Table of Contents

Bioinformatics

The field of "bioinformatics" is biology plus data science. For the latter, most people find Unix-like operating systems to be the most efficient way to conduct research. Almost all our tools run in Unix, most of them from the command line, so the bioinformatician must know how to move data around, run programs, and chain the output of one program into another to create analysis pipelines.

Metagenomics

Metagenomics is a subset of bioinformatics where we deal with unknown communities of organisms comprised of bacteria, viruses, fungi, archeae, etc. We deal exclusively with sequence-based data, usually measured in the gigabytes and -bases. Sequencing runs typically produce millions or tens of millions of reads. Our goal is to derive some useful knowledge from uncultured samples. We might try to identify pathogens in human wounds, protein function in open ocean water, or relative abundance of microbes in soil.

Code

All the code examples presented here can be found at:

```
https://github.com/kyclark/metagenomics-book
```

Authors and Contributors

- Ken Youens-Clark kyclark@gmail.com
- Bonnie Hurwitz bhurwitz@email.arizona.edu

About the authors

Ken

Ken went to college (the University of North Texas, 1990) thinking he might study music and become a professional drummer. He decided against that particular career but didn't have an alternate. He changed his major a couple of times (business, communications) before deciding on an English literature degree just so he could get out of undergrad in five years. NB: never did programming cross his mind, and biology was his most loathed science.

After college, Ken cast about for a job that 1) would be interesting and 2) people would pay him to do. At his first job (1995), he learned Microsoft's Access database on Windows 3.1 and created his company's first website by writing HTML into Notepad and using a Windows FTP client to upload to their ISP. The die was cast. His next job was writing a technical manual for a piece of software written in VisualBasic. He was offered a position to learn VB and support the program and another Access program. He went on to learn another database (DBase IV) and language (Delphi). His next job was developing Windows desktop applications in Delphi/Interbase, but he was itching to get into web applications. So his next job (c. 1998) was writing VBScript in Microsoft ASP with SQLServer as a database. At this point, Ken was fairly much fed up with Microsoft and rediscovered Unix.

Ken's first email client in college was "pine" running on the Unix machines which he accessed through the computer labs. He used "talk" to chat with friends, and so had been exposed to the celebrated "command line." Now he discovered that his ISP offered "shell" accounts on their servers accessible by "telnet." He was reading more about Unix on the Internet and how people were using the Perl programming language with CGI (common gateway interface) to create interactive web pages. The more Ken learned about Unix and Perl and "open source/free" software, the more he realized he'd found his tribe. His next job (1999) he moved into developing web apps on Linux platforms using the Apache web server with the MySQL database and Perl (the "LAMP" stack).

Around 2001, Ken saw that a very celebrated Perl developer named Lincoln Stein was looking to hire people. Ken got hired to work on the comparative plant genomics database called "Gramene," but really had no idea what Lincoln had done apart from his modules and books. Lincoln was a very important character in

a fairly new field called "bioinformatics" (cf. "How Perl Saved the Human Genome Project" https://www.foo.be/docs/tpj/issues/vol1_2/tpj0102-0001.html) and he ran a research lab at Cold Spring Harbor Laboratory in Cold Spring Harbor, NY. Lincoln hired Ken to write a web-based visual comparative map application (CMap, PMID: 19648141) to augment existing web genome browsers like the UCSC browser, the Ensembl browser, and Lincoln's own Gbrowse. This was Ken's entree into the world of biology and genomics. Around 2004, Lincoln hired Bonnie Hurwitz who left a few years later to earn her PhD from the University of Arizona. In 2014, Bonnie set up her new lab at the UofA and hired Ken to create the infrastructure for the iMicrobe project (http://imicrobe.us).

Ken currently enjoys research computing, teaching, and the pursuit of his MS at UA. He likes living in Tucson with his family (wife, three children), biking, cooking, and playing music.

Support

Do you like this book? Would you like to support the author? Consider a donation via PayPal to "kyclark@gmail.com"!

Why Unix?

There was a time when Excel spreadsheets were a legitimate tool for bioinformatics, but those days are long past. Sequencing runs produce massive quantities of data that must be moved from the sequencing facility to some place where you have plenty of data and processing power to complete your analysis. Unix-like operating systems are the de facto standard for almost all bioinformatics. I suggest you read the "Unix philosophy" (https://en.wikipedia.org/wiki/Unix_philosophy) to understand why this is so.

Our students must learn to use existing tools as well as create new programs to analyze data. Sometimes a new "tool" is just a simple shell script that chains together commands, but sometimes it is a suite of programs using shell, R, Perl, Python, and third-party binaries. To that end, half of our course is dedicated to teaching students about Unix, programming, and how to create and run pipelines.

Getting access to systems

Given that our class will have students on a variety of operating systems (Windows, OSX, Linux), we will use the HPC (high performance computing) cluster at the University of Arizona for our work. Students using the Windows operating system might find the Cygwin tools useful for getting access to tools like "ssh" and "scp." Students on OSX and Linux machines may choose to install software locally using package management tools like "homebrew" (OSX) or "apt-get" and "yum" (Linux).

University of Arizona HPC

We need to get into a Unix system. If you're on Linux or OSX, open a terminal and type:

```
ssh <NetID>@hpc.arizona.edu
```

If all goes well, you should see something like this:

```
Last login: Thu Aug 24 09:12:07 2017 from dhcp-10-142-130-41.uaw
ifi.arizona.edu
This is a bastion host used to access the rest of the environmen
t.

Shortcut commands to access each resource
-----------------------------------------
Ocelote:                El Gato:                Cluster(ICE)/HTC
/SMP:
$ ocelote               $ elgato                $ ice
```

The name of the machine should be "keymaster.hpc.arizona.edu." Verify this with the command "hostname". Use the command "menuon" to get the following menu:

```
===============
HPC.ARIZONA.EDU
===============
Please select a target system to connect to:

(1) Ocelote
(2) El Gato
(3) Cluster/HTC/SMP
(Q) Quit
(D) Disable menu
```

The menu will be present on your next login session. Make a selection from the menu to login to a head node.

As of 2016, Ocelot is UA's newest cluster. For more information, see https://confluence.arizona.edu/display/UAHPC/Ocelote+Quick+Start.

If you would like to avoid the 2-factor authentication, then read the following:

https://www.protocols.io/view/ssh-to-UA-HPC-fm7bk9n

Stampede (TACC)

The "stampede" cluster is another HPC resource that is freely available to our students. It is located at TACC, the Texas Advanced Computing Center at the University of Texas in Austin. Go to https://portal.tacc.utexas.edu/ to create an account. We recommend our students use the same username at TACC and Cyverse.

Cyverse (UA)

Cyverse is an NSF-funding cyberinfrastructure project headquartered at the University of Arizona. It began life as "iPlant" in 2008. Our students may choose to make use of Cyverse infrastructure, as well, such as "apps" for assemblies, gene calling, protein clustering, and such. Students should go to

https://user.cyverse.org/ to create an account. Again, we recommend you use the same username as your TACC account to make communication between the TACC and UA systems easier.

Docker, VirtalBox

If cannot gain access to the UA HPC or TACC, you can follow along at home using a virtual machine such as a Docker (https://www.docker.com/) or VirtualBox (https://www.virtualbox.org/) image installed on your machine.

First steps with Unix

I assume you are on a command line by now, so let's look at some commands.

- **whoami**: reports your username
- **w**: shows who is currently on a system
- **man**: show the manual page for a command
- **echo**: say something
- **cowsay**: have a cow say something
- **env**: print your environment
- **printenv**: print some/all of your environment
- **which/type**: tells you the location of a program
- **touch**: create an empty regular file
- **file**: briefly describe a file or directory
- **pwd**: print working directory, where you are right now
- **ls**: list files in current directory
- **find**: find files or directories matching some conditions
- **locate**: find files using a database, requires daemon to run
- **cd**: change directory (with no arguments, cd $HOME)
- **cp**: copy a file (or "cp -r" to copy a directory)
- **mv**: move a file or diretory
- **mkdir**: create a directory
- **rmdir**: remove a directory
- **rm**: remove a file (or "rm -r" to remove a directory)
- **cat**: concatenate files (cf. http://porkmail.org/era/unix/award.html)
- **column**: arrange text into columns
- **paste**: merge lines of files
- **sort**: sort text or numbers
- **uniq**: remove duplicates *from a sorted list*
- **sed**: stream editor for altering text
- **awk/gawk**: pattern scanning and processing language
- **grep**: global regular expression program (maybe?), cf. https://en.wikipedia.org/wiki/Regular_expression
- **history**: look at past commands, cf. CTRL-R for searching your history directly from the command line

- **head**: view the first few (10) lines of a file
- **tail**: view the last (10) lines of a file
- **comm**: find lines in common/unique in two sorted files
- **top**: view the programs taking the most system resources (memory, I/O, time, CPUs, etc.), cf. "htop"
- **ps**: view the currently running processes
- **cut**: select columns from the output of a program
- **wc**: (character) word (and line) count
- **more/less**: pager programs that show you a "page" of text at a time; cf. https://unix.stackexchange.com/questions/81129/what-are-the-differences-between-most-more-and-less/81131
- **bc**: calculator
- **df**: report file system disk space usage; useful to find a place to land your data
- **du**: report disk usage; recommend "du -shc"; useful to identify large directories that need to be removed
- **ssh**: secure shell, like telnet only with encryption
- **scp**: secure copy a file from/to remote systems using ssh
- **rsync**: remote sync; uses scp but only copies data that is different
- **ftp**: use "file transfer protocol" to retrieve large data sets (better than HTTP/browsers, esp for getting data onto remote systems)
- **ncftp**: more modern FTP client that automatically handles anonymous logins
- **wget**: web get a file from an HTTP location, cf. "wget is not a crime" and Aaron Schwartz
- **|**: pipe the output of a command into another command
- **>, >>**: redirect the output of a command into a file; the file will be created if it does not exist; the single arrow indicates that you wish to overwrite any existing file, while the double-arrows say to append to an existing file
- **<**: redirect contents of a file into a command
- **nano**: a very simple text editor; until you're ready to commit to vim or emacs, start here
- **md5sum**: calculate the MD5 checksum of a file
- **diff**: find the differences between two files

Pronunciation

- **/**: slash; the thing leaning the other way is a "backslash"
- **sh**: shuh or "ess-ach"
- **etc**: et-see
- **usr**: user
- **src**: source
- **#**: hash
- **$**: dollar
- **!**: bang
- **#!**: shebang
- **PID**: pid (not pee-eye-dee)
- **~**: twiddle or tilde; shortcut to your home directory when alone, shortcut to another user's home directory when used like "~bhurwitz"

Variables

You will see things like `$USER` and `$HOME` that start with the `$` sign. These are variables because they can change from person to person, system to system. On most systems, my username is "kyclark" but I might be "kclark" or "kyclark1" on others, but on all systems `$USER` refers to whatever value is defined for my username. Similarly, my `$HOME` directory might be "/Users/kyclark," "/home1/03137/kyclark," or "/home/u20/kyclark," but I can always refer to the idea of my home directory with the variable `$HOME`.

Make it stop!

If you launch a program that won't stop, you can use CTRL-C to send an "interrupt" signal to the program. If it is well-behaved, it should stop, but it may not. You can open another terminal on the machine and run `ps -fu $USER` to find all the programs you are running.

```
$ ps -fu $USER
UID        PID  PPID  C STIME TTY          TIME CMD
kyclark  31718 31692  0 12:16 ?        00:00:00 sshd: kyclark@pt
s/75
kyclark  31723 31718  0 12:16 pts/75   00:00:00 -bash
kyclark  33265 33247  0 12:16 ?        00:00:00 sshd: kyclark@pt
s/86
kyclark  33277 33265  1 12:16 pts/86   00:00:00 -bash
kyclark  33792 33277  9 12:17 pts/86   00:00:00 vim run.p6
kyclark  33806 31723  0 12:17 pts/75   00:00:00 ps -fu kyclark
```

The PID is the "process ID" and the PPID is the "parent process ID." In the above table, let's assume my "vim" session was locked up. I could `kill 33792`. If in a reasonable amount of time (a minute or so) that doesn't work, common wisdom says you `kill -9` to really, really tell it to shut down, but:

> No no no. Don't use kill -9.
>
> It doesn't give the process a chance to cleanly:
>
> - shut down socket connections
> - clean up temp files
> - inform its children that it is going away
> - reset its terminal characteristics
>
> and so on and so on and so on.
>
> Generally, send 15, and wait a second or two, and if that doesn't work, send 2, and if that doesn't work, send 1. If that doesn't, REMOVE THE BINARY because the program is badly behaved!
>
> Don't use kill -9. Don't bring out the combine harvester just to tidy up the flower pot.
> cf. http://porkmail.org/era/unix/award.html#uuk9letter

Along with CTRL-C, you should learn about CTRL-Z to put a process into the background. This could be handy if, say, you were in an editor, you make a change to your code, you CTRL-Z to background the editor, you run the script to see if it worked, then you `fg` to bring it back to the foreground or `bg` it to have

it resume running in the background. I would consider this a sub-optimal work environment, but it's fine if you were for some reason limited to a single terminal window.

Generally if you want a program to run in the background, you would launch it from the command line with an ampersand ("&") at the end:

```
$ my-background-prog.sh &
```

Lastly, know that most Unix programs interpret CTRL-D as the end-of-input signal. You can use this to send the "exit" command to most any interactive program, even your shell. Here's a way to enter some text into a file directly from the command line without using a text editor. After typing the last line (i.e., type "chickens.<Enter>"), type CTRL-D:

```
$ cat > wheelbarrow
so much depends
upon

a red wheel
barrow

glazed with rain
water

beside the white
chickens.
<CTRL-D>
$ cat wheelbarrow
so much depends
upon

a red wheel
barrow

glazed with rain
water

beside the white
chickens.
```

Handy command line shortcuts

- Tab: hit the Tab key for command completion; hit it early and often!
- **!!**: execute the last command again
- **!$**: the last argument from your previous command line (think of the $ as the right anchor in a regex)
- **!^**: the first argument from your previous command line (think of the ^ as the left anchor in a regex)
- CTRL-R: reverse search of your history
- Up/down cursor keys: go backwards/forwards in your history
- CTRL-A, CTRL-E: jump to the start, end of the command line when in emacs

mode (default)

> Protip: If you are on a Mac, it's easy to remap your (useless) CAPSLOCK key to CTRL. Much less strain on your hand as you will find you need CTRL quite a bit, even more so if you choose emacs for your $EDITOR.

Altering your environment

As you see above, "env" will list all the key-value pairs defining your environment. For instance, everyone has a $HOME directory that you can see with `echo $HOME`. The exact location of $HOME can vary among systems, e.g.:

- Mac: /Users/kyclark
- Ocelot: /home/u20/kyclark
- Stampede: /home1/03137/kyclark

Your $PATH setting is extremely important as it defines the directory locations that will be searched (in order) to find programs. Here's my $PATH on the UA HPC:

```
$ echo $PATH | sed "s/:/:\n/g"
/rsgrps/bhurwitz/hurwitzlab/bin:
/sbin:
/bin:
/usr/bin:
/usr/sbin:
/usr/lpp/mmfs/bin:
/usr/pbs/bin:
/var/spool/pas/repository/pas-appmaker:
/opt/sgi/sbin:
/opt/sgi/bin:
/usr/local/bin:
/home/u20/kyclark/bin:
/home/u20/kyclark/perl5/bin:
/rsgrps/bhurwitz/hurwitzlab/tools/bpipe-0.9.9/bin:
/home/u20/kyclark/.local/bin:
/home/u20/kyclark/.rakudobrew/bin:
/home/u20/kyclark/bin
```

I've used "sed" to add a newline after each colon so you can more easily see that the directories are separated by colons. Notice that I have the shared "hurwitzlab" directory first in my path. Much of our work will require access to tools that are not installed by default on the HPC. You could build them into your own $HOME directory, but it will be easier if you just add this shared directory to your $PATH. From the command line, you can do this:

```
PATH=/rsgrps/bhurwitz/hurwitzlab/bin:$PATH
```

You just told your shell (bash) to set the PATH variable to our "hurwitzlab" directory and then whatever it was set to before. Obviously we want this to happen each time we log in, so we can add this command to "$HOME/.bashrc":

```
echo "PATH=/rsgrps/bhurwitz/hurwitzlab/bin:$PATH" >> ~/.bashrc
```

> Side note: Dotfiles (https://en.wikipedia.org/wiki/Hidden_file_and_hidden_directory) are files with names that begin with a dot. They are normally hidden from view unless you use "ls -a" to list "all" files. A single dot "." means the current directory, and two dots ".." mean the parent directory.

Your ".bashrc" (or maybe ".profile" or maybe ".bash_profile" depending on your system) file is read every time you login to your system, so you can remember your customizations.

Aliases

Sometimes you'll find you're using a particular command quite often and want to create a shortcut. You can assign any command to a single "alias" like so:

```
alias cx='chmod +x'
alias up2='cd ../../'
alias up3='cd ../../../'
```

If you execute this on the command line, the alias will be saved until you log out. Adding these lines to your .bashrc will make it available every time you log in. When you make a change and want the shell to bring those into the current environment, you need to "source" the file. The command "." is an alias for "source":

```
$ source ~/.bashrc
$ . ~/.bashrc
```

Permissions

When you execute "ls -l," you'll see the "long" listing of the contents of a directory similar to this:

```
-rwxr-xr-x   1 kyclark  staff     174 Aug  9 20:21 abs.py*
drwxr-xr-x  14 kyclark  staff     476 Aug  3 12:14 anaconda3/
```

The first column of data contains a wealth of information represent in 10 bits. The first bit is:

- "-" for a regular file
- "d" for a directory
- "l" for a symlink (like a shortcut)

The other nine bits are broken into sets of three bits that represent the permissions for the "user," "group," and "other." The "abs.py" is a regular file we can tell from the first dash. The next three bits show "rwx" which means that the user ("kyclark") has read, write, and execute permissions for this file. The next three bits show "r-x" meaning that the group ("staff") can read and execute the file only. The same is true for all others.

When you create a file, the normal default is that it is not executable. You must specifically tell Unix to change the mode of the file using the "chmod" command. Often it's enough to say:

```
$ chmod +x myprog.sh
```

To turn on the execute bits for everyone, but it's possible to have much finer control of the permissions. If you only want user and group to have execute, then do:

```
$ chmod ug+x myprog.sh
```

Removing is done with a "-," so any combination of "[ugo][+-][rwx]" will usually get you what you want.

Sometimes you may see instructions to "chmod 775" a file. This is using octal notation where the three bits "rwx" correspond to the digits "421," so the first "7" is "4+2+1" which equals "rwx" whereas the "5" = "4+1" so only "rw":

```
 user   group   other
r w x   r w x   r w x
4 2 1 | 4 2 1 | 4 2 1
+ + +   + + +   + - +
 = 7     = 7     = 5
```

Therefore "chmod 775" is the same as:

```
$ chmod -rwx myfile
$ chmod ug+rwx myfile
$ chmod o+rw myfile
```

When you create ssh keys or config files, you are instructed to `chmod 600`:

```
 user   group   other
r w x   r w x   r w x
4 2 1 | 4 2 1 | 4 2 1
+ + -   - - -   - - -
 = 6     = 0     = 0
```

Which means that only you can read or write the file, and no one else can do anything with it. So you can see that it can be much faster to use the octal notation.

When you are trying to share data with your colleagues who are on the same system, you may put something into a shared location but they complain that they cannot read it or nothing is there. The problem is most likely permissions. The "uask" setting on a system determines the default permissions, and it may be that the directory and/or files are readable only by you. It may also be that you are not in a common group that you can use to grant permission, in which case you can either:

- politely ask your sysadmin to create a new group OR
- `chmod 777` the directory, which is probably the worst option as it makes the directory completely accessible to anyone to do anything. In short, don't do this unless you really don't care if someone accidentally or maliciously wipes out your data.

File system layout

The top level of a Unix file system is "/" which is called "root." Confusingly, there is also an account named "root" which is basically the super-user/sysadmin (systems administrator). Unix has always been a multi-tenant system ...

Installing software

Much of the time, "bioinformatics" seems like little more than installing software and chaining them together with scripts. Sometimes you may be lucky enough to have a "sysadmin" (systems administrator) who can assist you, but most of the time you'll find yourself needing to take care of business yourself.

My suggestions for installing software are (in order):

Sysadmin

Go introduce yourself to your sysadmins. Take them to lunch or order them some pizza or drop off some good beer or whiskey. Whatever it takes to be on good terms because a good sysadmin who is responsive to your needs is an enormous help. If they are willing to install software for you, that is the way to go. Often, though, this is a task far beneath them, and they would expect you to be able to fend for yourself. They may provide "sudo" (https://xkcd.com/149/) privileges to allow you to install software into shared locations (e.g., "/usr/local"), but it's more likely they would expect you to install into your $HOME.

Package managers

There are several package management systems for Linux and OSX including apt-get, yum, homebrew, macports, and more. These usually relieve the problems of software compatibility and shared libraries. Unless you have "sudo" to install globally, you can configure to install into your $HOME.

Binary installations

Quite often you'll be happy to find that the maintainers of the software you need have gone to the trouble to build binary distributions for your system, which is likely to be a generic 64-bit Linux platform. Often you can just download the binaries and put them into your $PATH. There is usually a "README" or "INSTALL" file that will explain exactly what to do.

Source installations

Installing from source usually means downloading a "tarball" ("tar" = "tape archive," a container of files, that is then compressed with a program like "gzip" to create a ".tar.gz" or ".tgz" file extension), running "configure" to figure out how it can build on your system, and then "make" to build the binaries. Usually you will run "make install" to put the binaries into their proper directory, but sometimes you just "make" and copy the files yourself.

The basic steps are usually:

```
$ tar xvf package.tgz
$ ./configure [--prefix=$HOME/local]
$ make && make install
```

When I'm in an environment with a directory I can share with my team (like the UA HPC), I'll configure the package to install into that shared space so that others can use the program. When I'm on a system like "stampede" where I cannot share with others, I'll usually install into my "$HOME/local" directory.

Unix exercises

> Note: When you see a "$" given in the example prompts, it is a metacharacter indicating that this is the prompt for a normal (not super-user) account. Your default prompt may be different, and it is highly configurable (search for "PS1 unix prompt" to learn more). Anyway, point is that you should type (copy/paste) all the stuff *after* the $. If you ever see a prompt with "#," it's indicating a command that should be run as the super-user/root account, e.g., installing some software into a system-wide directory so it can be shared by all users.

Number of unique users

Find the number of unique users on a shared system

We know that "w" will tell us the users logged in. Try it now on a system that has many users (i.e., not your laptop) and see the output. We'll connect the output of "w" to "head" using a pipe "|", but we only want the first five lines:

```
$ w | head -5
 14:36:01 up 21 days, 21:51, 176 users,  load average: 3.83, 4.3
1, 4.47
USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU
WHAT
antontre pts/1    149.165.156.129  Sat14    3days  0.10s  0.10s
/bin/sh -i
huddack  pts/3    128.4.131.189    09:38    4:56m  0.15s  0.15s
-bash
```

We want to see the first five *users*, not the first five lines of output. To skip the first two lines of headers from "w," we first pipe "w" into "awk" and tell it we only want to see output when the Number of Records (NR) is greater than 2:

```
$ w | awk 'NR>2' | head -5
antontre pts/1    149.165.156.129  Sat14    4days  0.10s  0.10s
/bin/sh -i
huddack  pts/3    128.4.131.189    09:38    5:13m  0.15s  0.15s
-bash
antontre pts/5    149.165.156.129  Sun19    2days  0.14s  0.14s
/bin/sh -i
minyard  pts/8    129.114.64.18    29Jul16  4:24m  3:46m  3:46m
top
antontre pts/11   149.165.156.129  Sun23    2days  0.24s  0.24s
/bin/sh -i
```

Let's "cut" out just the first column of data. The manpage for "cut" says that it defaults to using the tab character to determine columns, so we'll need to tell it to use spaces:

```
$ w | awk 'NR>2' | head -5 | cut -d ' ' -f 1
antontre
huddack
antontre
minyard
antontre
```

We can see right away that the some users like "antontre" are logged in multiple times, so let's "uniq" that output:

```
$ w | awk 'NR>2' | head -5 | cut -d ' ' -f 1 | uniq
antontre
huddack
antontre
minyard
antontre
```

Hmm, that's not right. Remember I said earlier that "uniq" only works *on sorted input*? So let's sort those names first:

```
$ w | awk 'NR>2' | head -5 | cut -d ' ' -f 1 | sort | uniq
antontre
huddack
minyard
```

OK, that is correct. Now let's remove the "head -5" and use "wc" to count all the lines (-l) of input:

```
$ w | awk 'NR>2' | cut -d ' ' -f 1 | sort | uniq | wc -l
138
```

So what you see is that we're connecting small, well-defined programs together using pipes to connect the "standard input" (STDIN) and "standard output (STDOUT) streams. There's a third basic file handle in Unix called "standard error" (STDERR) that we'll come across later. It's a way for programs to report problems without simply dying. You can redirect errors into a file like so:

```
$ program 2>err
$ program 1>out 2>err
```

The first example puts STDERR into a file called "err" and lets STDOUT print to the terminal. The second example captures STDOUT into a file called "out" while STDERR goes to "err."

> Protip: Sometimes a program will complain about things that you cannot fix, e.g., "find" may complain about file permissions that you don't care about. In those cases, you can redirect STDERR to a special filehandle called "/dev/null" where they are forgotten forever. Kind of like the "memory hole" in 1984.

```
find / -name my-file.txt 2>/dev/null
```

Count "oo" words

On almost every Unix system, you can find "/usr/share/dict/words." Let's use "grep" to find how many have the "oo" vowel combination. It's a long list, so I'll pipe it into "head" to see just the first five:

```
$ grep 'oo' /usr/share/dict/words | head -5
abloom
aboon
aboveproof
abrood
abrook
```

Yes, that works, so redirect those words into a file and count them:

```
$ grep 'oo' /usr/share/dict/words > oo-words
$ wc -l !$
10460 oo-words
```

Let's count them directly out of "grep":

```
$ grep 'oo' /usr/share/dict/words | wc -l
10460
```

How many of those words additionally contain the "ow" sequence?

```
$ grep 'oo' /usr/share/dict/words | grep 'ow' | wc -l
158
```

How many *do not* contain the "ow" sequence?

```
$ grep 'oo' /usr/share/dict/words | grep -v 'ow' | wc -l
10302
```

Do those numbers add up?

```
$ bc <<< 158+10302
10460
```

Excellent. Smithers, massage my brain.

Gapminder

Do the following:

```
$ git clone https://github.com/kyclark/metagenomics-book
$ cd metagenomics-book/problems/gapminder/data
```

How many files are in the "data" directory?

```
$ ls | wc -l
```

How many lines are in each/all of the files?

```
$ wc -l *
```

You can use `cat` to spew at the entire contents of a file into your shell, but if you'd just like to see the top of a file, you can use:

```
$ head Trinidad_and_Tobago.cc.txt
```

If you only want to see 5 lines, use `-n 5` or `-5` .

For our exercise, we'd like to combine all the files into one file we can analyze. That's easy enough with:

```
$ cat * > all.txt
```

Let's use `head` to look at the top of file:

```
$ head -5 all.txt
Afghanistan    1997    22227415    Asia    41.763    635.341351
Afghanistan    2002    25268405    Asia    42.129    726.7340548
Afghanistan    2007    31889923    Asia    43.828    974.5803384
Afghanistan    1952    8425333    Asia    28.801    779.4453145
Afghanistan    1957    9240934    Asia    30.332    820.8530296
```

Hmm, there are no column headers. Let's fix that. There's one file that's pretty different in content (it has only one line) and name ("country.cc.txt"):

```
$ cat country.cc.txt
country    year    pop    continent    lifeExp    gdpPercap
```

Those are the headers that you can combine to all the other files to get named columns, something very important if you want to look at the data in Excel and R/Python data frames.

```
$ rm all.txt
$ mv country.cc.txt headers
$ cat headers *.txt > all.txt
$ head -5 all.txt | column -t
country      year  pop       continent  lifeExp  gdpPercap
Afghanistan  1997  22227415  Asia       41.763   635.341351
Afghanistan  2002  25268405  Asia       42.129   726.7340548
Afghanistan  2007  31889923  Asia       43.828   974.5803384
Afghanistan  1952  8425333   Asia       28.801   779.4453145
```

Yes, that looks much better. Double-check that the number of lines in the `all.txt` match the number of lines of input:

```
$ wc -l *.cc.txt headers
$ wc -l all.txt
```

How many observations do we have for 1952?

```
$ grep 1952 all.txt | wc -l
$ cut -f 2 *.cc.txt | grep 1952 | wc -l
```

Those numbers aren't the same! Why is that?

```
$ grep 1952 all.txt | cut -f 2 | sort | uniq -c
 142 1952
   1 1982
   1 1987
$ grep 1952 all.txt | grep 198[27]
Lebanon    1982    3086876    Asia    66.983    7640.519521
Mozambique    1987    12891952    Africa    42.861    389.876184
6
```

How many observations for every year?

```
$ cut -f 2 *.cc.txt | sort | uniq -c
```

How many observations are present for Africa?

```
$ grep Africa all.txt | wc -l
```

How many for each continent?

```
$ cut -f 4 *.cc.txt | sort | uniq -c
```

What was the world population in 1952? As we've seen, just using `grep 1952` is not sufficient. We want to take the 3rd column if the 2nd column is equal to "1952." `awk` will let us do just that. Normally `awk` will split on whitespace, so we need to use `-F"\t"` to tell it to split on the tab (`\t`) character. Use `man awk` to learn more.

```
$ awk -F"\t" '$2 == "1952" { print $3 }' *.cc.txt
```

I'll bet you didn't notice that one of those numbers was in scientific notation. That's going to cause a problem. Here it is:

```
$ awk -F"\t" '$2 == "1952" { print $3 }' *.cc.txt | grep [a-z]
3.72e+08
```

We have to throw in a `grep -v` to get rid of that (the `-v` reverses the match), then use the `paste` command is used to put a "+" in between all the numbers:

```
$ awk -F"\t" '$2 == "1952" { print $3 }' *.cc.txt | grep -v [a-z
] | paste -sd+ -
```

and then we pipe that to the `bc` calculator:

```
$ awk -F"\t" '$2 == "1952" { print $3 }' *.cc.txt | grep -v [a-z
] | paste -sd+ - | bc
2034957150.999989
```

It bothers me that it's not an integer, so I'm going to use `printf` in the `awk` command to trim that:

```
$ awk -F"\t" '$2 == "1952" { printf "%d\n", $3 }' *.cc.txt | gre
p -v [a-z] | paste -sd+ - | bc
2406957150
```

How did population change over the years? Let's put a list of the unique years into a file called "years" and then `cat` over that to run the above for each year:

```
$ cut -f 2 *.txt | sort | uniq > years
$ for year in `cat years`; do echo -n $year ": " && awk -F"\t" "
\$2 == $year { printf \"%d\n\", \$3 }" *.cc.txt | grep -v [a-z]
| paste -sd+ - | bc; done
1952 : 2406957150
1957 : 2664404580
1962 : 2899782974
1967 : 3217478384
1972 : 3576977158
1977 : 3930045807
1982 : 4289436840
1987 : 4691477418
1992 : 5110710260
1997 : 5515204472
2002 : 5886977579
2007 : 6251013179
```

That's kind of useful! Here's how I might put that into a script:

```
$ cat pop-years.sh
#!/bin/bash

set -u

YEARS="years"

cut -f 2 ./*.cc.txt | sort | uniq > "$YEARS"
NUM=$(wc -l $YEARS | awk '{print $1}')

if [[ "$NUM" -lt 1 ]]; then
  echo "No years ($NUM)!"
  exit 1
fi

while read -r YEAR; do
    echo -n "$YEAR: "
    awk -F"\t" "\$2 == $YEAR { printf \"%d\\n\", \$3 }" ./*.cc.t
xt | grep -v "[a-z]" | paste -sd+ - | bc
done < "$YEARS"
$ ./pop-years.sh
1952: 2406957150
1957: 2664404580
1962: 2899782974
1967: 3217478384
1972: 3576977158
1977: 3930045807
1982: 4289436840
1987: 4691477418
1992: 5110710260
1997: 5515204472
2002: 5886977579
2007: 6251013179
```

How has life expectancy changed over the years? For this we'll need to write a little Python program. I'll `cat` the program so you can see it. You can type this in with `nano` and then do `chmod +x avg.py` to make it executable (or use the one I added):

```
$ cat avg.py
#!/usr/bin/env python3

import sys

args = list(map(float, sys.argv[1:]))
print(str(sum(args) // len(args)))
$ for year in `cat years`; do echo -n "$year: " && grep $year *.
txt | cut -f 5 | xargs ./avg.py; done
1952: 49.0
1957: 51.0
1962: 53.0
1967: 55.0
1972: 57.0
1977: 59.0
1982: 61.0
1987: 63.0
1992: 64.0
1997: 65.0
2002: 65.0
2007: 66.0
```

How many observations where the life expectancy ("lifeExp," field #5) is greater than 40? For this, let's use the `awk` tool.

```
$ awk -F"\t" '$5 > 40' all.txt | wc -l
```

How many of those are from Africa? We can either use `cut` to get the 4th field or ask `awk` to print the 4th field for us:

```
$ awk -F"\t" '$5 > 40' all.txt | cut -f 4 | grep Africa | wc -l
$ awk -F"\t" '$5 > 40 {print $4}' all.txt | grep Africa | wc -l
```

How many countries had a life expectancy greater than 70, grouped by year?

```
$ awk -F"\t" '$5 > 70 { print $2 }' *.cc.txt | sort | uniq -c
   5 1952
   9 1957
  16 1962
  25 1967
  30 1972
  38 1977
  44 1982
  49 1987
  54 1992
  65 1997
  75 2002
  83 2007
```

How could we add continent to this?

```
$ awk -F"\t" '$5 > 70 { print $2 ":" $4 }' *.cc.txt | sort | uni
q -c
```

As you look at the data and want to ask more complicated questions like how does `gdpPercap` affect `lifeExp`, you'll find you need more advanced tools like Python or R. Now that the data has been collated and the columns named, that will be much easier.

What if we want to add headers to each of the files?

```
$ mkdir wheaders
$ for file in *.txt; do cat headers $file > wheaders/$file; done
$ wc -l wheaders/* | head -5
      13 wheaders/Afghanistan.cc.txt
      13 wheaders/Albania.cc.txt
      13 wheaders/Algeria.cc.txt
      13 wheaders/Angola.cc.txt
      13 wheaders/Argentina.cc.txt
$ head wheaders/Vietnam.cc.txt
country    year    pop    continent    lifeExp    gdpPercap
Vietnam    1952    26246839    Asia    40.412    605.0664917
Vietnam    1957    28998543    Asia    42.887    676.2854478
Vietnam    1962    33796140    Asia    45.363    772.0491602
Vietnam    1967    39463910    Asia    47.838    637.1232887
Vietnam    1972    44655014    Asia    50.254    699.5016441
Vietnam    1977    50533506    Asia    55.764    713.5371196
Vietnam    1982    56142181    Asia    58.816    707.2357863
Vietnam    1987    62826491    Asia    62.82    820.7994449
Vietnam    1992    69940728    Asia    67.662    989.0231487
```

Something with sequences

Now we will get some sequence data from the iMicrobe FTP site. Both "wget" and "ncftpget" will do the trick:

```
$ mkdir -p ~/work/abe487/contigs
$ cd !$
$ wget ftp://ftp.imicrobe.us/abe487/contigs/contigs.zip
```

> Protip: How do we know we got the correct data? Go back and look at that FTP site, and you will see that there is a "contigs.zip.md5" file that we can "less" on the server to view the contents:

```
$ ncftp ftp://ftp.imicrobe.us/abe487/contigs
NcFTP 3.2.5 (Feb 02, 2011) by Mike Gleason (http://www.NcFTP.com
/contact/).
Connecting to 128.196.131.100...
Welcome to the imicrobe.us repository
Logging in...
Login successful.
Logged in to ftp.imicrobe.us.
Current remote directory is /abe487/contigs.
ncftp /abe487/contigs > ls
contigs.zip        contigs.zip.md5
ncftp /abe487/contigs > less contigs.zip.md5

1b7e58177edea28e6441843ddc3a68ab  contigs.zip
ncftp /abe487/contigs > exit
```

> You can read up on MD5 (https://en.wikipedia.org/wiki/Md5sum) to understand that this is a signature of the file. If we calculate the MD5 of the file we dowloaded and it matches what we see on the server, then we can be sure that we have the exact file that is on the FTP site:

```
$ md5sum contigs.zip
1b7e58177edea28e6441843ddc3a68ab  contigs.zip
```

> Yes, those two sums match. Note that sometimes the program is just called "md5."

So, back to the exercise. Let's unpack the contigs:

```
$ unzip contigs.zip
Archive:  contigs.zip
  inflating: group12_contigs.fasta
  inflating: group20_contigs.fasta
  inflating: group24_contigs.fasta
$ rm contigs.zip
```

These files are in FASTA format (https://en.wikipedia.org/wiki/FASTA_format), which basically looks like this:

```
>MCHU - Calmodulin - Human, rabbit, bovine, rat, and chicken
ADQLTEEQIAEFKEAFSLFDKDGDGTITTKELGTVMRSLGQNPTEAELQDMINEVDADGNGTID
FPEFLTMMARKMKDTDSEEEIREAFRVFDKDGNGYISAAELRHVMTNLGEKLTDEEVDEMIREA
DIDGDGQVNYEEFVQMMTAK*
>gi|5524211|gb|AAD44166.1| cytochrome b [Elephas maximus maximus
]
LCLYTHIGRNIYYGSYLYSETWNTGIMLLLITMATAFMGYVLPWGQMSFWGATVITNLFSAIPY
IGTNLV
EWIWGGFSVDKATLNRFFAFHFILPFTMVALAGVHLTFLHETGSNNPLGLTSDSDKIPFHPYYT
IKDFLG
LLILILLLLLLALLSPDMLGDPDNHMPADPLNTPLHIKPEWYFLFAYAILRSVPNKLGGVLALF
LSIVIL
GLMPFLHTSKHRSMMLRPLSQALFWTLTMDLLTLTWIGSQPVEYPYTIIGQMASILYFSIILAF
LPIAGX
IENY
```

Header lines start with ">", then the sequence follows. Sequences may be broken up over several lines of 50 or 80 characters, but it's just as common to see the sequences take only one (sometimes very long) line. Sequences may be nucleotides, proteins, very short DNA/RNA, longer contigs (shorter strands assembled into contiguous regions), or entire chromosomes or even genomes.

So, how many sequences are in "group12_contigs.fasta"? To answer, we just need to count how many times we see ">". We can do that with "grep":

```
$ grep > group12_contigs.fasta
Usage: grep [OPTION]... PATTERN [FILE]...
Try 'grep --help' for more information.
```

What is going on? Remember when we captured the "oo" words that we used the ">" symbol to tell Unix to *redirect* the output of "grep" into a file. We need to tell Unix that we mean a literal greater-than sign by placing it in single or double quotes or putting a backslash in front of it:

```
$ grep '>' group12_contigs.fasta
$ grep \> group12_contigs.fasta
```

You should actually see nothing because something quite insidious happened with that first "grep" statement -- it overwrote our original "group12_contigs.fasta" with the result of "grep"ing for nothing, which is nothing:

```
$ ls -l group12_contigs.fasta
-rw-rw---- 1 kyclark staff 0 Aug 10 15:08 group12_contigs.fasta
```

Ugh, OK, I have to go back and "wget" the "contigs.zip" file to restore it. That's OK. Things like this happen all the time.

```
$ ls -lh group12_contigs.fasta
-rw-rw---- 1 kyclark staff 2.9M Aug 10 14:38 group12_contigs.fas
ta
```

Now that I have restored my data, I want to count how many greater-than signs in the file:

```
$ grep '>' group12_contigs.fasta | wc -l
132
```

Hey, I could see doing that often. Maybe we should make this into an "alias" (see above). The problem is that the "argument" to the function (the filename) is stuck in the middle of the chain of commands, so it would make it tricky to use an alias for this. We can create a bash function that we add to our .bashrc:

```
function countseqs() {
  grep '>' $1 | wc -l
}
```

After you add this, remember to source this file to make it available:

```
$ source ~/.bashrc
$ countseqs group12_contigs.fasta
132
```

Same answer. Good. However, someone beat us to the punch. There is a powerful tool called "seqmagick" (https://github.com/fhcrc/seqmagick) that will do this (and much, much more). It's installed into the "hurwitzlab/bin" directory, or you can install it locally:

```
$ seqmagick info group12_contigs.fasta
name                  alignment    min_len   max_len   avg_len
num_seqs
group12_contigs.fasta FALSE           5136    116409  22974.30
     132
```

Run "seqmagick -h" to see everything it can do.

Moving on, let's find how many contig IDs in "group12_contigs.fasta" contain the number "47":

```
$ grep 47 group12_contigs.fasta > group12_ids_with_47
[login3@~/work/sequences]$ cat !$
cat group12_ids_with_47
>Contig_247
>Contig_447
>Contig_476
>Contig_1947
>Contig_4764
>Contig_4767
>Contig_13471
```

Here we did a little "useless use of cat," but it's OK. We also could have used "less" to view the file. Here's another useless use of cat to copy a file:

```
$ cat group12_ids_with_47 > temp1_ids
```

Additionally, we want to copy the file again to make duplicates:

```
$ cp group12_ids_with_47 temp2_ids
```

How can we be sure these files are the same? Let's use "diff":

```
$ diff temp1_ids temp2_ids
```

You should see nothing, which is a case of "no news is good news." They don't differ in any way. We can verify this with "md5sum":

```
$ md5sum temp*
957390ab4c31db9500d148854f542eee  temp1_ids
957390ab4c31db9500d148854f542eee  temp2_ids
```

They are the same file. If there were even one character difference, they would generate different hashes.

Now we will create a file with duplicate IDs:

```
$ cat temp1_ids temp2_ids > duplicate_ids
```

Check contents of "duplicate_ids" using "less" or "cat." Now grab all of the contigs IDs from "group20_contigs.fasta" that contain the number "51." Concatenate the new IDs to the duplicate_ids file in a file called "multiple_ids":

```
$ cp duplicate_ids multiple_ids
$ grep 51 group20_contigs.fasta >> !$
grep 51 group20_contigs.fasta >> multiple_ids
```

Notice the ">>" arrows to indicate that we are *appending* to the existing "multiple_ids" file.

Remove the existing "temp" files using a "*" wildcard:

```
$ rm temp*
```

Now let's explore more of what "sort" and "uniq" can do for us. We want to find which IDs are unique and which are duplicated. If we read the manpage ("man uniq"), we see that there are "-d" and "-u" flags for doing just that. However, we've already seen that input to "uniq" needs to be sorted, so we need to remember to do that:

```
$ sort multiple_ids | uniq -d > temp1_ids
$ sort multiple_ids | uniq -u > temp2_ids
$ diff temp*
1,7c1,11
< >Contig_13471
< >Contig_1947
< >Contig_247
< >Contig_447
< >Contig_476
< >Contig_4764
< >Contig_4767
---
> >Contig_10051
> >Contig_1651
> >Contig_4851
> >Contig_5141
> >Contig_5143
> >Contig_5164
> >Contig_5170
> >Contig_5188
> >Contig_6351
> >Contig_9651
> >Contig_9851
```

Let's remove our temp files again and make a "clean_ids" file:

```
$ rm temp*
$ sort multiple_ids | uniq > clean_ids
$ wc -l multiple_ids clean_ids
 25 multiple_ids
 18 clean_ids
 43 total
```

We can use "sed" to alter the IDs. The "s//" command say to "substitute" the first thing with the second thing, e.g., to replace all occurences of "foo" with "bar", use "s/foo/bar" (http://stackoverflow.com/questions/4868904/what-is-the-origin-of-foo-and-bar).

```
$ sed 's/C/c/' clean_ids
$ sed 's/_/./' clean_ids
$ sed 's/>//' clean_ids > newclean_ids
```

That last one removes the FASTA file artifact that identifies the beginning of an ID but is not part of the ID. We can use this with "seqmagick" now to extract those sequences and find out how many were found:

```
$ seqmagick convert --include-from-file newclean_ids group12_con
tigs.fasta newgroup12_contigs.fasta
$ seqmagick info !$
seqmagick info newgroup12_contigs.fasta
name                     alignment    min_len   max_len   avg_le
n  num_seqs
newgroup12_contigs.fasta FALSE           5587     30751  16768.1
4         7
```

We can get stats on all our files:

```
$ seqmagick info *fasta > fasta-info
$ cat !$
name                     alignment    min_len   max_len   avg_le
n  num_seqs
group12_contigs.fasta    FALSE           5136    116409  22974.3
0       132
group20_contigs.fasta    FALSE           5029     22601   7624.3
8       203
group24_contigs.fasta    FALSE           5024     81329  12115.7
0       139
newgroup12_contigs.fasta FALSE           5587     30751  16768.1
4         7
```

We can use "cut" to view various columns:

```
$ cut -f 2 fasta-info
$ cut -f 2,4 fasta-info
$ cut -f 2-4 fasta-info
```

But it does not line up very nicely. We can use "column" to fix this:

```
$ cut -f 2-4 fasta-info | column -t
alignment  min_len  max_len
FALSE      5136     116409
FALSE      5029     22601
FALSE      5024     81329
FALSE      5587     30751
```

Git basics

Linus Torvalds (https://en.wikipedia.org/wiki/Linus_Torvalds) is well known for creating the Linux operating system. Managing all the source code for Linux became onerous with existing systems, so he created "git" to do that, too. We'll learn enough git to get some things done.

Github

Github is a for-profit business that provides git hosting services. They offer free accounts for individuals and groups. When you have completed an assignment, you will "push" it to Github. When assignments are due, I will "pull" your code down to check it. When you are done with class, you will have a public repository of code you can point to as evidence of your skills in both coding and source code management. So, the first step is that you create a Github account. Go do that.

Git commands

A binary called "git" should be installed on any HPC or Unix system you have. Git takes as its first argument a command. To see a list, enter `git` or `git help`. To see more information on a particular command such as "add," use `git help add`.

Create a repo

Using the Github web interface, create a new repository called "abe487." It's best to have the repo initialized with a "README." Copy the "Clone or Download" link and then on your machine (laptop/HPC), clone your repository and the "metagenomics-book" repo:

```
git clone <your repo>
git@github.com:kyclark/metagenomics-book.git
cp -r metagenomics-book/problems <your repo>
```

Notice that there is a subtle difference between these two commands:

1. cp -r src dest
2. cp -r src/ dest

The first one will copy the "src" directory and then its contents to "dest" while the second one will copy *just the contents* of "src" to "dest."

Committing your work

There are three git commands you must use to put your files into Github so that they can be seen by others:

- add
- commit
- push

A basic workflow is:

```
$ echo hello > hello.txt
$ git add hello.txt
$ git commit -m 'added hello' !$
$ git push
```

The `-m` argument to `commit` is the commit message. If you don't specify a message, you will be dropped into your $EDITOR to type one.

If you cannot see your work the Github web interface, then I cannot check it out and grade it.

Minimally competent bash scripting

Bash is the worst shell scripting language except for all the others. For many of the analyses you'll write, all you will need is a simple bash script, so let's figure out how to write a decent one. I'll share with you what I've found to be the minimal amount of bash I use.

Statements

All programming language have a grammar where "statements" (like "sentences") are built up from other terms. Some languages like Python and Haskell use whitespace to figure out the end of a "statement," which is usually just the right side of the window. C-like languages such as bash and Perl define the end of a statement with a colon `;`. Bash is interesting because it uses both. If you hit <Enter> or type a newline in your code, Bash will execute that statement. If you want to put several commands on one line, you can separate each with a semicolon. If you want to stretch a command over more than one line, you can use a backslash `\` to continue the line:

```
$ echo Hi
Hi
$ echo Hello
Hello
$ echo Hi; echo Hello
Hi
Hello
$ echo \
> Hi
Hi
```

Comments

Every language has a way to indicate text in the source code that should not be executed by the program. Many Unix/c-style languages use the `#` (hash) sign to indicate that any text to the right should be ignored by the language, but some languages use other characters or character combinations like `//` in Javascript, Java, and Rust. Programmers may use comments to explain what some particularly bit of code is doing, or they may use the characters to temporarily disable some section of code. Here is an example of what you might see:

```
# cf. https://en.wikipedia.org/wiki/Factorial
sub fac(n) {
  # first check terminal condition
  if (n <= 1) {
    return 1
  }
  # no? let's recurse!
  else {
    n * fac(n - 1) # the number times one less the number
  }
}
```

It's worth investing time in an editor that can easily comment/uncomment whole sections of code. For instance, in vim, I have a function that will add or removed the appropriate comment character(s) (depending on the filetype) from the beginning of the selected section. If your editor can't do that (e.g., nano), then I suggest you find something more powerful.

Shebang

Scripting languages (sh, bash, Perl, Python, Ruby, etc.) are generally distinguished by the fact that the "program" is a regular file containing plain text that is interpreted into machine code at the time you run it. Other languages (c, C++, Java, Haskell, Rust) have a separate compilation step to turn their regular text source files into a binary executable. If you view a compiled file with an editor/pager, you'll see a mess that might even lock up your window. (If that happens, refer back to "Make it stop!" to kill it or just close the window and start over.)

So, basically a "script" is a plain text file that is often executable by virtue of having the executable bit(s) turned on (cf. "Permissions"). It does not have to be executable, however. It's acceptable to put some commands in a file and simply tell the appropriate program to interpret the file:

```
$ echo "echo Hello, World" > hello.sh
$ sh hello.sh
Hello, World
```

But it looks cooler to do this:

```
$ chmod +x hello.sh
$ ./hello.sh
Hello, World
```

But what's going on here?

```
$ echo 'print("Hello, World")' > hello.py
$ chmod +x hello.py
$ ./hello.py
./hello.py: line 1: syntax error near unexpected token `"Hello,
World"'
./hello.py: line 1: `print("Hello, World")'
```

We put some Python code in a file and then asked our shell (which is bash) to interpret it. That didn't work. If we ask Python to run it, everything is fine:

```
$ python hello.py
Hello, World
```

So we just need to let the shell know that this is Python code, and that is what the "shebang" (see "Pronunciations") line is for. It looks like a comment, but it's special line that the shell uses to interpret the script. I'll use an editor to add a shebang to the "hello.py" script, then I'll `cat` the file so you can see what it looks like.

```
$ cat hello.py
#!/usr/bin/env python
print("Hello, World")
$ ./hello.py
Hello, World
```

Often the shebang line will indicate the absolute path to a program like "/bin/bash" or "/usr/local/bin/gawk," but here I used an absolute path not to Python but to the "env" program which I then passed "python" as the argument. Why did I do that? To make this script "portable" (for certain values of "portable," cf. "It's easier to port a shell than a shell script." – Larry Wall), I prefer to use the "python" that is found by the environment. There are two major versions of Python (2.x and 3.x), and I can't actually be sure which is the user's default or where they are installed. In fact, just from my laptop to the two HPC systems I use, my "python" can be:

- /Users/kyclark/anaconda3/bin/python
- /work/03137/kyclark/local/bin/python
- /rsgrps/bhurwitz/hurwitzlab/bin/python

Go to your command line and type and type `env python`, and you should be dropped into the Python REPL (Read-Evaluate-Print-Loop):

```
$ env python
Python 2.7.11 (default, Apr 22 2016, 18:12:30)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-16)] on linux2
Type "help", "copyright", "credits" or "license" for more inform
ation.
>>> print("Hi there.")
Hi there.
>>> exit
Use exit() or Ctrl-D (i.e. EOF) to exit
```

Let's Make A Script!

Let's make our script say "Hello" to some people:

```
$ cat -n hello2.sh
     1    #!/bin/bash
     2
     3    NAME="Newman"
     4    echo "Hello," $NAME
     5    NAME="Jerry"
     6    echo "Hello, $NAME"
$ ./hello2.sh
Hello, Newman
Hello, Jerry
```

I've created a variable called `NAME` to hold the string "Newman" and print it. Notice there is no `$` when assigning to the variable, only when you use it. The value of `NAME` can be changed at any time. You can print it out like on line 4 as it's own argument to `echo` or inside of a string like on line 6. Notice that the version on line 4 puts a space between the arguments to `echo`.

Because all the variables from the environment (see `env`) are uppercase (e.g., `$HOME` and `$USER`), I tend to use all-caps myself, but this did lead to a problem once when I named a variable `PATH` and then overwrote the actual `PATH` and then my program stopped working entirely as it could no longer find any of the programs it needed. Just remember that everything in Unix is case-sensitive, so `$Name` is an entirely different variable from `$name`.

When assigning a variable, you can have NO SPACES around the `=` sign:

```
$ NAME1="Doge"
$ echo "Such $NAME1"
Such Doge
$ NAME2 = "Doge"
-bash: NAME2: command not found
$ echo "Such $NAME2"
Such
```

Getting Data Into Your Program: Arguments

We would like to get the NAME from the user rather than having it hardcoded in the script. I'll show you three ways our script can take in data from outside:

1. Command-line arguments, both positional (i.e., the first one, the second one, etc.) or named (e.g., `-n NAME`)
2. The environment
3. Reading a configuration file

First we'll cover the command-line arguments which are available through a few variables:

- `$#` : The number (think "#" == number) of arguments
- `$@` : All the arguments in a single string
- `$0` : The name of the script
- `$1, $2` : The first argument, the second argument, etc.

