Perl for Biologists

Session 6
April 8, 2015

Files, directories and I/O operations

Jaroslaw Pillardy
Reminder: What is a Hash?

### Array

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>apple</td>
</tr>
<tr>
<td>1</td>
<td>banana</td>
</tr>
<tr>
<td>2</td>
<td>cranberry</td>
</tr>
<tr>
<td>3</td>
<td>daikon</td>
</tr>
<tr>
<td>4</td>
<td>eggplant</td>
</tr>
<tr>
<td>5</td>
<td>81</td>
</tr>
</tbody>
</table>

### Hash

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>red fruit</td>
<td>apple</td>
</tr>
<tr>
<td>yellow fruit</td>
<td>banana</td>
</tr>
<tr>
<td>red berry</td>
<td>cranberry</td>
</tr>
<tr>
<td>white tuber</td>
<td>daikon</td>
</tr>
<tr>
<td>purple veggie</td>
<td>eggplant</td>
</tr>
<tr>
<td>nine squared</td>
<td>81</td>
</tr>
</tbody>
</table>
Reminder: Hash Syntax

• Declaring a variable to be a Hash, use “%”

```perl
%hash;
```

• Can initialize a hash using an Array

```perl
%hash = ( 'red fruit', 'apple', 'yellow fruit', 'banana', 'nine squared', 81 );
```
Reminder: Hash Syntax

• Accessing individual Hash element
  
  ```perl
  $hash{ 'red fruit' };
  ```

• Equivalently
  
  ```perl
  $key = 'red fruit';
  $hash{$key};
  ```
Reminder: Hash Syntax

• Assigning individual Hash element
  
  ```perl
  $hash{ 'purple veggie' } = 'eggplant';
  ```

• Can initialize an array using a Hash
  
  ```perl
  @array = %hash;
  ```
Reminder: The Big Arrow

```perl
%hash = (  
    'red fruit'    => 'apple',  
    'yellow fruit' => 'banana',  
    'red berry'    => 'cranberry',  
    'white tuber'  => 'daikon',  
    'purple veggie'=> 'eggplant',  
    'nine squared' => 81,  
);
```
Reminder: Hash Functions

• keys(%hash): returns an Array of the Keys

```perl
@hash_keys = keys (%hash);
@hash_keys = keys %hash;
```

• values(%hash): returns an Array of the Values

```perl
@hash_values = values (%hash);
@hash_values = values %hash;
```
Reminder: Hash Functions

- `each(%hash)`: returns the next Key Value pair as a 2 element array

  ```perl
  @pair = each (%hash);
  print "\$pair[0] = \$pair[1]\n";
  ```

```perl
while (@pair = each (%hash))
{
    print "\$pair[0] = \$pair[1]\n";
}
```
Session 5 Exercises Review

1. Modify the code from session 3 exercise 3 to generate a 9kb long random DNA sequence. Save this sequence to a variable.

2. Create a hash where the keys are unique sequences of 3 base pairs and the values are the counts of how often the key appeared in the randomly generated sequence. Print out/save to a file these keys and values.

Bonus: Print out/save keys and values sorted by values in decreasing order

/home/jarekp/perl_05/exercise1.pl

Person with the most number (at least 3) of distinct methods of populating a hash from a string wins 50 FREE CBSU computing hours!

/home/jarekp/perl_05/exercise2.pl
/home/jarekp/perl_05/exercise3.pl
/home/jarekp/perl_05/exercise4.pl
/home/jarekp/perl_05/exercise5.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
#!/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";
Opening files as streams

You can open file and create a stream with `open` function

```perl
open HANDLE, "//path/filename";
```

- name of the stream, also called `filehandle` or `iohandle` or `handle`
- file name: relative or absolute

the above opens the file for reading (default)
Opening files as streams

You can open file for writing:

```perl
open HANDLE, ">/path/filename";
```

The file will be created if it doesn't exist. If it does exist it will be overwritten.

Opening file for append requires ‘>>’

```perl
open HANDLE, " >>=/path/filename";
```
Function **open** returns value indicating success or failure

```perl
$res = open HANDLE, ">/path/filename";
if($res)
{
    print "open successful\n";
}

or

if(!open HANDLE, ">/path/filename")
{
    print "open failed\n";
}
```
If there is an error special variable $! is set to an error text message generated by the system (like “access denied”)

```perl
if(!open HANDLE, ">/path/filename")
{
    print "open failed\nError is: $!";
}
```
There is a short version of if, especially useful in one-line statements:

```perl
open HANDLE, ">filename" or die "Open failed\nError is: $!";
```

- `if substitute: execute whatever is past 'or' if previous statement returned false`
- `print the message into STDERR and terminate the program`
There is a short version of `if`, especially useful in one-line statements:

```perl
open HANDLE, ">filename" or print "Open failed\nError: $!";
```

- `if substitute`: execute whatever is past `or` if previous statement returned `false`
- `print the message into STDOUT`
Once opened, the file can be read the same way as `<STDIN>`

```perl
$svar=<in>;
```

the file should be closed with close when not needed – it will flush the buffers

```perl
close(out);
```
Script to read file1 and copy the content to file2

