Chapter 1

BE 434/535 Biosystems Analytics

The field of "bioinformatics" is biology plus data science. This course assumes you have some understanding of the Unix command line and some programming experience. At the conclusion of this course, you should be able to:

- Write, test, and document programs in bash and Python
- Use the source code management system Git to version, share, and distribute code
- Use parallelization techniques and hardware (HPC) to run programs faster
- Package and distribute software to create reproducible workflows

Ocelote (UA HPC)

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.

To gain access to Ocelote, students must:

1) Have a terminal application ("Terminal" or "iTerm2" on OSX, Gitbash on Windows)
2) Sign up for UA's NetID+ (https://netid-plus.arizona.edu/)
3) Be sponsored (https://portal.hpc.arizona.edu/portal/)

Once you have these, open a terminal and type:

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

You will be prompted for additional authentication for NetID+. If all goes well, you should see something like this:

```
Last login: Sat Jan  5 07:53:30 2019 from ip72-200-123-88.tc.ph.cox.net
This is a bastion host used to access the rest of the environment.

Shortcut commands to access each resource
-----------------------------------------
Ocelote:                El Gato:
$ ocelote               $ elgato
```

Or you may see this (e.g., if you have enabled the menu with the `menuon` command):

```
===============
HPC.ARIZONA.EDU
```

```
==============
Please select a target system to connect to:

(1) Ocelote
(2) El Gato
(Q) Quit
(D) Disable menu
```

Either way, proceed to log in to Ocelote for your work. If you are uncertain which machine you are on, use `hostname`. If you are on the bastion host, it will be "gatekeeper.hpc.arizona.edu." Once you are on Ocelote, the hostname should be something like "login1" or "login2."

SSH Keys

If you would like to avoid the 2-factor authentication, then copy your SSH private key to the target system like so:

1) On your local machine and `cd ~/.ssh`. If you do not have one, then run `ssh-keygen`.
2) Copy the contents of the `id_rsa.pub` file (the "public" part of your key). If you do not have one, then run `ssh-keygen`.
3) Login to the target system and `cd ~/.ssh`. If you do not have one, then run `ssh-keygen`.
4) Edit the `authorized_keys` file (e.g., with `nano`) and paste in the public key. Save and exit your editor.
5) Set the permissions with `chmod 600 authorized_keys`
6) Test your login from your local machine.

Git

You will use the "Git" source code management system (https://git-scm.com/) get class materials, maintain versions of your code, and turn in your assignments. Git was originally created by Linus Torvalds, the creator of Linux, for managing the Linux source code.

Github

Github is a commercially company that hosts Git repositories. We will use their free service to host public repositories, but they make money by hosting private repos. Recently Github was purchased by Microsoft, so they probably will be around at least until the end of the semester. The advantage to use Github (or Gitlab) is that, when you `push` your code to Github, you will have an "off-site"

backup. That is, if your computer were to crash, a copy of the repo would still exist on Github's server. It is not necessary to use Github to use Git. You can `git init` and directory you'd like and maintain a repo there. Github has a useful web interface and the ability to add your instructors as "collaborators" allowing us to `push` and `pull` from your repos.

So, step one is to create a Github account and then share your username with your instructors. I suggest you add your public SSH key (see "SSH Keys" above) into your Github "Settings/SSH and GPG Keys" so that you can more easily push and pull into your repositories. You'll need to add a key from each machine you intend to work from, i.e., both your laptop and Ocelote.

Forking the course repo

Once you have an account, visit the course repo in a web browser. Create a copy of our repository into your own account by clicking the "Fork" button in the upper-right corner.

`https://github.com/hurwitzlab/biosys-analytics`

You can see your Github repo from github.com, but you must use the command line `git` program to check it out, add files, etc. If you are working on Ocelote or any other Unix platform, it's quite likely that Git is already installed. Check the version like so:

```
[hpc:login3@~]$ git --version
git version 2.2.2
```

If you are on Windows, you will probably find it easiest to install "Gitbash" which will give you a Unix-like command line with `git` and many other Unix utilities (but apparently not `wget`).

Cloning your repo locally

Now you can `clone` your repository to your machine(s) like so:

```
$ git clone git@github.com:yourusername/biosys-analytics.git
Cloning into 'biosys-analytics'...
remote: Enumerating objects: 16, done.
remote: Counting objects: 100% (16/16), done.
remote: Compressing objects: 100% (14/14), done.
remote: Total 16 (delta 1), reused 12 (delta 0), pack-reused 0
Receiving objects: 100% (16/16), 104.71 KiB | 0 bytes/s, done.
Resolving deltas: 100% (1/1), done.
Checking connectivity... done.
```

Adding course repo upstream

Your version of the repository is a copy of the "hurwitzlab" repo at the time you forked it. To get new content from the "hurwitzlab" repo, you will add the "hurwitzlab" Github repo as an "upstream" repo:

```
git remote add upstream https://github.com/hurwitzlab/biosys-analytics.git
```

When you need to get new content from the "hurwitzlab" repo, `pull` from the upstream repo's "master" branch:

```
git pull upstream master
```

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 create one.

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

About The Author

> "Computer programming has always been a self-taught, maverick occupation." - Ellen Ullman

My undergraduate degree was in English and music. I learned to program on my own and on the job starting in 1995, so I started relatively late and wasn't formally trained in programming until much later in my MS program. I say this so you know that everyone starts somewhere, some later than others, but it's like the joke "When is the best time to plant a tree? Thirty years ago. When is the second best time? Now." Now is a great time to become a programmer. I look forward to teaching you what I've learned in course of 20+ years of programming in industry and bioinformatics.

Chapter 2

Bioinformatics

Want to be a Bioinformatician? It's easy. Just learn everything you can about:

- Genetics
- Molecular Biology
- Disease phenotypes
- >3 programming languages
- Software Development frameworks
- Best Practices
- Version control
- HPC
- Systems administration
- Converting all Data into Excel

Steven Hart

Unix Basics

Most of bioinformatics happens on Unix-like operating systems. Almost all our tools run from the Unix command line, so bioinformaticians must know how to move data around, run programs, and chain the output of one program into another to create analysis pipelines.

NB: When you see a `$` given in the example prompts, it is a metacharacter indicating that this is the prompt for a normal (not super-user) account. You should type/copy/paste all the stuff *after* the `$`. If you ever see a prompt with "#" in a tutorial, 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.

Common Unix Commands

> "The good news about computers is that they do what you tell them to do. The bad news is that they do what you tell them to do." - Ted Nelson

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
- **xargs**: take a list from one command, concatenate and pass as the arguments to another command

The Unix filesystem hierarchy

The Unix filesystem can thought of as a graph or a tree. The root is "/" and is called the "root directory." We can `ls` or `tree` commands to see what's there:

Tree listing of Ocelote's root directory (`tree -d -L 1 /`).

Try running `tree $HOME` to see what's there.

Moving around the filesystem

You can print your current working directory either with `pwd` or `echo $PWD`.

The `cd` command is used to "change directory," e.g., `cd /rsgrps/bh_class/`. If you wish to return to the previous working directory, use `cd -`.

If you provide no argument to `cd`, it will change to your `$HOME` directory which is also known in bash by the `~` (tilde or twiddle). So these three commands are equivalent:

- `cd`
- `cd ~`
- `cd $HOME`

Once you are in a directory, use `ls` to inspect the contents. If you do not provide an argument to `ls`, it assumes the current directory which has the alias `.` The parent directory is `..`

You can use both absolute and relative paths with `cd`. An absolute path starts from the root directory, e.g., "/usr/local/bin/". A relative path does not start with the leading `/` and assumes a path relative to your current working directory. If you were in the "/usr/local" directory and wanted to change to "/usr/local/bin", you could either `cd /usr/local/bin` (absolute) or `cd bin` (relative to "/usr/local").

Once you are in "/usr/local/bin", what would `pwd` show after you did `cd ../..`?

Chaining commands

> "Programming is breaking of one big impossible task into several very small possible tasks." - Jazzwant

Most Unix commands use STDIN ("standard in"), STDOUT ("standard out"), and STDERR ("standard error"). For instance, the `env` program will show you key/value pairs that describe. your environment – things like your user name (`$USER`), your shell (`$SHELL`), your current working directory (`$PWD`). It can be quite a long list, so you could send the output (STDOUT) of `env` to `head` to see just the first few lines:

```
$ env | head
TERM_PROGRAM=iTerm.app
TERM=xterm-256color
SHELL=/bin/bash
TMPDIR=/var/folders/0h/vjzkyy052qx4p70trn2p2h400000gn/T/
PERL5LIB=/Users/kyclark/work/imicrobe/lib
Apple_PubSub_Socket_Render=/private/tmp/com.apple.launchd.MoIOCra0uS/Render
TERM_PROGRAM_VERSION=3.2.6
OLDPWD=/Users/kyclark/work/biosys-analytics/lectures
TERM_SESSION_ID=w2t0p0:34E89330-9CEE-4FA6-9039-3674CBDE4655
USER=kyclark
```

If you provide a file name as an argument to `head`, it will work on that:

```
$ head /usr/share/dict/words
A
a
aa
aal
aalii
aam
Aani
```

```
aardvark
aardwolf
Aaron
```

Without any arguments, `head` assumes you must want it to read from STDIN. Many other programs will assume STDIN if not provided an argument. For instance, you could pipe `env` into `grep` to look for lines with the word "TERM" in them:

```
$ env | grep TERM
TERM_PROGRAM=iTerm.app
TERM=xterm-256color
TERM_PROGRAM_VERSION=3.2.6
TERM_SESSION_ID=w2t0p0:34E89330-9CEE-4FA6-9039-3674CBDE4655
ITERM_PROFILE=Default
ITERM_SESSION_ID=w2t0p0:34E89330-9CEE-4FA6-9039-3674CBDE4655
COLORTERM=truecolor
```

If you are fortunate enough to have the `fortune` and `cowsay` programs on your system, you can do this:

```
$ fortune | cowsay
 ________________________________________
/ Somebody ought to cross ball point pens \
| with coat hangers so that the pens will |
\ multiply instead of disappear.          /
 ----------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Manual pages

> "Programming isn't about what you know; it's about what you can figure out." - Chris Pine

The `man` program will show you the manual page for a program, if it exists. Just type `man <program>`, e.g., `man wget`. Inside a manpage, you can use the `/` to search for a string. Use `q` to "quit" `man`. Most programs will also show you a help/usage document if you run them with `-h`, `--help`, or `-help`. I often find it useful to `grep` the help, e.g.:

```
$ wget --help | grep clobber
  -nc, --no-clobber               skip downloads that would download to
       --unlink                   remove file before clobber
```

Pronunciation

- /: “slash”; the thing leaning the other way is a “backslash”
- **sh**: “shuh” or “ess-ach”
- **etc**: “et-see”
- **usr**: “user”
- **src**: “source”
- #: “hash” (NOT “hashtag”) or “pound”
- **$**: “dollar”
- **!**: “bang”
- **#!**: “shebang”
- ^: “caret”
- **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`.

When you are assigning a variable, you do not use the `$`.

```
[hpc:login3@~]$ SECRET=ilikecake
[hpc:login3@~]$ echo $SECRET
ilikecake
[hpc:login3@~]$ echo SECRET
SECRET
```

To remove a variable from your environment, use `unset`:

```
[hpc:login3@~]$ unset SECRET
[hpc:login3@~]$ echo $SECRET
```

Notice that there is no error when referencing a variable that does not exist or has not been set.

Control sequences

If you launch a program that won't stop, you can use CTRL-C (where "CTRL" is the "control" key sometime written "^C" or "^-C") to send an "interrupt" signal to the program. If it is well-behaved, it should stop, but it may not. For example, perhaps I've tried to use a text editor to open a 10G FASTA file and now my terminal is unresponsive because the editor is using all available memory. I could open another terminal on the machine and run `ps -fu $USER` to find all the programs I am running:

```
$ ps -fu $USER
UID        PID  PPID  C STIME TTY          TIME CMD
kyclark  31718 31692  0 12:16 ?        00:00:00 sshd: kyclark@pts/75
kyclark  31723 31718  0 12:16 pts/75   00:00:00 -bash
kyclark  33265 33247  0 12:16 ?        00:00:00 sshd: kyclark@pts/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 maize_genome.fasta
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 I want to kill `vim`, so I type `kill 33792`. If in a reasonable amount of time (a minute or so) that doesn't work, I could use `kill -9` (but it's considered a bit uncouth).

CTRL-Z is used 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, 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!
- **!!**: (bang-bang) execute the last command again
- **!$**: (bang-dollar) the last argument from your previous command line (think of the $ as the right anchor in a regex)
- **!^**: (band-caret) 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)

NB: 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 $PATH

Your `$PATH` setting is an ordered, colon-delimited list of directories that will be searched to find programs. Run `echo $PATH` to see yours. Here's my `$PATH` on the UA HPC:

```
[hpc:login3@~]$ echo $PATH | sed "s/:/\n/g"
```

```
/rsgrps/bh_class/bin
/home/u20/kyclark/.cargo/bin
/home/u20/kyclark/.local/bin
/cm/local/apps/gcc/6.1.0/bin
/cm/shared/uaapps/pbspro/18.2.1/sbin
/cm/shared/uaapps/pbspro/18.2.1/bin
/opt/TurboVNC/bin
/cm/shared/uabin
/usr/lib64/qt-3.3/bin
/cm/local/apps/environment-modules/4.0.0//bin
/usr/local/bin
/bin
/usr/bin
/usr/local/sbin
/usr/sbin
/sbin
/sbin
/usr/sbin
/cm/local/apps/environment-modules/4.0.0/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 "/rsgrps/bh_class/bin" 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:

```
export PATH="/rsgrps/bh_class/conda/bin:/rsgrps/bh_class/bin:$PATH"
```

You just told your shell (bash) to set the `$PATH` variable to have put the "/rsgrps/bh_class/conda/bin" and "/rsgrps/bh_class/bin" directories first and then whatever it was set to before. When you log out, however, this will be lost. Since we want this to happen each time we log in, so we can add this command to `$HOME/.bashrc`:

```
echo "export PATH=/rsgrps/bh_class/conda/bin:/rsgrps/bh_class/bin:$PATH" >> ~/.bashrc
```

As you find or create useful programs that you would like to have available globally on your system (i.e., not just in the current working directory), you can create a location like `$HOME/bin` (or my preferred `$HOME/.local/bin`) and add this to your `$PATH` as well. You can add as many directories as you like (within reason).

Dotfiles

"Dotfiles" 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. "Rc" may mean "resource configuration," but who really knows?

After a while, you may wish to collect your dotfiles into a Github repo, e.g., https://github.com/kyclark/dotfiles.

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. To use the binaries, you can:

1) Always refer to the full path to the binary
2) Place them into a directory in your `$PATH` like `$HOME/.local/bin`
3) Add the new directory to your `$PATH`

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 for installing into your `$HOME` 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` or some sort of "work" directory.

The (Data and Software) Carpentries

The Software Carpentry project aims to teach basic command-line usage. You should definitely look through `https://swcarpentry.github.io/shell-novice/`.

Chapter 3

Unix exercises

Here are some tasks that will introduce how the commands from the previous chapter can be combined to get things done. Sometimes you can get exactly what you need with command-line tools and never need to write a program!

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. Likely there are dozens of users, so we'll connect the output of `w` to `head` using a pipe `|` so that we only see the first five lines:

```
[hpc:login2@~]$ w | head -5
 09:39:27 up 65 days, 20:05, 10 users,  load average: 0.72, 0.75, 0.78
USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU WHAT
kyclark  pts/2    gatekeeper.hpc.a 09:38    0.00s  0.05s  0.02s w
emsenhub pts/0    gatekeeper.hpc.a 04:05   14.00s  0.87s  0.87s -bash
joneska  pts/3    gatekeeper.hpc.a 08:25    1:12m  0.16s  0.12s vim results_x2r
```

Really 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 can pipe `w` into `awk` and tell it we only want to see output when the Number of Records (`NR`) is greater than 2:

```
[hpc:login2@~]$ w | awk 'NR>2' | head -5
kyclark  pts/2    gatekeeper.hpc.a 09:38    0.00s  0.07s  0.03s w
emsenhub pts/0    gatekeeper.hpc.a 04:05   26.00s  0.87s  0.87s -bash
joneska  pts/3    gatekeeper.hpc.a 08:25    1:13m  0.16s  0.12s vim results_x2r
shawtaro pts/4    gatekeeper.hpc.a 08:06   58:34   0.17s  0.17s -bash
darrenc  pts/5    gatekeeper.hpc.a 07:58   51:07   0.14s  0.07s qsub -I -N pipe
```

`awk` takes a PREDICATE and a CODE BLOCK (contained within curly brackets `{}`). Without a PREDICATE, `awk` prints the whole line. I only want to see the first column, so I can tell `awk` to `print` just column `$1`:

```
[hpc:login2@~]$ w | awk 'NR>2 {print $1}' | head -5
kyclark
emsenhub
joneska
shawtaro
darrenc
```

We can see that the some users like "joneska" are logged in multiple times:

```
[hpc:login2@~]$ w | awk 'NR>2 {print $1}'
```

```
kyclark
emsenhub
joneska
shawtaro
darrenc
guven
guven
guven
joneska
dmarrone
```

Let's `uniq` that output:

```
[hpc:login2@~]$ w | awk 'NR>2 {print $1}' | uniq
kyclark
emsenhub
joneska
shawtaro
darrenc
guven
joneska
dmarrone
```

Hmm, that's not right – "joneska" is listed twice, and that is not unique. Remember that `uniq` only works *on sorted input*? So let's sort those names first:

```
[hpc:login2@~]$ w | awk 'NR>2 {print $1}' | sort | uniq
darrenc
dmarrone
emsenhub
guven
joneska
kyclark
shawtaro
```

To count how many unique users are logged in, we can use the `wc` (word count) program with the `-l` (lines) flag to count just the lines from the previous command

```
[hpc:login2@~]$ w | awk 'NR>2 {print $1}' | sort | uniq | wc -l
7
```

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."

NB: 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. Notice the use of `!$` (bang-dollar) to reference the last argument of the previous line so that I don't have to type it again (really useful if it's a long path):

```
$ 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
```

Do any of those words additionally contain the "ow" sequence?