A la:

```
$ cat -n args.sh
     1    #!/bin/bash
     2
     3    echo "Num of args    : \"$#\""
     4    echo "String of args : \"$@\""
     5    echo "Name of program: \"$0\""
     6    echo "First arg      : \"$1\""
     7    echo "Second arg     : \"$2\""
$ ./args.sh
Num of args    : "0"
String of args : ""
Name of program: "./args.sh"
First arg      : ""
Second arg     : ""
$ ./args.sh foo
Num of args    : "1"
String of args : "foo"
Name of program: "./args.sh"
First arg      : "foo"
Second arg     : ""
$ ./args.sh foo bar
Num of args    : "2"
String of args : "foo bar"
Name of program: "./args.sh"
First arg      : "foo"
Second arg     : "bar"
```

If you would like to iterate over all the arguments, you can use `$@` like so:

```
$ cat -n args2.sh
     1    #!/bin/bash
     2
     3    if [[ $# -lt 1 ]]; then
     4        echo "There are no arguments"
     5    else
     6        i=0
     7        for ARG in "$@"; do
     8            let i++
     9            echo "$i: $ARG"
    10        done
    11    fi
$ ./args2.sh
There are no arguments
$ ./args2.sh foo
1: foo
$ ./args2.sh foo bar "baz quux"
1: foo
2: bar
3: baz quux
```

Here I'm throwing in a conditional at line 3 to check if the script has any arguments. If the number of arguments (`$#`) is less than (`-lt`) 1, then let the user know there is nothing to show; otherwise (`else`) do the next block of code. The `for` loop on line 7 works by splitting the argument string (`$@`) on spaces just like the command line does. Both `for` and `while` loops require the `do/done` pair to delineate the block of code (some languages use `{}` , Haskell and Python use only indentation). Along those lines, line 11 is the close of the `if` -- "if" spell backwards; the close of a `case` statement in bash is `esac` .

The other bit of magic I threw in was a counter variable (which I always use lowercase `i` ["integer"], `j` if I needed an inner-counter and so on) which is initialized to "0" on line 6. I increment it, I could have written `$i=$(($i + 1))` , but it's easier to use the `let i++` shorthand. Lastly, notice that "baz quux" seen as a single argument because it was placed in quotes; otherwise arguments are separated by spaces.

Sidebar: Make It Pretty (or else)

Note that indentation doesn't matter as the program below works, but, honestly, which one is easier for you to read?

```
$ cat -n args3.sh
     1    #!/bin/bash
     2
     3    if [[ $# -lt 1 ]]; then
     4    echo "There are no arguments"
     5    else
     6    i=0
     7    for ARG in "$@"; do
     8    let i++
     9    echo "$i: $ARG"
    10    done
    11    fi
$ ./args3.sh foo bar
1: foo
2: bar
```

Sidebar: Catching Common Errors (set -u)

bash is a notoriously easy language to write incorrectly. One step you can take to ensure you don't misspell variables is to add `set -u` at the top of your script. E.g., if you type `echo $HOEM` on the command line, you'll get no output or warning that you misspelled the `$HOME` variable unless you `set -u` :

```
$ echo $HOEM

$ set -u
$ echo $HOEM
-bash: HOEM: unbound variable
```

This command tells bash to complain when you use a variable that was never initialized to some value. This is like putting on your helmet. It's not a requirement (depending on which state you live in), but you absolutely should do this because

there might come a day when you misspell a variable. Note that this will not save you from as error like this:

```
$ cat -n set-u-bug1.sh
     1    #!/bin/bash
     2
     3    set -u
     4
     5    if [[ $# -gt 0 ]]; then
     6      echo $THIS_IS_A_BUG; # never initialized
     7    fi
     8
     9    echo "OK";
$ ./set-u-bug1.sh
OK
$ ./set-u-bug1.sh foo
./set-u-bug1.sh: line 6: THIS_IS_A_BUG: unbound variable
```

You can see that the first execution of the script ran just fine. There is a bug on line 6, but bash didn't catch it because that line did not execute. On the second run, the error occurred, and the script blew up. (FWIW, this is a problem in Python, too.)

Here's another pernicious error:

```
$ cat -n set-u-bug2.sh
     1    #!/bin/bash
     2
     3    set -u
     4
     5    GREETING="Hi"
     6    if [[ $# -gt 0 ]]; then
     7      GRETING=$1 # misspelled
     8    fi
     9
    10    echo $GREETING
$ ./set-u-bug2.sh
Hi
$ ./set-u-bug2.sh Hello
Hi
```

We were foolishly hoping that `set -u` would prevent us from misspelling the `$GREETING`, but at line 7 we simple created a new variable called `$GRETING`. Perhaps you were hoping for more help from your language? This is why we try to limit how much bash we write.

NB: I highly recommend you use the program `shellcheck` (https://www.shellcheck.net/\ to find errors in your bash code.

Our First Argument

AT LAST, let's return to our "hello" script!

```
$ cat -n hello3.sh
     1    #!/bin/bash
     2
     3    echo "Hello, $1!"
$ ./hello3.sh Captain
Hello, Captain!
```

This should make perfect sense now. We are simply saying "hello" to the first argument, but what happens if we provide no arguments?

```
$ ./hello3.sh
Hello, !
```

Checking the Number of Arguments

Well, that looks bad. We should check that the script has the proper number of arguments which is 1:

```
$ cat -n hello4.sh
     1    #!/bin/bash
     2
     3    if [[ $# -ne 1 ]]; then
     4        printf "Usage: %s NAME\n" "$(basename "$0")"
     5        exit 1
     6    fi
     7
     8    echo "Hello, $1!"
$ ./hello4.sh
Usage: hello4.sh NAME
$ ./hello4.sh Captain
Hello, Captain!
$ ./hello4.sh Captain Picard
Usage: hello4.sh NAME
```

Line 3 checks if the number of arguments is not equal (`-ne`) to 1 and prints a help message to indicate proper "usage." Importantly, it also will `exit` the program with a value which is not zero to indicate that there was an error. (NB: An exit value of "0" indicates 0 errors.) Line 4 uses `printf` rather than `echo` so I can do some fancy substitution so that the results of calling the `basename` function on the `$0` (name of the program) is inserted at the location of the `%s` (a string value, cf. man pages for "printf" and "basename").

Here is an alternate way to write this script:

```
$ cat -n hello5.sh
     1    #!/bin/bash
     2
     3    if [[ $# -eq 1 ]]; then
     4        NAME=$1
     5        echo "Hello, $NAME!"
     6    else
     7        printf "Usage: %s NAME\n" "$(basename "$0")"
     8        exit 1
     9    fi
```

Here I check on line 3 if there is just one argument, and the `else` is devoted to handling the error; however, I prefer to check for all possible errors at the beginning and `exit` the program quickly. This also has the effect of keeping my code as far left on the page as possible.

Sidebar: Saving Function Results

In the previous script, you may have noticed `$(basename "$0")`. I was passing the script name (`$0`) to the function `basename` and then passing that to the `printf` function. To call a function in bash and save the results into a variable or use the results as an argument, we can use either backticks (``) (under the `~` on a US keyboard) or `$()`. I find backticks to be too similar to single quotes, so I prefer the latter. To demonstrate:

```
$ ls | head
args.sh*
args2.sh*
args3.sh*
basic.sh*
hello.sh*
hello2.sh*
hello3.sh*
hello4.sh*
hello5.sh*
hello6.sh*
$ FILES=`ls | head`
$ echo $FILES
args.sh args2.sh args3.sh basic.sh hello.sh hello2.sh hello3.sh
hello4.sh hello5.sh hello6.sh
```

Here is a script that shows:

1. Calling `basename` and having the result print out (line 5)
2. Using `$()` to capture the results of `basename` into a variable (line 8)
3. Using `$()` to call `basename` as the second argument to `echo`
4. Showing that `$()` can be interpolated **inside a string**
5. Using `$()` to call `basename` as an argument to `printf`

```
$ cat -n functions.sh
     1    #!/bin/bash
     2
     3    # call function
     4    echo -n "1: BASENAME: "
     5    basename "$0"
     6
     7    # put function results into variable
     8    BASENAME=$(basename "$0")
     9    echo "2: BASENAME: $BASENAME"
    10
    11    # use results of function as argument to another funct
ion
    12    echo "3: BASENAME:" "$(basename "$0")"
    13    echo "4: BASENAME: $(basename "$0")"
    14    printf "5: BASENAME: %s\n" "$(basename "$0")"
$ ./functions.sh
1: BASENAME: functions.sh
2: BASENAME: functions.sh
3: BASENAME: functions.sh
4: BASENAME: functions.sh
5: BASENAME: functions.sh
```

Providing Default Argument Values

Here is how you can provide a default value for an argument with `:-`:

```
$ cat -n hello6.sh
     1    #!/bin/bash
     2
     3    echo "Hello, ${1:-Stranger}!"
$ ./hello6.sh
Hello, Stranger!
$ ./hello6.sh Govnuh
Hello, Govnuh!
```

Arguments From The Environment

You can also use look in the environment for argument values. For instance, we could accept the `NAME` as either the first argument to the script (`$1`) or the `$USER` from the environment:

```
$ cat -n hello7.sh
     1    #!/bin/bash
     2
     3    NAME=${1:-$USER}
     4    [[ -z "$NAME" ]] && NAME='Stranger'
     5    echo "Hello, $NAME
$ ./hello7.sh
Hello, kyclark
$ ./hello7.sh Barbara
Hello, Barbara
```

What's interesting is that you can temporarily over-ride an environmental variable like so:

```
$ USER=Bart ./hello7.sh
Hello, Bart
$ ./hello7.sh
Hello, kyclark
```

Exporting Values to the Environment

Notice that I can set `USER` for the first run to "Bart," but the value returns to "kyclark" on the next run. I can permanently set a value in the environment by using the `export` command. Here is a version of the script that looks for an environmental variable called `WHOM` (please do override your `$USER` name in the environment as things will break):

```
$ cat -n hello8.sh
     1    #!/bin/bash
     2
     3    echo "Hello, ${WHOM:-Marie}"
$ ./hello8.sh
Hello, Marie
```

As before I can set it temporarily:

```
$ WHOM=Doris ./hello8.sh
Hello, Doris
$ ./hello8.sh
Hello, Marie
```

Now I will `export WHOM` so that it persists:

```
$ WHOM=Doris
$ export WHOM
$ ./hello8.sh
Hello, Doris
$ ./hello8.sh
Hello, Doris
```

To remove `WHOM` from the environment, use `unset` :

```
$ unset WHOM
$ ./hello8.sh
Hello, Marie
```

Some programs rely heavily on environmental variables (e.g., Centrifuge, TACC LAUNCHER) for arguments. Here is a short script to illustrate how you would use such a program:

```
$ cat -n hello9.sh
     1    #!/bin/bash
     2
     3    WHOM="Who's on first" ./hello8.sh
     4    WHOM="What's on second"
     5    export WHOM
     6    ./hello8.sh
     7    WHOM="I don't know's on third" ./hello8.sh
$ ./hello9.sh
Hello, Who's on first
Hello, What's on second
Hello, I don't know's on third
```

Required and Optional Arguments

Now we're going to accept two arguments, "GREETING" and "NAME" while providing defaults for both:

```
$ cat -n positional.sh
     1    #!/bin/bash
     2
     3    set -u
     4
     5    GREETING=${1:-Hello}
     6    NAME=${2:-Stranger}
     7
     8    echo "$GREETING, $NAME"
$ ./positional.sh
Hello, Stranger
$ ./positional.sh Howdy
Howdy, Stranger
$ ./positional.sh Howdy Padnuh
Howdy, Padnuh
$ ./positional.sh "" Pahnuh
Hello, Pahnuh
```

You notice that if I want to use the default argument for the greeting, I have to pass an empty string `""`.

What if I want to require at least one argument?

```
$ cat -n positional2.sh
     1    #!/bin/bash
     2
     3    set -u
     4
     5    if [[ $# -lt 1 ]]; then
     6        printf "Usage: %s GREETING [NAME]\n" "$(basename "
$0")"
     7        exit 1
     8    fi
     9
    10    GREETING=$1
    11    NAME=${2:-Stranger}
    12
    13    echo "$GREETING, $NAME"
$ ./positional2.sh "Good Day"
Good Day, Stranger
$ ./positional2.sh "Good Day" "Kind Sir"
Good Day, Kind Sir
```

It's also important to note the subtle hints given to the user in the "Usage" statement. `[NAME]` has square brackets to indicate that it is an option, but `GREETING` does not to say it is required. As noted before I wanted to use the GREETING "Good Day," so I had to put it in quotes so that the shell would not interpret them as two arguments. Same with the NAME "Kind Sir."

```
$ ./positional2.sh Good Day Kind Sir
Good, Day
```

Not Too Few, Not Too Many (Goldilocks)

Hmm, maybe we should detect that the script had too many arguments?

```
$ cat -n positional3.sh
     1    #!/bin/bash
     2
     3    set -u
     4
     5    if [[ $# -lt 1 ]] || [[ $# -gt 2 ]]; then
     6        printf "Usage: %s GREETING [NAME]\n" "$(basename "
$0")"
     7        exit 1
     8    fi
     9
    10    GREETING=$1
    11    NAME=${2:-Stranger}
    12
    13    printf "%s, %s\n" "$GREETING" "$NAME"
$ ./positional3.sh Good Day Kind Sir
Usage: positional3.sh GREETING [NAME]
$ ./positional3.sh "Good Day" "Kind Sir"
Good Day, Kind Sir
```

To check for too many arguments, I added an "OR" (the double pipes `||`) and another conditional ("AND" is `&&`). I also changed line 13 to use a `printf` command to highlight the importance of quoting the arguments *inside the script* so that bash won't get confused. Try it without those quotes and try to figure out why it's doing what it's doing. I highly recommend using the program "shellcheck" (https://github.com/koalaman/shellcheck) to find mistakes like this. Also, consider using more powerful/helpful/sane languages -- but that's for another discussion.

Named Arguments To The Rescue

I hope maybe by this point you're thinking that the script is getting awfully complicated just to allow for a combination of required an optional arguments all given in a particular order. You can manage with 1-3 positional arguments, but, after that, we really need to have named arguments and/or flags to indicate how we want to run the program. A named argument might be `-f mouse.fa` to indicate the value for the `-f` ("file," probably) argument is "mouse.fa," whereas a flag like `-v` might be a yes/no ("Boolean," if you like) indicator that we do or do

not want "verbose" mode. You've encountered these with programs like `ls -l` to indicate you want the "long" directory listing or `ps -u $USER` to indicate the value for `-u` is the `$USER`.

The best thing about named arguments is that they can be provided in any order:

```
$ ./named.sh -n Patch -g "Good Boy"
Good Boy, Patch!
```

Some may have values, some may be flags, and you can easily provide good defaults to make it easy for the user to provide the bare minimum information to run your program. Here is a version that has named arguments:

```
$ cat -n named.sh
     1    #!/bin/bash
     2
     3    set -u
     4
     5    GREETING=""
     6    NAME="Stranger"
     7    EXCITED=0
     8
     9    function USAGE() {
    10        printf "Usage:\n  %s -g GREETING [-e] [-n NAME]\n\
n" $(basename $0)
    11        echo "Required arguments:"
    12        echo " -g GREETING"
    13        echo
    14        echo "Options:"
    15        echo " -n NAME ($NAME)"
    16        echo " -e Print exclamation mark (default yes)"
    17        echo
    18        exit ${1:-0}
    19    }
    20
    21    [[ $# -eq 0 ]] && USAGE 1
    22
    23    while getopts :g:n:eh OPT; do
    24      case $OPT in
```

```
    25        h)
    26          USAGE
    27          ;;
    28        e)
    29          EXCITED=1
    30          ;;
    31        g)
    32          GREETING="$OPTARG"
    33          ;;
    34        n)
    35          NAME="$OPTARG"
    36          ;;
    37        :)
    38          echo "Error: Option -$OPTARG requires an argumen
t."
    39          exit 1
    40          ;;
    41        \?)
    42          echo "Error: Invalid option: -${OPTARG:-""}"
    43          exit 1
    44      esac
    45    done
    46
    47    [[ -z "$GREETING" ]] && USAGE 1
    48    PUNCTUATION="."
    49    [[ $EXCITED -ne 0 ]] && PUNCTUATION="!"
    50
    51    echo "$GREETING, $NAME$PUNCTUATION"
```

When run without arguments or with the `-h` flag, it produces a help message.

```
$ ./named.sh
Usage:
  named.sh -g GREETING [-e] [-n NAME]

Required arguments:
 -g GREETING

Options:
 -n NAME (Stranger)
 -e Print exclamation mark (default yes)
```

Our script just got much longer but also more flexible. I've written a hundred shell scripts with just this as the template, so you can, too. Go search for how `getopt` works and copy-paste this for your bash scripts, but the important thing to understand about `getopt` is that flags that take arguments have a `:` after them (`g:` == "-g something") and ones that do not, well, do not (`h` == "-h" == "please show me the help page). Both the "h" and "e" arguments are flags:

```
$ ./named.sh -n Patch -g "Good Boy"
Good Boy, Patch.
$ ./named.sh -n Patch -g "Good Boy" -e
Good Boy, Patch!
```

I've introduced a new function called `USAGE` that prints out the "Usage" statement so that it can be called when:

- the script is run with no arguments (line 21)
- the script is run with the "-h" flag (lines 25-26)
- the script is run with bad input (line 47)

I initialized the NAME to "Stranger" (line 6) and then let the user know in the "Usage" what the default value will be. When checking the GREETING in line 44, I'm actually checking that the length of the value is greater than zero because it's possible to run the script like this:

```
$ ./named01.sh -g ""
```

Which would technically pass muster but does not actually meet our requirements.

Reading a Configuration File

The last way I'll show you to get data into your program is to read a configuration file. This builds on the earlier example of using `export` to put values into the environment:

```
$ cat -n config1.sh
     1    export NAME="Merry Boy"
     2    export GREETING="Good morning"
$ cat -n read-config.sh
     1    #!/bin/bash
     2
     3    source config1.sh
     4    echo "$GREETING, $NAME!"
$ ./read-config.sh
Good morning, Merry Boy!
```

To make this more flexible, let's pass the config file as an argument:

```
$ cat -n read-config2.sh
     1    #!/bin/bash
     2
     3    CONFIG=${1:-config1.sh}
     4    if [[ ! -f "$CONFIG" ]]; then
     5        echo "Bad config \"$CONFIG\""
     6        exit 1
     7    fi
     8
     9    source $CONFIG
    10    echo "$GREETING, $NAME!"
$ ./read-config2.sh
Good morning, Merry Boy!
$ cat -n config2.sh
     1    export NAME="François"
     2    export GREETING="Salut"
$ ./read-config2.sh config2.sh
Salut, François!
$ ./read-config2.sh foo
Bad config "foo"
```

For Loops

Often we want to do some set of actions for all the files in a directory or all the identifiers in a file. You can use a `for` loop to iterate over the values in some command that returns a list of results:

```
$ for FILE in *.sh; do echo "FILE = $FILE"; done
FILE = args.sh
FILE = args2.sh
FILE = args3.sh
FILE = basic.sh
FILE = hello.sh
FILE = hello2.sh
FILE = hello3.sh
FILE = hello4.sh
FILE = hello5.sh
FILE = hello6.sh
FILE = named.sh
FILE = positional.sh
FILE = positional2.sh
FILE = positional3.sh
FILE = set-u-bug1.sh
FILE = set-u-bug2.sh
```

Here it is in a script:

```
$ cat -n for.sh
     1    #!/bin/bash
     2
     3    set -u
     4
     5    DIR=${1:-$PWD}
     6
     7    if [[ ! -d "$DIR" ]]; then
     8        echo "$DIR is not a directory"
     9        exit 1
    10    fi
    11
    12    i=0
    13    for FILE in $DIR/*; do
    14        let i++
    15        printf "%3d: %s\n" $i "$FILE"
    16    done
```

On line 5, I default `DIR` to the current working directory which I can find with the environmental variable `$PWD` (print working directory). I check on line 7 that the argument is actually a directory with the `-d` test (`man test`). The rest should look familiar. Here it is in action:

```
$ ./for.sh | head
  1: /Users/kyclark/work/metagenomics-book/bash/args.sh
  2: /Users/kyclark/work/metagenomics-book/bash/args2.sh
  3: /Users/kyclark/work/metagenomics-book/bash/args3.sh
  4: /Users/kyclark/work/metagenomics-book/bash/basic.sh
  5: /Users/kyclark/work/metagenomics-book/bash/config1.sh
  6: /Users/kyclark/work/metagenomics-book/bash/config2.sh
  7: /Users/kyclark/work/metagenomics-book/bash/count-fa.sh
  8: /Users/kyclark/work/metagenomics-book/bash/for-read-file.sh
  9: /Users/kyclark/work/metagenomics-book/bash/for.sh
 10: /Users/kyclark/work/metagenomics-book/bash/functions.sh
$ ./for.sh ../problems | head
  1: ../problems/cat-n
  2: ../problems/common-words
  3: ../problems/dna
  4: ../problems/gapminder
  5: ../problems/gc
  6: ../problems/greeting
  7: ../problems/hamming
  8: ../problems/hello
  9: ../problems/proteins
 10: ../problems/tac
```

You will see many examples of using `for` to read from a file like so:

```
$ cat -n for-read-file.sh
     1    #!/bin/bash
     2
     3    FILE=${1:-'srr.txt'}
     4    for LINE in $(cat "$FILE"); do
     5        echo "LINE \"$LINE\""
     6    done
$ cat srr.txt
SRR3115965
SRR516222
SRR919365
$ ./for-read-file.sh srr.txt
LINE "SRR3115965"
LINE "SRR516222"
LINE "SRR919365"
```

But that can break badly when the file contains more than one "word" per line (as defined by the `$IFS` [input field separator]):

```
$ column -t pov-meta.tab
name              lat_lon.ll
GD.Spr.C.8m.fa    -17.92522,146.14295
GF.Spr.C.9m.fa    -16.9207,145.9965833
L.Spr.C.1000m.fa  48.6495,-126.66434
L.Spr.C.10m.fa    48.6495,-126.66434
L.Spr.C.1300m.fa  48.6495,-126.66434
L.Spr.C.500m.fa   48.6495,-126.66434
L.Spr.I.1000m.fa  48.96917,-130.67033
L.Spr.I.10m.fa    48.96917,-130.67033
L.Spr.I.2000m.fa  48.96917,-130.67033
$ ./for-read-file.sh pov-meta.tab
LINE "name"
LINE "lat_lon.ll"
LINE "GD.Spr.C.8m.fa"
LINE "-17.92522,146.14295"
LINE "GF.Spr.C.9m.fa"
LINE "-16.9207,145.9965833"
LINE "L.Spr.C.1000m.fa"
LINE "48.6495,-126.66434"
LINE "L.Spr.C.10m.fa"
LINE "48.6495,-126.66434"
LINE "L.Spr.C.1300m.fa"
LINE "48.6495,-126.66434"
LINE "L.Spr.C.500m.fa"
LINE "48.6495,-126.66434"
LINE "L.Spr.I.1000m.fa"
LINE "48.96917,-130.67033"
LINE "L.Spr.I.10m.fa"
LINE "48.96917,-130.67033"
LINE "L.Spr.I.2000m.fa"
LINE "48.96917,-130.67033"
```

While Loops

The proper way to read a file line-by-line is with `while`:

```
$ cat -n while.sh
     1    #!/bin/bash
     2
     3    FILE=${1:-'srr.txt'}
     4    while read -r LINE; do
     5        echo "LINE \"$LINE\""
     6    done < "$FILE"
$ ./while.sh srr.txt
LINE "SRR3115965"
LINE "SRR516222"
LINE "SRR919365"
$ ./while.sh meta.tab
LINE "GD.Spr.C.8m.fa    -17.92522,146.14295"
LINE "GF.Spr.C.9m.fa    -16.9207,145.9965833"
LINE "L.Spr.C.1000m.fa    48.6495,-126.66434"
LINE "L.Spr.C.10m.fa    48.6495,-126.66434"
LINE "L.Spr.C.1300m.fa    48.6495,-126.66434"
LINE "L.Spr.C.500m.fa    48.6495,-126.66434"
LINE "L.Spr.I.1000m.fa    48.96917,-130.67033"
LINE "L.Spr.I.10m.fa    48.96917,-130.67033"
LINE "L.Spr.I.2000m.fa    48.96917,-130.67033"
```

Another advantage is that `while` can break the line into fields:

```
$ cat -n while2.sh
     1    #!/bin/bash
     2
     3    FILE='meta.tab'
     4    while read -r SITE LOC; do
     5        echo "$SITE is located at \"$LOC\""
     6    done < "$FILE"
$ ./while2.sh
GD.Spr.C.8m.fa is located at "-17.92522,146.14295"
GF.Spr.C.9m.fa is located at "-16.9207,145.9965833"
L.Spr.C.1000m.fa is located at "48.6495,-126.66434"
L.Spr.C.10m.fa is located at "48.6495,-126.66434"
L.Spr.C.1300m.fa is located at "48.6495,-126.66434"
L.Spr.C.500m.fa is located at "48.6495,-126.66434"
L.Spr.I.1000m.fa is located at "48.96917,-130.67033"
L.Spr.I.10m.fa is located at "48.96917,-130.67033"
L.Spr.I.2000m.fa is located at "48.96917,-130.67033"
```

Sidebar: Saving Function Results in Files

Often I want to iterate over the results of some calculation. Here is an example of saving the results of an operation (`find`) into a temporary file:

```
$ cat -n count-fa.sh
     1    #!/bin/bash
     2
     3    set -u
     4
     5    if [[ $# -ne 1 ]]; then
     6        printf "Usage: %s DIR\n" "$(basename "$0")"
     7        exit 1
     8    fi
     9
    10    DIR=$1
    11    TMP=$(mktemp)
    12    find "$DIR" -type f -name \*.fa > "$TMP"
    13    NUM_FILES=$(wc -l "$TMP" | awk '{print $1}')
    14
    15    if [[ $NUM_FILES -lt 1 ]]; then
    16        echo "Found no .fa files in $DIR"
    17        exit 1
    18    fi
    19
    20    NUM_SEQS=0
    21    while read -r FILE; do
    22        NUM_SEQ=$(grep -c '^>' "$FILE")
    23        NUM_SEQS=$((NUM_SEQS + NUM_SEQ))
    24        printf "%10d %s\n" "$NUM_SEQ" "$(basename "$FILE")
"
    25    done < "$TMP"
    26
    27    rm "$TMP"
    28
    29    echo "Done, found $NUM_SEQS sequences in $NUM_FILES fi
les."
$ ./count-fa.sh ../problems
        23 anthrax.fa
         9 burk.fa
Done, found 32 sequences in 2 files.
```

Line 11 uses the `mktemp` function to give us the name of a temporary file, then I `find` all the files ending in ".fa" or ".fasta" and put that into the temporary file. I could them to make sure I found something. Then I read from the tempfile and use the `FILE` name to count the number of times I see a greater-than sign at the beginning of a line.

A Full Bag of Tricks

Lastly I'm going to show you how to create some sane defaults, make missing directories, find user input, transform that input, and report back to the user. Here's a script that takes an IN_DIR, counts the lines of all the files therein, and reports said line counts into an optional OUT_DIR.

```
 1  #!/bin/bash
 2
 3  set -u
 4
 5  IN_DIR=""
 6  OUT_DIR="$PWD/$(basename "$0" '.sh')-out"
 7
 8  function lc() {
 9      wc -l "$1" | awk '{print $1}'
10  }
11
12  function USAGE() {
13      printf "Usage:\n  %s -i IN_DIR -o OUT_DIR\n\n" "$(
basename "$0")"
14
15      echo "Required arguments:"
16      echo " -i IN_DIR"
17      echo "Options:"
18      echo " -o OUT_DIR"
19      echo
20      exit "${1:-0}"
21  }
22
23  [[ $# -eq 0 ]] && USAGE 1
24
```

```
    25    while getopts :i:o:h OPT; do
    26        case $OPT in
    27            h)
    28                USAGE
    29                ;;
    30            i)
    31                IN_DIR="$OPTARG"
    32                ;;
    33            o)
    34                OUT_DIR="$OPTARG"
    35                ;;
    36            :)
    37                echo "Error: Option -$OPTARG requires an a
rgument."
    38                exit 1
    39                ;;
    40            \?)
    41                echo "Error: Invalid option: -${OPTARG:-""
}"
    42                exit 1
    43        esac
    44    done
    45
    46    if [[ -z "$IN_DIR" ]]; then
    47        echo "IN_DIR is required"
    48        exit 1
    49    fi
    50
    51    if [[ ! -d "$IN_DIR" ]]; then
    52        echo "IN_DIR \"$IN_DIR\" is not a directory."
    53        exit 1
    54    fi
    55
    56    echo "Started $(date)"
    57
    58    FILES_LIST=$(mktemp)
    59    find "$IN_DIR" -type f -name \*.sh > "$FILES_LIST"
    60    NUM_FILES=$(lc "$FILES_LIST")
    61
    62    if [[ $NUM_FILES -gt 0 ]]; then
```

```
63        echo "Will process NUM_FILES \"$NUM_FILES\""
64
65        [[ ! -d $OUT_DIR ]] && mkdir -p "$OUT_DIR"
66
67        i=0
68        while read -r FILE; do
69            BASENAME=$(basename "$FILE")
70            let i++
71            printf "%3d: %s\n" $i "$BASENAME"
72            wc -l "$FILE" > "$OUT_DIR/$BASENAME"
73        done < "$FILES_LIST"
74
75        rm "$FILES_LIST"
76        echo "See results in OUT_DIR \"$OUT_DIR\""
77    else
78        echo "No files found in \"$IN_DIR\""
79    fi
80
81    echo "Finished $(date)"
```

The IN_DIR argument is required (lines 46-49), and it must be a directory (lines 51-54). If the user does not supply an OUT_DIR, I will create a reasonable default using the current working directory and the name of the script plus "-out" (line 6). One thing I love about bash is that I can call functions inside of strings, so OUT_DIR is a string (it's in double quotes) of the variable $PWD, the character "/", and the result of the function call to `basename` where I'm giving the optional second argument ".sh" that I want removed from the first argument, and then the string "-out".

At line 58, I create a temporary file to hold the names of the files I need to process. A line 59, I look for the files in IN_DIR that need to be processed. You can read the manpage for `find` and think about what your script might need to find (".fa" files greater than 0 bytes in size last modified since some date, etc.). At line 60, I call my `lc` (line count) function to see how many files I found. If I found more than 0 files (line 62), then I move ahead with processing. I check to see if the OUT_DIR needs to be created (line 65), and then create a counter variable ("i") that I'll use to number the files as I process them. At line 68, I start a `while` loop to iterate over the input from redirecting *in* from the temporary file holding the

file names (line 73, `< "$FILES_LIST"`). Then a `printf` to let the user know which file we're processing, then a simple command (`wc`) but where you might choose to BLAST the sequence file to a database of pathogens to determine how deadly the sample is. When I'm done, I clean up the temp file (line 75).

The alternate path when I find no input files (line 77-79) is to report that fact. Bracketing the main processing logic are "Started/Finished" statements so I can see how long my script took. When you start your coding career, you will usually sit and watch your code run before you, but eventually you'll submit the your jobs to an HPC queue where they will be run for you on a separate machine when the resources become available.

The above is, I would say, a minimally competent bash script. If you can understand everything in there, then you know enough to be dangerous and should move on to learning more powerful languages -- like Python!

Exercises

Write a script that mimics "cat -n" for a given file.

IRC

Internet Relay Chat (IRC) is where you can find "chat rooms" to ask for help with various questions. I'll show you how to join the #perl6 (http://perl6.org/community/irc) chat room and a bit of etiquette for asking questions.

First off, you need to find an IRC client. On Unix platforms, you might like to use a command-line utility called "ircii." On the Mac, I prefer to use Colloquy. Regardless, you need to connect to the server "irc.freenode.net" and then the chat room called "#perl6."

Once you're in the room, just post your question and patiently wait for an answer. Be very precise about what's not working. It's probably best to go to a site like "pastie.org" or "lpaste.net" and paste a *minimum test case* for what's troubling you. Do not paste hundreds of lines of code. Write a small script that illustrates just one problem you are having. (You actually may find that this exercise helps you find your own errors.)

Inside the chat room, you can actually paste bits of code. Put "m:" at the start of your line, and whatever follows will be executed by Perl 6.

Organizing with Makefiles

GNU "make" (https://www.gnu.org/software/make/) will help you organize your code and testing. You only need to learn a few things about make to be productive.

First, you create text file called "Makefile" (or "makefile"). Whenever you run something on the command line that you might want to run again, you should probably put it into your Makefile so you don't have to go searching through your history to find the magic incantation of some command that actually did what you wanted. Don't be like me and think "It's totally OK to leave my sunglasses in the driver's seat because I definitely will not forget they are there and then sit on them when I come back to my car." Don't trust your memory, just add it to you Makefile.

If you check out "github.com:kyclark/metagenomics-book.git", you will find a "problems/dna" directory with a finished version of the "dna.pl6" program along with sample data, a test script, and a Makefile that looks like this

```
$ cat -n Makefile
     1    .PHONY: run file test
     2
     3    run:
     4        ./dna.py AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGT
GGATTAAAAAAAGAGTGTCTGATAGCAGC
     5
     6    file:
     7        ./dna.py test.txt
     8
     9    test:
    10        python3 -m pytest -v test.py
```

If you type `make`, you should see this:

```
$ make
./dna.py AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAAAAGA
GTGTCTGATAGCAGC
20 12 17 21
```

Without any arguments, `make` will execute the first "target" which is basically a word-ish thing followed by a colon, which in this case is "run." If you `make run`, you will see the same thing. The `make file` target executes the script with a file as the input rather than a string, and `make test` will run the "test.pl6" script:

```
$ make test
python3 -m pytest -v test.py
============================= test session starts ==============
================
platform darwin -- Python 3.5.3, pytest-3.0.7, py-1.4.33, pluggy
-0.4.0 -- /Users/kyclark/anaconda3/bin/python3
cachedir: .cache
rootdir: /Users/kyclark/work/metagenomics-book/problems/dna, ini
file:
collected 4 items

test.py::test_script_exists PASSED
test.py::test_usage PASSED
test.py::test_arg PASSED
test.py::test_file PASSED

========================== 4 passed in 0.13 seconds ===========
================
```

Find unclustered protein sequences

A labmate of mine wanted help finding the sequences of proteins that failed to cluster. Rather than writing the solution as a shell script, I found myself writing a Makefile as it easily allowed me to re-run certain steps while I worked out the kinks in my logic.

For this exercise, use the UA HPC and do the following setup:

```
$ mkdir ~/work/abe487/unclustered-proteins
$ cd !$
$ wget ftp://ftp.imicrobe.us/abe487/exercises/unclustered-protei
ns.tgz
$ tar xvf unclustered-proteins.tgz
$ cd unclustered-proteins
```

The "README" contains our instructions:

```
$ cat README
# Find unclustered proteins

The file "cdhit60.3+.clstr" contains all of the GI numbers for
proteins that were clustered and put into hmm profiles.  The fil
e
"proteins.fa" contains all proteins (the header is only the GI
number).  Extract the proteins from the "proteins.fa" file that
were
not clustered.
```

If you type "make" in the directory, you will execute a small pipeline to do this job:

```
$ make
find . \( -name unclustered-proteins.fa -o -name clean-proteins.
fa -o -name \*.o \) -exec rm {} \;
grep -ve '^>' cdhit60.3+.clstr | awk '{print $3}' | awk -F"|" '{
print $2}' | sort > clustered-ids.o
sed "s/|.*//" proteins.fa > clean-proteins.fa
grep -e '^>' clean-proteins.fa | sed "s/^>//" | sort > protein-i
ds.o
comm -23 protein-ids.o clustered-ids.o > unclustered-ids.o
seqmagick convert --include-from-file unclustered-ids.o clean-pr
oteins.fa unclustered-proteins.fa
seqmagick info unclustered-proteins.fa
name                    alignment    min_len   max_len   avg_len
  num_seqs
unclustered-proteins.fa FALSE              0      7391    293.74
    204264
```

Let's break this down step-by-step to understand what is happening by looking at the contents of the "Makefile":

```
$ cat -n Makefile
     1     all: clean info
     2
     3     clean:
     4         find . \( -name unclustered-proteins.fa -o -name
clean-proteins.fa -o -name \*.o \) -exec rm {} \;
     5
     6     clustered-ids:
     7         grep -ve '^>' cdhit60.3+.clstr | awk '{print $$3}
' | awk -F"|" '{print $$2}' | sort | uniq > clustered-ids.o
     8
     9     clean-proteins:
    10         sed "s/|.*//" proteins.fa > clean-proteins.fa
    11
    12     protein-ids: clean-proteins
    13         grep -e '^>' clean-proteins.fa | sed "s/^>//" | s
ort > protein-ids.o
    14
    15     unclustered-ids: clustered-ids protein-ids
    16         comm -23 protein-ids.o clustered-ids.o > uncluste
red-ids.o
    17
    18     unclustered-proteins: unclustered-ids
    19         seqmagick convert --include-from-file unclustered
-ids.o clean-proteins.fa unclustered-proteins.fa
    20
    21     info: unclustered-proteins
    22         seqmagick info unclustered-proteins.fa
    23
    24     dist: clean
    25         (cd .. && tar czvf unclustered-proteins.tgz unclu
stered-proteins)
    26
    27     check: unclustered-proteins
    28         ./check.sh
```

The first defined target in a Makefile will be the default if none is supplied to `make`, and it's typical to call this "all." I set this target to simply be a combination of "clean" (to get rid of all the intermediate files -- I have to use this complex

`find` command so as not to encounter a failure if I were to `rm` a file that does not exist) and "info." Notice that most of the targets indicate a dependency, so executing "info" requires that "unclustered-proteins" be run which in turn needs "unclustered-ids" and so on such that the first action that must be run is "clustered-ids."

The first task in "clustered-ids" is to find those protein IDs in the "cdhit60.3+.clstr" file that were clustered by `cd-hit`. Let's look at that file:

```
$ head -5 cdhit60.3+.clstr
>Cluster_5086
0    358aa, >gi|317183610|gb|ADV... at 66.76%
1    361aa, >gi|315661179|gb|ADU... at 70.36%
2    118aa, >gi|375968555|gb|AFB... at 70.34%
3    208aa, >gi|194307477|gb|ACF... at 61.54%
```

The format of the file is similar to a FASTA file where the ">" sign at the left-most column identifies a cluster with the following lines showing the IDs of the sequences in the cluster. To extract just the clustered IDs, we cannot just do `grep '>'` as we'll get both the cluster IDs and the protein IDs.

```
$ grep '>' cdhit60.3+.clstr | head -5
>Cluster_5086
0    358aa, >gi|317183610|gb|ADV... at 66.76%
1    361aa, >gi|315661179|gb|ADU... at 70.36%
2    118aa, >gi|375968555|gb|AFB... at 70.34%
3    208aa, >gi|194307477|gb|ACF... at 61.54%
```

We'll need to use a regular expression (the `-e` for "extended" on most greps, but sometimes not required):

```
$ grep -e '^>' cdhit60.3+.clstr | head -5
>Cluster_5086
>Cluster_10030
>Cluster_8374
>Cluster_13356
>Cluster_7732
```

and then invert that with "-v":

```
$ grep -v '^>' cdhit60.3+.clstr | head -5
0    358aa, >gi|317183610|gb|ADV... at 66.76%
1    361aa, >gi|315661179|gb|ADU... at 70.36%
2    118aa, >gi|375968555|gb|AFB... at 70.34%
3    208aa, >gi|194307477|gb|ACF... at 61.54%
4    358aa, >gi|291292536|gb|ADD... at 68.99%
```

From this output, we'd like to extract the ">gi|317183610|gb|ADV..." bit, which is the third column when split on whitespace. The tool `awk` is perfect for this, and whitespace is the default split point (as opposed to `cut` which uses tabs):

```
$ grep -ve '^>' cdhit60.3+.clstr | awk '{print $3}' | head -5
>gi|317183610|gb|ADV...
>gi|315661179|gb|ADU...
>gi|375968555|gb|AFB...
>gi|194307477|gb|ACF...
>gi|291292536|gb|ADD...
```

If we look at the IDs in the proteins file, we'll see they are integers:

```
$ grep '>' proteins.fa | head -5
>388548806
>388548807
>388548808
>388548809
>388548810
```

So we see that from "gi|317183610|gb|ADV..." we need to extract just the second field when splitting on the vertical bar. Again, `awk` is perfect, but we need to tell it to split on something other than the default by using the "-F" flag:

```
$ grep -ve '^>' cdhit60.3+.clstr | awk '{print $3}' | awk -F'|'
'{print $2}' | head -5
317183610
315661179
375968555
194307477
291292536
```

These are the protein IDs for those that were successfully clustered, so we just need to capture these to a file which we can do with a redirect `>`. Looking ahead, I know that these will need to be sorted for another tool to work, so I'll add that to the pipeline. Also, each protein might be clustered more than once, so I should `uniq` the list, and you'll recall that input must be sorted first. You'll notice that this is now the "clustered-ids" target. Because the dollar sign is used by `make` to indicate variables, we need to escape them with another dollar sign to make them literals:

```
clustered-ids:
           grep -ve '^>' cdhit60.3+.clstr | awk '{print  $$3}' |
 awk -F"|" '{print $$2}' | sort | uniq > clustered-ids.o
```

Now we need to extract the protein IDs which we can do with `grep`, but we also need to remove the ">" sign. Additionally, some of the IDs have characters other than digits that we will need to remove. To demonstrate this, I'll use `grep -P` to indicate a Perl regular expression and combine with "-v" to invert it:

```
$ grep -e '^>' proteins.fa | sed "s/^>//" | grep -v -P '^\d+$' |
 head -5
26788002|emb|CAD19173.1| putative RNA helicase, partial [Agaricu
s bisporus virus X]
26788000|emb|CAD19172.1| putative RNA helicase, partial [Agaricu
s bisporus virus X]
985757046|ref|YP_009222010.1| hypothetical protein [Alternaria b
rassicicola fusarivirus 1]
985757045|ref|YP_009222011.1| hypothetical protein [Alternaria b
rassicicola fusarivirus 1]
985757044|ref|YP_009222009.1| polyprotein [Alternaria brassicico
la fusarivirus 1]
```

Let's break down that regex:

```
^ \d + $
1 2  3 4
```

1. start of the line
2. a digit (0-9)
3. one or more
4. end of the line

Looking at the above output, we can see that it would be pretty easy to get rid of everything starting with the vertical bar. Since that is not a part of the sequence, we should be safe using `sed` to clean up the proteins file:

```
clean-protein-ids:
    sed "s/|.*//" proteins.fa > proteins-clean.fa
```

Also, I'll sort the output for use by a later step and redirect to a file, and this now becomes the "protein-ids" target:

```
protein-ids: clean-proteins
           grep -e '^>' clean-proteins.fa | sed "s/^>//" | sort
> protein-ids.o
```

To find the lines in "protein-ids" that are not in "clustered-ids," I can use the `comm` (common) command. Notice that this target has two dependencies:

```
unclustered-ids: clustered-ids protein-ids
	comm -23 protein-ids.o clustered-ids.o > unclustered-
ids.o
```

To extract the actual sequences from the "proteins.fa" file using the IDs in "unclustered-ids," `seqmagick convert` program is a great choice:

```
unclustered-proteins: unclustered-ids
	seqmagick convert --include-from-file unclustered-ids
.o clean-proteins.fa unclustered-proteins.fa
```

Lastly, we'd like to see confirmation that we got a reasonable output, so we can examine the resulting FASTA file with `seqmagick info`:

```
info: unclustered-proteins
    seqmagick info unclustered-proteins.fa
```

In the end, we need to verify that we have a reasonable answer. Let's add the number of clustered and unclustered protein IDs and see if they total the original number of proteins:

```
$ seqmagick info unclustered-proteins.fa
name                    alignment    min_len   max_len   avg_len
  num_seqs
unclustered-proteins.fa FALSE              0      7391    293.74
    204264
```

So we found 204,264 unclustered proteins sequences. Is that the right number?

```
$ wc -l clustered-ids.o
16257 clustered-ids.o
$ wc -l unclustered-ids.o
204263 unclustered-ids.o
$ bc <<< 16257+204263
220520
$ seqmagick info proteins.fa
name        alignment    min_len   max_len   avg_len  num_seqs
proteins.fa FALSE              0      7391    298.18    220520
```

I can put that into a shell script:

```
$ cat -n check.sh
     1    #!/bin/bash
     2
     3    set -u
     4
     5    function lc() {
     6      wc -l $1 | cut -d ' ' -f 1
     7    }
     8
     9    echo $(lc clustered-ids.o)   > count-clustered.o
    10    echo $(lc unclustered-ids.o) > count-unclustered.o
    11
    12    bc <<< "$(cat count-clustered.o)+$(cat count-uncluster
ed.o)" > count.o
    13
    14    grep -e '^>' proteins.fa | cut -d ' ' -f 1 | wc -l > p
roteins-count.o
    15
    16    MYCOUNT=$(cat count.o)
    17    PROTCOUNT=$(cat proteins-count.o)
    18
    19    if [[ "$MYCOUNT" -eq "$PROTCOUNT" ]]; then
    20      echo "Counts match, all's good."
    21    else
    22      echo "Not OK (MYCOUNT='$MYCOUNT', PROTCOUNT='$PROTCO
UNT')";
    23    fi
```

And run it as a target:

```
$ make check
<...output elided...>
Counts match, all's good.
```

When I create this pipeline, I worked out each step and then added it to the "Makefile." In doing so, I completely documented my work while also creating a way to execute the entire workflow in one command, `make`. Huzzah for reproducibility!

Programming

> Give someone a program, frustrate them for a day; teach them how to program, frustrate them for a lifetime. -- David Leinweber
>
> HOW DOES PROGRAMMING TAKE SO LONG YOU LITERALLY JUST TYPE WHAT YOU WANT IT TO DO THIS SHOULD BE EASY -- Jessie Newman @RainbowThyme

You absolutely must learn to program. If that makes you sad, you'll be even sadder to know that you'll probably need to know 2 or 3 languages, at least -- bash, something like Perl/Python/Ruby, almost certainly R, maybe PHP, perhaps c or Java. It depends on what you do. Each language has a niche, and you're going to be more productive if you know more than one. We are going to teach our students enough bash and Perl 6 to be productive. Once you learn the basics of reading/writing files, iterating through loops (for/while), reporting errors and such, then you can move on to most any other language (except perhaps Haskell).

Why Perl 6?

Larry Wall (https://en.wikipedia.org/wiki/Larry_Wall) created the Perl programming language around 1987. With what you've seen up to this point, you may be able understand that he essentially cobbled together bash, sed, awk, and grep. He was trying to make a basic Unix systems administration language, and so, like most other Unix languages, it was quite adept at dealing with text. In the mid-1990s, the web was taking off and people were writing CGI (common gateway interface) scripts. Perl was mature enough to be a really good fit for all the text manipulation involved. This was also about the time that bioinformatics was getting started, and most of that is also about dealing with massive quantities of text (e.g., "ACTG").

In 2000, Larry et al. decided to completely revise the language. Much can be said of Perl 5 -- and has -- that it looks like "line noise," that it's incomprehensible even to those who wrote the code, that it's unmaintainable. All this is true! Perl 6 was a completely non-backwards-compatible rewrite of the language. The designers jettisoned all that they thought bad and kept everything they thought good. Whether or not you agree with their decisions, Perl 6 looks to me to be a fine teaching language. Given what you will learn from Perl 6, you will understand "imperative," "declarative," and "functional" programming concepts that you can apply to any future language you teach yourself.

One of the downsides to choosing Perl 6 is that the language is relatively new. Although it has been in development for over 15 years, it was only "officially" release at Christmas 2015 (the long-running joke being that they kept saying it would be done "by Christmas," but they never said which Christmas). So, there are no books and very few examples to draw from on the web. On the other hand, there is a wonderful IRC community on "irc.freenode.net" in the "#perl6" chatroom, and https://docs.perl6.org is pretty comprehensive. Besides, this will build character. Anyway, I'm not trying to make you an expert Perl 6 programmer, just introducing programming concepts with a language that is actually pretty fun to play with.

In the following examples, I will evolve the programs from simpler to more complex to show you how learning higher-order programming techniques actually shorten your code and make it more maintainable. Don't be worried if you don't

quite understand the later versions. Just stick with the concepts in the earlier versions and keep reading and playing with the denser functions until you understand them.

With my rationalization complete, let's write some Perl (6)!

Hello, World

You've probably figured out already that the first thing you're supposed to write in any new language is "Hello, World." So, let's do that.

```
$ cat hello.pl6
#!/usr/bin/env perl6
put "Hello, World!"
$ ./hello.pl6
Hello, World!
```

Well, that looks almost exactly like bash and Python except that it uses `put` (like `puts` in Ruby) instead of `echo` or `print`. Maybe this won't be too hard?

OK, let's rewrite our "greeting" script in Perl6.

```
$ cat -n greet1.pl6
     1     #!/usr/bin/env perl6
     2
     3     unless (1 <= @*ARGS.elems <= 2) {
     4         note "Usage:\n\t{$*PROGRAM-NAME.IO.basename} GREE
TING [NAME]";
     5         exit 1;
     6     }
     7
     8     my ($greeting, $name) = @*ARGS;
     9
    10     put "$greeting, {$name // 'Stranger'}";
$ ./greet1.pl6 Howdy "Old Joe Clark"
Howdy, Old Joe Clark
```

Well, that looks suspiciously close to the bash version. Line 1 is our now-familiar shebang. We don't need to know the exact path to the Perl6 binary as long as it's somewhere in our $PATH. On line 3, we check the number of arguments to the script by looking at `@*ARGS` which is equivalent to bash's `$@`. In bash we had to write two separate conditionals to check if the number of arguments was with a

range, but in Perl we can write it just like in algebra class, "X <= Y <= Z" means Y is greater-than-or-equal to X and less-than-or-equal to Z. The call to `note` is a way to print to STDERR, and we're formatting a standard "Usage" statement. On line 5 we see that we can "exit 1" just as in bash.

One very bash-like thing we see on lines 4 and 10 Line 10 is that we can call a function *inside a string*. In bash, the function call is denoted with `$()` or backticks (``). In Perl curly braces `{}` create what is called a "block" of code that can be executed or used as an argument to another function. Inside the block on line 4, we're using the global/magic variable/object `$*PROGRAM-NAME` to find the `basename` of the program, and on line 10, we're using the `//` operator to say "what's in the $name variable or, if that is not defined, then the string 'Stranger'"). You can read about Perl 6's operators at https://docs.perl6.org/routine.html.

At line 8 is there's a `my`, which is new. Remember in bash how we had to `set -u` to tell bash to complain when we try to use a variable that was never initialized? That kind of checking is built into Perl so that if we were to write `put $greting` it would complain:

```
===SORRY!=== Error while compiling /Users/kyclark/work/abe487/bo
ok/perl6/./greet1.pl6
Variable '$greting' is not declared. Did you mean '$greeting'?
at /Users/kyclark/work/abe487/book/perl6/./greet1.pl6:10
```

Another difference at line 8 is that we can assign two variables, "greeting" and "name," in one line by using a list (the parentheses) on the left-hand side ("LHS" in CS parlance).

Now to talk about the differences in those sigils. We've seen them in bash, and they're in awk and sed, too, but they are usually just `$` signs. Now're we're seeing `$*SPEC` and `@*ARGS`. We need to talk about data shapes.

Data Shapes

Perl has various containers for different shapes of data. You can have one thing (a string of DNA), lots of things (a list of transcription factors), a key-value pair (metadata about a sample like species = 'H. sapiens,' age = 33, gender = 'male'), etc. Perl would call these things "scalars," "lists," and "hashes," and these shapes are almost universal to all programming languages. Perl has several other useful shapes like bags and sets, and we'll get to those later.

Perl's language designers feel that the variables should stand out from the language itself, and that the sigils (decorations on the front, cf. https://en.wikipedia.org/wiki/Sigil_(computer_programming)) on the variables should give the reader an indication of the shape of the data. Languages like Python, Ruby, Haskell, Java, make no visual distinction between their reserved words and variables.

Time to fire up `perl6` and start typing. The following show the result of typing the examples into the REPL.

Scalar $

If you have just one of a thing like our greeting or name, then you put it into a "scalar" or singular variable. These are prefixed with a `$` like `$greeting` and can hold only one value. If you set it to a second value, the first value is forever lost unless you set it to be immutable using `:=` .

```
> my $greeting = "How you doin'?";
How you doin'?
> $greeting.chars
14
> my $money := 1e+6;
1000000
> $money = 0
Cannot assign to an immutable value
  in block <unit> at <unknown file> line 1
```

Array @

When you have an undetermined number of somethings, they belong in an Array (mutable) or a List (immutable). These are plurals, and they start with the `@` sign. The items in a series are separated with commas or you can use the `<>` operator (https://docs.perl6.org/language/quoting). Perl also supports infinite lists -- just don't try to print them.

```
> my @nums = 8, 1, 43;
[8 1 43]
> put @nums.join(', ');
8, 1, 43
> my @sizes = <small medium large>
[small medium large]
> my @x = 1..10
[1 2 3 4 5 6 7 8 9 10]
> my @positive = 1..*
[...]
> @positive[^10]
(1 2 3 4 5 6 7 8 9 10)
```

Hash %

When you have a key-value association, that belongs in a hash AKA "map" or "dictionary" or "associative array," if you're not into the whole brevity thing. Yes, I know we call the `#` a "hash," but this "hash" is short for a "hash table" (https://en.wikipedia.org/wiki/Hash_table). Hashes start with the `%` sign.

```
> my %genome = species => "H_sapiens", taxid => 9606;
{species => H_sapiens, taxid => 9606}
> put %genome<species>;
H_sapiens
> %genome.keys
(species taxid)
```

In the case of each variable, something that should be very striking is that we can ask the variables to do things for us. We can ask a scalar how many "chars" (characters) it has, we can have a list join its elements together using a comma to create a string that we can print, and we can ask the hash to give us the value for some given key or even all the keys it has. We can even go meta (literally) and ask the variables what they can do for us. I will elide the output here, but you should try this in your own REPL:

```
> $greeting.^methods
(BUILD Int Num chomp pred succ simplematch match ...)
> @nums.^methods
(iterator from-iterator new STORE reification-target shape pop s
hift splice ...)
> %genome.^methods
(clone BIND-KEY name keyof of default dynamic push append ...)
```

This only begins to scratch the surface. You can read more at https://docs.perl6.org/type.html.

