Module 2 Content

Introduction

This week we will look at writing scripts in Perl and companion topics XML, JSON, web services and SQLite. We'll access some protein databases this week with Perl (we'll look at protein databases in more detail in Module 4). **NOTE**: you won't need most of this material until Homework 2 but it would be good to start absorbing it this week.

Topics to be covered:

- Perl
- XML
- JSON
- Web Services
- SQLite

Perl

Perl is an interpreted programming language known for its ability to easily and conveniently manipulate string data. It has long been used for bioinformatics applications and the access and manipulation of biological database content. Although there are obviously other programming languages like Python and Java that one can use, we are examining Perl because it has a long history in bioinformatics and everyone should have some knowledge of it.

For example, one important bioinformatics resource is Ensembl (http://www.ensembl.org), which we will visit in Module 8. But for the moment, Ensembl is, according to their website, "a genome browser for vertebrate genomes that supports research in comparative genomics, evolution, sequence variation, and and transcriptional regulation", and it has much of its code in Perl (see https://github.com/Ensembl).

If you are not familiar with Perl, **visit** these pages: http://www.perl.com/pub/2008/04/23/a-beginners-introduction-to-perl-510.html and http://www.perl.com/pub/2008/05/07/beginners-introduction-to-perl-510-part-2.html. Go to http://learn.perl.org/ and view some of the Examples (such as reading/writing to a file). Another useful page is here: http://cslibrary.stanford.edu/108/EssentialPerl.html. In addition NCBI has a page of handy Perl constructs: https://www.ncbi.nlm.nih.gov/Class/PowerTools/eutils/rules.html. Finally, there are numerous tutorials and posts about Perl that you can find via Google.

You have a homework assignment this week to try out some Perl constructs.

If you are using a Unix-flavored system, it will already have Perl. Make sure it is a Perl 5 version.

On a PC, I recommend using Strawberry Perl http://strawberryperl.com/ . Strawberry Perl already comes with modules that are required for the examples and the next homework (LWP, XML, JSON (Mac may be an exception – if you have a Mac check to see if you have it and if no, Google for how to obtain it), DBI, and Data).

CPAN, the Comprehensive Perl Archive Network, is a resource where you can find all kinds of Perl scripts and modules. CPAN: http://www.cpan.org/. If you have a UNIX-based system, you can use CPAN to install modules that were not automatically included with your Perl installation.

To see if your Perl has the modules installed, create a text file (my_script.pl) with the following:

```
use LWP;
use XML::Simple;
use JSON;
use DBI;
use Data::Dumper;
use strict; # For trapping errors. By the way # is the comment character.
use warnings;
```

On a Unix-flavored system add the path to your Perl installation as the first line (including the comment character #):

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

On Unix -- then make the Perl file executable if you wish.

On a PC, work with Perl in a terminal window. Strawberry Perl's installation will put the appropriate directories into your PATH variable.

On both kinds of systems, you can open a terminal window and execute a script with the following. If the modules are not installed, you will get an error. Download them from CPAN. But note: on a Macintosh, you may have to do some additional work to get modules. Unfortunately, I don't have a Mac and so can't help you with this. You may need to visit a Mac Store genius bar if you are not familiar with setting up code utilities on your Mac and you wish to use additional Perl modules beyond those for the homeworks.)

```
perl my_script.pl
```

Some Perl Constructs

A few quick facts about Perl:

- Scalar variables begin with $. $avar = 1; $avar = "some string";
- Strings can be created with double or single quotes. The advantage to double quoted strings is that Perl will do variable substitution in them. $my_str = "MY STRING"; print "Here is a string: $my_str\n"; This will print `Here is a string: MY STRING`
- The period . is Perl's string concatenation operator.
- Array variables begin with @. @myarr = ('A', 'G', 'C', 'T');
- By default, array indices start with 0. $myarr[0] is equal to the character A.
- Statements end with ; just like in C++, Java, and C#, but unlike Python.
- Perl has hash tables, which are called dictionaries or maps in other languages.
- Hash table variables begin with %. %a_map = ("one"=>"first", "two"=>"second");
- Hash tables are used to map a key to a value. $a_map{"one"} is equal to "first".
- You can construct complex data structures, such as arrays of hashes of hashes of arrays, or any combination you wish.
- Use == to compare numeric values and eq to compare strings.
- Often you can write the same thing in Perl in multiple ways. So don't be surprised to see functions called with and without using parenthesis, or different ways to access references, for example.
- Perl scripts can be written in object-oriented style or procedural style.
- Perl can be written in a very terse "Perl" style. I don't program that way because it can be hard for non-Perl programmers to understand it – and I find it hard to understand too, sometimes. You will see my Perl code examples are in a generic coding style. You may use whatever style you like, so long as it is readable and understandable.
- Perl has many built-in variables: @ARGV is the arguments passed into the script. $_ is the current variable being used behind the scenes (more on this in a moment).