```
$ grep 'oo' /usr/share/dict/words | grep 'ow' | head -5
arrowroot
arrowwood
balloonflower
bloodflower
blowproof
```

How many are there?

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

How many *do not* contain the "ow" sequence? Use `grep -v` to invert the match:

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

Do those numbers add up?

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

Find unclustered protein sequences

A labmate wants help finding the sequences of proteins that failed to cluster. Here is the setup:

```
$ wget ftp://ftp.imicrobe.us/biosys-analytics/exercises/unclustered-proteins.tgz
$ tar xvf unclustered-proteins.tgz
$ cd unclustered-proteins
```

The "README" contains our instructions:

```
The file "cdhit60.3+.clstr" contains all of the GI numbers for
proteins that were clustered and put into hmm profiles.  The file
"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 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
```

Where can we find those protein IDs in the "cdhit60.3+.clstr" 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) to say that we are looking at the beginning of a line `^` for a `>`:

```
$ 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%
```

The integer protein IDs we want are in the third column of this output when split on whitespace. The tool `awk` is perfect for this, and whitespace is the default split character (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...
```

The protein ID is still nestled there in the second field when splitting on the vertical bar (pipe). 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 need to capture these to a file which we can do with a redirect `>`. Since each protein might have been clustered more than once, so I should `sort | uniq` the list:

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

The "proteins.fa" is actually a little problematic. Some of the IDs have extra information. If you `grep '^>' proteins.fa`, you will see 220K IDs scroll by, not all of which are just integers. Let's isolate those that do not look like integers.

First we can remove the leading ">" from the FASTA header lines with this:

```
$ grep '^>' proteins.fa | sed "s/^>//"
```

If I can find a regular expression that matches what I want, then I can use `grep -v` to invert it to find the complement. `^\d+$` will do the trick. 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

This particular regex uses extensions introduced by the Perl programming language, so we need to use the `-P` flag. Add the `-v` to invert it:

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

Looking at the above output, we can see that it would be pretty easy to get rid of everything starting with the vertical bar, and `sed` is perfect for this. Note that we can tell `sed` to do more than one action by separating them with semicolons. Lastly, we need to ensure the IDs are sorted for the next step:

```
$ grep -e '^>' proteins.fa | sed "s/^>//; s/|.*//" | sort > protein-ids.o
```

To find the lines in "protein-ids.o" that are not in "clustered-ids.o", I can use the `comm` (common) command:

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

Did we get a reasonable answer?

```
$ wc -l clustered-ids.o unclustered-ids.o
  16257 clustered-ids.o
```

```
 204263 unclustered-ids.o
 220520 total
$ wc -l protein-ids.o
220520 protein-ids.o
```

Gapminder

For this exercise, look in the `biosys-analytics/data/gapminder` directory.

How many "txt" files are in the directory?

```
$ ls *.txt | wc -l
```

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

```
$ wc -l *.txt
```

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 *.cc.txt > 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? For this, we need to find all the rows where the second field is equal to "1952," and `awk` will let us do just that. Normally `awk` splits on whitespace, but we have tab-delimited so we need to use `-F"\t"`. Recipes in `awk` take the form of a CONDITIONAL and an ACTION. If the CONDITIONAL is missing, then the ACTION is applied to all lines. If the ACTION is missing, then the default is to print the entire line. Here we just provide the CONDITIONAL and then count the results:

```
$ awk -F"\t" '$2 == "1952"' all.txt | wc -l
```

How many observations for each year?

```
$ awk -F"\t" '{print $2}' all.txt | sort | uniq -c
```

How many observations are present for Africa (the fourth field is continent)?

```
$ awk -F"\t" '$4 == "Africa"' all.txt | wc -l
```

And what are the countries in Africa?

```
$ awk -F"\t" '$4 == "Africa" {print $1}' all.txt | sort | uniq
```

How many observations for for each continent?

```
$ awk -F"\t" 'NR>1 {print $4}' all.txt | sort | uniq -c
```

What was the world population in 1952? To answer this, we need to get the third column when the second column is "1952":

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

There's a problem because one of the numbers is in scientific notation:

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

Let's just remove that using `grep -v` (the `-v` reverses the match), then use the `paste` command 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 | grep -v [a-z] | paste -sd+ - | bc
2406957150
```

I know that's all a bit crude and absurd, but I thought you might be curious just how far you can take this.

How many observations where the life expectancy ("lifeExp," field #5) is greater than 40?

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

How many of those are from Africa?

```
$ awk -F"\t" '$5 > 40 && $4 == "Africa"' all.txt
```

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

```
$ awk -F"\t" '$5 > 70 { print $2 }' all.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 }' all.txt | sort | uniq -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
```

Chapter 4

Minimally competent bash scripting

> "We build our computer (systems) the way we build our cities: over time, without a plan, on top of ruins." - Ellen Ullman
>
> "The most effective debugging tool is still careful thought, coupled with judiciously placed print statements." - Brian W. Kernighan, Unix for Beginners (1979)

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:

```
$ python3 hello.py
Hello, World
```

So we just need to let the shell know that this is Python 3 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 python3
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 "python3" 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 "python3" that is found by the environment as I will usually put my preferred Python first in my `$PATH`.

Let's Make A Script!

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

```
$ cat -n hello2.sh
     1    #!/usr/bin/env 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
```

Sidebar: Catching Common Errors (set -u)

Bash is an 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.

Common Patterns

This is a cut-and-paste section for you. The idea is that I will describe many common patterns that you can use directly.

Test if a variable is a file or directory

Use the `-f` or `-d` functions to test if a variable identifies a "file" or a "directory," respectively

```
if [[ -f "$ARG" ]]; then
    echo "$ARG is a file"
fi

if [[ -d "$ARG" ]]; then
    echo "$ARG is a directory"
fi
```

Use `!` to negate this:

```
if [[ ! -f "$ARG" ]]; then
    echo "$ARG is NOT a file"
fi

if [[ ! -d "$ARG" ]]; then
    echo "$ARG is a directory"
fi
```

There are many other test you can use. See `man test` for a complete list. The `-s` is handy to see if a file is empty. You can use more than one test at a time with the `&&` ("and") or `||` ("or") operator.

```
if [[ -f "$ARG" ]] && [[ -s "$ARG" ]]; then
    echo "$ARG is a file and is not empty"
fi

if [[ ! -f "$ARG" ]] || [[ ! -s "$ARG" ]]; then
    echo "$ARG is a NOT file or is empty"
fi
```

Exit your script

The `exit` function will cease all operations and immediately exit. With no argument, it will use "0" which means "zero errors"; any other value is considered a error code, so `exit 1` is commonly used indicate some unspecified error.

```
if [[ -f "$ARG" ]]; then
    wc -l "$ARG"
    exit
else
    echo "$ARG must be a file"
    exit 1
fi
```

Check the number of arguments to your program

The first argument to your script is in `$1`, the second in `$2`, and so on. The number of arguments is in `$#`, so you can check the number like this:

```
if [[ $# -eq 0 ]]; then
    echo "Usage: foo.sh ARG"
    exit 1
fi
```

The `-eq` means "equal". You can also use `-gt` or `-gte` for "greater than (or equal)" and `-lt` or `-lte` for "less than or equal".

Put the arguments into named variables

You should assign `$1` and `$2` to names that have some meaning in your program.

```
INPUT_FILE=$1
NUM_ITERATIONS=$2
```

Set default values for optional arguments

If an argument is not needed, you can assign a default value. Here we can set `NUM_ITERATIONS` to have a default value of "10":

```
INPUT_FILE=$1
NUM_ITERATIONS=${2:-10}
```

Read a file

It's common to use a `while` loop to `read` a file, line-by-line, into some `VARIABLE`. Don't use a `$` on the `while` line (assigning), do use it when you want to interpolate it:

```
while read -r LINE; do
    echo "$LINE"
done < "$FILE"
```

Use a counter variable

It's common to use the variable `i` (for "integer" maybe?) as a temporary counter, e.g., iterating over lines in a file. The syntax to increment is clunky. This will print a line number and a line of text from a file:

```
i=0
while read -r LINE; do
    i=$((i+1))
    echo $i "$LINE"
done < "$FILE"
```

Loop operations

Use `continue` to skip to the next iteration of a loop. This will print only the even lines of a file:

```
i=0
while read -r LINE; do
    i=$((i+1))
    if [[ $(expr $i % 2) -eq 0 ]]; then
        continue
    else:
        echo "$i $LINE"
    fi
done < "$FILE"
```

Use `break` to leave a loop. This will print the first 10 lines of a file:

```
i=0
while read -r LINE; do
    echo "$LINE"
    i=$((i+1))
    if [[ $i -eq 10 ]]; then
        break
    fi
done < "$FILE"
```

Capture the output of a command

Historically `bash` used backticks (the same key as the tilde on a US QWERTY keyboard) to execute a command and put the results into a variable:

```
DIR=`ls`
```

Most people now use `$()` as it stands out much better:

```
DIR=$(ls)
LINES=$(grep foo bar.txt)
```

Count the number of lines in a file

```
NUM_LINES=$(wc -l "$FILE" | awk '{print $1}')

if [[ $NUM_LINES -lt 1 ]]; then
    echo "There is noting in $FILE"
    exit 1
fi
```

Get a temporary file or directory

Sometimes you need a temporary file to store something. If the name and location of the file is unimportant, use `mktemp` to get a temporary file or `mktemp -d` to get a temporary directory.

```
TMP_FILE=$(mktemp)
cat "foo\nbar\n" > "$TMP_FILE"

TMP_DIR=$(mktemp -d)
cd "$TMP_DIR"
```

Get the last part of a file or directory name

If you have "/path/to/my/file.txt" and you just want to print "file.txt", use `basename`:

```
FILE="/path/to/my/file.txt"
basename "$FILE"
```

Or put that into a variable name to use:

```
FILE="/path/to/my/file.txt"
BASENAME=$(basename "$FILE")
echo "Basename is $BASENAME"
```

Similary `dirname` is use to get "/path/to/my" from the above:

```
FILE="/path/to/my/file.txt"
DIRNAME=$(dirname "$FILE")
echo "Dirname is $DIRNAME"
```

Print with echo and printf

The `echo` command will print messages to the screen (standard out):

```
USER="Dave"
echo "I'm sorry, $USER, I can't do that."
```

The `printf` command is useful for formatting the output. The command expects a "template" first and then all the arguments for each formatting code in the template. The percent sign `%` is used in the template to indicate the type and options, e.g., an integer right-justified and three digits wide is `%3d`. Use `man printf` to learn more. Here is an example to print the line numbers in a file more prettier:

```
i=0
while read -r LINE; do
    i=$((i+1))
    printf "%3d: %s\n" $i $LINE
done < "$FILE"
```

Capture many items into a file for looping

Bash doesn't do lists (many items in a series) very well, so I usually put lists into files; e.g., I want to find how many files are in a directory and iterate over them:

```
FILES=$(mktemp)
find "$DIR" -type f -name \*.f[aq] > "$FILES"
NUM_FILES=$(wc -l "$FILES" | awk '{print $1}')

if [[ $NUM_FILES -lt 1 ]]; then
    echo "No usable files in $DIR"
    exit 1
fi

echo "Found $NUM_FILES in $DIR"

i=0
while read -r FILENAME; do
    i=$((i+1))
    BASENAME=$(basename "$FILENAME")
    printf "%3d: %s\n" $i "$FILENAME"
done < "$FILES"
```

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    #!/usr/bin/env 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    #!/usr/bin/env 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    #!/usr/bin/env 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    #!/usr/bin/env 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 files."
$ ./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.

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    #!/usr/bin/env 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    #!/usr/bin/env 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    #!/usr/bin/env 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
```

Our First Argument

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

```
$ cat -n hello3.sh
     1    #!/usr/bin/env 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    #!/usr/bin/env 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    #!/usr/bin/env 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.s
```

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    #!/usr/bin/env 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: $BASENAME"
    10
    11    # use results of function as argument to another function
    12    echo "3: BASENAME:" "$(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    #!/usr/bin/env 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    #!/usr/bin/env 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    #!/usr/bin/env 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    #!/usr/bin/env 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    #!/usr/bin/env 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    #!/usr/bin/env 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    #!/usr/bin/env 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    #!/usr/bin/env ash
     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 argument."
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    #!/usr/bin/env 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    #!/usr/bin/env 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"
```

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    #!/usr/bin/env 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 argument."
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!

Chapter 5

Intro to Python

> "Any fool can write code that a computer can understand. Good programmers write code that humans can understand." - Martin Fowler

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 (Read-Evaluate-Print-Loop, pronounced "reh-pull") to play:

```
$ ipython
Python 3.7.1 (default, Dec 14 2018, 19:28:38)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.2.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: name = 'Duderino'

In [2]: print('Hello,', name)
Hello, Duderino
```

Here I'm showing that it's easy to create a variable called `name` which we assign the value "Duderino." Unlike bash, we don't have to worry about spaces around the `=`. Just as in bash, we can use it in a `print` statement, but we can't directly stick it into the string:

```
In [3]: print('Hello, name')
Hello, name
```

Or we could to use the `+` operator to concatenate it to the literal string "Hello,":

```
In [4]: print('Hello, ' + name)
Hello, Duderino
```

Arguments

To say "hello" to an argument passed from the command line, we need the `sys` module. A module is 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 Professor
Hello, Professor!
```

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 Professor
Hello, Professor!
```

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 Professor
Hello, Professor!
```

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 call it at the end of the script. 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
     8  def main():
     9      """main"""
    10      args = sys.argv
    11
    12      if len(args) != 2:
    13          script = os.path.basename(args[0])
    14          print('Usage: {} NAME'.format(script))
    15          sys.exit(1)
    16
    17      name = args[1]
    18      print('Hello, {}!'.format(name))
    19
    20
    21  main()
```

Function Order

Note that you cannot put line 21 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 arguments:

```
$ cat -n hello_arg5.py
     1  #!/usr/bin/env python3
     2  """hello with to many"""
     3
     4  import sys
     5  import os
     6
     7
     8  def main():
     9      """main"""
    10      args = sys.argv
    11
    12      if len(args) < 2:
    13          script = os.path.basename(args[0])
```

```
    14          print('Usage: {} NAME [NAME2 ...]'.format(script))
    15          sys.exit(1)
    16
    17      names = args[1:]
    18      print('Hello, {}!'.format(', '.join(name)))
    19
    20
    21  main()
$ ./hello_arg5.py foo
Hello, foo!
$ ./hello_arg5.py foo bar baz
Hello, foo, bar, baz!
```

Look at line 18 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, type(n1), n2, type(n2)))
...
1 + 2 = 3
Cannot add 3 (<class 'int'>) and 4 (<class 'str'>)
```

Loops

As in bash, we can use `for` 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
     8  def main():
     9      """main"""
    10      args = sys.argv
    11
    12      if len(args) < 2:
    13          script = os.path.basename(args[0])
    14          print('Usage: {} NAME [NAME2 ...]'.format(script))
    15          sys.exit(1)
    16
    17      for name in args[1:]:
    18          print('Hello, ' + name + '!')
    19
    20
    21  main()
$ ./hello_arg6.py Salt Peppa
Hello, Salt!
Hello, Peppa!
```

You can use a `for` loop on anything that is like a list:

```
>>> 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.)

