Homework #8

Creative to use different commands

There are always multiple ways to do it

If you want to transfer between two linux machines, use scp

```
scp source destination
```

If I want to copy test-query.fa from glu to gly:

1. Go to /home/yyin/work/class/ of glu

```
scp test-query.fa yyin@131.156.41.193:~/
```

Or

2. Go to gly

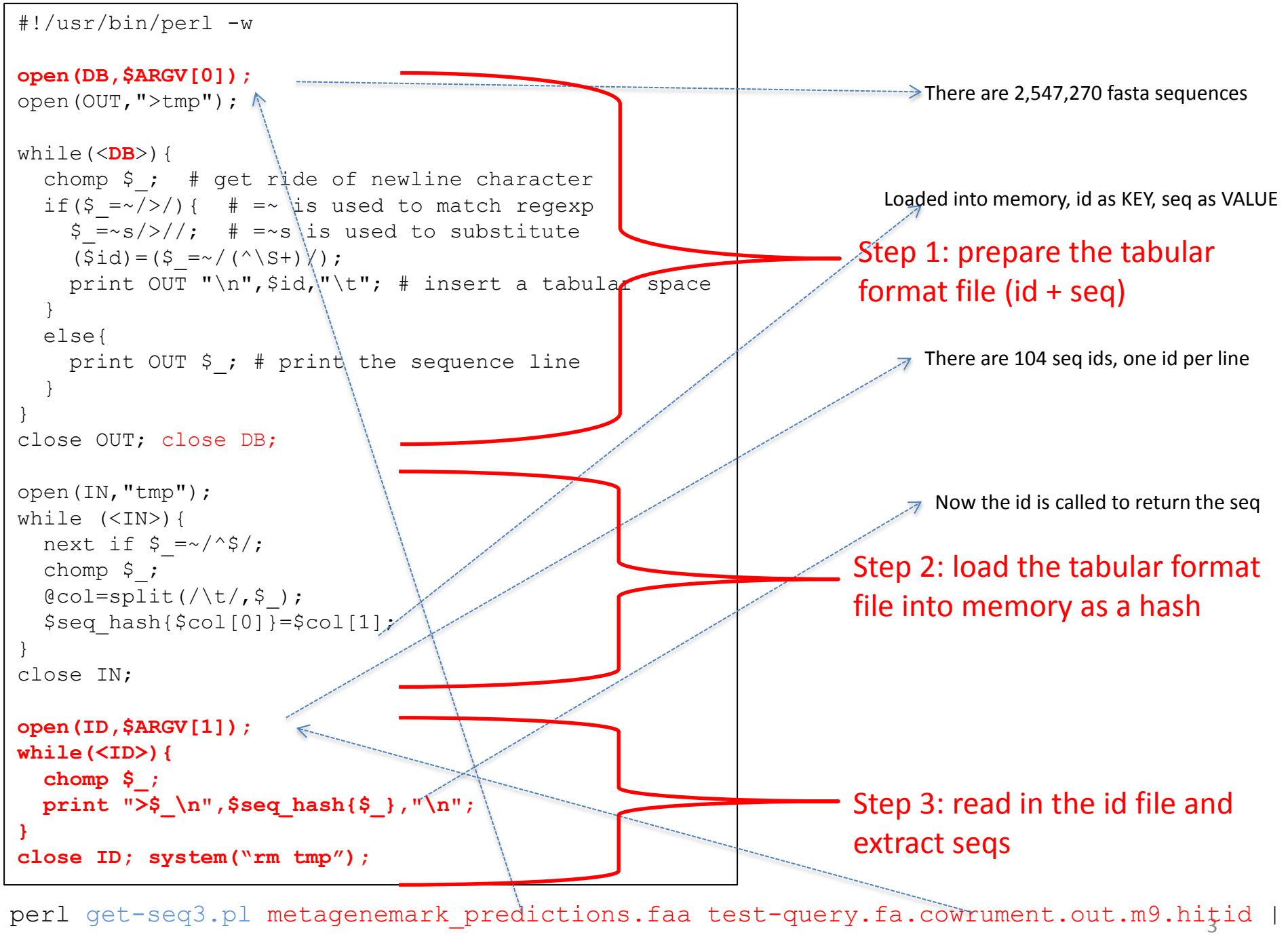
```
scp yyin@131.156.41.196:~/work/class/test-query.fa .
```

Copy folders need the –r option:

```
scp -r yyin@131.156.41.193:~/db/ .
scp -r db yyin@131.156.41.196:~/work/class/
```

