Contents

Introduction: Bioinformatics Primer for bash and Python

This book is dedicated to the students and post-docs in science who've come from the bench and now need to learn how to use the command line. In my experience, few ever get any formal training in how to write software, so I want to give you as many examples as possible of complete, command-line programs that work, have documentation, and are testable and reproducible.

I got into bioinformatics by coming from the software industry and having to learn enough biology to write the code my bosses requested. I know well the feeling of being in over your head. Even after 24 years of programming, 18 in bioinformatics, I still sometimes feel completely lost in how to write a particular piece of code. In my experience, the best help for me is working examples of code that I can copy and paste into my programs, and so that's what I'm giving you.

Organization

> "The only way to learn a new programming language is by writing programs in it." - Dennis Ritchie

The best way to learn is by *doing*, so along with reading material about bash and Python, I will give you many programs that I want you to *write* in those languages. Each one will have a README that lays out the specifications ("specs") along with examples of how the program should work. Also included will be sample input files and a `test.py` program which you can run with the command `make test` (if you have `make` on your system) or `pytest -v test.py`.

I include my own version of a solution that you can use to compare. I spent many years in the Perl community where "There Is More Than One Way To Do It" (TIMTOWTDI) is something of a mantra whereas the Python community espouses "There should be one – and preferably only one – obvious way to do it" (https://www.python.org/dev/peps/pep-0020/). I disagree with this notion and believe you can find many creative and beautiful solutions. More than anything, the solution that you figured out, that you understand, and that satisfies the test suite is the "right" one for you. Your style will change as you grow more knowledgeable and confident in your programming skills.

Shell

Most of bioinformatics happens on the Unix command line or "shell", so we need to start there. There are many terrific resources for learning from books to online courses like Coursera and the Software and Data Carpentries websites and workshops. I highly recommend you use all those to supplement the brief overview I will provide.

Shell Scripting

Shell scripts are commands written into a file and executed sequentially, top-to-bottom, so once you learn Unix commands, we can put those commands into files and run them later. This makes them documented and reproducible!

Make

We can further record commands and workflows by abusing GNU `make` and "makefiles."

Git

We use `git` to manage our source code.

Python

Once you have an idea how to use Unix to manage files, permissions, and execute programs, we will start learning Python.

Appendices

Several topics are common to many or all the programs presented such as how to use `argparse` or regular expressions. I also provide two appendices showing small pieces of code in `bash` and Python that do some specific task like check if a file exists or read a file line-by-line. You should look over this section and piece together ideas to accomplish the tasks.

After completing this material, 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

Getting the Source Code

To get the source code for the book, I recommend you go to https://github.com/kyclark/practical_python_for_
and click the "fork" button in the upper-right and then add this repo as an upstream source:

```
$ git clone <your_fork_of_the_repo> ppds
$ cd ppds
$ REPO=https://github.com/kyclark/practical_python_for_data_science.git
$ git remote add upstream $REPO
```

To get new content, use `git pull upstream master`. I write the exercises in such a way that you will create new content that should not conflict with content I make.

Programming Environment

The material begins with the Unix command line. If you are working on Windows, I highly recommend you install Windows Subsystem for Linux and probably GitBash. If you are on an Apple computer, you have a full Unix system available through your Terminal app. The author uses a Mac with iTerm and vim editor to write, debug, and run programs. You may wish to use an editor like Sublime, TextWrangler, or Atom or an integrated develoment environment (IDE) like VSCode or PyCharm. However you choose to *write* code, this material assume you will *run* it from the command line. For many reasons, I have chosen not to use Jupyter Notebooks. Some chapters may include a Notebook, but I would prefer to have students write command-line programs and use a testing framework like PyTest to ensure that code runs correctly, top to bottom.

Python

I personally prefer statically typed and "functional" languages like Rust, Elm, and Haskell, but I concede the dominance of dynamically typed languages such as Perl, Python, and Ruby. This material is intended to steer the student towards best practices when working in Python to avoid what I consider to be dangerous tendencies of the language. I will be using Python version 3.

Author

> "Computer programming has always been a self-taught, maverick occupation." - Ellen Ullman
>
> "Every great developer you know got there by solving problems they were unqualified to solve until they actually did it." - Patrick McKenzie

My name is Ken Youens-Clark (https://orcid.org/0000-0001-9961-144X), and I'm a Senior Scientific Programmer at the University of Arizona. While I have a Masters of Science in Biosystems Engineering, my undergraduate trianing was a BA in English Lit with a minor in music. As a kid, I had a RadioShack computer (probably a TRS-80, but I don't really remember) on which I wrote maybe two programs in BASIC. I never got into computing until after completing my bachelor's degree in 1995 when I started playing with computers and databases at my first job. The next year I landed a position where I learned Visual Basic on Windows 3.1, and that was when I actually got hooked on programming and problem solving. Since then, I've worked in several languages on various operating systems, and I've learned by doing – always having to tackle problems I was never trained to solve. I spent the longest part of my career using Perl in a bioinformatics settting, but Python has definitely taken over the data processing and machine learning space, so that's what I'll cover.

Acknowledgements

I would not be where I am professionally without a string of terrific bosses who took a chance and hired me, taught me, and let me move on, starting with Eric Thorsen who showed me Visual Basic, Mike Doll who taught me to double every time estimate, Steve Reppucci who honed my Perl and sysadmin skills, Dr. Lincoln Stein who introduced me to bioinformatics and created the "Programming for Biologists" at Cold Spring Harbor Laboratory, Dr. Doreen Ware who let me work on all kinds of terribly interesting problems, and Dr. Bonnie Hurwitz who allowed me to write and present novel materials to her metagenomics and bioinformatics classes. I also would like to thank the students who showed me how to improve my lectures and assignments. Most importantly, I would be nothing without the love and support of my wife and children.

Copyright

Chapter 1: The Unix Command Line

Whatever operating system you are on, I assume you have access to some Unix-like command line. That is, you are looking at a blank screen (probably) with a prompt like `$` waiting for you to type something. You could try typing "hello" and then the Enter key, and you'll probably see something like this:

```
$ hello
-bash: hello: command not found
```

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.

The first word on the command line needs to be a *command*. On my system, there's no program or built-in command called `hello` anywhere to be found (specifically in the `$PATH`, but we'll get to that in a bit), so it tells me `command not found`. Try `help` instead and you'll likely see quite a bit of output if you are on a system that is running GNU's `bash` shell. The command `hostname` should work pretty evrerywhere, telling you the name of your machine. Certainly `ls` will work to show you a directory listing. Try it!

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

Let's look at some more commands you can do. This is by no means an exhaustive list, just a few to get you going. Try running each of them. To learn more about the tools, try both `man cmd` or `cmd -h` or `cmd --help`.

- **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 `/` (which is a "slash" – the thing leaning the *other way* is a "backslash") and is called the "root directory." We can "list" the contents of a directory with `ls`. Without any arguments, this prints the contents of the current working directory which you can print with `pwd` (print working directory). You can `ls /` to see the contents of the root directory or `ls $HOME` to see your own home directory.

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

The `head` program will show you the first few lines of some text. Try it on a file:

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

It can also work on something called "standard in" which is often written as `STDIN`. The output it created above showed up on "standard out" (`STDOUT`). We'll also talk about "standard error" (`STDERR`). Most Unix commands print regular output to STDOUT and errors to STDERR. Many programs can read another program's STDOUT as their own STDIN.

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 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
```

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 |
                ||     ||
```

You could also chain that to `lolcat` if you're really, really lucky.

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 `$`.

```
$ SECRET=ilikecake
$ echo $SECRET
ilikecake
$ echo SECRET
SECRET
```

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

```
$ unset SECRET
$ 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

> I feel like there should seriously be some sort of first-year level class where people learn how to add something to their path and what that means. – Kristopher Micinski

Your `$PATH` setting is an ordered, colon-delimited list of directories that will be searched to find programs. Run `echo $PATH` to see yours. It probably looks something like this:

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
```

Here's my `$PATH` on my laptop:

```
$ echo $PATH | gsed "s/:/\n/g"
/Users/kyclark/.cargo/bin
/Users/kyclark/bin
/anaconda3/bin
/Users/kyclark/.local/bin
```

```
/Users/kyclark/work/cyverse-cli/bin
/usr/local/sbin
/usr/local/bin
/usr/bin
/bin
/usr/sbin
/sbin
/Library/TeX/texbin
```

And here is my `$PATH` on the HPC at the University of Ariziona:

```
$ 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" (or "gsed" which is GNU sed) to add a newline after each colon so you can more easily see that the directories are separated by colons. If you use the `which` command to see where a program lives, you can see that it is found in a directory that is included in your `$PATH`:

```
$ which sed
/usr/bin/sed
$ which python
/anaconda3/bin/python
```

By definition, if a program's location is not in your `$PATH`, then `which` cannot find it, and that means you cannot run it without giving a full path to the program, e.g., `/usr/sbin/foobar/baz`.

In this repository I have included a `bin` directory that has some useful Python programs like `new_py.py` which we will use later to stub out new Python programs. In order to use them, you have three options:

1. Use the complete path to the programs. E.g., if you have installed this into `$HOME/ppds`, then execute `$HOME/ppds/bin/new_py.py`
2. Copy the contents of the `bin` directory to one of the other directories that are already in your `$PATH`, e.g., `cp $HOME/ppds/bin/* /usr/local/bin`, but that might require root privilege that you don't have.
3. Add `$HOME/ppds/bin` to your `$PATH`

```
export PATH="$HOME/ppds/bin:$PATH"
```

You just told your shell (bash) to set the `$PATH` variable to `$HOME/ppds/bin` plus whatever is was before. Since we want this to happen each time we log in, so we can add this command to `$HOME/.bashrc`:

```
echo "export PATH=$HOME/ppds/bin:$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.

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:

```
$ 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:

```
$ 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`:

```
$ 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:

```
$ w | awk 'NR>2 {print $1}'
kyclark
emsenhub
joneska
```

```
shawtaro
darrenc
guven
guven
guven
joneska
dmarrone
```

Let's `uniq` that output:

```
$ 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:

```
$ 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

```
$ 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

The above were somewhat contrived examples. Here's a real problem I had to solve for a labmate who wanted 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
```

Chapter 2: 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)

Stories contain within them their own dimensions. Some are naturally short while others are sprawling novels. Likewise, sometimes a program is just a collection of `bash` commands, so let's figure out how to write a decent one. As a general rule, if you write more than 20-30 lines of `bash` in a program, you should probably move to a more powerful language like Python.

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 a newline (the `Return` or `Enter` key). 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
  }
}
```

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
```

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.

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
```

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

FILE='meta.tab'
while read -r SITE LOC; do
    echo "$SITE is located at \"$LOC\""
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"
```

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
#!/usr/bin/env bash

set -u

if [[ $# -ne 1 ]]; then
    printf "Usage: %s DIR\n" "$(basename "$0")"
    exit 1
fi

DIR=$1
TMP=$(mktemp)
find "$DIR" -type f -name \*.fa > "$TMP"
NUM_FILES=$(wc -l "$TMP" | awk '{print $1}')

if [[ $NUM_FILES -lt 1 ]]; then
    echo "Found no .fa files in $DIR"
    exit 1
fi

NUM_SEQS=0
while read -r FILE; do
    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.

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
#!/usr/bin/env bash

NAME=${1:-$USER}
[[ -z "$NAME" ]] && NAME='Stranger'
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
#!/usr/bin/env bash

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
```

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

set -u

GREETING=""
NAME="Stranger"
EXCITED=0

function USAGE() {
    printf "Usage:\n  %s -g GREETING [-e] [-n NAME]\n\n" $(basename $0)
    echo "Required arguments:"
    echo " -g GREETING"
    echo
    echo "Options:"
    echo " -n NAME ($NAME)"
    echo " -e Print exclamation mark (default yes)"
    echo
    exit ${1:-0}
}

[[ $# -eq 0 ]] && USAGE 1

while getopts :g:n:eh OPT; do
  case $OPT in
    h)
      USAGE
      ;;
    e)
      EXCITED=1
      ;;
    g)
      GREETING="$OPTARG"
      ;;
    n)
      NAME="$OPTARG"
      ;;
    :)
      echo "Error: Option -$OPTARG requires an argument."
      exit 1
      ;;
    \?)
      echo "Error: Invalid option: -${OPTARG:-""}"
      exit 1
  esac
done

```

```
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"
```

I wouldn't recommend trying to do much more with `bash` scripting. As you get more complicated arguments and options, it's really time to move to Python where we have libraries that do the hard work of parsing out the command line.

Chapter 3: Using GNU Parallel to Run Concurrent Processes

"GNU parallel is a shell tool for executing jobs in parallel using one or more computers." (https://www.gnu.org/software/parallel/). To imagine working in parallel, think about the construction of the First Transcontinental Railroad that linked the Omaha to San Francisco. The companies involved didn't start from one end and build it to the other. The track was built in independent sections that eventually connected as this was faster and more efficient. If you have a large job that can be broken into smaller tasks that can be run independently from each other, it's more efficient to use multiple processors possibly over many machines to run as many tasks concurrently than to run one big task.

Imagine you need to BLAST several million sequences. You could just run `blastn` on the file and wait a few days for it to finish. Alternatively, you could split the sequences into several files and distribute the BLAST commands to several machines each of which might finish in hours rather than days. At the end, you would need only to concatenate the BLAST hits to get the same answer you would have gotten from BLASTing all the sequences in one file.

The advantage of using an HPCC (high performance computing cluster) is that you have access to several "nodes" (machines), each of which can have many "cores" (CPUs). You split up the sequences, then tell the HPC scheduler how many machines with what kind of memory requirements you need for how long, and it will schedule and run the jobs for you as machines become available.

Unfortunately, not everyone working in bioinformatics has ready access to a HPC cluster. Still, it's possible that you could enjoy the benefits of parallel computing. It's likely that even your laptop has more than one CPU that could be used in parallel or maybe your lab or PI has a beefy server somewhere that has 12-24 processors. If you write all the commands you need to run into to a file, you can then use `parallel` to use execute those commands using as many CPUs as you desire. As jobs finish, `parallel` will launch more, always keeping all the cores busy, much like an HPC scheduler.

Jobs like BLAST aren't actually great to parallelize because BLAST will often require all the available memory on the node, but something like converting FASTQ files to FASTA format is perfect to farm out to multiple CPUs.

"Hello" Program

We'll start simple by pretending this "hello.sh" is something more interesting than it really is:

```
$ cat -n hello.sh
```

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

if [[ $# -lt 1 ]]; then
    printf "Usage: %s NAME\n" $(basename $0)
    exit 1
fi

NAME=$1

if [[ $NAME == 'Lord Voldemort' ]]; then
    echo "Upon advice of my counsel, I respectfully refuse to say that name."
    exit 1
fi

echo "Hello, $1!"
```

```
$ ./hello.sh
Usage: hello.sh NAME
$ ./hello.sh Jan
Hello, Jan!
$ ./hello.sh "Lord Voldemort"
Upon advice of my counsel, I respectfully refuse to say that name.
```

Jobs File

We'll write a `jobs` file that will run this program with various names:

```
$ cat jobs
./hello.sh Bobby
./hello.sh "Lord Voldemort"
./hello.sh Jan
./hello.sh Greg
./hello.sh Marcia
```

In a `Makefile`, I've documented several ways we could run this.

```
$ cat Makefile
.PHONY: shell parallel halt

JOBS = 'jobs'

shell:
    bash $(JOBS)

parallel:
    parallel -j 2 < $(JOBS)
```

```
halt:
	parallel -j 2 --halt soon,fail=1 < $(JOBS)
```

Running Jobs with bash

The simplest way to execute all the jobs is to tell `bash` to execute the lines in the `jobs` file:

```
$ make shell
bash 'jobs'
Hello, Bobby!
Upon advice of my counsel, I respectfully refuse to say that name.
Hello, Jan!
Hello, Greg!
Hello, Marcia!
```

Notice that the argument of "Lord Voldemort" actually triggers a non-zero exit code which is perceived by the system as an error (I think of an exit value of `0` as "zero errors"), but we didn't get a message that there was an error.

Running Jobs with parallel

Another option is to push the commands to `parallel` with an option `-j` to indicate how many CPUs to use concurrently. If you indicate more CPUs than you actually have, `parallel` will just use however many are available. If you don't tell `parallel` how many to use, it will use *all available CPUs* which is probably not what you want. It's often wise to leave 1 or 2 cores open for the machine itself! If we run `make parallel` to execute the `parallel` target, we see this:

```
$ make parallel
parallel -j 2 < 'jobs'
Hello, Bobby!
Upon advice of my counsel, I respectfully refuse to say that name.
Hello, Jan!
Hello, Greg!
Hello, Marcia!
make: *** [parallel] Error 1
```

Now we can easily see that one of the jobs failed. Sometimes you want everything to stop if you encounter a problem, e.g., one of your FASTA files was corrupted so you really need to fix it before finishing the rest of the analysis with incomplete data. You can tell `parallel` to "halt" when it encounters an error. See the `halt` target:

```
$ make halt
```

```
parallel -j 2 --halt soon,fail=1 < 'jobs'
Hello, Bobby!
Upon advice of my counsel, I respectfully refuse to say that name.
parallel: This job failed:
./hello.sh "Lord Voldemort"
parallel: Starting no more jobs. Waiting for 1 jobs to finish.
Hello, Jan!
make: *** [halt] Error 1
```

We didn't get to the end of the jobs file because the failure caused the whole process to stop. It happended that "Jan" was greeted after the error, but no other jobs were started because we told `parallel` to halt as soon as possible after encountering any error.