Scalars

Let's look at some of the things that fit in a scalar. This list isn't exhaustive, just an overture to what we'll see as we move through the text.

Numbers

Perl has many numeric types including Int (integer), UInt (unsigned integer), Num (floating point), Rat/FatRat/Rational, Real, and Complex.

Strings

A "string" (https://docs.perl6.org/type/Str) is a series of characters like "GATTAGA." If I put "2112" in quotes it's a string, but if I don't it's a number. Here's how to create the reverse complement of a strand of DNA:

```
my $dna = "GATTAGA";
> $dna.trans(<A C G T> => <T G C A>).flip
TCTAATC
```

Cool

"Cool" (https://docs.perl6.org/type/Cool) is short for "Convenient OO Loop" and can hold values that look either like numbers or strings, converting back and forth ("coercing," in the parlance) depending on how you use them.

Let's look at a situation where I have a string that I bounce back and forth from being a string or a number:

```
> my $x = "42"
42
> put "$x + 1 = " ~ $x + 1
42 + 1 = 43
```

Lots going on here. First `$x` is a string "42," and in `put "$x + 1 = "` it's treated like a string because it's being interpolated inside quotes. In `$x + 1`, `$x` is coerced to a number because I'm adding it (`+`):

```
> "42" + 1
43
> say ("42" + 1).WHAT
(Int)
```

but then to concantenate it (via `~`) to the other string, the sum of `42 + 1` is coerced to a string:

```
> say ("$x + 1 = " ~ "42" + 1).WHAT
(Str)
```

You can declare that your variables can only contain a certain Type (https://docs.perl6.org/type.html):

```
> my Int $x = "42"
Type check failed in assignment to $x; expected Int but got Str
("42")
  in block <unit> at <unknown file> line 1
> my Str $x = 42;
Type check failed in assignment to $x; expected Str but got Int
(42)
  in block <unit> at <unknown file> line 1
```

But Perl will still coerce values according to how you use them:

```
> my Str $x = "42";
42
> say $x.WHAT
(Str)
> say $x.Int.WHAT
(Int)
> say (+$x).WHAT
(Int)
> put "$x + 1 = " ~ $x + 1
42 + 1 = 43
```

Date, DateTime

The `Date` (https://docs.perl6.org/type/Date) and `DateTime` (https://docs.perl6.org/type/DateTime) types are pretty much what you'd expect:

```
> my $birthday = Date.new(1972, 5, 4);
1972-05-04
> $birthday.day-of-week
4 # Thursday
> DateTime.new(now)
2016-10-31T16:25:09.335788Z
> Date.new(now).year - $birthday.year
44
> my $days = Date.new(now) - $birthday
16251
> $days div 365, $days % 365
(44 191) # (years, days)
```

File handles (IO)

If "foo.txt" is a file that lives on your system, then you can `open` the file to get a file handle that will allow you to read the file. This is a function of I/O (input/output), and lots of interesting types inhabit IO (https://docs.perl6.org/type/IO):

```
my $fh = open "foo.txt";
for $fh.lines -> $line { say $line }
# or
for $fh.lines { .say }
```

Bool

Boolean types (https://docs.perl6.org/type/Bool, named for mathematician George Boole) are your typical `True` and `False` values, but be warned that they are a type of Enum (https://docs.perl6.org/language/typesystem#index-entry-Enumeration-_Enums-_enum), not a truely separate type.

```
> if True { put "true" } else { put "false" }
true
> True == 1
True
> 1 + True == 2
True
```

Pair

A Pair (https://docs.perl6.org/type/Pair) is a combination of a key and a value.

```
> my $bp = A => 'C';
A => C
> $bp.key
A
> $bp.value
C
> $bp.kv
(A C)
```

Proc

A "Proc" (process, https://docs.perl6.org/type/Proc) is the result of running a command outside of Perl such as "pwd":

```
> my $proc = run('pwd', :out)
Proc.new(in => IO::Pipe, out => IO::Pipe.new(:path(""),:chomp),
err => IO::Pipe, exitcode => 0, pid => Any, signal => 0, command
 => ["pwd"])
>
> $proc.out.get
/Users/kyclark
```

Sub

We'll talk about creating subroutines much later, but know that you can store them into a scalar, pass them around like data, and call them when you like.

```
> my $pony = sub { put "I dig a pony" }
sub () { #`(Sub|140377779867312) ... }
> $pony()
I dig a pony
```

Arrays

Arrays are ordered collections of things. Order is important, because later we'll talk about hashes and bags that are unordered. Arrays have lots of handy functions. Let's explore.

Length, Order

How long is that array? Here's a list of the main dogs in my life from when I was a kid to now:

```
> my @dogs = <Chaps Patton Bowzer Logan Lulu Patch>
[Chaps Patton Bowzer Logan Lulu Patch]
> @dogs.elems
6
> +@dogs
6
> @dogs.Int
6
> @dogs.Numeric
6
```

The `elems` methods returns the number of elements and is the most obvious way to find it. Putting a `+` plus sign in front coerces the array into a numerical context which returns the length of the array as does calling the `Int` and `Numeric` methods.

If I want my dogs going from current to first, I can `reverse` them:

```
> @dogs.reverse
[Patch Lulu Logan Bowzer Patton Chaps]
```

I can sort them either by their name:

```
> @dogs.sort
(Bowzer Chaps Logan Lulu Patch Patton)
```

Or, by passing a "unary" operator (one that takes a single argument), by the results of that operation such as the number of characters in the name (i.e., the length of the string):

```
> @dogs.sort(*.chars)
(Lulu Chaps Logan Patch Patton Bowzer)
> @dogs.sort(*.chars).reverse
(Bowzer Patton Patch Logan Chaps Lulu)
```

I can find all possible pairs of dogs:

```
> @dogs.combinations(2)
((Chaps Patton) (Chaps Bowzer) (Chaps Logan) (Chaps Lulu) (Chaps
 Patch) (Patton Bowzer) (Patton Logan) (Patton Lulu) (Patton Pat
ch) (Bowzer Logan) (Bowzer Lulu) (Bowzer Patch) (Logan Lulu) (Lo
gan Patch) (Lulu Patch))
```

I can find just the length of each dog's name by using `map` to apply a the `chars` function to each element:

```
> @dogs.map(*.chars)
(5 6 6 5 4 5)
> @dogs.map(&chars)
(5 6 6 5 4 5)
```

The first version is using the `chars` *method* called on each list object while the second version is applying the `chars` *function* to each element (https://docs.perl6.org/routine/chars). The leading ampersand `&` is passing the `chars` function as a reference. Note that you cannot call it like so:

```
> @dogs.map(chars)
===SORRY!=== Error while compiling:
Calling chars() will never work with proto signature ($)
------> @dogs.map(⏏chars)
```

The `map` function is something I'd really like you to understand, so let's break it down a bit. I can get the same answer (sort of) with a `for` loop:

```
> for @dogs -> $dog { say $dog.chars }
5
6
6
5
4
5
```

And I can capture the numbers by using `do`:

```
> my @chars = do for @dogs -> $dog { $dog.chars }
[5 6 6 5 4 5]
```

Or, more briefly:

```
> my @chars = do for @dogs { .chars }
[5 6 6 5 4 5]
```

So the `map` *function* just turns that around a bit:

```
> my @chars = map { .chars }, @dogs
[5 6 6 5 4 5]
```

And the `map` *method* (of an Array) turns that around again to look more like the `for` version:

```
> my @chars = @dogs.map({ .chars })
[5 6 6 5 4 5]
> my @chars = @dogs.map(*.chars)
[5 6 6 5 4 5]
> my @chars = @dogs.map: *.chars
[5 6 6 5 4 5]
```

You see there is more than one way to write a map. We'll break this down later.

The elements in a Array are not limited to scalars. Using `map`, I create a Array of lists that each combines a dog's name with its length using the `Z` "zip" operator (https://docs.perl6.org/routine/Z):

```
> @dogs Z @dogs.map(*.chars)
((Chaps 5) (Patton 6) (Bowzer 6) (Logan 5) (Lulu 4) (Patch 5))
```

Does that makes sense? Zip takes two lists and combines them element-by-element, stopping on the shorter list:

```
> 1..10 Z 'a'..'z'
((1 a) (2 b) (3 c) (4 d) (5 e) (6 f) (7 g) (8 h) (9 i) (10 j))
> 1..* Z 'Bowzer'.comb
((1 B) (2 o) (3 w) (4 z) (5 e) (6 r))
```

Lastly, I'll show you how `sum` (https://docs.perl6.org/routine/sum) total number of characters in all the dog names:

```
> @dogs.map(*.chars).sum
31
```

Iterating

One of the most common array operations is to iterate over the members while keeping track of the position. Here's a script that breaks a string (here maybe some DNA) into a list using the `comb` method and prints the position and the

letter:

```
$ cat -n iterate1.pl6
     1    #!/usr/bin/env perl6
     2
     3    sub MAIN (Str $dna) {
     4        my $i = 0;
     5        for $dna.comb -> $letter {
     6            $i++;
     7            say "$i: $letter";
     8        }
     9    }
 $ ./iterate1.pl6 AACTAG
1: A
2: A
3: C
4: T
5: A
6: G
```

This is so common that Arrays have shorter ways to do this:

```
$ cat -n iterate2.pl6
     1    #!/usr/bin/env perl6
     2
     3    sub MAIN (Str $dna) {
     4        for $dna.comb.kv -> $k, $v {
     5            say "{$k+1}: $v";
     6        }
     7    }
$ ./iterate2.pl6 AACTAG
1: A
2: A
3: C
4: T
5: A
6: G
```

Positions in Perl Arrays and Strings start at 0, so I have to add 1 to `$k`. Notice that I can run code *inside a string* by putting `{}` curly braces around it.

I don't have to give the pointy block signature `-> $k, $v` bit. I can use `$^k` and `$^v` (or `$^a` and `$^b` or whatever) to refer to the first and second arguments (in sorted Unicode order) to the block (https://docs.perl6.org/language/variables#index-entry-%24%5E):

```
$ cat -n iterate3.pl6
     1    #!/usr/bin/env perl6
     2
     3    sub MAIN (Str $dna) {
     4        for $dna.comb.kv { say join ": ", $^k + 1, $^v }
     5    }
$ ./iterate3.pl6 AACTAG
1: A
2: A
3: C
4: T
5: A
6: G
```

Here's a version using `pairs` to get a List of Pair types (https://docs.perl6.org/type/Pair) with the index (position) as the "key" and the letter as the "value":

```
$ cat -n iterate4.pl6
     1    #!/usr/bin/env perl6
     2
     3    sub MAIN (Str $dna) {
     4        for $dna.comb.pairs -> $pair {
     5            printf "%s: %s\n", $pair.key + 1, $pair.value;
     6        }
     7    }
 $ ./iterate3.pl6 AACTAG
1: A
2: A
3: C
4: T
5: A
6: G
```

Again, I don't have to have the `-> $pair` bit if I use the `^` twigil. I can just refer to the one positional argument as `$^pair` and call the `key` and `value` methods on that:

```
$ cat -n iterate5.pl6
     1    #!/usr/bin/env perl6
     2
     3    sub MAIN (Str $dna) {
     4        for $dna.comb.pairs { say join ': ', $^pair.key +
1, $^pair.value }
     5    }
[saguaro@~/work/metagenomics-book/perl6/lists]$ ./iterate5.pl6 A
ACTAG
1: A
2: A
3: C
4: T
5: A
6: G
```

Or I can use the `:` twigil (https://docs.perl6.org/language/variables#The_:_Twigil) a way to declare named parameters:

```
$ cat -n iterate6.pl6
     1    #!/usr/bin/env perl6
     2
     3    sub MAIN (Str $dna) {
     4        for $dna.comb.pairs -> (:$key, :$value) {
     5            say join ': ', $key + 1, $value;
     6        }
     7    }
$ ./iterate6.pl6 AACTAG
1: A
2: A
3: C
4: T
5: A
6: G
```

Filtering

Often you want to choose or remove certain members of an array. Let's find only the Gs and Cs in a string (https://en.wikipedia.org/wiki/GC-content). Note that I uppercase (`uc`) the `$dna` first so that I only have to check for one case of letters:

```
$ cat -n gc1.pl6
     1    #!/usr/bin/env perl6
     2
     3    sub MAIN (Str $dna) {
     4        my @gc;
     5        for $dna.uc.comb -> $base {
     6            @gc.push($base) if $base eq 'G' || $base eq 'C
';
     7        }
     8        say "$dna has {@gc.elems}";
     9    }
$ ./gc1.pl6 AACTAG
AACTAG has 2
```

But `grep` is a much shorter way to find all the elements matching a given condition. Like `map`, `grep` takes a block of code that will be executed for each member of the array. Any elements for which the block evaluates to "True-ish" are allowed through. The `$_` (topic, thing, "it") variable has the current element, so the code is asking "if the thing is a 'G' or if the thing is a 'C'". One can use the `*` to represent "it" and eschew the curly brackets:

```
> grep {$_ > 5}, 1..10
(6 7 8 9 10)
> grep * > 5, 1..10
(6 7 8 9 10)
```

Here's the GC filter written with `grep`:

```
$ cat -n gc2.pl6
     1    #!/usr/bin/env perl6
     2
     3    sub MAIN (Str $dna) {
     4        my @gc = $dna.uc.comb.grep({$_ eq 'G' || $_ eq 'C'
});
     5        say "$dna has {@gc.elems}";
     6    }
```

Here I'll use a Junction (https://docs.perl6.org/type/Junction) to compare to "G or C" in one go:

```
$ cat -n gc3.pl6
     1    #!/usr/bin/env perl6
     2
     3    sub MAIN (Str $dna) {
     4        my @gc = $dna.uc.comb.grep(* eq 'G' | 'C');
     5        say "$dna has {@gc.elems}";
     6    }
```

Another way to write the `|` Junction is with `any`. The `so` routine (https://docs.perl6.org/routine/so) collapses the various Booleans down to a single value.

```
> 'G' eq 'G' | 'C'
any(True, False)
> so 'G' eq 'G' | 'C'
True
> 'G' eq any(<G C>)
any(True, False)
> so 'G' eq any(<G C>)
True
```

It's extremely common to use regular expressions (https://docs.perl6.org/type/Regex) to filter lists. We'll cover these more later, but here I'm using a character class to represent either "G or C":

```
$ cat -n gc4.pl6
     1    #!/usr/bin/env perl6
     2
     3    sub MAIN (Str $dna) {
     4        my @gc = $dna.uc.comb.grep(/<[GC]>/);
     5        say "$dna has {@gc.elems}";
     6    }
```

Here's how you can find the prime numbers between 1 and 10:

```
> (1..10).grep(*.is-prime)
(2 3 5 7)
```

Classification

We can group elements based on predicates we supply. Here is how we can split up the numbers 1 through 10 based on whether they are or are not even divisible by 2:

```
> 2 %% 2
True
> 3 %% 2
False
> (1..10).classify(* %% 2)
{False => [1 3 5 7 9], True => [2 4 6 8 10]}
```

Going back to our G-C counter, we can group each base into whether it is or isn't a "G" or a "C":

```
$ cat -n gc5.pl6
     1    #!/usr/bin/env perl6
     2
     3    sub MAIN (Str $dna) {
     4        my %hash = $dna.uc.comb.categorize({?/<[GC]>/});
     5        say "$dna has {%hash<True>.elems}";
     6    }
```

In my opinion, it's not intuitive to use "True" or "False," so let's provide our own String value for the name of the bucket we want:

```
$ cat -n gc6.pl6
     1    #!/usr/bin/env perl6
     2
     3    sub MAIN (Str $dna) {
     4        my %hash = $dna.uc.comb.categorize({ /<[GC]>/ ?? '
GC' !! 'Other' });
     5        say "$dna has {%hash<GC>.elems}";
     6    }
```

It might help to see that one in the REPL:

```
> 'AACTAG'.uc.comb.classify({?/<[GC]>/})
{False => [A A T A], True => [C G]}
> 'AACTAG'.uc.comb.classify({/<[GC]>/ ?? 'GC' !! 'Other'})
{GC => [C G], Other => [A A T A]}
```

`classify` takes a code block and uses the resulting string to put the element into a bucket. Here I've used the same regular expression `/<[GC]>/` to return the string "GC" if it's a match or "Other" if it's not. The combination of the `?? !!` is the "ternary" operator that we'll talk about more later. The resulting Hash has a key called "GC" and its value is a list containing the "G" and "C" found in the string.

So you're seeing that lists can be inside of other Lists as well as inside of Hashes and other data structures.

I can classify my `@dogs` based on the length of their names using that same syntax variations we saw for `map`:

```
> @dogs.classify({.chars})
{4 => [Lulu], 5 => [Chaps Logan Patch], 6 => [Patton Bowzer]}
> @dogs.classify(*.chars)
{4 => [Lulu], 5 => [Chaps Logan Patch], 6 => [Patton Bowzer]}
> @dogs.classify(&chars)
{4 => [Lulu], 5 => [Chaps Logan Patch], 6 => [Patton Bowzer]}
```

Lists can also be composed of Pairs (https://docs.perl6.org/type/Pair). Here I'll redeclare my `@dogs` with their names as the "key" and thier sex as the "value." Then I can `classify` them on their `value`:

```
> my @dogs = Chaps => 'male', Patton => 'male', Bowzer => 'male'
, Logan => 'male', Lulu => 'female', Patch => 'male'
[Chaps => male Patton => male Bowzer => male Logan => male Lulu
=> female Patch => male]
> @dogs.classify(-> $dog {$dog.value})
{female => [Lulu => female], male => [Chaps => male Patton => ma
le Bowzer => male Logan => male Patch => male]}
> @dogs.classify({$^dog.value})
{female => [Lulu => female], male => [Chaps => male Patton => ma
le Bowzer => male Logan => male Patch => male]}
> @dogs.classify({.value})
{female => [Lulu => female], male => [Chaps => male Patton => ma
le Bowzer => male Logan => male Patch => male]}
> @dogs.classify(*.value)
{female => [Lulu => female], male => [Chaps => male Patton => ma
le Bowzer => male Logan => male Patch => male]}
```

Picking random elements

To finish off, let's write a Shakespearean insult generator that uses the `pick` method to randomly choose some perjoratives:

```
$ cat -n insult.pl6
     1    #!/usr/bin/env perl6
     2
     3    sub MAIN (Int :$n=1) {
     4        my @adjectives = qw{scurvy old filthy scurilous la
scivious
     5            foolish rascaly gross rotten corrupt foul loat
hsome irksome
     6            heedless unmannered whoreson cullionly false f
ilthsome
     7            toad-spotted caterwauling wall-eyed insatiate
vile peevish
     8            infected sodden-witted lecherous ruinous indis
tinguishable
     9            dishonest thin-faced slanderous bankrupt base
detestable
    10            rotten dishonest lubbery};
    11        my @nouns = qw{knave coward liar swine villain beg
gar
    12            slave scold jolthead whore barbermonger fishmo
nger carbuncle
    13            fiend traitor block ape braggart jack milksop
boy harpy
    14            recreant degenerate Judas butt cur Satan ass c
oxcomb dandy
    15            gull minion ratcatcher maw fool rogue lunatic
varlet worm};
    16
    17        printf "You %s, %s, %s %s!\n",
    18            @adjectives.pick(3), @nouns.pick for ^$n;
    19    }
$ ./insult.pl6 -n=5
You foul, dishonest, old recreant!
You irksome, gross, false degenerate!
You old, dishonest, toad-spotted jack!
You ruinous, unmannered, foolish slave!
You scurry, slanderous, peevish harpy!
```

You can also use `roll` so that each selection is made independently. Above I'm using the `qw{}` "quote-word" operator (https://docs.perl6.org/language/quoting#index-entry-qw_word_quote) to create a list of words rather than writing:

```
my @adjectives = "scurvy", "old", "filthy";
```

You should spend a few minutes reading about all the different quoting options available as they will come in handy.

Hashes

Hashes (https://docs.perl6.org/type/Hash) basically Arrays of Pairs (https://docs.perl6.org/type/Pair). They are also known as dictionaries or maps. The key in a Hash is usually a string, and the value can be pretty much anything. The order of the Pairs is not preserved in creation order:

```
> my %dog = name => 'Patch', age => 4, color => 'white', weight
=> '31 lbs'
{age => 4, color => white, name => Patch, weight => 31 lbs}
```

If I ask for one key, I get one value:

```
> %dog<name>
Patch
```

I can also ask for more than one key, and I will get a list:

```
> %dog<age color>
(4 white)
```

I can ask for all the keys and values using the `kv` method:

```
> %dog.kv
(color white name Patch age 4 weight 31 lbs)
> %dog.pairs
(color => white name => Patch age => 4 weight => 31 lbs)
> say join ", ", %dog.map(*.kv.join(": "))
color: white, name: Patch, age: 4, weight: 31 lbs
> %dog.pairs.map(*.kv.join(": ")).join(", ")
color: white, name: Patch, age: 4, weight: 31 lbs
```

If I want the keys back in a particular order, I either have to ask for them explicitly or `sort` them:

```
> %dog.keys
(color name age weight)
> %dog.keys.sort
(age color name weight)
```

I can store more than one thing for a value by using lists as in this amino-acid-to-codons table:

```
$ cat -n aa.pl6
     1    #!/usr/bin/env perl6
     2
     3    my %aa = Isoleucine    => <ATT ATC ATA>,
     4             Leucine       => <CTT CTC CTA CTG TTA TTG>,
     5             Valine        => <GTT GTC GTA GTG>,
     6             Phenylalanine => <TTT TTC>,
     7             Methionine    => <ATG>,
     8             Cysteine      => <TGT TGC>,
     9             Alanine       => <GCT GCC GCA GCG>,
    10             Glycine       => <GGT GGC GGA GGG>,
    11             Proline       => <CCT CCC CCA CCG>,
    12             Threonine     => <ACT ACC ACA ACG>,
    13             Serine        => <TCT, TCC, TCA, TCG, AGT, AG
C>,
    14             Tyrosine      => <TAT TAC>,
    15             Tryptophan    => <TGG>,
    16             Glutamine     => <CAA CAG>,
    17             Asparagine    => <AAT AAC>,
    18             Histidine     => <CAT CAC>,
    19             Glutamic_acid => <GAA GAG>,
    20             Aspartic_acid => <GAT GAC>,
    21             Lysine        => <AAA AAG>,
    22             Arginine      => <CGT CGC CGA CGG AGA AGG>,
    23             Stop          => <TAA TAG TGA>;
    24
    25    for %aa.keys.sort -> $key {
    26        printf "%-15s = %s\n", $key, %aa{ $key }.join(", "
);
    27    }
$ ./aa.pl6
```

```
Alanine         = GCA, GCC, GCG, GCT
Arginine        = AGA, AGG, CGA, CGC, CGG, CGT
Asparagine      = AAC, AAT
Aspartic_acid   = GAC, GAT
Cysteine        = TGC, TGT
Glutamic_acid   = GAA, GAG
Glutamine       = CAA, CAG
Glycine         = GGA, GGC, GGG, GGT
Histidine       = CAC, CAT
Isoleucine      = ATA, ATC, ATT
Leucine         = CTA, CTC, CTG, CTT, TTA, TTG
Lysine          = AAA, AAG
Methionine      = ATG
Phenylalanine   = TTC, TTT
Proline         = CCA, CCC, CCG, CCT
Serine          = AGC, AGT,, TCA,, TCC,, TCG,, TCT,
Stop            = TAA, TAG, TGA
Threonine       = ACA, ACC, ACG, ACT
Tryptophan      = TGG
Tyrosine        = TAC, TAT
Valine          = GTA, GTC, GTG, GTT
```

Hashes are great for counting things. You don't have to check if a key/value pair exists first -- just add one to a key and it will be created and start counting from 0:

```
$ cat -n word-count.pl6
     1    #!/usr/bin/env perl6
     2
     3    sub MAIN (Str $file where *.IO.f) {
     4        my %count;
     5        for $file.IO.words -> $word {
     6            %count{ $word }++;
     7        }
     8
     9        for %count.grep(*.value > 1) -> (:$key, :$value) {
    10            printf "Saw '%s' %s time.\n", $key, $value;
    11        }
    12    }
$ ./word-count.pl6 because.txt
Saw 'a' 3 times.
Saw 'of' 2 times.
Saw 'for' 2 times.
Saw 'I' 3 times.
Saw 'The' 3 times.
Saw 'but' 3 times.
Saw 'And' 2 times.
Saw 'We' 5 times.
Saw 'passed' 3 times.
Saw 'the' 6 times.
```

You can set the keys and values all-at-once like above or one-at-a-time:

```
$ cat -n bridge-of-death.pl6
     1    #!/usr/bin/env perl6
     2
     3    my %answers;
     4    for "name", "quest", "favorite color" -> $key {
     5        %answers{ $key } = prompt "What is your $key? ";
     6    }
     7
     8    if %answers{"favorite color"} eq "Blue" {
     9        put "Right.  Off you go.";
    10    }
    11    else {
    12        put "AAAAAAAAAAaaaaaaaaa!"
    13    }
$ ./bridge-of-death.pl6
What is your name? My name is Sir Launcelot of Camelot
What is your quest? To seek the Holy Grail
What is your favorite color? Blue
Right.  Off you go.
$ ./bridge-of-death.pl6
What is your name? Sir Galahad of Camelot.
What is your quest? I seek the Holy Grail.
What is your favorite color? Blue.  No yel--
AAAAAAAAAAaaaaaaaaa!
```

Sometimes you need to see if a key is defined in a hash, and for that you use the adverb `:exists`:

```
$ cat -n gashlycrumb.pl6
     1    #!/usr/bin/env perl6
     2
     3    # Text by Edward Gorey
     4
     5    my %alphabet = q:to/END/.lines.map(-> $line { $line.su
bstr(0,1) => $line });
     6    A is for Amy who fell down the stairs.
     7    B is for Basil assaulted by bears.
     8    C is for Clara who wasted away.
     9    D is for Desmond thrown out of a sleigh.
```

```
    10    E is for Ernest who choked on a peach.
    11    F is for Fanny sucked dry by a leech.
    12    G is for George smothered under a rug.
    13    H is for Hector done in by a thug.
    14    I is for Ida who drowned in a lake.
    15    J is for James who took lye by mistake.
    16    K is for Kate who was struck with an axe.
    17    L is for Leo who choked on some tacks.
    18    M is for Maud who was swept out to sea.
    19    N is for Neville who died of ennui.
    20    O is for Olive run through with an awl.
    21    P is for Prue trampled flat in a brawl.
    22    Q is for Quentin who sank on a mire.
    23    R is for Rhoda consumed by a fire.
    24    S is for Susan who perished of fits.
    25    T is for Titus who flew into bits.
    26    U is for Una who slipped down a drain.
    27    V is for Victor squashed under a train.
    28    W is for Winnie embedded in ice.
    29    X is for Xerxes devoured by mice.
    30    Y is for Yorick whose head was bashed in.
    31    Z is for Zillah who drank too much gin.
    32    END
    33
    34    loop {
    35        my $letter = uc prompt "Letter [0 to quit]? ";
    36
    37        if $letter eq '0' {
    38            put "Bye.";
    39            exit;
    40        }
    41        elsif %alphabet{ $letter }:exists {
    42            put %alphabet{ $letter };
    43        }
    44        else {
    45            put "I'm sorry, I don't know that letter ($let
ter).";
    46        }
    47    }
$ ./gashlycrumb.pl6
```

```
Letter [0 to quit]? j
J is for James who took lye by mistake.
Letter [0 to quit]? 9
I'm sorry, I don't know that letter (9).
Letter [0 to quit]? 0
Bye.
```

Sets

Bags

Bags are like instant histograms. They're perfect for counting the number of times each item occurs in a set. Here's a quick way to count the lengths of each line in a file:

```
$ cat -n line-count-histogram1.pl6
     1    #!/usr/bin/env perl6
     2
     3    sub MAIN (Str $file) {
     4        my $bag = $file.IO.lines.map(*.chars).Bag;
     5        put $bag.pairs.sort(*.key).map(*.join("\t")).join(
"\n");
     6    }
```

Lots of stuff happening in just two lines! Let's break it down. Given this file:

```
$ cat words
a
an
the
frog
cat
dog
horse
```

Here is how we get there. Read all the lines from the file:

```
> 'words'.IO.lines
(a an the frog cat dog horse)
```

Now `map` each line into a function to count how many characters are in the line. The `*` stands for the current line as it passes through the function:

```
> 'words'.IO.lines.map(*.chars)
(1 2 3 4 3 3 5)
```

Now "Bag" those -- Perl will count each unique entry. Since there are three words that have three characters, you see "3(3)":

```
> my $bag = 'words'.IO.lines.map(*.chars).Bag
bag(5, 4, 3(3), 1, 2)
```

Get the `pairs` from the Bag where the `key` represents the length of the word and the `value` is the number of times it was seen:

```
> $bag.pairs
(5 => 1 4 => 1 3 => 3 1 => 1 2 => 1)
```

You see these are not returned in any particular order, so let's have them sorted by the `key`:

```
> $bag.pairs.sort(*.key)
(1 => 1 2 => 1 3 => 3 4 => 1 5 => 1)
```

To keep this on one line, I'm going to join the `key` / `value` with a colon and each member of the bag with a comma:

```
> $bag.pairs.sort(*.key).map(*.kv.join(":"))
(1:1 2:1 3:3 4:1 5:1)
> $bag.pairs.sort(*.key).map(*.kv.join(":")).join(", ")
1:1, 2:1, 3:3, 4:1, 5:1
```

If I would rather see the words sorted by the number of times they occur with the most frequent ones first, it would look like this:

```
>  $bag.pairs.sort(*.value).reverse.map(*.kv.join(":")).join(",
")
3:3, 2:1, 1:1, 4:1, 5:1
```

Here is the output from the script:

```
$ ./line-count-histogram1.pl6 words
1    1
2    1
3    3
4    1
5    1
$ ./line-count-histogram1.pl6 /usr/share/dict/words
1    52
2    160
3    1420
4    5272
5    10230
6    17706
7    23869
8    29989
9    32403
10    30878
11    26013
12    20462
13    14939
14    9765
15    5925
16    3377
17    1813
18    842
19    428
20    198
21    82
22    41
23    17
24    5
```

SetHash, BagHash

Sets and Bags are immutable:

```
> my $bag = <foo bar foo baz>.Bag
bag(foo(2), baz, bar)
> $bag<foo> = 3
Cannot modify an immutable Int
  in block <unit> at <unknown file> line 1
```

If you want to change them, use the `SetHash` (https://docs.perl6.org/type/SetHash) and `BagHash` (https://docs.perl6.org/type/BagHash) variants:

```
> my $baghash = <foo bar foo baz>.BagHash
BagHash.new(foo(2), baz, bar)
> $baghash<foo>++
2
> $baghash
BagHash.new(foo(3), baz, bar)
```

These are both excellent data structures for when you have a unique group of items from which you would like to draw such that they are removed from the container. When I was writing my Blackjack game, I found a `Set` to be the natural container for a deck of 52 unique cards:

```
> my @suites = <D H C S>
[D H C S]
> my @faces = <2 3 4 5 6 7 8 9 10 J Q K A>
[2 3 4 5 6 7 8 9 10 J Q K A]
> my $deck = (@faces X @suites).map(~*).SetHash
SetHash.new(3 C, 5 H, 7 S, 8 D, 10 S, 2 D, 2 H, 9 D, Q S, A C, 6
 H, 6 C, 10 C, Q H, J S, 3 H, 9 H, A H, J D, 7 D, 8 H, Q C, 10 D
, J C, 3 D, 4 D, 6 D, 8 C, 5 C, 5 D, 4 H, 2 S, K D, 8 S, 7 C, 4
S, K C, 7 H, K H, K S, 6 S, 5 S, J H, A D, A S, 10 H, 9 S, 4 C,
2 C, 9 C, 3 S, Q D)
> $deck.elems
52
> $deck.grab(2).join(', ')
9 C, 9 D
> $deck.elems
50
```

The `X` (https://docs.perl6.org/routine/X) operator crosses two lists to produce a `Pair` for every combination, then I `map` each `Pair` into a stringification operation `~*` because the keys of a Set or Bag are strings. Rather than stringifying the pairs, I could flatten the whole list and use `pairup`:

```
> ((@faces X @suites) xx 2).flat.pairup
(2 => D 2 => H 2 => C 2 => S 3 => D 3 => H 3 => C 3 => S 4 => D
4 => H 4 => C 4 => S 5 => D 5 => H 5 => C 5 => S 6 => D 6 => H 6
 => C 6 => S 7 => D 7 => H 7 => C 7 => S 8 => D 8 => H 8 => C 8
=> S 9 => D 9 => H 9 => C 9 => S 10 => D 10 => H 10 => C 10 => S
 J => D J => H J => C J => S Q => D Q => H Q => C Q => S K => D
K => H K => C K => S A => D A => H A => C A => S 2 => D 2 => H 2
 => C 2 => S 3 => D 3 => H 3 => C 3 => S 4 => D 4 => H 4 => C 4
=> S 5 => D 5 => H 5 => C 5 => S 6 => D 6 => H 6 => C 6 => S 7 =
> D 7 => H 7 => C 7 => S 8 => D 8 => H 8 => C 8 => S 9 => D 9 =>
 H 9 => C 9 => S 10 => D 10 => H 10 => C 10 => S J => D J => H J
 => C J => S Q => D Q => H Q => C Q => S K => D K => H K => C K
=> S ...)
> ((@faces X @suites) xx 2).flat.pairup.elems
104
```

When I `grab` cards (at random) from the Set, they are removed so that I don't have to worry about seeing them again. When I realized that casinos regularly draw from a stack of many decks of cards, I realized I'd have to turn to a `BagHash` so that I could keep track of how many, e.g., Jack of Diamonds I have. I used the `xx` (https://docs.perl6.org/routine/xx) list repetition operator. Take a look at that:

```
> 'YYZ' xx 3
(YYZ YYZ YYZ)
> 10 xx 3
(10 10 10)
> foo => 'bar' xx 3
foo => (bar bar bar)
> { foo => 'bar' } xx 3
({foo => bar} {foo => bar} {foo => bar})
> [1,2] xx 3
([1 2] [1 2] [1 2])
```

Depending on how many decks I decide to deal from, it's easy to keep track of how many of each card has been dealt so far:

```
> my $baghash = (((@faces X @suites).map(~*)) xx 2).flat.BagHash
BagHash.new(3 C(2), 5 H(2), 7 S(2), 8 D(2), 10 S(2), 2 D(2), 2 H
(2), 9 D(2), Q S(2), A C(2), 6 H(2), 6 C(2), 10 C(2), Q H(2), J
S(2), 3 H(2), 9 H(2), A H(2), J D(2), 7 D(2), 8 H(2), Q C(2), J
C(2), 3 D(2), 4 D(2), 6 D(2), 8 C(2), 10 D(2), K D(2), 2 S(2), 4
 H(2), 5 D(2), 5 C(2), K C(2), 4 S(2), 7 C(2), 8 S(2), K S(2), K
 H(2), 7 H(2), A S(2), A D(2), J H(2), 5 S(2), 6 S(2), 2 C(2), 4
 C(2), 9 S(2), 10 H(2), Q D(2), 3 S(2), 9 C(2))
> $baghash.grab(60).elems
60
> $baghash.elems
37
```

Mixes

Mixes and MixHashes are like Bags and BagHashes except that the weights are Reals rather than Ints.

I Am Greet

Let's revisit our greeting script and see how we can do much better than bash-Perl.

```
$ cat -n greet2.pl6
     1    #!/usr/bin/env perl6
     2
     3    sub MAIN ($greeting, $name='Stranger') {
     4        put "$greeting, $name";
     5    }
$ ./greet2.pl6
Usage:
  ./greet2.pl6 <greeting> [<name>]
$ ./greet2.pl6 --help
Usage:
  ./greet2.pl6 <greeting> [<name>]
$ ./greet2.pl6 Hello
Hello, Stranger
$ ./greet2.pl6 Hello Kenny
Hello, Kenny
```

Just look at that script. Just look at it! Do you see how much disappeared, and yet we still have all the functionality that we wrote before?! We have a usage statement, we have required and optional arguments, we have variable assignment, we have sane defaults, we have clean syntax.

So what is this `MAIN` thing? It's a subroutine that gets run automatically with the arguments to the script (https://docs.perl6.org/language/functions#index-entry-MAIN). The great masters of old passed down the wisdom that a good starting point for programs should be a function called "main," and so we do.

Now, let's add named arguments:

```
$ cat -n greet3.pl6
     1  #!/usr/bin/env perl6
     2
     3  sub MAIN (:$greeting!, :$name='Stranger') {
     4      put "$greeting, $name";
     5  }
$ ./greet3.pl6
Usage:
  ./greet3.pl6 --greeting=<Any> [--name=<Any>]
$ ./greet3.pl6 --greeting=Salutations
Salutations, Stranger
$ ./greet3.pl6 --name=Wilbur --greeting=Salutations
Salutations, Wilbur
```

By putting a colon `:` in front of the argument names, we used a shorthand to create a `Pair` type (https://docs.perl6.org/type/Pair). This was much easier than all that `getopt` business in bash, and naming the arguments lets the user specify them in any order desired. The `!` at the end of `:$greeting!` indicates that this is a required argument, while the `=` after `$name` indicates a default value for an optional argument. You can learn more about the signatures of subroutines https://docs.perl6.org/type/Signature.

Did you notice also that the generated help included a hint as the type of data wanted by each argument? The `(Any)` is a data type (https://docs.perl6.org/type.html) in Perl 6 at the top of the type hierarchy (https://docs.perl6.org/type/Any). We could indicate that we want strings by adding a type to the signature:

```
$ cat -n greet4.pl6
     1     #!/usr/bin/env perl6
     2
     3     sub MAIN (Str :$greeting!, Str :$name='Stranger') {
     4         put "$greeting, $name";
     5     }
$ ./greet4.pl6
Usage:
  ./greet4.pl6 --greeting=<Str> [--name=<Str>]
$ ./greet4.pl6 --greeting="Top o' the morning"
Top o' the morning, Stranger
$ ./greet4.pl6 --greeting="Top o' the morning" --name=11
Top o' the morning, 11
```

We got off to a nice start there with the USAGE telling us to provide strings, so why didn't it complain about "11"? The problem is that everything coming from the command line looks like a string. There are ways to handle this, but they can't be so easily fixed. We'll come back to this later.

The MAIN Thing

In the last chapter, we started playing with the `MAIN` subroutine to get our arguments. Let's explore that a bit more.

If you're coming to Perl 6 from Perl 5, the global variable `@*ARGS` (https://docs.perl6.org/language/variables\#index-entry-%40%2AARGS\) will look familiar to you as the place to get the command-line arguments to your program:

```
$ cat -n main1.pl6
     1    #!/usr/bin/env perl6
     2
     3    put "ARGS = ", @*ARGS.join(', ');
$ ./main1.pl6 foo bar baz
ARGS = foo, bar, bar
```

The `@` is the sigil that denotes the variable as an array, and the `*` is the "twigle" that denotes that the variable is a global. If you follow the above link to the `@*ARGS` documentation, you'll find a whole host of other dynamic and environmental variables like user, hostname, PID, cwd, etc.

Dealing directly with `@*ARGS` is fine your program accepts a few positional arguments where the first argument means one thing (name), the second argument another thing (rank), the third another (serial number), etc:

```
$ cat -n main2.pl6
     1    #!/usr/bin/env perl6
     2
     3    my ($name, $rank, $serial-num) = @*ARGS;
     4    put "name ($name) rank ($rank) serial number ($serial-
num)";
$ ./main2.pl6 Patch Private 1656401
name (Patch) rank (Private) serial number (1656401)
```

Or if all the arguments are homogenous (e.g., a list of files to process), this works reasonably well:

```
$ cat -n main3.pl6
     1    #!/usr/bin/env perl6
     2
     3    for @*ARGS -> $file {
     4        put "Processing '$file'";
     5    }
     6
     7    put "Done."
[saguaro@~/work/perl6/main]$ ./main3.pl6 foo bar baz
Processing 'foo'
Processing 'bar'
Processing 'baz'
Done.
```

If we write the special `MAIN` (https://docs.perl6.org/language/functions\#index-entry-MAIN) subroutine, we can automatically assign the variables instead of taking them from `@*ARGS` :

```
$ cat -n main4.pl6
     1    #!/usr/bin/env perl6
     2
     3    sub MAIN ($name, $rank, $serial-num) {
     4        put "name ($name) rank ($rank) serial number ($ser
ial-num)";
     5    }
```

Perl will enforce the number of arguments and generate help when supplied the wrong number or when requested:

```
$ ./main4.pl6 Patch Private
Usage:
  ./main4.pl6 <name> <rank> <serial-num>
$ ./main4.pl6 Patch Private 1656401
name (Patch) rank (Private) serial number (1656401)
$ ./main4.pl6 -h
Usage:
  ./main4.pl6 <name> <rank> <serial-num>
$ ./main4.pl6 --help
Usage:
  ./main4.pl6 <name> <rank> <serial-num>
```

You can also add type (https://docs.perl6.org/type.html) constraints:

```
$ cat -n main5.pl6
     1    #!/usr/bin/env perl6
     2
     3    sub MAIN (Str $name, Str $rank, Int $serial-num) {
     4        put "name ($name) rank ($rank) serial number ($ser
ial-num)";
     5    }
$ ./main5.pl6 foo bar baz
Usage:
  ./main5.pl6 <name> <rank> <serial-num>
$ ./main5.pl6 foo bar 123
name (foo) rank (bar) serial number (123)
```

To make them named arguments, simply prefix with a `:` and then you can supply the arguments in any order:

```
$ cat -n main6.pl6
     1    #!/usr/bin/env perl6
     2
     3    sub MAIN (Str :$name, Str :$rank, Int :$serial-num) {
     4        put "name ($name) rank ($rank) serial number ($ser
ial-num)";
     5    }
$ ./main6.pl6 -h
Usage:
  ./main6.pl6 [--name=<Str>] [--rank=<Str>] [--serial-num=<Int>]
$ ./main6.pl6 --name=Patch --serial-num=1656401 --rank=Private
name (Patch) rank (Private) serial number (1656401)
```

There's a problem if I don't supply any arguments to this last version:

```
$ ./main6.pl6
Use of uninitialized value of type Str in string context.
Methods .^name, .perl, .gist, or .say can be used to stringify i
t to something meaningful.
  in sub MAIN at ./main6.pl6 line 3
Use of uninitialized value of type Str in string context.
Methods .^name, .perl, .gist, or .say can be used to stringify i
t to something meaningful.
  in sub MAIN at ./main6.pl6 line 3
Use of uninitialized value of type Int in string context.
Methods .^name, .perl, .gist, or .say can be used to stringify i
t to something meaningful.
  in sub MAIN at ./main6.pl6 line 3
name () rank () serial number ()
```

Perl is complaining that it can't print uninitialized values. I didn't declare that any of the arguments are required -- which I can do by post-fixing the variables with `!` -- nor did I provide defaults for those that are not -- which I can do with `=`:

```
$ cat -n main7.pl6
     1    #!/usr/bin/env perl6
     2
     3    sub MAIN (Str :$name!, Str :$rank='NA', Int :$serial-n
um=0) {
     4        put "name ($name) rank ($rank) serial number ($ser
ial-num)";
     5    }
$ ./main7.pl6
Usage:
  ./main7.pl6 --name=<Str> [--rank=<Str>] [--serial-num=<Int>]
```

The absence of `[]` around the `--name` argument indicates to the user that the argument is required, while `--rank` and `--serial-num` are optional:

```
$ ./main7.pl6 --name=Patch
name (Patch) rank (NA) serial number (0)
$ ./main7.pl6 --name=Patch --serial-num=1656401 --rank=Private
name (Patch) rank (Private) serial number (1656401)
```

Even though Perl is checking the types (string, integer) for our arguments, it's not enforcing any validation on the value. We can add a where clause (https://docs.perl6.org/type/Signature\#index-entry-where_clause_%28Signature%29\) to ensure we only get a positive value for `--serial-num`:

```
$ cat -n main8.pl6
     1    #!/usr/bin/env perl6
     2
     3    sub MAIN (Str :$name!, Str :$rank='NA', Int :$serial-n
um where * > 0 = 1) {
     4        put "name ($name) rank ($rank) serial number ($ser
ial-num)";
     5    }
$ ./main8.pl6 --name=Patch --serial-num=-10
Usage:
  ./main8.pl6 --name=<Str> [--rank=<Str>] [--serial-num=<Int>]
$ ./main8.pl6 --name=Patch
name (Patch) rank (NA) serial number (1)
```

As it turns out, the built-in `UInt` (https://docs.perl6.org/type/UInt\) type will work for ensuring positive, so maybe we would be better off ensuring that the serial number is of the correct length:

```
$ cat -n main9.pl6
     1    #!/usr/bin/env perl6
     2
     3    sub MAIN (
     4        Str :$name!,
     5        Str :$rank='NA',
     6        UInt :$serial-num where *.Str.chars == 7 = 1111111
     7    ) {
     8        put "name ($name) rank ($rank) serial number ($ser
ial-num)";
     9    }
$ ./main9.pl6 --name=Patch --serial-num=12345
Usage:
  ./main9.pl6 --name=<Str> [--rank=<Str>] [--serial-num=<Int>]
$ ./main9.pl6 --name=Patch --serial-num=1234567
name (Patch) rank (NA) serial number (1234567)
```

Unfortunately the usage doesn't include information about the length, so we can override the default `USAGE` (https://docs.perl6.org/language/functions\#index-entry-USAGE\):

```
$ cat -n main10.pl6
     1    #!/usr/bin/env perl6
     2
     3    sub MAIN (
     4        Str :$name!,
     5        Str :$rank='NA',
     6        UInt :$serial-num where *.Str.chars == 7 = 1111111
     7    ) {
     8        put "name ($name) rank ($rank) serial number ($ser
ial-num)";
     9    }
    10
    11    sub USAGE {
    12        printf "Usage:\n  %s --name=<Str> [--rank=<Str>] [
--serial-num=<Int>]\n",
    13            $*PROGRAM.basename;
    14
    15        put "Note: --serial-num must be 7 digits in length
."
    16    }
$ ./main10.pl6
Usage:
  main10.pl6 --name=<Str> [--rank=<Str>] [--serial-num=<Int>]
Note: --serial-num must be 7 digits in length.
```

I can move the `--serial-num` code into a `subset` (https://docs.perl6.org/language/typesystem\#index-entry-subset-subset\) and further constrain `--rank` to a list of acceptable values:

```
$ cat -n main11.pl6
     1    #!/usr/bin/env perl6
     2
     3    subset SerialNum of UInt where *.Str.chars == 7;
     4    my @ranks = <NA Recruit Private PSC PFC Specialist>;
     5    subset Rank of Str where * eq any(@ranks);
     6
     7    sub MAIN (
     8        Str :$name!,
     9        Rank :$rank='NA',
    10        SerialNum :$serial-num=1111111
    11    ) {
    12        put "name ($name) rank ($rank) serial number ($ser
ial-num)";
    13    }
    14
    15    sub USAGE {
    16        printf "Usage:\n  %s --name=<Str> [--rank=<Str>] [
--serial-num=<Int>]\n",
    17            $*PROGRAM.basename;
    18
    19        put "--rank must be one of: {@ranks.join(', ')}";
    20        put "--serial-num must be 7 digits in length."
    21    }
$ ./main11.pl6 --name=Patch --serial-num=1234567 --rank=XXX
Usage:
  main11.pl6 --name=<Str> [--rank=<Str>] [--serial-num=<Int>]
--rank must be one of: NA, Recruit, Private, PSC, PFC, Specialis
t
--serial-num must be 7 digits in length.
[saguaro@~/work/perl6/main]$ ./main11.pl6 --name=Patch --serial-
num=1234567 --rank=PFC
name (Patch) rank (PFC) serial number (1234567)
```

We can declare the `MAIN` with the keyword" `multi` instead of `sub` to indicate multiple signatures that the program can respond to. In this example, we're writing a script that will print either the reverse complement or the RNA translation of a string of DNA. This is a common pattern in command-line programs, for example,

the first argument to "git" is a command like "checkout" or "branch." We'll match `MAIN` to a literal string (either "revcom" or "rna") and use a custom `subset` to define what a string of DNA looks like:

```
$ cat -n main12.pl6
     1    #!/usr/bin/env perl6
     2
     3    subset DNA of Str where /^ :i <[ACTGN]>+ $/;
     4
     5    multi MAIN ('revcom', DNA $dna) {
     6        put $dna.trans(<A C G T a c g t> => <T G C A t g c
 a>).flip;
     7    }
     8
     9    multi MAIN ('rna', DNA $dna) {
    10        put $dna.subst('T', 'U', :g);
    11    }
$ ./main12.pl6
Usage:
  ./main12.pl6 revcom <dna>
  ./main12.pl6 rna <dna>
[saguaro@~/work/metagenomics-book/perl6/main]$ ./main12.pl6 revc
om TTACG
CGTAA
[saguaro@~/work/metagenomics-book/perl6/main]$ ./main12.pl6 rna
TTACG
UUACG
```

If the user provides arguments that don't match any of our signatures, they get a `USAGE`:

```
$ ./main12.pl6 foo TTACG
Usage:
  ./main12.pl6 revcom <dna>
  ./main12.pl6 rna <dna>
$ ./main12.pl6 revcom foo
Usage:
  ./main12.pl6 revcom <dna>
  ./main12.pl6 rna <dna>
```

Add It Up

Here are a few more examples of using types in `MAIN` signature and discussion of types a bit further with a program that will add two numbers:

```
$ cat -n adder1.pl6
     1     #!/usr/bin/env perl6
     2
     3     sub MAIN (Int $a!, Int $b!) { put $a + $b }
     4
     5     sub USAGE {
     6         printf "  %s <Int> <Int>\n", $*SPEC.basename($*PR
OGRAM-NAME);
     7     }
$ ./adder1.pl6
  adder1.pl6 <Int> <Int>
$ ./adder1.pl6 foo bar
  adder1.pl6 <Int> <Int>
$ ./adder1.pl6 2 4
6
$ ./adder1.pl6 2 4.8
  adder1.pl6 <Int> <Int>
```

At line 3, I'm declaring two required integer variables called `$a` and `$b`. If the user provides any arguments that don't match that signature, then we've seen how Perl will generate a decent usage statement. Here I've thrown in my own `USAGE` method to indicate specifically that I want integer values.

We have a problem in that we'd like to be able to add anything that looks like a number, and `Int` won't allow the argument "4.8." We can move up the type hierarchy to `Numeric` to be more generic:

```
$ cat -n adder2.pl6
     1  #!/usr/bin/env perl6
     2
     3  sub MAIN (Numeric $a!, Numeric $b!) { put $a + $b }
     4
     5  sub USAGE {
     6      note sprintf "  %s <Numeric> <Numeric>", $*SPEC.b
asename($*PROGRAM-NAME);
     7  }
$ ./adder2.pl6 2>err
$ cat err
  adder2.pl6 <Numeric> <Numeric>
$ ./adder2.pl6 2.71828 3.14159
5.85987
$ ./adder2.pl6 2.71828 5.4e+3
5402.71828
```

Something else I added was the `note` in `USAGE` so that the usage statement is printed to STDERR (standard error). You can read about other useful I/O (input/output) methods at https://docs.perl6.org/type/IO.

What I like about creating descriptive signatures for `MAIN` is that it helps us define what inputs we require and communicates the requirements efficiently to the user. You are encouraged to always use a `MAIN` entry point when possible.

But wait, there's more!

Here's a magical little secret: Everything that applies to `MAIN` also applies to every subroutine in Perl! I'll declare a `multi foo` subroutine that will work just like the above `MAIN` where I match on a literal string and some other sort of argument like an `Int` or a `Str`:

```
$ cat -n sub.pl6
     1    #!/usr/bin/env perl6
     2
     3    foo('bar', 'baz');
     4    foo('bar', 123);
     5    foo('quux', 3.14);
     6
     7    multi foo ('bar', Str $str) {
     8        put "Bar got a string '$str'";
     9    }
    10
    11    multi foo ('bar', Int $n) {
    12        put "Bar got an integer '$n'";
    13    }
    14
    15    multi foo ('quux', Any $x) {
    16        put "Bar got an argument '$x'";
    17    }
$ ./sub.pl6
Bar got a string 'baz'
Bar got an integer '123'
Bar got an argument '3.14'
```

The combination of pattern matching of Perl subroutines and its type system give you a simple yet powerful way to write explicit and safe code.

File handling

Most data in bioinformatics is exchanged an some sort of text format. Reading and writing from text is bread-and-butter work, so let's do some.

File tests

Given some input, how do you know it's a file, that you can read it, etc.? In our bash section, we came across Unix file tests with this where we wanted to check if `$OUT_DIR` was a directory using `-d` :

```
if [[ ! -d $OUT_DIR ]]; then
  mkdir -p $OUT_DIR
fi
```

Perl has similar tests, and to use them you simply tell Perl to treat the variable as an IO object and then execute the method for the test. You can find all the tests https://docs.perl6.org/type/IO$COLON$COLONPath#File_test_operators. So the above in Perl would look like this:

```
mkdir $out-dir unless $out-dir.IO.d;
```

Say it loud...