This week:

Subroutine
Perl module
Bioperl




```
perl get-seq3.pl metagenemark_predictions.faa test-query.fa.cowrument.out.m9.hitid | less
```

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

open(DB,$ARGV[0]);
open(OUT,">tmp");

while(<DB>){
  chomp $_;  # get ride of newline character
  if($_=~/>/){  # =~ is used to match regexp
    $_=~s/>//;  # =~s is used to substitute
    ($id)=($_=~/(^\S+)/);
    print OUT "\n",$id,"\t"; # insert a tabular space
  }
  else{
    print OUT $_; # print the sequence line
  }
}
close OUT; close DB;

open(ID,$ARGV[1]);
while(<ID>){
  chomp $_;
  $id_hash{$_}=1;
}

open(IN,"tmp");
while (<IN>){
  next if $_=~/^$/;
  chomp $_;
  @col=split(/\t/,$_);
  if(defined $id_hash{$col[0]}){
    print ">",$col[0],"\n",$col[1],"\n";
  }
}
close IN;
close ID; system("rm tmp");
```

There are 104 seq ids

There are 2,547,270 fasta sequences

If you want to load a huge database …

Computer memory will be an issue …

```
cp get-seq3.pl get-seq4.pl
vi get-seq4.pl
```

Step 1: prepare the tabular format file (id + seq)

Step 3: read in the id file and **load ids into memory (a hash)**

Step 2: **DO NOT** load the tabular format file into memory as a hash

Go through each sequence line to check if the ID is in the hash


```
perl get-seq4.pl metagenemark_predictions.faa test-query.fa.cowrument.out.m9.hitid | less
```

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

open(DB,$ARGV[0]);
open(OUT,">tmp");

while(<DB>){
  chomp $_;  # get ride of newline character
  if($_=~/>/){  # =~ is used to match regexp
    $_=~s/>//;  # =~s is used to substitute
    ($id)=($_=~/(^\S+)/);
    print OUT "\n",$id,"\t"; # insert a tabular space
  }
  else{
    print OUT $_; # print the sequence line
  }
}
close OUT; close DB;

open(ID,$ARGV[1]);
while(<ID>){
  chomp $_;
  $id_hash{$_}=1;
}

open(IN,"tmp");
while (<IN>){
  next if $_=~/^$/;
  chomp $_;
  @col=split(/\t/,$_);
  if(defined $id_hash{$col[0]}){
    print ">",$col[0],"\n",$col[1],"\n";
  }
}
close IN;
close ID; system("rm tmp");
```

If we use bioperl, the program will be much shorter

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

open(ID,$ARGV[1]);
while(<ID>){
  chomp $_;
  $id_hash{$_}=1;
}

use Bio::SeqIO;

$new=Bio::SeqIO->new(-
file=>$ARGV[0], -format=>"fasta");

while($seq=$new->next_seq){
  if(defined $id_hash{$seq->id}){
    print ">",$seq->id,"\n",$seq-
>seq."\n";
  }
}
```

Bioperl is a collection of perl **modules** that facilitate the development of perl scripts for biology use.

A module is a named container for a group of variables and **subroutines** which can be loaded into your program.

```
perl get-seq4.pl metagenemark_predictions.faa ecoli-all.faa test-query.fa.cowrument.out.m9.hitid | less
```

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

open(DB,$ARGV[0]);
open(OUT,">tmp");

while(<DB>){
  chomp $_;  # get ride of newline character
  if($_=~/>/){  # =~ is used to match regexp
    $_=~s/>//;  # =~s is used to substitute
    ($id)=($_=~/(^\S+)/);
    print OUT "\n",$id,"\t"; # insert a tabular space
  }
  else{
    print OUT $_; # print the sequence line
  }
}
close OUT; close DB;

open(ID,$ARGV[1]);
while(<ID>){
  chomp $_;
  $id_hash{$_}=1;
}

open(IN,"tmp");
while (<IN>){
  next if $_=~/^$/;
  chomp $_;
  @col=split(/\t/,$_);
  if(defined $id_hash{$col[0]}){
    print ">",$col[0],"\n",$col[1],"\n";
  }
}
close IN;
close ID; system("rm tmp");
```

Step 1

Step 2

Step 3

What if you want to read in another fasta database and convert the fasta format to a tabular format?

You will have to repeat this section of codes twice.

As your programs become more and more complex, you'll find yourself repeating the same chunk of code in multiple places within the same program.