Dynamically Writing a Jobs File

It's not typical that you would manually write a jobs file. Usually you have some input files or directories from the user and then need to go find all the files to process. Here is an example of reading the top 100 boys' names from 1945 birth records and sending those to our `hello.sh` program.

```
$ cat -n run_names.sh
     1	#!/usr/bin/env bash
     2	
     3	set -u
     4	
     5	HELLO=${1:-"./hello.sh"}
     6	NAMES="../../../inputs/1945-boys.txt"
     7	
     8	if [[ ! -f "$NAMES" ]]; then
     9	    echo "Missing NAMES \"$NAMES\""
    10	    exit 1
    11	fi
    12	
    13	JOBS=$(mktemp)
    14	i=0
    15	while read -r NAME; do
    16	    i=$((i+1))
    17	    echo "$HELLO \"#$i $NAME\"" >> "$JOBS"
    18	done < "$NAMES"
    19	
    20	parallel < "$JOBS"
    21	
    22	rm "$JOBS"
    23	echo "Done."
```

Notice that I include the rank of each name. If the jobs file were run by `bash`, you would see them printed in order from 1 to 100. Since we ask `parallel` to run it, you'll most likely see them out of order:

```
$ ./run_names.sh | tail
Hello, #94 Herbert!
Hello, #95 Victor!
Hello, #96 Gregory!
Hello, #97 Curtis!
Hello, #98 Bernard!
Hello, #99 Clifford!
Hello, #67 Ronnie!
Hello, #1 James!
Hello, #100 Gene!
Done.
```

Our `hello.sh` is trivial and runs too quickly for us to really see the benenfits of CPU usage. Here is an equally trival program that runs much more slowly:

```
$ cat long_hello.sh
#!/usr/bin/env bash

[[ $# -eq 1 ]] && sleep 3 && echo "Hello, $1!"
```

If there is an argument, the program waits (`sleep`) for 3 seconds and then greets the argument. Run it like the other and see how long it takes:

```
$ ./long_hello.sh Frank
Hello, Frank!
```

Now use this program with the `run_names.sh` program:

```
$ ./run_names.sh ./long_hello.sh
```

Then use a program like `top` or `htop` on your system to watch how the CPUs are being used!

Later we'll look at writing a pipeline in Python that writes a jobs file and executes it with `parallel` similar to `run_names.sh`.

Summary

I tend to use `parallel` in somewhere most of my pipelines, even if they will run on an HPC. (On the Stampede2 cluster at TACC, the machines on the default queue have 68 cores!) So, rather than write a Python program that will process all the files in a directory, I will tend to write it so that it handles just one file. Then I'll write a shell script to find all the input files and write a jobs file where each file is handled individually by the Python program. If `parallel` is available, then I'll have it execute the jobs using some given number of cores (like `n-2` where `n` is the total number of cores); otherwise I can always just run the commands as a `bash` batch file.

To paraphrase Dr. Ian Malcolm, just because you *can* parallelize jobs doesn't mean you always *should* do so. As stated before, jobs that use loads of memory like BLAST probably should not be unless more than one copy of the database can fit into memory. Jobs that are mostly I/O (input/output) are good candidates for use with `parallel`.

Chapter 4: Bash: Convert BAM to FASTA (bam2fa)

Building on the `parallel` chapter, here is an example of a `bash` program that will convert BAM files to FASTA. SAM and BAM are formats for storing sequence alignments and stand for "Sequence Alignment Map" which is plain text and "Binary Alignment Map" which is the same information but stored in a compressed format only readable by machines.

https://en.wikipedia.org/wiki/SAM_(file_format)

It's common to get SAM/BAM formats from a sequencing core as your sequences are often aligned to some reference like human. You can use `samtools` to look at the files and convert them to other formats easily enough. This program is designed to quickly convert a directory of BAM files into FASTA files that will live in some new directory.

Checking and unpacking arguments

After using `set -u` to catch basic mistakes, we check `$#` to see how many arguments we have (mnemonic: `#` is the symbol for "number"). We need at least one argument but no more than 3. Print a usage if needed. Then we put `$1`, `$2`, etc. into judiciously named variables that describe what they are. Too often I see programs where `$1` is used throughout; while valid, this makes the code unreadable whereas `$IN_DIR` reminds me what the variable is supposed to be.

Testing directories

The test `-d $IN_DIR` will be "True" if the string in `$IN_DIR` names a directory that exists. (Use `man test` to see other tests you can use.) We use `!` to negate it, meaning there is no directory with that name. If this is the case, we report the error and `exit 1`. It's very important to `exit` with a *non-zero exit code* when there is an error. If you are using `make` or `parallel` or other programs to chain this, you can ensure the chain will fail if a component fails. *This is wise and good.* You do not want to complete an analysis pipeline if some key step fails. You want to report errors and halt processing until the error is fixed!

Likewise with testing `$IN_DIR`, we test if `$OUT_DIR` exists and create it with `mkdir -p` if it does not. The `-p` option tells `mkdir` to create "parent" directories as needed. If the user wanted the output files to go into `$HOME/projects/foo/bar/fasta`, `mkdir` would fail if all the directories up to `fasta` didn't exist. With `-p` it will create any needed parent directories.

Finding input files

Next I want to `find` inside `$IN_DIR` any files with a `-name` ending with `.bam` that are greater in `-size` than 0 characters/bytes. I put these into a temporary file so I can count them and later iterate over them. I do not like using the `bash` syntax for lists, so I always put lists of things into files. I find the number of lines in the file using `wc -l` and see if `$NUM` is less than one (`-lt 1`). If so, I alert the user and `exit 1` to indicate an error.

Temporary files

I prefer to use `mktemp` to get a temporary file. You could just overwrite a statically named `files.txt` file if you want, but you run the risk of accidentally overwriting a file that is still being used by another process. It's much safer to use `mktemp` as it guarantees a uniquely named file in a temporary directory. Additionally, if you forget to `rm` the file when you are done, it will likely be created in a location where old, unused files are regularly removed by the system.

Create jobs file

I make a second temp file for the commands that need to be run, one for each file. I use `while read` to read each line from the `$FILES` into a variable called `BAM` that is the name of the BAM file. I like to print out "1: foo.bam", "2: bar.bam", etc., while processing so I can see what is happening and how many files are being processed, so I increment a counter `$i` by one and use `printf` to print out the counter as three-characters-wide digit (`%3d`), followed by a colon (`:`), followed by a string (`%s`) where I'll show the `basename` of the file where:

```
$ basename foo/bar/baz.bam
baz.bam
```

I want to use the basename of the file as the new filename but with the `.bam` extension removed which I can note as the optional second argument to `basename`:

```
$ basename foo/bar/baz.bam ".bam"
baz
```

I create the name of the new `FASTA` file by creating a new string `$OUT_DIR/$BASE.fa`. I use the test `! -f` to check if the file *does not exist*; if so, I `echo` the `samtools` command to convert it to `fasta` format, redirecting with the single `>` to put the output from `samtools` into the new `$FASTA` file. This whole command gets *appended* with the double `>>` into the `$JOBS` file. (I've have used a single `>` in that instance more times than I care to admit, which means that I will only run the last job because the single `>` *overwrites* any existing content!)

Running the jobs file

I look for the parallel command using which to inspect my $PATH for any program with this name. If you have installed parallel on your system but this fails, it's probably because parallel exists in some location that is not in your $PATH. You can fix this by putting parallel into one of the directories in your $PATH or by appending the directory location into your existing $PATH. You could modify this script like so:

```
PATH=/directory/with/parallel:$PATH
```

Be sure it's the *directory* where parallel lives, not the path to the parallel program itself.

The -z test checks if a string is null which will be the case if parallel is not found. In that case, we execute all the commands with sh; otherwise, we run parallel with some number of cores, noting that any failures should halt the process.

Cleaning up

Finally we remove (rm) our temporary files and say good-bye to the user. I always echo "Done." at the end of my bash programs just so I can see that I made it to the end of the program.

Summary

This program shows you how to find input files, create an output directory, use temporary files, process input files with some program, and run those processes either serially or in parallel. This program weighs in at just over 60 lines, which is about the maximum number I feel comfortable writing in bash. It's a capable program, but if I wanted it to do much more, I'd be more comfortable writing it in Python.

Solution

```
#!/usr/bin/env bash
# Convert BAM files to FASTA

set -u

# Check number of arguments
if [[ $# -lt 1 ]] || [[ $# -gt 3 ]]; then
    echo "Usage: $(basename "$0") IN_DIR OUT_DIR [CORES]"
    exit 1
fi

# Assign arguments into named variable
IN_DIR=$1
OUT_DIR=$2
CORES=${3:-40} # default

# Check input directory
if [[ ! -d "$IN_DIR" ]]; then
    echo "Bad IN_DIR \"$IN_DIR\""
    exit 1
fi

# Make output directory if necessary
[[ ! -d "$OUT_DIR" ]] && mkdir -p "$OUT_DIR"

# Find, check input files
FILES=$(mktemp)
find "$IN_DIR" -name \*.bam -size +0c > "$FILES"
NUM=$(wc -l "$FILES" | awk '{print $1}')
if [[ $NUM -lt 1 ]]; then
    echo "No BAM files in IN_DIR \"$IN_DIR\""
    exit 1
fi

# Iterate BAM file in input directory, create samtools command
JOBS=$(mktemp)
i=0
while read -r BAM; do
    i=$((i+1))
    BASE=$(basename "$BAM" ".bam")
    printf "%3d: %s\\n" $i "$BASE"

    # Only process if FASTA does not exist
    FASTA="$OUT_DIR/$BASE.fa"
    if [[ ! -f "$FASTA" ]]; then
        echo "samtools fasta \"$BAM\" > \"$FASTA\"" >> "$JOBS"
    fi
done < "$FILES"

# Look for parallel
PARALLEL=$(which parallel)
if [[ -z "$PARALLEL" ]]; then
    echo "Running serially, install GNU parallel for speed!"
    sh "$JOBS"
else
    echo "Running with $CORES cores in parallel"
    parallel -j "$CORES" --halt soon,fail=1 < "$JOBS"
fi

# Remove temp files, exit
rm "$FILES"
rm "$JOBS"
echo "Done."
```

Chapter 5: Bash: FASTQ-to-FASTA Converter (fq2fa)

Given a list of FASTQ files or directories containing FASTQ files, convert them to FASTA using `parallel`.

Solution

```
#!/bin/bash

# Convert FASTQ files to FASTA
# Author: Ken Youens-Clark <kyclark@gmail.com>

set -u

INPUT=""
OUT_DIR=""
NO_CLOBBER=0

function USAGE() {
    printf "Usage:\\n  %s -i INPUT -o OUTDIR\\n\\n" "$(basename "$0")"

    echo "Required arguments:"
    echo " -i INPUT (DIR/FILE[s])"
    echo " -o OUTDIR (DIR/FILE[s])"
    echo
    exit "${1:-0}"
}

[[ $# -eq 0 ]] && USAGE 1

while getopts :i:o:nh OPT; do
    case $OPT in
        i)
            INPUT="$OPTARG"
            ;;
        h)
            USAGE
            ;;
        n)
            NO_CLOBBER=1
            ;;
        o)
            OUT_DIR="$OPTARG"
            ;;
        :)
            echo "Error: Option -$OPTARG requires an argument."
            exit 1
            ;;
        \?)
            echo "Error: Invalid option: -${OPTARG:-""}"
            exit 1
    esac
done

[[ -z "$INPUT" ]] && USAGE 1
[[ ! -d "$OUT_DIR" ]] && mkdir -p "$OUT_DIR"

INPUT_FILES=$(mktemp)
[[ -f "$INPUT" ]] && echo "$INPUT" > "$INPUT_FILES"
[[ -d "$INPUT" ]] && find "$INPUT" -type f > "$INPUT_FILES"

NUM_INPUT=$(wc -l "$INPUT_FILES" | awk '{print $1}')
if [[ $NUM_INPUT -lt 1 ]]; then
    echo "No input"
    exit 1
fi

JOBS=$(mktemp)

i=0
while read -r FILE; do
    i=$((i+1))
    BASENAME=$(basename "$FILE")
    BASENAME=${BASENAME%%.*}

    printf "%3d: %s\\n" $i "$BASENAME"

    OUT_FILE="$OUT_DIR/$BASENAME.fa"
    if [[ -s "$OUT_FILE" ]] && [[ $NO_CLOBBER -gt 0 ]]; then
        echo "OUT_FILE \"$OUT_FILE\" already exists"
        continue
    fi

    echo "fq2fa.awk \"$FILE\" > \"$OUT_FILE\"" >> "$JOBS"
done < "$INPUT_FILES"

PARALLEL=$(which parallel)

if [[ -z "$PARALLEL" ]]; then
    echo "Running serially, install GNU parallel for speed!"
    sh "$JOBS"
else
    echo "Running with $CORES cores in parallel"
    parallel -j "$CORES" --halt soon,fail=1 < "$JOBS"
fi

```

```
rm "$JOBS"

echo "Done."
```

Chapter 6: Using a Makefile to Create Reproducible Workflows

GNU `make` is a program we can abuse to help create documented, reproducible workflows. It's intended purpose is to create executable files from source code for languages like `c` or `c++`. This process of turning text into machine instructions is called "compiling" and is often a long and tedious process. If a source code file has not changed since the last time the program was compile, `make` will not bother compiling it again. The compiler needs to compile some files before others and then go through a complicated graph of actions to make the executable. This is a workflow, and we can create our own `Makefile` that runs shell commands rather than compiling programs. It's not how `make` was intended to be used, but it works and you'd be surprised at just how far you can go with `make` before you need to investigate more complicated solutions like `snakemake` (which is `make` mixed with Python), Pegasus, Taverna, and the more than 100 other workflow management systems.

If you type `make` on the command line, it will look for a file called `Makefile` (or `makefile`) for instructions. If you look at the `Makefile.orig`, you will see that all the targets for this have been defined.

```
$ head Makefile.orig
.PHONY: all fasta features test clean

all: clean fasta genome chr-count chr-size features gene-count verified-genes uncharacterize

clean:
    find . \( -name \*gene\* -o -name chr-\* \) -exec rm {} \;

fasta:
    echo "Download files into \"fasta\" directory"
```

Make Targets

A "target" in a Makefile is a word starting a line followed by a colon `:` and possibly a number of commands which are all indented by a *tab* character (spaces are not allowed). If you wanted to run the `fasta` target in the file above, you'd type `make fasta` and the `echo` command would be run.

If you find yourself running the same commands over and over, especially if you are scrolling up through your command history to find various encantations, you should consider creating a `Makefile` with targets, e.g., for each of the various data sets you are running.

Automating Yeast Analysis

To start this exercise, copy this to start your `Makefile`:

```
$ cp Makefile.orig Makefile
$ git add Makefile
```

Your job is to figure out the correct Unix commands (or scripts) to create the correct content.

Add your Makefile and any other needed files (e.g., scripts) to your Git repo. DO NOT ADD ANYTHING ELSE (e.g., the FASTA files)!!!

You may notice that there is a `.gitignore` file in there that lists files that Git should … well, ignore. This is a great way to ensure you do not accidentally add files to Git that should not be there!

```
$ cat Makefile
.PHONY: all fasta features test clean

all: clean fasta genome chr-count chr-size features gene-count verified-genes uncharacterize

clean:
    find . \( -name \*gene\* -o -name chr-\* \) -exec rm {} \;

fasta:
    echo "Download files into \"fasta\" directory"

genome: fasta
    echo OK > fasta/genome.fa

chr-count: genome
    echo OK > chr-count

chr-size: genome
    echo OK > chr-size

features:
    echo "Download SGD_features.tab"

gene-count: features
    echo OK > gene-count

verified-genes: features
    echo OK > verified-genes

uncharacterized-genes: features
    echo OK > uncharacterized-genes
```

```
gene-types: features
    echo OK > gene-types

palinsreg:
    echo "Unzipping palinsreg"

terminated-genes: palinsreg
    echo OK > terminated-genes

test:
    pytest -v test.py
```

Targets

'fasta' target:

Download all the '.fsa' files (chr 1-16, mt) from http://downloads.yeastgenome.org/sequence/S288C_reference/c
a 'fasta' directory.

HINT: You can right-click on the links to copy the link location and then 'wget' the file.

"genome" target:

Make a single whole genome file called `fasta/genome.fa`

"chr-count" target:

Count the chromosomes in the whole genome file. Put the number into a file called `chr-count`.

HINT: Each of the original FASTA files contains a single chromosome.

"chr-size" target:

Find size of total genome. Put the answer into a file called `chr-size`.

HINT: Look up the command `wc` and find out what it does. The size of the genome can be determined by counting the number of characters in the genome (not on the same line as a fasta header).

“features” target:

Download the list of cerevisiae chromosome features: http://downloads.yeastgenome.org/curation/chromosoma

Columns:

- Primary Standfor Gene Database ID (SGDID) (mandatory)
- Feature type (mandatory)
- Feature qualifier (optional)
- Feature name (optional)
- Standard gene name (optional)
- Alias (optional, multiples separated by |)
- Parent feature name (optional)
- Secondary SGDID (optional, multiples separated by |)
- Chromosome (optional)1
- Start_coordinate (optional)1
- Stop_coordinate (optional)1
- Strand (optional)1
- Genetic position (optional)
- Coordinate version (optional)
- Sequence version (optional)
- Description (optional)

‘gene-count’ target:

Count total genes (’ORF’s) from `SGD_features.tab` into a file called `gene-count`.

‘verified-genes’ target:

Count only verified genes from `SGD_features.tab` into a file called `verified-genes`.

‘uncharacterized-genes’ target:

Count only uncharacterized genesfrom `SGD_features.tab` into a file called `uncharacterized-genes`.

‘gene-types’ target:

Create file called `gene-types` that contains the counts of all the types of genes.

‘palinsreg.txt’

The file `palinsreg.txt` has been provided for you in a zipped format. Unzip it.