- I highly recommend that you begin all your scripts with use strict; to help catch errors. I would also recommend that you turn do use warnings, too.
- You **don't** need to include a use statement with a specific Perl version, which you may see in some examples online (for example, use 5.010). I ask you to **not** do this in your homework.

Constructs you should be able to understand:

- How to work with all the different kinds of variables
- Loops and if-tests
- Regular expressions
- Dereferencing arrays and hash tables (we'll have examples below)
- Subroutines
- Reading and writing to/from files
- Plus the topics we cover this week – parsing XML and JSON, calling web services, and working with sqlite.

Here is an example and explanation of the $_ variable, which you may see in some examples, from http://cslibrary.stanford.edu/108/EssentialPerl.html. (I don't use this implicit variable because I like to have the variables spelled out, but you can use it if you wish.) Note: <FILE> is a file handle that was previously opened for reading in the example. This example also shows Perl's terse style after the comment, #.

```
while ($line = <FILE>) {
    print $line;
}

# Is the same thing as:

while (<FILE>) {
    print;
}
```

"It turns out that <FILE> assigns its value into $_ if no variable is specified, and likewise print reads from $_ if nothing is specified. Perl is filled with little shortcuts like that, so many phrases can be written more tersely by omitting explicit variables."

XML

Spend time familiarizing yourself with XML syntax to see how it is used in *data representation*. Read the XML "basic" tutorial on this web site: https://www.w3schools.com/xml/xml_whatis.asp — the Introduction, How Can XML be Used, XML Tree, XML Syntax Rules, XML Elements, and XML Attributes. (You do not need to know JavaScript to read the tutorial.) You could choose to store your information in XML files for your project or provide XML as the format of results generated from your database. You will see that some resources return data in XML format.

One important point to keep in mind regarding XML is — you can create any tags you like to model your information and give it semantics, but without an agreed-upon set of tags, you can't easily share your information with anyone else. Hence the need for standards, some of which we will talk about in later modules.

JSON

XML is often used to package data that is returned when data is obtained from a resource (such as by using a web service – more below). Another format often used is JSON. JSON is more compact and does not use descriptive tags to delimit information, but instead uses arrays and hashes (also called maps or dictionaries). Arrays use [] and hashes use {}. For example, an array of identifiers and values:

[{“ID”: “1”, “Value”: “one”}, {“ID”: “2”, “Value”: “two”}, {“ID”: “3”, “Value”: “three”}]

In this simple example the structure is an array of hashes, where each hash has a key “ID” and a key “Value”.

JSON can be created from structures in your script and you can generate structures from JSON strings. We’ll see an example of generating a structure below from a JSON string.

Web Services

Many biological data resources offer programmatic access to their information through web services. Two common approaches are REST and SOAP, which provide standards that allow data providers and data consumers to communicate in well-defined ways. A benefit of providing your data through a collection of services is that programmers can use and combine your data in ways (perhaps novel) that suit their needs — all you need to do is provide an API to access the information.

Because RESTful services have become popular, read the Wikipedia page about REST (https://en.wikipedia.org/wiki/Representational_state_transfer). In addition, read the EBI explanation page at https://www.ebi.ac.uk/seqdb/confluence/pages/viewpage.action?pageId=68165098

As you will read, REST is an architectural style, not a specific implementation, so it doesn’t have to rely on HTTP (how information is passed around on the World Wide Web). But the line between the style and the HTTP implementation can be blurred. I’m not too concerned about these fine distinctions; I’m more interested in your knowing the general concepts and how to obtain information from web services in practice.

You can get information from a web service (or a website in general) in Perl using the LWP module. We’ll see how to do this in the Perl examples given further down in the Content.