Stubbing new programs

Every program we've seen so far has had the same basic structure:

- Shebang
- Docstring
- imports
- def main()
- main()

```
#!/usr/bin/env python3
"""program docstring"""
```

```
import sys
import os


def main():
    """main"""
    ...

main()
```

Rather than type this out each time, let's use a program to help us start writing new programs. In `/rsgrps/bh_class/bin` (which should be in your `$PATH` by now), you will see `new_py.py`. (If you are working locally on your laptop – which I **strongly** recommend you learn how – you can find the program in `biosys-analytics/bin` which you can either copy into a directory in your `$PATH` or add that directory to your `$PATH`).

Try this:

```
$ new_py.py foo
Done, see new script "foo.py."
$ cat foo.py
#!/usr/bin/env python3
"""
Author : kyclark
Date   : 2019-01-24
Purpose: Rock the Casbah
"""

import os
import sys


# --------------------------------------------------
def main():
    args = sys.argv[1:]

    if len(args) != 1:
        print('Usage: {} ARG'.format(os.path.basename(sys.argv[0])))
        sys.exit(1)

    arg = args[0]

    print('Arg is "{}"'.format(arg))
```

```
# --------------------------------------------
main()
```

I will not require you to use this program to write new scripts, but I do suggest it could save you time and errors. I wrote this for myself, and I use it every time I start a new program. I first wrote a program like this in the mid-90s using Perl and have always relied on stubbers since.

Notice that the ".py" extension was added for you. You may specify `foo.py` if you prefer.

What happens if you try to initialize a script when one already exists with that name?

```
$ new_py.py foo
"foo.py" exists.  Overwrite? [yN] n
Will not overwrite. Bye!
```

Unless you answer "y", the script will not be overwritten. You could also use the `-f|--force` flag to force the overwritting of an existing file. Run with `-h|--help` to see all the options:

```
$ new_py.py -h
usage: new_py.py [-h] [-a] [-f] program

Create Python script

positional arguments:
  program         Program name

optional arguments:
  -h, --help      show this help message and exit
  -a, --argparse  Use argparse (default: False)
  -f, --force     Overwrite existing (default: False)
```

Hey, what is `--argparse` about? Let's try it! I will combine the two short flag `-a` and `-f` into `-fa` to "force" a new script that uses the "argparse" module to give us named options.

```
$ new_py.py -fa foo
Done, see new script "foo.py."
[hpc:login3@~]$ cat foo.py
#!/usr/bin/env python3
"""
Author : kyclark
Date   : 2019-01-24
Purpose: Rock the Casbah
"""

import argparse
```

```
import sys


# --------------------------------------------------
def get_args():
    """get command-line arguments"""
    parser = argparse.ArgumentParser(
        description='Argparse Python script',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument(
        'positional', metavar='str', help='A positional argument')

    parser.add_argument(
        '-a',
        '--arg',
        help='A named string argument',
        metavar='str',
        type=str,
        default='')

    parser.add_argument(
        '-i',
        '--int',
        help='A named integer argument',
        metavar='int',
        type=int,
        default=0)

    parser.add_argument(
        '-f', '--flag', help='A boolean flag', action='store_true')

    return parser.parse_args()


# --------------------------------------------------
def warn(msg):
    """Print a message to STDERR"""
    print(msg, file=sys.stderr)


# --------------------------------------------------
def die(msg='Something bad happened'):
    """warn() and exit with error"""
    warn(msg)
    sys.exit(1)
```

```
# --------------------------------------------------
def main():
    """Make a jazz noise here"""
    args = get_args()
    str_arg = args.arg
    int_arg = args.int
    flag_arg = args.flag
    pos_arg = args.positional

    print('str_arg = "{}"'.format(str_arg))
    print('int_arg = "{}"'.format(int_arg))
    print('flag_arg = "{}"'.format(flag_arg))
    print('positional = "{}"'.format(pos_arg))


# --------------------------------------------------
if __name__ == '__main__':
    main()
```

The advantage here is that we can now get quite detailed help documentation and very specific behavior from our arguments, e.g., one argument needs to be a string while another needs to be a number while another is a true/false, off/on flag:

```
$ ./foo.py -h
usage: foo.py [-h] [-a str] [-i int] [-f] str

Argparse Python script

positional arguments:
  str                A positional argument

optional arguments:
  -h, --help         show this help message and exit
  -a str, --arg str  A named string argument (default: )
  -i int, --int int  A named integer argument (default: 0)
  -f, --flag         A boolean flag (default: False)
```

All this without writing a line of Python! Quite useful.

Chapter 6

Python Strings, Lists, and Tuples

> "Good programming is good writing." - John Shore

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. We can ask for the first (or "zeroth") element from a string:

```
>>> name = 'Curly'
>>> name[0]
'C'
```

Or from a list:

```
>>> names = ['Larry', 'Moe', 'Curly', 'Shemp']
>>> names[0]
'Larry'
```

"Slice" operations let you take a range of items. Notice that we can operate on a string literal (in quotes):

```
>>> 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
```

Just as in bash, we can create a counter, increment it inside our loop, and print the element number before the element:

```
>>> names = ['Larry', 'Moe', 'Curly', 'Shemp']
>>> i = 0
>>> for name in names:
...   i += 1
...   print(i, name)
...
1 Larry
2 Moe
3 Curly
4 Shemp
```

Because we so often want this behavior, there is a function called `enumerate` that 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
```

You can use the `reversed` function on both strings and lists. Try it:

```
>>> reversed('cat')
<reversed object at 0x10bdc8358>
```

Probably you expected to see "tac"? Yeah, me, too. What is a "reversed object"? For now, think of it as a promise to give you the reversed string when you actually need it. We can force it into a list that we can look at by using the `list` function:

```
>>> list(reversed('cat'))
['t', 'a', 'c']
```

OK, closer, but I wanted to see "tac" and not a list containing those letters. We can put them back into a word by calling the `join` *function* of the *string element* that we want to put between the letter (which is an empty string):

```
>>> ''.join(list(reversed('cat')))
'tac'
```

Hmm, quite a bit of work to turn a word around. Still, could be useful, for example in finding CRISPR (clustered regularly interspaced short palindromic repeats) sequences? Here is a simple program to determine if a given string is a palindrome which is a string that is the same forwards and backwards.

```
$ 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 word.lower() == rev.lower() els
```

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, however, 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.

Lastly, let's use the `format` method to get rid of those pesky `str` calls:

```
$ 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_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('{} {} {} {}'.format(count_a, count_c, count_g, count_t))
$ ./dna3.py AACCTAG
3 2 1 1
```

If you're having trouble seeing the differences from `dna2.py` to `dna3.py`, try using `diff`:

```
$ diff dna2.py dna3.py
27c27
```

```
< print(' '.join([str(count_a), str(count_c), str(count_g), str(count_t)]))
---
> print('{} {} {} {}'.format(count_a, count_c, count_g, count_t))
```

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`

function on the list 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 quit] '.format('else ' if ite
    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(items.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 cullionly false filthsome
    10    toad-spotted caterwauling wall-eyed insatiate vile peevish infected
    11    sodden-witted lecherous ruinous indistinguishable dishonest thin-faced
    12    slanderous bankrupt base detestable rotten dishonest lubbery
    13    """.split()
    14
    15    NOUNS = """
    16    knave coward liar swine villain beggar slave scold jolthead whore barbermonger
    17    fishmonger carbuncle fiend traitor block ape braggart jack milksop boy harpy
    18    recreant degenerate Judas butt cur Satan ass coxcomb dandy 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.basename(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, coding, 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.basename(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_genes)]
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, coding, 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.basename(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, coding, 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.

Chapter 7

Python Dictionaries

Python has a data type called a "dictionary" that allows you to associate some "key" (often a string but it could be a number or even a tuple) to some "value" (which can be anything such as a string, number, tuple, list, set, or another dictionary). The same data structure in other languages 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', 'age': 4}
```

Or you can use the `dict` function and "keyword" arguments (which, in Pythonic style, do not use spaces around the `=` but the whitespace is not actually significant!):

```
>>> patch = dict(species='dog', age=4)
>>> patch
{'species': 'dog', 'age': 4}
```

You might be tempted to use the `{}` curly brackets to access the keys (e.g., if you were coming from Perl or you thought the language might be somehow internally consistent), but Python uses the `[]` square brackets to access dictionary fields just like arrays:

```
>>> patch['species']
'dog'
```

Since a dictionary key may be an integer, it can lead to dictionaries looking like arrays:

```
>>> patch[0] = 'food'
>>> patch[0]
'food'
```

Note that the data types of keys of the dictionary, like lists, may be heterogenous:

```
>>> patch
{'species': 'dog', 'age': 4, 0: 'food'}
>>> list(map(type, patch.keys()))
[<class 'str'>, <class 'str'>, <class 'int'>]
```

As may be the values:

```
>>> 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, 0: 'food', 'likes': ['walking', 'running', 'car trips', 'treats
<class 'list'>
>>> list(map(type, patch.values()))
[<class 'str'>, <class 'int'>, <class 'str'>, <class 'list'>]
```

You can directly use the dictionary values like the data types they are. Here we `join` the `list` that is in the `likes` slot:

```
>>> 'Patch is {} and likes {}'.format(patch['age'], ', '.join(patch['likes']))
'Patch is 4 and likes walking, running, car trips, treats, pets'
```

If you want to know if a key exists, `in` just as we did for list membership:

```
>>> 'likes' in patch
True
>>> 'dislikes' in patch
False
```

Just as you should not request a list position that does not exist in the list, you should not ask for a key that does not exist in a dictionary or you program will asplode at runtime:

```
>>> patch['dislikes']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'dislikes'
```

Better to check first:

```
>>> if 'dislikes' in patch:
...   print(patch['dislikes'])
... else:
...   print('Patch likes everything!')
...
Patch likes everything!
```

Or use the `get` method of the dictionary:

```
>>> patch.get('dislikes')
```

Wait, what did we get?

```
>>> type(patch.get('dislikes'))
<class 'NoneType'>
```

To find all the methods you can call on a dictionary, in the REPL type:

```
>>> help(dict)
```

Type `q` to "quit" the help. Use `/` to initiate a search, e.g., "/pop" to see how you can `pop` similar to the method in the `list` class.

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([
     7      'Stop!', 'Who would cross the Bridge of Death',
     8      'Must answer me these questions three,',
     9      '\'ere the other side he see.'
    10  ]))
    11
    12  for field in ['name', 'quest', 'favorite color']:
    13      person[field] = input('What is your {}? '.format(field))
    14      print(person)
    15
    16  if person['favorite color'].lower() == 'blue':
    17      print('Right, off you go.')
    18  else:
    19      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 Grail'}
What is your favorite color? Blue
{'name': 'Sir Lancelot of Camelot', 'quest': 'To seek the Holy Grail', 'favorite color': 'Bl
```

```
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 Grail'}
What is your favorite color? Blue. No yello--
{'name': 'Sir Galahad of Camelot', 'quest': 'I seek the Holy Grail', 'favorite color': 'Blue
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(sys.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.
```

```
    F is for Fanny sucked dry by a leech.
    G is for George smothered under a rug.
    H is for Hector done in by a thug.
    I is for Ida who drowned in a lake.
    J is for James who took lye by mistake.
    K is for Kate who was struck with an axe.
    L is for Leo who choked on some tacks.
    M is for Maud who was swept out to sea.
    N is for Neville who died of ennui.
    O is for Olive run through with an awl.
    P is for Prue trampled flat in a brawl.
    Q is for Quentin who sank on a mire.
    R is for Rhoda consumed by a fire.
    S is for Susan who perished of fits.
    T is for Titus who flew into bits.
    U is for Una who slipped down a drain.
    V is for Victor squashed under a train.
    W is for Winnie embedded in ice.
    X is for Xerxes devoured by mice.
    Y is for Yorick whose head was bashed in.
    Z is for Zillah who drank too much gin.
    """

    lookup = {}
    for line in text.splitlines():
        if line:
            lookup[line[0]] = line

    if letter in lookup:
        print(lookup[letter])
    else:
        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))
$ cat dna.txt
AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAAAAGAGTGTCTGATAGCAGC
$ ./dna3.py `cat dna.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:

```
>>> patch['dislikes']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'dislikes'
```

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(args[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](https://d

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  """
     3  Author : Ken Youens-Clark <kyclark@email.arizona.edu>
     4  Date   : 2019-02-06
     5  Purpose: Character Counter
     6  """
     7
     8  import argparse
     9  import os
    10  import sys
    11  from collections import Counter
    12
    13
    14  # --------------------------------------------------
    15  def get_args():
    16      """get command-line arguments"""
    17      parser = argparse.ArgumentParser(
    18          description='Character counter',
    19          formatter_class=argparse.ArgumentDefaultsHelpFormatter)
    20
    21      parser.add_argument('input', help='Filename or string to count', type=str)
    22
    23      parser.add_argument(
    24          '-c',
    25          '--charsort',
    26          help='Sort by character',
    27          dest='charsort',
    28          action='store_true')
    29
```

```
30      parser.add_argument(
31          '-n',
32          '--numsort',
33          help='Sort by number',
34          dest='numsort',
35          action='store_true')
36
37      parser.add_argument(
38          '-r',
39          '--reverse',
40          help='Sort in reverse order',
41          dest='reverse',
42          action='store_true')
43
44      return parser.parse_args()
45
46
47  # --------------------------------------------------
48  def warn(msg):
49      """Print a message to STDERR"""
50      print(msg, file=sys.stderr)
51
52
53  # --------------------------------------------------
54  def die(msg='Something bad happened'):
55      """warn() and exit with error"""
56      warn(msg)
57      sys.exit(1)
58
59
60  # --------------------------------------------------
61  def main():
62      """Make a jazz noise here"""
63      args = get_args()
64      input_arg = args.input
65      charsort = args.charsort
66      numsort = args.numsort
67      revsort = args.reverse
68
69      if charsort and numsort:
70          die('Please choose one of --charsort or --numsort')
71
72      if not charsort and not numsort:
73          charsort = True
74
75      text = ''
```

```
    if os.path.isfile(input_arg):
        text = ''.join(open(input_arg).read().splitlines())
    else:
        text = input_arg

    count = Counter(text.lower())

    if charsort:
        letters = sorted(count.keys())
        if revsort:
            letters.reverse()

        for letter in letters:
            print('{} {:5}'.format(letter, count[letter]))
    else:
        pairs = sorted([(x[1], x[0]) for x in count.items()])
        if revsort:
            pairs.reverse()

        for n, char in pairs:
            print('{} {:5}'.format(char, n))


# ------------------------------------------------
if __name__ == '__main__':
    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 bacronym.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  # --------------------------------------------------
    13  def get_args():
    14      """get arguments"""
    15      parser = argparse.ArgumentParser(
    16          description='Explain acronyms',
    17          formatter_class=argparse.ArgumentDefaultsHelpFormatter)
    18
    19      parser.add_argument('acronym', help='Acronym', type=str, metavar='STR')
    20
    21      parser.add_argument(
    22          '-n',
    23          '--num',
    24          help='Maximum number of definitions',
    25          type=int,
    26          metavar='NUM',
    27          default=5)
    28      parser.add_argument(
    29          '-w',
    30          '--wordlist',
    31          help='Dictionary/word file',
```