Here's a trivial example to UPPERCASE the contents of a file:

```
$ cat -n upper1.pl6
     1    #!/usr/bin/env perl6
     2
     3    sub MAIN (Str $file!) {
     4        die "Not a file ($file)" unless $file.IO.f;
     5
     6        for $file.IO.lines -> $line {
     7            put $line.uc;
     8        }
     9    }
$ ./upper1.pl6 foo
Not a file (foo)
  in sub MAIN at ./upper1.pl6 line 6
  in block <unit> at ./upper1.pl6 line 5
$ ./upper1.pl6 jabberwocky.txt | head -6
JABBERWOCKY -- LEWIS CARROLL

'TWAS BRILLIG, AND THE SLITHY TOVES
  DID GYRE AND GIMBLE IN THE WABE:
ALL MIMSY WERE THE BOROGOVES,
  AND THE MOME RATHS OUTGRABE.
```

At line 4, we should first check that the input was actually a file. If it is not, then we `die` with an error message which will halt execution (see above). At line 6, we start a `for` loop using the `$file.IO.lines` (https://docs.perl6.org/type/IO$COLON$COLONHandle#method_lines) method to open the file and read in each of the lines into the `$line` variable. Lastly we `put` the uppercase (`uc`) version of the line.

We can borrow a technique from functional programming with `map` (https://docs.perl6.org/routine/map) to make this shorter:

```
$ cat -n upper2.pl6
     1     #!/usr/bin/env perl6
     2
     3     sub MAIN (Str $file!) {
     4         die "Not a file ($file)" unless $file.IO.e;
     5
     6         put $file.IO.lines.map(*.uc).join("\n");
     7     }
```

This technique pushes the lines of the file directly into a the `uc` transformation and then into the join to make a new string.

> Golfing: Sometimes you may here about programmers (often Perl or c hackers) who like to "golf" their programs. It's an attempt to create a program using the fewest keystrokes as possible, similar to the strategy in golf where the player tries to strike the ball as few times as possible to put it into the cup. It's not a necessarily admirable quality to make one's code as terse as possible, but there is some truth to the notion that more code means more bugs. There are incredibly powerful ideas built into every language, and learning how they can save you from writing code is worth the effort of writing fewer bugs. If you understand `map`, then you probably also understand why `for` loops and mutable variables are dangerous. If you don't, then keep studying.

Reading two files in parallel

I came across a StackOverflow question where the user had sequence scores in Phred format in one file and the sequences in another file like so:

```
$ cat phred.txt
"#$%
&'()
$ cat seq.txt
ABCD
EFGH
```

The user needed to merge the files together, line-by-line, and assign "Sequence_#" headers like so:

```
Sequence_1
1    2    3    4
A    B    C    D
Sequence_2
5    6    7    8
E    F    G    H
```

In the following script, I use the `Z` zip operator (https://docs.perl6.org/routine/Z) to combine three lists together -- an increasing integer value (sequence number), a line from the Phred file, and a line from the sequence file. To understand that, look at zipping an infinite list of integers `1..*` with the first four letters of the alphabet `"a".."d"` and the lines from the local dictionary file. Remember that zipping stops on the shortest list:

```
> 1..* Z "a".."d" Z '/usr/share/dict/words'.IO.lines
((1 a A) (2 b a) (3 c aa) (4 d aal))
```

Here it is in action:

```
$ cat -n parallel.pl6
     1    #!/usr/bin/env perl6
     2
     3    subset File of Str where *.IO.f;
     4
     5    sub MAIN (File :$phred='phred.txt', File :$seq='seq.tx
t') {
     6        my $phred-fh = open $phred;
     7        my $seq-fh   = open $seq;
     8        my %xlate    = map { chr($_ + 33) => $_ }, 1..8;
     9
    10        for 1..* Z $phred-fh.IO.lines Z $seq-fh.IO.lines -
> ($i, $score, $bases) {
    11            put join "\n",
    12                "Sequence_$i",
    13                (map { %xlate{$_} }, $score.comb).join("\t
"),
    14                $bases.comb.join("\t");
    15        }
    16    }
```

Movie file reader

It seems to me that, even in movies set in the future, filmmakers still like to have computers spit out messages one-character-at-a-time as if they were arriving like telegrams. If you would like to read a file like this, I present the "movie file reader." First, an explicit, long-hand version:

```
$ cat -n reader1.pl6
     1    #!/usr/bin/env perl6
     2
     3    sub MAIN (Str $file) {
     4        for $file.IO.lines(:chomp(False)) -> $line {
     5            for $line.comb -> $letter {
     6                print $letter;
     7                my $pause = do given $letter {
     8                    when /<[.!?]>/ { .50 }
     9                    when /<[,;]>/  { .20 }
    10                    default        { .05 }
    11                }
    12                sleep $pause;
    13            }
    14        }
    15    }
```

So I'm reading the given $file line-by-line, telling Perl not to "chomp" each line (remove the newline for whatever value constitutes that, which, BTW, you can set with `nl-in`). Another way to write that is `:!chomp`. I `print` the letter, not `put` because I don't want a newline after it. Then I need to pause with the `sleep` because computers move way faster than the human eye. To figure out how long to sleep, I want to inspect the character for punctuation that either ends a sentence or introduces a pause. I use `<[]>` to create a character class that includes a period, exclamation point, and question mark or one that includes a comma or semi-colon. The `do given` lets me return the value of the `given` statement, effectively turning it into a `given` operator.

I always bounce my ideas off `#perl6` on IRC, and Zoffix suggested this much shorter version:

```
$ cat -n reader2.pl6
     1    #!/usr/bin/env perl6
     2
     3    sub MAIN (Str $file) {
     4        for $file.IO.comb {
     5            .print;
     6            sleep  /<[.!?]>/ ?? .30
     7                !! /<[,;]>/  ?? .10
     8                !!              .05;
     9        }
    10    }
```

Here we're reading the file character-by-character into the default `$_` topic variable upon which we can call the `.print` method. Then we `sleep` (perchance to dream) using a stacked ternary operator to find how long. Much shorter, but more cryptic to the uninitiated. I like both versions because they both work and they allow the programmer varying levels of expressiveness and efficiency.

Reading compressed files

Often it makes sense to keep data in a compressed format to save disk space. If you need to read a gzipped file, here's a way to use a pipe (Proc https://docs.perl6.org/language/ipc#index-entry-Proc_object-proc):

```
$ cat -n read-gzip.pl6
     1  #!/usr/bin/env perl6
     2
     3  sub MAIN (Str $file! where *.IO.f) {
     4      my $fh = $file ~~ /\.gz$/
     5             ?? run(«gunzip -c $file |», :out).out
     6             !! open $file;
     7
     8      for $fh.lines -> $line {
     9          put $line;
    10      }
    11  }
```

CSV & tab-delimited files

One of the most common text file formats is records separated by newlines ("\n" or "\r\n" on Windows) where the fields of the records are delimited by commas ("comma-separated values" or "CSV") or tabs ("tab-delimited"). Here is an example that will extract a field from a delimited text file showing "[t]he leading causes of death by sex and ethnicity in New York City in since 2007" (https://data.cityofnewyork.us/api/views/jb7j-dtam/rows.csv?accessType=DOWNLOAD).

```
$ cat -n parser1.pl6
     1    #!/usr/bin/env perl6
     2
     3    sub MAIN (Str $file!, Int :$n=0, Str :$sep=',') {
     4        die "Not a file ($file)" unless $file.IO.f;
     5
     6        for $file.IO.lines -> $line {
     7            my @fields = $line.split($sep);
     8            put @fields[$n];
     9        }
    10    }
$ ./parser1.pl6 causes.csv | head -3Year
2010
2010
```

The problem with relying on the position of data is that someone may change the number of columns, and that will break your code. It's better, when possible, to use the *name* of the fields, and this particular file does have named fields:

```
$ head -1 causes.csv
Year,Ethnicity,Sex,Cause of Death,Count,Percent
```

Here's a version that merges the headers on the first line with each data record to create a hash so that we can extract a named field:

```
$ cat -n parser2.pl6
     1    #!/usr/bin/env perl6
     2
     3    sub MAIN (Str $file!, Str :$field!, Str :$sep=',', Int
 :$limit=0) {
     4        die "Not a file ($file)" unless $file.IO.f;
     5
     6        my $fh     = open $file;
     7        my @fields = $fh.get.split($sep);
     8
     9        unless one(@fields) eq $field {
    10            die "No $field in $file";
    11        }
    12
    13        my $i = 0;
    14        for $fh.lines -> $line {
    15            $i++;
    16            my @values = $line.split($sep);
    17            my %record;
    18            for 0..^@fields.elems -> $i {
    19                my $key = @fields[$i];
    20                my $val = @values[$i];
    21                %record{ $key } = $val;
    22            }
    23
    24            put %record{ $field } // "";
    25
    26            last if $limit > 0 && $i == $limit;
    27        }
    28    }
$ ./parser2.pl6 --field='Year' --limit=3 causes.csv
2010
2010
2010
```

In this version, I `open` the file (line 6) to get a "filehandle" (often abbreviated "fh") which is a way to get access to the contents of the file. I did this so that I could call the `get` method (https://docs.perl6.org/type/IO$COLON$COLONHandle#method_get) to retrieve

just the first line which I expect to have the field names. That returns a string which I then call `split` using the `$sep` (separator) argument (which defaults to a comma).

Lines 15-20 are probably a little confusing, so let's break it down. I want to create a key-value structure that associates the headers in the first line to the fields in each record, so I declare `my %record` at line 15. Then I want to step through the *numbered positions* of the `@fields` so that I can pair them up with their `@values`. List-type structures in Perl start numbering at 0, so I start my `for` loop there. Then I use the `..^` operator to construct a Range (https://docs.perl6.org/type/Range) that goes up to *but not including* the number of elements in `@fields`. It's the hat `^` that says to the Range constructor to stop one less than the thing it's next to. You can also put it on the left or both (or neither). We have to stop before the *number* of elements in `@fields` because of the 0-based numbering -- that is, if there are 3 `@fields`, then they are in positions 0, 1, and 2. So that explains the loading of the `$i` variable (NB: "i" is a very common "integer" variable. If a second value is needed, then it's common to move on to "j," "k," etc.)

To explore this, pretend I'm making a hash like this:

```
> my %record;
{}
> %record<name> = 'Gene'
Gene
> %record
{name => Gene}
> %record<rank> = "Staff Sargeant"
Staff Sargeant
> %record
{name => Gene, rank => Staff Sargeant}
> %record<serial_number> = "1656401"
1656401
> %record
{name => Gene, rank => Staff Sargeant, serial_number => 1656401}
```

Instead of manually naming each key and value, I can use two arrays for ```:

```
# position =   0    1               2
my @keys   =  <name rank            serial_number>;
my @values = <<Gene "Staff Sargeant" 1656401>>;
> for 0..^@keys.elems -> $i { say "$i: @fields[$i]" }
0: name
1: rank
2: serial_number
> my %data
{}
> for 0..^@keys.elems -> $i { %data{ @keys[$i] } = @values[$i] }
(Any)
> %data
{name => Gene, rank => Staff Sargeant, serial_number => 1656401}
```

The result is the same whether we set the fields individually or all-at-once like so:

```
> my %record = name => "Gene", rank => "Staff Sargeant", serial_
number => "1656401";
> %record
{name => Gene, rank => Staff Sargeant, serial_number => 1656401}
```

At line 24, I handle the "--limit" option I added so that I could stop processing on the first record. Here I'm using `last` to exit the loop if I have a positive value for `$limit` (the default is 0) and the number of records I've seen.

At line 25, I've introduced the `say` function so you can look at the data that was collected. The function `dd` (data dump) will also show you the structure:

```
Array @data = [(:Year("2010"), :Ethnicity("NON-HISPANIC BLACK"),
 :Sex("MALE"), "Cause of Death" => "HUMAN IMMUNODEFICIENCY VIRUS
 DISEASE", :Count("297"), :Percent("5")).Seq, (:Year("2010"), :E
thnicity("NON-HISPANIC BLACK"), :Sex("MALE"), "Cause of Death" =
> "INFLUENZA AND PNEUMONIA", :Count("201"), :Percent("3")).Seq]
```

These differ from `print` and `put` which both show:

```
Year       2010 Ethnicity     NON-HISPANIC BLACK Sex     MALE Ca
use of Death        HUMAN IMMUNODEFICIENCY VIRUS DISEASE Count
       297 Percent        5 Year     2010 Ethnicity    NON-HISPA
NIC BLACK Sex      MALE Cause of Death        INFLUENZA AND PNEU
MONIA Count     201 Percent         3
```

This next version will use a powerful idea from functional programming called a "zip" (`Z` https://docs.perl6.org/routine/Z) to pair up the `@fields` and `@values` . Like an actual zipper, this takes one element from each list and creates a new list of lists:

```
> @fields Z @values
((name Gene) (rank "Staff) (serial_number Sargeant"))
```

However, what we want is a new list of Pairs which are created with the `=>` operator (https://docs.perl6.org/routine/$EQUALS_SIGN$GREATER-THAN_SIGN), so we just mush the two operators together to create a new zip-and-pair operator `Z=>` :

```
> my %record = @fields Z=> @values
{name => Gene, rank => "Staff, serial_number => Sargeant"}
```

Zips can work with lists of different values. Here is how I can pair the list ("a", "b", "c") with a lazy, infinite list of integers:

```
> <a b c> Z 1..*
((a 1) (b 2) (c 3))
```

As you will see this saves many lines of code. I've also thrown in the ability to skip lines that look like "comments" based on how the user defines a comment:

```
$ cat -n parser3.pl6
     1     #!/usr/bin/env perl6
     2
     3     use v6;
     4
     5     sub MAIN (Str $file!, Str :$sep=',', Int :$limit=0, S
tr :$comment) {
     6         die "Not a file ($file)" unless $file.IO.f;
     7
     8         # copy to make mutable
     9         (my $delim = $sep) ~~ s/\\t/\t/;
    10         my $fh     = open $file;
    11         my @fields = $fh.get.split($delim);
    12
    13         my @data;
    14         for $fh.lines -> $line {
    15             next if $comment.defined &&
    16                     $line.substr(0, $comment.chars) eq $c
omment;
    17             @data.push(@fields Z=> $line.split($delim));
    18             last if $limit > 0 && @data.elems > $limit;
    19         }
    20
    21         say @data;
    22     }
```

Line 9 is pretty cryptic and needs to be explained. First off, when parsing tab-delimited files, it's difficult to pass in a non-printable character on the command line. Passing an *actual* tab (by hitting the "Tab" key) doesn't work, and most people wouldn't know to enter:

```
./parser3.pl6 --sep=$(\t) ...
```

So we assume that the user will enter this:

```
./parser3.pl6 --sep=\t ...
```

Which will then come in to us looking like "\t". We need to turn that into just "\t" which is the escape character representing the "Tab." We can do that with the `s//` substitution operator (https://docs.perl6.org/language/operators#Substitution_Operators), but not on the `$sep` variable because it is *read-only.* So we copy it into `$delim` and then immediately bind (with `~~`) to the substitution command. Et voila, we have a "Tab" we can use.

Line 15 is the logic for skipping comments. In the way that `last` will exit a loop, `next` will immediately jump to the next iteration of the loop. Line 16 use a `substr` (substring) operation to look at the first part of the `$line` to see if it matches the `$comment` string. I'm taking into account that the `$comment` may be more than one character (e.g., "//"). Line 17 doesn't bother to create a `%record` but just pushes the result of the zip `Z=>` .

And with that, we have written a fairly decent delimited text parser! This sort of thing in very standard, and so it would actually behoove us to use an existing module such as "CSV::Parser" or "Text::CSV" (http://modules.perl6.org/#q=csv). They will both accomplish the task, so it's just a matter of which one appeals to you more.

You can install a module using the "panda" tool:

```
$ panda install CSV::Parser
==> Fetching CSV::Parser
==> Building CSV::Parser
==> Testing CSV::Parser
t/01_multiline_csv.t ... ok
t/02_escaped_csv.t ..... ok
t/03_delimiters_csv.t .. ok
t/04_binary_csv.t ...... ok
All tests successful.
Files=4, Tests=4,  2 wallclock secs ( 0.02 usr  0.01 sys +  1.75
 cusr  0.22 csys =  2.00 CPU)
Result: PASS
==> Installing CSV::Parser
==> Successfully installed CSV::Parser
```

Here is how the code would look:

```
$ cat -n parser4.pl6
     1    #!/usr/bin/env perl6
     2
     3    use v6;
     4    use CSV::Parser;
     5
     6    sub MAIN (Str $file!, Str :$sep=',', Int :$limit=0, S
tr :$comment) {
     7        die "Not a file ($file)" unless $file.IO.f;
     8        my $fh     = open $file;
     9        my $parser = CSV::Parser.new(
    10                       file_handle => $fh, contains_header_
row => True );
    11
    12        my @data;
    13        until $fh.eof {
    14            @data.push(%($parser.get_line))
    15        }
    16
    17        say @data;
    18    }
```

This module does not handle comment lines. If you needed that, you could either go back to your own module or you could look at the author's code and add to it. Most author's will have a Github repository that you can fork and then submit a "pull request" (PR). Also, you might just write the author and ask very nicely to add the feature you want.

GFF

The "general feature format" (https://en.wikipedia.org/wiki/General_feature_format) is a tab-delimited specification for describing genomic features. Version 3 of GFF specifies nine fields:

1. sequence
2. source
3. feature
4. start
5. end
6. score
7. strand
8. frame
9. attributes

Let's download a GFF file for yeast:

```
$ wget http://downloads.yeastgenome.org/curation/chromosomal_fea
ture/saccharomyces_cerevisiae.gff
$ head -20 saccharomyces_cerevisiae.gff
##gff-version 3
#date Sun Aug 28 19:50:04 2016
#
# Saccharomyces cerevisiae S288C genome (version=R64-2-1)
#
# Features from the 16 nuclear chromosomes labeled chrI to chrXV
I,
# plus the mitochondrial genome labeled chrmt.
#
# Created by Saccharomyces Genome Database (http://www.yeastgeno
me.org/)
#
# Weekly updates of this file are available for download from:
# http://downloads.yeastgenome.org/curation/chromosomal_feature/
saccharomyces_cerevisiae.gff
#
# Please send comments and suggestions to sgd-helpdesk@lists.sta
nford.edu
#
# SGD is funded as a National Human Genome Research Institute Bi
omedical Informatics Resource from
# the U. S. National Institutes of Health to Stanford University
.
#
chrI       SGD        chromosome         1          230218     .
          .          .          ID=chrI;dbxref=NCBI:;Name=chrI
chrI       SGD        telomere           1          801        .
          -          .          ID=TEL01L;Name=TEL01L;Note=Telom
eric%20region%20on%20the%20left%20arm%20of%20Chromosome%20I%3B%2
0composed%20of%20an%20X%20element%20core%20sequence%2C%20X%20ele
ment%20combinatorial%20repeats%2C%20and%20a%20short%20terminal%2
0stretch%20of%20telomeric%20repeats;display=Telomeric%20region%2
0on%20the%20left%20arm%20of%20Chromosome%20I;dbxref=SGD:S0000288
62
```

It's obvious that the lines beginning with "#" are comments and metadata. After that, the data begins. If we wanted to find out how many "chromosome" features exist and their names:

```
$ awk -F"\t" '$3 == "chromosome"' saccharomyces_cerevisiae.gff |
 wc -l
      17
$ awk -F"\t" '$3 == "chromosome" {print $1}' saccharomyces_cerev
isiae.gff | cat -n
     1     chrI
     2     chrII
     3     chrIII
     4     chrIV
     5     chrV
     6     chrVI
     7     chrVII
     8     chrVIII
     9     chrIX
    10     chrX
    11     chrXI
    12     chrXII
    13     chrXIII
    14     chrXIV
    15     chrXV
    16     chrXVI
    17     chrmt
```

Let's look at the top 10 most numerous features in the yeast genome:

```
$ awk -F"\t" '{print $3}' saccharomyces_cerevisiae.gff | sort |
uniq -c | sort -rn | head
152010
7058 CDS
6600 mRNA
6600 gene
 484 noncoding_exon
 383 long_terminal_repeat
 377 intron
 352 ARS
 299 tRNA_gene
 196 ARS_consensus_sequence
```

There's only so far we can get with `awk`, though, before we need to get into Perl. For instance, let's find two overlapping genes:

```
$ cat -n gff1.pl6
     1     #!/usr/bin/env perl6
     2
     3     # http://downloads.yeastgenome.org/curation/chromosom
al_feature/saccharomyces_cerevisiae.gff
     4
     5     use URI::Encode;
     6
     7     sub MAIN (Str $gff! where *.IO.f) {
     8         my $fh = open $gff;
     9
    10         my %loc;
    11         for $fh.lines -> $line {
    12             next if $line ~~ / ^ '#' /; # comment line
    13             my ($sequence, $source, $feature, $start, $en
d, $score,
    14                 $strand, $frame, $attributes) = $line.spl
it(/\t/);
    15
    16             next unless ($feature.defined && $feature eq
'gene')
    17                     && ($strand.defined  && $strand ~~ /
 ^ <[+-]> $/);
```

```
    18
    19              my %attr = $attributes.split(';').map(&uri_de
code)
    20                          .map(*.split('=')).map({ $^a.[0].l
c => $^a.[1] });
    21              my $name = %attr<name> || 'NA';
    22
    23              %loc{ $sequence }{ $strand }.push(
    24                  ($name, [+$start .. +$end]) # force numer
ic, important!
    25              );
    26          }
    27          $fh.close;
    28
    29          for %loc.keys -> $chr {
    30              my @forward-genes = %loc{ $chr }{'+'}.list;
    31              my @reverse-genes = %loc{ $chr }{'-'}.list;
    32              note sprintf "chr (%s) has %s forward genes,
%s reverse genes\n",
    33                  $chr, @forward-genes.elems, @reverse-gene
s.elems;
    34
    35              for @forward-genes -> ($gene1, $pos1) {
    36                  for @reverse-genes -> ($gene2, $pos2) {
    37                      if so $pos1 (&) $pos2 {
    38                          printf "%s [%s] (%s) => %s [%s] (
%s)\n",
    39                              $gene1, '+', $pos1[0,*-1].joi
n('..'),
    40                              $gene2, '-', $pos2[0,*-1].joi
n('..');
    41                      }
    42                  }
    43              }
    44          }
    45      }
```

Feature annotation

Imagine you've run uproc to classify your gene calls with annotations from KEGG and Pfam. The output from uproc is a comma-separated file like so:

```
$ head -1 uproc-out.pfam
1,0|contig:9|start:0|stop:630|direction:r|rev_compd:True|length:
630,630,1,1,210,PF01119,1.318
$ head -1 uproc-out.kegg
1,0|contig:9|start:0|stop:630|direction:r|rev_compd:True|length:
630,630,1,1,210,K10858,1.494
```

If we look at `uproc-dna --help`, we can find what the fields mean:

```
    Columns to be printed when -p is used. By default, all of th
em are printed
    in the order as below:
        n: sequence number (starting from 1)
        h: sequence header up to the first whitespace
        l: sequence length (this is a lowercase L)
        F: ORF frame number (1-6)
        I: ORF index in the DNA sequence (starting from 1)
        L: ORF length
        f: predicted protein family
        s: classification score
```

So we need to use the second-to-last field (e.g., "PF01119" or "K10858") and find the ID in the KEGG and Pfam files which have just two fields (id, description) which are delimited by tabs:

```
$ grep PF01119 pfam_to_domain
PF01119    DNA_mis_repair
$ grep K10858 kegg_to_desc
K10858    PMS2; DNA mismatch repair protein PMS2
```

So the user will need to provide us two pairs of files: the Pfam and KEGG files and the Pfam and KEGG annotations from uproc, and our script will merge all this information into a new file which we can default to just "out."

```
$ cat -n annotate-uproc.pl6
     1    #!/usr/bin/env perl6
     2
     3    sub MAIN (
     4        Str :$kegg-out,
     5        Str :$pfam-out,
     6        Str :$kegg-desc,
     7        Str :$pfam-desc,
     8        Str :$out="out"
     9    ) {
    10        die "Must have --kegg-out or --pfam-out" unless $k
egg-out || $pfam-out;
    11
    12        my $out-fh = open $out, :w;
    13        $out-fh.put(<gene_callers_id source accession func
tion e_value>.join("\t"));
    14        process('kegg', $kegg-out, $kegg-desc, $out-fh);
    15        process('pfam', $pfam-out, $pfam-desc, $out-fh);
    16        put "Done, see output '$out'";
    17    }
    18
    19    sub process ($source, $uproc-out, $desc-file, $fh) {
    20        return unless $uproc-out && $desc-file;
    21        my %id_to_desc;
    22        for $desc-file.IO.lines -> $line {
    23            my ($id, $desc) = $line.split(/\t/);
    24            %id_to_desc{ $id } = $desc;
    25        }
    26
    27        for $uproc-out.IO.lines -> $line {
    28            my @fields = $line.split(',');
    29            my $gene   = @fields[1].subst(/'|' .*/, '');
    30            my $id     = @fields[6];
    31            my $score  = @fields[7];
    32            my $desc   = %id_to_desc{ $id } || "NONE";
    33            $fh.put(join("\t", $gene, $source, $id, $desc,
 $score));
    34        }
    35    }
```

Handling a comma- or tab-separated file is as simple as splitting each line on the proper character as shown on lines 23 and 28. In lines 21-25, we're splitting each line of the Pfam or KEGG files into the `$id` and `$desc` and then storing these values into a hash called `%id_to_desc` that we can then use on line 32 to add the description from the file or "NONE" if none exists. On line 29, we want to removed everything after the first "|" character with the `subst` (substitute) operation. The other lines are just picking other desired fields from the file, and then we `put` the data into our output file handle `$fh`.

Tetranucleotide profiling

Larry has often claimed that Perl is designed to support users from beginners to advanced. We can start with simple syntax and ideas and refine our programs as we learn about more powerful techniques. For this section, I'm going to walk through an evolution of the "DNA" problem from the Rosalind.info website (http://rosalind.info/problems/dna). I'll start with basic imperative programming methods and move towards shorter, more declarative and functional code.

Here is a first version:

```
$ cat -n dna1.pl6
     1     #!/usr/bin/env perl6
     2
     3     sub MAIN (Str $dna!) {
     4         my ($num-A, $num-C, $num-G, $num-T) = 0, 0, 0, 0;
     5
     6         for $dna.comb -> $letter {
     7             if    $letter eq 'a' || $letter eq 'A' { $num
-A++ }
     8             elsif $letter eq 'c' || $letter eq 'C' { $num
-C++ }
     9             elsif $letter eq 'g' || $letter eq 'G' { $num
-G++ }
    10             elsif $letter eq 't' || $letter eq 'T' { $num
-T++ }
    11         }
    12
    13         put "$num-A $num-C $num-G $num-T";
    14     }
$ ./dna1.pl6 AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAA
AAGAGTGTCTGATAGCAGC
20 12 17 21
$ ./dna1.pl6 foobar
1 0 0 0
```

Not a bad start. Our program expects a String of DNA on the command line (line 3). We declare four variables at line 4 to hold the counts for A, C, G, and T. Line 6 sets up a `for` loop (https://docs.perl6.org/syntax/for) that calls the `comb` method (https://docs.perl6.org/type/Str#routine_comb) on the `$dna` so that we can inspect each `$letter` individually. Lines 7-10 compare each letter's upper- and lowercase version to decide which counter to increment with `++` (https://docs.perl6.org/routine/$PLUS_SIGN$PLUS_SIGN). *NB: Lowercase is often used to "softmask" highly repetitive regions of DNA like in plant genomes.* At line 13, we print out the results as described on the website.

Here's a version that makes some improvements:

```
$ cat -n dna2.pl6
     1    #!/usr/bin/env perl6
     2
     3    sub MAIN (Str $dna!) {
     4        my ($num-A, $num-C, $num-G, $num-T) = 0, 0, 0, 0;
     5
     6        for $dna.comb -> $letter {
     7            if    $letter eq 'a' | 'A' { $num-A++ }
     8            elsif $letter eq 'c' | 'C' { $num-C++ }
     9            elsif $letter eq 'g' | 'G' { $num-G++ }
    10            elsif $letter eq 't' | 'T' { $num-T++ }
    11        }
    12
    13        put "$num-A $num-C $num-G $num-T";
    14    }
```

Rather than repeat `$letter eq` for both the upper- and lowercase letters, I'm using a Junction (https://docs.perl6.org/type/Junction) to evaluate either option at the same time. Here's another idea:

```
$ cat -n dna3.pl6
     1    #!/usr/bin/env perl6
     2
     3    sub MAIN (Str $dna!) {
     4        my ($num-A, $num-C, $num-G, $num-T) = 0, 0, 0, 0;
     5
     6        for $dna.lc.comb -> $letter {
     7            if    $letter eq 'a' { $num-A++ }
     8            elsif $letter eq 'c' { $num-C++ }
     9            elsif $letter eq 'g' { $num-G++ }
    10            elsif $letter eq 't' { $num-T++ }
    11        }
    12
    13        put join ' ', $num-A, $num-C, $num-G, $num-T;
    14    }
```

I wanted to get rid of checking both cases of the letters, so at line 6 I first `lc` (lowercase) `$dna` and then get the letters. Notice that you can chain methods.

Most languages have a "switch" or "case" (like for our options checking in bash) statement. Perl's is called `given` (https://docs.perl6.org/language/control#given), it's like a giant if/elsif statement, only much, much better:

```
$ cat -n dna4.pl6
     1    #!/usr/bin/env perl6
     2
     3    sub MAIN (Str $dna!) {
     4        my ($num-A, $num-C, $num-G, $num-T) = 0, 0, 0, 0;
     5
     6        for $dna.lc.comb -> $letter {
     7            given $letter {
     8                when 'a' { $num-A++ }
     9                when 'c' { $num-C++ }
    10                when 'g' { $num-G++ }
    11                when 't' { $num-T++ }
    12            }
    13        }
    14
    15        put ($num-A, $num-C, $num-G, $num-T).join(' ');
    16    }
```

Notice the change at line 15 where I make a temporary (anonymous) list from the four counters and then call the `.join` method on the list.

What's fun is that `for` can act like `given` and use smart matching on `$_` AKA "the topic" (or "thing" or "it"). Here's a shorter version:

```
$ cat -n dna5.pl6
     1    #!/usr/bin/env perl6
     2
     3    sub MAIN (Str $dna!) {
     4        my ($num-A, $num-C, $num-G, $num-T) = 0, 0, 0, 0;
     5
     6        for $dna.lc.comb {
     7            when 'a' { $num-A++ }
     8            when 'c' { $num-C++ }
     9            when 'g' { $num-G++ }
    10            when 't' { $num-T++ }
    11        }
    12
    13        put ($num-A, $num-C, $num-G, $num-T).join(' ');
    14    }
```

Now I'm going to show you a version where we use a hash instead of a bunch of scalars:

```
$ cat -n dna5.pl6
     1     #!/usr/bin/env perl6
     2
     3     sub MAIN (Str $dna!) {
     4         my %count = A => 0, C => 0, G => 0, T => 0;;
     5
     6         for $dna.lc.comb {
     7             when 'a' { %count<A>++ }
     8             when 'c' { %count<C>++ }
     9             when 'g' { %count<G>++ }
    10             when 't' { %count<T>++ }
    11         }
    12
    13         put join ' ', %count<A C G T>;
    14     }
```

At line 4, I declare the hash `%count` that will hold key-value pairs where the keys will be the nucleotides and the values will be the number of times we saw them. It's important (for this version) to initialize the counts or line 13 will complain

if a given base does not have a value. To get around that, we can use a `map` statement:

```
$ cat -n dna7.pl6
     1    #!/usr/bin/env perl6
     2
     3    sub MAIN (Str $dna!) {
     4        my %count;
     5
     6        for $dna.lc.comb {
     7            when 'a' { %count<A>++ }
     8            when 'c' { %count<C>++ }
     9            when 'g' { %count<G>++ }
    10            when 't' { %count<T>++ }
    11        }
    12
    13        put join ' ', %count<A C G T>.map({ $_ // 0 });
    14    }
```

This version has no initialization and waits until line 13 to determine if there is something that exists for a given base. Let's examine this in the REPL:

```
> my %count = A => 1, G => 5;
{A => 1, G => 5}
> %count<A C T G>
(1 (Any) (Any) 5)
> join ' ', %count<A C T G>
Use of uninitialized value <element> of type Any in string conte
xt
Any of .^name, .perl, .gist, or .say can stringify undefined thi
ngs, if needed.  in block <unit> at <unknown file> line 1
Use of uninitialized value <element> of type Any in string conte
xt
Any of .^name, .perl, .gist, or .say can stringify undefined thi
ngs, if needed.  in block <unit> at <unknown file> line 1
> say %count<A C T G>.WHAT
(List)
> %count<A C T G>.map({ $_  // 0 })
(1 0 0 5)
> join ' ', %count<A C G T>.map({ $_  // 0 })
1 0 5 0
```

What the above shows is that we get a List back when we ask for multiple keys from a hash. That list might contain nothing for a given key, so we use the `map` function to apply an anonymous function (the bit in `{}`) to each member of the list. The function says "either *the topic* or, if it's not defined, then 0."

Here's a version that introduces a new type called a "Bag" (https://docs.perl6.org/type/Bag):

```
$ cat -n dna8.pl6
     1    #!/usr/bin/env perl6
     2
     3    sub MAIN (Str $dna!) {
     4        my $bag = $dna.lc.comb.Bag;
     5        put join ' ', $bag<a c g t>;
     6    }
```

Next, I'd like to create a version that can handle input from a file as well as a string:

```
$ cat -n dna9.pl6
     1    #!/usr/bin/env perl6
     2
     3    sub MAIN (Str $input!) {
     4        my $dna = $input.IO.f ?? $input.IO.slurp !! $input
;
     5        my $bag = $dna.lc.comb.Bag;
     6        put join ' ', $bag<a c g t>
     7    }
 $ echo AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAAAAGAG
TGTCTGATAGCAGC > dna
$ ./dna8.pl6 dna
20 12 17 21
$ ./dna8.pl6 foobar
1 0 0 0
```

At line 4, I'm going to set my `$dna` equal to either the contents of the `$input` (if it is a file) or the input itself. I'm using a ternary operator (https://docs.perl6.org/language/operators#index-entry-Ternary_operator) that follows the pattern:

```
result = conditional ?? true-branch !! false-branch
```

In my case, the conditional is to the the `$input` as an `IO` object and test the `f` ("file", https://docs.perl6.org/type/IO$COLON$COLONPath#File_test_operators). If it is `True`, then execute the `slurp` method on the IO object; otherwise, just use the `$input`. Notice that the Bag returns "0" for any missing elements, so no initialization was required.

The advantage to using the "bag" solution is that we can easily expand the script to also count, for instance, "N"s which are unknown base calls or the bases in RNA or proteins:

```
$ cat -n char-count.pl6
     1     #!/usr/bin/env perl6
     2
```

```
     3     sub MAIN (Str $input!) {
     4         my $text  = $input.IO.e ?? $input.IO.slurp !! $in
put;
     5         my $bag   = $text.comb.Bag;
     6         for $bag.keys.grep(/<[a..zA..Z]>/).sort -> $char
{
     7             put join "\t", $char, $bag{$char};
     8         }
     9     }
$ ./char-count.pl6 AANCTANGNACTAGG
A          5
C          2
G          3
N          3
T          2
$ ./char-count.pl6 prot.fa
A          58
C          47
D          42
E          55
F          36
G          60
H          47
I          47
K          42
L          44
M          51
N          54
P          56
Q          50
R          59
S          42
T          45
V          50
W          55
Y          60
```

Handling FASTA

It's most likely you'll encounter sequence data in some standard format that includes information about the sequences (metadata, or data about the data). Usually we see FASTA format, and it's not necessarily easy to parse; therefore, we're going to reach for a tool someone else built to do this for us. Let's try out the BioInfo (https://github.com/MattOates/BioInfo) and BioPerl6 (https://github.com/cjfields/bioperl6) libraries. To install:

```
$ panda install BioInfo
$ panda install BioPerl6
```

And here is how we can incorporate it into our code. First the version with BioInfo:

```
$ cat -n fasta-stat1.pl6
     1    #!/usr/bin/env perl6
     2
     3    use BioInfo::Parser::FASTA;
     4    use BioInfo::IO::FileParser;
     5
     6    sub MAIN (Str $file!) {
     7        die "Not a file ($file)" unless $file.IO.f;
     8
     9        my $seq_file = BioInfo::IO::FileParser.new(
    10            file     => $file,
    11            parser   => BioInfo::Parser::FASTA
    12        );
    13
    14        my @bases = <A C G T>;
    15        my %count;
    16        while (my $seq = $seq_file.get()) {
    17            my $b = $seq.sequence.uc.comb.Bag;
    18
    19            for @bases -> $base {
    20                %count{ $base } += $b{ $base }
    21            }
    22        }
    23
    24        for @bases -> $base {
    25            printf "%10d %s\n", %count{ $base }, $base;
    26        }
    27    }
$ ./fasta-stat1.pl6 mouse.fa
     12168 A
      8944 C
     11990 T
      9059 G
```

And here is a version using BioPerl6:

```
$ cat -n fasta-stat2.pl6
     1    #!/usr/bin/env perl6
     2
     3    use Bio::SeqIO;
     4
     5    sub MAIN (Str $file!) {
     6        die "Not a file ($file)" unless $file.IO.f;
     7
     8        my $seqIO = Bio::SeqIO.new(format => 'fasta', file
 => $file);
     9
    10        my @bases = <A C G T>;
    11        my %count;
    12        while (my $seq = $seqIO.next-Seq) {
    13            my $b = $seq.seq.uc.comb.Bag;
    14
    15            for @bases -> $base {
    16                %count{ $base } += $b{ $base }
    17            }
    18        }
    19
    20        for @bases -> $base {
    21            printf "%10d %s\n", %count{ $base }, $base;
    22        }
    23    }
$ ./fasta-stat2.pl6 mouse.fa
     12168 A
      8944 C
      9059 G
     11990 T
```

Sequence similarity

As our field deals with the sequences of unknown communities of organisms, we are very interested in comparing our samples to known sequences. There are many programs and algorithms for comparing strings, from the venerable BLAST (Basic Local Alignment Search Tool) to hashing to kmers.

K-mers

A "k-mer" is a *k*-length *mer* (from Greek *meros* "part") of contiguous sequence as in the word "polymer." So the given sequence has 3-mers like so:

```
AGCTTTTC
AGC
 GCT
  CTT
   TTT
    TTT
     TTC
```

Here's a simple Perl script to get kmers from a sequence:

```
$ cat -n kmer1.pl6
     1     #!/usr/bin/env perl6
     2
     3     sub MAIN (Str $input!, Int :$k=10) {
     4         if $k < 0 {
     5             note "k ($k) must be positive\n";
     6             exit 1;
     7         }
     8
     9         my $seq = $input.IO.f ?? $input.IO.slurp.chomp !!
 $input;
    10         my $n   = $seq.chars - $k + 1;
    11
    12         if $n < 1 {
    13             note "Cannot extract {$k}-mers from seq lengt
h {$seq.chars}";
    14             exit 1;
    15         }
    16
    17         for 0..^$n -> $i {
    18             put $seq.substr($i, $k);
    19         }
    20     }
```

In lines 4-7, we first want to ensure that we have a positive value for *k*, so we need to explore this idea briefly. Another way to write this would be to move that condition into the signature:

```
$ cat -n pos1.pl6
     1     #!/usr/bin/env perl6
     2
     3     sub MAIN (Int :$k where * > 0 = 10) {
     4         put "k = $k";
     5     }
$ ./pos1.pl6 -k=5
k = 5
$ ./pos1.pl6 -k=-1
Usage:
  ./pos1.pl6 [-k=<Int>]
```

Because that sort of thing is so useful, Perl let's us create our own types as a `subset` of an existing type. I'll arbitrarily limit *k* to a positive integer less than 50:

```
$ cat -n pos2.pl6
     1     #!/usr/bin/env perl6
     2
     3     subset SmallPosInt of Int where 0 < * < 50 ;
     4     sub MAIN (SmallPosInt :$k=10) {
     5         put "k = $k";
     6     }
$ ./pos2.pl6 -k=5
k = 5
$ ./pos2.pl6 -k=-1
Usage:
  ./pos2.pl6 [-k=<Int>]
$ ./pos2.pl6 -k=51
Usage:
  ./pos2.pl6 [-k=<Int>]
```

As it happens, there is a built-in type that will constrain us to a positive integer called `UInt` for "unsigned integer":

```
$ cat -n pos3.pl6
     1     #!/usr/bin/env perl6
     2
     3     sub MAIN (UInt :$k=10) {
     4         put "k = $k";
     5     }
$ ./pos3.pl6 -k=5
k = 5
$ ./pos3.pl6 -k=-1
Usage:
  ./pos3.pl6 [-k=<Int>]
```

Now we can shorten our code and get type checking for free:

```
$ cat -n kmer2.pl6
     1     #!/usr/bin/env perl6
     2
     3     sub MAIN (Str $input!, UInt :$k=10) {
     4         my $seq = $input.IO.f ?? $input.IO.slurp.chomp !!
 $input;
     5         my $n   = $seq.chars - $k + 1;
     6
     7         if $n < 1 {
     8             note "Cannot extract {$k}-mers from seq lengt
h {$seq.chars}";
     9             exit 1;
    10         }
    11
    12         for 0..^$n -> $i {
    13             put $seq.substr($i, $k);
    14         }
    15     }
$ ./kmer2.pl6 -k=20 input.txt
AGCTTTTCATTCTGACTGCA
GCTTTTCATTCTGACTGCAA
CTTTTCATTCTGACTGCAAC
TTTTCATTCTGACTGCAACG
TTTCATTCTGACTGCAACGG
TTCATTCTGACTGCAACGGG
```

At line 5, use a simple formula to determine the number of k-mers we can extract from the given sequence. At lines 7-10, we decide if we can continue based on the input from the user. Lines 12-14 should look somewhat familiar by now. Since we're going to use the zero-based `substr` to extract part of the sequence, we need to use the "..^" range operator to go *up to but not including* our value for `$n`. After that, we just `put` the extracted k-mer.

Here's a slightly shorter version that again uses the `map` function (https://docs.perl6.org/routine/map) to eschew a `for` loop:

```
$ cat -n kmer3.pl6
     1     #!/usr/bin/env perl6
     2
     3     sub MAIN (Str $input!, UInt :$k=10) {
     4         my $seq = $input.IO.f ?? $input.IO.slurp.chomp !!
 $input;
     5         my $n   = $seq.chars - $k + 1;
     6
     7         if $n < 1 {
     8             note "Cannot extract {$k}-mers from seq lengt
h {$seq.chars}";
     9             exit 1;
    10         }
    11
    12         put join "\n", map { $seq.substr($_, $k) }, 0..^$
n;
    13     }
```

It's worth spending time to really learn `map` as it's a very safe and efficient function. It takes two arguments:

- a code block to run
- a list of things

The code block is applied to each element in the list coming in from the right and is emitted on the left:

```
new <- map { code } <- original
```

It's important to remember that a `map` will always return the same number of elements it was given as opposed to a `grep` which looks very similar:

```
> map *.uc, <foo bar baz>
(FOO BAR BAZ)
> grep /ba/, <foo bar baz>
(bar baz)
```

So the "list" argument to `map` in line 12 is the range `0..^$n`. Each number in turn goes into the code block as the "topic" `$_` where it is used in the `substr` call to extract the k-mer which are then returned as a new list to the `join` which then puts a newline between them before the call to `put`.

I'll present one more version that puts the kmers into an array using `gather/take` (https://docs.perl6.org/language/control#gather/take):

```
$ cat -n kmer4.pl6
     1    #!/usr/bin/env perl6
     2
     3    sub MAIN (Str $input!, Int :$k=10) {
     4        my $seq   = $input.IO.f ?? $input.IO.slurp.chomp !
! $input;
     5        my $n     = $seq.chars - $k + 1;
     6        my @kmers = gather for 0..^$n -> $i {
     7            take $seq.substr($i, $k);
     8        }
     9
    10        dd @kmers;
    11    }
```

The `dd` is the built-in "data dumper" that shows you a textual representation of a data structure:

```
$ ./kmer4.pl6 input.txt
Array @kmers = ["AGCTTTTCAT", "GCTTTTCATT",
"CTTTTCATTC", "TTTTCATTCT", "TTTCATTCTG",
"TTCATTCTGA", "TCATTCTGAC", "CATTCTGACT",
"ATTCTGACTG", "TTCTGACTGC", "TCTGACTGCA",
"CTGACTGCAA", "TGACTGCAAC", "GACTGCAACG",
"ACTGCAACGG", "CTGCAACGGG"]
```

Kmers from FASTA

Now let's combine our parsing of FASTA with extraction of k-mers:

```
$ cat -n fasta-kmer1.pl6
     1     #!/usr/bin/env perl6
     2
     3     use Bio::SeqIO;
     4
     5     sub MAIN (Str $file!, UInt :$k=10) {
     6         die "Not a file ($file)" unless $file.IO.f;
     7         my $seqIO = Bio::SeqIO.new(format => 'fasta', fil
e => $file);
     8
     9         while (my $seq = $seqIO.next-Seq) {
    10             put join "\n", get-kmers($k, $seq.seq);
    11         }
    12     }
    13
    14     sub get-kmers(Int $k, Str $str) {
    15         my $n = $str.chars - $k + 1;
    16         map { $str.substr($_, $k) }, 0..^$n;
    17     }
```

I copied the FASTA script from the previous chapter and introduced a function called `get-kmers`. You've already been writing functions, of course, like `MAIN` and calling functions like `join` and `put`, but now we're calling a function we've written. We define that it takes two positional arguments, an integer `$k` and a string `$string`. Right away the type checking caught an error as I tried to pass in the `$seq` *object* and not the `$seq.seq` *string of the sequence.*

Just like with our `MAIN`, we can change our positional arguments into named arguments by placing a `:` before the names (line 14 below):

```
$ cat -n fasta-kmer2.pl6
     1  #!/usr/bin/env perl6
     2
     3  use Bio::SeqIO;
     4
     5  sub MAIN (Str $file! where *.IO.f, UInt :$k=10) {
     6      my $seqIO = Bio::SeqIO.new(format => 'fasta', fil
e => $file);
     7
     8      while (my $seq = $seqIO.next-Seq) {
     9          put join "\n", get-kmers(str => $seq.seq, k =
> $k);
    10      }
    11  }
    12
    13  sub get-kmers(Int :$k, Str :$str) {
    14      my $n = $str.chars - $k + 1;
    15      map { $str.substr($_, $k) }, 0..^$n;
    16  }
```

This means we have to call `get-kmers` with Pairs for arguments (line 10), but it also means the arguments can be specified in any order. Generally, if a function takes more than 2 or 3 arguments, I would recommend using named arguments.

Something else you might have noticed in the above version is that I moved the check of the file-ness of `$input` into a constraint in the `MAIN` signature. If you needed this contraint more than once, it would behoove you to create a new `subset`.

Here is another version that again dips into the function programming world and uses a new string function called `rotor` (http://perl6.party/post/Perl-6-.rotor-The-King-of-List-Manipulation):

```
$ cat -n fasta-kmer3.pl6
     1     #!/usr/bin/env perl6
     2
     3     use Bio::SeqIO;
     4
     5     sub MAIN (Str $file!, UInt :$k=10) {
     6         die "Not a file ($file)" unless $file.IO.f;
     7         my $seqIO = Bio::SeqIO.new(format => 'fasta', fil
e => $file);
     8
     9         my $j = -1 * ($k - 1);
    10         while (my $seq = $seqIO.next-Seq) {
    11             put $seq.seq.comb.rotor($k => $j).map(*.join)
.join("\n");
    12         }
    13     }
```

It's probably easiest to understand this by going into the REPL:

```
> my $s = "ACGTACGT";
ACGTACGT
> $s.comb
(A C G T A C G T)
> $s.comb.rotor: 3
((A C G) (T A C))
> $s.comb.rotor: 3 => -2
((A C G) (C G T) (G T A) (T A C) (A C G) (C G T))
> $s.comb.rotor(3 => -2).map(*.join)
(ACG CGT GTA TAC ACG CGT)
> $s.comb.rotor(3 => -2).map(*.join).join("\n")
ACG
CGT
GTA
TAC
ACG
CGT
```

We've seen before how we can use `comb` to explode a string into a list of characters. Next we call `rotor` to break the string into sublists of 3 elements each, but for k-mers we want `rotor` to back up two (*k* - 1) steps before making the next list, so we pass the Pair (3 => -2). We then pass each sublist into a `map` where we use the `*` to mean "whatever" or "the thing" and call the `join` method (with no argument) to create a string. That list of strings then gets `join`ed on newlines to print all the k-mers in the sequence.

Non-sequence data

You can use k-mers to determine the similarity of non-sequence data. This program will process all the given files in a pair-wise, all-versus-all fashion to determine if any two files are more similar than the overall similarity of all the files as figured with two-sided Student's t-test. Notice that I can use Unicode characters like "μ" (mu) as variable names:

```
$ cat -n kmer-counter.pl6
     1    #!/usr/bin/env perl6
     2
     3    subset PosInt of Int where * > 0;
     4
     5    sub MAIN (PosInt :$k=5, PosInt :$max-sd=5, *@files) {
     6        die "No files" unless @files;
     7        my @bags = map { find-kmers(+$k, $_) }, @files;
     8        my %counts;
     9        for (1..@bags.elems).combinations(2) -> ($i, $j) {
    10            my $bag1  = @bags[$i-1];
    11            my $bag2  = @bags[$j-1];
    12            my $s1    = $bag1.Set;
    13            my $s2    = $bag2.Set;
    14            my @union = ($s1 (&) $s2).keys;
    15            my $sum   = (map { $bag1{ $_ } }, @union)
    16                      + (map { $bag2{ $_ } }, @union);
    17            %counts{"$i-$j"} = $sum;
    18        }
    19
    20        my @n  = %counts.values;
```

```
    21        my $μ  = mean @n;
    22        my $sd = std-dev @n;
    23        my $d  = $sd/(@n.elems).sqrt;
    24        for %counts.kv -> $pair, $sum {
    25            # https://en.wikipedia.org/wiki/Student%27s_t-
test
    26            my $t = ($sum - $μ) / $d;
    27            if $t.abs > $max-sd {
    28                my ($i, $j) = $pair.split('-');
    29                my $f1 = @files[$i-1].IO.basename;
    30                my $f2 = @files[$j-1].IO.basename;
    31                put "$pair ($sum) = $t [$f1, $f2]";
    32            }
    33        }
    34    }
    35
    36    sub find-kmers (Int $k, Str $file) {
    37        my $text = $file.IO.lines.lc.join(' ')
    38                    .subst(/:i <-[a..z\s]>/, '', :g).subst(
/\s+/, ' ');
    39        $text.comb.rotor($k => -1 * ($k - 1)).map(*.join).
Bag;
    40    }
    41
    42    sub mean (*@n) { @n.sum / @n.elems }
    43
    44    sub std-dev (*@n) {
    45        # https://en.wikipedia.org/wiki/Standard_deviation
    46        my $mean = mean(@n);
    47        my @dev  = map { ($_ - $mean)² }, @n;
    48        my $var  = @dev.sum / @dev.elems;
    49        return $var.sqrt;
    50    }
```

Here is the result of running this program on the homework submissions for a class of mine:

```
1-9 (74) = -6.36237878762838 [s01.pl6, s09.pl6]
1-4 (60) = -8.97194821224158 [s01.pl6, s04.pl6]
5-6 (136) = 5.19428580708723 [s05.pl6, s06.pl6]
3-7 (208) = 18.6149285622408 [s03.pl6, s07.pl6]
2-4 (76) = -5.98958315554078 [s02.pl6, s04.pl6]
6-8 (142) = 6.31267270335003 [s06.pl6, s08.pl6]
4-10 (74) = -6.36237878762838 [s04.pl6, s10.pl6]
7-8 (178) = 13.0229940809268 [s07.pl6, s08.pl6]
3-10 (208) = 18.6149285622408 [s03.pl6, s10.pl6]
3-8 (142) = 6.31267270335003 [s03.pl6, s08.pl6]
6-9 (78) = -5.61678752345318 [s06.pl6, s09.pl6]
4-5 (66) = -7.85356131597878 [s04.pl6, s05.pl6]
7-10 (198) = 16.7509504018028 [s07.pl6, s10.pl6]
4-6 (68) = -7.48076568389118 [s04.pl6, s06.pl6]
4-9 (72) = -6.73517441971598 [s04.pl6, s09.pl6]
5-8 (148) = 7.43105959961283 [s05.pl6, s08.pl6]
2-9 (80) = -5.24399189136558 [s02.pl6, s09.pl6]
```

The amount of similarity is fairly high, which is not unexpected from running this on Perl code; however, a few of these are really too high to be explained by chance. The extremely high ones can be isolated:

```
$ ./kmer-counter.pl6 --max-sd=16 ~/homework/*.pl6
3-7 (208) = 18.6149285622408 [s03.pl6, s07.pl6]
3-10 (208) = 18.6149285622408 [s03.pl6, s10.pl6]
7-10 (198) = 16.7509504018028 [s07.pl6, s10.pl6]
```

And, indeed, I found that "s03.pl6" was submitted with minor changes by two other students as "s07.pl6" and "s10.pl6" (names changed to protect the innocent, of course).