SQLite

SQLite is a lightweight implementation of most of SQL. It differs from MySQL in that the database is just a file. There is no database server to connect to. This setup makes it easy to use an SQL database in your program for storing/access information that your program needs, instead of using a flat file. (Mobile devices, for example, frequently use SQLite to store information.) We’ll see a Perl example below and you’ll be using it in the second homework.

If you are interested: View a talk that gives an overview of SQLite: http://www.youtube.com/watch?v=jN_YdMdjVpU. There are a number of uses of SQLite discussed and it explains some disadvantages of SQLite, too.

You can download an SQLite database command line tool and browser from http://www.sqlite.org – but this is **not** necessary for any of the homeworks. The DBI module in Perl will take care of creating and accessing your database. You won’t need an external DBMS to work with the database. But if you want one to look at your database tables, you can do so with this tool.

You should already be familiar with SQL commands. The Perl example later in the Content notes will get you started with using SQLite from Perl. Essentially, all the Perl needs to do is connect to the database (the file), construct SQL commands, and execute them, then disconnect when done. Perl will handle creating the database file if necessary.

Examples with Perl

Let's see some examples of using Perl to work with XML, JSON, web services, and SQLite. The exact examples below are not uploaded to the course, but similar files are available in the ZIP in this module and will be identified in the notes below. You are expected to download them, run them, and modify them to explore these concepts. So please do so **this week**. You can also use these scripts as the basis for the programming homeworks.

Accessing Web Services

RCSB PDB is a database for (primarily) protein structures. It provides web services for accessing its data. UniProt is a protein database and you can access its records easily via a URL. (We'll visit these two resources later in the course.) In the upcoming example, we'll get the data returned in XML format from RCSB PDB and UniProt. (Neither provides JSON formatted information as of this writing.) This example uses the object-oriented style of using LWP.

```
use LWP::Simple;
use strict;
use warnings;

# Use RCSB PDB to get the GO terms associated with structure 4HHB
my $url = "http://www.rcsb.org/pdb/rest/goTerms?structureId=4HHB";
my $browser = LWP::UserAgent->new; # object oriented style

# Issue request, with an HTTP header.
# This next statement creates the request and gets it, all in one step.
# DO create a "user agent" - some services won't return anything without it.
my $response = $browser->get($url,
          'User-Agent' => 'Mozilla/4.0 (compatible; MSIE 7.0)');
die "Error getting $url" unless $response->is_success;
print 'Content type is ', $response->content_type;
print 'Content is:';
print $response->content; # This prints what the web service returned.

# And now for the UniProt call:
$url = "https://www.uniprot.org/uniprot/Q16539.xml";
$response = $browser->get($url,
          'User-Agent' => 'Mozilla/4.0 (compatible; MSIE 7.0)');

die "Error getting $url" unless $response->is_success;
print 'Content type is ', $response->content_type;
print 'Content is:';
print $response->content;
```

REST services and web pages can be accessed via a get or a post. The above two calls used get. Normally, I would use get unless the documentation says to use post. The post setup is a little more

difficult. **Note below that the post data is contained in a hash variable** %post_data, and that **it is passed into the post as a reference (the \ makes it a reference)**. The following example comes from an older EBI tutorial. I should point out that this web service returns neither XML nor JSON, but text (called raw) or HTML.

```
# FROM AN OLDER EBI TUTORIAL

use LWP::Simple;
use strict;
use warnings;

my $db = 'uniprotkb'; # Database: UniProtKB
my $id = 'Q16539'; # Entry identifiers
my $format = 'uniprot'; # Result format
my $style = 'raw'; # Result style

# Create a user agent
my $ua = LWP::UserAgent->new(); # object-oriented again, but used ()

# URL for service (endpoint)
my $url = 'http://www.ebi.ac.uk/Tools/dbfetch/dbfetch';

# Populate POST data fields (key => value pairs)
my (%post_data) = (
               'db' => $db,
               'id' => $id,
               'format' => $format,
               'style' => $style
               );

# Perform the request
my $response = $ua->post($url, \%post_data); # reference the hash var

# Check for HTTP error codes
die 'http status: ' . $response->code . ' ' . $response->message unless
($response->is_success);

# Output the entry
print $response->content();
```