File names read from arguments of the script

Every other new line replaced with a space
#!/usr/local/bin/perl

# we want 2 file names as parameters
if ($#ARGV != 1) {
    print STDERR "USAGE: script1.pl file_name1 file_name2\n"
    exit;
}

# now try to open files
open in, $ARGV[0] or die "ERROR1: $!\n";
open out, "->" . $ARGV[1] or die "ERROR2: $!\n";
# lets read file 1 and write file 2 in a loop
# lets replace line endings with space on every other line
# when writing to file 2
$n=1;
while ($txt=<in>) {
    chomp $txt;  # remove ending \n character
    print "line \n length is " . length($txt) . "\n";
    print out "$txt";
    if ($n % 2 == 0) {
        print out "\n";
    } else {
        print out " ";
    }
    $n++;
}
# close files
close (in);
close (out);
What happens when we forget to open the file?

If reading, we will always get an empty string.

If writing, the data is ignored (goes to /dev/null).
What happens when we forget to close the file?

The file will be closed automatically when program exits, or when the handle is reused (opened again).

However, if the program crashes, the data being written to a file may be lost.

The data is written to a buffer first, then transferred to the disk later. This procedure speeds up read/write a lot, but if interrupted data may be lost.
Perl script

```
print HANDLE
```

FAST: immediate write

RAM temporary storage area (BUFFER)

SLOW: delayed write, usually when buffer full, or after timeout

Permanent storage (HARD DRIVE)
Very little memory used by processes.

Yet the memory is almost full – here are the buffers.
Buffering is a feature of both system and Perl interpreter, and it can be controlled by setting special variable $|

$| = 1;  # don’t buffer current stream
$| = 0;  # do buffer current stream

A stream is made current by using select() function

$prev = select(out);

previously selected stream

stream now selected for changing

A stream handle can be kept in a variable instead of bareword

open $handle, "filename";
script2.pl (1)

Script printing random numbers to a file

File name is the argument

User decides to buffer the output or not
#!/usr/local/bin/perl

if($#ARGV < 0)
{
    print STDERR "USAGE: script2.pl file_name
";
    exit;
}

print "Do you want to flush? (y/n) ";
while($fl=<STDIN>)
{
    chomp $fl;
    if($fl ne "y" && $fl ne "n")
    {
        print "error: invalid input $fl
";
        print "Do you want to flush? (y/n) ";
    }
    else
    {
        last;
    }
}
open out, "">" . $ARGV[0] or die "ERROR: $!
";
if($fl eq "y")
{
    $prev = select(out);  #choose which stream buffer we will modify
    $| = 1;               #switch buffering off in selected stream
    select($prev);       #switch back to previously selected stream
}

$n=0;
while($n<1000_000_000)
{
    $n++;
    if($n % 1000 == 0)
        {printf out "%010d %17.16f
", $n, sqrt(rand(100));}  #print every 1000
}

close(out);
Binary files

By default, any stream opened is treated as ASCII (text) stream.

Reading a file in ASCII (text) mode means some binary characters may be lost (converted) and in general the written binary file becomes corrupted.

See what happens when script1.pl is used for ~jarekp/perl_06/picture.jpg (you can view picture with eog).

File handle must be marked as binary in order to stop character conversion.
Binary files

```perl
open HANDLE1, "'/path/filename1';
binmode(HANDLE1);
$count = read(HANDLE1, $data, $size);
```

- how many bytes have been read
- binary data from file is stored in a variable
- how many bytes to read

```perl
open HANDLE2, ">/path/filename2";
binmode(HANDLE2);
print HANDLE2 $data;
```
Script copying binary file

Source file is argument 1, destination file argument 2

Print number of bytes copied
#!/usr/local/bin/perl

# we want 2 file names as parameters
if ($#ARGV != 1) {
    print STDERR "USAGE: script3.pl file_name1 file_name2\n"
    exit;
}

open in, $ARGV[0] or die "ERROR1: $!\n"
binmode(in);
open out, "">" . $ARGV[1] or die "ERROR2: $!\n"
binmode(out);

# lets read file 1 and write file 2 in a loop
$n=0;
while ($cnt=read(in, $data, 1024)) {
    $n += $cnt;
    print "$n bytes total read so far, $cnt this iteration\n";
    print out $data or die "Error writing file\n$!";
}

print "$n bytes copied\n";
# close files
close(in);
close(out);
Opening pipelines

Perl can open output stream of a program and read it as a file.

```perl
open HANDLE, "program1 | ";
```

The file handle will reach the end when program1 ends.

```perl
open HANDLE, "program1 | program2 | ";
```

output from program1 goes as input to program2
output from program2 goes to our file handle
Executing a program inside Perl