These are detected terminator sequences in the E. coli genome (using the program GeSTer, if you’re curious). The command grep ’/G=[^]*’ somefile will find all lines that match /G=somegenename, where somegenename is a sequence of non-blank characters. Read the output of man grep and figure out how to -only print /G=somegenename, rather than the whole line. Pipe the results of part (2) through a cut command to get only everything after the ‘=’ Store the unique, sorted results of part (3) into a file named ‘terminated-genes’

Solution

```
.PHONY: all fasta features test clean

all: clean fasta genome chr-count chr-size features gene-count verified-genes uncha

clean:
    find . \( -name \*gene\* -o -name chr-\* \) -exec rm {} \;

fasta:
    ./download.sh

genome: fasta
    (cd fasta && cat *.fsa > genome.fa)

chr-count: genome
    grep -e '^>' "fasta/genome.fa" | grep 'chromosome' | wc -l > chr-count

chr-size: genome
    grep -ve '^>' "fasta/genome.fa" | wc -c > chr-size

features:
    wget -nc http://downloads.yeastgenome.org/curation/chromosomal_feature/SGD_featu

gene-count: features
    cut -f 2 SGD_features.tab | grep ORF | wc -l > gene-count

verified-genes: features
    awk -F"\t" '$$3 == "Verified" {print}' SGD_features.tab | wc -l > verified-gene
#awk -F"\t" '$$2 == "ORF" && $$3 == "Verified" {print $$2}' SGD_features.tab | wc -

uncharacterized-genes: features
    awk -F"\t" '$$2 == "ORF" && $$3 == "Uncharacterized" {print $$2}' SGD_features.t

gene-types: features
    awk -F"\t" '{print $$3}' SGD_features.tab | sort | uniq -c > gene-types
    #cut -f 3 SGD_features.tab | sort | uniq -c > gene-types

palinsreg:
    unzip palinsreg.txt.gz

terminated-genes: palinsreg
    grep -o '/G=[^ ]*' palinsreg.txt | cut -d = -f 2 | sort -u > terminated-genes

test:
```

```
44      ./test.pl6
```

Chapter 7: Git Basics

Source Code Management

Basic Commands: clone, add, commit, push

Git vs GitHub, GitLab

Collaborators

Using Git to Copy Code

What Does Not Go In

SSH Keys

Standard Workflow

- Create a new repo in GitHub
- `git clone <repo>` on your machine(s)
- `git add` new files
- `git commit|push` early and often!

Forking

Chapter 8: Programming with 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 the "shebang" on line 1. I'm going to use `env` to find the first `python3` binary in the user's `$PATH`. 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.

Variables

It's not so interesting to just say "Hello, World!" all the time. Let's make a program that will say "hello" to some value that we pass in. This value can change each time we run the program, so it's common to call this a "variable."

Let's use the REPL (Read-Evaluate-Print-Loop, pronounced "reh-pul") to play with variables. Type `python` (or `python3` or `ipython`) to get into a REPL:

```
>>> name = 'Duderino'
>>> print('Hello,', name)
Hello, Duderino
```

Here I'm showing that we can create variable called `name` by assigning it some value like "Duderino." Unlike `bash`, we don't have to worry about spaces around the `=`. You can put any number of spaces around the equal sign, but it's most common (and readable) to put just one on each side. Notice that `print` will accept more than one argument and will put spaces between the arguments. You can tell it to use some other "separator" by indicating the `sep` keyword argument. Notice the Pythonic style is that there are *no* spaces around the `=` for keywords:

```
>>> print('Hello', name, sep=', ')
Hello, Duderino
```

It's not easy to tell, but `print` is also putting a newline on the end. We can change that with the `end` keyword argument:

```
>>> print('Hello', name, sep=', ', end='!')
Hello, Duderino!>>>
```

Unlike in `bash`, we cannot use a variable directly in a `print` statement or we get the equivalent of George Burns telling Gracie "Say 'Good night,' Gracie" and she says "Good night, Gracie!":

```
>>> print('Hello, name')
Hello, name
```

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

```
>>> print('Hello, ' + name)
Hello, Duderino
```

Types: Strings and Numbers

As you might expect, the "plus" operator `+` is also used to perform numeric addition:

```
>>> n = 10
>>> n + 1
11
```

The `name` variable above is of the type `str` (string) because we put the value in quotes (single or double, it doesn't matter).

```
>>> name = 'The Dude'
>>> type(name)
<class 'str'>
```

Numbers don't have quotes. Number can be integers (`int`) or floating-point numbers (`float`) if they have a decimal somewhere or you write them in scientific notation:

```
>>> type(10)
<class 'int'>
>>> type(3.14)
<class 'float'>
>>> type(1.)
<class 'float'>
>>> type(2.864e-10)
<class 'float'>
```

The "plus" operator behaves completely differently with different *types* of arguments as long as the arguments are both strings or both numbers. Things go wobbly when you mix them:

```
>>> 'Hello, ' + 'Mr. Lebowski'
'Hello, Mr. Lebowski'
```

```
>>> 1 + 2
3
>>> 1 + 'Mr. Lebowski'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Lists

Before we go further, I will introduce a different variable type called a "list" as we are going to need that immediately. You create a list by putting values in `[]` (square brackets or just "brackets"):

```
>>> vals = ['foo', 'bar', 'baz']
>>> vals
['foo', 'bar', 'baz']
```

You can get the length of a list using the `len` function:

```
>>> len(vals)
3
```

But note that, like so many other languages, Python starts counting at `0`, so the first element is in the "zeroth" position. The value at position `1` is actually the *second* value.

```
>>> vals[0]
'foo'
>>> vals[1]
'bar'
```

We'll talk much more about lists in the next chapter. I needed to tell you that so I could tell you this next bit.

Command-line Arguments: sys.argv is a list

Now let's get our "hello" program to greet an argument passed from the command line. We discussed in the `bash` section that programs can take *positional* arguments, e.g. `ls` can accept the name of the directory you wish to list or `wc` can take the name of a file to count. *Positional* arguments mean the first argument, the second argument, and so on. In the command `ls ~`, the `~` (tilde which means `$HOME` in `bash`) is the one and only positional argument. In the command `ls /bin /usr/bin/`, there are two positional arguments, `/bin` and `/usr/bin/`.

Named options have some sort of prefix, e.g., `find` can take a `-maxdepth` argument to indicate how many levels deep to search. Lastly, commands may also take flags like the `-l` flags to `ls` that indicates you wish to see the "long" listing.

To get access to the positional arguments to our program, we need to `import sys` which is a package of code that will interact with the system. Those arguments will be a *list*:

```
$ 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 the first argument (in the zeroth position) – `args[0]`. That means the first *actual* "argument" to the script is in `args[1]`.

```
$ ./hello_arg.py Geddy
Hello, Geddy!
```

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. Before I show you how to do that, let me explain something about *slicing* lists. Inside the `[]`, you can indicate a start and stop positon like so:

```
>>> vals
['foo', 'bar', 'baz']
>>> vals[1:]
['bar', 'baz']
>>> vals[1:2]
['bar']
```

In the last example, you see that the stop position is not inclusive. Even though you've seen that Python gets very upset by asking for a position in a list that does not exist, it has no problem giving you nothing when you ask for a *slice* that doesn't exist:

```
>>> vals[1000:]
[]
```

So we can use that to ask for `sys.argv[1:]` to get all the *actual* arguments to our program, skipping over the name of the program itself:

```
$ cat -n hello_arg2.py
     1	#!/usr/bin/env python3
     2
     3	import sys
     4
     5	args = sys.argv[1:]
     6
     7	if len(args) < 1:
     8	    print('Usage:', sys.argv[0], 'NAME')
     9	    sys.exit(1)
    10
    11	name = args[0]
    12	print('Hello, ' + name + '!')
```

If there are fewer than 1 argument, 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 Alex
Hello, Alex!
```

Here is the same functionality but using some new functions, `str.format` so we can introduce a different way to join strings, and `os.path.basename` so we can get the name of the program without any leading path information like `./`:

```
$ cat -n hello_arg3.py
     1	#!/usr/bin/env python3
     2
     3	import sys
     4	import os
     5
     6	args = sys.argv[1:]
     7
     8	if len(args) != 1:
     9	    script = os.path.basename(sys.argv[0])
    10	    print('Usage: {} NAME'.format(script))
    11	    sys.exit(1)
    12
    13	name = args[0]
    14	print('Hello, {}!'.format(name))
$ ./hello_arg3.py
Usage: hello_arg3.py NAME
$ ./hello_arg3.py Neil
```

```
Hello, Neil!
```

The main() thing

Many languages (e.g., Perl, Rust, Haskell) have the idea of a `main` module/function where all the processing starts. If you define a `main` function using `def main`, most people reading your code would understand that the program *ought* to begin there. (I say "ought" because Python won't actually make that happen. You still have to *call* the `main` function to make your program run!) I usually put my `main` first and then call it at the end of the script with this `__name__ == '__main__'` business. This looks a bit of a hack, but it is fairly Pythonic.

```
$ cat -n hello_arg4.py
     1	#!/usr/bin/env python3
     2	
     3	import sys
     4	import os
     5	
     6	def main():
     7	    args = sys.argv[1:]
     8	
     9	    if len(args) != 1:
    10	        script = os.path.basename(sys.argv[0])
    11	        print('Usage: {} NAME'.format(script))
    12	        sys.exit(1)
    13	
    14	    name = args[0]
    15	    print('Hello, {}!'.format(name))
    16	
    17	
    18	if __name__ == '__main__':
    19	    main()
$ ./hello_arg4.py
Usage: hello_arg4.py NAME
$ ./hello_arg4.py '2013 Rock and Roll Hall of Fame Inductees'
Hello, 2013 Rock and Roll Hall of Fame Inductees!
```

Function Order

Note that you cannot put call to `main()` before `def main` 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 greet to any number of arguments:

```
$ cat -n hello_arg5.py
     1  #!/usr/bin/env python3
     2
     3  import sys
     4  import os
     5
     6  def main():
     7      names = sys.argv[1:]
     8
     9      if len(names) < 1:
    10          script = os.path.basename(sys.argv[0])
    11          print('Usage: {} NAME [NAME2 ...]'.format(script))
    12          sys.exit(1)
    13
    14      print('Hello, {}!'.format(', '.join(names)))
    15
    16  if __name__ == '__main__':
    17      main()
$ ./hello_arg5.py
Usage: hello_arg5.py NAME [NAME2 ...]
$ ./hello_arg5.py Geddy Alex Neil
Hello, Geddy, Alex, Neil!
```

Notice on line 14 to see how we can `join` all the arguments on a comma + space.

Conditionals

So far we've been using an `if` condition to see if we have enough arguments. If you want to test for more than one condition, you can use `elif` (else if) and `else` ("otherwise" or the "default" branch if all others fail). Here we'll use the `input` function to present the user with a prompt and get their input:

```
$ cat -n if-else.py
     1  #!/usr/bin/env python3
     2
     3  name = input('What is your name? ')
     4  age = int(input('Hi, ' + name + '. What is your age? '))
     5
     6  if age < 0:
     7      print("That isn't possible.")
     8  elif age < 18:
     9      print('You are a minor.')
    10  else:
    11      print('You are an adult.')
$ ./if-else.py
What is your name? Ken
Hi, Ken. What is your age? -4
That isn't possible.
$ ./if-else.py
What is your name? Lincoln
Hi, Lincoln. What is your age? 29
You are an adult.
```

On line 3, we can put the first answer directly into the `name` variable; however, on line 4, I need to 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
```

Things go very badly if we blindly try to coerce a string into an `int`:

```
$ ./if-else.py
What is your name? Doreen
Hi, Doreen. What is your age? Ageless
Traceback (most recent call last):
  File "./if-else.py", line 4, in <module>
    age = int(input('Hi, ' + name + '. What is your age? '))
ValueError: invalid literal for int() with base 10: 'Ageless'
```

Later we'll talk about how to avoid problems like this.

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	
     3	import sys
     4	import os
     5	
     6	def main():
     7	    names = sys.argv[1:]
     8	
     9	    if len(names) < 1:
    10	        prg = os.path.basename(sys.argv[0])
    11	        print('Usage: {} NAME [NAME2 ...]'.format(prg))
    12	        sys.exit(1)
    13	
    14	    for name in names:
    15	        print('Hello, ' + name + '!')
    16	
    17	
```

```
if __name__ == '__main__':
    main()
$ ./hello_arg6.py
Usage: hello_arg6.py NAME [NAME2 ...]
$ ./hello_arg6.py Salt Peppa
Hello, Salt!
Hello, Peppa!
```

You can use a `for` loop on anything that is like a list! A string is a list of characters:

```
>>> for letter in "abc":
...    print(letter)
...
a
b
c
```

The `range` function returns something that can be "iterated" like a list:

```
>>> for number in range(0, 5):
...    print(number)
...
0
1
2
3
4
```

Lists, of course:

```
>>> for word in ['foo', 'bar']:
...    print(word)
...
foo
bar
```

You can use the `str.split` function to split a string (the default is to split on spaces):

```
>>> for word in 'We hold these truths'.split():
...    print(word)
...
We
hold
these
truths
```

And we can use the `open` function to open a file and read each line using a `for` loop:

```
>>> for line in open('input1.txt'):
...     print(line, end='')
...
this is
some text
from a file.
```

The 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 with new.py

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

- Shebang
- import modules
- define main()
- call main()

Additionally we keep having to write the same few lines of code to get the arguments from `sys.argv[1:]` and then test that we have the right number and then print a usage and `sys.exit(1)`. Rather than type all that boilerplate or copy-paste from other programs, let's use a program to help us create new programs.

Included in the `bin` directory of the GitHub repo, there is a program called `new.py` that will stub out all this code for you. Make sure you either add that directory to you `$PATH` or copy that program into your existing `$PATH`, e.g., I like to have `$HOME/.local/bin` for programs like this:

```
$ which new.py
/Users/kyclark/.local/bin/new_py.py
```

Now run it with no arguments. As you might expect, it gives you a usage statement:

```
$ new.py
usage: new.py [-h] [-s] [-f] program
new.py: error: the following arguments are required: program
```

Give it the name of a new program (either with or without `.py`):

```
$ new.py foo
Done, see new script "foo.py."
$ cat -n foo.py
     1	#!/usr/bin/env python3
     2	"""
     3	Author : kyclark
```

```
Date   : 2019-06-13
Purpose: Rock the Casbah
"""

import argparse
import os
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 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()
```

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.

In my experience, perhaps 20-50% of the effort to solve most of the exercises can be handled by using `argparse` well. You can specify an exact number of positional arguments, you can specify named arguments that must be constrained to a list of choices, you can force one argument to be an `int` and another to be a `float`, you can get Boolean flags or ensure that arguments are existing files that can be opened and read. I urge you to read the documentation for `argparse` thoroughly. I often find the REPL is quite useful for this:

```
>>> import argparse
>>> help(argparse)
```

Chapter 9: Greeter: Positional Command-line Arguments

Write a Python program named `hello.py` that warmly greets the names you provide. When there are two names, join them with "and." When there are three or more, join them on commas (INCLUDING THE OXFORD WE ARE NOT SAVAGES) and "and." If no names are supplied, print a usage.

```
$ ./hello.py
Usage: hello.py NAME [NAME...]
$ ./hello.py Alice
Hello to the 1 of you: Alice!
$ ./hello.py Mike Carol
Hello to the 2 of you: Mike and Carol!
$ ./hello.py Greg Peter Bobby Marcia Jane Cindy
Hello to the 6 of you: Greg, Peter, Bobby, Marcia, Jane, and Cindy!
```

Solution

```
#!/usr/bin/env python3
"""
Author : Ken Youens-Clark <kyclark@gmail.com>
Date   : 2019-05-14
Purpose: Greet the arguments
"""

import os
import sys


# --------------------------------------------------
def main():
    """main"""
    names = sys.argv[1:]
    num = len(names)

    if num < 1:
        print('Usage: {} NAME [NAME...]'.format(os.path.basename(sys.argv[0])))
        sys.exit(1)

    phrase = ''
    if num == 1:
        phrase = names[0]
    elif num == 2:
        phrase = '{} and {}'.format(names[0], names[1])
    else:
        last = names.pop()
        phrase = '{}, and {}'.format(', '.join(names), last)

    print('Hello to the {} of you: {}!'.format(num, phrase))


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

Chapter 10: Hello: Named Command-line Options

Write a Python program called `hello.py` that accepts three named arguments, `-g|--greeting` which is the greeting, `-n|--name` which is the name, and `-e|--excited` which is a flag to indicate whether to use a "!" in the output `<greeting>, <name><punctuation>`.

```
$ ./hello.py -h
usage: hello.py [-h] [-g str] [-n str] [-e]

Greetings and saluatations

optional arguments:
  -h, --help            show this help message and exit
  -g str, --greeting str
                        The greeting (default: Hello)
  -n str, --name str    The name (default: World)
  -e, --excited         Whether to use an "!" (default: False)
$ ./hello.py
Hello, World.
$ ./hello.py -g Howdy
Howdy, World.
$ ./hello.py -n Stranger
Hello, Stranger.
$ ./hello.py --name Pig --greeting "That'll do"
That'll do, Pig.
$ ./hello.py -n Gracie -g 'Good Night' -e
Good Night, Gracie!
```

Solution

```
#!/usr/bin/env python3
"""
Author : kyclark
Date   : 2019-05-16
Purpose: Greetings and saluatations
"""

import argparse
import sys


# --------------------------------------------------
def get_args():
    """get command-line arguments"""
    parser = argparse.ArgumentParser(
        description='Greetings and saluatations',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('-g',
                        '--greeting',
                        help='The greeting',
                        metavar='str',
                        type=str,
                        default='Hello')

    parser.add_argument('-n',
                        '--name',
                        help='The name',
                        metavar='str',
                        type=str,
                        default='World')

    parser.add_argument('-e',
                        '--excited',
                        help='Whether to use an "!"',
                        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()
    print('{}, {}{}'.format(args.greeting, args.name,
                            '!' if args.excited else '.'))


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

Chapter 11: File Handling: Emulate "head"

Create a Python program called `head.py` that expects one or two arguments. If there are no arguments, print a "Usage" statement. The first argument is required and much be a regular file; if it is not, print " is not a file" and exit *with an error code.* The second argument is optional. If given, it must be a positive number (non-zero); if it is not, then print "lines () must be a positive number". If no argument is provided, use a default value of 3. You can expect that the test will only give you a value that can be safely converted to a number using the `int` function. If given good input, it should act like the normal `head` utility and print the expected number of lines from the given file.