Here is an example where the post parameters are passed directly into the post call instead of via a separate variable:

```
$response = $ua->post($url,
      ['db'=>$db, 'id'=>'P69905', 'format'=>$format, 'style'=>$style]);
```

Processing XML

The RCSB PDB and UniProt calls above returned XML-formatted information. Let's look at parsing the UniProt information. We'll use a function called Data::Dumper **to help decipher** what is returned. (See example uniprot_1.pl.)

```
use LWP::Simple;
use XML::Simple;
```

```
use Data::Dumper;
use strict;


my $url = "https://www.uniprot.org/uniprot/Q16539.xml";
my $browser = LWP::UserAgent->new;

my $response = $browser->get($url,
  'User-Agent' => 'Mozilla/4.0 (compatible; MSIE 7.0)');

if ($response->is_success) {
  # We got something.
  print 'Content type is ', $response->content_type, "\n";

  if ($response->content_type eq "application/xml") {
    print "Process an XML response\n";
    print $response->content;
    my $xml = new XML::Simple();
    my $data = $xml->XMLin($response->content);
    print Dumper($data);
  }
} else {
  print "Error getting $url\n";
}
```

While looking at this code, **view** the actual page in a browser (https://www.uniprot.org/uniprot/Q16539.xml). The following has the first part of the XML from UniProt.

```
<uniprot xsi:schemaLocation="https://uniprot.org/uniprot
https://www.uniprot.org/support/docs/uniprot.xsd">
<entry dataset="Swiss-Prot" created="1997-11-01" modified="2013-04-03"
version="164">
<accession>Q16539</accession>
<accession>A6ZJ92</accession>
<accession>A8K6P4</accession>
<accession>B0LPH0</accession>
<accession>O60776</accession>
<accession>Q13083</accession>
<accession>Q14084</accession>
<accession>Q8TDX0</accession>
<name>MK14_HUMAN</name>
```

NOTE: it is important to view the raw XML or JSON that will be returned so you know how to parse it. This means looking at the raw data at the source if possible and viewing the output of Data Dumper. For Homework #2 (assigned next week) you will save yourself time if you view the data before jumping into coding. (You'll also save yourself time if you try the example code scripts in this module and understand the concepts and how these scripts work before you begin Homework #2.)

We can see the raw XML with print $response->content; the following shows the first part of what gets printed out (see uniprot_1.pl).

```
Content type is application/xml
Process an XML response
<?xml version='1.0' encoding='UTF-8'?>
<uniprot xmlns="https://uniprot.org/uniprot"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://uniprot.org/uniprot
https://www.uniprot.org/support/docs/uniprot.xsd">
<entry dataset="Swiss-Prot" created="1997-11-01" modified="2014-07-09"
version="179">
<accession>Q16539</accession>
<accession>A6ZJ92</accession>
<accession>A8K6P4</accession>
<accession>B0LPH0</accession>
<accession>O60776</accession>
```

And next is the first part of the output, as shown by Data::Dumper (it looks similar to JSON – but the data returned by our browser->get call was XML as shown above when printing $response->content). Notice that the **order of the information is different** from what is found on the XML page when you visit it in the browser (which is fine). (See file uniprot_2.pl.) In XML the order of child nodes can vary.

```
Content type is application/xml
Process an XML response
$VAR1 = {
          'xmlns' => 'http://uniprot.org/uniprot',
          'entry' => {
                     'keyword' => {
                                  'KW-0808' => {
                                                 'content' => 'Transferase'
                                               },
                                  'KW-0597' => {
                                                 'content' => 'Phosphoprotein'
                                               },
                                  'KW-0007' => {
                                                 'content' => 'Acetylation'
                                               },
                                  'KW-0053' => {
                                                 'content' => 'Apoptosis'
                                               },
                                  'KW-0346' => {
                                                 'content' => 'Stress response'
                                               },
                                  'KW-0832' => {
                                                 'content' => 'Ubl conjugation'
                                               },
```

Lines deleted...

```
                     'name' => 'MK14_HUMAN',
                     'dataset' => 'Swiss-Prot',
                     'accession' => [
                                      'Q16539',
                                      'A6ZJ92',
                                      'A8K6P4',
                                      'B0LPH0',
                                      'O60776',
                                      'Q13083',
```

```
'Q14084',
'Q8TDX0'
]
```

More lines deleted…

From the above, we can see that the **accession** data is found through a key called accession in a hash table, and that the accession value is an array (note the []). If you look above where 'entry' => { is, you can see that entry is a hash table because of the {, which has 'keyword', 'name', 'dataset', and 'accession' as keys. 'entry' is also a key in the top-level hash table which contains all the XML, processed into hashes and arrays.