```
32          type=str,
33          metavar='STR',
34          default='/usr/share/dict/words')
35      parser.add_argument(
36          '-x',
37          '--exclude',
38          help='List of words to exclude',
39          type=str,
40          metavar='STR',
41          default='a,an,the')
42      return parser.parse_args()
43
44
45  # --------------------------------------------------
46  def main():
47      """main"""
48      args = get_args()
49      acronym = args.acronym
50      wordlist = args.wordlist
51      limit = args.num
52      goodword = r'^[a-z]{2,}$'
53      badwords = set(re.split(r'\s*,\s*', args.exclude.lower()))
54
55      if not re.match(goodword, acronym.lower()):
56          print('"{}" must be >1 in length, only use letters'.format(acronym))
57          sys.exit(1)
58
59      if not os.path.isfile(wordlist):
60          print('"{}" is not a file.'.format(wordlist))
61          sys.exit(1)
62
63      seen = set()
64      words_by_letter = defaultdict(list)
65      for word in open(wordlist).read().lower().split():
66          clean = re.sub('[^a-z]', '', word)
67          if not clean:  # nothing left?
68              continue
69
70          if re.match(goodword,
71                      clean) and clean not in seen and clean not in badwords:
72              seen.add(clean)
73              words_by_letter[clean[0]].append(clean)
74
75      len_acronym = len(acronym)
76      definitions = []
77      for i in range(0, limit):
```

```
78          definition = []
79          for letter in acronym.lower():
80              possible = words_by_letter.get(letter, [])
81              if len(possible) > 0:
82                  definition.append(
83                      random.choice(possible).title() if possible else '?')
84
85          if len(definition) == len_acronym:
86              definitions.append(' '.join(definition))
87
88      if len(definitions) > 0:
89          print(acronym.upper() + ' =')
90          for definition in definitions:
91              print(' - ' + definition)
92      else:
93          print('Sorry I could not find any good definitions')
94
95
96  # --------------------------------------------------
97  if __name__ == '__main__':
98      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 "{}" and "{}."'
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.lower().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.lower().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(words2.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"
  # 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 litre of
beer, the colour and flavour of which put us into such a good humour
that we forgot our labours.  We set about to analyse our behaviour,
organise our thoughts, recognise our faults, catalogue our merits, and
generally have a dialogue without pretence as a licence to improve
ourselves.
$ cat american.txt
I went to the theater last night with my neighbor and had a liter of
beer, the color and flavor of which put us into such a good humor that
we forgot our labors.  We set about to analyze our behavior, organize
our thoughts, recognize our faults, catalog our merits, and generally
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 "american.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 "american.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.

Chapter 8

Common Patterns in Python

Get positional command-line arguments

You can get the command-line arguments using `sys.argv` (argument vector), but it's annoying that the name of the Python program itself is in the first position (`sys.argv[0]`). To skip over this, take a slice of the argument vector starting at the second position (index `1`) which will succeed even if there are no arguments – you'll get an empty list, which is safe.

```
$ cat -n args.py
     1  #!/usr/bin/env python3
     2
     3  import os
     4  import sys
     5
     6  args = sys.argv[1:]
     7  num = len(args)
     8
     9  print('There are {} arg{}'.format(num, '' if num == 1 else 's'))
$ ./args.py
There are 0 args
$ ./args.py foo
There are 1 arg
$ ./args.py foo bar
There are 2 args
```

Put positional arguments into named variables

If you use `sys.argv[1]` and `sys.argv[2]` throughout your program, it degrades readability. It's better to copy the values into variables that have meaningful names like "file" or "num_lines".

```
$ cat -n name_args.py
     1  #!/usr/bin/env python3
     2
     3  import os
     4  import sys
     5
     6  args = sys.argv[1:]
     7
     8  if len(args) != 2:
```

```
     9      print('Usage: {} FILE NUM'.format(os.path.basename(sys.argv[0])))
    10      sys.exit(1)
    11
    12  file, num = args
    13
    14  file = args[0]
    15  num = args[1]
    16
    17  print('FILE is "{}", NUM is "{}"'.format(file, num))
$ ./name_args.py
Usage: name_args.py FILE NUM
$ ./name_args.py nobody.txt 10
FILE is "nobody.txt", NUM is "10"
```

Set defaults for optional arguments

```
$ cat -n default_arg.py
     1  #!/usr/bin/env python3
     2
     3  import os
     4  import sys
     5
     6  args = sys.argv[1:]
     7  num_args = len(args)
     8
     9  if not 1 <= num_args <= 2:
    10      print('Usage: {} FILE [NUM]'.format(os.path.basename(sys.argv[0])))
    11      sys.exit(1)
    12
    13  file = args[0]
    14  num = args[1] if num_args == 2 else 10
    15
    16  print('FILE is "{}", NUM is "{}"'.format(file, num))
$ ./default_arg.py
Usage: default_arg.py FILE [NUM]
$ ./default_arg.py nobody.txt
FILE is "nobody.txt", NUM is "10"
$ ./default_arg.py nobody.txt 5
FILE is "nobody.txt", NUM is "5"
```

Test argument is file and read

This program takes an argument, tests that it is a file, and then reads it. It's basically `cat`.

```
$ cat -n read_file.py
     1  #!/usr/bin/env python3
     2  """Read a file argument"""
     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  filename = args[0]
    14
    15  if not os.path.isfile(filename):
    16      print('"{}" is not a file'.format(filename), file=sys.stderr)
    17      sys.exit(1)
    18
    19  for line in open(filename):
    20      print(line, end='')
$ ./read_file.py foo
"foo" is not a file
$ ./read_file.py nobody.txt
I'm Nobody! Who are you?
Are you - Nobody - too?
Then there's a pair of us!
Don't tell! they'd advertise - you know!

How dreary - to be - Somebody!
How public - like a Frog -
To tell one's name - the livelong June -
To an admiring Bog!

Emily Dickinson
```

Write data to a file

To write a file, you need to `open` some filename with a second argument of the "mode" where

- `r`: read (default)
- `w`: write
- `t`: text mode (default)
- `b`: binary

You can combine the flags so that `wt` means "write a text file" which is what is done here.

If you `open` a file for writing and the file already exists, it will be overwritten, so it may behoove you to check if the file exists first!

```
$ cat -n write_file.py
     1  #!/usr/bin/env python3
     2
     3  import os
     4  import sys
     5
     6
     7  args = sys.argv[1:]
     8
     9  if len(args) < 1:
    10      print('Usage: {} ARG1 [ARG2...]'.format(os.path.basename(sys.argv[0])))
    11      sys.exit(1)
    12
    13  outfile = 'out.txt'
    14  out_fh = open(outfile, 'wt')
    15
    16  for arg in args:
    17      out_fh.write(arg + '\n')
    18
    19  out_fh.close()
    20  print('Done, see "{}"'.format(outfile))
$ ./write_file.py foo bar baz
Done, see "out.txt"
$ cat out.txt
foo
bar
baz
```

Test if an argument is a directory and list the contents

```
$ cat -n list_dir.py
     1  #!/usr/bin/env python3
     2  """Show contents of directory argument"""
     3
     4  import os
     5  import sys
     6
     7  args = sys.argv[1:]
     8
     9  if len(args) != 1:
```

```
    10      print('Usage: {} DIR'.format(os.path.basename(sys.argv[0])))
    11      sys.exit(1)
    12
    13  dirname = args[0]
    14
    15  if not os.path.isdir(dirname):
    16      print('"{}" is not a directory'.format(dirname), file=sys.stderr)
    17      sys.exit(1)
    18
    19  for entry in os.listdir(dirname):
    20      print(entry)
$ ./list_dir.py
Usage: list_dir.py DIR
$ ./list_dir.py .
list_dir.py
kmers.py
skip_loop.py
unpack_dict2.py
nobody.txt
name_args.py
create_dir.py
sort_dict_by_values.py
foo
args.py
sort_dict_by_keys.py
read_file.py
sort_dict_by_keys2.py
unpack_dict.py
codons.py
default_arg.py
```

Skip an iteration of a loop

Sometimes in a loop (`for` or `while`) you want to skip immediately to the top of the loop. You can use `continue` to do this. In this example, we skip the even-numbered lines by using the modulus `%` operator to find those line numbers which have a remainder of 0 after dividing by 2. We can use the `enumerate` function to provide both the array index and value of any list.

```
$ cat -n skip_loop.py
     1  #!/usr/bin/env python3
     2
     3  import os
     4  import sys
     5
```

```
     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), file=sys.stderr)
    17      sys.exit(1)
    18
    19  for i, line in enumerate(open(file)):
    20      if (i + 1) % 2 == 0:
    21          continue
    22
    23      print(i + 1, line, end='')
$ ./skip_loop.py
Usage: skip_loop.py FILE
$ ./skip_loop.py nobody.txt
1 I'm Nobody! Who are you?
3 Then there's a pair of us!
5
7 How public - like a Frog -
9 To an admiring Bog!
11 Emily Dickinson
```

Create a directory if it does not exist

This program takes a directory name and looks to see if it already exists or needs to be created.

```
$ cat -n create_dir.py
     1  #!/usr/bin/env python3
     2  """Test for a directory and create if needed"""
     3
     4  import os
     5  import sys
     6
     7  args = sys.argv[1:]
     8
     9  if len(args) != 1:
    10      print('Usage: {} DIR'.format(os.path.basename(sys.argv[0])))
    11      sys.exit(1)
```

```
    12
    13  dirname = args[0]
    14
    15  if os.path.isdir(dirname):
    16      print('"{}" exists'.format(dirname))
    17  else:
    18      print('Creating "{}"'.format(dirname))
    19      os.makedirs(dirname)
$ ./create_dir.py
Usage: create_dir.py DIR
$ ./create_dir.py foo
Creating "foo"
$ ./create_dir.py foo
"foo" exists
```

Unpack a dictionary's key/values pairs

The `.items()` method on a dictionary will return a list of tuples:

```
$ cat -n unpack_dict.py
     1  #!/usr/bin/env python3
     2  """Unpack dict"""
     3
     4  import os
     5  import sys
     6
     7  albums = {
     8      "2112": 1976,
     9      "A Farewell To Kings": 1977,
    10      "All the World's a Stage": 1976,
    11      "Caress of Steel": 1975,
    12      "Exit, Stage Left": 1981,
    13      "Fly By Night": 1975,
    14      "Grace Under Pressure": 1984,
    15      "Hemispheres": 1978,
    16      "Hold Your Fire": 1987,
    17      "Moving Pictures": 1981,
    18      "Permanent Waves": 1980,
    19      "Power Windows": 1985,
    20      "Signals": 1982,
    21  }
    22
    23  for tup in albums.items():
    24      album = tup[0]
```

```
    25      year = tup[1]
    26      print('{:4} {}'.format(year, album))
$ ./unpack_dict.py
1976 2112
1977 A Farewell To Kings
1976 All the World's a Stage
1975 Caress of Steel
1981 Exit, Stage Left
1975 Fly By Night
1984 Grace Under Pressure
1978 Hemispheres
1987 Hold Your Fire
1981 Moving Pictures
1980 Permanent Waves
1985 Power Windows
1982 Signals
```

But the `for` loop could unpack the tuple directly. Compare line 23 in the above and below programs.

```
$ cat -n unpack_dict2.py
     1  #!/usr/bin/env python3
     2  """Unpack dict"""
     3
     4  import os
     5  import sys
     6
     7  albums = {
     8      "2112": 1976,
     9      "A Farewell To Kings": 1977,
    10      "All the World's a Stage": 1976,
    11      "Caress of Steel": 1975,
    12      "Exit, Stage Left": 1981,
    13      "Fly By Night": 1975,
    14      "Grace Under Pressure": 1984,
    15      "Hemispheres": 1978,
    16      "Hold Your Fire": 1987,
    17      "Moving Pictures": 1981,
    18      "Permanent Waves": 1980,
    19      "Power Windows": 1985,
    20      "Signals": 1982,
    21  }
    22
    23  for album, year in albums.items():
    24      print('{:4} {}'.format(year, album))
$ ./unpack_dict2.py
1976 2112
```

```
1977 A Farewell To Kings
1976 All the World's a Stage
1975 Caress of Steel
1981 Exit, Stage Left
1975 Fly By Night
1984 Grace Under Pressure
1978 Hemispheres
1987 Hold Your Fire
1981 Moving Pictures
1980 Permanent Waves
1985 Power Windows
1982 Signals
```

Sort a dictionary by keys

To sort a dictionary by the keys, you have to understand that the `.sort()` method of an list mutates the list *in-place*. We get the keys of a dictionary with the `.keys()` method which does not support the `.sort()` method:

```
>>> d = dict(foo=1, bar=2)
>>> d.keys()
dict_keys(['foo', 'bar'])
>>> d.keys().sort()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'dict_keys' object has no attribute 'sort'
```

We could copy the keys into a list to sort like so:

```
>>> k = list(d.keys())
>>> k
['foo', 'bar']
>>> k.sort()
>>> k
['bar', 'foo']
```

Or we can use the `sorted()` function that accepts a list and *returns a sorted list*:

```
>>> d.keys()
dict_keys(['foo', 'bar'])
>>> sorted(d.keys())
['bar', 'foo']
```

Either way, once we have the sorted keys, we can get the associated values:

```
$ cat -n sort_dict_by_keys.py
     1  #!/usr/bin/env python3
```

```
     2
     3  import os
     4  import sys
     5
     6  albums = {
     7      "2112": 1976,
     8      "A Farewell To Kings": 1977,
     9      "All the World's a Stage": 1976,
    10      "Caress of Steel": 1975,
    11      "Exit, Stage Left": 1981,
    12      "Fly By Night": 1975,
    13      "Grace Under Pressure": 1984,
    14      "Hemispheres": 1978,
    15      "Hold Your Fire": 1987,
    16      "Moving Pictures": 1981,
    17      "Permanent Waves": 1980,
    18      "Power Windows": 1985,
    19      "Signals": 1982,
    20  }
    21
    22  for album in sorted(albums.keys()):
    23      print('{:25} {}'.format(album, albums[album]))
$ ./sort_dict_by_keys.py
2112                      1976
A Farewell To Kings       1977
All the World's a Stage   1976
Caress of Steel           1975
Exit, Stage Left          1981
Fly By Night              1975
Grace Under Pressure      1984
Hemispheres               1978
Hold Your Fire            1987
Moving Pictures           1981
Permanent Waves           1980
Power Windows             1985
Signals                   1982
```

Or we could unpack the tuples directly like above:

```
$ cat -n sort_dict_by_keys2.py
     1  #!/usr/bin/env python3
     2
     3  import os
     4  import sys
     5
     6  albums = {
     7      "2112": 1976,
```

```
     8      "A Farewell To Kings": 1977,
     9      "All the World's a Stage": 1976,
    10      "Caress of Steel": 1975,
    11      "Exit, Stage Left": 1981,
    12      "Fly By Night": 1975,
    13      "Grace Under Pressure": 1984,
    14      "Hemispheres": 1978,
    15      "Hold Your Fire": 1987,
    16      "Moving Pictures": 1981,
    17      "Permanent Waves": 1980,
    18      "Power Windows": 1985,
    19      "Signals": 1982,
    20  }
    21
    22  for album, year in sorted(albums.items()):
    23      print('{:25} {}'.format(album, year))
$ ./sort_dict_by_keys2.py
2112                      1976
A Farewell To Kings       1977
All the World's a Stage   1976
Caress of Steel           1975
Exit, Stage Left          1981
Fly By Night              1975
Grace Under Pressure      1984
Hemispheres               1978
Hold Your Fire            1987
Moving Pictures           1981
Permanent Waves           1980
Power Windows             1985
Signals                   1982
```

