.01632653061225
981   3.01732925586137
982   3.0183299389002
983   3.01525940996948
984   3.01626016260163
985   3.01725888324873
986   3.01825557809331
987   3.01925025329281
988   3.02024291497976
989   3.02123356926188
990   3.02222222222222
991   3.01917255297679
992   3.02016129032258
993   3.02114803625378
994   3.02213279678068
995   3.02311557788945
996   3.02409638554217
997   3.02507522567703
998   3.02204408817635
999   3.02302302302302
1000   3.02
After 1000 iterations pi is  3.02
```

script6a.pl

run longer

print every 1000 iterations

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

#initialize random number generator and counters
srand(1484638389);
$ntot = 0;
$nc = 0;

while($ntot<1_000_000)
{
        #get two random numbers
        $x = rand(1);
        $y = rand(1);
        #check if they are inside circle
        if(sqrt($x*$x + $y*$y) < 1)
        {
                $nc++;
        }
        $ntot++;
        #compute current pi and print it
        $pi = 4*$nc/$ntot;
        if($ntot%1000==0){printf(" %15d  %18.16f\n", $ntot, $pi);}
}

printf "After %d iterations pi is %18.16f\n ", $ntot, $pi;
```

RUN IT

Looks good, but:

1. 1000000 iterations is not enough

```
979000  3.1437589376915218
980000  3.1436979591836733
981000  3.1437268093781854
982000  3.1436659877800408
983000  3.1437070193285859
984000  3.1437439024390246
985000  3.1436994923857866
986000  3.1436470588235292
987000  3.1436393110435663
988000  3.1436477732793522
989000  3.1436481294236605
990000  3.1435636363636363
991000  3.1435358224016143
992000  3.1435362903225808
993000  3.1434964753272912
994000  3.1434406438631792
995000  3.1434733668341708
996000  3.1435301204819277
997000  3.1435185556670011
998000  3.1435511022044089
999000  3.1435595595595593
1000000  3.1436679999999999
After 1000000 iterations pi is 3.1436679999999999
```

Exercises

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 π number converging to the real π? If yes, what does it say about our computer? If no, what is the problem?

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

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.