You will need **subroutines** to avoid the repetitions or reuse existing codes.

A subroutine is a named block of code that can be reused in multiple places

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

fasta2tab($ARGV[0]);

open(ID,$ARGV[1]);
while(<ID>){
  chomp $_;
  $id_hash{$_}=1;
}

open(IN,"tmp");
while (<IN>){
  next if $_=~/^$/;
  chomp $_;
  @col=split(/\t/,$_);
  if(defined $id_hash{$col[0]}){
    print ">",$col[0],"\n",$col[1],"\n";
  }
}
close IN;
close ID; system("rm tmp");

###################################
sub fasta2tab {
  ($fastafile)=@_;

  open(DB,$fastafile);
  open(OUT,">tmp");

  while(<DB>){
    chomp $_;  # get ride of newline character
    if($_=~/>/){  # =~ is used to match regexp
      $_=~s/>//;  # =~s is used to substitute
      ($id)=($_=~/(^\S+)/);
      print OUT "\n",$id,"\t"; # insert a tabular space
    }
    else{
      print OUT $_; # print the sequence line
    }

  }
  close OUT; close DB;
}
```

Step 2

Step 3

Step 1

```
cp get-seq4.pl get-seq-sub.pl

vi get-seq-sub.pl
```

Call the subroutine **fasta2tab** and pass the file name in

@_ is a special variable used to capture arguments passed from outside

Filename -> $ARGV[0] -> @_ -> $fastafile

You may have multiple subroutines defined in one script as long as they have different names and called in the main program. This type of coding makes your program look more organized and easy to debug

This code black was in the above, now is in a subroutine called **fasta2tab**

Subroutine syntax: **sub NAME {}**

```
perl get-seq-sub.pl metagenemark_predictions.faa test-query.fa.cowrument.out.m9.hitid | less
```

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

use lib "/home/yyin/work/class/";
use module::mymodule;

mymodule::fasta2tab($ARGV[0]);

open(ID,$ARGV[1]);
while(<ID>){
  chomp $_;
  $id_hash{$_}=1;
}

open(IN,"tmp");
while (<IN>){
  next if $_=~/^$/;
  chomp $_;
  @col=split(/\t/,$_);
  if(defined $id_hash{$col[0]}){
    print ">",$col[0],"\n",$col[1],"\n";
  }
}
close IN;
close ID; system("rm tmp");
```

Step 2

Step 3

It would be nice to be able to **have a generic library code that we can include in our programs**, so all that we have to do is to call the subroutine and not have to worry about copying the subroutine from one program to another.

As we write more and more programs, we often find one subroutine we used in one script might also be useful in another script.

We don't want to copy the often-used subroutines from one script to another.

Step 1, the subroutine is now in a separate file

```
package mymodule;

sub fasta2tab{

($fastafile)=@_;
open(DB,$fastafile);
open(OUT,">tmp");

while(<DB>){
  chomp $_;  # get ride of newline character
  if($_=~/>/){  # =~ is used to match regexp
    $_=~s/>//;  # =~s is used to substitute
    ($id)=($_=~/(^\S+)/);
    print OUT "\n",$id,"\t"; # insert a
tabular space
  }
  else{
    print OUT $_; # print the sequence line
  }

}
close OUT; close DB;
}

1;
```

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

use lib "/home/yyin/work/class/";
use module::mymodule;

mymodule::fasta2tab($ARGV[0]);

open(ID,$ARGV[1]);
while(<ID>){
  chomp $_;
  $id_hash{$_}=1;
}

open(IN,"tmp");
while (<IN>){
  next if $_=~/^$/;
  chomp $_;
  @col=split(/\t/,$_);
  if(defined $id_hash{$col[0]}){
    print ">",$col[0],"\n",$col[1],"\n";
  }
}
close IN;
close ID; system("rm tmp");
```

Tell perl where to find the **module** folder

Tell perl where the **mymodule.pm** file is

Call the **fasta2tab** subrountine in the mymodule file in the module folder

`vi get-seq-mymodule.pl`

Change this to the path of your current folder

`mkdir module`

`vi module/mymodule.pm`

```
package mymodule;

sub fasta2tab{

($fastafile)=@_;
open(DB,$fastafile);
open(OUT,">tmp");

while(<DB>){
  chomp $_;  # get ride of newline character
  if($_=~/>/){  # =~ is used to match regexp
    $_=~s/>//;  # =~s is used to substitute
    ($id)=($_=~/(^\S+)/);
    print OUT "\n",$id,"\t"; # insert a
tabular space
  }
  else{
    print OUT $_; # print the sequence line
  }