Types and Regular Expressions

Regular expressions (https://docs.perl6.org/language/regexes) constitute a sub-language within Perl. There have been many changes from Perl 5's regex syntax, but the principles remain the same.

Let's play in the REPL:

```
$ perl6
To exit type 'exit' or '^D'
> 1234 ~~ /\d/
「1」
> 1234 ~~ /\d+/
「1234」
> 1234 ~~ Int
True
> 'foobar' ~~ /oo/
「oo」
> 'foooobar' ~~ /o+/
「oooo」
```

Here's a more complicated regex for detecting US-style phone numbers:

```
> my $re = /'('? \d ** 3 ')'? <[\s*-]>? \d ** 3 <[\s*-]>? \d **
4 /
> /'('? \d ** 3 ')'? <[\s*-]>? \d ** 3 <[\s*-]>? \d ** 4 /
> my @phones = "(520) 321 9087", "520 321 9087", "520-321-9087",
 "5203219087", "(520)321-9087"
[(520) 321 9087 520 321 9087 520-321-9087 5203219087 (520)321-90
87]
> for @phones -> $phone { say $phone ~~ $re }
True
True
True
True
True
> all(@phones) ~~ $re
True
```

The last method uses a Junction (https://docs.perl6.org/type/Junction) to compare all the phone numbers at once.

The subject of regexes is far too deep to cover here. For now, just know that, if you can describe the pattern of text you are searching for, then you can probably write a regex to match it. I want to use regexes as a bridge to talking about creating your own types and something called "multiple dispatch."

In the last chapter, I introduced a "subset" where you can create a new type in Perl. First, let's look at a simple case where you might want to detect the type of sequence you have.

```
$ cat -n seq-type1.pl6
     1    #!/usr/bin/env perl6
     2
     3    sub MAIN (Str $input!) {
     4        if $input ~~ /^ :i <[ACTGN]>+ $/ {
     5            put "Looks like DNA";
     6        }
     7        elsif $input ~~ /^ :i <[ACUGN]>+ $/ {
     8            put "Looks like RNA";
     9        }
    10        elsif $input ~~ /^ :i <[A..Z]>+ $/ {
    11            put "Looks like protein";
    12        }
    13        else {
    14            put "Unknown sequence type";
    15        }
    16    }
$ ./seq-type1.pl6 ACGT
Looks like DNA
$ ./seq-type1.pl6 ACGU
Looks like RNA
$ ./seq-type1.pl6 EEDS
Looks like protein
$ ./seq-type1.pl6 883k19!
Unknown sequence type
```

The `~~` smart-match operator https://docs.perl6.org/language/operators#infix_~~ matches the left side to the regex on the right. Each regex is asking if the `$input` matches entirely to a given alphabet. To break it down:

```
/ ^ :i <[ACTGN]>+ $ /
1 2 3  4        5 6 7
```

1. the start of a regular expression
2. hat or caret anchors to the beginning of the string
3. an "adverb" saying that the next part matches case-insensitively

4. a character class composed of the letters inside
5. one or more of the preceding
6. dollar anchors to the end of the string
7. the end of the regular expression

We've already seen that a long if/elsif chain is better represented with `given/when`:

```
$ cat -n seq-type2.pl6
     1     #!/usr/bin/env perl6
     2
     3     sub MAIN (Str $input!) {
     4         given $input {
     5             when /^ :i <[ACTGN]>+ $/ { put "Looks like DN
A" }
     6             when /^ :i <[ACUGN]>+ $/ { put "Looks like RN
A" }k
     7             when /^ :i <[A..Z]>+  $/ { put "Looks like pr
otein";
     8             default { put "Unknown sequence type" }
     9         }
    10     }
```

It would be much nicer to isolate and document our regexes so they are reusable. We can store them in scalar variables that have sensible names. There are many reasons for doing this, not the least of which is that they are then documented and represented in just one place in your code. Suppose that you initially defined the DNA pattern as just "ACGT" and used it several places in a script. Later you realize you want to add "N" or maybe even the full IUPAC table (http://www.bioinformatics.org/sms/iupac.html). If you fail to update the regex in every place in your code, you've introduced a very subtle bug that you may spend hours fixing -- or worse, you may never find!

```
$ cat -n seq-type3.pl6
     1    #!/usr/bin/env perl6
     2
     3    sub MAIN (Str $input!) {
     4        my $dna     = /^ :i <[ACTGN]>+ $/;
     5        my $rna     = /^ :i <[ACUGN]>+ $/;
     6        my $protein = /^ :i <[A..Z]>+  $/;
     7
     8        given $input {
     9            when $dna     { put "Looks like DNA" }
    10            when $rna     { put "Looks like RNA" }
    11            when $protein { put "Looks like protein"; }
    12            default       { put "Unknown sequence type" }
    13        }
    14    }
```

Now I'll show you how to move our regular expressions into types.

```
$ cat -n seq-type4.pl6
     1    #!/usr/bin/env perl6
     2
     3    subset DNA     of Str where * ~~ /^ :i <[ACTGN]>+ $/;
     4    subset RNA     of Str where * ~~ /^ :i <[ACUGN]>+ $/;
     5    subset Protein of Str where * ~~ /^ :i <[A..Z]>+  $/;
     6
     7    sub MAIN (Str $input!) {
     8        given $input {
     9            when DNA     { put "Looks like DNA" }
    10            when RNA     { put "Looks like RNA" }
    11            when Protein { put "Looks like protein"; }
    12            default      { put "Unknown sequence type" }
    13        }
    14    }
```

Now whenever you need the DNA pattern, you just reach for your `DNA` type. You could even define common regexes/patterns in a single module (file) that you import into your scripts so that you only ever define (and fix) them in one place.

Now for that "multiple dispatch" I mentioned earlier. Perl can do pattern matching on the signatures of functions. We can use the `multi` keyword instead of `sub` to indicate to Perl that we intend to define `MAIN` in different ways:

```
$ $ cat -n seq-type5.pl6
     1    #!/usr/bin/env perl6
     2
     3    subset DNA     of Str where * ~~ /^ :i <[ACTGN]>+ $/;
     4    subset RNA     of Str where * ~~ /^ :i <[ACUGN]>+ $/;
     5    subset Protein of Str where * ~~ /^ :i <[A..Z]>+  $/;
     6
     7    multi MAIN (DNA     $input!) { put "Looks like DNA" }
     8    multi MAIN (RNA     $input!) { put "Looks like RNA" }
     9    multi MAIN (Protein $input!) { put "Looks like Protein
" }
    10    multi MAIN (Str     $input!) { put "Unknown sequence t
ype" }
$ ./seq-type5.pl6 ACGT
Looks like DNA
$ ./seq-type5.pl6 ACGU
Looks like RNA
$ ./seq-type5.pl6 EEDS
Looks like Protein
$ ./seq-type5.pl6 2112
Unknown sequence type
```

After you get the feel of using regexes, you can take them further into writing grammars. After you get the hang of using types, you can take them further into creating objects which package up data and methods, e.g., a DNA object that can produce it's reverse complement or an RNA object that can find its ORFs and translate its coding regions into amino acids.

Counting Things

Given a file like this:

```
$ cat -n names.txt
     1     cat
     2     dog
     3     mouse
     4     bird
     5     cat
     6     cat
     7     dog
```

We can count the number of times each animal occurs with command-line tools:

```
$ sort names.txt | uniq -c
   1 bird
   3 cat
   2 dog
   1 mouse
```

If we wanted the output sorted numerically up or down:

```
$ sort names.txt | uniq -c | sort -n
   1 bird
   1 mouse
   2 dog
   3 cat
$ sort names.txt | uniq -c | sort -nr
   3 cat
   2 dog
   1 mouse
   1 bird
```

If we wanted them sorted instead on the animals:

```
$ sort names.txt | uniq -c | awk 'OFS="\t" { print $2,$1 }' | so
rt
bird       1
cat        3
dog        2
mouse      1
$ sort names.txt | uniq -c | awk 'OFS="\t" { print $2,$1 }' | so
rt -r
mouse      1
dog        2
cat        3
bird       1
```

Obviously this is getting a bit complicated, and it will get worse as we need to handle ugly data like "Mouse/MOUSE/mice/muose," etc. And what if we needed to handle this file:

```
$ cat -n name-value.txt
     1     mouse      1
     2     cat        3
     3     bird       1
     4     dog        10
     5     frog       5
     6     cat        2
     7     mouse      3
     8     frog       1
```

Let's write a Perl script!

```
$ cat -n name-count1.pl6
     1     #!/usr/bin/env perl6
     2
     3     sub MAIN (Str $file! where *.IO.f) {
     4         my %count;
     5         for $file.IO.lines -> $key {
     6             %count{ $key }++;
     7         }
     8
     9         for %count.kv -> $key, $value {
    10             put join "\t", $key, $value;
    11         }
    12     }
$ ./name-count1.pl6 names.txt
dog        2
cat        3
bird       1
mouse      1
```

By now, this code should look pretty self-explanatory. We set up the `%count` hash so that we can increment (`++`) for each line in the file (lines 4-7), and then we call the key-value pairs via `%count.kv` to `put` the keys and their counts joined on a tab character. One problem is that `%count.kv` returns the data in no particular order, so let's add an option to sort the data with an option to use descending order.

```
$ cat -n name-count2.pl6
     1    #!/usr/bin/env perl6
     2
     3    sub MAIN (Str $file! where *.IO.f, :$desc=False) {
     4        my %count;
     5        for $file.IO.lines -> $key {
     6            %count{ $key }++;
     7        }
     8
     9        my @sorted = %count.sort;
    10        @sorted   .= reverse if $desc;
    11
    12        for @sorted -> $pair {
    13            put join "\t", $pair.key, $pair.value;
    14        }
    15    }
$ ./name-count2.pl6 --desc names.txt
mouse      1
dog        2
cat        3
bird       1
```

I added a "flag" called `--desc` set by default to `False`. At line 9, we sort the `%count` hash to populate an Array of Pair objects. Remember that hashes are stored in an unordered manner, so it's necessary to put the Pairs into a container that will respect their ordering. The only reason to put it into a Array is for line 10 where we mutate the Array *in place* by calling `.= reverse` on it if the `$desc` flag is `True`.

Now let's look at a version that sorts by the *count* of the keys:

```
$ cat -n name-count3.pl6
     1     #!/usr/bin/env perl6
     2
     3     sub MAIN (Str $file! where *.IO.f, Bool :$desc=False)
 {
     4         my %keys;
     5         for $file.IO.lines -> $key {
     6             %keys{ $key }++;
     7         }
     8
     9         my @sorted = %keys.sort(*.value);
    10         @sorted   .= reverse if $desc;
    11
    12         for @sorted -> $pair {
    13             put join "\t", $pair.key, $pair.value;
    14         }
    15     }
 $ ./name-count3.pl6 names.txt
bird       1
mouse      1
dog        2
cat        3
$ ./name-count3.pl6 --desc names.txt
cat        3
dog        2
mouse      1
bird       1
```

There are just two changes to this script. First, I added a `Bool` constraint on the `$desc` to indicate that I want to treat this as a binary, on/off flag. The other change is on line 9 where I pass to the `sort` call the *unary* operation `*.value`. Take this example:

```
> my @names = <fred barney Wilma Betty>
[fred barney Wilma Betty]
> @names.sort
(Betty Wilma barney fred)
> @names.sort(*.lc)
(barney Betty fred Wilma)
> @names.sort(*.chars)
(fred Wilma Betty barney)
```

A call to `sort` use an lexicographic sort where uppercase characters occur before lowercase (cf. the ASCII table). Passing the code `*.lc` tells `sort` to call the lowercase method on each of the arguments before sorting, so we get a case-insensitive ordering. Likewise, passing `*.chars` says to order by the number of characters in the strings.

Now let's merge these scripts so that the user can choose whether to sort on the keys (default) or the counts either in ascending (default) or descending order:

```
$ cat -n name-count4.pl6
     1  #!/usr/bin/env perl6
     2
     3  subset SortBy of Str where /^keys?|values?$/;
     4  sub MAIN (Str $file! where *.IO.f, SortBy :$sort-by='
key', Bool :$desc=False) {
     5      my %keys;
     6      for $file.IO.lines -> $key {
     7          %keys{ $key }++;
     8      }
     9
    10      my @sorted = $sort-by ~~ /key/ ?? %keys.sort !! %
keys.sort(*.value);
    11      @sorted   .= reverse if $desc;
    12
    13      for @sorted -> $pair {
    14          put join "\t", $pair.key, $pair.value;
    15      }
    16  }
$ ./name-count4.pl6 --sort-by=value --desc names.txt
cat     3
dog     2
mouse   1
bird    1
```

I've create a `subset` on line 3 to describe the allowed values for the `$sort-by` argument. I've described it as a String that must match the regular expression:

```
/ ^ keys? | values? $ /
1 2 3   4 5 6     7 8 9
```

1. beginning of regular expression
2. start of the string
3. the string "key"
4. optional string "s"
5. OR
6. the string "value"

7. optional string "s"
8. the end of the string
9. end of regular expression

This allows both "key" or "keys," "value" or "values." Line 10 is now using the ternary operator (`predicate ?? true-branch !! false-branch`) with a test on whether the `$sort-by` matches the string "key" (which will also match "keys"). If it does match, it sort as normal, otherwise it sorts on the values. The rest is identical to before.

As you might have guessed, I'm now going to show you a significantly shorter version:

```
$ cat -n name-count5.pl6
     1     #!/usr/bin/env perl6
     2
     3     sub MAIN (Str $file! where *.IO.f) {
     4         put $file.IO.lines.Bag.map(*.kv.join("\t")).join(
"\n");
     5     }
```

As great as hashes are, a Bag (immutable) or a BagHash (mutable) is the better option for this task. We saw a Bag in the DNA problem, so that much should not be new. The processing of it with `map` and `join` might be confusing, so let's look more closely at that in the RELP:

```
> my $bag = 'names.txt'.IO.lines.Bag
bag(mouse, cat(3), bird, dog(2))
> $bag.map(*.kv.join('::'))
(mouse::1 cat::3 bird::1 dog::2)
> $bag.map(*.kv.join('::')).join(' -- ')
mouse::1 -- cat::3 -- bird::1 -- dog::2
```

I'm using commas and dashes as delimiters as tabs and newlines won't show well in the REPL. The Bag has a `map` method that allows us to apply a block of code to each element. The Bag is hold Pairs like "mouse => 1," "cat => 3," so we call `*.kv` to get a List:

```
> (mouse => 1).kv
(mouse 1)
> say (mouse => 1).kv.WHAT
(List)
```

Then we ask that the List be joined:

```
> say (mouse => 1).kv.join('::')
mouse::1
```

The result of the `map` operation is a new List of Strings that we then want to `join` on newlines.

Obviously this is a big of golf on my part. It's not a script I would actually release or maintain, in part because it's not as functional as the previous one. If you wanted to take it a bit further, we can get down to a "one-liner":

```
$ perl6 -e 'put $*IN.lines.Bag.map(*.join("\t")).join("\n")' < n
ames.txt
mouse      1
cat        3
bird       1
dog        2
```

Here's is a more realistic program:

```
$ cat -n name-count6.pl6
     1     #!/usr/bin/env perl6
     2
     3     subset SortBy of Str where * (elem) <key value keys v
alues>.Bag;
     4     sub MAIN (Str $file! where *.IO.f, SortBy :$sort-by='
key', Bool :$desc=False) {
     5         my $bag    = $file.IO.lines.Bag;
     6         my @sorted = $sort-by ~~ /key/ ?? $bag.sort !! $b
ag.sort(*.value);
     7         @sorted   .= reverse if $desc;
     8         put join "\n", map *.join("\t"), @sorted;
     9     }
```

On line 3, I'm showing you another way to define a set of acceptable strings using the test `(elem)` to see if *the thing* (`*`) is in the Bag. Otherwise, this script is pretty close to the earlier versions, only shorter and less error prone.

Here is another version with some interesting variations:

```
$ cat -n name-count7.pl6
     1     #!/usr/bin/env perl6
     2
     3     subset SortBy of Str where * ∈ <key value keys values
>.Bag;
     4     sub MAIN (Str $file! where *.IO.f, SortBy :$sort-by='
key', Bool :$desc=False) {
     5         my $bag    = $file.IO.lines.Bag;
     6         my @sorted = $sort-by ~~ /key/ ?? $bag.sort !! $b
ag.sort(*.value);
     7         @sorted   .= reverse if $desc;
     8         put @sorted.map(*.join("\t")).join("\n");
     9     }
```

Line 3 uses the Unicode version `∈` of the `(elem)` operator. Perl handles Unicode natively for both code and data! I confess I show you this just because I think it looks cool, but I actually prefer the regular expression version. The other

change is at line 8 where I use chained method calls instead of functions to generate the output.

Now let's move on to a version that sums the counts for various keys:

```
$ cat -n name-value-count1.pl6
     1     #!/usr/bin/env perl6
     2
     3     subset SortBy of Str where /:i ^keys?|values?$/;
     4     sub MAIN (Str $file! where *.IO.f, SortBy :$sort-by='
key', Bool :$desc=False) {
     5         my %counts;
     6         for $file.IO.lines -> $line {
     7             my ($key, $value) = $line.split("\t");
     8             %counts{ $key } += $value;
     9         }
    10
    11         my @sorted = $sort-by ~~ /key/ ?? %counts.sort !!
 %counts.sort(*.value);
    12         @sorted   .= reverse if $desc;
    13
    14         for @sorted -> $pair {
    15             put join "\t", $pair.key, $pair.value;
    16         }
    17     }
```

This is almost identical to one of our first versions. I've gone back to the regular expression match for the `SortBy` because now I'm using the `:i` modifier to make the match case-insensitive. At line 7, I'm calling `$line.split` to break the line on the tab character into the `$key` and `$value`. Since I'm assiging them as a list, I need the parentheses around the `my ()`. At line 8, I'm using the `+=` operator to add the `$value` to whatever was in `$count{ $key }`. If there was nothing there, the key is created and set to 0. The rest of the script continues as before.

Here is another version to consider:

```
$ cat -n name-value-count2.pl6
     1  #!/usr/bin/env perl6
     2
     3  subset SortBy of Str where  /:i ^ [key|value]s? $ /;
     4  sub MAIN (Str $file! where *.IO.f, SortBy :$sort-by='
key', Bool :$desc=False) {
     5      my %counts;
     6      for $file.IO.lines.map(*.split(/\s+/)) -> [$key,
$value] {
     7          %counts{ $key } += $value;
     8      }
     9
    10      my @sorted = $sort-by ~~ /key/ ?? %counts.sort !!
 %counts.sort(*.value);
    11      @sorted   .= reverse if $desc;
    12
    13      put @sorted.map(*.join("\t")).join("\n");
    14  }
$ ./name-value-count2.pl6 --desc name-value.txt
mouse   4
frog    6
dog     10
cat     5
bird    1
$ ./name-value-count2.pl6 --desc --sort-by=value name-value.txt
dog     10
frog    6
cat     5
mouse   4
bird    1
```

On line 3, I wrote the regex a little more succinctly to say "either 'key' or 'value' and then an optional 's'." On line 6, I'm using a regular expression in the `split` call to say I want to split on any number of whitespace characters. This would break if we had a key like "grey wolf," so it's important to know your data. I'm just showing you another way to specify the `split`. The other big change here is that the `for` has a signature after the `-> [$key, $value]`. Remember that curlies `{}` create code blocks, and code blocks can have signatures. Here we're

having Perl match the pattern `[$key, $value]` to say we're expecting something that looks like a two-element list and to put those values into the variables we named.

Parsing Structured Data

We've looked at parsing structured data formats like delimited text files (commas, tabs) and FASTA format. If there's one thing that we do not have a shortage of in bioinformatics, it's structured file formats. Let's look a some more examples.

JSON

JavaScript Object Notation is a very popular way to exchange data on the internet. It has mostly supplanted XML as the de facto hierarchical text file (i.e., not just lines of records all having the same fields but things that can contain things like how genes can contain exons, introns, CDS, etc.).

Let's say you have a PubMed ID (27208118), and you'd like to get the details of the article. NCBI has web-accessible tools for this (http://www.ncbi.nlm.nih.gov/books/NBK25499/). Here is how we can use `wget` to fetch a JSON file:

```
$ wget --quiet -O 27208118.json 'https://eutils.ncbi.nlm.nih.gov
/entrez/eutils/esummary.fcgi?db=pubmed&retmode=json&id=27208118'
$ head 27208118.json
{
    "header": {
        "type": "esummary",
        "version": "0.3"
    },
    "result": {
        "uids": [
            "27208118"
        ],
        "27208118": {
```

And here is a simple way to incorporate this into a Perl script and parse the JSON. For this to work, we'll do `panda install JSON::Tiny`. Here's the script:

```
$ cat -n pubmed1.pl6
     1     #!/usr/bin/env perl6
     2
     3     use File::Temp;
     4     use JSON::Tiny;
     5
     6     # http://www.ncbi.nlm.nih.gov/books/NBK25499/
     7     constant $PUBMED_URL = 'https://eutils.ncbi.nlm.nih.g
ov/entrez/eutils/'
     8                           ~ 'esummary.fcgi?db=pubmed&retmo
de=json&id=';
     9
    10     sub MAIN (Int $pubmed-id=27208118) {
    11         my ($tmpfile, $tmpfh) = tempfile();
    12         $tmpfh.close;
    13         run(«wget --quiet -O $tmpfile "$PUBMED_URL$pubmed
-id"»);
    14         my $json = $tmpfile.IO.slurp;
    15         my $data = from-json($json);
    16         $tmpfile.IO.unlink;
    17
    18         if $data{'result'}{$pubmed-id}.defined {
    19             my %pubmed = $data{'result'}{$pubmed-id};
    20             put "$pubmed-id = %pubmed{'title'} (%pubmed{'
lastauthor'})";
    21         }
    22         else {
    23             put "Cannot find PubMed ID '$pubmed-id'";
    24             exit 1;
    25         }
    26     }
$ ./pubmed1.pl6
27208118 = Potential Mechanisms for Microbial Energy Acquisition
 in Oxic Deep-Sea Sediments. (Heidelberg JF)
$ ./pubmed1.pl6 00000001
Cannot find PubMed ID '00000001'
```

On lines 3-4, I brought in a couple of modules I'll need to create a temporary file and parse JSON. Line 7 have a `constant` declaration to indicate I don't want anything to change this string which is the URL of the "esummary" tool. At line 11, I use `tempfile` to create a temporary file. This can be harder to get right than you imagine. Here's a reason:

```
Knock, knock.
Race condition.
Who's there?
```

So to avoid race conditions, please use the `File::Temp` module. I don't actually need the file handle, so I `close` it so I can pass the temp filename to the `wget` command. Then I can `slurp` in the file (line 14) and parse the JSON (line 15) before getting rid of the tempfile (line 16).

If the call to "esummary" was successful, then the given PubMed ID would exist in the `result` section of the JSON. From there I can easily extract the "title" and "lastauthor" (line 20). If not, I need to let the user know and exit with a failure status (lines 23-24).

As it happens, we use web services like this quite a bit, so Perl has much better tools than just running `wget` . Here is an example using the "LWP::Simple" module (use `panda install` to install it):

```
$ cat -n pubmed2.pl6
     1    #!/usr/bin/env perl6
     2
     3    use LWP::Simple;
     4    use JSON::Tiny;
     5
     6    # http://www.ncbi.nlm.nih.gov/books/NBK25499/
     7    constant $PUBMED_URL = 'http://eutils.ncbi.nlm.nih.go
v/entrez/eutils/'
     8                           ~ 'esummary.fcgi?db=pubmed&retmo
de=json&id=';
     9
    10    sub MAIN (Int $pubmed-id=27208118) {
    11        my $lwp  = LWP::Simple.new;
    12        my $json = $lwp.get("$PUBMED_URL$pubmed-id");
    13        my $data = from-json($json);
    14
    15        if $data{'result'}{$pubmed-id}.defined {
    16            my %pubmed = $data{'result'}{$pubmed-id};
    17            put "$pubmed-id = %pubmed{'title'} (%pubmed{'
lastauthor'})";
    18        }
    19        else {
    20            put "Cannot find PubMed ID '$pubmed-id'";
    21            exit 1;
    22        }
    23    }
[gila@~/work/metagenomics-book/perl6/structured-data]$ ./pubmed2
.pl6
27208118 = Potential Mechanisms for Microbial Energy Acquisition
 in Oxic Deep-Sea Sediments. (Heidelberg JF)
```

Now we don't have to worry about those tempfiles. LWP will impersonate a web browser and `get` the URL for us, returning the JSON (or XML or HTML or whatever) to us. The rest looks the same.

Problems

1. Alter the above script to print a list of all the authors sorted by their names.
2. Turn this into a script that takes a list of PubMed IDs either from the command line or a file and prints the titles and last authors for each.

Parsing, Regular Expressions, Grammars

Sharing code with modules and objects

As you write more code, you will find yourself solving some problems repeatedly. Your first instinct might be to copy and paste the needed code, but it is far better to put it into a module or object to make it easier to re-use it. For one thing, if you find a bug, you'll want to change it just once and not have to remember all the places you pasted it. Eventually you may find you've solved a problem that many other people have or, vice versa, others have solved your problem. Modules are the way to package and share code.

Modules

Here's an example of a simple module that has some code/type/functions that we've seen before:

```
$ cat -n DNA.pm6
     1    unit module DNA;
     2
     3    subset DNA of Str is export where /^ :i <[ACGTN]>+ $/;
     4
     5    sub revcom (DNA $seq) is export {
     6        $seq.trans(<A C G T a c g t> => <T G C A t g c a>)
.flip;
     7    }
     8
     9    sub hamming (DNA $s1, DNA $s2) is export {
    10        ($s1.chars - $s2.chars).abs +
    11        ($s1.comb Z $s2.comb).grep({ $^a[0] ne $^a[1] });
    12    }
```

I created a file called "DNA.pm6" ("pm" == "Perl module") with a `unit module DNA` line to establish for Perl that this is the "DNA" module. Inside it, I have defined a `subset` type called "DNA" that allows me to easily reuse my regular expression. I also have two functions that are generally useful, `revcom` to compute the reverse complement of a string of DNA, and `hamming` to compute

the number of point variations between two pieces of DNA. Notice the `is export` statement that allows the "DNA" module to allow the code to be used in another program like so:

```
$ cat -n module1.pl6
     1    #!/usr/bin/env perl6
     2
     3    use lib '.';
     4    use DNA;
     5
     6    sub MAIN (DNA $seq) {
     7        put "$seq revcom is {revcom($seq)}";
     8    }
$ ./module1.pl6 AACTAGAN
AACTAGAN revcom is NTCTAGTT
```

The "DNA.pm6" and this script live in the same directory. For the same security reason as why your `$PATH` does not normally include your current working directory, Perl does not automatically include "." in your library path. You either need to put your module somewhere in your `$PERL6LIB` path, or you need to `use lib '.'`. After that, then you can `use DNA` to bring in the "DNA" module exported code.

Here is how you might bring in the `hamming` function:

```
$ cat -n hamming.pl6
     1    #!/usr/bin/env perl6
     2
     3    use lib '.';
     4    use DNA;
     5
     6    sub MAIN (DNA $seq1, DNA $seq2) {
     7        printf "Hamming distance from '%s' to '%s': %s\n",
     8            $seq1, $seq2, hamming($seq1, $seq2);
     9    }
$ ./hamming.pl6 AACTAG CAAGAA
Hamming distance from 'AACTAG' to 'CAAGAA': 4
```

OOPs, I did it again

Objects are another way to package up code. The TLA (three-letter acronym) "OOP" stands for "object-oriented programming." It's a way to couple data/state with functions that act on that data.

To illustrate, let's model a "Person," first with a hash:

```
$ cat -n person1.pl6
     1    #!/usr/bin/env perl6
     2
     3    my %geddy = first_name => 'Geddy', last_name => 'Lee';
     4    my %alex  = first_name => 'Alex',  last_name => 'Leifs
on';
     5    my %neil  = first_name => 'Neil',  last_name => 'Peart
';
     6
     7    for %geddy, %alex, %neil -> %person {
     8        printf "%s %s\n", %person<first_name>, %person<las
t_name>;
     9    }
$ ./person1.pl6
Geddy Lee
Alex Leifson
Neil Peart
```

And now with objects:

```
$ cat -n person2.pl6
     1    #!/usr/bin/env perl6
     2
     3    class Person {
     4        has Str $.first_name;
     5        has Str $.last_name;
     6    }
     7
     8    my $geddy = Person.new(first_name => 'Geddy', last_nam
e => 'Lee');
     9    my $alex  = Person.new(first_name => 'Alex',  last_nam
e => 'Leifson');
    10    my $neil  = Person.new(first_name => 'Neil',  last_nam
e => 'Peart');
    11
    12    for $geddy, $alex, $neil -> $person {
    13        printf "%s %s\n", $person.first_name, $person.last
_name;
    14    }
$ ./person2.pl6
Geddy Lee
Alex Leifson
Neil Peart
```

OK, pretty similar, so why go with objects when hashes could work just fine. Well, for one thing, hashes won't enforce key names, and mistakes could be made:

```
$ cat -n person3.pl6
     1    #!/usr/bin/env perl6
     2
     3    my %geddy = first_name => 'Geddy', last_name => 'Lee';
     4    my %alex  = frist_name => 'Alex',  last_name => 'Leifs
on';
     5    my %neil  = first_name => 'Neil',  last_neme => 'Peart
';
     6
     7    for %geddy, %alex, %neil -> %person {
     8        printf "%s %s\n", %person<first_name>, %person<las
t_name>;
     9    }
[saguaro@~/work/metagenomics-book/perl6/oop]$ ./person3.pl6
Geddy Lee
Use of uninitialized value of type Any in string context.
Methods .^name, .perl, .gist, or .say can be used to stringify i
t to something meaningful.
  in block  at ./person3.pl6 line 7
Use of uninitialized value of type Any in string context.
Methods .^name, .perl, .gist, or .say can be used to stringify i
t to something meaningful.
  in block  at ./person3.pl6 line 7
 Leifson
Use of uninitialized value of type Any in string context.
Methods .^name, .perl, .gist, or .say can be used to stringify i
t to something meaningful.
  in block  at ./person3.pl6 line 7
Use of uninitialized value of type Any in string context.
Methods .^name, .perl, .gist, or .say can be used to stringify i
t to something meaningful.
  in block  at ./person3.pl6 line 7
Neil
```

Ugh. Can you spot the typos in the code -- "frist_name" and "last_neme"? Contrast with this:

```
$ cat -n person4.pl6
     1    #!/usr/bin/env perl6
     2
     3    class Person {
     4        has Str $.first_name is required;
     5        has Str $.last_name  is required;
     6    }
     7
     8    my $geddy = Person.new(first_name => 'Geddy', last_nam
e => 'Lee');
     9    my $alex  = Person.new(frist_name => 'Alex',  last_nam
e => 'Leifson');
    10    my $neil  = Person.new(first_name => 'Neil',  last_nem
e => 'Peart');
    11
    12    for $geddy, $alex, $neil -> $person {
    13        printf "%s %s\n", $person.first_name, $person.last
_name;
    14    }
$ ./person4.pl6
The attribute '$!first_name' is required, but you did not provid
e a value for it.
  in block <unit> at ./person4.pl6 line 9
```

Because we declared that the "first_name" and "last_name" are required attributes, we can't even run our code with those typos now. We could also try misspelling the method (notice "list_name" in the `for` loop):

```
$ cat -n person5.pl6
     1    #!/usr/bin/env perl6
     2
     3    class Person {
     4        has Str $.first_name is required;
     5        has Str $.last_name  is required;
     6    }
     7
     8    my $geddy = Person.new(first_name => 'Geddy', last_nam
e => 'Lee');
     9    my $alex  = Person.new(first_name => 'Alex',  last_nam
e => 'Leifson');
    10    my $neil  = Person.new(first_name => 'Neil',  last_nam
e => 'Peart');
    11
    12    for $geddy, $alex, $neil -> $person {
    13        printf "%s %s\n", $person.first_name, $person.list
_name;
    14    }
$ ./person5.pl6
No such method 'list_name' for invocant of type 'Person'
  in block <unit> at ./person5.pl6 line 12
```

What we're doing, obviously, is printing the full name for each member of the greatest rock band in the history of all mankind. Everytime we want to do this, we have to manually concantenate the first and last names, or we could make our users (ourselves) duplicate the names into a "full_name" of the hash (very Bad Idea), or we could write a function to do it for us:

```
$ cat -n person6.pl6
     1    #!/usr/bin/env perl6
     2
     3    my %geddy = first_name => 'Geddy', last_name => 'Lee';
     4    my %alex  = first_name => 'Alex',  last_name => 'Leifs
on';
     5    my %neil  = first_name => 'Neil',  last_name => 'Peart
';
     6
     7    sub full_name (%person) {
     8        join ' ', %person<first_name>, %person<last_name>;
     9    }
    10
    11    for %geddy, %alex, %neil -> %person {
    12        printf "Full Name: %s\n", full_name(%person);
    13    }
$ ./person6.pl6
Full Name: Geddy Lee
Full Name: Alex Leifson
Full Name: Neil Peart
```

Contrast with having the method being contained within the object:

```
$ cat -n person7.pl6
     1    #!/usr/bin/env perl6
     2
     3    class Person {
     4        has Str $.first_name is required;
     5        has Str $.last_name  is required;
     6        method full_name { join ' ', $!first_name, $!last_
name }
     7    }
     8
     9    my $geddy = Person.new(first_name => 'Geddy', last_nam
e => 'Lee');
    10    my $alex  = Person.new(first_name => 'Alex',  last_nam
e => 'Leifson');
    11    my $neil  = Person.new(first_name => 'Neil',  last_nam
e => 'Peart');
    12
    13    for $geddy, $alex, $neil { put "Full Name: ", .full_na
me }
$ ./person7.pl6
Full Name: Geddy Lee
Full Name: Alex Leifson
Full Name: Neil Peart
```

Objects provide us the ability to make all sorts of requirements and validation of the data we're using. Notice we required the first and last names to be the type `Str`, and we could easily add "birthday" as a `Date` or even create our own type like `Instrument`:

```
$ cat -n person8.pl6
     1    #!/usr/bin/env perl6
     2
     3    enum Instrument <Guitar Bass Drums>;
     4
     5    class Person {
     6        has Str $.first_name is required;
     7        has Str $.last_name  is required;
     8        has Date $.birthday;
     9        has Instrument $.instrument;
    10        method full_name { join ' ', $!first_name, $!last_
name }
    11    }
    12
    13    my Person $geddy .= new(first_name => 'Geddy', last_na
me => 'Lee',
    14                           birthday => Date.new(1953, 6, 29),
 instrument => Bass);
    15    my Person $alex  .= new(first_name => 'Alex',  last_na
me => 'Leifson',
    16                           birthday => Date.new(1953, 10, 27)
, instrument => Guitar);
    17    my Person $neil  .= new(first_name => 'Neil',  last_na
me => 'Peart',
    18                           birthday => Date.new(1952, 9, 12),
 instrument => Drums);
    19
    20    for $geddy, $alex, $neil {
    21        printf "%s (%s) born %s\n", .full_name, .instrumen
t, .birthday;
    22    }
$ ./person8.pl6
Geddy Lee (Bass) born 1953-06-29
Alex Leifson (Guitar) born 1953-10-27
Neil Peart (Drums) born 1952-09-12
```

With this `Person` class, it is not possible to instantiate an object with a Integer for the `birthday` or any `Instrument` we have not enumerated. While it may not seem like that big of a gain when you are manually creating the objects, if you

are reading in a file given to you by a collaborator, such data validation can help you clean and normalize your data. Hashes will never give you that security.

Let's say someone new joins the band like Michael J. Fox. How are we going to handle printing his full name? Obviously we must include his middle initial, so we can expand the class to have such a field but make it optional with a default value of "". We can leave the implementation of "full_name" to the object and continue with the data we had before:

```
$ cat -n person9.pl6
     1    #!/usr/bin/env perl6
     2
     3    enum Instrument <Guitar Bass Drums>;
     4
     5    class Person {
     6        has Str $.first_name is required;
     7        has Str $.middle_initial = '';
     8        has Str $.last_name  is required;
     9        has Date $.birthday;
    10        has Instrument $.instrument;
    11        method full_name {
    12            sprintf "%s %s%s",
    13            $!first_name,
    14            $!middle_initial ?? $!middle_initial ~ ' ' !!
'',
    15            $!last_name
    16        }
    17    }
    18
    19    my Person $geddy .= new(first_name => 'Geddy', last_na
me => 'Lee',
    20                        birthday => Date.new(1953, 6, 29),
 instrument => Bass);
    21    my Person $alex  .= new(first_name => 'Alex',  last_na
me => 'Leifson',
    22                        birthday => Date.new(1953, 10, 27)
, instrument => Guitar);
    23    my Person $neil  .= new(first_name => 'Neil',  last_na
me => 'Peart',
```

```
    24                          birthday => Date.new(1952, 9, 12),
 instrument => Drums);
    25    my Person $mjfox .= new(first_name => 'Michael',  last
_name => 'Fox',
    26                          middle_initial => 'J.',
    27                          birthday => Date.new(1961, 6, 9),
instrument => Guitar);
    28
    29    for $geddy, $alex, $neil, $mjfox {
    30        printf "%s (%s) born %s\n", .full_name, .instrumen
t, .birthday;
    31    }
$ ./person9.pl6
Geddy Lee (Bass) born 1953-06-29
Alex Leifson (Guitar) born 1953-10-27
Neil Peart (Drums) born 1952-09-12
Michael J. Fox (Guitar) born 1961-06-09
```

Now let's take our code from the "DNA" module and turn it into an object. We'll start really simply:

```
$ cat -n dna1.pl6
     1    #!/usr/bin/env perl6
     2
     3    class DNA is Str {}
     4
     5    sub MAIN (Str $seq) {
     6        my $dna = DNA.new(value => $seq);
     7        dd $dna;
     8        printf "%s is %s characters long.\n", $dna, $dna.c
hars;
     9    }
$ ./dna1.pl6 GAAGACT
DNA $dna = "GAAGACT"
GAAGACT is 7 characters long.
```

Remember that our `subset DNA` was derived from `Str`, so here we are inheriting from the `Str` class for our `DNA` module because it really is just a string. By subclassing `Str`, we get all the native String methods for free!

There's at least one big problem with this object:

```
$ ./dna1.pl6 foo
DNA $dna = "foo"
```

That's not a valid string of DNA, so we need to fix that:

```
$ cat -n dna2.pl6
     1    #!/usr/bin/env perl6
     2
     3    class DNA is Str {}
     4
     5    sub MAIN (Str $seq) {
     6        if so $seq.uc ~~ /^ <[ACGTN]>+ $/ {
     7            my $dna = DNA.new(value => $seq);
     8            dd $dna;
     9        }
    10        else {
    11            put "'$seq' not a DNA sequence.";
    12        }
    13    }
$ ./dna2.pl6 GACTAG
DNA $dna = "GACTAG"
$ ./dna2.pl6 foo
'foo' not a DNA sequence.
```

OK, that works, but it's really a bad implementation because the code to check whether the string is valid DNA belongs in the object like so:

```
$ cat -n dna3.pl6
     1    #!/usr/bin/env perl6
     2
     3    class DNA is Str {
     4        multi method ACCEPTS (Str $seq) {
     5            return $seq ~~ /^ :i <[ACGTN]>+ $/;
     6        }
     7    }
     8
     9    sub MAIN (Str $seq) {
    10        if $seq ~~ DNA {
    11            my $dna = DNA.new(value => $seq);
    12            dd $dna;
    13        }
    14        else {
    15            put "'$seq' not a DNA sequence.";
    16        }
    17    }
$ ./dna3.pl6 GGACT
DNA $dna = "GGACT"
$ ./dna3.pl6 foo
'foo' not a DNA sequence.
```

Now I can pattern-match on the `DNA` class because I implemented the `ACCEPTS` (https://docs.perl6.org/routine/ACCEPTS) method. I used the `multi` keyword BECAUSE...?

There is still a big problem with this version, though, because the check of the data is still happening outside the object. We need to override the default `new` method to make sure we are getting valid data:

```
$ cat -n dna4.pl6
     1    #!/usr/bin/env perl6
     2
     3    class DNA is Str {
     4        multi method ACCEPTS (Str $seq) {
     5            return $seq.uc ~~ /^ :i <[ACGTN]>+ $/;
     6        }
     7
     8        method new (*%args) {
     9            my $value = %args<value>.Str;
    10            if $value !~~ DNA {
    11                fail "'$value' not a DNA sequence.";
    12            }
    13            self.bless(|%args);
    14        }
    15    }
    16
    17    sub MAIN (Str $seq) {
    18        try {
    19            my $dna = DNA.new(value => $seq);
    20            dd $dna;
    21            CATCH { default { .Str.say } }
    22        }
    23    }
$ ./dna4.pl6 GGACTA
DNA $dna = "GGACTA"
$ ./dna4.pl6 foo
'foo' not a DNA sequence.
```

Now it is impossible to create a `DNA` object with invalid input because the `new` method will `fail` if the string does not match our regex. You'll notice a new keyword `self` in the `new` function -- that references the object itself and is not preceded by a `$` (WHY?). Because of the `fail`, I have introduced a `try` block in the calling code where the `CATCH` block is executed on any exceptions.

I'd like to be able to create my DNA object by just passing the string and not the Pair `value => $seq`. I can change the way the `new` constructor works:

```
$ cat -n dna5.pl6
     1    #!/usr/bin/env perl6
     2
     3    class DNA is Str {
     4        multi method ACCEPTS (Str $seq) {
     5            return $seq.uc ~~ /^ <[ACGTN]>+ $/;
     6        }
     7
     8        method new (Str $str) {
     9            if $str.uc !~~ DNA {
    10                fail "'$str' not a DNA sequence.";
    11            }
    12
    13            self.bless(value => $str);
    14        }
    15    }
    16
    17    sub MAIN (Str $str) {
    18        try {
    19            my $dna = DNA.new($str);
    20            dd $dna;
    21            CATCH { default { .Str.say } }
    22        }
    23    }
$ ./dna5.pl6 GAAACT
DNA $dna = "GAAACT"
$ ./dna5.pl6 foo
'foo' not a DNA sequence.
```

Now let's move our `DNA` class into a separate file like our earlier module, both to reduce the size of our program and to allow us to reuse it in another program:

```
$ cat -n DNA1.pm6
     1    class DNA is Str {
     2        multi method ACCEPTS (Str $seq) {
     3            return $seq.uc ~~ /^ <[ACGTN]>+ $/;
     4        }
     5
     6        method new (Str $str) {
     7            if $str.uc !~~ DNA {
     8                fail "'$str' not a DNA sequence.";
     9            }
    10
    11            self.bless(value => $str);
    12        }
    13    }
$ cat -n dna6.pl6
     1    #!/usr/bin/env perl6
     2
     3    use lib '.';
     4    use DNA1;
     5
     6    sub MAIN (Str $str) {
     7        try {
     8            my $dna = DNA.new($str);
     9            dd $dna;
    10            CATCH { default { .Str.say } }
    11        }
    12    }
$ ./dna6.pl6 GAACTG
DNA $dna = "GAACTG"
$ ./dna6.pl6 foo
'foo' not a DNA sequence.
```

Let's add some more functionality to our `DNA` class such as our `revcom` and `hamming` methods, a method to tell us the length of the DNA, and an attribute to let us know if we have the forward or reverse strand. While we're at it, let's have both of our `new` methods to allow users to create a `DNA` object either with a single argument or a Pair.

```
$ cat -n DNA2.pm6
     1    enum Direction <Forward Reverse>;
     2
     3    class DNA is Str {
     4        has Direction $.direction = Forward;
     5
     6        multi method ACCEPTS (Str $seq) {
     7            return $seq ~~ /^ :i <[ACGTN]>+ $/;
     8        }
     9
    10        # e.g., DNA.new($seq1);
    11        multi method new (Str $seq) {
    12            if $seq !~~ DNA {
    13                fail "'$seq' not a DNA sequence.";
    14            }
    15            self.bless(value => $seq);
    16        }
    17
    18        # e.g., DNA.new(value => $seq1);
    19        multi method new (*%args) {
    20            my $value = %args<value>.Str;
    21            if $value !~~ DNA {
    22                fail "'$value' not a DNA sequence.";
    23            }
    24            self.bless(|%args);
    25        }
    26
    27        method revcom {
    28            self.trans(<A C G T a c g t> => <T G C A t g c
 a>).flip;
    29        }
    30
    31        method length { self.chars }
    32
    33        method hamming (DNA $other) {
    34            return (self.chars - $other.chars).abs +
    35                   (self.comb Z $other.comb).grep({ $^a[0]
 ne $^a[1] });
    36        }
```

```
    37    }
```

I created an `enum` type for the `Direction` that consists of the two values, `Forward` and `Reverse`. This is a type like we've seen before, and it allows me to constrain the new attribute `direction` to only those two values. The constructor for that constraint:

```
has Direction $.direction = Forward;
```

Says that the class has a scalar value called `direction` that is of the type `Direction` and which has a default value of `Forward`. The `has` keyword will create accessor/mutator methods called `direction` for us to get (access) or change (mutate, if we so allow) the direction. By default, object attributes are read-only, so we'd have to explicitly say `is rw` to declare that it is read-write. Here read-only is the correct way to go because it is not something we should allow to be changed.

Here is how you can use the code:

```
$ cat -n dna7.pl6
     1    #!/usr/bin/env perl6
     2
     3    use lib '.';
     4    use DNA2;
     5
     6    sub MAIN (Str $str) {
     7        try {
     8            my $i    = 0;
     9            my $temp = "%s: %s has the direction '%s' and
length '%s'.\n";
    10
    11            my $dna1 = DNA.new($str);
    12            printf $temp, ++$i, $dna1, $dna1.direction, $d
na1.length;
    13
    14            my $dna2 = DNA.new(value => $str, direction =>
 Forward);
    15            printf $temp, ++$i, $dna2, $dna2.direction, $d
na2.length;
    16
    17            my $dna3 = DNA.new(value => $str, direction =>
 Direction.pick);
    18            printf $temp, ++$i, $dna3, $dna3.direction, $d
na3.length;
    19
    20            CATCH { default { .Str.say } }
    21        }
    22    }
```

Notice at line 11 I create the `DNA` object with a single argument and no `direction,` but at line 14 I use two Pairs to define the `value` and `direction`, while at line 17 I define the `value` and randomly pick a `Direction`. Here's what it looks like when it runs:

```
$ ./dna7.pl6 GATAGA
1: GATAGA has the direction 'Forward' and length '6'.
2: GATAGA has the direction 'Forward' and length '6'.
3: GATAGA has the direction 'Reverse' and length '6'.
$ ./dna7.pl6 foo
'foo' not a DNA sequence.
```

Here is how we could call our `hamming` and `revcom` functions:

```
$ cat -n hamming.pl6
     1    #!/usr/bin/env perl6
     2
     3    use lib '.';
     4    use DNA2;
     5
     6    sub MAIN (Str $seq1, Str $seq2) {
     7        try {
     8            my $dna1 = DNA.new($seq1);
     9            my $dna2 = DNA.new($seq2);
    10            put "Hamming distance from '$dna1' to '$dna2':
 ", $dna1.hamming($dna2);
    11            CATCH { default { .Str.say } }
    12        }
    13    }
$ ./hamming.pl6 GGATC GGCCC
Hamming distance from 'GGATC' to 'GGCCC': 2
$ cat -n revcom.pl6
     1    #!/usr/bin/env perl6
     2
     3    use lib '.';
     4    use DNA2;
     5
     6    sub MAIN (Str $str) {
     7        try {
     8            my $dna = DNA.new($str);
     9            printf "Input : %s\nRevcom: %s\n", $dna, $dna.
revcom;
    10            CATCH { default { .Str.say } }
    11        }
    12    }
[saguaro@~/work/metagenomics-book/perl6/oop]$ ./revcom.pl6 GATTA
GA
Input : GATTAGA
Revcom: TCTAATC
```

And remember that, since we inherited from `Str`, we get all those methods, too!

```
$ cat -n substr.pl6
     1    #!/usr/bin/env perl6
     2
     3    use lib '.';
     4    use DNA2;
     5
     6    sub MAIN (Str :$seq, Int :$start=0, Int :$stop=0) {
     7        try {
     8            my $dna = DNA.new($seq);
     9            $stop ||= $dna.chars;
    10            put "DNA from $start to $stop: ", $dna.substr(
$start, $stop);
    11            CATCH { default { .Str.say } }
    12        }
    13    }
$ ./substr.pl6 --seq=GGATACC --start=2 --stop=5
DNA from 2 to 5: ATACC
```

The main idea behind both modules and objects is to hide complexity from the user (even if the user is just you). You package up everything belonging to DNA or whatever into a file or module or object, and the code that uses it doesn't have to worry about how the algorigthms or objects or whatever is implemented.

Hangman

I think lots of examples are the way to teach, and probably you're tired of just thinking about DNA by now. So let's create a "Puzzle" object for playing "Hangman." Here's how a game looks when won:

```
$ ./hangman.pl6
puzzle = _ _ _ _ _ _ _ []
What is your guess? a
puzzle = _ _ _ _ _ _ _ [a]
What is your guess? i
puzzle = _ _ _ _ _ _ _ [ai]
What is your guess? e
puzzle = _ e _ _ _ e _ [aei]
What is your guess? o
puzzle = _ e _ _ _ e _ [aeio]
What is your guess? u
puzzle = _ e _ _ u e _ [aeiou]
What is your guess? s
puzzle = _ e s _ u e _ [aeiosu]
What is your guess? t
puzzle = _ e s _ u e _ [aeiostu]
What is your guess? r
puzzle = r e s _ u e r [aeiorstu]
What is your guess? c
puzzle = r e s c u e r [aceiorstu]
You won!
```

The game needs to have a random dictionary word which we can easily pluck from "/usr/share/dict/words" of some minimum (default 5) and maximum (default 9) length. During the game, we need to know the word, what the user has guessed so far, and how many of the letters have been found. After each guess, we need to decide if the puzzle has been solved or if the user has exceeded the allowed number of attempts (default 10). While it's not at all necessary to use a new type/object to handle this, we can see quite a benefit to wrapping all the puzzle logic up in one place and the user interface in another. Here's the code:

```
#!/usr/bin/env perl6

class Puzzle {
    has Str $.word;
    has Str @.state;
    has Int %.guesses;
```

```
    submethod BUILD (Str :$word) {
        $!word  = $word;
        @!state = '_' xx $word.chars;
    }

    method Str {
        "puzzle = {@.state.join(' ')} [{%.guesses.keys.sort.join
}]";
    }

    method guess (Str $char) {
        unless %.guesses{ $char }++ {
            for $.word.indices($char) -> $i {
                @.state[$i] = $char;
            }
        }
    }

    method was-guessed (Str $char) {
        return %.guesses{ $char }.defined;
    }

    method number-guessed {
        return %.guesses.keys.elems;
    }

    method is-solved {
        none(@.state) eq '_';
    }
}

sub MAIN(Int :$num-guesses = 10, :$min-word-len=5, :$max-word-le
n=9) {
    my $words  = '/usr/share/dict/words';
    my $word   = $words.IO.lines.grep(
                   {$min-word-len <= .chars <= $max-word-len}).pic
k.lc;
    my $puzzle = Puzzle.new(word => $word);

    loop {
```

```
        put ~$puzzle;
        if $puzzle.is-solved {
            put "You won!";
            last;
        }

        if $puzzle.number-guessed >= $num-guesses {
            put "Too many guesses. The word was '$word\.' You lo
se.";
            last;
        }

        my $guess = (prompt "What is your guess? ").lc;
        if $guess !~~ m:i/^<[a..z]>$/ {
            put "Please guess just one letter";
            next;
        }

        if $puzzle.was-guessed($guess) {
            put "You guessed that before!";
            next;
        }

        $puzzle.guess($guess);
    }
}
```

The `Puzzle` holds the current state of the game. It knows the word which is being guessed, which letters have been guessed so far (and therefore how many guesses have been made), and which letters have been found. When we `loop` through the game, we just need to ask the `Puzzle` for information like if the game is over either because too many guesses were made or because the puzzle has been solved. The game could be written entirely without objects, but objects give us a way to wrap up state -- how things change through time -- with methods -- how to change things, how to ask about change.