Sort a dictionary by values

To sort a dictionary by the values rather than the keys, we need to reverse the tuples which is what happens on line 24. Notice that in years when two albums were released, the `sorted` first sorts by the first tuple member (the year) and then the second (album name):

```
$ cat -n sort_dict_by_values.py
     1  #!/usr/bin/env python3
     2
     3  import os
     4  import sys
     5
     6  albums = {
```

```
     7      "2112": 1976,
     8      "A Farewell To Kings": 1977,
     9      "All the World's a Stage": 1976,
    10      "Caress of Steel": 1975,
    11      "Exit, Stage Left": 1981,
    12      "Fly By Night": 1975,
    13      "Grace Under Pressure": 1984,
    14      "Hemispheres": 1978,
    15      "Hold Your Fire": 1987,
    16      "Moving Pictures": 1981,
    17      "Permanent Waves": 1980,
    18      "Power Windows": 1985,
    19      "Signals": 1982,
    20  }
    21
    22  # Create a list of (value, key) tuples
    23  # sorted in descending order by the values
    24  pairs = sorted([(x[1], x[0]) for x in albums.items()])
    25
    26  for year, album in pairs:
    27      print('{} {}'.format(year, album))
$ ./sort_dict_by_values.py
1975 Caress of Steel
1975 Fly By Night
1976 2112
1976 All the World's a Stage
1977 A Farewell To Kings
1978 Hemispheres
1980 Permanent Waves
1981 Exit, Stage Left
1981 Moving Pictures
1982 Signals
1984 Grace Under Pressure
1985 Power Windows
1987 Hold Your Fire
```

Extract codons from DNA

This example assumes a codon length (`k`) of 3 and uses a handy third argument to `range` that indicates the distance to skip in each iteration. The goal is to start at position 0, then jump to position 3, then 6, etc., to extract all the codons. Imagine how you could expand this to get all the codons in all the frames (this one starts at "1" which is really "0" in the string):

```
$ cat -n codons.py
     1  #!/usr/bin/env python3
     2  """Extract codons from DNA"""
     3
     4  import os
     5  import sys
     6
     7  args = sys.argv[1:]
     8  num_args = len(args)
     9
    10  if not 1 <= num_args <= 2:
    11      print('Usage: {} DNA'.format(os.path.basename(sys.argv[0])))
    12      sys.exit(1)
    13
    14  string = args[0]
    15  k = 3
    16  n = len(string) - k + 1
    17
    18  for i in range(0, n, k):
    19      print(string[i:i+k])
$ ./codons.py
Usage: codons.py DNA
$ ./codons.py AAACCCGGGTTT
AAA
CCC
GGG
TTT
```

Extract k-mers from a string

K-mers are k-length contiguous sub-sequences from a string. They are similar to codons (which are 3-mers), but we tend to move across the string by one character than than the codon length (3). Notice this script guards against a 2nd argument that should be a number but is not:

```
$ cat -n kmers.py
     1  #!/usr/bin/env python3
     2  """Extract k-mers from string"""
     3
     4  import os
     5  import sys
     6
     7  args = sys.argv[1:]
     8  num_args = len(args)
```

```
 9
10  if not 1 <= num_args <= 2:
11      print('Usage: {} STR [K]'.format(os.path.basename(sys.argv[0])))
12      sys.exit(1)
13
14  string = args[0]
15  k = args[1] if num_args == 2 else '3'
16
17  # Guard against a string like "foo"
18  if not k.isdigit():
19      print('k "{}" is not a digit'.format(k))
20      sys.exit(1)
21
22  # Safe to convert now
23  k = int(k)
24
25  if len(string) < k:
26      print('There are no {}-length substrings in "{}"'.format(k, string))
27  else:
28      n = len(string) - k + 1
29      for i in range(0, n):
30          print(string[i:i+k])
```

```
$ ./kmers.py
Usage: kmers.py STR [K]
$ ./kmers.py foobar 10
There are no 10-length substrings in "foobar"
$ ./kmers.py AAACCCGGGTTT 3
AAA
AAC
ACC
CCC
CCG
CGG
GGG
GGT
GTT
TTT
```

Chapter 9

Parsing with Python

We'll use the term "parsing" to mean deriving meaning from structured text. For example, we can use `argparse` to find meaning from command-line arguments that may or may not have flags or be defined by positions. In this chapter, we'll look at common file file formats in bioinformatics like CSV, FASTA/Q, and GFF.

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:

```
 1  #!/usr/bin/env python3
 2  """
 3  Author : kyclark
 4  Date   : 2019-02-19
 5  Purpose: Rock the Casbah
 6  """
 7
 8  import argparse
 9  import sys
10
11
12  # --------------------------------------------------
13  def get_args():
14      """get command-line arguments"""
15      parser = argparse.ArgumentParser(
16          description='Argparse Python script',
17          formatter_class=argparse.ArgumentDefaultsHelpFormatter)
18
19      parser.add_argument(
20          'positional', metavar='str', help='A positional argument')
21
22      parser.add_argument(
```

```
23          '-a',
24          '--arg',
25          help='A named string argument',
26          metavar='str',
27          type=str,
28          default='')
29
30      parser.add_argument(
31          '-i',
32          '--int',
33          help='A named integer argument',
34          metavar='int',
35          type=int,
36          default=0)
37
38      parser.add_argument(
39          '-f', '--flag', help='A boolean flag', action='store_true')
40
41      return parser.parse_args()
42
43
44  # --------------------------------------------------
45  def warn(msg):
46      """Print a message to STDERR"""
47      print(msg, file=sys.stderr)
48
49
50  # --------------------------------------------------
51  def die(msg='Something bad happened'):
52      """warn() and exit with error"""
53      warn(msg)
54      sys.exit(1)
55
56
57  # --------------------------------------------------
58  def main():
59      """Make a jazz noise here"""
60      args = get_args()
61      str_arg = args.arg
62      int_arg = args.int
63      flag_arg = args.flag
64      pos_arg = args.positional
65
66      print('str_arg = "{}"'.format(str_arg))
67      print('int_arg = "{}"'.format(int_arg))
68      print('flag_arg = "{}"'.format(flag_arg))
```

```
69      print('positional = "{}"'.format(pos_arg))
70
71
72  # --------------------------------------------------
73  if __name__ == '__main__':
74      main()
```

If you run without any arguments or with `-h|--help`, you get a usage statement:

```
$ ./test.py
usage: test.py [-h] [-a str] [-i int] [-f] str
test.py: error: the following arguments are required: str
[cholla@~/work/biosys-analytics/lectures/09-python-parsing]$ ./test.py -h
usage: test.py [-h] [-a str] [-i int] [-f] str

Argparse Python script

positional arguments:
  str                A positional argument

optional arguments:
  -h, --help         show this help message and exit
  -a str, --arg str  A named string argument (default: )
  -i int, --int int  A named integer argument (default: 0)
  -f, --flag         A boolean flag (default: False)
```

And the `argparse` module is able to turn the command line arguments into useful information:

```
$ ./test.py -a foo -i 42 -f ABCDE
str_arg = "foo"
int_arg = "42"
flag_arg = "True"
positional = "ABCDE"
```

If you try to write the code to parse `-a foo -i 42 -f ABCDE`, you will quickly appreciate how much effort using this module will save you!

CSV Files

"CSV" stands for "comma-separated values" and describes structured text that looks like:

```
foo,bar,baz
flip,burp,quux
```

More generally, these are values that are separated by some marker. Commas are typical but can cause problems when a comma can be a legitimate value,

e.g., in addresses or formatted numbers, so tabs are often used as delimiters. Tab-delimited files may have the extension ".tsv," ".dat," ".tab", or ".txt." Usually CSV files have ".csv" and are especially common in the R/Pandas world.

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 to parse the output from Centrifuge (http://www.ccb.jhu.edu/software/centrifuge/). Despite the name, this module parses any line-oriented, delimited text, not just CSV files.

For this, we'll use some data from a study from Yellowstone National Park (https://www.imicrobe.us/#/samples/1378). For each input file, Centrifuge creates two tab-delimited output files:

1. 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
```

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.basename(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(ext))
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.centrifuge.sum
count   taxID
6432    321327
80  321332
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.basename(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(ext))
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_file))
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.py

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 like this:

```
$ head oceanic_mesopelagic_zone_biome.csv
Analysis,Pipeline version,Sample,MGnify ID,Experiment type,Assembly,ENA run,ENA WGS sequence
MGYA00005220,2.0,ERS490373,MGYS00000410,metagenomic,,ERR599044,
MGYA00005081,2.0,ERS490507,MGYS00000410,metagenomic,,ERR599005,
MGYA00005208,2.0,ERS492680,MGYS00000410,metagenomic,,ERR598999,
MGYA00005133,2.0,ERS490633,MGYS00000410,metagenomic,,ERR599154,
MGYA00005272,2.0,ERS488769,MGYS00000410,metagenomic,,ERR599062,
MGYA00005209,2.0,ERS490714,MGYS00000410,metagenomic,,ERR599124,
MGYA00005243,2.0,ERS493822,MGYS00000410,metagenomic,,ERR599051,
MGYA00005117,2.0,ERS491980,MGYS00000410,metagenomic,,ERR599132,
MGYA00005135,2.0,ERS493705,MGYS00000410,metagenomic,,ERR599152,
```

I'd rather see it formatted vertically:

```
$ tabchk.py oceanic_mesopelagic_zone_biome.csv
// ****** Record 1 ****** //
Analysis             : MGYA00005220
Pipeline version     : 2.0
Sample               : ERS490373
MGnify ID            : MGYS00000410
Experiment type      : metagenomic
Assembly             :
ENA run              : ERR599044
ENA WGS sequence set :
```

Sometimes I have many more fields and lots of missing values, so I can use the `-d` flag to the program indicates to show a "dense" matrix, i.e., leave out the empty fields:

```
$ tabchk.py -d oceanic_mesopelagic_zone_biome.csv
// ****** Record 1 ****** //
Analysis         : MGYA00005220
Pipeline version : 2.0
Sample           : ERS490373
MGnify ID        : MGYS00000410
Experiment type  : metagenomic
```

```
ENA run          : ERR599044
```

Here is the `tabchk.py` program I wrote to do that. The program is generally useful, so I added it to the main `bin` directory of the repo so that you can use that if you have already added it to your `$PATH`.

```
#!/usr/bin/env python3
"""
Author:  Ken Youens-Clark <kyclark@email.arizona.edu>
Purpose: Check the first/few records of a delimited text file
"""

import argparse
import csv
import os
import re
import sys


# --------------------------------------------------
def get_args():
    """Get command-line arguments"""
    parser = argparse.ArgumentParser(
        description='Check a delimited text file',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('file', metavar='FILE', help='Input file')

    parser.add_argument(
        '-s',
        '--sep',
        help='Field separator',
        metavar='str',
        type=str,
        default='')

    parser.add_argument(
        '-f',
        '--field_names',
        help='Field names (no header)',
        metavar='str',
        type=str,
        default='')

    parser.add_argument(
        '-l',
        '--limit',
```

```
42          help='How many records to show',
43          metavar='int',
44          type=int,
45          default=1)
46
47      parser.add_argument(
48          '-d',
49          '--dense',
50          help='Not sparse (skip empty fields)',
51          action='store_true')
52
53      parser.add_argument(
54          '-n',
55          '--number',
56          help='Show field number (e.g., for awk)',
57          action='store_true')
58
59      parser.add_argument(
60          '-N',
61          '--no_headers',
62          help='No headers in first row',
63          action='store_true')
64
65      return parser.parse_args()
66
67
68  # --------------------------------------------------
69  def main():
70      """main"""
71      args = get_args()
72      file = args.file
73      limit = args.limit
74      sep = args.sep
75      dense = args.dense
76      show_numbers = args.number
77      no_headers = args.no_headers
78
79      if not os.path.isfile(file):
80          print('"{}" is not a file'.format(file))
81          sys.exit(1)
82
83      if not sep:
84          _, ext = os.path.splitext(file)
85          if ext == '.csv':
86              sep = ','
87          else:
```

```
 88              sep = '\t'
 89
 90      with open(file) as csvfile:
 91          dict_args = {'delimiter': sep}
 92
 93          if args.field_names:
 94              regex = re.compile(r'\s*,\s*')
 95              names = regex.split(args.field_names)
 96              if names:
 97                  dict_args['fieldnames'] = names
 98
 99          if args.no_headers:
100              num_flds = len(csvfile.readline().split(sep))
101              dict_args['fieldnames'] = list(
102                  map(lambda i: 'Field' + str(i), range(1, num_flds + 1)))
103              csvfile.seek(0)
104
105          reader = csv.DictReader(csvfile, **dict_args)
106
107          for i, row in enumerate(reader, start=1):
108              vals = dict(
109                  [x for x in row.items() if x[1] != '']) if dense else row
110              flds = vals.keys()
111              longest = max(map(len, flds))
112              fmt = '{:' + str(longest + 1) + '}: {}'
113              print('// ****** Record {} ****** //'.format(i))
114              n = 0
115              for key, val in vals.items():
116                  n += 1
117                  show = fmt.format(key, val)
118                  if show_numbers:
119                      print('{:3} {}'.format(n, show))
120                  else:
121                      print(show)
122
123              if i == limit:
124                  break
125
126
127  # -------------------------------------------------
128  if __name__ == '__main__':
129      main()
```

BLAST's tab-delimited output (`-outfmt 6`) does not include headers, so I have this alias:

```
alias blast6chk='tabchk.py -f "qseqid,sseqid,pident,length,mismatch,gapopen,qstart,qend,ssta
```

tabget.py

Here's a program that extracts columns from a delimited text file using the column names instead of the number (yes, I know we could just use `awk`):

```
$ cat -n tabget.py
     1  #!/usr/bin/env python3
     2  """
     3  Author : Ken Youens-Clark <kyclark@gmail.com>
     4  Date   : 2018-11-16
     5  Purpose: Get fields from a tab/csv file
     6  """
     7
     8  import argparse
     9  import csv
    10  import os
    11  import re
    12  import sys
    13
    14
    15  # --------------------------------------------------
    16  def get_args():
    17      """get command-line arguments"""
    18      parser = argparse.ArgumentParser(
    19          description='Argparse Python script',
    20          formatter_class=argparse.ArgumentDefaultsHelpFormatter)
    21
    22      parser.add_argument(
    23          'file', nargs='+', metavar='FILE', help='Input file(s)')
    24
    25      parser.add_argument(
    26          '-d',
    27          '--delimiter',
    28          help='Field delimiter',
    29          metavar='str',
    30          type=str,
    31          default='')
    32
    33      parser.add_argument(
    34          '-f',
    35          '--field',
    36          help='Name of field(s)',
    37          metavar='str',
    38          type=str,
    39          default='')
    40
```

```
41      return parser.parse_args()
42
43
44  # --------------------------------------------------
45  def warn(msg):
46      """Print a message to STDERR"""
47      print(msg, file=sys.stderr)
48
49
50  # --------------------------------------------------
51  def die(msg='Something bad happened'):
52      """warn() and exit with error"""
53      warn(msg)
54      sys.exit(1)
55
56
57  # --------------------------------------------------
58  def main():
59      """Make a jazz noise here"""
60      args = get_args()
61      files = args.file
62      default_delim = args.delimiter
63      field_names = re.split('\s*,\s*', args.field)
64
65      for file in files:
66          with open(file, 'rt') as fh:
67              delim = default_delim
68              if not delim:
69                  _, ext = os.path.splitext(file)
70                  if ext == '.csv':
71                      delim = ','
72                  else:
73                      delim = '\t'
74
75              reader = csv.DictReader(fh, delimiter=delim)
76
77              print(delim.join(field_names))
78
79              for row in reader:
80                  flds = list(map(lambda f: row[f], field_names))
81                  print(delim.join(flds))
82
83
84  # --------------------------------------------------
85  if __name__ == '__main__':
86      main()
```

tab2json.py

At some point I must have needed to turn a flat, delimited text file into a hierarchical, JSON structured, but I cannot at this moment remember why. Anyway, here's a program that will do that.