```
$ ./head.py
Usage: head.py FILE [NUM_LINES]
$ ./head.py foo
foo is not a file
$ ./head.py files/issa.txt
Selected Haiku by Issa

Don't worry, spiders,
$ ./head.py files/issa.txt 5
Selected Haiku by Issa

Don't worry, spiders,
I keep house
casually.
```

Solution

```
#!/usr/bin/env python3
"""
Author : Ken Youens-Clark <kyclark@gmail.com>
Date   : 2019-02-04
Purpose: Emulate head
"""

import os
import sys


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

    if len(args) < 1 or len(args) > 2:
        print('Usage: {} FILE [NUM_LINES]'.format(os.path.basename(sys.argv[0])))
        sys.exit(1)

    filename = args[0]
    num_lines = int(args[1]) if len(args) == 2 else 3

    if num_lines < 1:
        print('lines ({}) must be a positive number'.format(num_lines))
        sys.exit(1)

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

    for i, line in enumerate(open(filename)):
        print(line, end='')
        if i + 1 == num_lines:
            break


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

Chapter 12: File Handling: Emulate "cat -n"

Create a Python program called `cat_n.py` that expects exactly one argument which is a regular file and prints usage statement if either condition fails. It should print each line of the file argument preceeded by the line number which is right-justified in spaces and a colon. You may the format '{:5}: {}' to make it look exactly like the output below, but the test is just checking for a leading space, some number(s), a colon, and the line of text.

```
$ ./cat_n.py
Usage: cat_n.py FILE
$ ./cat_n.py foo
foo is not a file
$ ./cat_n.py files/sonnet-29.txt
    1: Sonnet 29
    2: William Shakespeare
    3:
    4: When, in disgrace with fortune and men’s eyes,
    5: I all alone beweep my outcast state,
    6: And trouble deaf heaven with my bootless cries,
    7: And look upon myself and curse my fate,
    8: Wishing me like to one more rich in hope,
    9: Featured like him, like him with friends possessed,
   10: Desiring this man’s art and that man’s scope,
   11: With what I most enjoy contented least;
   12: Yet in these thoughts myself almost despising,
   13: Haply I think on thee, and then my state,
   14: (Like to the lark at break of day arising
   15: From sullen earth) sings hymns at heaven’s gate;
   16: For thy sweet love remembered such wealth brings
   17: That then I scorn to change my state with kings.
```

Solution

```
#!/usr/bin/env python3
"""
Author : Ken Youens-Clark <kyclark@gmail.com>
Date   : 2019-02-04
Purpose: Emulate cat-n
"""

import os
import sys


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

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

    file = args[0]

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

    for i, line in enumerate(open(file), start=1):
        print('{:5}: {}'.format(i, line), end='')


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

Chapter 13: Emulating Word Count (wc) in Python

Write your own implementation in Python of the `wc` program where you print lines, words, and characters contained in a file.

Solution

```
#!/usr/bin/env python3
"""Emulate wc"""

import argparse


# --------------------------------------------------
def get_args():
    """Get command-line arguments"""

    parser = argparse.ArgumentParser(
        description='Emulate wc',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('file',
                        metavar='FILE',
                        type=argparse.FileType('r'),
                        nargs='+',
                        help='Input file')

    return parser.parse_args()


# --------------------------------------------------
def main():
    """Make a jazz noise here"""

    args = get_args()
    for fh in args.file:
        chars, words, lines = 0, 0, 0
        for line in fh:
            lines += 1
            chars += len(line)
            words += len(line.split())

        print('{:>8}{:>8}{:>8} {}'.format(lines, words, chars, fh.name))


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

Discussion

The program needs to take a list of files, so we use the `nargs='+'` to indicate one or more and `type=argparse.FileType('r')` to say they must be "readable" (`'r'`) files. We can use a `for fh in args.file` to iterate over the file handles (hence the name `fh`). We want to initialize counters for `chars`, `words`, and `lines` with the value `0` which we can do with a shorthand unpacking of the tuple `(0, 0, 0)` (parentheses not strictly necessary) on line 30. We can then iterate each line in the open file handle with `for line in fh` and do:

1. Increment `lines` by `1`
2. Increment `chars` by the length of the `line` (number of characters)
3. Increment `words` by the length of the list created by splitting the `line` on spaces

Finally we need to print output similar to the actual `wc` program which appears to right-justify each of the numbers for lines, words, and characters in a column 8-characters wide followed by a space and then the name of the file. The call `'{:8}'.format()` will format a string into 8 characters, but they will be left-justified:

```
>>> '{:8}'.format('hello')
'hello   '
```

We can add `>` to right-justify. (Think of it like an arrow pointing to the right where you want the text.)

```
>>> '{:>8}'.format('hello')
'   hello'
```

Chapter 14: Finding GC Content in Sequences

Write a Python program called `gc.py` that takes a single positional argument which should be a file. Die with a warning if the argument is not a file. For each line in the file, print the line number and the percentage of the characters on that line that are a “G” or “C” (case-insensitive).

```
$ ./gc.py
usage: gc.py [-h] FILE
gc.py: error: the following arguments are required: FILE
$ ./gc.py foo
"foo" is not a file
$ ./gc.py samples/sample1.txt
  1:   9%
  2:  19%
  3:  19%
  4:  22%
  5:  32%
  6:  21%
```

Solution

```
#!/usr/bin/env python3
"""
Author : Ken Youens-Clark <kyclark@gmail.com>
Date   : 2019-02-19
Purpose: Calculate GC content
"""

import argparse
import os
import sys


# --------------------------------------------------
def get_args():
    """get command-line arguments"""
    parser = argparse.ArgumentParser(
        description='Calculate GC content',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

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

    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()
    file = args.file

    if not os.path.isfile(file):
        die('"{}" is not a file'.format(file))

    for i, line in enumerate(open(file), start=1):
        # Method 1
        gc = 0
        for char in line.lower():
            if char == 'g' or char == 'c':
                gc += 1

        # Method 2
        line = line.lower()
        gc = line.count('g') + line.count('c')

        pct = int((gc / len(line)) * 100)
        print('{:3}: {:3}%'.format(i, pct))


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

Chapter 15: FASTA format: Text to FASTA (txt2fa)

Write a Python program called `txt2fa.py` that turns lines of sequences into FASTA formatted output.

Solution

```
#!/usr/bin/env python3
"""txt2fa"""

import argparse
import os
import sys


# --------------------------------------------------
def get_args():
    """Get command-line arguments"""

    parser = argparse.ArgumentParser(
        description='Text to FASTA',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('file',
                        metavar='FILE',
                        nargs='+',
                        type=argparse.FileType('r'),
                        help='Input file(s)')

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

    return parser.parse_args()


# --------------------------------------------------
def main():
    """Make a jazz noise here"""

    args = get_args()
    out_dir = args.outdir

    if not os.path.isdir(out_dir):
        os.makedirs(out_dir)

    for fnum, fh in enumerate(args.file, start=1):
        basename = os.path.basename(fh.name)
        print('{:3}: {}'.format(fnum, basename))
        out_file = os.path.join(out_dir, basename)
        out_fh = open(out_file, 'wt')
        for i, line in enumerate(fh, start=1):
            out_fh.write('>{}\n{}'.format(i, line))
        out_fh.close()
        num = fnum

    print('Done, processed {} file{}.'.format(fnum, '' if fnum == 1 else 's'))

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

Chapter 16: Transcribe DNA to RNA

RNA on Rosalind.

Solution

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

import sys

def main(file):
    f = open(file, 'r')
    rna = ''.join(map(lambda s: s.rstrip(), f.read())).replace('T', 'U')
    print(rna);

if __name__ == "__main__":
    main(sys.argv[1])
```

Chapter 17: Calculating Tetranucleotide Frequency

The `DNA` problem from Rosalind.

Solution

```
#!/usr/bin/env python3
"""Tetra-nucleotide counter"""

import sys
import os

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

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

    dna = args[0]

    num_a, num_c, num_g, num_t = 0, 0, 0, 0

    for base in dna.lower():
        if base == 'a':
            num_a += 1
        elif base == 'c':
            num_c += 1
        elif base == 'g':
            num_g += 1
        elif base == 't':
            num_t += 1

    print('{} {} {} {}'.format(num_a, num_c, num_g, num_t))

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

Chapter 18: List Products to Find Recombinations

Jumble promoter/coding/terminators.

Solution

```
#!/usr/bin/env python3
"""Show recominations"""

import os
import sys
from itertools import product


def die(msg):
    """print and exit with an error"""
    print(msg)
    sys.exit(1)


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

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

    if not args[0].isdigit():
        die('"{}" does not look like an integer'.format(args[0]))

    num_genes = int(args[0])
    if not 2 <= num_genes <= 10:
        die('NUM_GENES must be greater than 1, less than 10')

    def gen(prefix):
        return [prefix + str(n) for n in range(1, num_genes + 1)]

    print('N = "{}"'.format(num_genes))
    combos = product(gen('P'), gen('C'), gen('T'))
    for i, combo in enumerate(combos, start=1):
        print('{:4}: {}'.format(i, ' - '.join(combo)))


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

Chapter 19: Strings: Run-Length Encoding of DNA

Information content, compression, strings.

Solution

```
#!/usr/bin/env python3
"""Compress text/DNA by marking repeated letters"""

import os
import sys


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

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

    # If the argument is a file, the text should be the file contents
    arg = args[0]
    text = ''
    if os.path.isfile(arg):
        text = ''.join(open(arg).read().split())
    else:
        text = arg.strip()

    # Make sure we have something
    if len(text) == 0:
        print('No usable text')
        sys.exit(1)

    counts = []
    count = 0
    prev = None
    for letter in text:
        # We are at the start
        if prev is None:
            prev = letter
            count = 1
        # This letter is the same as before
        elif letter == prev:
            count += 1
        # This is a new letter, so record the count
        # of the previous letter and reset the counter
        else:
            counts.append((prev, count))
            count = 1
            prev = letter

    # get the last letter after we fell out of the loop
    counts.append((prev, count))

    for letter, num in counts:
        print('{}{}'.format(letter, '' if num == 1 else num), end='')

    print()


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

Chapter 20: Strings: Find and Format Sequence Lengths

Change this to process short sequences.

Create a Python program called `column.py` that takes a list of words and creates a columnar output of each word and their length. If given no words as positional, command-line arguments, print a usage statement. For the output, first print a header of “word” and “len”, then lines which are the width of the longest word and the longest numbers with a minimum for each of the column headers themselves. The words should be left-justified in the first column and the numbers should be right-justified in the second column.

```
$ ./column.py
Usage: column.py WORD [WORD...]
$ ./column.py a an the
word  len
----  ---
a       1
an      2
the     3
$ ./column.py `cat out/1.in`
word               len
-----------------  ---
Iphis                5
cyclone              7
dare                 4
umbraculiferous     15
indescribableness   17
prattling            9
pediculine          10
pondwort             8
lava                 4
adipoma              7
```

Solution

```
#!/usr/bin/env python3
"""
Author : Ken Youens-Clark <kyclark@gmail.com>
Date   : 2019-05-03
Purpose: Columnar output
"""

import os
import sys


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

    if len(words) < 1:
        print('Usage: {} WORD [WORD...]'.format(os.path.basename(sys.argv[0])))
        sys.exit(1)

    word_lengths = map(len, words)
    longest_word = max(word_lengths)
    longest_num = len(str(longest_word))

    if longest_word < 4:
        longest_word = 4

    if longest_num < 3:
        longest_num = 3

    fmt = '{:' + str(longest_word + 1) + '}{:>' + str(longest_num + 1) + '}'

    print(fmt.format('word', 'len'))
    print(fmt.format('-' * longest_word, '-' * longest_num))

    for word in words:
        print(fmt.format(word, len(word)))


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

Chapter 21: Strings: Find Conversed Bases in Aligned Sequences

Multiple sequence alignment

Solution

```
#!/usr/bin/env python3
"""
Author : kyclark
Date   : 2019-01-14
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)

    file = args[0]
    seqs = list(filter(lambda s: len(s) > 0, open(file).read().split('\n')))
    lens = set(map(len, seqs))

    if len(lens) > 1:
        print('Not all the same length!')
        sys.exit(1)

    length = list(lens)[0]
    is_conserved = []
    for i in range(length):
        chars = map(lambda s: s[i], seqs)
        is_conserved.append('|' if len(set(chars)) == 1 else 'X')

    print('\n'.join(seqs))
    print(''.join(is_conserved))


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

Chapter 22: Introduction to Python Dictionaries

> Sometimes I feel like my job is deeply meaningful and then I remember that at the end of the day most of what I do is asking students to read error messages from compilers. – Kristopher Micinski

In addition to lists and tuples, 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 `{}` ("curly") 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 lists and tuples:

```
>>> 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']
>>> patch
{'species': 'dog', 'age': 4, 0: 'food', 'likes': ['walking', 'running', 'car trips']}
<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.'
```

If you want to know if a key exists, use `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.

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	
     3	import sys
     4	import os
     5	
     6	def main():
     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	    count = {}
    15	    for base in dna.lower():
    16	        if not base in count:
    17	            count[base] = 0
    18	        count[base] += 1
    19	
    20	    counts = []
    21	    for base in "acgt":
    22	        num = count[base] if base in count else 0
    23	        counts.append(str(num))
    24	
    25	    print(' '.join(counts))
    26	
    27	if __name__ == '__main__':
    28	    main()
$ cat dna.txt
AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAAAAGAGTGTCTGATAGCAGC
$ ./dna3.py `cat dna.txt`
20 12 17 21
```

This has the great advantage of not having to declare four variables to count the four bases. True, we're only checking (in line 21) for those four, but we can now count all the letters in any string.

Notice that we create a new dict on line 14 with empty curlies `{}`. On line 16, 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 22, we have to be careful when asking for a key that doesn't exist. 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:

```
counts = []
for base in "acgt":
    num = count.get(base, 0)
    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'
```

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
#!/usr/bin/env python3
"""Tetra-nucleotide counter"""

import sys
import os
from collections import Counter

args = sys.argv[1:]

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

dna = args[0]

count = Counter(dna.lower())

counts = []
for base in "acgt":
    counts.append(str(count[base]))

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
#!/usr/bin/env python3
"""Character counter"""

import sys
import os
from collections import Counter

args = sys.argv

if len(args) != 2:
    print('Usage: {} INPUT'.format(os.path.basename(args[0])))
    sys.exit(1)

    arg = args[1]
    text = ''
    if os.path.isfile(arg):
        text = ''.join(open(arg).read().splitlines())
    else:
        text = arg

    count = Counter(text.lower())

    for letter, num in count.items():
        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:/

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(
        '-n',
        '--numsort',
        help='Sort by number',
        dest='numsort',
        action='store_true')

    parser.add_argument(
        '-r',
        '--reverse',
        help='Sort in reverse order',
        dest='reverse',
        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()
    input_arg = args.input
    charsort = args.charsort
    numsort = args.numsort
    revsort = args.reverse

    if charsort and numsort:
        die('Please choose one of --charsort or --numsort')

    if not charsort and not numsort:
        charsort = True

    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()
```

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
#!/usr/bin/env python3
"""Count words in common between two files"""

import os
import re
import sys
import string

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

    if len(files) != 2:
        msg = 'Usage: {} FILE1 FILE2'
        print(msg.format(os.path.basename(sys.argv[0])))
        sys.exit(1)

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

    file1, file2 = files[0], files[1]
    words1 = uniq_words(file1)
    words2 = uniq_words(file2)
    common = words1.intersection(words2)
    num_common = len(common)
    msg = 'There {} {} word{} in common between "{}" and "{}."'
    print(msg.format('is' if num_common == 1 else 'are',
                     num_common,
                     '' if num_common == 1 else 's',
                     os.path.basename(file1),
                     os.path.basename(file2)))

    for i, word in enumerate(sorted(common)):
        print('{:3}: {}'.format(i + 1, word))

# --------------------------------------------------
def uniq_words(file):
    regex = re.compile('[' + string.punctuation + ']')
    words = set()
    for line in open(file):
        for word in [regex.sub('', w) for w in line.lower().split()]:
            words.add(word)

    return words

# --------------------------------------------------
if __name__ == '__main__':
    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
#!/usr/bin/env python3
"""Count words/frequencies in two files"""

import os
import re
import sys
import string
from collections import defaultdict

# --------------------------------------------------
def word_counts(file):
    """Return a dictionary of words/counts"""
    words = defaultdict(int)
    regex = re.compile('[' + string.punctuation + ']')
    for line in open(file):
        for word in [regex.sub('', w) for w in line.lower().split()]:
            words[word] += 1

    return words

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

    if len(args) != 2:
        msg = 'Usage: {} FILE1 FILE2'
        print(msg.format(os.path.basename(sys.argv[0])))
        sys.exit(1)

    for file in args[0:2]:
        if not os.path.isfile(file):
            print('"{}" is not a file'.format(file))
            sys.exit(1)

    file1 = args[0]
    file2 = args[1]
    words1 = word_counts(file1)
    words2 = word_counts(file2)
    common = set(words1.keys()).intersection(set(words2.keys()))
    num_common = len(common)
    verb = 'is' if num_common == 1 else 'are'
    plural = '' if num_common == 1 else 's'
    msg = 'There {} {} word{} in common between "{}" ({}) and "{}" ({}).'
    tot1 = sum(words1.values())
    tot2 = sum(words2.values())
    print(msg.format(verb, num_common, plural, file1, tot1, file2, tot2))

    if num_common > 0:
        fmt = '{:>3} {:20} {:>5} {:>5}'
        print(fmt.format('#', 'word', '1', '2'))
        print('-' * 36)
        shared1, shared2 = 0, 0
        for i, word in enumerate(sorted(common)):
            c1 = words1[word]
            c2 = words2[word]
            shared1 += c1
            shared2 += c2
            print(fmt.format(i + 1, word, c1, c2))

        print(fmt.format('', '-----', '--', '--'))
        print(fmt.format('', 'total', shared1, shared2))
        print(fmt.format('', 'pct',
                         int(shared1/tot1 * 100), int(shared2/tot2 * 100)))

# --------------------------------------------------
if __name__ == '__main__':
    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 23: Count Character Frequency with Dictionaries

Improvement over GC content.

Solution

```
#!/usr/bin/env python3
"""
Author : Ken Youens-Clark <kyclark@gmail.com>
Date   : 2019-02-06
Purpose: Character Counter
"""

import argparse
import os
import sys
from collections import Counter


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

    parser.add_argument('input', help='Filename or string to count', type=str)

    parser.add_argument(
        '-c',
        '--charsort',
        help='Sort by character',
        dest='charsort',
        action='store_true')

    parser.add_argument(
        '-n',
        '--numsort',
        help='Sort by number',
        dest='numsort',
        action='store_true')

    parser.add_argument(
        '-r',
        '--reverse',
        help='Sort in reverse order',
        dest='reverse',
        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()
    input_arg = args.input
    charsort = args.charsort
    numsort = args.numsort
    revsort = args.reverse

    if charsort and numsort:
        die('Please choose one of --charsort or --numsort')

    if not charsort and not numsort:
        charsort = True

    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()
```

Chapter 24: Word Frequency with Dictionaries

Does not include Hamming.