We need to know how the data is stored so we can use the right dereferencing to get the information out of the data string returned. When the statement

```
my $data = $xml->XMLin($response->content);
```

is executed, $data has the XML, but not as a simple string – it has been deserialized into the appropriate structure – namely, all those nested hashes and arrays.

We want to get to the value for the key "entry" in the first level of the data returned. 'entry' is a key, so use this syntax `$hash_var->{key_name}`:

```
my $entry_node = $data->{entry};
```

If this entry node exists, the variable will be defined and will have the hash table. We can step through each key of the table and get the value for each key. Here's what that code looks like:

```
# This gives the level under the entry node
 my $entry_node = $data->{entry};
 if (defined($entry_node)) {
   while (my ($key, $value) = each(%$entry_node)) {
     # Some are true values, but some are references to other structures
     print "$key gives $value\n";
   }
```

And on the next page is the output for that code snippet. Notice that version (164), name, dataset, modified, and created have values that can be output directly. Everything else is a reference. **The kind of reference shows how we have to dereference the value**. Here is the entry_node loop output (partial):

```
keyword gives HASH(0x1bccd94)
sequence gives HASH(0x2adfa74)
version gives 164
gene gives HASH(0x27a3c44)
name gives MK14_HUMAN
dataset gives Swiss-Prot
accession gives ARRAY(0x2af7ba4)
modified gives 2013-04-03
feature gives ARRAY(0x27c7a44)
proteinExistence gives HASH(0x2a915f4)
dbReference gives HASH(0x1bcd174)
created gives 1997-11-01
protein gives HASH(0x27a30dc)
comment gives ARRAY(0x2aea5ec)
reference gives HASH(0x1bd17c4)
evidence gives HASH(0x2ae425c)
organism gives HASH(0x27a4034)
```

One of the keys is “accession” and it has an ARRAY reference as its value (which matches what we would expect based on the Data::Dumper output). So we use an **array dereference** to get to it. Note the @ in front of $acc_node.

```
# Know accession gives an array from Data::Dumper
my $acc_node = $entry_node->{accession};
if (defined($acc_node)) {
  foreach my $val (@$acc_node) { # array dereference
    print "Accession: $val\n";
  }
}
```

And the output from the print statement:

```
Accession: Q16539
Accession: A6ZJ92
Accession: A8K6P4
Accession: B0LPH0
Accession: O60776
Accession: Q13083
Accession: Q14084
Accession: Q8TDX0
```

In UniProt, the dbReference gives links to other databases where data originated from. For example, if you view the XML at UniProt for an entry, it will list all the PDB structure records that are available for the protein, if any. Here is one PDB:

```
<dbReference type="PDB" id="3FMN">
  <property type="method" value="X-ray"/>
  <property type="resolution" value="1.90"/>
  <property type="chains" value="A=1-360"/>
</dbReference>
```

Based on the output from the entry-node loop on the previous page, dbReference is a HASH reference, so we need to dereference the variable's value with a %. (If the variable contains a HASH reference, you want to get what the reference is pointing to – do this with the **%** in front of the $variable name.)

Note that dbReference refers to many other items besides PDB entries (refer to the original XML). But if you just want PDB id references, you need to add a test to extract only those dbReferences. Here is the code snippet (but *there is a problem*! See further down):

```
# Know dbReference is a hash
my $dbref = $entry_node->{dbReference};
if (defined($dbref)) {
  while (my ($key, $value) = each(%$dbref)) {
    while (my ($subkey, $subvalue) = each(%$value)) {
      print "$key is  $subkey = $subvalue\n";
      if (($subkey eq "type") and ($subvalue eq "PDB")) {
        # For Q16539, it's only finding the PDBsum's with the same id
        print "dbRef $key is a PDB ID, $value\n";
      }
    }
  }
}
```

This code gives output like this:

```
IPI00221141 is  type = IPI
CR536505 is  type = EMBL
CR536505 is  property = ARRAY(0x2965f8c)
EU332860 is  type = EMBL
EU332860 is  property = ARRAY(0x2965f3c)
1WBO is  type = PDBsum
NP_001306.1 is  type = RefSeq
NP_001306.1 is  property = HASH(0x2a7a49c)
3RIN is  type = PDBsum
```

PROBLEM: Try the code yourself and look through the output. You will **not** see any information where the type is just PDB (only PDBsum – not what we want). What has happened to the PDB entries?