}
close OUT; close DB;
}

1;
```



```
perl get-seq-mymodule.pl metagenemark_predictions.faa test-query.fa.cowrument.out.m9.hitid | less
```

Modules (packages) are an important and powerful part of the Perl programming language. A module is a named container for a group of variables and subroutines which can be loaded into your program. By naming this collection of behaviors and storing it outside of the main program, you are able to refer back to them from multiple programs and solve problems in manageable chunks.

Modular programs are more easily tested and maintained because you avoid repeating code, so you only have to change it in one place. Perl modules may also contain documentation, so they can be used by multiple programmers without each programmer needing to read all of the code.

Modules are the foundation of the CPAN (Comprehensive Perl Archive Network, http://www.cpan.org/), which contains **114,000** ready-to-use modules, many of which you will likely use on a regular basis.

Some modules also call or depend on other modules, so they are all interconnected.

http://learnperl.scratchcomputing.com/tutorials/modules/

http://www.bioperl.org/

Page Discussion Read

Main Page

Welcome to BioPerl, a community effort to produce Perl code which is useful in biology.

For more background on the BioPerl project please see the History of BioPerl.

BioPerl is distributed under the Perl Artistic License. For more information, see licensing BioPerl.

Main Links

Main Page
Getting Started
Downloads
Installation
Recent changes
Random page

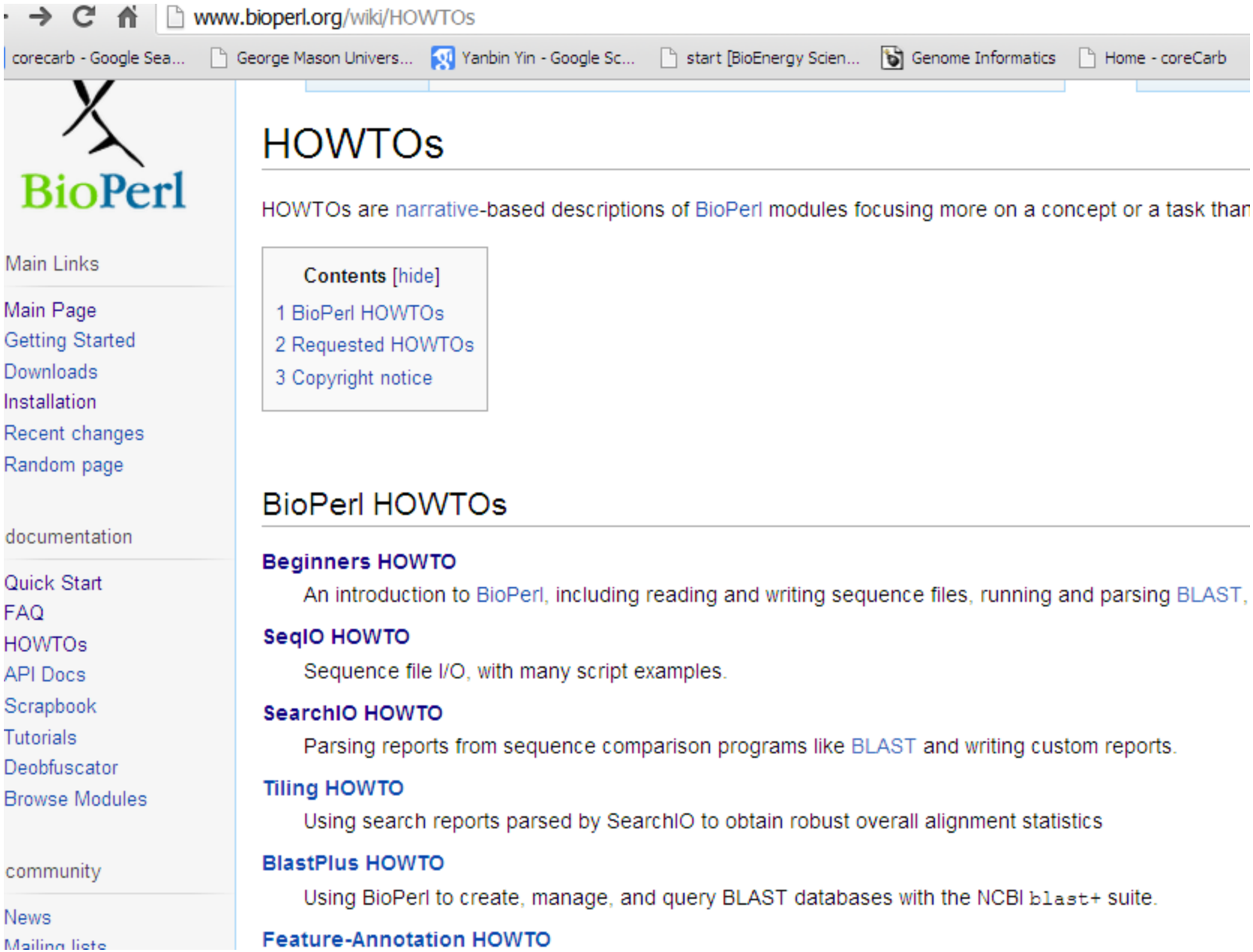
documentation

Quick Start
FAQ
HOWTOs
API Docs
Scrapbook
Tutorials
Deobfuscator
Browse Modules

community

News
Mailing lists
Supporting BioPerl
BioPerl Media
Hot Topics
About this site
News
Mailing lists
Supporting BioPerl
BioPerl Media
Hot Topics

Installation	Documentation	Support
• Linux • Windows • Mac OSX • Ubuntu Server • FreeBSD • Fedora	• API Docs and BioPerl docs • HOWTO • Scrapbook • The (in)famous Deobfuscator	• FAQ • IRC : #bioperl Webchat • Mailing lists • Search mail list archives • BioPerl Media options
Developers	**How Do I...?**	**BioPerl-related Distributions**
• Using Git • Advanced BioPerl • The SeqIO Modules • Features and Annotations in BioPerl • Writing BioPerl Tests • BioPerl Best Practices • Browse the Core Modules • Bugs	• ...install BioPerl? • ...find a nice, readable BioPerl overview? • ...get help? • ...learn Perl? • ...read a sequence file? • ...parse a BLAST/FASTA search? • ...get genomic sequences and coordinates? • ...query GenBank sequence/annotations? • ...contribute code to BioPerl? • ...submit a bug report? • ...see open BioPerl bugs?	• Core • BioSQL adaptors (BioPerl-DB) • BioPerl-Run wrappers • Network analysis (BioPerl-Network) • Bio::Graphics • BioPerl-Pedigree • Gbrowse


www.bioperl.org/wiki/HOWTOs
corecarb - Google Sea...
George Mason Univers...
Yanbin Yin - Google Sc...
start [BioEnergy Scien...
Genome Informatics
Home - coreCarb
BioPerl
Main Links
Main Page
Getting Started
Downloads
Installation
Recent changes
Random page
documentation
Quick Start
FAQ
HOWTOs
API Docs
Scrapbook
Tutorials
Deobfuscator