Solution

```
#!/usr/bin/env python3
"""Count words/frequencies in two files"""

import os
import re
import sys
import string
from collections import defaultdict

# --------------------------------------------------
def word_counts(file):
    """Return a dictionary of words/counts"""
    words = defaultdict(int)
    regex = re.compile('[' + string.punctuation + ']')
    for line in open(file):
        for word in [regex.sub('', w) for w in line.lower().split()]:
            words[word] += 1

    return words

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

    if len(args) != 2:
        msg = 'Usage: {} FILE1 FILE2'
        print(msg.format(os.path.basename(sys.argv[0])))
        sys.exit(1)

    for file in args[0:2]:
        if not os.path.isfile(file):
            print('"{}" is not a file'.format(file))
            sys.exit(1)

    file1 = args[0]
    file2 = args[1]
    words1 = word_counts(file1)
    words2 = word_counts(file2)
    common = set(words1.keys()).intersection(set(words2.keys()))
    num_common = len(common)
    verb = 'is' if num_common == 1 else 'are'
    plural = '' if num_common == 1 else 's'
    msg = 'There {} {} word{} in common between "{}" ({}) and "{}" ({}).'
    tot1 = sum(words1.values())
    tot2 = sum(words2.values())
    print(msg.format(verb, num_common, plural, file1, tot1, file2, tot2))

    if num_common > 0:
        fmt = '{:>3} {:20} {:>5} {:>5}'
        print(fmt.format('#', 'word', '1', '2'))
        print('-' * 36)
        shared1, shared2 = 0, 0
        for i, word in enumerate(sorted(common)):
            c1 = words1[word]
            c2 = words2[word]
            shared1 += c1
            shared2 += c2
            print(fmt.format(i + 1, word, c1, c2))

        print(fmt.format('', '-----', '--', '--'))
        print(fmt.format('', 'total', shared1, shared2))
        print(fmt.format('', 'pct',
                         int(shared1/tot1 * 100), int(shared2/tot2 * 100)))

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

Chapter 25: Character Frequency Histogram

Write a Python program called `histy.py` that takes a single positional argument that may be plain text or the name of a file to read for the text. Count the frequency of each character (not spaces) and print a histogram of the data. By default, you should order the histogram by the characters but include `-f|--frequency_sort` option to sort by the frequency (in descending order). Also include a `-c|--character` option (default `|`) to represent a mark in the histogram, a `-m|--minimum` option (default `1`) to include a character in the output, a `-w|--width` option (default `70`) to limit the size of the histogram, and a `-i|--case_insensitive` flag to force all input to uppercase.

```
$ ./histy.py
usage: histy.py [-h] [-c str] [-m int] [-w int] [-i] [-f] str
histy.py: error: the following arguments are required: str
$ ./histy.py -h
usage: histy.py [-h] [-c str] [-m int] [-w int] [-i] [-f] str

Histogrammer

positional arguments:
  str                   Input text or file

optional arguments:
  -h, --help            show this help message and exit
  -c str, --character str
                        Character for marks (default: |)
  -m int, --minimum int
                        Minimum frequency to print (default: 1)
  -w int, --width int   Maximum width of output (default: 70)
  -i, --case_insensitive
                        Case insensitive search (default: False)
  -f, --frequency_sort  Sort by frequency (default: False)
$ ./histy.py ../inputs/fox.txt
T      1 |
a      1 |
b      1 |
c      1 |
d      1 |
e      3 |||
f      1 |
g      1 |
h      2 ||
i      1 |
j      1 |
k      1 |
l      1 |
m      1 |
n      1 |
o      4 ||||
p      1 |
q      1 |
r      2 ||
s      1 |
t      1 |
u      2 ||
v      1 |
w      1 |
x      1 |
y      1 |
z      1 |
$ ./histy.py ../inputs/const.txt -fim 100 -w 50 -c '#'
E   5107 ##################################################
T   3751 ####################################
O   2729 ##########################
S   2676 ##########################
A   2675 ##########################
N   2630 #########################
I   2433 #######################
R   2206 #####################
H   2029 ###################
L   1490 ##############
D   1230 ############
C   1164 ###########
F   1021 #########
U    848 ########
P    767 #######
M    730 #######
B    612 #####
Y    504 ####
V    460 ####
G    444 ####
W    375 ###
```

Solution

```
#!/usr/bin/env python3
"""Histogrammer"""

import argparse
import os
import re
from collections import Counter
from dire import die


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

    parser.add_argument('text', metavar='str', help='Input text or file')

    parser.add_argument('-c',
                        '--character',
                        help='Character for marks',
                        metavar='str',
                        type=str,
                        default='|')

    parser.add_argument('-m',
                        '--minimum',
                        help='Minimum frequency to print',
                        metavar='int',
                        type=int,
                        default=1)

    parser.add_argument('-w',
                        '--width',
                        help='Maximum width of output',
                        metavar='int',
                        type=int,
                        default=70)

    parser.add_argument('-i',
                        '--case_insensitive',
                        help='Case insensitive search',
                        action='store_true')

    parser.add_argument('-f',
                        '--frequency_sort',
                        help='Sort by frequency',
                        action='store_true')

    return parser.parse_args()


# --------------------------------------------------
def main():
    """Make a jazz noise here"""

    args = get_args()
    text = args.text
    char = args.character
    width = args.width
    min_val = args.minimum

    if len(char) != 1:
        die('--character "{}" must be one character'.format(char))

    if os.path.isfile(text):
        text = open(text).read()
    if args.case_insensitive:
        text = text.upper()

    freqs = Counter(filter(lambda c: re.match(r'\w', c), list(text)))
    high = max(freqs.values())
    scale = high / width if high > width else 1
    items = map(lambda t: (t[1], t[0]),
                sorted([(v, k) for k, v in freqs.items()],
                       reverse=True)) if args.frequency_sort else sorted(
                           freqs.items())

    for c, num in items:
        if num < min_val:
            continue
        print('{} {:6} {}'.format(c, num, char * int(num / scale)))


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

Chapter 26: Amino Acid Translation

Using dict!

Solution

```
#!/usr/bin/env python3
"""Codon/Amino Acid table conversion"""

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

aa2codons = {}
for line in codon_table.strip().splitlines():
    [aa, codons] = line.split(maxsplit=1)
    aa2codons[aa] = codons.split()

print('AA -> codons')
print(aa2codons)

codon2aa = {}
for aa, codons in aa2codons.items():
    for codon in codons:
        codon2aa[codon] = aa

print('Codon -> AA')
print(codon2aa)
```

Chapter 27: Translate DNA/RNA to Amino Acids

Write a Python program called `translate_proteins.py` that translates a given DNA/RNA sequence to amino acids using a provided codon table. The output will be written to a file either provided by the user or a default of "out.txt".

The DNA/RNA string and codon table are both required, so be sure to set `required=True` if creating with `parser.add_argument` so that your program produces a usage statement when no arguments are provided:

```
$ ./translate_proteins.py
usage: translate_proteins.py [-h] -c FILE [-o FILE] STR
translate_proteins.py: error: the following arguments are required: STR, -c/--codons
$ ./translate_proteins.py -h
usage: translate_proteins.py [-h] -c FILE [-o FILE] STR

Translate DNA/RNA to proteins

positional arguments:
  STR                   DNA/RNA sequence

optional arguments:
  -h, --help            show this help message and exit
  -c FILE, --codons FILE
                        A file with codon translations (default: None)
  -o FILE, --outfile FILE
                        Output filename (default: out.txt)
```

Die on a bad `--codons` argument:

```
$ ./translate_proteins.py -c foo AAA
--codons "foo" is not a file
```

If given good input, write the results to the proper output file:

```
$ ./translate_proteins.py -c codons.rna UGGCCAUGGCGCCCAGAACUGAGAUCAAUAGUACCCGUAUUAACGGGUGAA
Output written to "out.txt"
$ cat out.txt
WPWRPELRSIVPVLTGE
$ ./translate_proteins.py -c codons.dna gaactacaccgttctcctggt -o dna.out
Output written to "dna.out"
$ cat dna.out
ELHRSPG
```

Note that you might (well, you definitely *will*) be given the wrong codon table for a given sequence type. If you are creating a dictionary from the codon table, e.g.:

```
$ head -3 codons.rna
AAA K
AAC N
AAG K
```

Such that you have something like this:

```
>>> codons = dict(AAA='K', AAC='N', AAG='K')
```

Everything is fine as long as you ask for codons that are defined but will fail at runtime if you ask for a codon that does not exist:

```
>>> codons['AAC']
'N'
>>> codons['AAT']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'AAT'
```

If a codon does not appear in the table, use "-" instead:

```
$ ./translate_proteins.py -c codons.rna gaactacaccgttctcctggt
Output written to "out.txt"
$ cat out.txt
E-H----
```

The "Python Patterns" has an example of how to "Extract Codons from DNA" that will help you.

Solution

```
#!/usr/bin/env python3
"""
Author : Ken Youens-Clark <kyclark@gmail.com>
Date   : 2019-02-07
Purpose: Translate DNA/RNA to proteins
"""

import argparse
import os
import sys


# --------------------------------------------------
def get_args():
    """get command-line arguments"""
    parser = argparse.ArgumentParser(
        description='Translate DNA/RNA to proteins',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('sequence', metavar='STR', help='DNA/RNA sequence')

    parser.add_argument(
        '-c',
        '--codons',
        help='A file with codon translations',
        metavar='FILE',
        type=str,
        required=True)

    parser.add_argument(
        '-o',
        '--outfile',
        help='Output filename',
        metavar='FILE',
        type=str,
        default='out.txt')

    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()
    seq = args.sequence.upper()
    codon_file = args.codons
    out_file = args.outfile

    if not os.path.isfile(codon_file):
        die('--codons "{}" is not a file'.format(codon_file))

    out_fh = open(out_file, 'wt')

    codon_table = dict()
    for line in open(codon_file):
        codon, prot = line.upper().rstrip().split()
        codon_table[codon] = prot

    k = 3
    for codon in [seq[i:i+k] for i in range(0, len(seq), k)]:
        out_fh.write(codon_table.get(codon, '-'))

    out_fh.write('\n')
    out_fh.close()
    print('Output written to "{}"'.format(out_file))

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

Chapter 28: Parsing Files with Python

CSV

SRA metadata.

Excel, CSV, tab-delimited.

Bioinformatics File Formats

- FASTA
- FASTQ
- EMBL
- GenBank
- Muscle
- GFF
- SwissProt

BioPython Parsers

Data Exchange Formats on the Web

PubMed, NCBI Taxonomy, APIs

- JSON
- XML

Chapter 29: Parsing Tab-delimited BLAST Hits

Write a Python program called "blastomatic.py" that takes a BLAST hits file (-outfmt 6, tab-delimited format) as a single positional argument and a named "–annotations" argument that is an annotations file that gives genus and species information for a given sequence ID. Check that both are actually files and die '"XXX" is not a file' if they are not. Iterate over the BLAST hits and use the sequence ID (`saccver`) to lookup the sequence in the annotations file so that you can print out the seq ID and the percent identity (`pident`) from the hits file along with the `genus` and `species` from the annotations file.

As a BLAST tab-delimited file does not include headers, it would be helpful for you to read `blastn -help` to find what they are:

```
   When not provided, the default value is:
   'qaccver saccver pident length mismatch gapopen qstart qend sstart send
   evalue bitscore', which is equivalent to the keyword 'std'
```

You have two "hits" files that look like this (using my "blast6chk" alias, cf notes):

```
$ blast6chk hits1.tab
// ****** Record 1 ****** //
qseqid   : NR_125480.1
sseqid   : bfb6f5dfb4d0ef0842be8f5df6c86459
pident   : 99.567
length   : 231
mismatch : 1
gapopen  : 0
qstart   : 728
qend     : 958
sstart   : 1
send     : 231
evalue   : 3.93e-118
bitscore : 422
```

The providede "centroids.csv" annotation file looks like this:

```
$ tabchk.py centroids.csv
// ****** Record 1 ****** //
centroid : e5d49c0803f04032b482a1ee836e18ab
domain   : Bacteria
kingdom  : Proteobacteria
phylum   : Alphaproteobacteria
class    : Rhodospirillales
order    : AEGEAN-169 marine group
genus    : uncultured bacterium
species  : uncultured bacterium
```

When looking up the genus and species, print 'NA' when no useable value is present. For any sequence that cannot be found in the annotations file, print `Cannot find seq "XXX" in lookup` to STDERR.

Accept an optional "–out" argument that is the name of an output file to which to write the STDOUT of the program. If not provided, you will print to STDOUT.

The output should be tab-delimited with the fields "seq_id," "pident," "genus," and "species."

```
$ tabchk.py out
// ****** Record 1 ****** //
seq_id  : 229584169f4724188010dcfc36f2c933
pident  : 90.526
genus   : NA
species : NA
```

It should print a help message.

```
$ ./blastomatic.py -h
usage: blastomatic.py [-h] [-a FILE] [-o FILE] FILE

Annotate BLAST output

positional arguments:
  FILE                  BLAST output (-outfmt 6)

optional arguments:
  -h, --help            show this help message and exit
  -a FILE, --annotations FILE
                        Annotation file (default: )
  -o FILE, --outfile FILE
                        Output file (default: )
$ ./blastomatic.py -a foo bar
"bar" is not a file
$ ./blastomatic.py -a centroids.csv foo
"foo" is not a file
$ ./blastomatic.py -a centroids.csv hits1.tab -o out 2>&1 | head
Cannot find seq "875518c5d2436c94f50924425cb37f42" in lookup
Cannot find seq "2e5eeadcccb672a3410ddc6a8ff9ceee" in lookup
Cannot find seq "e16e05492dbcdbeb1de332614d5d002d" in lookup
Cannot find seq "39491c3b0dce84b718a274eafff3915c" in lookup
Cannot find seq "f42d5121911f169e12fd4c6bac1977f3" in lookup
Cannot find seq "1caa4b8dabc32ca88ce99513239e0a45" in lookup
Cannot find seq "e064229aac7487f068c9b8abf4a741e0" in lookup
Cannot find seq "661c26e0a8ac2956e6ba5b52dcaf11f2" in lookup
Cannot find seq "b4cd45a37eefcc49e5e9e153dffa783d" in lookup
Cannot find seq "197b74f559ec647315375dd5588792f3" in lookup
$ ./blastomatic.py -a centroids.csv hits1.tab 2>err | head | column -t
seq_id                            pident  genus             species
bfb6f5dfb4d0ef0842be8f5df6c86459  99.567  Prochlorococcus  MIT9313  NA
0dab11245fb6fe800362cdc20953d0f6  98.701  Prochlorococcus  MIT9313  Ambiguous_taxa
9c2271504f3393684fd1ed93d1d1a9ab  98.701  Prochlorococcus  MIT9313  Ambiguous_taxa
26cbd1b8b6fcd255774f4f79be2f259c  98.701  Prochlorococcus  MIT9313  NA
6192b152a8c84ff13fe6a7dced9c9357  98.268  Prochlorococcus  MIT9313  NA
61d060a46dadd0fbcdb099bbf4a36221  98.268  Prochlorococcus  MIT9313  NA
6da08abcdd74ae66dd2ef4112384faa5  98.268  Prochlorococcus  MIT9313  Ambiguous_taxa
50d394faf698e238e9bd05b251499cee  97.835  Prochlorococcus  MIT9313  NA
1642658999590e25a39926d281dea501  96.537  Synechococcus    CC9902   NA
```

Solution

```
#!/usr/bin/env python3
"""
Author : Ken Youens-Clark <kyclark@gmail.com>
Date   : 2019-02-25
Purpose: Rock the Casbah
"""

import argparse
import csv
import os
import sys


# --------------------------------------------------
def get_args():
    """get command-line arguments"""
    parser = argparse.ArgumentParser(
        description='Annotate BLAST output',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument(
        'hits', metavar='FILE', help='BLAST output (-outfmt 6)')

    parser.add_argument(
        '-a',
        '--annotations',
        help='Annotation file',
        metavar='FILE',
        type=str,
        default='')

    parser.add_argument(
        '-o',
        '--outfile',
        help='Output file',
        metavar='FILE',
        type=str,
        default='')

    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()
    hits_file = args.hits
    annots_file = args.annotations
    out_file = args.outfile

    for file in [hits_file, annots_file]:
        if not os.path.isfile(file):
            die('"{}" is not a file'.format(file))

    lookup = {}
    with open(annots_file) as csvfile:
        reader = csv.DictReader(csvfile, delimiter=',')
        for row in reader:
            lookup[row['centroid']] = row

    blast_flds = [
        'qseqid', 'sseqid', 'pident', 'length', 'mismatch', 'gapopen',
        'qstart', 'qend', 'sstart', 'send', 'evalue', 'bitscore'
    ]

    out_fh = open(out_file, 'wt') if out_file else sys.stdout
    out_fh.write('\t'.join(['seq_id', 'pident', 'genus', 'species']) + '\n')

    with open(hits_file) as csvfile:
        reader = csv.DictReader(csvfile, delimiter='\t', fieldnames=blast_flds)
        for row in reader:
            seq_id = row['sseqid']
            if seq_id not in lookup:
                warn('Cannot find seq "{}" in lookup'.format(seq_id))
                continue

            info = lookup[seq_id]
            out_fh.write('\t'.join(
                [row['sseqid'], row['pident'], info['genus'] or 'NA',
                 info['species'] or 'NA']) + '\n')

    out_fh.close()


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

Chapter 30: Parsing Putative Genes in Prodigal 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) or may be blank, so you should skip those. Further, the last field in GFF is basically a dumping ground for whatever metadata the author felt like putting there. Usually it's a bunch of "key=value" pairs, but there's no guarantee.

In our example, we have run the Prodigal gene predictor on a sample and wish to find putative genes with a minimum score. Prodigal can create output in either GenBank or GFF format, but the information contained is the same. Cf:

https://github.com/hyattpd/Prodigal/wiki/Understanding-the-Prodigal-Output

GFF Structure

Take a look at the GFF output from Prodigal in `HUMANGUT_SMPL_INB.fa.prodigal.gff`. The first line is:

```
##gff-version  3
```

The double `##` means the line is not just a comment but a "directive" or a "pragma" and says this file follows the conventions of GFF version 3 which can be found at https://github.com/The-Sequence-Ontology/Specifications/blob/master/gff3.md.