```
$ cat -n tab2json.py
     1  #!/usr/bin/env python3
     2  """
     3  Author:  Ken Youens-Clark <kyclark@gmail.com>
     4  Purpose: Convert a delimited text file to JSON
     5  """
     6
     7  import argparse
     8  import csv
     9  import json
    10  import os
    11  import re
    12  import sys
    13
    14
    15  # --------------------------------------------------
    16  def get_args():
    17      """get args"""
    18      parser = argparse.ArgumentParser(
    19          description='Argparse Python script',
    20          formatter_class=argparse.ArgumentDefaultsHelpFormatter)
    21
    22      parser.add_argument(
    23          'tabfile', metavar='str', nargs='+', help='A positional argument')
    24
    25      parser.add_argument(
    26          '-s',
    27          '--sep',
    28          help='Field separator',
    29          metavar='str',
    30          type=str,
    31          default='\t')
    32
    33      parser.add_argument(
    34          '-o',
    35          '--outdir',
    36          help='Output dir',
    37          metavar='str',
    38          type=str,
    39          default='')
```

```
40
41      parser.add_argument(
42          '-i',
43          '--indent',
44          help='Indent level',
45          metavar='int',
46          type=int,
47          default=2)
48
49      parser.add_argument(
50          '-n',
51          '--normalize_headers',
52          help='Normalize headers',
53          action='store_true')
54
55      return parser.parse_args()
56
57
58  # --------------------------------------------------
59  def main():
60      """main"""
61      args = get_args()
62      indent_level = args.indent
63      out_dir = args.outdir
64      fs = args.sep
65      norm_hdr = args.normalize_headers
66      tabfiles = args.tabfile
67
68      if len(tabfiles) < 1:
69          print('No input files')
70          sys.exit(1)
71
72      if indent_level < 0:
73          indent_level = 0
74
75      if out_dir and not os.path.isdir(out_dir):
76          os.makedirs(out_dir)
77
78      for i, tabfile in enumerate(tabfiles, start=1):
79          basename = os.path.basename(tabfile)
80          filename, _ = os.path.splitext(basename)
81          dirname = os.path.dirname(os.path.abspath(tabfile))
82          print('{:3}: {}'.format(i, basename))
83          write_dir = out_dir if out_dir else dirname
84          out_path = os.path.join(write_dir, filename + '.json')
85          out_fh = open(out_path, 'wt')
```

```
 86
 87          with open(tabfile) as fh:
 88              reader = csv.DictReader(fh, delimiter=fs)
 89              if norm_hdr:
 90                  reader.fieldnames = list(map(normalize, reader.fieldnames))
 91              out_fh.write(json.dumps(list(reader), indent=indent_level))
 92
 93
 94  # --------------------------------------------------
 95  def normalize(hdr):
 96      return re.sub(r'[^A-Za-z0-9_]', '', hdr.lower().replace(' ', '_'))
 97
 98
 99  # --------------------------------------------------
100  if __name__ == '__main__':
101      main()
```

FASTA

Now let's finally get into parsing good, old FASTA files. We're going to need to install the BioPython (http://biopython.org/) module to get a FASTA parser. 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/#/samples/578). You can download the full file with this command:

```
$ iget /iplant/home/shared/imicrobe/projects/26/samples/578/CAM_SMPL_GS108.fa
```

Since that file is 725M, I've added a sample to the repo in the `examples` directory.

```
$ head -5 CAM_SMPL_GS108.fa
>CAM_READ_0231669761 /library_id="CAM_LIB_GOS108XLRVAL-4F-1-400" /sample_id="CAM_SMPL_GS108'
ATTTACAATAATTTAATAAAATTAACTAGAAATAAAATATTGTATGAAAATATGTTAAAT
AATGAAAGTTTTTCAGATCGTTTAATAATATTTTTCTTCCATTTTGCTTTTTTCTAAAAT
TGTTCAAAAACAAACTTCAAAGGAAAATCTTCAAAATTTACATGATTTTATATTTAAACA
AATAGAGTTAAGTATAAGAGAAATTGGATATGGTGATGCTTCAATAAATAAAAAAATGAA
```

The format of a FASTA file is:

- A record starts with a header row which has `>` as the first character on a line
- The string following the `>` up until the first whitespace is the record ID
- Anything following the ID up to the newline can be the "description," but here we see this space has been set up as key/value pairs of metadata

- Any line after a header that does not start with `>` is the sequence. The sequence may be one long line or many shorter lines.

We **could** write our own FASTA parser, and we would definitely learn much along the way, but let's not and instead use the BioPython `SeqIO` (sequence input-output) module to read and write all the different formats. FASTA is one of the most common, but other formats may include FASTQ (FASTA but with "Quality" scores for the base calls), GenBank, EMBL, and more. See https://biopython.org/wiki/SeqIO for an exhaustive list.

There is a useful program called `seqmagick` that will give you information like the following:

```
$ seqmagick info *.fa
name              alignment    min_len   max_len   avg_len  num_seqs
CAM_SMPL_GS108.fa FALSE             47       594    369.65       499
CAM_SMPL_GS112.fa FALSE             50       624    383.50       500
```

You can install it like so:

```
$ python -m pip install seqmagick
```

Let's write a toy program to mimic part of the output. 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         47        594 369.45            500
CAM_SMPL_GS112.fa         50        624 383.50            500
```

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  """
     3  Author:  Ken Youens-Clark <kyclark@email.arizona.edu>
     4  Purpose: Mimic seqmagick, print stats on FASTA sequences
     5  """
     6
     7  import os
     8  import sys
     9  import numpy as np
    10  from Bio import SeqIO
    11
    12  files = sys.argv[1:]
    13
```

```
14  if not files:
15      print('Usage: {} F1.fa [F2.fa...]'.format(os.path.basename(sys.argv[0])))
16      sys.exit(1)
17
18  info = []
19  for file in files:
20      lengths = []
21      for record in SeqIO.parse(file, 'fasta'):
22          lengths.append(len(record.seq))
23
24      info.append({
25          'name': os.path.basename(file),
26          'min_len': min(lengths),
27          'max_len': max(lengths),
28          'avg_len': '{:.2f}'.format(np.mean(lengths)),
29          'num_seqs': len(lengths)
30      })
31
32  if info:
33      longest_file_name = max([len(f['name']) for f in info])
34      fmt = '{:' + str(longest_file_name) + '} {:10} {:10} {:10} {:10}'
35      flds = ['name', 'min_len', 'max_len', 'avg_len', 'num_seqs']
36      print(fmt.format(*flds))
37      for rec in info:
38          print(fmt.format(*[rec[fld] for fld in flds]))
39  else:
40      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 subset_fastx.py
     1  #!/usr/bin/env python3
     2  """
     3  Author:  Ken Youens-Clark <kyclark@email.arizona.edu>
     4  Purpose: Subset FASTA/Q files
     5  """
     6
     7  import argparse
     8  import os
```

```
import sys
from Bio import SeqIO


# --------------------------------------------------
def get_args():
    """get args"""
    parser = argparse.ArgumentParser(
        description='Split FASTA files',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('file', help='Input file', metavar='FILE')

    parser.add_argument(
        '-f',
        '--infmt',
        help='Input file format',
        type=str,
        metavar='FMT',
        choices=['fasta', 'fastq'],
        default='fasta')

    parser.add_argument(
        '-F',
        '--outfmt',
        help='Output file format',
        type=str,
        metavar='FMT',
        default=None)

    parser.add_argument(
        '-n',
        '--num',
        help='Number of records per file',
        type=int,
        metavar='NUM',
        default=500000)

    parser.add_argument(
        '-o',
        '--outdir',
        help='Output directory',
        type=str,
        metavar='DIR',
        default='subset')

```

```
55      return parser.parse_args()
56
57  # --------------------------------------------------
58  def warn(msg):
59      """Print a message to STDERR"""
60      print(msg, file=sys.stderr)
61
62
63  # --------------------------------------------------
64  def die(msg='Something bad happened'):
65      """warn() and exit with error"""
66      warn(msg)
67      sys.exit(1)
68
69
70  # --------------------------------------------------
71  def main():
72      """main"""
73      args = get_args()
74      in_file = args.file
75      in_fmt = args.infmt
76      out_fmt = args.outfmt if args.outfmt else args.infmt
77      out_dir = args.outdir
78      num_seqs = args.num
79
80      if not os.path.isfile(in_file):
81          die('--file "{}" is not a file'.format(in_file))
82
83      if os.path.dirname(os.path.abspath(in_file)) == os.path.abspath(out_dir):
84          die('--outdir "{}" cannot be the same as input files'.format(out_dir))
85
86      if num_seqs < 1:
87          die("--num cannot be less than one")
88
89      if not os.path.isdir(out_dir):
90          os.mkdir(out_dir)
91
92      basename = os.path.basename(in_file)
93      out_file = os.path.join(out_dir, basename)
94      out_fh = open(out_file, 'wt')
95      num_written = 0
96
97      for record in SeqIO.parse(in_file, in_fmt):
98          SeqIO.write(record, out_fh, out_fmt)
99          num_written += 1
100
```

```
101             if num_written == num_seqs:
102                 break
103
104         print('Done, wrote {} sequence{} to "{}"'.format(
105             num_written, '' if num_written == 1 else 's', out_file))
106
107
108     # --------------------------------------------------
109     if __name__ == '__main__':
110         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 fa_split.py
     1  #!/usr/bin/env python3
     2  """
     3  Author:  Ken Youens-Clark
     4  Purpose: Split FASTA files
     5  NB:      If you have FASTQ files, maybe just use "split"?
     6  """
     7
     8  import argparse
     9  import os
    10  import sys
    11  from Bio import SeqIO
    12
    13
    14  # --------------------------------------------------
    15  def get_args():
    16      """get args"""
    17      parser = argparse.ArgumentParser(
    18          description='Split FASTA/Q files',
    19          formatter_class=argparse.ArgumentDefaultsHelpFormatter)
    20
    21      parser.add_argument('file', help='FASTA input file(s)', nargs='+')
    22
    23      parser.add_argument(
    24          '-f',
    25          '--input_format',
```

```
26          help='Input file format',
27          type=str,
28          metavar='FORMAT',
29          choices=['fasta', 'fastq'],
30          default='fasta')
31
32      parser.add_argument(
33          '-F',
34          '--output_format',
35          help='Output file format',
36          type=str,
37          metavar='FORMAT',
38          choices=['fasta', 'fastq'],
39          default='fasta')
40
41      parser.add_argument(
42          '-n',
43          '--sequences_per_file',
44          help='Number of sequences per file',
45          type=int,
46          metavar='NUM',
47          default=50)
48
49      parser.add_argument(
50          '-o',
51          '--out_dir',
52          help='Output directory',
53          type=str,
54          metavar='DIR',
55          default='fasplit')
56
57      return parser.parse_args()
58
59
60  # --------------------------------------------------
61  def warn(msg):
62      """Print a message to STDERR"""
63      print(msg, file=sys.stderr)
64
65
66  # --------------------------------------------------
67  def die(msg='Something bad happened'):
68      """warn() and exit with error"""
69      warn(msg)
70      sys.exit(1)
71
```

```
72
73  # --------------------------------------------------
74  def main():
75      """main"""
76      args = get_args()
77      files = args.file
78      input_format = args.input_format
79      output_format = args.output_format
80      out_dir = args.out_dir
81      seqs_per_file = args.sequences_per_file
82
83      if not os.path.isdir(out_dir):
84          os.mkdir(out_dir)
85
86      if seqs_per_file < 1:
87          die('--sequences_per_file "{}" cannot be less than one'.format(
88              seqs_per_file))
89
90      num_files = 0
91      num_seqs_written = 0
92      for i, file in enumerate(files, start=1):
93          print('{:3d}: {}'.format(i, os.path.basename(file)))
94          num_files += 1
95          num_seqs_written += process(
96              file=file,
97              input_format=input_format,
98              output_format=output_format,
99              out_dir=out_dir,
100             seqs_per_file=seqs_per_file)
101
102     print('Done, processed {} sequence{} from {} file{} into "{}"'.format(
103         num_seqs_written, '' if num_seqs_written == 1 else 's', num_files, ''
104         if num_files == 1 else 's', out_dir))
105
106
107 # --------------------------------------------------
108 def process(file, input_format, output_format, out_dir, seqs_per_file):
109     """
110     Spilt file into smaller files into out_dir
111     Optionally convert to output format
112     Return number of sequences written
113     """
114     if not os.path.isfile(file):
115         warn('"{}" is not valid'.format(file))
116         return 0
117
```

```
118      basename, ext = os.path.splitext(os.path.basename(file))
119      out_fh = None
120      i = 0
121      num_written = 0
122      nfile = 0
123      for record in SeqIO.parse(file, input_format):
124          if i == seqs_per_file:
125              i = 0
126              if out_fh is not None:
127                  out_fh.close()
128                  out_fh = None
129
130          i += 1
131          num_written += 1
132          if out_fh is None:
133              nfile += 1
134              path = os.path.join(out_dir,
135                                  basename + '.' + '{:04d}'.format(nfile) + ext)
136              out_fh = open(path, 'wt')
137
138          SeqIO.write(record, out_fh, output_format)
139
140      return num_written
141
142
143  # --------------------------------------------------
144  if __name__ == '__main__':
145      main()
```

You can run this on the FASTA files in the `examples` directory to split them into files of 50 sequences each:

```
$ ./fa_split.py *.fa
  1: CAM_SMPL_GS108.fa
  2: CAM_SMPL_GS112.fa
Done, processed 1000 sequences from 2 files into "fasplit"
$ ls -lh fasplit/
total 1088
-rw-r--r--  1 kyclark  staff    22K Feb 19 15:41 CAM_SMPL_GS108.0001.fa
-rw-r--r--  1 kyclark  staff    28K Feb 19 15:41 CAM_SMPL_GS108.0002.fa
-rw-r--r--  1 kyclark  staff    27K Feb 19 15:41 CAM_SMPL_GS108.0003.fa
-rw-r--r--  1 kyclark  staff    23K Feb 19 15:41 CAM_SMPL_GS108.0004.fa
-rw-r--r--  1 kyclark  staff    22K Feb 19 15:41 CAM_SMPL_GS108.0005.fa
-rw-r--r--  1 kyclark  staff    26K Feb 19 15:41 CAM_SMPL_GS108.0006.fa
-rw-r--r--  1 kyclark  staff    29K Feb 19 15:41 CAM_SMPL_GS108.0007.fa
-rw-r--r--  1 kyclark  staff    27K Feb 19 15:41 CAM_SMPL_GS108.0008.fa
-rw-r--r--  1 kyclark  staff    26K Feb 19 15:41 CAM_SMPL_GS108.0009.fa
```

```
-rw-r--r--  1 kyclark  staff    24K Feb 19 15:41 CAM_SMPL_GS108.0010.fa
-rw-r--r--  1 kyclark  staff    26K Feb 19 15:41 CAM_SMPL_GS112.0001.fa
-rw-r--r--  1 kyclark  staff    27K Feb 19 15:41 CAM_SMPL_GS112.0002.fa
-rw-r--r--  1 kyclark  staff    28K Feb 19 15:41 CAM_SMPL_GS112.0003.fa
-rw-r--r--  1 kyclark  staff    27K Feb 19 15:41 CAM_SMPL_GS112.0004.fa
-rw-r--r--  1 kyclark  staff    27K Feb 19 15:41 CAM_SMPL_GS112.0005.fa
-rw-r--r--  1 kyclark  staff    27K Feb 19 15:41 CAM_SMPL_GS112.0006.fa
-rw-r--r--  1 kyclark  staff    28K Feb 19 15:41 CAM_SMPL_GS112.0007.fa
-rw-r--r--  1 kyclark  staff    29K Feb 19 15:41 CAM_SMPL_GS112.0008.fa
-rw-r--r--  1 kyclark  staff    27K Feb 19 15:41 CAM_SMPL_GS112.0009.fa
-rw-r--r--  1 kyclark  staff    16K Feb 19 15:41 CAM_SMPL_GS112.0010.fa
```

We can verify that things worked:

```
$ for file in fasplit/*; do echo -n $file && grep '^>' $file | wc -l; done
fasplit/CAM_SMPL_GS108.0001.fa      50
fasplit/CAM_SMPL_GS108.0002.fa      50
fasplit/CAM_SMPL_GS108.0003.fa      50
fasplit/CAM_SMPL_GS108.0004.fa      50
fasplit/CAM_SMPL_GS108.0005.fa      50
fasplit/CAM_SMPL_GS108.0006.fa      50
fasplit/CAM_SMPL_GS108.0007.fa      50
fasplit/CAM_SMPL_GS108.0008.fa      50
fasplit/CAM_SMPL_GS108.0009.fa      50
fasplit/CAM_SMPL_GS108.0010.fa      50
fasplit/CAM_SMPL_GS112.0001.fa      50
fasplit/CAM_SMPL_GS112.0002.fa      50
fasplit/CAM_SMPL_GS112.0003.fa      50
fasplit/CAM_SMPL_GS112.0004.fa      50
fasplit/CAM_SMPL_GS112.0005.fa      50
fasplit/CAM_SMPL_GS112.0006.fa      50
fasplit/CAM_SMPL_GS112.0007.fa      50
fasplit/CAM_SMPL_GS112.0008.fa      50
fasplit/CAM_SMPL_GS112.0009.fa      50
fasplit/CAM_SMPL_GS112.0010.fa      50
```

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  Author:  Ken Youens-Clark <kyclark@email.arizona.edu>
     4  Purpose: Parse the GFF output of Prodigal
     5  """
     6
     7  import argparse
     8  import os
     9  import sys
    10
    11
    12  # --------------------------------------------------
    13  def get_args():
    14      """get args"""
    15      parser = argparse.ArgumentParser(
    16          description='Prodigal GFF parser',
    17          formatter_class=argparse.ArgumentDefaultsHelpFormatter)
    18
    19      parser.add_argument('gff', metavar='FILE', help='Prodigal GFF file')
    20
    21      parser.add_argument(
    22          '-m',
    23          '--min',
    24          help='Min score',
    25          metavar='float',
    26          type=float,
    27          default=0)
    28
    29      return parser.parse_args()
    30
    31
    32  # --------------------------------------------------
    33  def warn(msg):
    34      """Print a message to STDERR"""
    35      print(msg, file=sys.stderr)
    36
    37
    38  # --------------------------------------------------
    39  def die(msg='Something bad happened'):
    40      """warn() and exit with error"""
    41      warn(msg)
    42      sys.exit(1)
    43
    44
    45  # --------------------------------------------------
```

```
46  def main():
47      """main"""
48      args = get_args()
49      gff_file = args.gff
50      min_score = args.min
51
52      if not os.path.isfile(gff_file):
53          die('GFF "{}" is not a file'.format(gff_file))
54
55      flds = [
56          'seqname', 'source', 'feature', 'start', 'end', 'score', 'strand',
57          'frame', 'attribute'
58      ]
59
60      for line in open(gff_file):
61          if line.startswith('#'):
62              continue
63
64          vals = line.rstrip().split('\t')
65          rec = dict(zip(flds, vals))
66          attrs = {}
67
68          for x in rec['attribute'].split(';'):
69              if '=' in x:
70                  key, value = x.split('=')
71                  attrs[key] = value
72
73          score = attrs.get('score')
74          if score is not None and float(score) >= min_score:
75              print('{} {}'.format(rec['seqname'], score))
76
77
78  # --------------------------------------------------
79  if __name__ == '__main__':
80      main()
```

XML

Maybe something with parsing NCBI taxonomy?

JSON

There's lots of JSON in the world that needs to be parsed.

Chapter 10

Writing Simple Games in Python

Games are a terrific way to learn. If you take something simple you know well, you have all the information you need to complete it. Something simple like tic-tac-toe – you know you need a board, some way for the user to select a cell, you need to keep track of who's playing (X or O), when they've made a bad move, and when someone has won. Games often need random values, interact with the user, employ infinite loops – in short, they are fascinating and fun to program and play.

Guessing Game

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='Argparse Python script')
11        parser.add_argument('positional', metavar='str', help='A positional argument')
12        parser.add_argument('-a', '--arg', help='A named string argument',
13                            metavar='str', type=str, default='')
14        parser.add_argument('-i', '--int', help='A named integer argument',
15                            metavar='int', type=int, default=0)
16        parser.add_argument('-f', '--flag', help='A boolean 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 guessing 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 guesses',
                        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 "{}."'.format(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  """
     3  Author:  Ken Youens-Clark <kyclark@gmail.com>
     4  Purpose: Guess-the-number game
     5  """
     6
     7  import argparse
     8  import random
     9  import sys
    10
    11
    12  # --------------------------------------------------
    13  def get_args():
    14      """get args"""
    15      parser = argparse.ArgumentParser(
```

```
16          description='Guessing game',
17          formatter_class=argparse.ArgumentDefaultsHelpFormatter)
18
19      parser.add_argument(
20          '-m',
21          '--min',
22          help='Minimum value',
23          metavar='int',
24          type=int,
25          default=1)
26
27      parser.add_argument(
28          '-x',
29          '--max',
30          help='Maximum value',
31          metavar='int',
32          type=int,
33          default=50)
34
35      parser.add_argument(
36          '-g',
37          '--guesses',
38          help='Number of guesses',
39          metavar='int',
40          type=int,
41          default=5)
42
43      return parser.parse_args()
44
45
46  # --------------------------------------------------
47  def main():
48      """main"""
49      args = get_args()
50      low = args.min
51      high = args.max
52      guesses_allowed = args.guesses
53      secret = random.randint(low, high)
54
55      if low < 1:
56          print('--min cannot be lower than 1')
57          sys.exit(1)
58
59      if guesses_allowed < 1:
60          print('--guesses cannot be lower than 1')
61          sys.exit(1)
```

```
62
63      if low > high:
64          print('--min "{}" is higher than --max "{}"'.format(low, high))
65          sys.exit(1)
66
67      prompt = 'Guess a number between {} and {} (q to quit): '.format(low, high)
68      num_guesses = 0
69
70      while True:
71          guess = input(prompt)
72          num_guesses += 1
73
74          if guess == 'q':
75              print('Now you will never know the answer.')
76              sys.exit(0)
77
78          if not guess.isdigit():
79              print('"{}" is not a number'.format(guess))
80              continue
81
82          print('You guessed "{}"'.format(guess))
83          num = int(guess)
84
85          if num_guesses >= guesses_allowed:
86              print('Too many guesses! The number was "{}."'.format(secret))
87              sys.exit()
88          elif not low < num < high:
89              print('Number is not in the allowed range')
90          elif num == secret:
91              print('You win!')
92              break
93          elif num < secret:
94              print('Too low.')
95          else:
96              print('Too high.')
97
98
99  # --------------------------------------------------
100 if __name__ == '__main__':
101     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  """
     3  Author:  Ken Youens-Clark <kyclark@gmail.com>
     4  Purpose: Hangman game
     5  """
     6
     7  import argparse
     8  import os
     9  import random
    10  import re
    11  import sys
    12
    13
    14  # --------------------------------------------------
    15  def get_args():
    16      """parse arguments"""
    17      parser = argparse.ArgumentParser(
    18          description='Hangman',
    19          formatter_class=argparse.ArgumentDefaultsHelpFormatter)
    20
    21      parser.add_argument(
    22          '-l', '--maxlen', help='Max word length', type=int, default=10)
    23
    24      parser.add_argument(
    25          '-n', '--minlen', help='Min word length', type=int, default=5)
    26
    27      parser.add_argument(
    28          '-m', '--misses', help='Max number of misses', type=int, default=10)
    29
    30      parser.add_argument(
    31          '-w',
```

```
32          '--wordlist',
33          help='Word list',
34          type=str,
35          default='/usr/share/dict/words')
36
37      return parser.parse_args()
38
39
40  # --------------------------------------------------
41  def main():
42      """main"""
43      args = get_args()
44      max_len = args.maxlen
45      min_len = args.minlen
46      max_misses = args.misses
47      wordlist = args.wordlist
48
49      if not os.path.isfile(wordlist):
50          print('--wordlist "{}" is not a file.'.format(wordlist))
51          sys.exit(1)
52
53      if min_len < 1:
54          print('--minlen must be positive')
55          sys.exit(1)
56
57      if not 3 <= max_len <= 20:
58          print('--maxlen should be between 3 and 20')
59          sys.exit(1)
60
61      if min_len > max_len:
62          print('--minlen ({}) is greater than --maxlen ({})'.format(
63              min_len, max_len))
64          sys.exit(1)
65
66      regex = re.compile('^[a-z]{' + str(min_len) + ',' + str(max_len) + '}$')
67      words = [w for w in open(wordlist).read().split() if regex.match(w)]
68      word = random.choice(words)
69      play({'word': word, 'max_misses': max_misses})
70
71
72  # --------------------------------------------------
73  def play(state):
74      """Loop to play the game"""
75      word = state.get('word') or ''
76
77      if not word:
```

```
78          print('No word!')
79          sys.exit(1)
80
81      guessed = state.get('guessed') or list('_' * len(word))
82      prev_guesses = state.get('prev_guesses') or set()
83      num_misses = state.get('num_misses') or 0
84      max_misses = state.get('max_misses') or 0
85
86      if ''.join(guessed) == word:
87          msg = 'You win. You guessed "{}" with "{}" miss{}!'
88          print(msg.format(word, num_misses, '' if num_misses == 1 else 'es'))
89          sys.exit(0)
90
91      if num_misses >= max_misses:
92          print('You lose, loser!  The word was "{}."'.format(word))
93          sys.exit(0)
94
95      print('{} (Misses: {})'.format(' '.join(guessed), num_misses))
96      new_guess = input('Your guess? ("?" for hint, "!" to quit) ').lower()
97
98      if new_guess == '!':
99          print('Better luck next time, loser.')
100         sys.exit(0)
101     elif new_guess == '?':
102         new_guess = random.choice([x for x in word if x not in guessed])
103         num_misses += 1
104
105     if not re.match('^[a-zA-Z]$', new_guess):
106         print('"{}" is not a letter'.format(new_guess))
107         num_misses += 1
108     elif new_guess in prev_guesses:
109         print('You already guessed that')
110     elif new_guess in word:
111         prev_guesses.add(new_guess)
112         last_pos = 0
113         while True:
114             pos = word.find(new_guess, last_pos)
115             if pos < 0:
116                 break
117             elif pos >= 0:
118                 guessed[pos] = new_guess
119                 last_pos = pos + 1
120     else:
121         num_misses += 1
122
123     play({
```

```
124          'word': word,
125          'guessed': guessed,
126          'num_misses': num_misses,
127          'prev_guesses': prev_guesses,
128          'max_misses': max_misses
129      })
130
131
132  # -----------------------------------------------
133  if __name__ == '__main__':
134      main()
```

Chapter 11

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 sapiens', 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 Centrifuge data')
    parser.add_argument('tsv_file', metavar='file',
                        help='Sample TSV file', nargs='+')
    parser.add_argument('-d', '--dbname', help='Centrifuge db name',
                        metavar='str', type=str, default='centrifuge.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_file))
            sys.exit(1)

        if sample_name.endswith('.centrifuge'):
            sample_name = re.sub(r'\.centrifuge$', '', sample_name)

        sample_id = import_sample(sample_name, db)
        print('{:3}: Importing "{}" ({})'.format(fnum + 1,
                                                 sample_name, sample_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_unique_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_size)
        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(ncbi_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:

```
#!/usr/bin/env python3
"""Query centrifuge.db for NCBI tax id"""

import argparse
import os
import re
import sys
import sqlite3
from tabulate import tabulate

# --------------------------------------------------
def get_args():
    """get args"""
    parser = argparse.ArgumentParser(description='Argparse Python script')
    parser.add_argument('-d', '--dbname', help='Centrifuge db name',
                        metavar='str', type=str, default='centrifuge.db')
    parser.add_argument('-o', '--orderby', help='Order by',
                        metavar='str', type=str, default='abundance')
    parser.add_argument('-s', '--sortorder', help='Sort order',
                        metavar='str', type=str, default='desc')
    parser.add_argument('-t', '--taxid', help='NCBI taxonomy id',
                        metavar='str', type=str, required=True)
    return parser.parse_args()

# --------------------------------------------------
def main():
    """main"""
    args = get_args()
    dbname = args.dbname
    order_by = args.orderby
    sort_order = args.sortorder

    if not os.path.isfile(dbname):
        print('"{}" is not a valid file'.format(dbname))
        sys.exit(1)

    flds = set(['tax_name', 'num_reads', 'abundance', 'sample_name'])
    if not order_by in flds:
        print('"{}" not an allowed --orderby, choose from {}'.format(
            order_by, ', '.join(flds)))
        sys.exit(1)

    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_reads, 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_order)
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
```

Chapter 12

Regular Expressions

The term "regular expression" is a formal, linguistic term that describes the ability to desc you might be interested to read about (https://en.wikipedia.org/wiki/Regular_language). For our purposes, regular expressions (AKA "regexes" or a "regex") is a way to formally describe some string that we want to find. Regexes are a DSL (domain-specific language) that we use inside Python, just like in the previous chapter we use SQL statements to communite with SQLite. We can `import re` to use the Python regular expression module and use it to search text.

In the tic-tac-toe exercise, we needed to see if the `--player` argument was exactly one character that was either an 'X' or an 'O'. Here's code that can do that:

```
>>> player = 'X'
>>> if len(player) == 1 and (player == 'X' or player == 'O'):
...   print('OK')
...
OK
>>> player = 'B'
>>> if len(player) == 1 and (player == 'X' or player == 'O'):
...   print('OK')
...
```

A shorter way to write this could be:

```
if len(player) == 1 and player in 'XO':
```

It's not too onerous, but it quickly gets worse as we get more complicated requirements. In that same exercise, we needed to check if `--state` was exactly 9 characters composed entirely of '.', 'X', 'O':

```
>>> state = 'XXX...OOO'
>>> 'OK' if len(state) == 9 and all(map(lambda x: x in 'XO.', state)) else 'No'
'OK'
>>> state = 'XXX...OOA'
>>> 'OK' if len(state) == 9 and all(map(lambda x: x in 'XO.', state)) else 'No'
'No'
```

A regular expression allows us to **describe** what we want rather than **implement** the code to find what we want. We can create a class of allowed characters with `[XO]` and additionally constraint it to be exactly one character wide with `{1}` after the class. (Note that `{}` for match length can be in the format `{exactly}`, `{min,max}`, `{min,}`, or `{,max}`.)