Browse Modules
community
News
Mailing lists
HOWTOs
HOWTOs are narrative-based descriptions of BioPerl modules focusing more on a concept or a task than
Contents [hide]
1 BioPerl HOWTOs
2 Requested HOWTOs
3 Copyright notice
BioPerl HOWTOs
Beginners HOWTO
An introduction to BioPerl, including reading and writing sequence files, running and parsing BLAST,
SeqIO HOWTO
Sequence file I/O, with many script examples.
SearchIO HOWTO
Parsing reports from sequence comparison programs like BLAST and writing custom reports.
Tiling HOWTO
Using search reports parsed by SearchIO to obtain robust overall alignment statistics
BlastPlus HOWTO
Using BioPerl to create, manage, and query BLAST databases with the NCBI blast+ suite.
Feature-Annotation HOWTO

http://search.cpan.org/~cjfields/BioPerl-1.6.901/BioPerl.pm

BioPerl is the product of a community effort to produce Perl code useful in biology. Examples include Sequence **objects (modules)**, Alignment objects and database searching objects.

These objects **also interact** - Alignment objects are made from the Sequence objects, Sequence objects have access to Annotation and SeqFeature objects and databases, Blast objects can be converted to Alignment objects, and so on. This means that the objects provide a coordinated and extensible framework to do computational biology.

As the objects do most of the hard work for you, all you have to do is combine a number of objects together sensibly to make useful scripts.

The intent of the BioPerl development effort is to make reusable tools that aid people in creating their own sites or job-specific applications.

Bioperl is a collection of perl modules that facilitate the development of perl scripts for bioinformatics applications. As such, it does not include ready to use programs in the sense that may commercial packages and free web-based interfaces.

On the other hand, bioperl does provide reusable perl modules that facilitate writing perl scripts for sequence manipulation, accessing of databases using a range of data formats and execution and parsing of the results of various molecular biology programs including Blast, clustalw, TCoffee, genscan, ESTscan and HMMER etc.

Jamison D. Perl Programming for Biologists (Wiley,2003) (ISBN 0471430595)

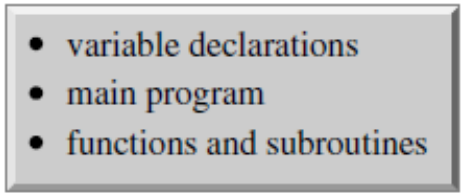
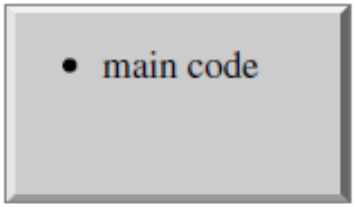
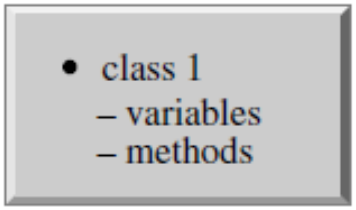


Figure 10.1 Procedural code layout



Classes: modules
Methods: subroutines



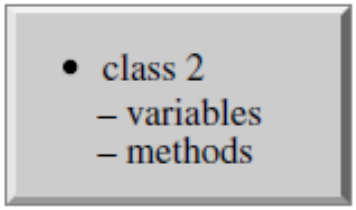


Figure 10.2 Object-oriented code layout

Bioperl modules are called in the main perl scripts in a fashion of Object-Oriented paradigm, which is in contrast to the procedural paradigm. Procedural code is typically used for short programs while OOP is often used for complex medium and long programs.

The OOP is built upon an important concept called **reference**, where all variable types, modules, subroutines can be "referenced" as a hash.

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

use Bio::SeqIO;

$new=Bio::SeqIO->new(-file=>$ARGV[0],
-format=>"fasta");

while($seq=$new->next_seq){
  print $seq->id,"\t", length $seq-
>seq, "\n";
}
```