According to the specifications, the columns are:

1. seqid
2. source
3. type
4. start
5. end
6. score
7. strand
8. phase
9. attributes

The ninth column reserves the following fields:

- ID
- Name
- Alias
- Parent

- Target
- Gap
- Derives_from
- Note
- Dbxref
- Ontology_term
- Is_circular

The second line is rather long and looks basically like this:

```
# Sequence Data: seqnum=1;seqlen=4867;
seqhdr="HumanGut_CONTIG_00235296
/accession=HumanGut_CONTIG_00235296
/length=4867 /length=4867
/sample_id=1340106823570556171
/sample_acc=HUMANGUT_SMPL_INB
/sample_name=HUMANGUT_SMPL_INB
/site_id_n=HUMANGUT_SITE_INB"
```

This is metadata provided by Prodigal about the subject file. You can see there are a couple of levels of *key=value* pairs. First in the pairs separated by `;` (`seqnum`, `seqlen`, and `seqhdr`), and then `seqhdr` contains additional *key=value* pairs separated by spaces and starting with a `/`.

The third line is also long and provides metadata about the model Prodigal used in the analysis:

```
# Model Data: version=Prodigal.v2.6.3;
run_type=Single;model="Ab initio";gc_cont=54.91;transl_table=11;uses_sd=0
```

Finally on line 4 the actual data starts. We can inspect it with a bit of command-line fu:

```
$ awk 'NR==4' HUMANGUT_SMPL_INB.fa.prodigal.gff | tabchk.py -N -
// ****** Record 1 ****** //
Field1 : HumanGut_CONTIG_00235296
Field2 : Prodigal_v2.6.3
Field3 : CDS
Field4 : 17
Field5 : 157
Field6 : 3.3
Field7 : +
Field8 : 0
Field9 : ID=1_1;partial=00;start_type=GTG;rbs_motif=None;\
rbs_spacer=None;gc_cont=0.688;conf=68.33;score=3.35;cscore=11.92;\
sscore=-8.57;rscore=-4.31;uscore=-1.25;tscore=-3.01;
```

It's not helpful to see "Field1" and such, so let's add in the field names:

```
$ awk 'NR==4' HUMANGUT_SMPL_INB.fa.prodigal.gff | \
> tabchk.py -f seqid,source,type,start,end,score,strand,frame,attributes -
// ****** Record 1 ****** //
seqid      : HumanGut_CONTIG_00235296
source     : Prodigal_v2.6.3
type       : CDS
start      : 17
end        : 157
score      : 3.3
strand     : +
frame      : 0
attributes : ID=1_1;partial=00;start_type=GTG;rbs_motif=None;rbs_spacer=None;\
             gc_cont=0.688;conf=68.33;score=3.35;cscore=11.92;sscore=-8.57;\
             rscore=-4.31;uscore=-1.25;tscore=-3.01;
```

Now we'd like to find all the CDS records that had a `score` greater than some threshold. For that, we're going to need to check the `type` field and then find the `score` hidden in the `attributes` field.

Just to be sure, what are the values for the `type` field?

```
$ awk -F"\t" 'NR>4 {print ($3==""?"NA":$3)}' \
> HUMANGUT_SMPL_INB.fa.prodigal.gff | sort | uniq -c
 356 CDS
 148 NA
```

Parsing GFF with csv.DictReader

BioPython does not yet include a GFF parser, so we'll just handle it ourselves. Besides, there's so much we can learn! As stated before, there may be some comment lines that start with `#` that we need to skip, and the data lines are 9 tab-delimited fields for which we have names. All of this can be succinctly described to the `csv.DictReader` like so:

```
flds = 'seqid source type start end score strand frame attributes'.split()
reader = csv.DictReader(filter(lambda line: line[0] != '#', fh),
                        fieldnames=flds,
                        delimiter='\t')
```

The first argument to `csv.DictReader` should be a "stream" – something that will produce the next line of input. An open file handle is usually what we pass, but it's also possible to use, say `io.StringIO` to make a string behave like a file handle. Here we are creating a `filter` object that will only allow lines from the `fh` that do not begin with `#`. We can also pass a list of `fieldnames`, and that the `delimiter` is a Tab character (`\t`).

We could write a *generator* function that would `yield` each line of the file like so:

```
def src():
    for line in fh:
        if line[0] != '#':
            yield line

reader = csv.DictReader(src(), fieldnames=flds, delimiter='\t')
```

If you have more complex logic than will comfortably fit into a `filter`, it might be better to write it like this.

Iterating GFF Records

With our handy CSV `reader`, we can do `for rec in reader` to iterate over a sequence of dictionaries that look like this:

```
OrderedDict([('seqid', 'HumanGut_CONTIG_00235296'),
             ('source', 'Prodigal_v2.6.3'),
             ('type', 'CDS'),
             ('start', '17'),
             ('end', '157'),
             ('score', '3.3'),
             ('strand', '+'),
             ('frame', '0'),
             ('attributes',
              'ID=1_1;partial=00;start_type=GTG;<...elided...>')])
```

I cut down the `attributes` for the moment. Since we only want "CDS" records, we `continue` (skip to the next iteration of the loop) `if rec['type'] != 'CDS'`.

Parsing GFF Attributes

The `rec['attributes']` are separated by the semi-colon `;`, so we can use that to `split` the string:

```
>>> attrs = 'ID=1_1;partial=00;start_type=GTG;rbs_motif=None;score=3.35'
>>> attrs.split(';')
['ID=1_1', 'partial=00', 'start_type=GTG', 'rbs_motif=None', 'score=3.35']
```

As it happens, they all have the structure "key=value", so we could `split` each of those on the equal sign `=`, but it's far safer to use a regular expression to validate that we have something that *really* looks like a key and value. This will make the script far more flexible and reusable!

```
>>> import re
>>> kv = re.compile('([^=]+)=([^=]+)')
>>> match = kv.match('partial=00')
>>> match
```

```
<re.Match object; span=(0, 10), match='partial=00'>
>>> match.groups()
('partial', '00')
```

When the `match` fails, it returns `None`, so it's important that we check that each attribute actually matched the regex. I chose to `map` each of the attributes into the regex. Note that here I introduce a fake attribute that won't match so you can see the `None`:

```
>>> attrs = 'ID=1_1;partial=00;start_type=GTG;rbs_motif=None;score=3.35;ABC'
>>> from pprint import pprint as pp
>>> pp(list(map(kv.match, attrs.split(';'))))
[<re.Match object; span=(0, 6), match='ID=1_1'>,
 <re.Match object; span=(0, 10), match='partial=00'>,
 <re.Match object; span=(0, 14), match='start_type=GTG'>,
 <re.Match object; span=(0, 14), match='rbs_motif=None'>,
 <re.Match object; span=(0, 10), match='score=3.35'>,
 None]
```

If there is a `match`, I use the `groups` method to unpack the two capturing groups into `key, value` so I can set my `attr` dictionary to those.

Printing Wanted Values

Lastly we need to see if there is a `score` attribute by asking if that string exists in the keys of the `attrs` dictionary. If it does, we `try` to convert the value to a `float`. The `str` class has a very handy `isnumeric` method that can tell us if a string looks like an integer but it doesn't work with a float:

```
>>> '1'.isnumeric()
True
>>> '1.2'.isnumeric()
False
```

We could write a regular expression to check if a string looks like a floating point number, but there are really quite a lot of ways to represent a float, so it's easier to just see if Python is able to make the conversion. If we fall into the `except` branch, we just `pass` on to the next record. Only if we can make the conversion of `score` to a float and if that value is greather than or equal to the given `min_score` do we print out the sequence ID and the `score`.

Chapter 31: Summarize Centrifuge Hits by Tax Name

Solution

```python
#!/usr/bin/env python3
"""Counts by tax name"""

import csv
import os
import sys
from collections import defaultdict

args = sys.argv[1:]

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

sum_file = args[0]

basename, ext = os.path.splitext(sum_file)
if not ext == '.sum':
    print('File extention "{}" is not ".sum"'.format(ext))
    sys.exit(1)

tsv_file = basename + '.tsv'
if not os.path.isfile(tsv_file):
    print('Cannot find expected TSV "{}"'.format(tsv_file))
    sys.exit(1)

tax_name = {}
with open(tsv_file) as csvfile:
    reader = csv.DictReader(csvfile, delimiter='\t')
    for row in reader:
        tax_name[row['taxID']] = row['name']

counts = defaultdict(int)
with open(sum_file) as csvfile:
    reader = csv.DictReader(csvfile, delimiter='\t')
    for row in reader:
        taxID = row['taxID']
        counts[taxID] += 1

print('\t'.join(['count', 'taxID']))
for taxID, count in counts.items():
    name = tax_name.get(taxID) or 'NA'
    print('\t'.join([str(count), name]))
```

Chapter 32: Find Pairwise Sample Geographic Distance

Given a list of sample/lat/lon, find/filter all pairwise sample distances.

Solution

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

import argparse
import csv
import re
import os
import sys
from itertools import combinations
from geopy.distance import vincenty

# --------------------------------------------------
def get_args():
    """get args"""
    parser = argparse.ArgumentParser(description='Argparse Python script')
    parser.add_argument('-d', '--data', help='Tab-delimited samples file',
                        metavar='str', type=str, default='samples.tab')
    parser.add_argument('-s', '--sample_ids', help='Sample IDs (comma-sep)',
                        metavar='str', type=str, default='')
    return parser.parse_args()

# --------------------------------------------------
def main():
    """main"""
    args = get_args()
    infile = args.data
    sample_ids = re.split(r'\s*,\s*', args.sample_ids) if args.sample_ids else []
    print(sample_ids)

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

    records = []
    with open(infile) as fh:
        reader = csv.DictReader(fh, delimiter='\t')
        for rec in reader:
            records.append(rec)
    print('# rec = {}'.format(len(records)))

    combos = combinations(range(len(records)), 2)
    for i, j in combos:
        s1, s2 = records[i], records[j]
        dist = vincenty((s1['latitude'], s1['longitude']),
                        (s2['latitude'], s2['longitude']))
        lat1, long1 = s1['latitude'], s1['longitude']
        print('{} -> {} = {}'.format(s1['sample_id'], s2['sample_id'], dist))
        print(s1)
        print(s2)



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

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`.

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
```

Solution

```
#!/usr/bin/env python3
"""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='str', help='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',
        help='How many records to show',
        metavar='int',
        type=int,
        default=1)

    parser.add_argument(
        '-d',
        '--dense',
        help='Not sparse (skip empty fields)',
        action='store_true')

    parser.add_argument(
        '-n',
        '--number',
        help='Show field number (e.g., for awk)',
        action='store_true')

    parser.add_argument(
        '-N',
        '--no_headers',
        help='No headers in first row',
        action='store_true')

    return parser.parse_args()


# --------------------------------------------------
def main():
    """main"""
    args = get_args()
    file = args.file
    limit = args.limit
    sep = args.sep
    dense = args.dense
    show_numbers = args.number
    no_headers = args.no_headers

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

    if not sep:
        _, ext = os.path.splitext(file)
        if ext == '.csv':
            sep = ','
        else:
            sep = '\t'

    with open(file) as csvfile:
        dict_args = {'delimiter': sep}
        if args.field_names:
            regex = re.compile(r'\s*,\s*')
            names = regex.split(args.field_names)
            if names:
                dict_args['fieldnames'] = names

        if args.no_headers:
            num_flds = len(csvfile.readline().split(sep))
            dict_args['fieldnames'] = list(map(lambda i: 'Field' + str(i),
                                           range(1, num_flds + 1)))
            csvfile.seek(0)

        reader = csv.DictReader(csvfile, **dict_args)

        for i, row in enumerate(reader):
            vals = dict([x for x in row.items()
                         if x[1] != '']) if dense else row
            flds = vals.keys()
            longest = max(map(len, flds))
            fmt = '{:' + str(longest + 1) + '}: {}'
            print('// ****** Record {} ****** //'.format(i + 1))
            n = 0
            for key, val in vals.items():
                n += 1
                show = fmt.format(key, val)
                if show_numbers:
                    print('{:3} {}'.format(n, show))
                else:
                    print(show)

            if i + 1 == limit:
                break


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

Chapter 34: 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.

Solution

```
#!/usr/bin/env python3
"""
Author:  Ken Youens-Clark <kyclark@gmail.com>
Purpose: Convert a delimited text file to JSON
"""

import argparse
import csv
import json
import os
import re
import sys


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

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

    parser.add_argument(
        '-s',
        '--sep',
        help='Field separator',
        metavar='str',
        type=str,
        default='\t')

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

    parser.add_argument(
        '-i',
        '--indent',
        help='Indent level',
        metavar='int',
        type=int,
        default=2)

    parser.add_argument(
        '-n',
        '--normalize_headers',
        help='Normalize headers',
        action='store_true')

    return parser.parse_args()


# --------------------------------------------------
def main():
    """main"""
    args = get_args()
    indent_level = args.indent
    out_dir = args.outdir
    fs = args.sep
    norm_hdr = args.normalize_headers
    tabfiles = args.tabfile

    if len(tabfiles) < 1:
        print('No input files')
        sys.exit(1)

    if indent_level < 0:
        indent_level = 0

    if out_dir and not os.path.isdir(out_dir):
        os.makedirs(out_dir)

    for i, tabfile in enumerate(tabfiles, start=1):
        basename = os.path.basename(tabfile)
        filename, _ = os.path.splitext(basename)
        dirname = os.path.dirname(os.path.abspath(tabfile))
        print('{:3}: {}'.format(i, basename))
        write_dir = out_dir if out_dir else dirname
        out_path = os.path.join(write_dir, filename + '.json')
        out_fh = open(out_path, 'wt')

        with open(tabfile) as fh:
            reader = csv.DictReader(fh, delimiter=fs)
            if norm_hdr:
                reader.fieldnames = list(map(normalize, reader.fieldnames))
            out_fh.write(json.dumps(list(reader), indent=indent_level))


# --------------------------------------------------
def normalize(hdr):
    return re.sub(r'[^A-Za-z0-9_]', '', hdr.lower().replace(' ', '_'))


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

Chapter 35: FASTA Summary With Seqmagique

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.

Solution

```
#!/usr/bin/env python3
"""
Author:  Ken Youens-Clark <kyclark@gmail.com>
Purpose: Mimic seqmagick, print stats on FASTA sequences
"""

import os
import sys
import numpy as np
from Bio import SeqIO

files = sys.argv[1:]

if not files:
    print('Usage: {} F1.fa [F2.fa...]'.format(os.path.basename(sys.argv[0])))
    sys.exit(1)

info = []
for file in files:
    lengths = []
    for record in SeqIO.parse(file, 'fasta'):
        lengths.append(len(record.seq))

    info.append({
        'name': os.path.basename(file),
        'min_len': min(lengths),
        'max_len': max(lengths),
        'avg_len': '{:.2f}'.format(np.mean(lengths)),
        'num_seqs': len(lengths)
    })

if info:
    longest_file_name = max([len(f['name']) for f in info])
    fmt = '{:' + str(longest_file_name) + '} {:10} {:10} {:10} {:10}'
    flds = ['name', 'min_len', 'max_len', 'avg_len', 'num_seqs']
    print(fmt.format(*flds))
    for rec in info:
        print(fmt.format(*[rec[fld] for fld in flds]))
else:
    print('I had trouble parsing your data')
```

Chapter 36: 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:

Solution

```
#!/usr/bin/env python3
"""
Author:  Ken Youens-Clark <kyclark@gmail.com>
Purpose: Subset FASTA/Q files
"""

import argparse
import os
import sys
from Bio import SeqIO


# --------------------------------------------------
def get_args():
    """get args"""
    parser = argparse.ArgumentParser(
        description='Subset 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 sequences to take',
        type=int,
        metavar='NUM',
        default=500000)

    parser.add_argument(
        '-o',
        '--outfile',
        help='Output file',
        type=str,
        metavar='FILE',
        default='subset')

    parser.add_argument(
        '--force',
        help='Force overwrite of existing file',
        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():
    """main"""
    args = get_args()
    in_file = args.file
    in_fmt = args.infmt
    out_fmt = args.outfmt if args.outfmt else args.infmt
    out_file = args.outfile
    num_seqs = args.num

    if not os.path.isfile(in_file):
        die('--file "{}" is not a file'.format(in_file))

    if out_file == in_file:
        die('--outfile "{}" cannot be the same as input file'.format(out_file))

    if num_seqs < 1:
        die("--num cannot be less than one")

    if os.path.isfile(out_file) and not args.force:
        while True:
            answer = input(
                '--outfile "{}" exists. Overwrite [yes|no]? '.format(
                    out_file)).lower()
            if answer == 'no':
                print('Bye')
                sys.exit(1)
            elif answer == 'yes':
                break
            else:
                print('Please answer yes or no')

    out_fh = open(out_file, 'wt')
    num_written = 0

    for record in SeqIO.parse(in_file, in_fmt):
        SeqIO.write(record, out_fh, out_fmt)
        num_written += 1

        if num_written == num_seqs:
            break

    print('Done, wrote {} sequence{} to "{}"'.format(
        num_written, '' if num_written == 1 else 's', out_file))


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

Chapter 37: Randomly Subset a FASTA file

Here is a version that will randomly select some percentage of the reads from the input file. I had to write this version because we had created an artificial metagenome from a set of known organisms, and I was testing a program with input of various numbers of reads. I did not realize at first that, in creating the artificial set, reads from each organism had been added in blocks. Since I was taking all my reads from the top of the file down, I was mostly getting just the first few species. Randomly selecting reads when there are potentially millions of records is a bit tricky, so I decided to use a non-deterministic approach where I just roll the dice and see if the number I get on each read is less than the percentage of reads I want to take. This program will also stop at a given number of reads so you could use it to randomly subset an unevenly sized number of samples down to the same number of reads per sample.