Bouncy Balls

Here's a simple game that bounces balls inside a box in your terminal. When two balls collide, they explode with a Unicode star "★" and disappear from the board:

```
 1  #!/usr/bin/env perl6
 2
 3  subset PosInt of Int where * > 0;
 4
 5  class Ball {
 6      has Int $.rows;
 7      has Int $.cols;
 8      has Int $.row is rw = (2..^$!rows).pick;
 9      has Int $.col is rw = (2..^$!cols).pick;
10      has Int $.horz-dir is rw = (1, -1).pick;
11      has Int $.vert-dir is rw = (1, -1).pick;
12
13      method Str { join ',', $!row, $!col }
14
15      method pos { ($.row, $.col) }
16
17      method move {
18          $!col += $!horz-dir;
19          $!row += $!vert-dir;
20          $!horz-dir *= -1 if $!col <= 1 || $!col >= $!c
ols;
21          $!vert-dir *= -1 if $!row <= 1 || $!row >= $.r
ows;
22      }
23  }
24
25  my $DOT           = "\x25A0"; # ■
26  my $STAR          = "\x2605"; # ★
27  my $SMILEY-FACE   = "\x263A"; # ☺
28  my ($ROWS, $COLS) = qx/stty size/.words;
29
30  sub MAIN (
31      PosInt :$rows=$ROWS - 4,
32      PosInt :$cols=$COLS - 2,
33      PosInt :$balls=10,
34      Numeric :$refresh=.075,
35      Bool :$smiley=False,
```

```
    36    ) {
    37        print "\e[2J";
    38        my Str $bar    = '+' ~ '-' x $cols ~ '+';
    39        my $icon       = $smiley ?? $SMILEY-FACE !! $DOT;
    40        my Ball @balls = Ball.new(:$rows, :$cols) xx $ball
s;
    41
    42        loop {
    43            .move for @balls;
    44
    45            my $positions = (@balls».Str).Bag;
    46
    47            my %row;
    48            for $positions.list -> (:$key, :$value) {
    49                my ($row, $col) = $key.split(',');
    50                %row{ $row }.append: $col => $value;
    51            }
    52
    53            print "\e[H";
    54            my $screen = "$bar\n";
    55
    56            for 1..$rows -> $this-row {
    57                my $line = '|' ~ " " x $cols;
    58                if %row{ $this-row }:exists {
    59                    for %row{ $this-row }.list -> (:$key,
:$value) {
    60                        $line.substr-rw($key, 1) = $value
== 1 ?? $icon !! $STAR;
    61                    }
    62                }
    63
    64                $screen ~= "$line|\n";
    65            }
    66
    67            $screen ~= $bar;
    68            put $screen;
    69            sleep $refresh;
    70            my @collisions = $positions.grep(*.value > 1).
map(*.key);
    71            @balls = @balls.grep(none(@collisions) eq *.St
```

```
r);
    72        }
    73    }
```

While it's not a requirement to use objects for this game, they do allow me to wrap up the state of each ball into a little package that I can interact with by saying `move` to get the ball to figure:

1. know its current location
2. know its current direction (up/down, left/right)
3. reverse its course if it encounters a boundary

Once I've `move`d all the balls, I can get all their positions and draw them to the screen. I need to know if more than one ball occupies any location, so I `Bag` them up to get counts. If a position has more than one ball, I use the star icon to indicate the collision and remove the offenders from the `@balls` array.

Tests

You can easily write a test suite in Perl using the "Test" module. The DNA test script checks that the script prints a "usage"-like statement when run with no arguments or if it prints the correct output given a known sequence from either the command line or a file. You can look at the "test.pl6" script to understand more how it works. You should totally write tests.

One methodology to writing software is to actually write the tests *first* (called "test-driven development" or "TDD") and then write the software passes the tests. It's a good way to define at least the behavior of a program, e.g., the expected output for a given input. This is another reason I would encourage you to create a `MAIN` entry point with named parameters as it forces you to consider what you want from the user.

Testing scripts

As a simple example, let's look at the test for the parallel file reader:

```
$ cat test.pl6
#!/usr/bin/env perl6

use Test;

my $expected = q:to/END/;
Sequence_1
1    2    3    4
A    B    C    D
Sequence_2
5    6    7    8
E    F    G    H
END

ok my $proc =
    run("./parallel.pl6", "--phred=phred.txt", "--seq=seq.txt",
:out),
    "Ran script";

is $proc.out.slurp-rest, $expected, "Got expected output";

done-testing();
```

My first test is just to see if I'm able to `run` the script with a couple of known input files, and so I use `ok` to find out if the `Proc` (https://docs.perl6.org/type/Proc) was created. Next I ask `is` the standard out of the process the same as the expected text.

I usually like to include a Makefile to assist me in running and testing my scripts:

```
$ cat Makefile
run:
    ./parallel.pl6 --phred=phred.txt --seq=seq.txt

test:
    ./test.pl6
```

If I run `make` with no target, the first target is run:

```
$ make
./parallel.pl6 --phred=phred.txt --seq=seq.txt
Sequence_1
1    2    3    4
A    B    C    D
Sequence_2
5    6    7    8
E    F    G    H
```

And it's so very satisfying to run `make test` and see all those "ok"s:

```
$ make test
./test.pl6
ok 1 - Ran script
ok 2 - Got expected output
1..2
```

Testing libraries

Testing scripts is a bit laborious and tricky as they are usually designed to complete some monolithic tasks, so you are often limited to giving some input and checking that the output was what you expect. As you move your code into libraries and modules to aid in sharing, you'll find it's much easier to write tests for each particular method. In my blackjack example, each object (Card, Deck, Player, Game) had complex interactions, and I quickly found I wanted to test each part individually. By moving the `class` code into a module, I could `use` any part and test them in isolation.

Use tests outside of test suites

Tests don't have to be limited to test suites for your scirpts or libraries. In this example, I needed to ensure that I had completely downloaded all the files from a collaborator by using MD5 checksums. That's a test, so I decided to use the `is` assertion for an elegant solution and obvious output.

Perl was created for systems administration, and Perl 6 has all the chops you've come to expect from the brand. Here I needed to use MD5 checksums from my collaborator to verify that I downloaded all their data without errors. Each data "$file" has an accompanying "$file.md5" that looks like this:

```
$ cat HOT232_1_0770m/prodigal.gff.md5
a36e4adfaa62cc4adb8cea44c4f7825f  HOT232_1_0770m/prodigal.gff
```

So I need to read the contents of this file, get just the first field, then execute my local "md5" (or "md5sum") program on the file without the ".md5" extension and determine if they are the same. All standard stuff, and I think Perl 6 gives us elegant ways to accomplish all of these, including a dead-simple testing framework. Here's my solution:

```
$ cat -n check-md5.pl6
     1    #!/usr/bin/env perl6
     2
     3    use File::Find;
     4    use Test;
     5
     6    sub MAIN (Str :$dir=~$*CWD, Str :$md5="md5sum", Str :$
ext='.md5') {
     7        my $rx = rx/$ext $/;
     8        for find(:dir($dir), :name($rx)) -> $md5-file {
     9            my $basename    = $md5-file.basename;
    10            my ($remote, $) = $md5-file.slurp.split(' ');
    11            my $local-file  = $basename.subst($rx, '');
    12            my $path        = $*SPEC.catfile(
    13                                $md5-file.dirname, $local-fi
le);
    14            my ($local, $)  = run($md5, $path, :out)
    15                                .out.slurp-rest.split(' ');
    16            is $local, $remote, $md5-file;
    17        }
    18
    19        done-testing();
    20    }
```

Here I'll default to looking in the current directory which is accessible as the global variable `$*CWD`. Remember that this variable uses a `$*` "twigle" (two sigils) to denote a global variable which in this case is actually an IO object. I need to stringify it either by putting it in double-quotes or coercing it with the `~` operator. The "md5" argument is the name of my local "md5sum" binary which is often called just "md5." Lastly, I assume the file extension of the remote checksums is ".md5."

Users of Perl 5 will be pleased to note that "File::Find" is available. I was always more of a fan of "File::Find::Rule," I think the interface for the Perl 6 module is quite nice. The output is a list of IO::Path objects.

As a side note, I first wrote it like this:

```
for (find(:dir($dir), :name(rx/$ext $/))).kv -> $i, $md5-file {
    printf "%3d: %s\n", $i + 1, $md5-file;
}
```

Because I like to know as each file is being processed and how many have gone by. Take any list in Perl 6 and call `.kv` (key-value) on it to get the index (position) and the value:

```
> my @list = "foo", "bar", "baz"
[foo bar baz]
> @list.kv
(0 foo 1 bar 2 baz)
> for @list.kv -> $key, $val { put "$key = $val" }
0 = foo
1 = bar
2 = baz
```

Reading an entire file is done by calling slurp on it. In my case, there's only one line, and I want to immediately split it on spaces:

```
my ($remote, $) = $md5-file.slurp.split(' ');
```

Since I'm only interested in the first field (the second one, remember, is the name of the file, which I already know), I can ignore it by assigning the position to an anonymous scalar `$`. To remove the extension, I use the `Str.subst` (substitute) method with the regular expression I put into the `$rx` variable. (The first argument to `subst` can also be a string.) So, if the $md5-file is "/work/03137/kyclark/ohana/HOT/HOT237_2_1000m/readpool.fastq.md5", then the basename is "readpool.fastq.md5" and the result of the `subst` is "readpool.fastq":

```
my $local-file = $basename.subst($rx, '');
```

To piece together the local file that needs to be tested, I can use the global `$*SPEC` object to get access to OS-dependent methods like `catfile` that will use the proper directory separators to construct the path. Again, the `$*` is a twigle to set the global variable/object apart from everything else. The local file should be in the same directory as the MD5 checksum file.

```
my $path = $*SPEC.catfile($md5-file.dirname, $local-file);
```

To get the local checksum, I need to run the correct "md5" binary and then read STDOUT from that program. (To capture STDERR, I could put ":err" in the call.) I put "backticks" in the title to highlight that this is now the way to call external programs and capture the output. Like the MD5 file, I'm only interested in the first field of output and can throw away the rest. I can accomplish it all in one line:

```
my ($local, $) = run($md5, $path, :out).out.slurp-rest.split(' '
);
```

Finally, I brought in the "Test" module so that I can ask:

```
is $local, $remote, $md5-file;
```

If they are equal, I get output like this:

```
$ ./check-md5.pl6
ok 1 - /work/03137/kyclark/ohana/HOT/HOT224_1_0025m/prodigal.gff
.md5
ok 2 - /work/03137/kyclark/ohana/HOT/HOT224_1_0025m/readpool.fas
tq.md5
ok 3 - /work/03137/kyclark/ohana/HOT/HOT224_1_0025m/ribosomal_rR
NA.gff.md5
...
```

When they aren't, I see this:

```
not ok 193 - /work/03137/kyclark/ohana/HOT/HOT224_1_0045m/readpo
ol.fastq.md5

# Failed test '/work/03137/kyclark/ohana/HOT/HOT224_1_0045m/read
pool.fastq.md5'
# at ./check-md5.pl6 line 13
# expected: '4b49aaa2b0c60342e90cf37e30c30886'
#      got: '4cdd81767c2ffed712c0b8ffb6de8b38'
```

And so I know to get that file again!

Web Development with Perl 5

As of fall 2016, I still do all my web development in Perl 5 because the ecology of modules is so mature. Here I will describe how I typically go about creating a website. For example, I will reference a small project I built for an affordable housing non-profit in Tucson (https://github.com/kyclark/metagenomics-book/tree/master/web). You can instantiate the entire website and database from the Github repo.

Data Model

> Show me your flowcharts and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowcharts; they'll be obvious. -- Fred Brooks

Nothing captures my philosophy better than the above quote. Everything in my method starts from the database design and is propagated by code outward. I tend to use MySQL with InnoDB tables so that my database has foreign key constraints. (The only time I don't use InnoDB is when I need a feature like FULLTEXT indexes.)

Here is an example schema:

```
SET foreign_key_checks=0;

drop table if exists case_worker;
create table case_worker (
  case_worker_id int unsigned NOT NULL AUTO_INCREMENT PRIMARY KE
Y,
  name varchar(100)
) ENGINE=InnoDB;

drop table if exists case_worker_to_client;

drop table if exists client_phone;
create table client_phone (
```

```
  client_phone_id int unsigned NOT NULL AUTO_INCREMENT PRIMARY K
EY,
  client_id int unsigned not null,
  phone varchar(50),
  type varchar(50),
  foreign key (client_id) references client (client_id) on delet
e cascade
) ENGINE=InnoDB;

drop table if exists client;
create table client (
  client_id int unsigned NOT NULL AUTO_INCREMENT,
  case_worker_id int unsigned default null,
  case_num int unsigned not null default 0,
  first_name varchar(100) DEFAULT NULL,
  last_name varchar(100) DEFAULT NULL,
  aka varchar(100) default null,
  dob date default null,
  sex enum('U', 'M', 'F') null default 'U',
  ethnicity varchar(50) default null,
  race varchar(50) default null,
  is_homeless enum('Y', 'N') default 'N',
  is_disabled enum('Y', 'N') default 'N',
  is_employed enum('Y', 'N') default 'N',
  marital_status varchar(100),
  original_date_service date default null,
  address_street varchar(100) default null,
  address_city varchar(100) default null,
  address_state varchar(100) default null,
  address_zip varchar(100) default null,
  email varchar(255) default null,
  notes text,
  KEY last_name (last_name),
  foreign key (case_worker_id) references case_worker (case_work
er_id) on delete cascade,
  PRIMARY KEY (client_id)
) ENGINE=InnoDB;

drop table if exists dependent;
create table dependent (
```

```
  dependent_id int unsigned NOT NULL AUTO_INCREMENT primary key,
  client_id int unsigned NOT NULL,
  first_name varchar(100) DEFAULT NULL,
  last_name varchar(100) DEFAULT NULL,
  sex enum('U', 'M', 'F') null default 'U',
  relationship varchar(100) DEFAULT NULL,
  dob date default null,
  ethnicity varchar(50) default null,
  race varchar(50) default null,
  is_disabled enum('Y', 'N') default 'N',
  foreign key (client_id) references client (client_id) on delet
e cascade
) ENGINE=InnoDB;
```

Here is a picture of the database:

Some of my rules for database nomenclature:

1. Name tables in the singular
2. Define a primary key as an auto-incrementing field with a combination of the

table name + "_id." Avoid using this suffix as much as possible, but I may exceptions for "pubmed_id" or "genbank_id" where the ID is in some external database.

3. Use "is_" as a prefix for Boolean fields, e.g., "is_disabled" so that you can write `$client->is_disabled`
4. Use "ENUM" where possible to constrain data input, e.g., `sex enum('U', 'M', 'F')`
5. Define reasonable defaults where possible, "sex" can default to "U" for "unknown/undefined"

Once I have a database, I use SQL::Translator (http://search.cpan.org/dist/SQL-Translator) to generate my object-relational model (ORM). SQLT was a module I created to help me translate MySQL schemas to Oracle. With the help of dozens of others, it expanded to be able to create documentation, images (such as the above E/R diagram), and code directly from schema definitions. Here I have a script that digests the tables, fields, and relationships directly from the schema to create DBIx::Class (http://search.cpan.org/dist/DBIx-Class) modules for interacting with the database.

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

use strict;
use warnings;
use autodie;
use feature 'say';
use DBIx::Class::Schema::Loader qw/ make_schema_at /;
use FindBin '$Bin';
use File::Path 'mkpath';
use File::Spec::Functions 'canonpath';
use Getopt::Long;
use Compass::DB;
use Pod::Usage;
use Readonly;

my $out_dir = "/usr/local/compass/lib";
my $debug   = 0;
my ( $help, $man_page );
GetOptions(
```

```
    'd|debug' => \$debug,
    'help'    => \$help,
    'man'     => \$man_page,
) or pod2usage(2);

if ( $help || $man_page ) {
    pod2usage({
        -exitval => 0,
        -verbose => $man_page ? 2 : 1
    });
};

if (!-d $out_dir) {
    mkpath $out_dir;
}

my $db = Compass::DB->new;

make_schema_at(
    'Compass::Schema',
    {
        debug          => 0,
        dump_directory => $out_dir,
        use_moose      => 1,
        overwrite_modifications => 1,
    },
    [ $db->dsn, $db->user, $db->password ]
);

__END__

# ------------------------------------------------------

=pod

=head1 NAME

mk-dbix.pl - creates the DBIx::Schema classes

=head1 SYNOPSIS
```

```
  mk-dbix.pl

Options:

  -o|--out    Output directory (defaults to ../lib)
  -d|--debug  Show debug info
  --help      Show brief help and exit
  --man       Show full documentation

=head1 DESCRIPTION

Reads the configured "imicrobe" db and creates the DBIx::Class s
chema.

=head1 SEE ALSO

DBIx::Class.

=head1 AUTHOR

Ken Youens-Clark E<lt>E<gt>.

=head1 COPYRIGHT

Copyright (c) 2014 Ken Youens-Clark

This module is free software; you can redistribute it and/or
modify it under the terms of the GPL (either version 1, or at
your option, any later version) or the Artistic License 2.0.
Refer to LICENSE for the full license text and to DISCLAIMER for
additional warranty disclaimers.

=cut
```

After I run this, I have a complete ORM that I can use to avoid writing SQL (structure query language). This lets me interact with the database using objects that abstract the nitty-gritty of database-specific SQL. E.g., I could easily move from MySQL to PostgreSQL.

Mojolicious

Using SQL::Translator and DBIx::Class, I avoid writing pretty much anything to do with data access. Now to employ the same level of laziness to avoid writing a website, I'll use Mojolicious (http://search.cpan.org/dist/Mojolicious). I tend to write in typical REST (https://en.wikipedia.org/wiki/Representational_state_transfer) fashion, so "/object/verb" (e.g., "/client/list" or "/client/view/:client_id"), and I use the MVC (model-view-controller) layout (https://en.wikipedia.org/wiki/Model-view-controller) where the Model is the database/ORM, the view is Template Toolkit template (http://search.cpan.org/dist/Template-Toolkit), and the Controller is a Mojolicious module.

In Mojolicious (in "mojo/lib/Compass.pm"), I define routes for each endpoint that point to "Compass::Controller::[Object]" classes. The methods in the classes correspond to the [verb]s that I define, and the template for the view is in "templates/[object]/[verb]." For example, the endpoint "/client/list" has a model of "Compass::Controller::Client::list" and the view is controlled with "/templates/client/list."

Helper modules

In "lib", you'll find a few helper modules I create to abstract the configuration (Compass::Config) and database access (Compass::DB). I like to use Moose (http://search.cpan.org/dist/Moose) as uses a declarative syntax very much in the style of Perl 6 to create object attributes and methods. All the "Schema" code in "lib/Compass" is the parts that are generated by the above schema dumper.

Interface

For the user interface, I like to use Bootstrap (http://getbootstrap.com) for CSS (cascading style sheets) and a smattering of jQuery (https://jquery.com) and other modules like Google Maps for interactivity. I tend to stick with the simplest possible forms and views so that pages load quickly; also, I'm usually in a hurry to write these things, so I often just re-use whatever I wrote before. If I need some

sort of user-generated documentation site, I'll probably create a separate site like "docs.foo.org" install Wordpress or Drupal. Whatever might constitute my style is evident on sites like Gramene.org and imicrobe.us.

Infrastructure

I used Apache as my web server for the longest time, but in the last couple of years I only use Nginx. Perhaps accidentally creating an open proxy with Apache and then getting absolutely hammered for a few weeks soured me on Apache. My servers are usually CentOS, and I rely a fair amount on pre-built packages for most things I need with the exception of Perl which I like to compile from source and stay on the very latest release as much as possible. It's nice to use virtual machines to segregate services, e.g., the databases (MySQL), the web servers (Nginx, PHP, Perl), search engines (MongoDB, esp., as it will eat pretty much all available resources).

Problem Sets

Be sure to check out https://github.com:kyclark/abe487.git for the tests!

Hello!

Alter the greeting script to accept the flag "--excited" that adds an exclamation point to the end of the greeting:

```
$ ./greet.pl6 --excited --greeting="Hola" --name="Amigo"
Hola, Amigo!
```

Sum and mean

Create a script that reads in numbers from the command line and prints out their sum and mean.

Clean sequences

Write a bash script to clean up some raw sequence.

Create a FASTA file

Create a script called "txt2fasta.pl6" that accepts an input file of sequences, one-per-line, and emits FASTA-formatted sequences. Sequence IDs should be an incrementing integer value starting at 1. Extra credit: block the sequences to a maximum column width, default 50.

FASTA stats

Given one or more input files of sequences in FASTA format, recreate this output:

```
$ seqmagick info mouse.fa | awk '{print $1,$3,$4,$5,$6}' | colum
n -t
name      min_len  max_len  avg_len  num_seqs
mouse.fa  50       100      84.32    500
```

Compute GC content

Solve the GC content problem on Rosalind (http://rosalind.info/problems/gc/). Then use that program to profile FASTA files and predict species.

Find motifs

Solve http://rosalind.info/problems/subs/ to find motifs in strings. Use this to find ORFs.

Compute Hamming

Solve http://rosalind.info/problems/hamm/ to find the number of mutations (SNPs/SNVs) between two sequences. Use this to determine sequence similarity.

Protein translation

Solve http://rosalind.info/problems/prot/. Read the translation table from the given "table.txt."

Shared k-mers

Create a program that will find the number of shared k-mers of a given size among a set of sequences in a FASTA file. Use this to determine sequence similarity. You can use the "fasta-kmer" program to create a list of k-mers in the given sequences.

HPC

HPC is an acronym for "high-performance computing," and it generally means using a cluster of computers. Our students have access to several clusters (Ocelote, HPC, ICE) at the University of Arizona, and most anyone is welcome to use the clusters at TACC. To use a cluster, it's necessary to submit a batch job along with a description of the resources you need (e.g., memory, number of CPUs, number of nodes) to a scheduler that will start your job when the resources become available. We will discuss schedulers "PBS" used at UA and "SLURM" used at TACC. In the Github repo, you will find an "hpc" directory that contains examples for submitting to each queue.

To interact PBS and SLURM, you must log in to the "head" node(s). Often you will be placed on a random nodes such as "login1." YOU ARE NOT ALLOWED TO DO HEAVY LIFTING ON THE HEAD NODE. For our class, you can write files, interact with the Perl RELP, run small scripts, etc., but you should never run BLAST or launch long-running jobs on these machines. They are intended to be used to submit jobs to the queue.

Handy aliases

To make it easier to go back and forth between PBS and SLURM, I create aliases so that I can execute the same command on both systems:

```
alias qstat="/usr/local/bin/qstat_local"
ME="kyclark"
alias qs="qstat -u $ME"
alias qt="qstat -Jtu $ME"
function qkill() {
  if [[ "${#1}" -eq 0 ]]; then
    echo {Now I crush you!"
    OUT=$(qstat -u $ME | grep $ME | cut -f 1 -d ' ' | sed 's/\[\
]\..*/[]/' | xargs qdel)

    if [[ $? -eq 0 ]]; then
        echo "Jobs killed"
    else
        echo -e "\nError submitting job\n$OUT\n"
    fi
  else
    echo Argument = \"$1\"
    echo "This isn't the command you're looking for. I don't tak
e arguments"
  fi
}

function qr() {
  WHO=${1:-$ME}
  echo qstat for \"$WHO\"
  OUT=`qstat -Jtu $WHO | tail -n +6 | awk '{print $10}' | sort |
 uniq -c`
  if [ -n "$OUT" ]; then
      echo "$OUT"
  else
      echo No jobs currently running.
  fi
}
```

Stampede/SLURM

```
ME="kyclark"
alias qs='squeue -u $ME | column -t'
```

PBS

The University of Arizona's HPC cluster uses the "PBS Pro" (portable batch system) scheduler.

Here are some important links:

- hpc-consult@list.arizona.edu (email list for help)
- https://confluence.arizona.edu/display/UAHPC/HPC+Documentation
- http://rc.arizona.edu/hpc-htc/high-performance-computing-high-throughput-computing
- http://rc.arizona.edu/hpc-htc/using-systems/pbs-example

Allocations

Your "allocation" is how much compute time you are allowed on the cluster. Use the command `va` to view your allocation of compute hours, e.g.:

```
$ va
kyclark current allocation (remaining/encumbered/total):
-------------------------------------------
Group               standard                  qualified
bhurwitz            17215:23/00:00/108000:00    99310:56/72:00/1
00000:00
bh_admin            00:00/00:00/00:00    00:00/00:00/00:00
bh_dev              00:00/00:00/00:00    00:00/00:00/00:00
gwatts              12000:00/00:00/24000:00    00:00/00:00/00:00
mbsulli             228000:00/00:00/228000:00    00:00/00:00/00:
00
```

The UA has three queues: high-priority, normal, and windfall. If you exhaust your normal hours in a month, then your jobs must run under "windfall" (catch as catch can) until your hours are replenished.

Job submission

The PBS command for submitting to the queue is `qsub`. Since this command takes many arguments, I usually write a small script to gather all the arguments and execute the command so it's documented how I ran the job. Most of the time I call this "submit.sh" it basically does `qsub $ARGS run.sh`. To view your queue, use `qstat -u $USER`.

Hello

Here is a "hello" script:

```
$ cat -n hello.sh
     1     #!/bin/bash
     2
     3     #PBS -W group_list=bhurwitz
     4     #PBS -q standard
     5     #PBS -l jobtype=cluster_only
     6     #PBS -l select=1:ncpus=1:mem=1gb
     7     #PBS -l walltime=01:00:00
     8     #PBS -l cput=01:00:00
     9
    10     echo "Hello from sunny \"$(hostname)\"!"
```

The `#PBS` lines almost look like comments, but they are directives to PBS to describe your job. Lines 6-7 says that we require a very small machine with just one CPU and 1G of memory and that we only want it for 1 hour. The less you request, the more likely you are to get a machine meeting (or exceeding) your needs. On line 10, we are including the `hostname` of the compute node so that we can see that, though we submit the job from a head node (e.g., "login1"), the job is run on a different machine.

Here is a Makefile to submit it:

```
$ cat -n Makefile
     1     submit: clean
     2         qsub hello.sh
     3
     4     clean:
     5         find . -name hello.sh.[eo]* -exec rm {} \;
```

Just typing `make` will run the "clean" command to remove any previous out/error files, and this it will `qsub` our "hello.sh" script:

```
$ make
find . -name hello.sh.[eo]* -exec rm {} \;
qsub hello.sh
818089.service0
$ type qs
qs is aliased to `qstat -u kyclark'
$ qs

service0:
                                                                  Req'
d  Req'd   Elap
Job ID          Username Queue    Jobname    SessID NDS TSK Memo
ry Time  S Time
--------------- -------- -------- ---------- ------ --- --- ----
-- ----- - -----
818089.service0 kyclark  clu_stan hello.sh      --    1   1    1
gb 01:00 Q   --
```

Until the job is picked up, the "S" (status) column will show "Q" for "queued," then it will change to "R" for "running," "E" for "error," or "X" for "exited." When `qstat` returns nothing, then the job has finished. You should see files like "hello.sh.o[jobid]" for the output and "hello.sh.e[jobid]" for the errors (which we hope are none):

```
$ ls -lh
total 128K
-rw-rw-r-- 1 kyclark staff     210 Aug 30 10:09 hello.sh
-rw------- 1 kyclark bhurwitz    0 Aug 30 10:19 hello.sh.e818089
-rw------- 1 kyclark bhurwitz  181 Aug 30 10:19 hello.sh.o818089
-rw-rw-r-- 1 kyclark staff      81 Aug 30 10:08 Makefile
-rw-rw-r-- 1 kyclark staff    1.3K Aug 26 08:12 README.md
$ cat hello.sh.o818089
Hello from sunny "r1i3n10"!
Your group bhurwitz has been charged 00:00:01 for 1 cpus.
You previously had 42575:01:07.  You now have 42575:01:06 remain
ing for the queue clu_standard
```

FTP

While Makefiles can be a great way to document for myself (and others) how I submitted and ran a job, I will often write a "submit.sh" script to check input, decide on resources, etc. Here is a more complicated submission for retrieving data from an FTP server:

```
$ cat -n submit.sh
     1     #!/bin/bash
     2
     3     set -u
     4
     5     export FTP_LIST=get-me
     6     export OUT_DIR=$PWD
     7     export NCFTPGET=/rsgrps/bhurwitz/hurwitzlab/bin/ncftp
get
     8     export PBSDIR=pbs
     9
    10     if [[ -d $PBSDIR ]]; then
    11       rm -rf $DIR/*
    12     else
    13       mkdir $DIR
    14     fi
    15
    16     NUM_FILES=$(wc -l $FTP_LIST | cut -d ' ' -f 1)
    17
    18     if [[ $NUM_FILES -gt 0 ]]; then
    19       JOB_ID=$(qsub -N ftp -v OUT_DIR,FTP_LIST,NCFTPGET -
j oe -o $PBSDIR ftp-get.sh)
    20       echo "Submitted \"$FILE\" files to job \"$JOB_ID\""
    21     else
    22       echo "Can't find any files in \"$FTP_LIST\""
    23     fi
$ cat get-me
ftp://ftp.imicrobe.us/projects/33/CAM_PROJ_HumanGut.asm.fa.gz
ftp://ftp.imicrobe.us/projects/121/CAM_P_0001134.csv.gz
ftp://ftp.imicrobe.us/projects/66/CAM_PROJ_TwinStudy.csv.gz
```

Here I have a file "get-me" with a few files on an FTP server that I want to download using the program "ncftpget" (http://ncftp.com/) which is installed in our shared "bin" directory. Since I don't like having the output files from PBS scattered about my working directory, I like to make a place ("pbs") to put them (lines 8-14), and then I include the "-j oe" flag to "join output/error" files together and "-o" to put

the output files in $PBSDIR. On line 16, I check that there is legitimate input from the user. Line 19 captures the output from the `qsub` command to report on the submission.

One way to pass arguments to the compute node is by `export`ing variables (lines 5-8) and then using the "-v" option to send those parts of the environment with the job. If you ever get an error on `qsub` that say it can't send the environment, it's because you failed to `export` the variable.

Here is the script that actually downloads the files:

```
$ cat -n ftp-get.sh
     1     #!/bin/bash
     2
     3     #PBS -W group_list=bhurwitz
     4     #PBS -q standard
     5     #PBS -l jobtype=serial
     6     #PBS -l select=1:ncpus=2:mem=4gb
     7     #PBS -l place=pack:shared
     8     #PBS -l walltime=24:00:00
     9     #PBS -l cput=24:00:00
    10
    11     set -u
    12
    13     cd $OUT_DIR
    14
    15     echo "Started $(date)"
    16
    17     i=0
    18     while read FTP; do
    19       let i++
    20       printf "%3d: %s\n" $i $FTP
    21       $NCFTPGET $FTP
    22     done < $FTP_LIST
    23
    24     echo "Ended $(date)"
```

All of the "#PBS" directives in this script could also have been specified as options to the `qsub` command in the submit script. Even though I have "set -u" on and have not declared `$OUT_DIR`, I can `cd` to it because it was exported from the submit script. When your job is placed on the compute node, it will be placed into your $HOME directory, so it's important to have your job place its output files into the correct location. The rest of the script is fairly self-explanatory, reading the `$FTP_LIST` one line at a time, using "ncftpget" to fetch it ("wget" would work just fine, too).

Job Arrays

Downloading files doesn't usually take a long time, but for our purposes let's pretend each file would take upwards of 10 hours. We are only allowed 24 hours on a compute node, so we think we can fetch at most two files for each job. If we have 200 files, then we need 100 jobs which exceeds the polite and allowed number of jobs we can put into queue at any one time. This is when we would use a job array to submit just one job that will be turned into the required 100 jobs to handle the 200 files:

```
$ cat -n submit.sh
     1     #!/bin/bash
     2
     3     set -u
     4
     5     export FTP_LIST=get-me
     6     export OUT_DIR=$PWD
     7     export NCFTPGET=/rsgrps/bhurwitz/hurwitzlab/bin/ncftp
get
     8     export PBSDIR=pbs
     9     export STEP_SIZE=2
    10
    11     if [[ -d $PBSDIR ]]; then
    12       rm -rf $DIR/*
    13     else
    14       mkdir $DIR;
    15     fi
    16
    17     NUM_FILES=$(wc -l $FTP_LIST | cut -d ' ' -f 1)
    18
    19     if [[ $NUM_FILES -lt 1 ]]; then
    20       echo "Can\'t find any files in \"$FTP_LIST\""
    21       exit 1
    22     fi
    23
    24     JOBS=""
    25     if [[ $NUM_FILES -gt 1 ]]; then
    26       JOBS="-J $NUM_FILES"
    27       if [[ $STEP_SIZE -gt 1 ]]; then
    28         JOBS="$JOBS:$STEP_SIZE"
    29       fi
    30     fi
    31
    32     JOB_ID=$(qsub $JOBS -N ftp -v OUT_DIR,STEP_SIZE,FTP_L
IST,NCFTPGET -j oe -o $PBSDIR ftp-get.sh)
    33
    34     echo "Submitted \"$FILE\" files to job \"$JOB_ID\""
```

The only difference from the previous version is that we have a new `$STEP_SIZE` variable set to "2" meaning we want to handle 2 jobs per node. Lines 24-30 build up the a string to describe the job array which will only be needed if there is more than 1 job.

The FTP downloading now needs to take into account which files to download from the `$FTP_LIST`

```
$ cat -n ftp-get.sh
     1     #!/bin/bash
     2
     3     #PBS -W group_list=bhurwitz
     4     #PBS -q standard
     5     #PBS -l jobtype=serial
     6     #PBS -l select=1:ncpus=1:mem=1gb
     7     #PBS -l place=pack:shared
     8     #PBS -l walltime=24:00:00
     9     #PBS -l cput=24:00:00
    10
    11     set -u
    12
    13     echo "Started $(date)"
    14
    15     cd $OUT_DIR
    16
    17     TMP_FILES=$(mktemp)
    18     sed -n "${PBS_ARRAY_INDEX:-1},${STEP_SIZE:-1}" $FTP_F
ILES > $TMP_FILES
    19     NUM_FILES=$(wc -l $TMP_FILES | cut -d ' ' -f 1)
    20
    21     if [[ $NUM_FILES -lt 1 ]]; then
    22       echo "Failed to fetch files"
    23       exit 1
    24     fi
    25
    26     echo "Will fetch $NUM_FILES"
    27
    28     i=0
    29     while read FTP; do
    30       let i++
    31       printf "%3d: %s\n" $i $FTP
    32       $NCFTPGET $FTP
    33     done < $TMP_FILES
    34
    35     rm $TMP_FILES
    36
    37     echo "Ended $(date)"
```

To extract the files for the given compute node, we use the `$PBS_ARRAY_INDEX` variable created by PBS along with the `$STEP_SIZE` variable as arguments to a `sed` command, redirecting that output into a temporary file. From there, the script proceeds as before only reading from the `$TMP_FILES` and removing it when the job is done.

Interactive job

You can use `qsub -I` flag to be placed onto a compute node to run your job interactively. This is a good way to debug your script in the actual runtime environment. TACC has a nifty alias called `idev` that will fire up an interactive node for you to play with, so here is a PBS version to do the same. Place this line in your "~/.bashrc" (be sure to "source" the file afterwards):

```
alias idev="qsub -I -N idev -W group_list=bhurwitz -q standard -
l walltime=01:00:00 -l select=1:ncpus=1:mem=1gb"
```

Then from a login node (here "service2") I can type `idev` to get a compute node. When I'm finished, I can CTRL-D or type "exit" or "logout" to go back to the login node:

```
$ hostname
service2
$ idev
qsub: waiting for job 652560.service2 to start
qsub: job 652560.service2 ready

$ hostname
htc50
$ logout

qsub: job 652560.service2 completed
```

Dependency Chain

To schedule a job with a dependency chain, it's necessary to be on the correct login/head node for the type of job:

- service0 - cluster scheduler
- service1 - smp scheduler
- service2 - htc scheduler

SLURM

SLURM's command for queue submission is `sbatch`, and `showq` will show you your queue. Compute nodes are shared by default. You must request exclusive access if you need.

- hpc-consult@list.arizona.edu is the help account

TACC/Stampede

- TACC is part of the XSEDE (xsede.org) project.
- TACC does not allow the use of job arrays on their clusters. Instead, they have written their "parametric launcher" (https://www.tacc.utexas.edu/research-development/tacc-software/the-launcher).
- Your three important directories are `$HOME`, `$WORK`, and `$SCRATCH`, and they can be accessed with `cd`, `cdw`, and `cds`, respectively.
- Compute nodes are not shared

SLURM Hello

Here is our "hello" script modified from PBS to SLURM:

```
$ cat -n hello.sh
     1     #!/bin/bash
     2
     3     #SBATCH -A iPlant-Collabs
     4     #SBATCH -p development # or "normal"
     5     #SBATCH -t 01:00:00
     6     #SBATCH -N 1
     7     #SBATCH -n 1
     8     #SBATCH -J hello
     9     #SBATCH --mail-user=kyclark@email.arizona.edu
    10     #SBATCH --mail-type=BEGIN,END,FAIL
    11
    12     echo "Hello from sunny \"$(hostname)\"!"
```

As with the "#PBS" directives, we have "#SBATCH" to describe the job and resources. Most important for TACC is the "-A" allocation argument that decides which account will be charge for the compute time. For the "-p" partition, I can choose either "normal" or "development," the latter of which allows me a maximum of two hours. The idea is that your job gets picked up relatively quickly, which makes it much faster to test new code. The "-J" here is not "job array" (those are not allowed on stampede) but the job name, and I also threw in the options to email me when the job starts and stops.

The Makefile is pretty similar to before. The command "make" will run "clean" and then "sbatch" for us. The "qs" alias shows the job in "R" running state and the "CG" for "completing." The standard error and output go into "slurm-[jobid]" files.

```
$ cat -n Makefile
     1     submit: clean
     2         sbatch hello.sh
     3
     4     clean:
     5         find . -name slurm-\* -exec rm {} \;
$ make
find . -name slurm-\* -exec rm {} \;
sbatch hello.sh
-----------------------------------------------------------------
-
```

```
              Welcome to the Stampede Supercomputer
--------------------------------------------------------------------
-

No reservation for this job
--> Verifying valid submit host (login4)...OK
--> Verifying valid jobname...OK
--> Enforcing max jobs per user...OK
--> Verifying availability of your home dir (/home1/03137/kyclar
k)...OK
--> Verifying availability of your work dir (/work/03137/kyclark
)...OK
--> Verifying valid ssh keys...OK
--> Verifying access to desired queue (development)...OK
--> Verifying job request is within current queue limits...OK
--> Checking available allocation (iPlant-Collabs)...OK
Submitted batch job 7560143
$ type qs
qs is aliased to `squeue -u kyclark | column -t'
$ qs
JOBID    PARTITION    NAME   USER     ST  TIME  NODES  NODELIST(
REASON)
7560143  development  hello  kyclark  R   0:00  1      c557-904
$ qs
JOBID    PARTITION    NAME   USER     ST  TIME  NODES  NODELIST(
REASON)
7560143  development  hello  kyclark  CG  0:08  1      c557-904
$ ls -l
total 16
-rw------- 1 kyclark G-814141  262 Aug 30 12:56 hello.sh
-rw------- 1 kyclark G-814141   77 Aug 30 13:01 Makefile
-rw------- 1 kyclark G-814141 1245 Aug 30 12:52 README.md
-rw------- 1 kyclark G-814141   54 Aug 30 13:09 slurm-7560143.ou
t
[tacc:login4@/work/03137/kyclark/metagenomics-book/hpc/slurm/hel
lo]$ cat slurm-7560143.out
Hello from sunny "c557-904.stampede.tacc.utexas.edu"!
```

Hello

Let's use our familiar "Hello, World!" to get started:

```
$ cat -n hello.py
     1    #!/usr/bin/env python3
     2
     3    print('Hello, World!')
```

The first thing to notice is a change to the "shebang" line. I'm going to use `env` to find `python3` so I won't have a hard-coded path that my user will have to change. In bash, we could use either `echo` or `printf` to print to the terminal (or a file). In Python, we have `print()` noting that we must use parentheses now to invoke functions. (One difference between versions 2 and 3 of Python was that the parens to `print` were not necessary in version 2).

Variables

Let's use the REPL to play:

```
$ python3
Python 3.6.1 |Anaconda custom (x86_64)| (default, May 11 2017, 1
3:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more inform
ation.
>>> name = 'Gorgeous'
>>> print('Hello, ' + name)
Hello, Gorgeous
```

Here I'm showing that it's easy to create a variable called `name` which we assign the value "Gorgeous." Just as in bash, we can use it in a `print` statement, but we can't directly stick it into the string:

```
>>> print('Hello, name')
Hello, name
```

We have to use the `+` operator to concatenate it to the literal string "Hello, ":

```
>>> print('Hello, ' + name)
Hello, Gorgeous
```

We can also pass a list of arguments to `print`, but notice the extra space:

```
>>> print('Hello, ', name)
Hello,  Gorgeous
```

So, we've just found that Python will automatically put a space between all the arguments:

```
>>> print('foo', 'bar', 'baz')
foo bar baz
```

Arguments

To say hello to an argument passed from the command line, we need to a module which is just a package of code we can use:

```
$ cat -n hello_arg.py
     1    #!/usr/bin/env python3
     2
     3    import sys
     4
     5    args = sys.argv
     6    print('Hello, ' + args[1] + '!')
```

From the `sys` module, we call the `argv` function to get the "argument vector." This is a list, and, like bash, the name of the script is in the zeroth position (`args[0]`), so the first "argument" to the script is in `args[1]` . It works as you would expect:

```
$ ./hello_arg.py Sally
Hello, Sally!
```

But there is a problem if we fail to pass any arguments:

```
$ ./hello_arg.py
Traceback (most recent call last):
  File "./hello_arg.py", line 6, in <module>
    print('Hello, ' + args[1] + '!')
IndexError: list index out of range
```

We tried to access something in `args` that doesn't exist, and so the entire program came to a halt ("crashed"). As in bash, we need to check how many arguments we have:

```
$ cat -n hello_arg2.py
     1    #!/usr/bin/env python3
     2
     3    import sys
     4
     5    args = sys.argv
     6
     7    if len(args) < 2:
     8        print('Usage:', args[0], 'NAME')
     9        sys.exit(1)
    10
    11    print('Hello, ' + args[1] + '!')
```

If there are fewer than 2 arguments (remembering that the script name is in the "first" position), then we print a usage statement and use `sys.exit` to send the operating system a non-zero exit status, just like in bash. It works much better now:

```
$ ./hello_arg2.py
Usage: ./hello_arg2.py NAME
$ ./hello_arg2.py Sally
Hello, Sally!
```

On line 7 above, you see we can use the `len` function to ask how long the `args` list is. You can play with the Python REPL to understand `len`. Both strings (like "foobar") and lists (like the arguments to our script) have a "length." Type `help(list)` in the REPL to read the docs on lists.

```
>>> len('foobar')
6
>>> len(['foobar'])
1
>>> len(['foo', 'bar'])
2
```

Here is the same functionality but using two new functions, `printf` (from the base package) and `os.path.basename`:

```
$ cat -n hello_arg3.py
     1    #!/usr/bin/env python3
     2    """hello with args"""
     3
     4    import sys
     5    import os
     6
     7    args = sys.argv
     8
     9    if len(args) != 2:
    10        script = os.path.basename(args[0])
    11        print('Usage: {} NAME'.format(script))
    12        sys.exit(1)
    13
    14    name = args[1]
    15    print('Hello, {}!'.format(name))
$ ./hello_arg3.py
Usage: hello_arg3.py NAME
$ ./hello_arg3.py Sally
Hello, Sally!
```

Notice the usage doesn't have a "./" on the script name because we used `basename` to clean it up.

main()

Lastly, let me introduce the `main` function. Many languages (e.g., Python, Perl, Rust, Haskell) have the idea of a "main" module/function where all the processing starts. If you define a "main" function, most people reading your code would understand that the program ought to begin there. I usually put my "main" as the first `def` (the keyword to "define" a function), and then use a little test at the end of the program to see if the magical "double-under" `name` is equal to the string double-under "main." It's a bit of a hack, but it seems to be standard Python.

```
$ cat -n hello_arg4.py
     1    #!/usr/bin/env python3
     2    """hello with args/main"""
     3
     4    import sys
     5    import os
     6
     7    def main():
     8        """main"""
     9        args = sys.argv
    10
    11        if len(args) != 2:
    12            script = os.path.basename(args[0])
    13            print('Usage: {} NAME'.format(script))
    14            sys.exit(1)
    15
    16        name = args[1]
    17        print('Hello, {}!'.format(name))
    18
    19    if __name__ == '__main__':
    20        main()
```

Function Order

Note that you cannot put lines 19-20 first because you cannot call a function that hasn't been defined (lexically) in the program yet. To add insult to injury, this is a **run-time error** -- meaning the mistake isn't caught by the compiler when the program is parsed into byte-code; instead the program just crashes.

```
$ cat -n func-def-order.py
     1    #!/usr/bin/env python3
     2
     3    print('Starting the program')
     4    foo()
     5    print('Ending the program')
     6
     7    def foo():
     8        print('This is foo')
$ ./func-def-order.py
Starting the program
Traceback (most recent call last):
  File "./func-def-order.py", line 4, in <module>
    foo()
NameError: name 'foo' is not defined
```

To contrast:

```
$ cat -n func-def-order2.py
     1    #!/usr/bin/env python3
     2
     3    def foo():
     4        print('This is foo')
     5
     6    print('Starting the program')
     7    foo()
     8    print('Ending the program')
$ ./func-def-order2.py
Starting the program
This is foo
Ending the program
```

Handle All The Args!

If we like, we can say hi to any number of names:

```
$ cat -n hello_arg5.py
     1    #!/usr/bin/env python3
     2    """hello with to many"""
     3
     4    import sys
     5    import os
     6
     7    def main():
     8        """main"""
     9        args = sys.argv
    10
    11        if len(args) < 2:
    12            script = os.path.basename(args[0])
    13            print('Usage: {} NAME [NAME2 ...]'.format(scri
pt))
    14            sys.exit(1)
    15
    16        print('Hello, {}!'.format(', '.join(args[1:])))
    17
    18    if __name__ == '__main__':
    19        main()
$ ./hello_arg5.py foo
Hello, foo!
$ ./hello_arg5.py foo bar baz
Hello, foo, bar, baz!
```

Look at line 16 to see how we can `join` all the arguments on a comma-space, e.g.,:

```
>>> ', '.join(['foo', 'bar', 'baz'])
'foo, bar, baz'
>>> ':'.join("hello")
'h:e:l:l:o'
```

Notice the second example where we can treat a string like a list of characters.

The other interesting bit on line 16 is how to take a slice of a list. We want all the elements of `args` starting at position 1, so `args[1:]`. You can indicate a start and/or end position. It's best to play with it to understand:

```
>>> x = ['foo', 'bar', 'baz']
>>> x[1]
'bar'
>>> x[1:]
['bar', 'baz']
>>> a = "abcdefghijklmnopqrstuvwxyz"
>>> a[2:4]
'cd'
>>> a[:3]
'abc'
>>> a[3:]
'defghijklmnopqrstuvwxyz'
>>> a[-1]
'z'
>>> a[-3]
'x'
>>> a[-3:]
'xyz'
>>> a[-3:26]
'xyz'
>>> a[-3:27]
'xyz'
```

Conditionals

Above we saw a simple `if` condition, but what if you want to test for more then one condition? Here is a program that shows you how to take input directly from the user:

```
$ cat -n if-else.py
     1    #!/usr/bin/env python3
     2    """conditions"""
     3
     4    name = input('What is your name? ')
     5    age = int(input('Hi, ' + name + '. What is your age? '
))
     6
     7    if age < 0:
     8        print("That isn't possible.")
     9    elif age < 18:
    10        print('You are a minor.')
    11    else:
    12        print('You are an adult.')
$ ./if-else.py
What is your name? Geoffrey
Hi, Geoffrey. What is your age? 47
You are an adult.
```

On line 4, we can put the first answer into the `name` variable; however, on line 5, I convert the answer to an integer with `int` because I will need to compare it numerically, cf:

```
>>> 4 < 5
True
>>> '4' < 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: str() < int()
>>> int('4') < 5
True
```

Types

Which leads into the notion that Python, unlike bash, has types -- variables can hold string, integers, floating-point numbers, lists, dictionaries, and more:

```
>>> type('foo')
<class 'str'>
>>> type(4)
<class 'int'>
>>> type(3.14)
<class 'float'>
>>> type(['foo', 'bar'])
<class 'list'>
>>> type(range(1,3))
<class 'range'>
>>> type({'name': 'Geoffrey', 'age': 47})
<class 'dict'>
```

As noted earlier, you can use `help` on any of the class names to find out more of what you can do with them.

So let's return to the `+` operator earlier and check out how it works with different types:

```
>>> 1 + 2
3
>>> 'foo' + 'bar'
'foobar'
>>> '1' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not int
```

Python will crash if you try to "add" two different types together, but the type of the argument depends on the run-time conditions:

```
>>> x = 4
>>> y = 5
>>> x + y
9
>>> z = '1'
>>> x + z
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

To avoid such errors, you can coerce your data:

```
>>> int(x) + int(z)
5
```

Or check the types at run-time:

```
>>> for pair in [(1, 2), (3, '4')]:
...     n1, n2 = pair[0], pair[1]
...     if type(n1) == int and type(n2) == int:
...         print('{} + {} = {}'.format(n1, n2, n1 + n2))
...     else:
...         print('Cannot add {} ({}) and {} ({})'.format(n1, ty
pe(n1), n2, type(n2)))
...
1 + 2 = 3
Cannot add 3 (<class 'int'>) and 4 (<class 'str'>)
```

Loops

As in bash, we can use `for` and `while` loops in Python. Here's another way to greet all the people:

```
$ cat -n hello_arg6.py
     1    #!/usr/bin/env python3
     2    """hello with to many"""
     3
     4    import sys
     5    import os
     6
     7    def main():
     8        """main"""
     9        args = sys.argv
    10
    11        if len(args) < 2:
    12            script = os.path.basename(args[0])
    13            print('Usage: {} NAME [NAME2 ...]'.format(scri
pt))
    14            sys.exit(1)
    15
    16        for name in args[1:]:
    17            print('Hello, ' + name + '!')
    18
    19    if __name__ == '__main__':
    20        main()
$ ./hello_arg6.py Jack Jill
Hello, Jack!
Hello, Jill!
```

You can see more in the REPL:

```
>>> for letter in "abc":
...     print(letter)
...
a
b
c
>>> for number in range(0, 5):
...     print(number)
...
0
1
2
3
4
>>> for word in ['foo', 'bar']:
...     print(word)
...
foo
bar
>>> for word in 'We hold these truths'.split():
...     print(word)
...
We
hold
these
truths
>>> for line in open('input1.txt'):
...     print(line, end='')
...
this is
some text
from a file.
```

In each case, we're iterating over the members of a list as produced from a string, a range, an actual list, a list produced by a function, and an open file, respectively. (That last example either needs to suppress the newline from `print` or do `rstrip()` on the line to remove it as the text coming from the file has a newline.)