The **arrow operator ->** is widely used to “dereference” a module or subroutine and build an object (**an object is a specific instance of a module or subroutine**).

vi get-length.pl

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

use Bio::SeqIO;

$new=Bio::SeqIO->new(-file=>$ARGV[0], -format=>"fasta");

while($seq=$new->next_seq){
  print $seq->id,"\t", length $seq->seq, "\n";
}
```

Find out where bioperl modules are installed to:

```
locate bioperl | less

locate Bio | less
```

/usr/share/perl5/Bio/SeqIO

Check the Bio folder to see the SeqIO etc.
http://www.bioperl.org/wiki/HOWTO:SeqIO

```
perl get-length.pl metagenemark_predictions.faa | less
```

Step 1: Create a $new object from a fasta file to hold the reference to the fasta format sequences

Step 2: Call the next_seq method to extract one seq block per cycle and create the $seq object to hold the block

Step 3: Call the id method and the seq method

vi get-seq-bioperl.pl

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

open(ID,$ARGV[1]);
while(<ID>){
  chomp $_;
  $id_hash{$_}=1;
}

use Bio::SeqIO;

$new=Bio::SeqIO->new(-file=>$ARGV[0], -format=>"fasta");

while($seq=$new->next_seq){
  if(defined $id_hash{$seq->id}){
    print ">",$seq->id,"\n",$seq->seq."\n";
  }
}
```

```
perl get-seq-bioperl.pl test-query.fa.cowrument.out.m9.hitid
metagenemark_predictions.faa | less
```

Perl one-liner

You don't write codes into a file and then issue "`perl file.pl`" on the command line;
You write the codes directly on the command line, like you are typing regular Linux commands

```
#!/usr/bin/perl

open (IN,$ARGV[0]);

@a=<IN>;

foreach(@a){
   @col=split(/\t/,$_);
   print $col[1],"\tmutation\n";
}
```

=

```
#!/usr/bin/perl

while (<>){
   @col=split(/\t/,$_);
   print $col[1],"\tmutation\n";
}
```

=

```
perl -e 'while(<>){@col=split(/\t/,$_);print $col[1],"\tmutation\n";}' cosmicRaw.txt.head10.6col
```

=

```
cat cosmicRaw.txt.head10.6col | cut -f2 | awk '{print $1,"mutation"}' | sed 's/ /\t/'
```