IMPORTANT: In this XML, the identifier is being used as the key in the hash table for both PDB and PDBsum. You can only have 1 key that is 3RIN or 3FMN, for example, in the table. So we have to tell the XMLin function that we do not want these duplicate keys to be overwriting each other. Instead we will have to force this hash to be an array. To do that we change the line that parses the XML to the following (see uniprot_3.pl):

SOLUTION:

```
my $data = $xml->XMLin($response->content,
                       forcearray=>[qw(dbReference)],
                       keyattr=>[]);
```

HOWEVER: Note now the dbReference will be an ARRAY, not a hash. This change means we can't access the values like before using a key because they aren't associated with keys any longer.

```
      print "Handle dbref now\n";
      my $dbref = $entry_node->{dbReference};

      if (defined($dbref)) {
        foreach my $val (@$dbref) { # this is an array dereference now
          # Each dbReference's $val is a hash.
          # But since forced away the key, we don't have it any more. So do
this instead:
          if ($val->{type} && ($val->{type} eq "PDB") && ($val->{id})) {
            print "DB ref " . $val->{id} . " is a PDB\n";
            push(@my_pdbs, $val->{id});
          } else {
            print "NOT PDB " . $val->{id} . "  " . $val->{type} . "\n";
          }
        }
      } else {
        print "No dbReferences\n";
      }
```

→Be sure you understand how to access arrays and hashes and what to do if your hash has duplicate keys. Once you understand how to access hash tables, arrays, and nested structures, you will find the same type of syntax works for JSON data as well.

And, once you can extract the information from the structure, you can output it in any format you like. See the end of uniprot_3.pl for examples. But briefly, here is capturing PDB identifiers extracted from the UniProt XML.

```
# Declare array
my @my_pdbs;

# Example line from the code loop on previous page
# Save the PDB id in the my_pdbs array:
push(@my_pdbs, $val->{id});
```

We can output our own XML in this fashion:

```
  my %my_xml = ("UniProt" => $uniprot_id,
                "PDB" => \@my_pdbs);  # pass in reference to array
  my $out_xml = new XML::Simple(NoAttr=>1, RootName=>'Proteins');
  my $out_data = $out_xml->XMLout(\%my_xml);
  print "$out_data\n";
```

When you try this script, you'll see that the UniProt and PDB elements are not nested. You can use XML::Simple for outputting nested XML elements, but it can get tricky. For creating more complicated XML for output, you should use XML::LibXML or XML::Twig. Another option is XML::Writer. This API has methods startTag and endTag to output opening and closing tags. It also has a method, dataElement, which lets you write elements. Depending upon how you call these methods, you can build nested XML output. You may want to look at XML:Writer for Homework 2 (but it is not required).

Example with JSON

I've captured the response of a web service that returns data in JSON format to a file and have and have put that file in the Module 2 examples zip so you can experiment with JSON. It gives information about chemical structures that match a query structure. The following shows just part of the response.

{"status":"OK","timestamp":"2013-04-03 23:14:24 +0000","error_message":"","data":
{"query":{…},
"results":{"total_hits":37,
"structures":{"1017607":{"id":"1017607", … ,"mol_weight":333.3900146484375, …
"log_p":3.134660005569458, … },"1443406":{"id":"1443406", … }, ….
},"score":{}}}}

In this case, the returned information is a hash table made of nested hash tables. The following code will read in the file using a module called File::Slurp, which is very handy (but not needed for JSON, so another method for reading in the file into one string can also be done), then do the actual parsing, and finally will output all the structure keys with the molecular weight and log_p value for each structure.