Perl can execute any program from inside a script:

```perl
system("program1 arg1 arg2");
```
STDOUT from program1 will go to script’s STDOUT
STDERR will go to script’s STDERR

```perl
system("program1 arg1 arg2 1> out");
```
STDOUT from program1 will go to file out
STDERR will go to script’s STDERR

```perl
system("program1 arg1 arg2 1> out 2> err");
```
STDOUT from program1 will go to file out
STDERR will go to file err

Perl script will WAIT until program1 finishes
Executing a program inside Perl

```perl
system("program1 arg1 arg2 1> out 2> err &");
```

now Perl WILL NOT WAIT for program1 to finish, will continue immediately and program1 will run in parallel.
Executing a program inside Perl

$n = \text{system}("program1 \ arg1 \ arg2");

system() returns an integer representing completion code of the program

usually 0 for success and something else for error.
Executing a program inside Perl

```perl
system("program1 arg1 arg2");
```

can be also written using back quotes, in this case the return is the OUTPUT of the command

```perl
$str = `program1 arg1 arg2`
```
Script to find the number of sequences and number of amino acids in swissprot BLAST database
#!/usr/local/bin/perl

open in, "fastacmd -d /shared_data/genome_db/BLAST_NCBI/swissprot -p T -D 1 |" or die "ERROR: $!\n";

$n=0;
$aa=0;
|$|=1;
while($txt=<in>)
{
    if(substr($txt, 0, 1) eq ">")
    {
        $n++;
        if($n % 1000 == 0){print ".";} 
        if($n % 80_000 == 0){print "\n";}
    }
    else
    {
        $aa += length($txt) - 1;
    }
}
if($n % 80_000 != 0){print "\n";}
close(in);
print "swissprot contains $n sequences and $aa aa\n";
Logical operators for files and directories

- `e "name"`    file or directory exists
- `f "name"`    `name` is a file
- `d "name"`    `name` is a directory
- `s "name"`    `name` is non-zero size
- `r "name"`    `name` is readable
- `w "name"`    `name` is writable
- `x "name"`    `name` is executable
- `z "name"`    `name` exists and has zero size

... and more
Functions operating on files and directories

- `mkdir("name")` create directory *name*
- `rmdir("name")` delete directory *name*
- `chdir("name")` change current SCRIPT directory to *name*
- `unlink("name")` delete file *name*
- `rename("name")` rename file or directory *name*

... and more
Functions operating on files and directories

Many Perl files and directories functions do the same as system functions:

```perl
chdir("name"); \iff system("cd name"); \iff NOT THE SAME
```

changes current directory of Perl script

changes current directory of system command shell, Perl script current directory is NOT affected

```perl
mkdir("name"); \iff system("mkdir name") \iff OK, but system dependent
```

```perl
unlink("name"); \iff system("rm name") \iff OK, but system dependent
```

The most important difference between the functions and system calls is that the system calls will only work on one system (e.g. Linux if using “rm name”, on Windows it should be “del name”), while the functions will work on ANY system
#!/usr/local/bin/perl
$pwd = `pwd`;
print "1. Our current directory is: $pwd"
mkdir("tmpdir");
if(! -e "tmpdir")
{
    print "ERROR!\n"
    exit;
}
else
{
    print "mkdir worked!\n"
}
chdir("tmpdir");

print "2. Our current directory is: ";
system("pwd");

print "----\n"
system("cd /tmp; pwd ");
print "----\n";

print "3. Our current directory is: ";
system("pwd");

chdir("/tmp");

print "4. Our current directory is: ";
system("pwd");
Opening and reading a directory

Perl can open a directory and retrieve all its entries:

```
opendir DIRHANDLE, "\path\dirname";
```

Similar as to file `open, opendir` returns success or failure code

```
@ent = readdir(DIRHANDLE);
```

returns an array containing all entries in a directory – i.e. names of all files and directories it contains (including ‘.’ and ‘..').

```
closedir DIRHANDLE;
```
#!/usr/local/bin/perl

opendir DIR, "/home/jarekp";

foreach $entry (readdir DIR)
{
    $fullentry = "/home/jarekp/$entry";
    if(-d $fullentry)
    {
        print "directory $entry\n";
    }
    elsif(-x $fullentry)
    {
        print "executable $entry\n";
    }
    elsif(-f $fullentry)
    {
        print "file $entry\n";
    }
    else
    {
        print "other entry $entry\n";
    }
}
Exercises

1. Directory /home/jarekp/perl_06/files contains a set of fastq files with short reads. Write a script that lists all the files in this directory.

2. Modify the script from exercise 1 to open each file, read it, and produce a hash containing the distribution of sequence lengths in ALL files. Print the distribution out in descending order to a file. Plot it in Excel (no binning).

   Hint 1: Fastq file contains 4 lines for each sequence: header (starting with @), sequence itself, ‘+’ line, and quality score line. Check script4.pl – there we had two lines per sequence.

   Hint 2: Create a hash where sequence lengths are the keys and values are frequencies of the lengths.

3. Modify the script from exercise 2 to produce fasta file containing ALL the sequences from ALL fastq files.

   Hint: Open fasta file at the beginning, then write each header (replacing first @ with >) followed by sequence.