Strings, Lists, and Tuples

There's some overlap among Python's strings, lists, and tuples. In a way, you could think of strings as lists of characters. Many list operations work exactly the same over strings like subscripting to get a particular item:

```
>>> names = ['Larry', 'Moe', 'Curly', 'Shemp']
>>> names[0]
'Larry'
>>> 'Curly'[0]
'C'
```

"Slice" operations let you take a range of items:

```
>>> names[2:4]
['Curly', 'Shemp']
>>> 'Curly'[2:4]
'rl'
```

Functions like `join` that take lists can also work on strings:

```
>>> ', '.join(names)
'Larry, Moe, Curly, Shemp'
>>> ', '.join(names[0])
'L, a, r, r, y'
```

You can ask if a list contains a certain member, and you can also ask if a string contains a certain character or substring:

```
>>> 'Moe' in names
True
>>> 'r' in 'Larry'
True
>>> 'url' in 'Curly'
True
>>> 'x' in 'Larry'
False
>>> 'Joe' in names
False
```

You can iterate with a `for` loop over both the items in a list or the characters in a word:

```
>>> names = ['Larry', 'Moe', 'Curly', 'Shemp']
>>> for name in names:
...   print(name)
...
Larry
Moe
Curly
Shemp
>>> for letter in 'Curly':
...   print(letter)
...
C
u
r
l
y
```

Another very useful function called `enumerate` takes a list/string and returns the index/position along with the item/character:

```
>>> for i, name in enumerate(names):
...   print('{:3} {}'.format(i, name))
...
  0 Larry
  1 Moe
  2 Curly
  3 Shemp
>>> for i, letter in enumerate('Curly'):
...   print('{:3} {}'.format(i, letter))
...
  0 C
  1 u
  2 r
  3 l
  4 y
```

For an example, here is a simple program to determine if a given string is a palindrome:

```
$ cat -n word_is_palindrome.py
     1    #!/usr/bin/env python3
     2    """Report if the given word is a palindrome"""
     3
     4    import sys
     5    import os
     6
     7    args = sys.argv[1:]
     8
     9    if len(args) != 1:
    10        print('Usage: {} STR'.format(os.path.basename(args
[0])))
    11        sys.exit(1)
    12
    13    word = args[0]
    14    rev = ''.join(reversed(word))
    15    print('"{}" is{} a palindrome.'.format(word, '' if wor
d.lower() == rev.lower() else ' NOT'))
```

As we discussed earlier, `sys.argv` returns exactly what the operating system thinks of as "the program" it's running, namely that the program name is in the first (zeroth) position, and anything else you type on the command line follows. If you run this as `./word_is_palindrome.py foo` then `sys.argv` looks like `['./word_is_palindrome.py', 'foo']`. While discussing this with a student, I realized the confusion over the program name being in the `[0]` position, so rather than doing:

```
args = sys.argv
```

I think it makes more sense to have you do:

```
args = sys.argv[1:]
```

Then you really are only dealing with the arguments to the script, and you can say more logical things like:

```
if len(args) == 0:
    print('Usage: blah blah blah')
    sys.exit(1)
```

Note that Python will throw an exception if you try to reference an index position in a list that doesn't exist:

```
>>> 'foo'[0]
'f'
>>> 'foo'[10]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Python will not blow up if you take a slice of an array starting or ending at non-existent positions:

```
>>> 'foo'[1:10]
'oo'
>>> 'foo'[5:]
''
```

Which is why it's safe to say `sys.argv[1:]` to slice out everything starting at position 1 even if there is nothing there.

We can expand our palindrome program to one that searches in a file:

```
$ cat -n find_palindromes.py
     1    #!/usr/bin/env python3
     2    """Report if the given word is a palindrome"""
     3
     4    import sys
     5    import os
     6
     7    args = sys.argv[1:]
     8
     9    if len(args) != 1:
    10        print('Usage: {} FILE'.format(os.path.basename(sys
.argv[0])))
    11        sys.exit(1)
    12
    13    file = args[0]
    14
    15    if not os.path.isfile(file):
    16        print('"{}" is not a file'.format(file))
    17        sys.exit(1)
    18
    19    for line in open(file):
    20        for word in line.lower().split():
    21            if len(word) > 2:
    22                rev = ''.join(reversed(word))
    23                if rev == word:
    24                    print(word)
```

Lines 19-20 read each `line` and then lowercase and `split` (on spaces) into each `word`. You could compress this like so (see "find_palindromes2.py"):

```
for word in open(file).read().lower().split():
```

This will call `read` on the opened file handle to bring the entire file contents into memory, lowercase, and `split` into words. The first way is probably more efficient with memory, but you will likely see files being read. Another common idiom to read all the lines of a file (and remove the newlines!) is:

```
all_lines = open(file).read().splitlines()
```

Tetranucleotide Composition

A common operation in bioinformatics is to determine sequence composition. Here is a program to find the frequencies of the DNA bases (A, C, T, G):

```
$ cat -n dna1.py
     1    #!/usr/bin/env python3
     2    """Tetra-nucleotide counter"""
     3
     4    import sys
     5    import os
     6
     7    args = sys.argv[1:]
     8
     9    if len(args) != 1:
    10        print('Usage: {} DNA'.format(os.path.basename(sys.
argv[0])))
    11        sys.exit(1)
    12
    13    dna = args[0]
    14
    15    count_a, count_c, count_g, count_t = 0, 0, 0, 0
    16
    17    for letter in dna:
    18        if letter == 'a' or letter == 'A':
    19            count_a += 1
    20        elif letter == 'c' or letter == 'C':
    21            count_c += 1
    22        elif letter == 'g' or letter == 'G':
    23            count_g += 1
    24        elif letter == 't' or letter == 'T':
    25            count_t += 1
    26
    27    print(' '.join([str(count_a), str(count_c), str(count_
g), str(count_t)]))
$ ./dna1.py AACCTAG
3 2 1 1
```

On line 15, we initiate four variables to count each DNA base. Just as we can use a `for` loop to iterate through a list, we can iterate through each letter in a string on line 17. We need to check for both upper- and lowercase strings to determine which counter to increment. Line 27 points out that the "count_*" variables are numbers that must be converted to strings in order to `print` them.

To save quite a bit of typing, let's force the input sequence to lowercase:

```
$ cat -n dna2.py
     1    #!/usr/bin/env python3
     2    """Tetra-nucleotide counter"""
     3
     4    import sys
     5    import os
     6
     7    args = sys.argv[1:]
     8
     9    if len(args) != 1:
    10        print('Usage: {} DNA'.format(os.path.basename(sys.
argv[0])))
    11        sys.exit(1)
    12
    13    dna = args[0]
    14
    15    count_a, count_c, count_g, count_t = 0, 0, 0, 0
    16
    17    for letter in dna.lower():
    18        if letter == 'a':
    19            count_a += 1
    20        elif letter == 'c':
    21            count_c += 1
    22        elif letter == 'g':
    23            count_g += 1
    24        elif letter == 't':
    25            count_t += 1
    26
    27    print(' '.join([str(count_a), str(count_c), str(count_
g), str(count_t)]))
```

There are better ways than this to count the characters, but we'll save this until we talk about dictionaries.

Run-length Encoding

Along the lines of counting characters in a string, we can write a very simple string compression program that encodes repetitions of characters:

```
$ ./compress.py AAACAATTTTGGGGGAC
A3CA2T4G5AC
$ cat -n compress.py
     1    #!/usr/bin/env python3
     2    """Compress text/DNA by marking repeated letters"""
     3
     4    import os
     5    import sys
     6
     7    args = sys.argv[1:]
     8
     9    if len(args) != 1:
    10        print('Usage: {} ARG'.format(os.path.basename(sys.
argv[0])))
    11        sys.exit(1)
    12
    13    arg = args[0]
    14    text = ''
    15    if os.path.isfile(arg):
    16        text = ''.join(open(arg).read().split())
    17    else:
    18        text = arg.strip()
    19
    20    if len(text) == 0:
    21        print('No usable text')
    22        sys.exit(1)
    23
    24    counts = []
    25    count = 0
    26    prev = None
    27    for letter in text:
    28        if prev is None:
    29            prev = letter
    30            count = 1
    31        elif letter == prev:
    32            count += 1
    33            prev = letter
```

```
    34        else:
    35            counts.append((prev, count))
    36            count = 1
    37            prev = letter
    38
    39    # get the last letter after we fell out of the loop
    40    counts.append((prev, count))
    41
    42    for letter, count in counts:
    43        print('{}{}'.format(letter, '' if count == 1 else
count), end='')
    44
    45    print('')
```

Line 15 uses the `os.path.isfile` function to determine if the argument is a file; if so, line 16 uses the code from earlier to `split` the entire file into "words" and then `join` s them back together on the empty string. This would concatenate all sequence lines into one long sequence. If the argument is not a file, then we use `rstrip` to get rid of any spaces on the right-hand side.

This program makes use of a `counts` list to keep track of each letter we saw. We add a "tuple" to the list:

```
>>> counts = []
>>> counts.append(('A', 3))
>>> counts
[('A', 3)]
>>> counts.append(('C', 1))
>>> counts
[('A', 3), ('C', 1)]
```

Tuples are similar to lists, but they are immutable:

```
>>> tup = ('white', 'dog')
>>> tup[1]
'dog'
>>> tup[1] = 'cat'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

You see they are subscripted like strings and lists, but you cannot change a value inside a tuple. Tuples are not limited to pairs:

```
>>> tup = ('white', 'dog', 'bird')
>>> tup[-1]
'bird'
```

tac

We all know and love the venerable `cat` program, but do you know about `tac`? It prints a file in reverse. We can use lists in Python to read a file into list and `reverse` it:

```
$ cat input.txt
first line
second line
third line
fourth line
$ ./tac1.py input.txt
fourth line
third line
second line
first line
```

Here is the code:

```
$ cat -n tac1.py
     1    #!/usr/bin/env python3
     2    """print a file in reverse"""
     3
     4    import sys
     5    import os
     6
     7    args = sys.argv[1:]
     8    if len(args) != 1:
     9        print('Usage: {} FILE'.format(os.path.basename(sys
.argv[0])))
    10        sys.exit(1)
    11
    12    file = args[0]
    13    if not os.path.isfile(file):
    14        print('"{}" is not a file'.format(file))
    15        sys.exit(1)
    16
    17    lines = []
    18    for line in open(file):
    19        lines.append(line)
    20
    21    lines.reverse()
    22
    23    for line in lines:
    24        print(line, end='')
```

We initialize a new list on line 17, then read through the file line-by-line and call the `append` method to add the line to the end of our list. Then we call `reverse` to mutate the list **IN PLACE**:

```
>>> names
['Larry', 'Moe', 'Curly', 'Shemp']
>>> names.reverse()
>>> names
['Shemp', 'Curly', 'Moe', 'Larry']
```

After `reverse` we see that the `names` are permanently changed. We can put them back with another call:

```
>>> names.reverse()
>>> names
['Larry', 'Moe', 'Curly', 'Shemp']
```

If we had simply wanted to use them in a reversed order **WITHOUT ALTERING THE ACTUAL LIST**, we could call the `reversed` function:

```
>>> list(reversed(names))
['Shemp', 'Curly', 'Moe', 'Larry']
>>> names
['Larry', 'Moe', 'Curly', 'Shemp']
```

It's really easy to read an entire file directly into a list with `readlines` (this preserves newlines), but you should be sure that you have at least as much memory on your machine as the file is big. Compare these various ways to read an entire file. `read` will give you the contents as one string, and newlines will be present to denote the end of each line:

```
>>> open('input.txt').read()
'first line\nsecond line\nthird line\nfourth line\n'
```

Whereas `readlines` will return a list of strings broken on the newlines (but not removing them):

```
>>> open('input.txt').readlines()
['first line\n', 'second line\n', 'third line\n', 'fourth line\n
']
```

Calling `read().splitlines()` will suck in the whole file, then break on the newlines, removing them in the process:

```
>>> open('input.txt').read().splitlines()
['first line', 'second line', 'third line', 'fourth line']
```

Similarly, you can `read().split()` to break all the input on spaces to get the words:

```
>>> open('input.txt').read().split()
['first', 'line', 'second', 'line', 'third', 'line', 'fourth', '
line']
```

Here is a version that uses `readlines()`:

```
$ cat -n tac2.py
     1    #!/usr/bin/env python3
     2    """print a file in reverse"""
     3
     4    import sys
     5    import os
     6
     7    args = sys.argv[1:]
     8    if len(args) != 1:
     9        print('Usage: {} FILE'.format(os.path.basename(sys
.argv[0])))
    10        sys.exit(1)
    11
    12    file = args[0]
    13    if not os.path.isfile(file):
    14        print('"{}" is not a file'.format(file))
    15        sys.exit(1)
    16
    17    lines = open(file).readlines()
    18    lines.reverse()
    19
    20    for line in lines:
    21        print(line, end='')
```

This version uses the `reversed` function:

```
$ cat -n tac3.py
     1    #!/usr/bin/env python3
     2    """print a file in reverse"""
     3
     4    import sys
     5    import os
     6
     7    args = sys.argv[1:]
     8    if len(args) != 1:
     9        print('Usage: {} FILE'.format(os.path.basename(sys
.argv[0])))
    10        sys.exit(1)
    11
    12    file = args[0]
    13    if not os.path.isfile(file):
    14        print('"{}" is not a file'.format(file))
    15        sys.exit(1)
    16
    17    lines = open(file).readlines()
    18
    19    for line in reversed(lines):
    20        print(line, end='')
```

And finally I will introduce the `with/open` convention that you will see in Python:

```
$ cat -n tac4.py
     1    #!/usr/bin/env python3
     2    """print a file in reverse"""
     3
     4    import sys
     5    import os
     6
     7    args = sys.argv[1:]
     8    if len(args) != 1:
     9        print('Usage: {} FILE'.format(os.path.basename(sys
.argv[0])))
    10        sys.exit(1)
    11
    12    file = args[0]
    13    if not os.path.isfile(file):
    14        print('"{}" is not a file'.format(file))
    15        sys.exit(1)
    16
    17    with open(file) as fh:
    18        lines = fh.readlines()
    19        for line in reversed(lines):
    20            print(line, end='')
```

Picnic

Here is a little memory game you might have played with your bored siblings on family car trips:

```
$ ./picnic.py
What are you bringing? [q to quit] chips
We'll have chips.
What else are you bringing? [q to quit] ham sammich
We'll have chips and ham sammich.
What else are you bringing? [q to quit] Coke
We'll have chips, ham sammich, and Coke.
What else are you bringing? [q to quit] cupcakes
We'll have chips, ham sammich, Coke, and cupcakes.
What else are you bringing? [q to quit] apples
We'll have chips, ham sammich, Coke, cupcakes, and apples.
What else are you bringing? [q to quit] q
Bye.
```

Each person introduces a new item, and the other person has to remember all the previous items and add a new one. This is a classic "stack" that can be implemented with lists:

```
$ cat -n picnic.py
     1    #!/usr/bin/env python3
     2    """What are you bringing to the picnic?"""
     3
     4    # --------------------------------------------------
     5    def joiner(items):
     6        """properly conjuct items"""
     7        num_items = len(items)
     8        if num_items == 0:
     9            return ''
    10        elif num_items == 1:
    11            return items[0]
    12        elif num_items == 2:
    13            return ' and '.join(items)
    14        else:
    15            items[-1] = 'and ' + items[-1]
    16            return ', '.join(items)
    17
    18    # --------------------------------------------------
    19    def main():
```

```
    20        """start here"""
    21        items = []
    22
    23        while True:
    24            item = input('What {}are you bringing? [q to q
uit] '.format('else ' if items else ''))
    25            if item == 'q':
    26                break
    27            elif len(item.strip()) > 0:
    28                if item in items:
    29                    print('You said "{}" already.'.format(
item))
    30                else:
    31                    items.append(item)
    32                    print("We'll have {}.".format(joiner(i
tems.copy())))
    33
    34        print('Bye.')
    35
    36    # -----------------------------------------------
    37    if __name__ == '__main__':
    38        main()
```

One bug that got me in writing this program was line 32. Because I mutate the last item in the list in my `joiner` function, I was actually mutating the original list! I had to learn to pass `items.copy()` so as to work on a copy of the data and not the actual list.

Insults

Sometimes (esp when writing games) you may want a random selection from a list of items. Here is an insult generator that draws from the fabulous vocabulary of Shakespeare:

```
$ cat -n insult.py
     1    #!/usr/bin/env python3
     2    """Shakespearean insult generator"""
     3
```

```
     4    import sys
     5    import random
     6
     7    ADJECTIVES = """
     8    scurvy old filthy scurilous lascivious foolish rascaly
 gross rotten corrupt
     9    foul loathsome irksome heedless unmannered whoreson cu
llionly false filthsome
    10    toad-spotted caterwauling wall-eyed insatiate vile pee
vish infected
    11    sodden-witted lecherous ruinous indistinguishable dish
onest thin-faced
    12    slanderous bankrupt base detestable rotten dishonest l
ubbery
    13    """.split()
    14
    15    NOUNS = """
    16    knave coward liar swine villain beggar slave scold jol
thead whore barbermonger
    17    fishmonger carbuncle fiend traitor block ape braggart
jack milksop boy harpy
    18    recreant degenerate Judas butt cur Satan ass coxcomb d
andy gull minion
    19    ratcatcher maw fool rogue lunatic varlet worm
    20    """.split()
    21
    22    args = sys.argv[1:]
    23    num = 5
    24    if len(args) > 0 and args[0].isdigit():
    25        num = int(args[0])
    26
    27    for i in range(0, num):
    28        adjs = []
    29        for j in range(0, 3):
    30            adjs.append(random.choice(ADJECTIVES))
    31
    32        print('You {} {}!'.format(', '.join(adjs), random.
choice(NOUNS)))
$ ./insult.py foo
You bankrupt, cullionly, detestable milksop!
```

```
You foul, indistinguishable, false Satan!
You lascivious, scurilous, bankrupt villain!
You lascivious, lecherous, rotten jack!
You toad-spotted, base, foolish Satan!
$ ./insult.py 3
You detestable, cullionly, wall-eyed scold!
You peevish, caterwauling, caterwauling traitor!
You thin-faced, foul, dishonest Judas!
```

Notice how the program takes an optional argument that I expect to be an integer. On line 24, I test both that there is an argument present and that it `isdigit()` before attempting to use it as a number. The real work is done by the `random.choice` function to grab my adjectives and noun. The `"""` operator lets us write strings with newlines, then we `split` the long string into words. This is a common idiom in Python. Notice the use of `append` to grow the list of adjectives on line 30, then we `join` them on line 32.

Synthetic Biology

Lists could represent biological entities such as promotor, coding, and terminator regions. Let's say we wanted to design synthetic microbes where we tested all possible permutations of these regions with each other to see if we were able to increase production of a desired enzyme. Since the operation is N^3, I will only show the output for 2 genes:

```
$ ./recomb.py 2
N = "2"
  1: ('P1', 'C1', 'T1')
  2: ('P1', 'C1', 'T2')
  3: ('P1', 'C2', 'T1')
  4: ('P1', 'C2', 'T2')
  5: ('P2', 'C1', 'T1')
  6: ('P2', 'C1', 'T2')
  7: ('P2', 'C2', 'T1')
  8: ('P2', 'C2', 'T2')
```

Here is the Python code:

```
$ cat -n recomb.py
     1    #!/usr/bin/env python3
     2    """Show recominations"""
     3
     4    import os
     5    import sys
     6    from itertools import product, chain
     7
     8    args = sys.argv[1:]
     9
    10    if len(args) != 1:
    11        print('Usage: {} NUM_GENES'.format(os.path.basenam
e(sys.argv[0])))
    12        sys.exit(1)
    13
    14    if not args[0].isdigit():
    15        print('"{}" does not look like an integer'.format(
args[0]))
    16        sys.exit(1)
    17
    18    num_genes = int(args[0])
    19    if not 2 <= num_genes <= 10:
    20        print('NUM_GENES must be greater than 1, less than
 10')
    21        sys.exit(1)
    22
    23    promotors = []
    24    coding = []
    25    terminators = []
    26    for i in range(0, num_genes):
    27        n = str(i + 1)
    28        promotors.append('P' + n)
    29        coding.append('C' + n)
    30        terminators.append('T' + n)
    31
    32    print('N = "{}"'.format(num_genes))
    33    for i, combo in enumerate(chain(product(promotors, cod
ing, terminators))):
    34        print('{:3}: {}'.format(i + 1, combo))
```

The heavy lifting is being done on line 33 by the `product` function we get from the `itertools` module. Because this function is given three lists to cross, it returns a list of three sub-lists which I want to combine into one list with `chain`. Then I call the `enumerate` function (shown in the first section) to get the list index and the list member in one loop so I don't have to keep up with a counter variable.

I don't like lines 26-30, so I tried rewriting using a list comprehension (one of the most useful things you can do with lists). Here's an example of using list comprehensions to square the numbers from 1 to 4:

```
>>> [x ** 2 for x in range(1, 5)]
[1, 4, 9, 16]
```

You can add a predicate for item selection to the end:

```
>>> [x ** 2 for x in range(1, 5) if x % 2 == 0]
[4, 16]
```

Here is the comprehensions in the program (lines 23-25):

```
$ cat -n recomb2.py
     1    #!/usr/bin/env python3
     2    """Show recominations"""
     3
     4    import os
     5    import sys
     6    from itertools import product, chain
     7
     8    args = sys.argv[1:]
     9
    10    if len(args) != 1:
    11        print('Usage: {} NUM_GENES'.format(os.path.basenam
e(sys.argv[0])))
    12        sys.exit(1)
    13
    14    if not args[0].isdigit():
    15        print('"{}" does not look like an integer'.format(
args[0]))
    16        sys.exit(1)
    17
    18    num_genes = int(args[0])
    19    if not 2 <= num_genes <= 10:
    20        print('NUM_GENES must be greater than 1, less than
 10')
    21        sys.exit(1)
    22
    23    promotors = ['P' + str(n + 1) for n in range(0, num_ge
nes)]
    24    coding = ['C' + str(n + 1) for n in range(0, num_genes
)]
    25    terminators = ['T' + str(n + 1) for n in range(0, num_
genes)]
    26
    27    print('N = "{}"'.format(num_genes))
    28    for i, combo in enumerate(chain(product(promotors, cod
ing, terminators))):
    29        print('{:3}: {}'.format(i + 1, combo))
```

But these lines are identical with the exception of the character I'm using, so I can put that code into a little function:

```
$ cat -n recomb3.py
     1    #!/usr/bin/env python3
     2    """Show recominations"""
     3
     4    import os
     5    import sys
     6    from itertools import product, chain
     7
     8    args = sys.argv[1:]
     9
    10    if len(args) != 1:
    11        print('Usage: {} NUM_GENES'.format(os.path.basenam
e(sys.argv[0])))
    12        sys.exit(1)
    13
    14    if not args[0].isdigit():
    15        print('"{}" does not look like an integer'.format(
args[0]))
    16        sys.exit(1)
    17
    18    num_genes = int(args[0])
    19    if not 2 <= num_genes <= 10:
    20        print('NUM_GENES must be greater than 1, less than
 10')
    21        sys.exit(1)
    22
    23    def gen(prefix):
    24        return [prefix + str(n + 1) for n in range(0, num_
genes)]
    25
    26    promotors = gen('P')
    27    coding = gen('C')
    28    terminators = gen('T')
    29
    30    print('N = "{}"'.format(num_genes))
    31    for i, combo in enumerate(chain(product(promotors, cod
ing, terminators))):
    32        print('{:3}: {}'.format(i + 1, combo))
```

Now all the repeated code is in the `gen` function (line 23-24), and I simply call that for each character I want.

Dictionaries

Python has a data type called a "dictionary" that allows you to associate some "key" (often a string) to some "value" (which can be anything such as a string, number, tuple, list, set, or another dictionary). The same data structure is also called a map, hash, and associative array.

You can define the define a dictionary with all the key/value pairs using the `{}` braces:

```
>>> patch = {'species': 'dog', 'age': 4}
>>> patch['species']
'dog'
>>> type(patch['species'])
<class 'str'>
>>> patch['age']
4
>>> type(patch['age'])
<class 'int'>
>>> patch['likes'] = ['walking', 'running', 'car trips', 'treats
', 'pets']
>>> patch
{'species': 'dog', 'age': 4, 'likes': ['walking', 'running', 'ca
r trips', 'treats', 'pets']}
>>> type(patch['likes'])
<class 'list'>
>>> 'Patch is {} and likes {}'.format(patch['age'], ', '.join(pa
tch['likes']))
'Patch is 4 and likes walking, running, car trips, treats, pets'
```

Or you can use the `dict()` constructor and add key/value pairs:

```
>>> cat = dict()
>>> cat['name'] = 'Patrick the Catrick'
>>> cat
{'name': 'Patrick the Catrick'}
```

Bridge of Death

Let's write a script to play with a dictionary:

```
$ cat -n bridge_of_death.py
     1    #!/usr/bin/env python3
     2
     3    person = {}
     4    print(person)
     5
     6    print('\n'.join(['Stop!', 'Who would cross the Bridge
of Death',
     7                     'must answer me these questions three
,',
     8                     'ere the other side he see.']))
     9
    10    for field in ['name', 'quest', 'favorite color']:
    11        person[field] = input('What is your {}? '.format(f
ield))
    12        print(person)
    13
    14    if person['favorite color'].lower() == 'blue':
    15        print('Right, off you go.')
    16    else:
    17        print('You have been eaten by a grue.')
```

And here it is in action:

```
$ ./bridge_of_death.py
{}
Stop!
Who would cross the Bridge of Death
must answer me these questions three,
ere the other side he see.
What is your name? Sir Lancelot of Camelot
{'name': 'Sir Lancelot of Camelot'}
What is your quest? To seek the Holy Grail
{'name': 'Sir Lancelot of Camelot', 'quest': 'To seek the Holy G
rail'}
What is your favorite color? Blue
{'name': 'Sir Lancelot of Camelot', 'quest': 'To seek the Holy G
rail', 'favorite color': 'Blue'}
Right, off you go.
$ ./bridge_of_death.py
{}
Stop!
Who would cross the Bridge of Death
must answer me these questions three,
ere the other side he see.
What is your name? Sir Galahad of Camelot
{'name': 'Sir Galahad of Camelot'}
What is your quest? I seek the Holy Grail
{'name': 'Sir Galahad of Camelot', 'quest': 'I seek the Holy Gra
il'}
What is your favorite color? Blue. No yello--
{'name': 'Sir Galahad of Camelot', 'quest': 'I seek the Holy Gra
il', 'favorite color': 'Blue. No yello--'}
You have been eaten by a grue.
```

Gashlycrumb

Dictionaries are perfect for looking up some bit of information by some value:

```
$ ./gashlycrumb.py c
C is for Clara who wasted away.
```

```
$ ./gashlycrumb.py t
T is for Titus who flew into bits.
$ cat -n gashlycrumb.py
     1    #!/usr/bin/env python3
     2    """dictionary lookup"""
     3
     4    import os
     5    import sys
     6
     7    args = sys.argv[1:]
     8
     9    if len(args) != 1:
    10        print('Usage: {} LETTER'.format(os.path.basename(s
ys.argv[0])))
    11        sys.exit(1)
    12
    13    letter = args[0].upper()
    14
    15    text = """
    16    A is for Amy who fell down the stairs.
    17    B is for Basil assaulted by bears.
    18    C is for Clara who wasted away.
    19    D is for Desmond thrown out of a sleigh.
    20    E is for Ernest who choked on a peach.
    21    F is for Fanny sucked dry by a leech.
    22    G is for George smothered under a rug.
    23    H is for Hector done in by a thug.
    24    I is for Ida who drowned in a lake.
    25    J is for James who took lye by mistake.
    26    K is for Kate who was struck with an axe.
    27    L is for Leo who choked on some tacks.
    28    M is for Maud who was swept out to sea.
    29    N is for Neville who died of ennui.
    30    O is for Olive run through with an awl.
    31    P is for Prue trampled flat in a brawl.
    32    Q is for Quentin who sank on a mire.
    33    R is for Rhoda consumed by a fire.
    34    S is for Susan who perished of fits.
    35    T is for Titus who flew into bits.
    36    U is for Una who slipped down a drain.
```

```
    37    V is for Victor squashed under a train.
    38    W is for Winnie embedded in ice.
    39    X is for Xerxes devoured by mice.
    40    Y is for Yorick whose head was bashed in.
    41    Z is for Zillah who drank too much gin.
    42    """
    43
    44    lookup = {}
    45    for line in text.splitlines():
    46        if line:
    47            lookup[line[0]] = line
    48
    49    if letter in lookup:
    50        print(lookup[letter])
    51    else:
    52        print('I do not know "{}"'.format(letter))
$ ./gashlycrumb.py
Usage: gashlycrumb.py LETTER
$ ./gashlycrumb.py a
A is for Amy who fell down the stairs.
$ ./gashlycrumb.py b
B is for Basil assaulted by bears.
$ ./gashlycrumb.py 8
I do not know "8"
```

On line 47, we create the `lookup` using the first character of the line (`line[0]`). On line 49, we look to see if we have that letter in the `lookup`, printing the line of text if we do or complaining if we don't.

If we return to our previous chapter's DNA base counter, we can use dictionaries for this:

```
$ cat -n dna3.py
     1    #!/usr/bin/env python3
     2    """Tetra-nucleotide counter"""
     3
     4    import sys
     5    import os
     6
     7    args = sys.argv[1:]
     8
     9    if len(args) != 1:
    10        print('Usage: {} DNA'.format(os.path.basename(sys.
argv[0])))
    11        sys.exit(1)
    12
    13    dna = args[0]
    14
    15    count = {}
    16
    17    for base in dna.lower():
    18        if not base in count:
    19            count[base] = 0
    20
    21        count[base] += 1
    22
    23    counts = []
    24    for base in "acgt":
    25        num = count[base] if base in count else 0
    26        counts.append(str(num))
    27
    28    print(' '.join(counts))
$ ./dna3.py `cat input.txt `
20 12 17 21
```

But why? Well, this has the great advantage of not having to declare four variables to count the four bases. True, we're only checking (in line 24) for those four, but we can now count all the letters in any string.

Notice that we create a new dict on line 15 with empty curlies `{}`. In line 18, we have to check if the base exists in the dict; if it doesn't, we initialize it to 0, and then we increment it by one. In line 25, we have to be careful when asking for a key that doesn't exist:

```
>>> cat['likes']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'likes'
```

If we were counting a string of DNA like "AAAAAA," then there would be no C, G or T to report, so we have to use an `if/then` expression:

```
>>> seq = 'AAAAAA'
>>> counts = {}
>>> for base in seq:
...   if not base in counts:
...     counts[base] = 0
...   counts[base] += 1
...
>>> counts
{'A': 6}
>>> counts['G']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'G'
>>> g = counts['G'] if 'G' in counts else 0
```

Or we can use the `get` method of a dictionary to safely get a value by a key even if the key doesn't exist:

```
>>> counts.get('G')
>>> type(counts.get('G'))
<class 'NoneType'>
```

If you look at "dna4.py," you'll see it's exactly the same as "dna3.py" with this exception:

```
    23    counts = []
    24    for base in "acgt":
    25        num = count.get(base, 0)
    26        counts.append(str(num))
```

The `get` method will not blow up your program, and it accepts an optional second argument for the default value when nothing is present:

```
>>> cat.get('likes')
>>> type(cat.get('likes'))
<class 'NoneType'>
>>> cat.get('likes', 'Cats like nothing')
'Cats like nothing'
```

Sidebar: Truthiness

Note that you might be tempted to write:

```
>>> cat.get('likes') or 'Cats like nothing'
'Cats like nothing'
```

Which appears to do the same thing, but compare with this:

```
>>> d = {'x': 0, 'y': '', 'z': None}
>>> for k in sorted(d.keys()):
...   print('{} = "{}"'.format(k, d.get(k) or 'NA'))
...
x = "NA"
y = "NA"
z = "NA"
>>> for k in sorted(d.keys()):
...   print('{} = "{}"'.format(k, d.get(k, 'NA')))
...
x = "0"
y = ""
z = "None"
```

This is a minor but potentially pernicious error due to Python's idea of Truthiness (tm):

```
>>> 1 == True
True
>>> 0 == False
True
```

The integer `1` is not actually the same thing as the boolean value `True`, but Python will treat it as such. Vice verse for `0` and `False`. The only true way to get around this is to explicitly check for `None`:

```
>>> for k in sorted(d.keys()):
...   val = d.get(k)
...   print('{} = "{}"'.format(k, 'NA' if val is None else val))
...
x = "0"
y = ""
z = "NA"
```

To get around the check, we could initialize the dict:

```
$ cat -n dna5.py
     1    #!/usr/bin/env python3
     2    """Tetra-nucleotide counter"""
     3
     4    import sys
     5    import os
     6
     7    args = sys.argv[1:]
     8
     9    if len(args) != 1:
    10        print('Usage: {} DNA'.format(os.path.basename(sys.
argv[0])))
    11        sys.exit(1)
    12
    13    dna = args[0]
    14
    15    count = {'a': 0, 'c': 0, 'g': 0, 't': 0}
    16
    17    for base in dna.lower():
    18        if base in count:
    19            count[base] += 1
    20
    21    counts = []
    22    for base in "acgt":
    23        counts.append(str(count[base]))
    24
    25    print(' '.join(counts))
```

Back To Our Program

Now when we check on line 18, we're only going to count bases that we initialized; further, we can then just use the `keys` method to get the bases:

```
$ cat -n dna5.py
     1    #!/usr/bin/env python3
     2    """Tetra-nucleotide counter"""
     3
     4    import sys
     5    import os
     6
     7    args = sys.argv[1:]
     8
     9    if len(args) != 1:
    10        print('Usage: {} DNA'.format(os.path.basename(sys.
argv[0])))
    11        sys.exit(1)
    12
    13    dna = args[0]
    14
    15    count = {'a': 0, 'c': 0, 'g': 0, 't': 0}
    16
    17    for base in dna.lower():
    18        if base in count:
    19            count[base] += 1
    20
    21    counts = []
    22    for base in sorted(count.keys()):
    23        counts.append(str(count[base]))
    24
    25    print(' '.join(counts))
```

This kind of checking and initializing is so common that there is a standard module to define a dictionary with a default value. Unsurprisingly, it is called "defaultdict":

```
$ cat -n dna6.py
     1    #!/usr/bin/env python3
     2    """Tetra-nucleotide counter"""
     3
     4    import sys
     5    import os
     6    from collections import defaultdict
     7
     8    args = sys.argv[1:]
     9
    10    if len(args) != 1:
    11        print('Usage: {} DNA'.format(os.path.basename(sys.
argv[0])))
    12        sys.exit(1)
    13
    14    dna = args[0]
    15
    16    count = defaultdict(int)
    17
    18    for base in dna.lower():
    19        count[base] += 1
    20
    21    counts = []
    22    for base in "acgt":
    23        counts.append(str(count[base]))
    24
    25    print(' '.join(counts))
```

On line 16, we create a `defaultdict` with the `int` type (not in quotes) for which the default value will be zero:

```
>>> from collections import defaultdict
>>> counts = defaultdict(int)
>>> counts['a']
0
```

Finally, I will show you the `Counter` that will do all the base-counting for you, returning a `defaultdict`:

```
>>> from collections import Counter
>>> c = Counter('AACTAC')
>>> c['A']
3
>>> c['G']
0
```

And here is it in the script:

```
 $ cat -n dna7.py
     1    #!/usr/bin/env python3
     2    """Tetra-nucleotide counter"""
     3
     4    import sys
     5    import os
     6    from collections import Counter
     7
     8    args = sys.argv[1:]
     9
    10    if len(args) != 1:
    11        print('Usage: {} DNA'.format(os.path.basename(sys.
argv[0])))
    12        sys.exit(1)
    13
    14    dna = args[0]
    15
    16    count = Counter(dna.lower())
    17
    18    counts = []
    19    for base in "acgt":
    20        counts.append(str(count[base]))
    21
    22    print(' '.join(counts))
```

So we can take that and create a program that counts all characters either from the command line or a file:

```
$ cat -n char_count1.py
     1    #!/usr/bin/env python3
     2    """Character counter"""
     3
     4    import sys
     5    import os
     6    from collections import Counter
     7
     8    args = sys.argv
     9
    10    if len(args) != 2:
    11        print('Usage: {} INPUT'.format(os.path.basename(ar
gs[0])))
    12        sys.exit(1)
    13
    14    arg = args[1]
    15    text = ''
    16    if os.path.isfile(arg):
    17        text = ''.join(open(arg).read().splitlines())
    18    else:
    19        text = arg
    20
    21    count = Counter(text.lower())
    22
    23    for letter, num in count.items():
    24        print('{} {:5}'.format(letter, num))
$ ./char_count1.py input.txt
a    20
g    17
c    12
t    21
```

Methods

The `keys` from a dict are in no particular order:

```
>>> c = Counter('AAACTAGGGACTGA')
>>> c
Counter({'A': 6, 'G': 4, 'C': 2, 'T': 2})
>>> c.keys()
dict_keys(['A', 'C', 'T', 'G'])
```

If you want them sorted, you must be explicit:

```
>>> sorted(c.keys())
['A', 'C', 'G', 'T']
```

Note that, unlike a list, you cannot call `sort` which makes sense as that will try to sort a list in-place:

```
>>> c.keys().sort()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'dict_keys' object has no attribute 'sort'
```

You can also just call `values` to get those:

```
>>> c.values()
dict_values([6, 2, 2, 4])
```

Often you will want to go through the `items` in a dict and do something with the key and value:

```
>>> for base, count in c.items():
...   print('{} = {}'.format(base, count))
...
A = 6
C = 2
T = 2
G = 4
```

But if you want to have the `keys` in a particular order, you can do this:

```
>>> for base in sorted(c.keys()):
...   print('{} = {}'.format(base, c[base]))
...
A = 6
C = 2
G = 4
T = 2
```

Or you can notice that `items` returns a list of tuples:

```
>>> c.items()
dict_items([('A', 6), ('C', 2), ('T', 2), ('G', 4)])
```

And you can call `sorted` on that:

```
>>> sorted(c.items())
[('A', 6), ('C', 2), ('G', 4), ('T', 2)]
```

Which means this will work:

```
>>> for base, count in sorted(c.items()):
...   print('{} = {}'.format(base, count))
...
A = 6
C = 2
G = 4
T = 2
```

Note that `sorted` will sort by the first elements of all the tuples, then by the second, and so forth:

```
>>> genes = [('Indy', 4), ('Boss', 2), ('Lush', 10), ('Boss', 4)
, ('Lush', 1)]
>>> sorted(genes)
[('Boss', 2), ('Boss', 4), ('Indy', 4), ('Lush', 1), ('Lush', 10
)]
```

If we want to sort the bases instead by their frequency, we have to use some trickery like a list comprehension to first reverse the tuples:

```
>>> [(x[1], x[0]) for x in c.items()]
[(6, 'A'), (2, 'C'), (2, 'T'), (4, 'G')]
>>> sorted([(x[1], x[0]) for x in c.items()])
[(2, 'C'), (2, 'T'), (4, 'G'), (6, 'A')]
```

But what is particularly nifty about Counters is that they have built-in methods to help you with such actions:

```
>>> c.most_common(2)
[('A', 6), ('G', 4)]
>>> c.most_common()
[('A', 6), ('G', 4), ('C', 2), ('T', 2)]
```

You should read the documentation to learn more (https://docs.python.org/3/library/collections.html\.

Character Counter with the works

Finally, I'll show you a version of the character counter that takes some other arguments to control how to show the results:

```
$ cat -n char_count2.py
     1    #!/usr/bin/env python3
     2    """Character counter"""
     3
     4    import argparse
     5    import os
     6    import sys
     7    from collections import Counter
     8
     9    # --------------------------------------------------
    10    def get_args():
    11        """get args"""
    12        parser = argparse.ArgumentParser(description='Argp
```

```
arse Python script')
    13        parser.add_argument('arg', help='File/string to co
unt', type=str)
    14        parser.add_argument('-c', '--charsort', help='Sort
 by character',
    15                            dest='charsort', action='store
_true')
    16        parser.add_argument('-n', '--numsort', help='Sort
by number',
    17                            dest='numsort', action='store_
true')
    18        parser.add_argument('-r', '--reverse', help='Sort
in reverse order',
    19                            dest='reverse', action='store_
true')
    20        return parser.parse_args()
    21
    22    # --------------------------------------------------
    23    def main():
    24        """main"""
    25        args = get_args()
    26        arg = args.arg
    27        charsort = args.charsort
    28        numsort = args.numsort
    29        revsort = args.reverse
    30
    31        if charsort and numsort:
    32            print('Please choose one of --charsort or --nu
msort')
    33            sys.exit(1)
    34
    35        if not charsort and not numsort:
    36            charsort = True
    37
    38        text = ''
    39        if os.path.isfile(arg):
    40            text = ''.join(open(arg).read().splitlines())
    41        else:
    42            text = arg
    43
```

```
    44        count = Counter(text.lower())
    45
    46        if charsort:
    47            letters = sorted(count.keys())
    48            if revsort:
    49                letters.reverse()
    50
    51            for letter in letters:
    52                print('{} {:5}'.format(letter, count[lette
r]))
    53        else:
    54            pairs = sorted([(x[1], x[0]) for x in count.it
ems()])
    55            if revsort:
    56                pairs.reverse()
    57
    58            for n, char in pairs:
    59                print('{} {:5}'.format(char, n))
    60
    61    # ----------------------------------------------------
    62    if __name__ == '__main__':
    63        main()
```

Acronym Finder

Similar to the `gashlycrumb.py` program that looked up a line of text for a given letter, we could randomly create meanings for a given acronym:

```
$ ./bacryonym.py NSF
NSF =
 - Nonrepresentationalism Staunchness Forever
 - Naturing Significantly Fontal
 - Nonclinical Solecistical Folkmoter
 - Nonhumanist Scaledrake Fellani
 - Naumk Sulpha Fause
$ ./bacryonym.py FBI
FBI =
 - Folksiness Boxmaker Interviewer
 - Flavorless Bumbler Incorruption
 - Flusterate Bakuninism Isopilocarpine
 - Freshen Bondsman Indigene
 - Fluotantalate Bornyl Interligamentous
```

That is just using the standard dictionary to look up words, so we could make it more interesting by using the works of Shakespeare:

```
$ ./bacryonym.py -w shakespeare.txt FBI
FBI =
 - Furthermore Burnet Instigation
 - Favor Bursting Insisting
 - Flower Beart Immanity
 - Fearfully Borne Itmy
 - Fooleries Blunts Intoxicates
```

Here is the Python for that:

```
$ cat -n bacryonym.py
     1    #!/usr/bin/env python3
     2    """Make guesses about acronyms"""
     3
     4    import argparse
     5    import sys
     6    import os
     7    import random
     8    import re
     9    from collections import defaultdict
```

```
    10
    11    # --------------------------------------------------
    12    def main():
    13        """main"""
    14        args = get_args()
    15        acronym = args.acronym
    16        wordlist = args.wordlist
    17        limit = args.num
    18        goodword = r'^[a-z]{2,}$'
    19        badwords = set(re.split(r'\s*,\s*', args.exclude.l
ower()))
    20
    21        if not re.match(goodword, acronym.lower()):
    22            print('"{}" must be >1 in length, only use let
ters'.format(acronym))
    23            sys.exit(1)
    24
    25        if not os.path.isfile(wordlist):
    26            print('"{}" is not a file.'.format(wordlist))
    27            sys.exit(1)
    28
    29        seen = {}
    30        words_by_letter = defaultdict(list)
    31        for word in open(wordlist).read().lower().split():
    32            clean = re.sub('[^a-z]', '', word)
    33            if re.match(goodword, clean) and clean not in
seen and clean not in badwords:
    34                seen[clean] = 1
    35                words_by_letter[clean[0]].append(clean)
    36
    37        len_acronym = len(acronym)
    38        definitions = []
    39        for i in range(0, limit):
    40            definition = []
    41            for letter in acronym.lower():
    42                possible = words_by_letter[letter]
    43                if len(possible) > 0:
    44                    definition.append(random.choice(possib
le).title())
    45
```

```
    46            if len(definition) == len_acronym:
    47                definitions.append(' '.join(definition))
    48
    49        if len(definitions) > 0:
    50            print(acronym.upper() + ' =')
    51            for definition in definitions:
    52                print(' - ' + definition)
    53        else:
    54            print('Sorry I could not find any good definit
ions')
    55
    56    # --------------------------------------------------
    57    def get_args():
    58        """get arguments"""
    59        parser = argparse.ArgumentParser(description='Expl
ain acronyms')
    60        parser.add_argument('acronym', help='Acronym', typ
e=str, metavar='STR')
    61        parser.add_argument('-n', '--num', help='Maximum n
umber of definitions',
    62                            type=int, metavar='NUM', defau
lt=5)
    63        parser.add_argument('-w', '--wordlist', help='Dict
ionary/word file',
    64                            type=str, metavar='STR',
    65                            default='/usr/share/dict/words
')
    66        parser.add_argument('-x', '--exclude', help='List
of words to exclude',
    67                            type=str, metavar='STR', defau
lt='a,an,the')
    68        return parser.parse_args()
    69
    70    # --------------------------------------------------
    71    if __name__ == '__main__':
    72        main()
```

Sequence Similarity

We can use dictionaries to count how many words are in common between any two texts. Since I'm only trying to see if a word is present, I can use a `set` which is like a `dict` where the values are just "1." Here is the code:

```
$ cat -n common_words.py
     1    #!/usr/bin/env python3
     2    """Count words in common between two files"""
     3
     4    import os
     5    import re
     6    import sys
     7    import string
     8
     9    # --------------------------------------------------
    10    def main():
    11        files = sys.argv[1:]
    12
    13        if len(files) != 2:
    14            msg = 'Usage: {} FILE1 FILE2'
    15            print(msg.format(os.path.basename(sys.argv[0])
))
    16            sys.exit(1)
    17
    18        for file in files:
    19            if not os.path.isfile(file):
    20                print('"{}" is not a file'.format(file))
    21                sys.exit(1)
    22
    23        file1, file2 = files[0], files[1]
    24        words1 = uniq_words(file1)
    25        words2 = uniq_words(file2)
    26        common = words1.intersection(words2)
    27        num_common = len(common)
    28        msg = 'There {} {} word{} in common between "{}" a
nd "{}."'
    29        print(msg.format('is' if num_common == 1 else 'are
',
    30                         num_common,
    31                         '' if num_common == 1 else 's',
    32                         os.path.basename(file1),
```

```
    33                          os.path.basename(file2)))
    34
    35        for i, word in enumerate(sorted(common)):
    36            print('{:3}: {}'.format(i + 1, word))
    37
    38    # ------------------------------------------------
    39    def uniq_words(file):
    40        regex = re.compile('[' + string.punctuation + ']')
    41        words = set()
    42        for line in open(file):
    43            for word in [regex.sub('', w) for w in line.lo
wer().split()]:
    44                words.add(word)
    45
    46        return words
    47
    48    # ------------------------------------------------
    49    if __name__ == '__main__':
    50        main()
```

Let's see it in action using a common nursery rhyme and a poem by William Blake (1757-1827):

```
$ cat mary-had-a-little-lamb.txt
Mary had a little lamb,
It's fleece was white as snow,
And everywhere that Mary went,
The lamb was sure to go.
$ cat little-lamb.txt
Little Lamb, who made thee?
Dost thou know who made thee?
Gave thee life, & bid thee feed
By the stream & o'er the mead;
Gave thee clothing of delight,
Softest clothing, wooly, bright;
Gave thee such a tender voice,
Making all the vales rejoice?
Little Lamb, who made thee?
Dost thou know who made thee?
Little Lamb, I'll tell thee,
Little Lamb, I'll tell thee,
He is called by thy name,
For he calls himself a Lamb.
He is meek, & he is mild;
He became a little child.
I a child, & thou a lamb,
We are called by his name.
Little Lamb, God bless thee!
Little Lamb, God bless thee!
$ ./common_words.py mary-had-a-little-lamb.txt little-lamb.txt
There are 4 words in common between "mary-had-a-little-lamb.txt"
 and "little-lamb.txt."
  1: a
  2: lamb
  3: little
  4: the
```

Well, that's pretty uninformative. Sure "a" and "the" are shared, but we don't much care about those. And while "little" and "lamb" are present, it hardly tells us about how prevalent they are. In the nursery rhyme, they occur a total of 3 times, but

they make up a significant portion of the Blake poem. Let's try to work in word frequency:

```
$ cat -n common_words2.py
     1    #!/usr/bin/env python3
     2    """Count words/frequencies in two files"""
     3
     4    import os
     5    import re
     6    import sys
     7    import string
     8    from collections import defaultdict
     9
    10    # -------------------------------------------------
    11    def word_counts(file):
    12        """Return a dictionary of words/counts"""
    13        words = defaultdict(int)
    14        regex = re.compile('[' + string.punctuation + ']')
    15        for line in open(file):
    16            for word in [regex.sub('', w) for w in line.lo
wer().split()]:
    17                words[word] += 1
    18
    19        return words
    20
    21    # -------------------------------------------------
    22    def main():
    23        """Start here"""
    24        args = sys.argv[1:]
    25
    26        if len(args) != 2:
    27            msg = 'Usage: {} FILE1 FILE2'
    28            print(msg.format(os.path.basename(sys.argv[0])
))
    29            sys.exit(1)
    30
    31        for file in args[0:2]:
    32            if not os.path.isfile(file):
    33                print('"{}" is not a file'.format(file))
    34                sys.exit(1)
```

```
    35
    36        file1 = args[0]
    37        file2 = args[1]
    38        words1 = word_counts(file1)
    39        words2 = word_counts(file2)
    40        common = set(words1.keys()).intersection(set(words
2.keys()))
    41        num_common = len(common)
    42        verb = 'is' if num_common == 1 else 'are'
    43        plural = '' if num_common == 1 else 's'
    44        msg = 'There {} {} word{} in common between "{}" (
{}) and "{}" ({}).'
    45        tot1 = sum(words1.values())
    46        tot2 = sum(words2.values())
    47        print(msg.format(verb, num_common, plural, file1,
tot1, file2, tot2))
    48
    49        if num_common > 0:
    50            fmt = '{:>3} {:20} {:>5} {:>5}'
    51            print(fmt.format('#', 'word', '1', '2'))
    52            print('-' * 36)
    53            shared1, shared2 = 0, 0
    54            for i, word in enumerate(sorted(common)):
    55                c1 = words1[word]
    56                c2 = words2[word]
    57                shared1 += c1
    58                shared2 += c2
    59                print(fmt.format(i + 1, word, c1, c2))
    60
    61            print(fmt.format('', '-----', '--', '--'))
    62            print(fmt.format('', 'total', shared1, shared2
))
    63            print(fmt.format('', 'pct',
    64                             int(shared1/tot1 * 100), int(
shared2/tot2 * 100)))
    65
    66    # ------------------------------------------------
    67    if __name__ == '__main__':
    68        main()
```

And here it is in action:

```
$ ./common_words2.py mary-had-a-little-lamb.txt little-lamb.txt
There are 4 words in common between "mary-had-a-little-lamb.txt"
 (22) and "little-lamb.txt" (113).
  # word                     1     2
-------------------------------------
  1 a                        1     5
  2 lamb                     2     8
  3 little                   1     7
  4 the                      1     3
    -----                   --    --
    total                    5    23
    pct                     22    20
```

It is interesting (to me, at least) that the shared content actually works out to about the same proportion no matter the direction. Imagine comparing a large genome to a smaller one -- what is a significant portion of shared sequence space from the smaller genome might be only a small fraction of the larger one. Here we see that just those few words make up an equivalent proportion of both texts because of how repeated the words are in the Blake poem.

This is all pretty good as long as the words are spelled the same, but take the two texts here that show variations between British and American English:

```
$ cat british.txt
I went to the theatre last night with my neighbour and had a lit
re of
beer, the colour and flavour of which put us into such a good hu
mour
that we forgot our labours.  We set about to analyse our behavio
ur,
organise our thoughts, recognise our faults, catalogue our merit
s, and
generally have a dialogue without pretence as a licence to impro
ve
ourselves.
$ cat american.txt
I went to the theater last night with my neighbor and had a lite
```

```
r of
beer, the color and flavor of which put us into such a good humo
r that
we forgot our labors.  We set about to analyze our behavior, org
anize
our thoughts, recognize our faults, catalog our merits, and gene
rally
have a dialog without pretense as a license to improve ourselves
.
$ ./common_words2.py british.txt american.txt
There are 34 words in common between "british.txt" (63) and "ame
rican.txt" (63).
  # word                      1     2
-------------------------------------
  1 a                         4     4
  2 about                     1     1
  3 and                       3     3
  4 as                        1     1
  5 beer                      1     1
  6 faults                    1     1
  7 forgot                    1     1
  8 generally                 1     1
  9 good                      1     1
 10 had                       1     1
 11 have                      1     1
 12 i                         1     1
 13 improve                   1     1
 14 into                      1     1
 15 last                      1     1
 16 merits                    1     1
 17 my                        1     1
 18 night                     1     1
 19 of                        2     2
 20 our                       5     5
 21 ourselves                 1     1
 22 put                       1     1
 23 set                       1     1
 24 such                      1     1
 25 that                      1     1
 26 the                       2     2
```

```
 27 thoughts                1     1
 28 to                      3     3
 29 us                      1     1
 30 we                      2     2
 31 went                    1     1
 32 which                   1     1
 33 with                    1     1
 34 without                 1     1
    -----                  --    --
    total                  48    48
    pct                    76    76
```

Obviously we will miss all those words because the are not spelled exactly the same. Neither are genomes. So we need a way to decide if two words or sequences are similar enough. One way is through sequence alignment:

```
l a b o u r     c a t a l o g u e     p r e t e n c e    l i
t r e
| | | |   |     | | | | | | |         | | | | | |   |    | |
|
l a b o   r     c a t a l o g         p r e t e n s e    l i
t e r
```

Try writing a sequence alignment program (no, really!), and you'll find it's really quite difficult. Decades of research have gone into Smith-Waterman and BLAST and BLAT and LAST and more. Alignment works very well, but it's computationally expensive. We need a faster approximation of similarity. Enter k-mers!

A k-mer is a `k` length of "mers" or contiguous sequence (think "polymers"). Here are the 3/4-mers in my last name:

```
$ ./kmer_tiler.py youens
There are 4 3-mers in "youens."
youens
you
 oue
  uen
   ens
$ ./kmer_tiler.py youens 4
There are 3 4-mers in "youens."
youens
youe
 ouen
  uens
```

If instead looking for shared "words" we search for k-mers, we will find very different results, and the length of the k-mer matters. For instance, the first 3-mer in my name, "you" can be found 81 times in my local dictionary, but the 4-mer "youe" not at all. The longer the k-mer, the greater the specificity. Let's try our English variations with a k-mer counter:

```
$ ./common_kmers.py british.txt american.txt
There are 112 kmers in common between "british.txt" (127) and "a
merican.txt" (127).
  # kmer                      1     2
-----------------------------------
  1 abo                       2     2
  2 all                       1     1
...
111 whi                       1     1
112 wit                       2     2
    -----                    --    --
    total                   142   133
    pct                      86    86
```

Our word counting program thought these two texts only 76% similar, but our kmer counter thinks they are 86% similar.

Command-line Arguments

If you have not already, I encourage you to copy the "new_py.py" script into your `$PATH` and then execute it with the `-a` argument to start a new script with `argparse`:

```
$ ./new_py.py -a test
Done, see new script "test.py."
```

If you check out the new script, it has a `get_args` function that will show you how to create named arguments for strings, integers, booleans, and positional arguments:

```
$ cat -n test.py
     1    #!/usr/bin/env python3
     2
     3    import argparse
     4    import os
     5    import sys
     6
     7    # --------------------------------------------------
     8    def get_args():
     9        parser = argparse.ArgumentParser(description='Argp
arse Python script')
    10        parser.add_argument('positional', metavar='str', h
elp='A positional argument')
    11        parser.add_argument('-a', '--arg', help='A named s
tring argument',
    12                            metavar='str', type=str, defau
lt='')
    13        parser.add_argument('-i', '--int', help='A named i
nteger argument',
    14                            metavar='int', type=int, defau
lt=0)
    15        parser.add_argument('-f', '--flag', help='A boolea
n flag',
```

```
    16                              action='store_true')
    17         return parser.parse_args()
    18
    19     # -----------------------------------------------
    20     def main():
    21         args = get_args()
    22         str_arg = args.arg
    23         int_arg = args.int
    24         flag_arg = args.flag
    25         pos_arg = args.positional
    26
    27         print('str_arg = "{}"'.format(str_arg))
    28         print('int_arg = "{}"'.format(int_arg))
    29         print('flag_arg = "{}"'.format(flag_arg))
    30         print('positional = "{}"'.format(pos_arg))
    31
    32     # -----------------------------------------------
    33     if __name__ == '__main__':
    34         main()
```

If you run without any arguments, you get a nice usage statement:

```
$ ./test.py
usage: test.py [-h] [-a str] [-i int] [-f] str
test.py: error: the following arguments are required: str
$ ./test.py foobar
str_arg = ""
int_arg = "0"
flag_arg = "False"
positional = "foobar"
$ ./test.py -a XYZ -i 42 -f foobar
str_arg = "XYZ"
int_arg = "42"
flag_arg = "True"
```

Please RTFM (https://docs.python.org/3/library/argparse.html\ to find all the other Fine things you can do with argparse.