If you don't have the File module, you can read in the file the "regular" way – one line at a time or all at once with File::Slurper). For large files you will need to do one line at a time, but this is a very small file. (See json_1.pl.)

```
use JSON;
use File::Slurp; # just handy to use
use strict;
use warnings;

# Gets the whole file all at once (here into a string)
my $entire_file = read_file("sc_response.txt"); # File::Slurp

# This simulates decoding a json string returned from a web service:
my $decoded_json = decode_json($entire_file);

print "Status: " .$decoded_json->{status} . "\n";
print "# Hits: " . $decoded_json->{data}->{results}->{total_hits} . "\n";

my $structures_node = $decoded_json->{data}->{results}->{structures};
while (my ($key, $value) = each(%$structures_node)) {
  print "Structure key $key: ";
  print "mol wgt: " . $value->{mol_weight} . " ";
  print "log_p: " . $value->{log_p} . "\n";
}
```

Here is the first part of the output:

```
Status: OK
# Hits: 37
Structure key 14806035: mol wgt: 574.432006835938 log_p: 5.10949993133545
Structure key 14806211: mol wgt: 527.02197265625 log_p: 4.62154006958008
```

For an example of creating a data structure and outputting it as JSON, see http://stackoverflow.com/questions/8463919/how-to-convert-a-simple-hash-to-json-in-perl

Here is a simple example using the xml created at the end of uniprot_3.pl:

```
  my $json = encode_json(\%my_xml); # pass in a reference to the hash: \%
  print "$json\n";
```

EXERCISE: Be sure you can run the script json_1.pl in the mod02_example_files.zip. One of the scripts for Homework 3 will require JSON.

Example with SQLite

This is an example that creates a table and puts some values into it. Notice that because the create command has an "or die" clause associated with it, if the create cannot finish successfully (such as if the table already exists), the script exits. (See sqltest_1.pl.) The script could be extended to add an if test around the create and to skip the create and insert statements if the table is there. You should add such tests to your code for the 2nd homework (not something to worry about for this week).

```
use DBI;
use strict;
use warnings;

my $dbfile = "database1.db";
my $dbh = DBI->connect("dbi:SQLite:dbname=$dbfile");

# Create a table PDB with two columns:
$dbh->do("create table PDB (PDBId, method)") or die $DBI::errstr;
# Insert some values
$dbh->do("insert into PDB values ('3FMN', 'X-Ray')");
$dbh->do("insert into PDB values ('2LTY', 'NMR')");

# Get all the rows. Uppercase SQL commands are OK, too.
my $all_rows = $dbh->selectall_arrayref("SELECT * FROM PDB");

foreach my $row (@$all_rows) { # an array ref from selectall_arrayref
  # Row is also an array reference.
  # Notice getting the information from the array into individual variables:
  my ($id, $method) = @$row;
  print "$id created with $method\n";
}

$dbh->disconnect;
```

Because the SQL commands are strings, we can use variables to build them. (See sqltest_2.pl.)

```
# Show example using variables
my $table = 'test';
my @rows = qw(id pdb data); # qw will make these items strings
my $sql_cmd = "create table $table(" .join(',', @rows) . ")";
print "sql command: $sql_cmd\n";
$dbh->do($sql_cmd) or die $DBI::errstr;

my @values = ("'Q16539'", "'3FMN'");
push(@values, "'Here is some data'");
$sql_cmd = "insert into $table values (" . join(',', @values) . ")";
print "sql command: $sql_cmd\n";
```

```
$dbh->do($sql_cmd) or die $DBI::errstr;
```

This causes the following output:

```
sql command: create table test(id,pdb,data)
sql command: insert into test values ('Q16539','3FMN','Here is some data')
```

And, we can do something more sophisticated with the above two tables. Notice that the result from the selectall_arrayref returns an array reference to all the rows. Each row is also an array reference to the columns. Both require @ to dereference them.

```
my $res = $dbh->selectall_arrayref( # selectall_arrayref returns array refs
   "SELECT a.id, a.pdb, b.method FROM PDB b, test a WHERE a.pdb = b.PDBId");

foreach my $row (@$res) {
  print join(" ", @$row);
  print "\n";
}
```

The above code, when added to the code that creates the tables, results in this output:

```
Q16539 3FMN X-Ray
```

Wrap Up

This module we focused on learning Perl. We also looked at other ways of obtaining information from databases/resources – SQL and RESTful web services. And we saw different ways data can be made available, in particular, XML and JSON.