Solution

```
#!/usr/bin/env python3
"""
Author:  Ken Youens-Clark <kyclark@gmail.com>
Purpose: Probabalistically subset FASTQ/A
"""

import argparse
import os
import re
import sys
from random import randint
from Bio import SeqIO


# --------------------------------------------------
def get_args():
    """get args"""
    parser = argparse.ArgumentParser(
        description='Randomly subset FASTQ',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('file', metavar='FILE', help='FASTQ/A file')

    parser.add_argument(
        '-p',
        '--pct',
        help='Percent of reads',
        metavar='int',
        type=int,
        default=50)

    parser.add_argument(
        '-m',
        '--max',
        help='Maximum number of reads',
        metavar='int',
        type=int,
        default=0)

    parser.add_argument(
        '-f',
        '--input_format',
        help='Intput format',
        metavar='IN_FMT',
        type=str,
        choices=['fastq', 'fasta'],
        default='')

    parser.add_argument(
        '-F',
        '--output_format',
        help='Output format',
        metavar='OUT_FMT',
        type=str,
        choices=['fastq', 'fasta'],
        default='')

    parser.add_argument(
        '-o',
        '--outfile',
        help='Output file',
        metavar='FILE',
        type=str,
        default='')

    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():
    """main"""
    args = get_args()
    file = args.file
    pct = args.pct
    out_file = args.outfile
    max_num_reads = args.max
    min_num = 0
    max_num = 100

    if not os.path.isfile(file):
        die('"{}" is not a file'.format(file))

    in_fmt = args.input_format
    if not in_fmt:
        _, ext = os.path.splitext(file)
        in_fmt = 'fastq' if re.match('\.f(ast)?q$', ext) else 'fasta'

    out_fmt = args.output_format or in_fmt

    if not min_num < pct < max_num:
        msg = '--pct "{}" must be between {} and {}'
        die(msg.format(pct, min_num, max_num))

    if not out_file:
        base, _ = os.path.splitext(file)
        out_file = '{}.sub{}.{}'.format(base, pct, out_fmt)

    out_fh = open(out_file, 'wt')
    num_taken = 0
    total_num = 0

    with open(file) as fh:
        for rec in SeqIO.parse(fh, in_fmt):
            total_num += 1
            if randint(min_num, max_num) <= pct:
                num_taken += 1
                SeqIO.write(rec, out_fh, out_fmt)
                if max_num_reads > 0 and num_taken == max_num_reads:
                    break

    out_fh.close()

    print('Wrote {} of {} ({:.02f}%) to "{}"'.format(
        num_taken, total_num, num_taken / total_num * 100, out_file))


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

Chapter 38: 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.

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
```

Solution

```
#!/usr/bin/env python3
"""
Author:  Ken Youens-Clark <kyclark@gmail.com>
Purpose: Split FASTA files
NB:      If you have FASTQ files, maybe just use "split"?
"""

import argparse
import os
import sys
from Bio import SeqIO


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

    parser.add_argument('file', help='FASTA input file(s)', nargs='+')

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

    parser.add_argument(
        '-F',
        '--output_format',
        help='Output file format',
        type=str,
        metavar='FORMAT',
        choices=['fasta', 'fastq'],
        default='fasta')

    parser.add_argument(
        '-n',
        '--sequences_per_file',
        help='Number of sequences per file',
        type=int,
        metavar='NUM',
        default=50)

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

    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():
    """main"""
    args = get_args()
    files = args.file
    input_format = args.input_format
    output_format = args.output_format
    out_dir = args.out_dir
    seqs_per_file = args.sequences_per_file

    if not os.path.isdir(out_dir):
        os.mkdir(out_dir)

    if seqs_per_file < 1:
        die('--sequences_per_file "{}" cannot be less than one'.format(
            seqs_per_file))

    num_files = 0
    num_seqs_written = 0
    for i, file in enumerate(files, start=1):
        print('{:3d}: {}'.format(i, os.path.basename(file)))
        num_files += 1
        num_seqs_written += process(
            file=file,
            input_format=input_format,
            output_format=output_format,
            out_dir=out_dir,
            seqs_per_file=seqs_per_file)

    print('Done, processed {} sequence{} from {} file{} into "{}"'.format(
        num_seqs_written, '' if num_seqs_written == 1 else 's', num_files, ''
        if num_files == 1 else 's', out_dir))


# --------------------------------------------------
def process(file, input_format, output_format, out_dir, seqs_per_file):
    """
    Spilt file into smaller files into out_dir
    Optionally convert to output format
    Return number of sequences written
    """
    if not os.path.isfile(file):
        warn('"{}" is not valid'.format(file))
        return 0

    basename, ext = os.path.splitext(os.path.basename(file))
    out_fh = None
    i = 0
    num_written = 0
    nfile = 0
    for record in SeqIO.parse(file, input_format):
        if i == seqs_per_file:
            i = 0
            if out_fh is not None:
                out_fh.close()
                out_fh = None

        i += 1
        num_written += 1
        if out_fh is None:
            nfile += 1
            path = os.path.join(out_dir,
                                basename + '.' + '{:04d}'.format(nfile) + ext)
            out_fh = open(path, 'wt')

        SeqIO.write(record, out_fh, output_format)

    return num_written


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

Chapter 39: FASTA Segregator by GC Content

Write a Python program called `gc.py` that takes

- One or more FASTA files as positional arguments
- An option `-o|--outdir` directory name to write the output (default "out")
- An option `-p|--pct_gc` integer value between 1-100 (inclusive) for the percent GC to decide high/low content (default 50)

Generate a "usage" if given no arguments or the `-h|--help` argument.

The program should iterate through the FASTA files, calculate the GC content of each sequence, and write the sequence to a "high" file if the percent GC is greater than or equal the `--pct_gc` argument or to the "low" file if it is lower than the argument. The name of the high/low files should be the basename of the input file plus "_[low/high]"; that is, if the input file is"foo.fa" then you should create "foo_high.fa" and "foo_low.fa" in the given `--outdir`. Note that if the output directory does not exist, you will need to create it.

If a given "file" argument is not a file, print ' "XXX" is not a file' to STDERR and continue processing.

```
$ ./gc.py
usage: gc.py [-h] [-o DIR] [-p int] FASTA [FASTA ...]
gc.py: error: the following arguments are required: FASTA
$ ./gc.py -h
usage: gc.py [-h] [-o DIR] [-p int] FASTA [FASTA ...]

Segregate FASTA sequences by GC content

positional arguments:
  FASTA                 Input FASTA file(s)

optional arguments:
  -h, --help            show this help message and exit
  -o DIR, --outdir DIR  Output directory (default: out)
  -p int, --pct_gc int  Dividing line for percent GC (default: 50)
$ ./gc.py foo
"foo" is not a file
Done, wrote 0 sequences to out dir "out"
$ ./gc.py fasta/CAM_SMPL_GS108.fa
  1: CAM_SMPL_GS108.fa
Done, wrote 500 sequences to out dir "out"
$ ./gc.py -p 32 -o splits fasta/*
  1: CAM_SMPL_GS108.fa
  2: CAM_SMPL_GS112.fa
Done, wrote 1000 sequences to out dir "splits"
```

Hints

You will proabably want to use the following methods:

- os.makedirs: to create directories
- os.path.basename: to get the filename from a path
- os.path.join: join (in a OS-independent way) directory names and files
- os.path.splitext: to split “foo.fa” into “foo” and “.fa”
- Use the SeqIO module from Bio to `parse` and `write` sequences

Look back at the many examples of counting DNA to pick a method you like to count the number of Gs and Cs, then divide by the length of the sequence. Remember this will give you something like “0.43219” and you are comparing to an integer like “43,” so do the proper math.

Solution

```
#!/usr/bin/env python3
"""
Author : kyclark
Date   : 2019-02-19
Purpose: Rock the Casbah
"""

import argparse
import os
import sys
from Bio import SeqIO
from collections import Counter


# --------------------------------------------------
def get_args():
    """get command-line arguments"""
    parser = argparse.ArgumentParser(
        description='Segregate FASTA sequences by GC content',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument(
        'fasta', metavar='FASTA', help='Input FASTA file(s)', nargs='+')

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

    parser.add_argument(
        '-p',
        '--pct_gc',
        help='Dividing line for percent GC',
        metavar='int',
        type=int,
        default=50)

    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()
    out_dir = args.outdir
    pct_gc = args.pct_gc

    if not os.path.isdir(out_dir):
        os.makedirs(out_dir)

    if not 0 < pct_gc <= 100:
        die('--pct_gc "{}" must be between 0 and 100'.format(pct_gc))

    num_seqs = 0
    for i, file in enumerate(args.fasta, start=1):
        if not os.path.isfile(file):
            warn('"{}" is not a file'.format(file))
            continue

        print('{:3}: {}'.format(i, os.path.basename(file)))

        base, ext = os.path.splitext(os.path.basename(file))
        high_file = os.path.join(out_dir, ''.join([base, '_high', ext]))
        low_file = os.path.join(out_dir, ''.join([base, '_low', ext]))

        high_fh = open(high_file, 'wt')
        low_fh = open(low_file, 'wt')

        for rec in SeqIO.parse(file, 'fasta'):
            num_seqs += 1
            bases = Counter(rec.seq.upper())
            gc = bases.get('G', 0) + bases.get('C', 0)
            pct = int((gc / len(rec.seq)) * 100)
```

```
                SeqIO.write(rec, low_fh if pct < pct_gc else high_fh, 'fasta')

    print('Done, wrote {} sequence{} to out dir "{}"'.format(
        num_seqs, '' if num_seqs == 1 else 's', out_dir))


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

Chapter 40: FASTA Interleaved Paired Read Splitter

Some sequencing platforms (e.g., Illumina) will create read pairs (forward/reverse) that may be interleaved together into one file with the forward read immediately followed by the reverse read or the reads may be in two separate files like `foo_1.fastq` and `foo_2.fastq` where `_1` is the forward read file and `_2` contains the reverse reads (or sometimes `_R1`/`_R2`).

Write a Python program called `au_pair.py` that accepts a list of positional arguments that are FASTA sequence files in interleaved format and splits them into `_1`/`_2` files in a `-o|--outdir` argument (default `split`). You should use the original extension of the file, e.g., `inputs/reads1.fa` should be split into `outdir/reads1_1.fa` and `outdir/reads1_2.fa` while `inputs/reads2.fasta` should be split into `outdir/reads2_1.fasta` and `outdir/reads2_2.fasta`.

As always, the program should provide usage statements on `-h|--help` or when run with no arguments. If one of the positional arguments is not a file, print `"<file>" is not a file` to STDERR and continue processing. If the `--outdir` does not exist, create it.

For the purposes of this exercise, assume the reads are properly interleaved such that the first read is forward and the second read is its reverse mate. Do not worry about testing the read IDs for forward/reverse or mate pair information. Also assume all input files are in FASTA format and should be written in FASTA format.

```
$ ./au_pair.py
usage: au_pair.py [-h] [-o DIR] FILE [FILE ...]
au_pair.py: error: the following arguments are required: FILE
$ ./au_pair.py -h
usage: au_pair.py [-h] [-o DIR] FILE [FILE ...]

Split interleaved/paired reads

positional arguments:
  FILE                  Input file(s)

optional arguments:
  -h, --help            show this help message and exit
  -o DIR, --outdir DIR  Output directory (default: split)
$ ./au_pair.py foo
"foo" is not a file
$ ./au_pair.py inputs/reads1.fa
  1: reads1.fa
    Split 4 sequences to dir "split"
$ ./au_pair.py inputs/reads2.fasta -o out
```

```
  1: reads2.fasta
    Split 500 sequences to dir "out"
$ ./au_pair.py inputs/* -o all
  1: reads1.fa
    Split 4 sequences to dir "all"
  2: reads2.fasta
    Split 500 sequences to dir "all"
```

Solution

```
#!/usr/bin/env python3
"""
Purpose: Split interleaved, paired reads into _1/2 files
Author:  Ken Youens-Clark <kyclark@email.arizona.edu>
"""

import argparse
import os
import sys
from Bio import SeqIO


# --------------------------------------------------
def get_args():
    """get args"""
    parser = argparse.ArgumentParser(
        description='Split interleaved/paired reads',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

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

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

    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():
    """main"""
    args = get_args()
    out_dir = args.outdir

    if out_dir and not os.path.isdir(out_dir):
        os.makedirs(out_dir)

    for fnum, file in enumerate(args.file):
        if not os.path.isfile(file):
            warn('"{}" is not a file'.format(file))
            continue

        filename = os.path.basename(file)
        base, ext = os.path.splitext(filename)
        forward = open(os.path.join(out_dir, base + '_1' + ext), 'wt')
        reverse = open(os.path.join(out_dir, base + '_2' + ext), 'wt')

        print("{:3d}: {}".format(fnum + 1, filename))

        num_seqs = 0
        for i, rec in enumerate(SeqIO.parse(file, 'fasta')):
            SeqIO.write(rec, forward if i % 2 == 0 else reverse, 'fasta')
            num_seqs += 1

        print('\tSplit {:,d} sequences to dir "{}"'.format(num_seqs, out_dir))


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

Chapter 41: FASTQ to FASTA

FASTA (sequence) plus "quality" scores for each base call gives us "FASTQ." Here is an example:

```
$ head -4 !$
head -4 input.fastq
@M00773:480:000000000-BLYPT:1:2106:12063:1841 1:N:0:AGGCGACCTTA
TTTCTGTGCCAGCAGCCGCGGTAAGACAGAGGTGGCGAGCGTTGTTCGGATTTACTGGGCGTAAAGCGCGGGTAGGCGGTTCGGCCAGTCAC
+
CCCCCGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGEFGGFEGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGFGGC
```

Because of inherent logical flaws in this file format, the only sane representation is for the record to consist of four lines:

1. header ('@', ID, desc, yadda yadda yadda)
2. sequence
3. spacer
4. quality scores (phred 33/64)

Here is what the record looks like:

```
>>> from Bio import SeqIO
>>> rec = list(SeqIO.parse('input.fastq', 'fastq'))[0]
>>> rec = list(SeqIO.parse('input.fastq', 'fastq'))[0]
>>> print(rec)
ID: M00773:480:000000000-BLYPT:1:2106:12063:1841
Name: M00773:480:000000000-BLYPT:1:2106:12063:1841
Description: M00773:480:000000000-BLYPT:1:2106:12063:1841 1:N:0:AGGCGACCTTA
Number of features: 0
Per letter annotation for: phred_quality
Seq('TTTCTGTGCCAGCAGCCGCGGTAAGACAGAGGTGGCGAGCGTTGTTCGGATTTA...CGC', SingleLetterAlphabet())
```

But this looks pretty much like a FASTA file, so where is the quality information? We have to look here (http://biopython.org/DIST/docs/api/Bio.SeqIO.QualityIO-module.html):

```
>>> print(rec.format("qual"))
>M00773:480:000000000-BLYPT:1:2106:12063:1841 1:N:0:AGGCGACCTTA
34 34 34 34 34 38 38 38 38 38 38 38 38 38 38 38 38 38 38 38
38 38 38 38 38 38 38 38 38 38 38 38 38 38 38 38 38 36 37 38
38 37 36 38 38 38 38 38 38 38 38 38 38 38 38 38 38 38 38 38
38 38 38 38 38 38 38 38 38 38 38 38 38 38 38 38 38 38 38 38
38 38 38 38 38 38 38 38 37 38 38 38 38 38 38 38 38 38 38 38
38 38 38 38 38 38 38 36 38 38 38 38 38 38 38 36 38 38 38 38
38 38 35 38 35 38 38 38 38 38 38 38 38 38 38 38 37 35 38 38
38 38 38 38 38 38 37 37 37 37 35 37 38 22 37 37 38 38 38 38
38 38 38 38 38 22 36 38 38 38 38 38 35 38 38 36 38 38 38 38
38 38 28 36 37 38 35 38 38 37 38 38 35 36 38 38 38 38 37 37
```

```
34 20 26 36 36 35 37 36 37 38 36 38 37 34 37 38 36 36 34 34
23 30 20 34 36 36 9 25 20 9 26 30 37 38 38 37 38 34 34 37
38 32 37 37 38 38 38 35 38 38 37 37 38 34 35 36 34 38 38 38
38 36 26 36 36 23 36 34 28 18 24 15 26 20 22 20 29 23 27 10
24 37 38 38 37 34 27 23 34 38 37 37 25 24 10 24 11 27 35 20
8
```

We can combine the bases and their quality scores into a list of tuples (which can naturally become a dictionary):

```
>>> list(zip(rec.seq, rec.format('qual')))
[('T', '>'), ('T', 'M'), ('T', 'O'), ('C', 'O'), ...
>>> for base, qual in zip(rec.seq, rec.format('qual')):
...   print('base = "{}" qual = "{}"'.format(base, qual))
...   break
...
base = "T" qual = ">"
```

The scores are based on the ordinal represenation of the quality characters' ASCII values. Cf:

- https://www.rapidtables.com/code/text/ascii-table.html
- https://www.drive5.com/usearch/manual/quality_score.html

We can convert FASTQ to FASTA by simply changing the leading "@" in the header to ">" and then removing lines 3 and 4 from each record. Here is an [g]awk one-liner to do that:

```
#!/bin/gawk -f

### fq2fa.awk
##
## Copyright Tomer Altman
##
### Desription:
##
## Given a FASTQ formatted file, transform it into a FASTA nucleotide file.