CSV Files

Delimited text files are a standard way to distribute non/semi-hierarchical data -- e.g., records that can be represented each on one line. (When you get into data that have relationships, e.g., parents/children, then structures like XML and JSON are more appropriate, which is not to say that people haven't sorely abused this venerable format, e.g., GFF3.) Let's first take a look at the `csv` module in Python (https://docs.python.org/3/library/csv.html\ to parse the output from Centrifuge (http://www.ccb.jhu.edu/software/centrifuge/.

For this, we'll use some data from a study from Yellowstone National Park (https://www.imicrobe.us/sample/view/1378\ a file ("YELLOWSTONE_SMPL_20723.sum") showing the taxonomy ID for each read it was able to classify and 2) a file ("YELLOWSTONE_SMPL_20723.tsv") of the complete taxonomy information for each taxonomy ID. One record from the first looks like this:

```
       readID: Yellowstone_READ_00007510
        seqID: cid|321327
        taxID: 321327
        score: 640000
2ndBestScore: 0
   hitLength: 815
 queryLength: 839
  numMatches: 1
```

One from the second looks like this:

```
           name: synthetic construct
          taxID: 32630
        taxRank: species
    genomeSize: 26537524
      numReads: 19
numUniqueReads: 19
     abundance: 0.0
```

Read Count by taxID

Let's write a program that shows a table of the number of records for each "taxID":

```
$ cat -n read_count_by_taxid.py
     1    #!/usr/bin/env python3
     2    """Counts by taxID"""
     3
     4    import csv
     5    import os
     6    import sys
     7    from collections import defaultdict
     8
     9    args = sys.argv[1:]
    10
    11    if len(args) != 1:
    12        print('Usage: {} SAMPLE.SUM'.format(os.path.basena
me(sys.argv[0])))
    13        sys.exit(1)
    14
    15    sum_file = args[0]
    16
    17    _, ext = os.path.splitext(sum_file)
    18    if not ext == '.sum':
    19        print('File extention "{}" is not ".sum"'.format(e
xt))
    20        sys.exit(1)
    21
    22    counts = defaultdict(int)
    23    with open(sum_file) as csvfile:
    24        reader = csv.DictReader(csvfile, delimiter='\t')
    25        for row in reader:
    26            taxID = row['taxID']
    27            counts[taxID] += 1
    28
    29    print('\t'.join(['count', 'taxID']))
    30    for taxID, count in counts.items():
    31        print('\t'.join([str(count), taxID]))
```

As always, it prints a "usage" statement when run with no arguments. It also uses the `os.path.splitext` function to get the file extension and make sure that it is ".sum." Finally, if the input looks OK, then it uses the `csv.DictReader` module to parse each record of the file into a dictionary:

```
$ ./read_count_by_taxid.py
Usage: read_count_by_taxid.py SAMPLE.SUM
$ ./read_count_by_taxid.py YELLOWSTONE_SMPL_20723.tsv
File extention ".tsv" is not ".sum"
$ ./read_count_by_taxid.py YELLOWSTONE_SMPL_20723.sum
count    taxID
80    321332
6432    321327
19    32630
```

That's a start, but most people would rather see the a species name rather than the NCBI taxonomy ID, so we'll need to go look up the taxIDs in the ".tsv" file:

```
$ cat -n read_count_by_tax_name.py
     1    #!/usr/bin/env python3
     2    """Counts by tax name"""
     3
     4    import csv
     5    import os
     6    import sys
     7    from collections import defaultdict
     8
     9    args = sys.argv[1:]
    10
    11    if len(args) != 1:
    12        print('Usage: {} SAMPLE.SUM'.format(os.path.basena
me(sys.argv[0])))
    13        sys.exit(1)
    14
    15    sum_file = args[0]
    16
    17    basename, ext = os.path.splitext(sum_file)
    18    if not ext == '.sum':
    19        print('File extention "{}" is not ".sum"'.format(e
```

```
xt))
    20        sys.exit(1)
    21
    22    tsv_file = basename + '.tsv'
    23    if not os.path.isfile(tsv_file):
    24        print('Cannot find expected TSV "{}"'.format(tsv_f
ile))
    25        sys.exit(1)
    26
    27    tax_name = {}
    28    with open(tsv_file) as csvfile:
    29        reader = csv.DictReader(csvfile, delimiter='\t')
    30        for row in reader:
    31            tax_name[row['taxID']] = row['name']
    32
    33    counts = defaultdict(int)
    34    with open(sum_file) as csvfile:
    35        reader = csv.DictReader(csvfile, delimiter='\t')
    36        for row in reader:
    37            taxID = row['taxID']
    38            counts[taxID] += 1
    39
    40    print('\t'.join(['count', 'taxID']))
    41    for taxID, count in counts.items():
    42        name = tax_name.get(taxID) or 'NA'
    43        print('\t'.join([str(count), name]))
$ ./read_count_by_tax_name.py YELLOWSTONE_SMPL_20723.sum
count    taxID
6432    Synechococcus sp. JA-3-3Ab
80    Synechococcus sp. JA-2-3B'a(2-13)
19    synthetic construct
```

tabchk

A huge chunk of my time is spent doing ETL operations -- extract, transform, load -- meaning someone sends me data (Excel or delimited-text, JSON/XML), and I put it into some sort of database. I usually want to inspect the data to see what it

looks like, and it's hard to see the data when it's in columnar format. I'd rather see it formatted vertically. E.g., here is an export of some iMicrobe sample annotations:

```
$ tabchk.py -d imicrobe-blast-annots.txt
// ****** Record 1 ****** //
sample_id             : 1
project_name          : Acid Mine Drainage Metagenome
ontologies            : ENVO:01000033 (oceanic pelagic zone biom
e),ENVO:00002149 (sea water),ENVO:00000015 (ocean)
project_id            : 1
source_mat_id         : 33
sample_type           : metagenome
sample_description    : ACID_MINE_02 - 5-Way (CG) Acid Mine Drai
nage Biofilm
sample_name           : ACID_MINE_02
collection_start_time : 2002-03-01 00:00:00.0
collection_stop_time  : 2002-03-01 00:00:00.0
site_name             : Richmond Mine
latitude              : 40.666668
longitude             : -122.51667
site_description      : Acid Mine Drainage
country               : UNITED STATES
region                : Iron Mountain, CA
library_acc           : JGI_AMD_5WAY_IRNMTN_LIB_20020301
sequencing_method     : dideoxysequencing (Sanger)
dna_type              : gDNA
num_of_reads          : 180713
habitat_name          : waste water
temperature           : 42
sample_acc            : JGI_AMD_5WAY_IRNMTN_SMPL_20020301
```

If you open that file in a text editor, you will see there are many more column headers than are shown here. The lines are extremely long, and it's a pretty sparse matrix of data. The `-d` flag to the program indicates to show a "dense" matrix, i.e., leave out the empty fields. I find the above format much easier to read. Here is the program to do that:

```
$ cat -n tabchk.py
     1    #!/usr/bin/env python3
     2
     3    import argparse
     4    import csv
     5    import os
     6    import sys
     7
     8    # --------------------------------------------------
     9    def get_args():
    10        parser = argparse.ArgumentParser(description='Chec
k a delimited text file')
    11        parser.add_argument('file', metavar='str', help='F
ile')
    12        parser.add_argument('-s', '--sep', help='Field sep
arator',
    13                            metavar='str', type=str, defau
lt='\t')
    14        parser.add_argument('-l', '--limit', help='How man
y records to show',
    15                            metavar='int', type=int, defau
lt=1)
    16        parser.add_argument('-d', '--dense', help='Not spa
rse (skip empty fields)',
    17                            action='store_true')
    18        return parser.parse_args()
    19
    20    # --------------------------------------------------
    21    def main():
    22        args = get_args()
    23        file = args.file
    24        limit = args.limit
    25        sep = args.sep
    26        dense = args.dense
    27
    28        if not os.path.isfile(file):
    29            print('"{}" is not a file'.format(file))
    30            sys.exit(1)
    31
```

```
    32         with open(file) as csvfile:
    33             reader = csv.DictReader(csvfile, delimiter=sep
)
    34
    35             for i, row in enumerate(reader):
    36                 vals = dict([x for x in row.items() if x[1
] != '']) if dense else row
    37                 flds = vals.keys()
    38                 longest = max(map(len, flds))
    39                 fmt = '{:' + str(longest + 1) + '}: {}'
    40                 print('// ****** Record {} ****** //'.form
at(i+1))
    41                 for key, val in vals.items():
    42                     print(fmt.format(key, val))
    43
    44                 if i + 1 == limit:
    45                     break
    46
    47     # ------------------------------------------------
    48     if __name__ == '__main__':
    49         main()
```

FASTA

Now let's finally get into parsing good, old FASTA files. The `argparse` and `csv` modules are standard in Python, but for FASTA we're going to need to install the BioPython (http://biopython.org/\ module. This should work for you:

```
$ python3 -m pip install biopython
```

For this exercise, I'll use a few reads from the Global Ocean Sampling Expedition (https://imicrobe.us/sample/view/578\:

```
$ head -5 CAM_SMPL_GS108.fa
>CAM_READ_0231669761 /library_id="CAM_LIB_GOS108XLRVAL-4F-1-400"
 /sample_id="CAM_SMPL_GS108" raw_id=SRA_ID=SRR066139.70645 raw_i
d=FG67BMZ02PUFIF
ATTTACAATAATTTAATAAAATTAACTAGAAATAAAATATTGTATGAAAATATGTTAAATAATG
AAAGTTTTT
CAGATCGTTTAATAATATTTTTCTTCCATTTTGCTTTTTTCTAAAATTGTTCAAAAACAAACTT
CAAAGGAAA
ATCTTCAAAATTTACATGATTTTATATTTAAACAAATAGAGTTAAGTATAAGAGAAATTGGATA
TGGTGATGC
TTCAATAAATAAAAAAATGAAAGAGTATGTCAATGTGATGTACGCAATAATTGACAAAGTTGAT
TCATGGGAA
```

Let's write a script that mimics `seqmagick`:

```
$ seqmagick info CAM_SMPL_GS108.fa
name              alignment    min_len   max_len   avg_len  num_
seqs
CAM_SMPL_GS108.fa FALSE            168       440    325.60
   5
```

We'll skip the "alignment" and just do min/max/avg lengths and the number of sequences. You can pretty much copy and paste the example code from http://biopython.org/wiki/SeqIO. Here is the output from our script, "seqmagique.py":

```
$ ./seqmagique.py *.fa
name                 min_len    max_len    avg_len   num_seqs
CAM_SMPL_GS108.fa        168        440      325.6          5
CAM_SMPL_GS112.fa         71        517      398.4          5
```

The code to produce this builds on our earlier skills of lists and dictionaries as we will parse each file and save a dictionary of stats into a list, then we will iterate over that list at the end to show the output.

```
$ cat -n seqmagique.py
     1    #!/usr/bin/env python3
```

```
     2    """Mimic seqmagick, print stats on FASTA sequences"""
     3
     4    import os
     5    import sys
     6    from statistics import mean
     7    from Bio import SeqIO
     8
     9    files = sys.argv[1:]
    10
    11    if len(files) < 1:
    12        print('Usage: {} F1.fa [F2.fa...]'.format(os.path.
basename(sys.argv[0])))
    13        sys.exit(1)
    14
    15    info = []
    16    for file in files:
    17        lengths = []
    18        for record in SeqIO.parse(file, "fasta"):
    19            lengths.append(len(record.seq))
    20
    21        info.append({'name': os.path.basename(file),
    22                     'min_len': min(lengths),
    23                     'max_len': max(lengths),
    24                     'avg_len': mean(lengths),
    25                     'num_seqs': len(lengths)})
    26
    27    if len(info):
    28        longest_file_name = max([len(f['name']) for f in i
nfo])
    29        fmt = '{:' + str(longest_file_name) + '} {:>10} {:
>10} {:>10} {:>10}'
    30        flds = ['name', 'min_len', 'max_len', 'avg_len', '
num_seqs']
    31        print(fmt.format(*flds))
    32        for rec in info:
    33            print(fmt.format(*[rec[fld] for fld in flds]))
    34    else:
    35        print('I had trouble parsing your data')
```

FASTA subset

Sometimes you may only want to use part of a FASTA file, e.g., you want the first 1000 sequences to test some code, or you have samples that vary wildly in size and you want to sub-sample them down to an equal number of reads. Here is a Python program that will write the first N samples to a given output directory:

```
$ cat -n fa_subset.py
     1    #!/usr/bin/env python3
     2    """Subset FASTA files"""
     3
     4    import argparse
     5    import os
     6    import sys
     7    from Bio import SeqIO
     8
     9    # --------------------------------------------------
    10    def get_args():
    11        """get args"""
    12        parser = argparse.ArgumentParser(description='Spli
t FASTA files')
    13        parser.add_argument('fasta', help='FASTA input fil
e', metavar='FILE')
    14        parser.add_argument('-n', '--num', help='Number of
 records per file',
    15                            type=int, metavar='NUM', defau
lt=500000)
    16        parser.add_argument('-o', '--out_dir', help='Outpu
t directory',
    17                            type=str, metavar='DIR', defau
lt='subset')
    18        return parser.parse_args()
    19
    20    # --------------------------------------------------
    21    def main():
    22        """main"""
    23        args = get_args()
    24        fasta = args.fasta
    25        out_dir = args.out_dir
```

```
    num_seqs = args.num

    if not os.path.isfile(fasta):
        print('--fasta "{}" is not valid'.format(fasta
))
        sys.exit(1)

    if os.path.dirname(fasta) == out_dir:
        print('--outdir cannot be the same as input fi
les')
        sys.exit(1)

    if num_seqs < 1:
        print("--num cannot be less than one")
        sys.exit(1)

    if not os.path.isdir(out_dir):
        os.mkdir(out_dir)

    basename = os.path.basename(fasta)
    out_file = os.path.join(out_dir, basename)
    out_fh = open(out_file, 'wt')
    num_written = 0

    for record in SeqIO.parse(fasta, "fasta"):
        SeqIO.write(record, out_fh, "fasta")
        num_written += 1

        if num_written == num_seqs:
            break

    print('Done, wrote {} sequence{} to "{}"'.format(
        num_written, '' if num_written == 1 else 's',
out_file))

# --------------------------------------------------
if __name__ == '__main__':
    main()
```

FASTA splitter

I seem to have implemented my own FASTA splitter a few times in as many languages. Here is one that writes a maximum number of sequences to each output file. It would not be hard to instead write a maximum number of bytes, but, for the short reads I usually handle, this works fine. Again I will use the BioPython `SeqIO` module to parse the FASTA files

```
$ cat -n fasplit.py
     1    #!/usr/bin/env python3
     2    """split FASTA files"""
     3
     4    import argparse
     5    import os
     6    from Bio import SeqIO
     7
     8    # --------------------------------------------------
     9    def main():
    10        """main"""
    11        args = get_args()
    12        fasta = args.fasta
    13        out_dir = args.out_dir
    14        max_per = args.num
    15
    16        if not os.path.isfile(fasta):
    17            print('--fasta "{}" is not valid'.format(fasta
))
    18            exit(1)
    19
    20        if not os.path.isdir(out_dir):
    21            os.mkdir(out_dir)
    22
    23        if max_per < 1:
    24            print("--num cannot be less than one")
    25            exit(1)
    26
    27        i = 0
    28        nseq = 0
    29        nfile = 0
```

```
    30        out_fh = None
    31        basename, ext = os.path.splitext(os.path.basename(
fasta))
    32
    33        for record in SeqIO.parse(fasta, "fasta"):
    34            if i == max_per:
    35                i = 0
    36                if out_fh is not None:
    37                    out_fh.close()
    38                    out_fh = None
    39
    40            i += 1
    41            nseq += 1
    42            if out_fh is None:
    43                nfile += 1
    44                path = os.path.join(out_dir, basename + '.
' + str(nfile) + ext)
    45                out_fh = open(path, 'wt')
    46
    47            SeqIO.write(record, out_fh, "fasta")
    48
    49        print('Done, wrote {} sequence{} to {} file{}'.for
mat(
    50            nseq, '' if nseq == 1 else 's',
    51            nfile, '' if nfile == 1 else 's'))
    52
    53    # --------------------------------------------------
    54    def get_args():
    55        """get args"""
    56        parser = argparse.ArgumentParser(description='Spli
t FASTA files')
    57        parser.add_argument('-f', '--fasta', help='FASTA i
nput file',
    58                            type=str, metavar='FILE', requ
ired=True)
    59        parser.add_argument('-n', '--num', help='Number of
 records per file',
    60                            type=int, metavar='NUM', defau
lt=50)
    61        parser.add_argument('-o', '--out_dir', help='Outpu
```

```
t directory',
    62                              type=str, metavar='DIR', defau
lt='fasplit')
    63        return parser.parse_args()
    64
    65    # --------------------------------------------------
    66    if __name__ == '__main__':
    67        main()
```

If you type `make` in the "python/fasta-splitter" directory, you should see:

```
$ make
./fasplit.py -f POV_L.Sum.O.1000m_reads.fa -o pov -n 100
Done, wrote 2061 sequences to 21 files
```

We can verify that things worked:

```
$ for file in *; do echo -n $file && grep '^>' $file | wc -l; do
ne
POV_L.Sum.O.1000m_reads.1.fa     100
POV_L.Sum.O.1000m_reads.10.fa     100
POV_L.Sum.O.1000m_reads.11.fa     100
POV_L.Sum.O.1000m_reads.12.fa     100
POV_L.Sum.O.1000m_reads.13.fa     100
POV_L.Sum.O.1000m_reads.14.fa     100
POV_L.Sum.O.1000m_reads.15.fa     100
POV_L.Sum.O.1000m_reads.16.fa     100
POV_L.Sum.O.1000m_reads.17.fa     100
POV_L.Sum.O.1000m_reads.18.fa     100
POV_L.Sum.O.1000m_reads.19.fa     100
POV_L.Sum.O.1000m_reads.2.fa     100
POV_L.Sum.O.1000m_reads.20.fa     100
POV_L.Sum.O.1000m_reads.21.fa      61
POV_L.Sum.O.1000m_reads.3.fa     100
POV_L.Sum.O.1000m_reads.4.fa     100
POV_L.Sum.O.1000m_reads.5.fa     100
POV_L.Sum.O.1000m_reads.6.fa     100
POV_L.Sum.O.1000m_reads.7.fa     100
POV_L.Sum.O.1000m_reads.8.fa     100
POV_L.Sum.O.1000m_reads.9.fa     100
```

GFF

Two of the most common output files in bioinformatics, GFF (General Feature Format) and BLAST's tab/CSV files do not include headers, so it's up to you to merge in the headers. Additionally, some of the lines may be comments (they start with "#" just like bash and Python), so you should skip those. Further, the last field in GFF is basically a dumping ground for whatever else the data provider felt like putting there. Usually it's a bunch of "key=value" pairs, but there's no guarantee. Let's take a look at parsing the GFF output from Prodigal:

```
$ cat -n parse_prodigal_gff.py
     1    #!/usr/bin/env python3
     2
```

```
     3    import argparse
     4    import csv
     5    import os
     6    import sys
     7
     8    # --------------------------------------------------
     9    def get_args():
    10        parser = argparse.ArgumentParser(description='Pars
e Prodigal GFF')
    11        parser.add_argument('gff', metavar='GFF', help='Pr
odigal GFF output')
    12        parser.add_argument('-m', '--min', help='Minimum s
core',
    13                            metavar='float', type=float, d
efault=0.0)
    14        return parser.parse_args()
    15
    16    # --------------------------------------------------
    17    def main():
    18        args = get_args()
    19        gff_file = args.gff
    20        min_score = args.min
    21        flds = 'seqname source feature start end score str
and frame attribute'.split()
    22
    23        for line in open(gff_file):
    24            if line[0] == '#':
    25                continue
    26
    27            row = dict(zip(flds, line.split('\t')))
    28            attrs = {}
    29            if 'attribute' in row:
    30                for fld in row['attribute'].split(';'):
    31                    if '=' in fld:
    32                        name, val = fld.split('=')
    33                        attrs[name] = val
    34
    35            if 'score' in attrs and float(attrs['score'])
> min_score:
    36                print(row)
```

```
            break

# ------------------------------------------------
if __name__ == '__main__':
    main()
```

Guessing Game

I firmly believe that writing your own simple games is a terrific way to learn how to program. Let's write a simple program where the user has to guess a random number.

```
$ ./guess.py
[0] Guess a number between 1 and 50 (q to quit): 25
You guessed "25"
Too high.
[1] Guess a number between 1 and 50 (q to quit): 12
You guessed "12"
Too low.
[2] Guess a number between 1 and 50 (q to quit): 20
You guessed "20"
Too high.
[3] Guess a number between 1 and 50 (q to quit): 17
You guessed "17"
Too low.
[4] Guess a number between 1 and 50 (q to quit): 18
You guessed "18"
Too many guesses! The number was "19."
```

To start, we'll use the "new_py.py" script to stub out the boilerplate. I'll use the `-a` flag to indicate that I want the program to use the `argparse` module so we can accept some named arguments to our script:

```
$ new_py.py -a guess
Done, see new script "guess.py."
$ cat -n guess.py
     1    #!/usr/bin/env python3
     2    """docstring"""
     3
     4    import argparse
     5    import sys
     6
```

```
     7    # ------------------------------------------------
     8    def get_args():
     9        """get args"""
    10        parser = argparse.ArgumentParser(description='Argp
arse Python script')
    11        parser.add_argument('positional', metavar='str', h
elp='A positional argument')
    12        parser.add_argument('-a', '--arg', help='A named s
tring argument',
    13                            metavar='str', type=str, defau
lt='')
    14        parser.add_argument('-i', '--int', help='A named i
nteger argument',
    15                            metavar='int', type=int, defau
lt=0)
    16        parser.add_argument('-f', '--flag', help='A boolea
n flag',
    17                            action='store_true')
    18        return parser.parse_args()
    19
    20    # ------------------------------------------------
    21    def main():
    22        """main"""
    23        args = get_args()
    24        str_arg = args.arg
    25        int_arg = args.int
    26        flag_arg = args.flag
    27        pos_arg = args.positional
    28
    29        print('str_arg = "{}"'.format(str_arg))
    30        print('int_arg = "{}"'.format(int_arg))
    31        print('flag_arg = "{}"'.format(flag_arg))
    32        print('positional = "{}"'.format(pos_arg))
    33
    34    # ------------------------------------------------
    35    if __name__ == '__main__':
    36        main()
```

The template shows how to create named arguments for strings, integers, Booleans, as well as positional (unnamed) values. For this program, we want to know the min/max numbers to guess and the number of guesses allowed. We will provide reasonable defaults for all of them so that they will be completely optional. The thing I like best about this step is that it makes me think carefully about what I expect from the user. Change your program to this:

```
def get_args():
    """get args"""
    parser = argparse.ArgumentParser(description='Number guessin
g game')
    parser.add_argument('-m', '--min', help='Minimum value',
                        metavar='int', type=int, default=1)
    parser.add_argument('-x', '--max', help='Maximum value',
                        metavar='int', type=int, default=50)
    parser.add_argument('-g', '--guesses', help='Number of guess
es',
                        metavar='int', type=int, default=5)
    return parser.parse_args()
```

Now in the `main` we will need to unpack the input:

```
def main():
    """main"""
    args = get_args()
    low = args.min
    high = args.max
    guesses_allowed = args.guesses
```

Alway assume you get garbage from the user, so let's check the input:

```
    if low < 1:
        print('--min cannot be lower than 1')
        sys.exit(1)

    if guesses_allowed < 1:
        print('--guesses cannot be lower than 1')
        sys.exit(1)

    if low > high:
        print('--min "{}" is higher than --max "{}"'.format(low,
 high))
        sys.exit(1)
```

The next thing we need is a random number between `--min` and `--max` for the user to guess. We can `import random` to do:

```
secret = random.randint(low, high)
```

The meat of the program will be an infinite loop where we keep asking the user:

```
prompt = 'Guess a number between {} and {} (q to quit): '.format
(low, high)
```

Before we enter that loop, we'll need a variable to keep track of the number of guesses the user has made:

```
num_guesses = 0
```

The beginning of the play loop looks like this:

```
    while True:
        guess = input('[{}] {}'.format(num_guesses, prompt))
        num_guesses += 1
```

Here I want the user to know how many guesses they've made so far. We want to give them a way out, so they can enter "q" to quit:

```
        if guess == 'q':
            print('Now you will never know the answer.')
            sys.exit(0)
```

The input from the user will be a string, and we are going to need to convert it to an integer to see if it is the secret number. Before we do that, we must check that it is a digit:

```
        if not guess.isdigit():
            print('"{}" is not a number'.format(guess))
            continue
```

If it's not a digit, we `continue` to go to the next iteration of the loop. If we move ahead, then it's OK to convert the guess:

```
        print('You guessed "{}"'.format(guess))
        num = int(guess)
```

Now we need to determine if the user has guessed too many times, if the number if too high or low, or if they've won the game:

```
        if num_guesses >= guesses_allowed:
            print('Too many guesses! The number was "{}."'.forma
t(secret))
            sys.exit()
        elif num < low or num > high:
            print('Number is not in the allowed range')
        elif num == secret:
            print('You win!')
            break
        elif num < secret:
            print('Too low.')
        else:
            print('Too high.')
```

The final version looks like this:

```
$ cat -n guess.py
     1    #!/usr/bin/env python3
     2    """guess the number game"""
     3
     4    import argparse
     5    import random
     6    import sys
     7
     8    # --------------------------------------------------
     9    def get_args():
    10        """get args"""
    11        parser = argparse.ArgumentParser(description='Numb
er guessing game')
    12        parser.add_argument('-m', '--min', help='Minimum v
alue',
    13                            metavar='int', type=int, defau
lt=1)
    14        parser.add_argument('-x', '--max', help='Maximum v
alue',
    15                            metavar='int', type=int, defau
lt=50)
    16        parser.add_argument('-g', '--guesses', help='Numbe
```

```
r of guesses',
    17                              metavar='int', type=int, defau
lt=5)
    18        return parser.parse_args()
    19
    20    # --------------------------------------------------
    21    def main():
    22        """main"""
    23        args = get_args()
    24        low = args.min
    25        high = args.max
    26        guesses_allowed = args.guesses
    27
    28        if low < 1:
    29            print('--min cannot be lower than 1')
    30            sys.exit(1)
    31
    32        if guesses_allowed < 1:
    33            print('--guesses cannot be lower than 1')
    34            sys.exit(1)
    35
    36        if low > high:
    37            print('--min "{}" is higher than --max "{}"'.f
ormat(low, high))
    38            sys.exit(1)
    39
    40        secret = random.randint(low, high)
    41        num_guesses = 0
    42        prompt = 'Guess a number between {} and {} (q to q
uit): '.format(low, high)
    43
    44        while True:
    45            guess = input('[{}] {}'.format(num_guesses, pr
ompt))
    46            num_guesses += 1
    47
    48            if guess == 'q':
    49                print('Now you will never know the answer.
')
    50                sys.exit(0)
```

```
    51
    52            if not guess.isdigit():
    53                print('"{}" is not a number'.format(guess)
)
    54                continue
    55
    56            print('You guessed "{}"'.format(guess))
    57            num = int(guess)
    58
    59            if num_guesses >= guesses_allowed:
    60                print('Too many guesses! The number was "{
}."'.format(secret))
    61                sys.exit()
    62            elif num < low or num > high:
    63                print('Number is not in the allowed range'
)
    64            elif num == secret:
    65                print('You win!')
    66                break
    67            elif num < secret:
    68                print('Too low.')
    69            else:
    70                print('Too high.')
    71
    72    # ----------------------------------------------------
    73    if __name__ == '__main__':
    74        main()
```

Hangman

Here is an implementation of the game "Hangman" that uses dictionaries to maintain the "state" of the program -- that is, all the information needed for each round of play such as the word being guessed, how many misses the user has made, which letters have been guessed, etc. The program uses the `argparse` module to gather options from the user while providing default values so that nothing needs to be provided. The `main` function is used just to gather the parameters and then run the `play` function which recursively calls itself, each

time passing in the new "state" of the program. Inside `play`, we use the `get` method of `dict` to safely ask for keys that may not exist and use defaults. When the user finishes or quits, `play` will simply call `sys.exit` to stop. Here is the code:

```
$ cat -n hangman.py
     1    #!/usr/bin/env python3
     2    """Hangman game"""
     3
     4    import argparse
     5    import os
     6    import random
     7    import re
     8    import sys
     9
    10    # --------------------------------------------------
    11    def get_args():
    12        """parse arguments"""
    13        parser = argparse.ArgumentParser(description='Hang
man')
    14        parser.add_argument('-l', '--maxlen', help='Max wo
rd length',
    15                            type=int, default=10)
    16        parser.add_argument('-n', '--minlen', help='Min wo
rd length',
    17                            type=int, default=5)
    18        parser.add_argument('-m', '--misses', help='Max nu
mber of misses',
    19                            type=int, default=10)
    20        parser.add_argument('-w', '--wordlist', help='Word
 list',
    21                            type=str, default='/usr/share/
dict/words')
    22        return parser.parse_args()
    23
    24    # --------------------------------------------------
    25    def main():
    26        """main"""
    27        args = get_args()
```

```
    28        max_len = args.maxlen
    29        min_len = args.minlen
    30        max_misses = args.misses
    31        wordlist = args.wordlist
    32
    33        if not os.path.isfile(wordlist):
    34            print('--wordlist "{}" is not a file.'.format(
wordlist))
    35            sys.exit(1)
    36
    37        if min_len < 1:
    38            print('--minlen must be positive')
    39            sys.exit(1)
    40
    41        if not 3 <= max_len <= 20:
    42            print('--maxlen should be between 3 and 20')
    43            sys.exit(1)
    44
    45        if min_len > max_len:
    46            print('--minlen ({}) is greater than --maxlen
({})'.format(min_len, max_len))
    47            sys.exit(1)
    48
    49        regex = re.compile('^[a-z]{' + str(min_len) + ','
+ str(max_len) + '}$')
    50        words = [w for w in open(wordlist).read().split()
if regex.match(w)]
    51        word = random.choice(words)
    52        play({'word': word, 'max_misses': max_misses})
    53
    54    # -----------------------------------------------
    55    def play(state):
    56        """Loop to play the game"""
    57        word = state.get('word') or ''
    58
    59        if not word:
    60            print('No word!')
    61            sys.exit(1)
    62
    63        guessed = state.get('guessed') or list('_' * len(w
```

```
ord))
    64        prev_guesses = state.get('prev_guesses') or set()
    65        num_misses = state.get('num_misses') or 0
    66        max_misses = state.get('max_misses') or 0
    67
    68        if ''.join(guessed) == word:
    69            msg = 'You win. You guessed "{}" with "{}" mis
s{}!'
    70            print(msg.format(word, num_misses, '' if num_m
isses == 1 else 'es'))
    71            sys.exit(0)
    72
    73        if num_misses >= max_misses:
    74            print('You lose, loser!  The word was "{}."'.f
ormat(word))
    75            sys.exit(0)
    76
    77        print('{} (Misses: {})'.format(' '.join(guessed),
num_misses))
    78        new_guess = input('Your guess? ("?" for hint, "!"
to quit) ').lower()
    79
    80        if new_guess == '!':
    81            print('Better luck next time, loser.')
    82            sys.exit(0)
    83        elif new_guess == '?':
    84            new_guess = random.choice([x for x in word if
x not in guessed])
    85            num_misses += 1
    86
    87        if not re.match('^[a-zA-Z]$', new_guess):
    88            print('"{}" is not a letter'.format(new_guess)
)
    89            num_misses += 1
    90        elif new_guess in prev_guesses:
    91            print('You already guessed that')
    92        elif new_guess in word:
    93            prev_guesses.add(new_guess)
    94            last_pos = 0
    95            while True:
```

```
    96                 pos = word.find(new_guess, last_pos)
    97                 if pos < 0:
    98                     break
    99                 elif pos >= 0:
   100                     guessed[pos] = new_guess
   101                     last_pos = pos + 1
   102         else:
   103             num_misses += 1
   104
   105         play({'word': word, 'guessed': guessed, 'num_misse
s': num_misses,
   106               'prev_guesses': prev_guesses, 'max_misses':
max_misses})
   107
   108     # ------------------------------------------------
   109     if __name__ == '__main__':
   110         main()
```

SQLite in Python

SQLite (https://www.sqlite.org\ is a lightweight, SQL/relational database that is available by default with Python (https://docs.python.org/3/library/sqlite3.html\. By using `import sqlite3` you can interact with an SQLite database. So, let's create one, returning to our earlier Centrifuge output. Here is the file "tables.sql" containing the SQL statements needed to drop and create the tables:

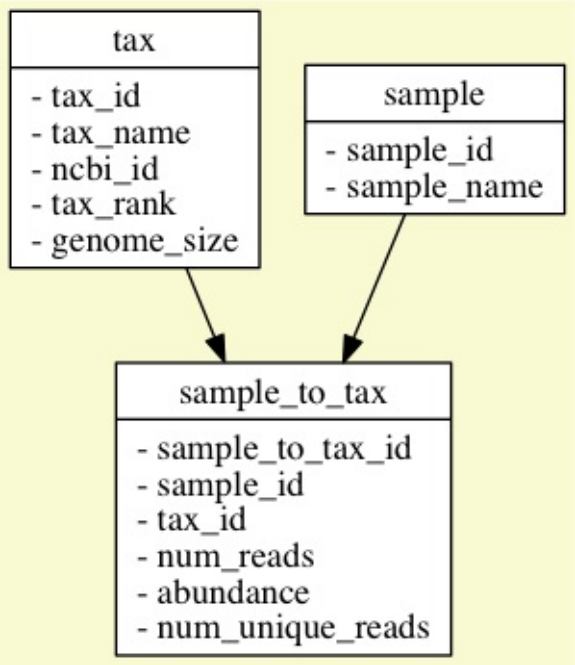
```
drop table if exists tax;
create table tax (
    tax_id integer primary key,
    tax_name text not null,
    ncbi_id int not null,
    tax_rank text default '',
    genome_size int default 0,
    unique (ncbi_id)
);

drop table if exists sample;
create table sample (
    sample_id integer primary key,
    sample_name text not null,
    unique (sample_name)
);

drop table if exists sample_to_tax;
create table sample_to_tax (
    sample_to_tax_id integer primary key,
    sample_id int not null,
    tax_id int not null,
    num_reads int default 0,
    abundance real default 0,
    num_unique_reads integer default 0,
    unique (sample_id, tax_id),
    foreign key (sample_id) references sample (sample_id),
    foreign key (tax_id) references tax (tax_id)
);
```

Like Python, has data types of strings, integers, and floats (https://sqlite.org/datatype3.html\). Primary keys are unique values defining a record in a table. You can place constraints on the allowed values of a field with conditions like `default` values or `not null` requirements as well as having the database enforce that some values are `unique` (such as NCBI taxonomy IDs). You can also require that a particular combination of fields be unique, e.g., the sample/tax table has a unique constraint on the pairing of the sample/tax IDs. Additionally, this database uses foreign keys (https://sqlite.org/foreignkeys.html\) to maintain relationships between tables. We will see in a moment how that prevents us from accidentally creating "orphan" records.

We are going to create a minimal database to track the abundance of species in various samples. The biggest rule of relational databases is to not repeat data. There should be one place to store each entity. For us, we have a "sample" (the Centrifuge ".tsv" file), a "taxonomy" (NCBI tax ID/name), and the relationship of the sample to the taxonomy. I have my own particular naming convention when it comes to relational tables/fields:

1. Name tables in the singular, e.g. "sample" not "samples"
2. Name the primary key [tablename] + underscore + "id", e.g., "sample_id"
3. Name linking tables [table1] + underscore + "to" + underscore + [table2]
4. Always have a primary key that is an auto-incremented integer

Here is a simple E/R (entity-relationship) graph of the schema (created with SQL::Translator):



You can instantiate the database by calling `make db` in the "csv" directory to *first remove the existing database* and then recreate it by redirecting the "tables.sql" file into `sqlite3` :

```
$ make db
find . -name centrifuge.db -exec rm {} \;
sqlite3 centrifuge.db < tables.sql
```

You can then run `sqlite3 centrifuge.db` to use the CLI (command-line interface) to the database. Use `.help` inside SQLite to see all the "dot" commands (they begin with a `.` , cf. https://sqlite.org/cli.html\):

```
$ sqlite3 centrifuge.db
SQLite version 3.13.0 2016-05-18 10:57:30
Enter ".help" for usage hints.
sqlite>
```

I often rely on the `.schema` command to look at the tables in an SQLite db. If you run that, you should see essentially the same thing as was in the "tables.sql" file. An alternate way to create the database is to use the `.read tables.sql` command from within SQLite to have it read and execute the SQL statements in that file.

We can manually insert a record into the `tax` table with an `insert` statement (https://sqlite.org/lang_insert.html\). Note how SQLite treats strings and numbers exactly like Python -- strings must be in quotes, numbers should be plain:

```
sqlite> insert into tax (tax_name, ncbi_id) values ('Homo sapien
s', 3606);
```

We can add a dummy "sample" and link them like so:

```
sqlite> insert into sample (sample_name) values ('foo');
sqlite> insert into sample_to_tax (sample_id, tax_id, num_reads,
 abundance)
   ...> values (1, 1, 100, .01);
```

Verify that the data is there with a `select` statement (https://sqlite.org/lang_select.html\):

```
sqlite> select count(*) from tax;
1
sqlite> select * from tax;
1|Homo sapiens|3606||0
```

Use `.headers on` to see the column names:

```
sqlite> .headers on
sqlite> select * from tax;
tax_id|tax_name|ncbi_id|tax_rank|genome_size
1|Homo sapiens|3606||0
sqlite> select * from sample;
sample_id|sample_name
1|foo
```

That's still a bit hard to read, so we can set `.mode column` to see a bit better:

```
sqlite> select * from sample;
sample_id   sample_name
----------  -----------
1           foo
sqlite> select * from tax;
tax_id      tax_name      ncbi_id     tax_rank    genome_size
----------  ------------  ----------  ----------  -----------
1           Homo sapiens  3606                    0
sqlite> select * from sample_to_tax;
sample_to_tax_id  sample_id   tax_id      num_reads   abundance
  num_unique_reads
----------------  ----------  ----------  ----------  ----------
  ----------------
1                 1           1           100         0.01
  0
```

Often what we want is to `join` the tables so we can see just the data we want, e.g.:

```
sqlite> select s.sample_name, t.tax_name, s2t.num_reads
   ...> from sample s, tax t, sample_to_tax s2t
   ...> where s.sample_id=s2t.sample_id
   ...> and s2t.tax_id=t.tax_id;
sample_name  tax_name      num_reads
-----------  ------------  ----------
foo          Homo sapiens  100
```

Now let's try to delete the `sample` record after we have turned on the enforcement of foreign keys:

```
sqlite> PRAGMA foreign_keys = ON;
sqlite> delete from sample where sample_id=1;
Error: FOREIGN KEY constraint failed
```

It would be bad to remove our sample and leave the sample/tax records in place. This is what foreign keys do for us. (Other databases -- PostgreSQL, MySQL, Oracle, etc. -- do this without having to explicitly turn on this feature, but keep in

mind that this is an extremely lightweight, fast, and easy database to create and administer. When you need more speed/power/safety, then you will move to another database.)

Obviously we're not going to manually enter our data by hand, so let's write a script to import some data. This script is going to be somewhat long, so let's break it down. Here's the start. We need to take as arguments the Centrifuge "*.tsv" (tab-separated values) file which is the summary table for all the species found in a given sample. The script will take one or more of these positional arguments. It will also take as a named argument the `--db` name of the SQLite database. Note that the `sqlite3` module is available by default with Python -- no need to install anything!

```
#!/usr/bin/env python3
"""Load Centrifuge into SQLite db"""

import argparse
import csv
import os
import re
import sqlite3
import sys

# --------------------------------------------------
def get_args():
    """get args"""
    parser = argparse.ArgumentParser(description='Load Centrifug
e data')
    parser.add_argument('tsv_file', metavar='file',
                        help='Sample TSV file', nargs='+')
    parser.add_argument('-d', '--dbname', help='Centrifuge db na
me',
                        metavar='str', type=str, default='centri
fuge.db')
    return parser.parse_args()
```

Our `main` is going to handle the arguments, ensuring the `--dbname` is a valid file, then processing each of the `tsv_file` arguments (note the `nargs` declaration to show that the program takes one or more TSV files). Note that in order to keep this function short, I created two other functions, to import the samples and TSV files:

```
# --------------------------------------------------
def main():
    """main"""
    args = get_args()
    tsv_files = args.tsv_file
    dbname = args.dbname

    if not os.path.isfile(dbname):
        print('Bad --dbname "{}"'.format(dbname))
        sys.exit(1)

    db = sqlite3.connect(dbname)

    for fnum, tsv_file in enumerate(tsv_files):
        if not os.path.isfile(tsv_file):
            print('Bad tsv_file "{}"'.format(tsv_file))
            sys.exit(1)

        sample_name, ext = os.path.splitext(tsv_file)

        if ext != '.tsv':
            print('"{}" does not end with ".tsv"'.format(tsv_fil
e))
            sys.exit(1)

        if sample_name.endswith('.centrifuge'):
            sample_name = re.sub(r'\.centrifuge$', '', sample_na
me)

        sample_id = import_sample(sample_name, db)
        print('{:3}: Importing "{}" ({})'.format(fnum + 1,
                                                 sample_name, sa
mple_id))
        import_tsv(db, tsv_file, sample_id)

    print('Done')
```

Here is the code to import a "sample." It needs a sample_name (which we assume to be unique) and a database handle (which is a bit like filehandles which we've been dealing with -- it's the actual conduit from your code to the database). First we have to check if the sample already exists in our table, and this requires we use a `cursor` (https://docs.python.org/3/library/sqlite3.html\) to issue our `select` statement. Rather than putting the sample name directly into the SQL (which is very insecure, see SQL injection/"Bobby Tables" XKCD https://xkcd.com/327\), we use a `?` and pass the string as an argument to the `execute` function. If nothing (`None`) is returned, we can safely `insert` the new record and get the newly created sample ID from the `lastrowid` function of the cursor; otherwise, the sample ID is in the `res` result list as the first field:

```
# -------------------------------------------------
def import_sample(sample_name, db):
    """Import sample"""
    cur = db.cursor()
    cur.execute('select sample_id from sample where sample_name=
?',
                (sample_name,))
    res = cur.fetchone()

    if res is None:
        cur.execute('insert into sample (sample_name) values (?)
',
                    (sample_name,))
        sample_id = cur.lastrowid
    else:
        sample_id = res[0]

    return sample_id
```

The code to import the TSV file is similar. We establish SQL statements to find/insert/update the sample/tax record, then we use the `csv` module to parse the TSV file, creating dictionaries of each record (a product of merging the first line/headers with each row of data). Again, to keep this function short enough to fit on a "page," there is a separate function to find or create the taxonomy record.

```
# ------------------------------------------------
def import_tsv(db, file, sample_id):
    """Import TSV file"""
    find_sql = """
        select sample_to_tax_id
        from   sample_to_tax
        where  sample_id=?
        and    tax_id=?
    """

    insert_sql = """
        insert
        into   sample_to_tax
               (sample_id, tax_id, num_reads, abundance, num_uni
que_reads)
        values (?, ?, ?, ?, ?)
    """

    update_sql = """
        update sample_to_tax
        set    sample_id=?, tax_id=?, num_reads=?,
               abundance=?, num_unique_reads=?
        where  sample_to_tax_id=?
    """

    cur = db.cursor()
    with open(file) as csvfile:
        reader = csv.DictReader(csvfile, delimiter='\t')
        for row in reader:
            tax_id = find_or_create_tax(db, row)
            if tax_id:
                cur.execute(find_sql, (sample_id, tax_id))
                res = cur.fetchone()
                num_reads = row.get('numReads', 0)
                abundance = row.get('abundance', 0)
                num_uniq = row.get('numUniqueReads', 0)

                if res is None:
                    cur.execute(insert_sql,
```

```
                                (sample_id, tax_id, num_reads,
                                 abundance, num_uniq))
                else:
                    s2t_id = res[0]
                    cur.execute(update_sql,
                                (sample_id, tax_id, num_reads,
                                 abundance, num_uniq, s2t_id))
            else:
                print('No tax id!')

        db.commit()

    return 1
```

The find/create tax function works just the same as that for the sample:

```
# --------------------------------------------------
def find_or_create_tax(db, rec):
    """find or create the tax"""
    find_sql = 'select tax_id from tax where ncbi_id=?'
    insert_sql = """
        insert into tax (tax_name, ncbi_id, tax_rank, genome_siz
e)
        values (?, ?, ?, ?)
    """

    cur = db.cursor()
    ncbi_id = rec.get('taxID', '')
    if re.match('^\d+$', ncbi_id):
        cur.execute(find_sql, (ncbi_id,))
        res = cur.fetchone()

        if res is None:
            name = rec.get('name', '')
            if name:
                print('Loading "{}" ({})'.format(name, ncbi_id))
                cur.execute(insert_sql,
                            (name, ncbi_id, rec['taxRank'],
                             rec['genomeSize']))
                tax_id = cur.lastrowid
            else:
                print('No "name" in {}'.format(rec))
                return None
        else:
            tax_id = res[0]

        return tax_id
    else:
        print('"{}" does not look like an NCBI tax id'.format(nc
bi_id))
        return None
```

If you use `make data`, several files will be downloaded from the iMicrobe FTP site for use by the `make load` step run the loader program:

```
$ make load
./load_centrifuge.py *.tsv
  1: Importing "YELLOWSTONE_SMPL_20717" (1)
Loading "Synechococcus sp. JA-3-3Ab" (321327)
Loading "Synechococcus sp. JA-2-3B'a(2-13)" (321332)
  2: Importing "YELLOWSTONE_SMPL_20719" (2)
Loading "Streptococcus suis" (1307)
Loading "synthetic construct" (32630)
  3: Importing "YELLOWSTONE_SMPL_20721" (3)
Loading "Staphylococcus sp. AntiMn-1" (1715860)
  4: Importing "YELLOWSTONE_SMPL_20723" (4)
  5: Importing "YELLOWSTONE_SMPL_20725" (5)
  6: Importing "YELLOWSTONE_SMPL_20727" (6)
Done
```

Now we can inspect how many records were loaded into the database:

```
$ sqlite3 centrifuge.db
SQLite version 3.13.0 2016-05-18 10:57:30
Enter ".help" for usage hints.
sqlite> select count(*) from tax;
5
sqlite> select count(*) from sample;
6
sqlite> select count(*) from sample_to_tax;
18
```

But, again, we're not going to just sit here and manually write SQL to check out the data. Let's write a program that takes an NCBI tax id as an argument and reports the samples where it is found. You will need to `make tabulate` to run the command to install the "tabulate" module (https://pypi.python.org/pypi/tabulate\) in order to run this program:

```
     1    #!/usr/bin/env python3
     2    """Query centrifuge.db for NCBI tax id"""
     3
     4    import argparse
```

```
     5    import os
     6    import re
     7    import sys
     8    import sqlite3
     9    from tabulate import tabulate
    10
    11    # --------------------------------------------------
    12    def get_args():
    13        """get args"""
    14        parser = argparse.ArgumentParser(description='Argp
arse Python script')
    15        parser.add_argument('-d', '--dbname', help='Centri
fuge db name',
    16                            metavar='str', type=str, defau
lt='centrifuge.db')
    17        parser.add_argument('-o', '--orderby', help='Order
 by',
    18                            metavar='str', type=str, defau
lt='abundance')
    19        parser.add_argument('-s', '--sortorder', help='Sor
t order',
    20                            metavar='str', type=str, defau
lt='desc')
    21        parser.add_argument('-t', '--taxid', help='NCBI ta
xonomy id',
    22                            metavar='str', type=str, requi
red=True)
    23        return parser.parse_args()
    24
    25    # --------------------------------------------------
    26    def main():
    27        """main"""
    28        args = get_args()
    29        dbname = args.dbname
    30        order_by = args.orderby
    31        sort_order = args.sortorder
    32
    33        if not os.path.isfile(dbname):
    34            print('"{}" is not a valid file'.format(dbname
))
```

```
    35            sys.exit(1)
    36
    37        flds = set(['tax_name', 'num_reads', 'abundance',
'sample_name'])
    38        if not order_by in flds:
    39            print('"{}" not an allowed --orderby, choose f
rom {}'.format(
    40                order_by, ', '.join(flds)))
    41            sys.exit(1)
    42
    43        sorting = set(['asc', 'desc'])
    44        if not sort_order in sorting:
    45            print('"{}" not an allowed --sortorder, choose
 from {}'.format(
    46                order_by, ', '.join(sorting)))
    47            sys.exit(1)
    48
    49        tax_ids = []
    50        for tax_id in re.split(r'\s*,\s*', args.taxid):
    51            if re.match(r'^\d+$', tax_id):
    52                tax_ids.append(tax_id)
    53            else:
    54                print('"{}" does not look like an NCBI tax
 id'.format(tax_id))
    55
    56        if len(tax_ids) == 0:
    57            print('No tax ids')
    58            sys.exit(1)
    59
    60        db = sqlite3.connect(dbname)
    61        cur = db.cursor()
    62        sql = """
    63            select   s.sample_name, t.tax_name, s2t.num_re
ads, s2t.abundance
    64            from     sample s, tax t, sample_to_tax s2t
    65            where    s.sample_id=s2t.sample_id
    66            and      s2t.tax_id=t.tax_id
    67            and      t.ncbi_id in ({})
    68            order by {} {}
    69        """.format(', '.join(tax_ids), order_by, sort_orde
```

```
r)
    70
    71        cur.execute(sql)
    72
    73        samples = cur.fetchall()
    74        if len(samples) > 0:
    75            cols = [d[0] for d in cur.description]
    76            print(tabulate(samples, headers=cols))
    77        else:
    78            print('No results')
    79
    80    # ------------------------------------------------
    81    if __name__ == '__main__':
    82        main()
```

It takes as arguments a required NCBI tax id that can be a single value or a comma-separated list. Options include the SQLite Centrifuge db, a column name to sort by, and whether to show in ascending or descending order. The output is formatted with the `tabulate` module to produce a simple text table. To query by one tax ID:

```
$ ./query_centrifuge.py -t 321327
sample_name             tax_name                       num_reads
   abundance
----------------------  -------------------------  -----------
 -----------
YELLOWSTONE_SMPL_20721  Synechococcus sp. JA-3-3Ab          315
        0.98
YELLOWSTONE_SMPL_20723  Synechococcus sp. JA-3-3Ab         6432
        0.98
YELLOWSTONE_SMPL_20727  Synechococcus sp. JA-3-3Ab         1219
        0.96
YELLOWSTONE_SMPL_20717  Synechococcus sp. JA-3-3Ab           19
        0.53
YELLOWSTONE_SMPL_20719  Synechococcus sp. JA-3-3Ab          719
        0.27
YELLOWSTONE_SMPL_20725  Synechococcus sp. JA-3-3Ab         3781
        0.2
```

To query by more than one:

```
$ ./query_centrifuge.py -t 321327,1307
sample_name              tax_name                       num_reads
   abundance
----------------------  -------------------------  -----------
 -----------
YELLOWSTONE_SMPL_20721  Synechococcus sp. JA-3-3Ab          315
        0.98
YELLOWSTONE_SMPL_20723  Synechococcus sp. JA-3-3Ab         6432
        0.98
YELLOWSTONE_SMPL_20727  Synechococcus sp. JA-3-3Ab         1219
        0.96
YELLOWSTONE_SMPL_20717  Synechococcus sp. JA-3-3Ab           19
        0.53
YELLOWSTONE_SMPL_20719  Synechococcus sp. JA-3-3Ab          719
        0.27
YELLOWSTONE_SMPL_20725  Synechococcus sp. JA-3-3Ab         3781
        0.2
YELLOWSTONE_SMPL_20719  Streptococcus suis                   1
        0
```

To order by "num_reads" instead of "abundance":

```
$ ./query_centrifuge.py -t 321327,1307 -o num_reads
sample_name              tax_name                      num_reads
   abundance
----------------------  -------------------------  -----------
 -----------
YELLOWSTONE_SMPL_20723  Synechococcus sp. JA-3-3Ab         6432
        0.98
YELLOWSTONE_SMPL_20725  Synechococcus sp. JA-3-3Ab         3781
        0.2
YELLOWSTONE_SMPL_20727  Synechococcus sp. JA-3-3Ab         1219
        0.96
YELLOWSTONE_SMPL_20719  Synechococcus sp. JA-3-3Ab          719
        0.27
YELLOWSTONE_SMPL_20721  Synechococcus sp. JA-3-3Ab          315
        0.98
YELLOWSTONE_SMPL_20717  Synechococcus sp. JA-3-3Ab           19
        0.53
YELLOWSTONE_SMPL_20719  Streptococcus suis                    1
        0
```

To sort ascending:

```
$ ./query_centrifuge.py -t 321327,1307 -o num_reads -s asc
sample_name              tax_name                      num_reads
   abundance
----------------------  -------------------------  -----------
 -----------
YELLOWSTONE_SMPL_20719  Streptococcus suis                   1
        0
YELLOWSTONE_SMPL_20717  Synechococcus sp. JA-3-3Ab          19
        0.53
YELLOWSTONE_SMPL_20721  Synechococcus sp. JA-3-3Ab         315
        0.98
YELLOWSTONE_SMPL_20719  Synechococcus sp. JA-3-3Ab         719
        0.27
YELLOWSTONE_SMPL_20727  Synechococcus sp. JA-3-3Ab        1219
        0.96
YELLOWSTONE_SMPL_20725  Synechococcus sp. JA-3-3Ab        3781
        0.2
YELLOWSTONE_SMPL_20723  Synechococcus sp. JA-3-3Ab        6432
        0.98
```