```
>>> import re
>>> player = 'X'
>>> re.match('[XO]{1}', player)
<_sre.SRE_Match object; span=(0, 1), match='X'>
>>> player = 'A'
>>> re.match('[XO]{1}', player)
```

We can extend this to our state problem:

```
>>> state = 'XXX...OOO'
>>> re.match('[XO.]{9}', state)
<_sre.SRE_Match object; span=(0, 9), match='XXX...OOO'>
>>> state = 'XXX...OOA'
>>> re.match('[XO.]{9}', state)
```

When we were starting out with the Unix command line, one exercise had us using `grep` to look for lines that start with vowels. One solution was:

```
$ grep -io '^[aeiou]' scarlet.txt | sort | uniq -c
  59 A
  10 E
  91 I
  20 O
   6 U
 651 a
 199 e
 356 i
 358 o
 106 u
```

We used square brackets `[]` to enumerate all the vowels `[aeiou]` and used the `-i` flag to `grep` to indicate it should match case **insensitively**. Additionally, the `^` indicated that the match should occur at the start of the string.

ENA Metadata

Let's examine the ENA metadata from the XML parsing example. We see there are many ways that latitude/longitude have been represented:

```
$ ./xml_ena.py *.xml | grep lat_lon
attr.lat_lon             : 27.83387,-65.4906
attr.lat_lon             : 29.3 N 122.08 E
attr.lat_lon             : 28.56_-88.70377
attr.lat_lon             : 39.283N 76.611 W
attr.lat_lon             : 78 N 5 E
attr.lat_lon             : missing
attr.lat_lon             : 0.00 N, 170.00 W
```

```
attr.lat_lon              : 11.46'45.7" 93.01'22.3"
```

How can we go about parsing all the various ways this data has been encoded? Regular expressions provide us a way to describe in very specific way what we want.

Let's start just with the idea of matching a number (where "number" is a string that could be parsed into a number) like "27.83387":

```
>>> import re
>>> re.search('\d', '27.83387')
<_sre.SRE_Match object; span=(0, 1), match='2'>
```

The `\d` pattern means "any number" which is the same as `[0-9]` where the `[]` creates a class of characters and `0-9` expands to all the numbers from zero to nine. The problem is that it only matches one number, `2`. Change it to `\d+` to indicate "one or more numbers":

```
>>> re.search('\d+', '27.83387')
<_sre.SRE_Match object; span=(0, 2), match='27'>
```

Now let's capture the decimal point:

```
>>> re.search('\d+.', '27.83387')
<_sre.SRE_Match object; span=(0, 3), match='27.'>
```

You might think that's perfect, but the `.` has a special meaning in regex. It means "one of anything", so it matches this, too:

```
>>> re.search('\d+.', '27x83387')
<_sre.SRE_Match object; span=(0, 3), match='27x'>
```

To indicate we want a literal `.` we have to make it `\.` (backslash-escape):

```
>>> re.search('\d+\.', '27.83387')
<_sre.SRE_Match object; span=(0, 3), match='27.'>
>>> re.search('\d+\.', '27x83387')
```

Notice that the second try returns nothing.

To capture the bit after the `.`, add more numbers:

```
>>> re.search('\d+\.\d+', '27.83387')
<_sre.SRE_Match object; span=(0, 8), match='27.83387'>
```

But we won't always see floats. Can we make this regex match integers, too? We can indicate that part of a pattern is optional by putting a `?` after it. Since we need more than one thing to be optional, we need to wrap it in parens:

```
>>> re.search('\d+\.\d+', '27')
>>> re.search('\d+(\.\d+)?', '27')
<_sre.SRE_Match object; span=(0, 2), match='27'>
>>> re.search('\d+(\.\d+)?', n1)
<_sre.SRE_Match object; span=(0, 8), match='27.83387'>
```

What if there is a negative symbol in front? Add `-?` (an optional dash) at the beginning:

```
>>> re.search('-?\d+(\.\d+)?', '-27.83387')
<_sre.SRE_Match object; span=(0, 9), match='-27.83387'>
>>> re.search('-?\d+(\.\d+)?', '27.83387')
<_sre.SRE_Match object; span=(0, 8), match='27.83387'>
>>> re.search('-?\d+(\.\d+)?', '-27')
<_sre.SRE_Match object; span=(0, 3), match='-27'>
>>> re.search('-?\d+(\.\d+)?', '27')
<_sre.SRE_Match object; span=(0, 2), match='27'>
```

Sometimes we actually find a `+` at the beginning, so we can make an optional character class `[+-]?`:

```
>>> re.search('[+-]?\d+(\.\d+)?', '-27.83387')
<_sre.SRE_Match object; span=(0, 9), match='-27.83387'>
>>> re.search('[+-]?\d+(\.\d+)?', '+27.83387')
<_sre.SRE_Match object; span=(0, 9), match='+27.83387'>
>>> re.search('[+-]?\d+(\.\d+)?', '27.83387')
<_sre.SRE_Match object; span=(0, 8), match='27.83387'>
```

Now we can match things that basically look like a floating point number or an integer, both positive and negative.

Usually the data we want to find it part of a larger string, however, and the above fails to capture more than one thing, e.g.:

```
>>> re.search('[+-]?\d+(\.\d+)?', 'Lat is "-27.83387" and lon is "+132.43."')
<_sre.SRE_Match object; span=(8, 17), match='-27.83387'>
```

We really need to match more than once using our pattern matching to extract data. We saw earlier that we can use parens to group optional patterns, but the parens also end up creating a **capture group** that we can refer to by position:

```
>>> re.findall('([+-]?\d+(\.\d+)?)', '(-27.83387, +132.43)')
[('-27.83387', '.83387'), ('+132.43', '.43')]
```

OK, it was a bit unexpected that we have matches for both the whole float and the decimal part. This is because of the dual nature of the parens, and in the case of using them to group the optional part we are also creating another capture. If we change `()` to `(?:)`, we make this a non-capturing group:

```
>>> re.findall('([+-]?\d+(?:\.\d+)?)', 'lat_lon: (-27.83387, +132.43)')
['-27.83387', '+132.43']
```

There are many resources you can use to thoroughly learn regular expressions, so I won't try to cover them completely here. I will mostly try to introduce the general idea and show you some useful regexes you could steal.

Here is an example of how you can embed regexes in your Python code. This version can parse all the versions of latitude/longitude shown above. This code

uses parens to create capture groups which it then uses `match.group(n)` to extract:

```
$ cat -n parse_lat_lon.py
     1  #!/usr/bin/env python3
     2
     3  import os
     4  import re
     5  import sys
     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  float_ = r'[+-]?\d+\.*\d*'
    16  ll1 = re.compile('(' + float_ + ')\s*[,_]\s*(' + float_ + ')')
    17  ll2 = re.compile('(' + float_ + ')(?:\s*([NS]))?(?:\s*,)?\s+(' + float_ +
    18                   ')(?:\s*([EW])?)')
    19  loc_hms = r"""
    20  \d+\.\d+'\d+\.\d+"
    21  """.strip()
    22  ll3 = re.compile('(' + loc_hms + ')\s+(' + loc_hms + ')')
    23
    24  for line in open(file):
    25      line = line.rstrip()
    26      ll_match1 = ll1.search(line)
    27      ll_match2 = ll2.search(line)
    28      ll_match3 = ll3.search(line)
    29
    30      if ll_match1:
    31          lat, lon = ll_match1.group(1), ll_match1.group(2)
    32          lat = float(lat)
    33          lon = float(lon)
    34          print('lat = {}, lon = {}'.format(lat, lon))
    35      elif ll_match2:
    36          lat, lat_dir, lon, lon_dir = ll_match2.group(
    37              1), ll_match2.group(2), ll_match2.group(
    38                  3), ll_match2.group(4)
    39          lat = float(lat)
    40          lon = float(lon)
    41
    42          if lat_dir == 'S':
```

```
    43              lat *= -1
    44
    45          if lon_dir == 'W':
    46              lon *= -1
    47          print('lat = {}, lon = {}'.format(lat, lon))
    48      elif ll_match3:
    49          lat, lon = ll_match3.group(1), ll_match3.group(2)
    50          print('lat = {}, lon = {}'.format(lat, lon))
    51      else:
    52          print('No match: "{}"'.format(line))
$ cat lat_lon.txt
attr.lat_lon             : 27.83387,-65.4906
attr.lat_lon             : 29.3 N 122.08 E
attr.lat_lon             : 28.56_-88.70377
This line will not be included
attr.lat_lon             : 39.283N 76.611 W
attr.lat_lon             : 78 N 5 E
attr.lat_lon             : missing
attr.lat_lon             : 0.00 N, 170.00 W
attr.lat_lon             : 11.46'45.7" 93.01'22.3"
$ ./parse_lat_lon.py lat_lon.txt
lat = 27.83387, lon = -65.4906
lat = 29.3, lon = 122.08
lat = 28.56, lon = -88.70377
No match: "This line will not be included"
lat = 39.283, lon = -76.611
lat = 78.0, lon = 5.0
No match: "attr.lat_lon             : missing"
lat = 0.0, lon = -170.0
lat = 11.46'45.7", lon = 93.01'22.3"
```

We see a similar problem with "collection_date":

```
$ ./xml_ena.py *.xml | grep collection
attr.collection_date     : March 24, 2014
attr.collection_date     : 2013-08-15/2013-08-28
attr.collection_date     : 20100910
attr.collection_date     : 02-May-2012
attr.collection_date     : Jul-2009
attr.collection_date     : missing
attr.collection_date     : 2013-12-23
attr.collection_date     : 5/04/2012
```

Imagine how you might go about parsing all these various representations of dates. Be aware that parsing date/time formats is so problematic and ubiquitous that many people have already written modules to assist you!

To run the code below, you will need to install the `dateparser` module:

```
$ python3 -m pip install dateparser
```

Et voila!

```
>>> import dateparser as p
>>> p.parse('March 24, 2014')
datetime.datetime(2014, 3, 24, 0, 0)
>>> p.parse('2013-08-15')
datetime.datetime(2013, 8, 15, 0, 0)
>>> p.parse('20100910')
datetime.datetime(2000, 2, 1, 9, 1)
>>> p.parse('02-May-2012')
datetime.datetime(2012, 5, 2, 0, 0)
>>> p.parse('Jul-2009')
datetime.datetime(2009, 7, 23, 0, 0)
>>> p.parse('5/04/2012')
datetime.datetime(2012, 5, 4, 0, 0)
```

You can see it's not perfect, e.g., "Jul-2009" should not resolve to the 23rd of July, but, honestly, what should it be? (Is the 1st any better?!) Still, this saves you writing a lot of code. And, trust me, **THIS IS REAL DATA**! While trying to parse latitude, longitude, collection date, and depth for 35K marine metagenomes from the ENA, I wrote a hundreds of lines of code and dozens of regular expressions!

Chapter 13

Writing Pipelines in Python

There are existing frameworks for writing pipelines such as Snakemake. Even humble `make` could be coerced into handling a pipeline, but it is good to know how to roll your own in Python as you will learn many valuable skills by doing so.

Chapter 14

HPC

HPC is an acronym for "high-performance computing," and it generally means using a cluster of computers. Our students have access to the Ocelote cluster 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.

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 Python 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 take 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
}
```

Parallel

One of the easiest ways to boost the performance of a simple bash or Python program is to run many copies of it in parallel. The GNU Parallel program (https://www.gnu.org/software/parallel/) will read all the jobs you wish to run and will distribute them over all the available CPUs, starting new jobs as others finish, and exiting if there is a problem (one of your programs exits with an error code). There is a similar implementation written in the Rust language you may also wish to use (https://github.com/mmstick/parallel).

Here is a bash program to illustrate how to use GNU Parallel. We read a dictionary file and print the first 25 words with their order to demonstrate that the order in which the jobs are executed is not necessarily the way they are written to the jobs file.

```
$ cat -n parallel.sh
     1    #!/usr/bin/env bash
     2
     3    set -u
     4
     5    MAX=25
     6    WORDS=/usr/share/dict/words
     7    CORES=4
     8
     9    if [[ ! -f "$WORDS" ]]; then
    10        echo "WORDS \"$WORDS\" is not a file"
    11        exit 1
    12    fi
    13
    14    TMP=$(mktemp)
    15    i=0
    16    while read -r WORD; do
    17        i=$((i+1))
    18        echo "echo \"$i $WORD\"" >> "$TMP"
```

```
    19        if [[ $i -eq $MAX ]]; then
    20            break
    21        fi
    22    done < "$WORDS"
    23
    24    echo "Starting parallel on $CORES cores"
    25    parallel -j $CORES --halt soon,fail=1 < "$TMP"
    26    echo "Finished parallel"
$ ./parallel.sh
Starting parallel on 4 cores
3 aa
4 aal
5 aalii
6 aam
7 Aani
8 aardvark
9 aardwolf
10 Aaron
11 Aaronic
12 Aaronical
13 Aaronite
14 Aaronitic
15 Aaru
16 Ab
17 aba
18 Ababdeh
19 Ababua
20 abac
21 abaca
22 abacate
23 abacay
24 abacinate
2 a
25 abacination
1 A
Finished parallel
```

And here is a Python implementation of the same idea:

```
$ cat -n parallel.py
     1    #!/usr/bin/env python3
     2    """
     3    Author : kyclark
     4    Date   : 2019-02-25
     5    Purpose: Demonstrate GNU Parallel
     6    """
     7
```

```
 8  import argparse
 9  import os
10  import sys
11  import subprocess
12  import tempfile as tmp
13
14
15  # --------------------------------------------------
16  def get_args():
17      """get command-line arguments"""
18      parser = argparse.ArgumentParser(
19          description='Demonstrate GNU Parallel',
20          formatter_class=argparse.ArgumentDefaultsHelpFormatter)
21
22      parser.add_argument(
23          '-f',
24          '--file',
25          metavar='FILE',
26          help='A positional argument',
27          default='/usr/share/dict/words')
28
29      parser.add_argument(
30          '-c',
31          '--cores',
32          help='Number of cores',
33          metavar='INT',
34          type=int,
35          default=4)
36
37      parser.add_argument(
38          '-m',
39          '--max_lines',
40          help='Maximum number of input lines',
41          metavar='INT',
42          type=int,
43          default=25)
44
45      return parser.parse_args()
46
47
48  # --------------------------------------------------
49  def warn(msg):
50      """Print a message to STDERR"""
51      print(msg, file=sys.stderr)
52
53
```

```
    54    # --------------------------------------------------
    55    def die(msg='Something bad happened'):
    56        """warn() and exit with error"""
    57        warn(msg)
    58        sys.exit(1)
    59
    60
    61    # --------------------------------------------------
    62    def main():
    63        """Make a jazz noise here"""
    64        args = get_args()
    65        in_file = args.file
    66        max_lines = args.max_lines
    67        num_cores = args.cores
    68
    69        if not os.path.isfile(in_file):
    70            die('"{}" is not a file'.format(in_file))
    71
    72        jobfile = tmp.NamedTemporaryFile(delete=False, mode='wt')
    73        for i, line in enumerate(open(in_file), start=1):
    74            jobfile.write('echo "{} {}"\n'.format(i, line.rstrip()))
    75            if i == max_lines: break
    76
    77        jobfile.close()
    78
    79        print('Starting parallel on {} cores'.format(num_cores))
    80        cmd = 'parallel -j {} --halt soon,fail=1 < {}'.format(
    81            num_cores, jobfile.name)
    82
    83        try:
    84            subprocess.run(cmd, shell=True, check=True)
    85        except subprocess.CalledProcessError as err:
    86            die('Error:\n{}\n{}\n'.format(err.stderr, err.stdout))
    87        finally:
    88            os.remove(jobfile.name)
    89
    90        print('Finished parallel')
    91
    92
    93    # --------------------------------------------------
    94    if __name__ == '__main__':
    95        main()
$ ./parallel.py
Starting parallel on 4 cores
3 aa
4 aal
```

```
5 aalii
6 aam
7 Aani
8 aardvark
9 aardwolf
10 Aaron
11 Aaronic
12 Aaronical
13 Aaronite
14 Aaronitic
15 Aaru
16 Ab
17 aba
18 Ababdeh
19 Ababua
20 abac
21 abaca
22 abacate
23 abacay
24 abacinate
2 a
25 abacination
1 A
Finished parallel
```

The key to being able to use this idea to your advantage is to see when you can split and process your input files without affecting the outcome of your analysis. For instance, if you have 100K sequences you wish to BLAST against a database, perhaps you can split sequences into 10 files of 1K sequences (see "fa_split.py" in the Python Parsing chapter), write out a job file with 10 BLAST jobs, and have `parallel` process them as quickly as possible. Keep in mind, however, that the number of cores may not be your limiting factor. If, for instance, the BLAST database is large and requires a significant amount of your RAM (https://www.youtube.com/watch?v=NdREEcfaihg), then running several jobs in parallel is likely to consume all available memory causing your OS to start swapping to disk ("thrashing") and performance will nosedive. If you have need of multiple, high-memory jobs, then it's necessary to move from parallelizing over all the cores on one machine to parallelizing over many machines (and possibly over all their cores).

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    99310:56/72:00/100000:00
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 Memory Time  S Time
--------------- -------- -------- ---------- ------ --- --- ------ ----- - -----
818089.service0 kyclark  clu_stan hello.sh      --    1   1    1gb 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 remaining for the queue clu_standar
```

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/ncftpget
     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 (and we don't know about or have access to GNU Parallel). 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     #!/usr/bin/env bash
     2
     3     set -u
     4
     5     export FTP_LIST=get-me
     6     export OUT_DIR=$PWD
     7     export NCFTPGET=/rsgrps/bhurwitz/hurwitzlab/bin/ncftpget
     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_LIST,NCFTPGET -j oe -o $PBSDI
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     #!/usr/bin/env 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_FILES > $TMP_FILES
    19     NUM_FILES=$(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 selec
```

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
```

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/kyclark)...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.out
[tacc:login4@/work/03137/kyclark/metagenomics-book/hpc/slurm/hello]$ cat slurm-7560143.out
Hello from sunny "c557-904.stampede.tacc.utexas.edu"!
```