(FNR % 4) == 1 || (FNR % 4) == 2 { gsub("^@", ">"); print }
```

Can you write one in Python?

Solution

```
#!/usr/bin/env python3

import argparse
import os
import sys
from Bio import SeqIO

# --------------------------------------------------
def get_args():
    """get command-line arguments"""
    parser = argparse.ArgumentParser(
        description='Converst FASTQ to FASTA',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument(
        'fastq', metavar='FILE', nargs='+', help='FASTQ file(s)')

    parser.add_argument(
        '-e',
        '--extension',
        help='File extension',
        metavar='str',
        type=str,
        default='fa')

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

    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()
    out_dir = args.outdir

    if not os.path.isdir(out_dir):
        os.makedirs(out_dir)

    ext = args.extension
    if not ext.startswith('.'):
        ext = '.' + ext

    for i, fq in enumerate(args.fastq, start=1):
        basename = os.path.basename(fq)
        root, _ = os.path.splitext(basename)
        print('{:3}: {}'.format(i, basename))

        out_file = os.path.join(out_dir, root + ext)
        out_fh = open(out_file, 'wt')

        for record in SeqIO.parse(fq, 'fastq'):
            SeqIO.write(record, out_fh, 'fasta')

    print('Done, see output in "{}".'.format(out_dir))

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

Chapter 42: Concatenate FASTX Files

Given a directory/list of FASTQ/A files like this:

```
1.SRR170176.fastq
2.SRR170506.fastq
3.SRR170739.fastq
4.SRR328519.fastq
5.SRR047943.fastq
6.SRR048028.fastq
```

Concatenate all the sequences into one file. If a header looks like this:

```
@GPSBU5C02GK9PQ
```

Turn it into this:

```
@1.SRR170176_GPSBU5C02GK9PQ
```

Solution

```
#!/usr/bin/env python3
"""
Given a directory/list of FASTQ/A files like this:

    1.SRR170176.fastq
    2.SRR170506.fastq
    3.SRR170739.fastq
    4.SRR328519.fastq
    5.SRR047943.fastq
    6.SRR048028.fastq

Concatenate all the sequences into one file. If a header looks like this:

    @GPSBU5C02GK9PQ

Turn it into this:

    @1.SRR170176_GPSBU5C02GK9PQ

Author: Ken Youens-Clark <kyclark@email.arizona.edu>
Date: 17 September 2018
"""

import argparse
import os
import re
import sys
from Bio import SeqIO


# --------------------------------------------------
def get_args():
    """get args"""
    parser = argparse.ArgumentParser(
        description='Input file(s)/directories',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument(
        'input', metavar='FILE_DIR', help='File or directory', nargs='+')

    parser.add_argument(
        '-l',
        '--limit',
        help='Limit per file',
        metavar='int',
        type=int,
        default=0)

    parser.add_argument(
        '-o',
        '--outfile',
        help='Output filename',
        metavar='str',
        type=str,
        default='')

    parser.add_argument(
        '-i',
        '--in_format',
        help='Input file format',
        metavar='str',
        type=str,
        default='')

    parser.add_argument(
        '-f',
        '--out_format',
        help='Output file format',
        metavar='str',
        type=str,
        default='')

    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 find_files(inputs):
    files = []
    for arg in inputs:
        if os.path.isfile(arg):
            files.append(arg)
        elif os.path.isdir(arg):
            for filename in os.listdir(arg):
                files.append(os.path.join(arg, filename))

    if not files:
        die('No input files!')

    files.sort()

    return files


# --------------------------------------------------
def main():
    """Make a jazz noise here"""
    args = get_args()
    files = find_files(args.input)
    input_format = args.in_format
    output_format = args.out_format
    out_file = args.outfile
    limit_per_file = args.limit

    if not out_file:
        die('Missing --outfile')

    out_fh = open(out_file, 'wt')
    fastq_re = re.compile(r'^\.f(ast)?q$')
    num_seqs = 0
    num_files = len(files)

    print('Will process {} file{}'.format(num_files, ''
                                          if num_files == 1 else 's'))

    for i, filename in enumerate(files):
        basename = os.path.basename(filename)
        acc, ext = os.path.splitext(basename)
        print('{:3}: Processing {}'.format(i + 1, basename))

        file_format = input_format if input_format else 'fastq' if fastq_re.match(
            ext) else 'fasta'

        for j, record in enumerate(SeqIO.parse(filename, file_format)):
            num_seqs += 1
            record.id = '{}_{}'.format(acc, record.id)
            record.description = ''
            SeqIO.write(record, out_fh, output_format
                        if output_format else file_format)
            if limit_per_file > 0 and j + 1 == limit_per_file:
                break

    out_fh.close()

    print('Done, wrote {} to outfile "{}"'.format(num_seqs, out_file))


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

Chapter 43: Parsing NCBI Taxonomy XML

Here's an example that looks at XML from the NCBI taxonomy. Here is what the raw file looks like:

```
$ head ena-101.xml
<?xml version="1.0" encoding="UTF-8"?>
<SAMPLE alias="SAMD00024455" accession="DRS018892" broker_name="DDBJ">
     <IDENTIFIERS>
          <PRIMARY_ID>DRS018892</PRIMARY_ID>
          <EXTERNAL_ID namespace="BioSample">SAMD00024455</EXTERNAL_ID>
          <SUBMITTER_ID namespace="">SAMD00024455</SUBMITTER_ID>
     </IDENTIFIERS>
     <TITLE>Surface water bacterial community from the East China Sea Site 100</TITLE>
     <SAMPLE_NAME>
          <TAXON_ID>408172</TAXON_ID>
```

The whitespace in XML is not significant and simply bloats the size of the file, so often you will get something that is unreadable. I recommend you install the program `xmllint` to look at such files. If you inspect the file, you can see that XML gives us a way to represent hierarchical data unlike CSV files which are essentially “flat” (unless you start sticking things like lists and key/value pairs [dictionaries]). We need to use a specific XML parser and use accessors that look quite a bit like file paths. There is a “root” of the XML from which we can descend into the structure to find data. Here is a program that will extract various parts of the XML.

```
$ ./xml_ena.py ena-101.xml
>>>>>> ena-101.xml
sample.alias            : SAMD00024455
sample.accession        : DRS018892
sample.broker_name      : DDBJ
id.PRIMARY_ID           : DRS018892
id.EXTERNAL_ID          : SAMD00024455
id.SUBMITTER_ID         : SAMD00024455
attr.sample_name        : 100A
attr.collection_date    : 2013-08-15/2013-08-28
attr.depth              : 0.5m
attr.env_biome          : coastal biome
attr.env_feature        : natural environment
attr.env_material       : water
attr.geo_loc_name       : China:the East China Sea
attr.lat_lon            : 29.3 N 122.08 E
attr.project_name       : seawater bacterioplankton
attr.BioSampleModel     : MIMARKS.survey.water
attr.ENA-SPOT-COUNT     : 54843
attr.ENA-BASE-COUNT     : 13886949
attr.ENA-FIRST-PUBLIC    : 2015-02-15
attr.ENA-LAST-UPDATE     : 2018-08-15
```

Solution

```
#!/usr/bin/env python3
"""
Author : Ken Youens-Clark <kyclark@gmail.com>
Date   : 2019-02-22
Purpose: Rock the Casbah
"""

import argparse
import os
import sys
from xml.etree.ElementTree import ElementTree


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

    parser.add_argument('xml', metavar='XML', help='XML input', nargs='+')

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

    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()
    xml_files = args.xml
    out_dir = args.outdir

    if not os.path.isdir(out_dir):
        os.makedirs(out_dir)

    for file in xml_files:
        print('>>>>>>', file)
        tree = ElementTree()
        root = tree.parse(file)

        d = []
        for key, value in root.attrib.items():
            d.append(('sample.' + key, value))

        for id_ in root.find('IDENTIFIERS'):
            d.append(('id.' + id_.tag, id_.text))

        for attr in root.findall('SAMPLE_ATTRIBUTES/SAMPLE_ATTRIBUTE'):
            d.append(('attr.' + attr.find('TAG').text, attr.find('VALUE').text))

        for key, value in d:
            print('{:25}: {}'.format(key, value))

        print()

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

Chapter 44: Fetching and Parsing PubMed JSON

Oh yeah.

Solution

```
#!/usr/bin/env python3
"""
Author : kyclark
Date   : 2019-05-16
Purpose: Fetch PubMed info, cf. http://www.ncbi.nlm.nih.gov/books/NBK25499/
"""

import argparse
import json
import pprint
import requests
import sys


# --------------------------------------------------
def get_args():
    """get command-line arguments"""
    parser = argparse.ArgumentParser(
        description='Fetch PubMed info',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('pubmed_id',
                        metavar='int',
                        type=int,
                        nargs='+',
                        help='PubMed ID(s)')

    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()

    pubmed_url = ('https://eutils.ncbi.nlm.nih.gov/entrez/eutils/'
                  'esummary.fcgi?db=pubmed&retmode=json&id={}')

    for pubmed_id in args.pubmed_id:
        r = requests.get(pubmed_url.format(pubmed_id))
        if r.status_code == 200:
            data = json.loads(r.text)
            result = data.get('result')
            if result:
                info = result.get(str(pubmed_id))
                if info:
                    pprint.PrettyPrinter().pprint(info)
                    print(info['title'], info['lastauthor'])


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

Chapter 45: Parsing SwissProt

The SwissProt format is one, like GenBank and EMBL, that allows for detailed annotation of a sequence whereas FASTA/Q are primarily devoted to the sequence/quality and sometimes metadata/annotations are crudely shoved into the header line. Parsing SwissProt, however, is no more difficult thanks to the `SeqIO` module. Most of the interesting non-sequence data is in the `annotations` which is a dictionary where the keys are strings like “accessions” and “keywords” and the values are ints, strings, and lists.

Here is an example program to print out the accessions, keywords, and taxonomy in a SwissProt record.

```
$ ./swissprot.py input.swiss
  1: G5EEM5
    ANNOT accessions://
        Nematoda
        Chromadorea
        Rhabditida
        Rhabditoidea
        Rhabditidae
        Peloderinae
        Caenorhabditis
```

You should look at the sample “input.swiss” file to get a greater understanding of what is contained.

Solution

```
#!/usr/bin/env python3

import argparse
import sys
from Bio import SeqIO


# --------------------------------------------------
def get_args():
    """get args"""
    parser = argparse.ArgumentParser(
        description='Parse Swissprot file',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

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

    return parser.parse_args()


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


# --------------------------------------------------
def main():
    """main"""
    args = get_args()
    file = args.file

    for i, record in enumerate(SeqIO.parse(file, "swiss"), start=1):
        print('{:3}: {}'.format(i, record.id))
        annotations = record.annotations

        for annot_type in ['accessions', 'keywords', 'taxonomy']:
            if annot_type in annotations:
                print('\tANNOT {}:'.format(annot_type))
                val = annotations[annot_type]
                if type(val) is list:
                    for v in val:
                        print('\t\t{}'.format(v))
                else:
                    print('\t\t{}'.format(val))



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

Chapter 46: Find Overlapping Genes in GFF

Solution

```
#!/usr/bin/env python3
"""
Author : kyclark
Date   : 2019-05-16
Purpose: Find overlapping genes in GFF file
"""

import argparse
import csv
import re
import roman
import sys
from urllib.parse import unquote
from collections import defaultdict
from itertools import chain, product


# --------------------------------------------------
def get_args():
    """get command-line arguments"""
    parser = argparse.ArgumentParser(
        description='Find overlapping genes in GFF file',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

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

    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 make_num(s):
    """String -> Int"""

    if s and s.isdigit():
        return int(s)
    else:
        return 0


# --------------------------------------------------
def roman_sort(chrs, skip=[]):
    """Sort chromosome names by Roman numeral values"""

    # Build a list of tuple with integer value and chr/Roman
    # e.g., [(1, 'chrI'), (5, 'chrV')]
    # if the chr should be skipped, assign the value -1
    ret = []
    for chr_name in chrs:
        num = -1
        if chr_name not in skip:
            match = re.match('chr(.+)', chr_name)
            if match:
                num = roman.fromRoman(match.group(1))
        ret.append((num, chr_name))

    # Sort the tuple/list, then take only
    # the 2nd member (chr name) of the tuple
    return list(map(lambda t: t[1], sorted(ret)))


# --------------------------------------------------
def main():
    """Make a jazz noise here"""
    args = get_args()

    flds = ('sequence source feature start end score '
            'strand frame attributes').split()

    # dictionaries for the forward/reverse genes on each chromosome
    forward = defaultdict(list)
    reverse = defaultdict(list)

    with open(args.file) as fh:
        # Remove comment lines in the GFF
        src = filter(lambda row: not row.startswith('#'), fh)
        reader = csv.DictReader(src, fieldnames=flds, delimiter='\t')

        for row in reader:
            if row['feature'] != 'gene': continue

            # Attributes look like "key1=val1;key2=val2"
            # Values may be URI encoded, so use "unquote" to fix
            attr = dict(
                map(
                    lambda x: (x[0], unquote(x[1])),
                    map(lambda s: s.split('='),
                        row['attributes'].split(';'))))

            gene_name = attr.get('Name', 'NA')
            chr_name = row['sequence']
            d = forward if row['strand'] == '+' else reverse
            d[chr_name].append(
                (gene_name, make_num(row['start']), make_num(row['end'])))

    chrs = roman_sort(set(chain(forward.keys(), reverse.keys())), skip='chrmt')
    longest = max(map(len, chrs))
    fmt = '{:' + str(longest) + 's}: {} [{}..{}] (+) => {} [{}..{}] (-)'

    for chr_name in chrs:
        combos = product(forward[chr_name], reverse[chr_name])
        for f, r in combos:
            if range(max(f[1], r[1]), min(f[2], r[2]) + 1):
                print(fmt.format(chr_name, *f, *r))


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

Chapter 47: Parsing SwissProt

> "Without requirements or design, programming is the art of adding bugs to an empty text file." - Louis Srygley

Create a Python program called "swisstake.py" that processes a SwissProt-formatted file as a positional argument. It should have a *required* `-k|--keyword` argument of the keyword to match in the "keyword" field of the input record in order to determine which sequences to "take" (hence the name). It should also have an *optional* `-s|--skip` argument to "skip" records with given taxa (which could be many so `nargs='+'`), as well as an *optional* `-o|--output` argument to where to write the output in FASTA format (default "out.fa").

If the given input file is not a file, it should die with ' "XXX" is not a file'.

```
$ ./swisstake.py -h
usage: swisstake.py [-h] [-s STR [STR ...]] -k STR [-o FILE] FILE

Filter Swissprot file for keywords, taxa

positional arguments:
  FILE                  Uniprot file

optional arguments:
  -h, --help            show this help message and exit
  -s STR [STR ...], --skip STR [STR ...]
                        Skip taxa (default: )
  -k STR, --keyword STR
                        Take on keyword (default: None)
  -o FILE, --output FILE
                        Output filename (default: out.fa)
$ ./swisstake.py swiss.txt
usage: swisstake.py [-h] [-s STR [STR ...]] -k STR [-o FILE] FILE
swisstake.py: error: the following arguments are required: -k/--keyword
$ ./swisstake.py -k proteome foo
"foo" is not a file
$ ./swisstake.py swiss.txt -k "complete proteome" -s Metazoa FUNGI viridiplantae
Processing "swiss.txt"
Done, skipped 14 and took 1. See output in "out.fa".
$ ./swisstake.py swiss.txt -k "complete proteome" -s metazoa fungi
Processing "swiss.txt"
Done, skipped 13 and took 2. See output in "out.fa".
```

BioPython SwissProt Record

A FASTA record had three attributes: ID, description, and sequence. A SwissProt record has considerably more which will make sense once you look at the file.

There are at least two ways I've found to parse a SwissProt record. One is use `SeqIO.parse(fh, 'swiss')` which gives you a record very similar to a FASTA record which has an `annotations` attribute which is a dictionary that looks like this:

```
>>> import pprint
>>> pp = pprint.PrettyPrinter(indent=4)
>>> pp.pprint(rec.annotations)
{   'accessions': ['P13813'],
    'date': '01-JAN-1990',
    'date_last_annotation_update': '20-JAN-2016',
    'date_last_sequence_update': '01-JAN-1990',
    'entry_version': 42,
    'keywords': ['Malaria', 'Repeat'],
    'ncbi_taxid': ['5850'],
    'organism': 'Plasmodium knowlesi',
    'protein_existence': 2,
    'references': [   Reference(title='Cloning and characterization of an abundant Plasmodiu
    'sequence_version': 1,
    'taxonomy': [   'Eukaryota',
                    'Alveolata',
                    'Apicomplexa',
                    'Aconoidasida',
                    'Haemosporida',
                    'Plasmodiidae',
                    'Plasmodium',
                    'Plasmodium (Plasmodium)']}
```

The other way is use the `Bio.SwissProt` module which has attributes for the same kind of information though sometimes called slightly different names, e.g.:

```
>>> sw1.organism_classification
['Eukaryota', 'Alveolata', 'Apicomplexa', 'Aconoidasida', 'Haemosporida', 'Plasmodiidae', 'P
```

Cf:

- https://biopython.readthedocs.io/en/latest/Tutorial/chapter_uniprot.html
- http://biopython.org/DIST/docs/api/Bio.SwissProt.Record-class.html

However you choose to parse, you should be able to pass the tests. FWIW, I used the first method.

Sets

We've talked about dictionaries quite a bit, and for this exercise I think you'll want something that is a natural extension of a dictionary called `set()` which is just a dictionary where the values are all `1`. If you have two lists, you can test for equality:

```
>>> a = ['foo', 'bar']
>>> b = ['foo', 'bar']
>>> a == b
True
>>> c = ['bar', 'foo']
>>> a == c
False
```

The list `c` has the same members but in a different order, so the lists definitely are not the same; however, if all you cared about what if the two lists shared the same items, you could sort them:

```
>>> sorted(a) == sorted(c)
True
```

But what if you wanted to know if there was some overlap. Clearly you can't use equality:

```
>>> d = ['foo', 'bar', 'baz']
>>> a == d
False
```

You have to individually check each element of `a` to see if they are in `d`:

```
>>> [e for e in a if e in d]
['foo', 'bar']
>>> [e for e in d if e in a]
['foo', 'bar']
>>> any([e for e in a if e in d])
True
```

That is the "intersection" of the two lists. The "difference" would be:

```
>>> [e for e in a if e not in d]
[]
>>> [e for e in d if e not in a]
['baz']
```

The "union" would be everything in both lists, which we can't easily do in one line of code; however, if we convert these lists to sets, then we can do all those calculations easily:

```
>>> a = set(['foo', 'bar'])
>>> d = set(['foo', 'bar', 'baz'])
>>> a.union(d)
set(['baz', 'foo', 'bar'])
>>> a.intersection(d)
set(['foo', 'bar'])
>>> a.difference(d)
set([])
>>> d.difference(a)
set(['baz'])
```

Keep this in mind when you are trying to find if there is an intersection of the taxa you are given with the taxa that are in the record.

Solution

```
#!/usr/bin/env python3
"""
Author:  Ken Youens-Clark <kyclark@gmail.com>
Purpose: Filter Swissprot file for keywords, taxa
"""

import argparse
import os
import sys
from Bio import SeqIO


# --------------------------------------------------
def get_args():
    """get args"""
    parser = argparse.ArgumentParser(
        description='Filter Swissprot file for keywords, taxa',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

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

    parser.add_argument(
        '-s',
        '--skip',
        help='Skip taxa',
        metavar='STR',
        type=str,
        nargs='+',
        default='')

    parser.add_argument(
        '-k',
        '--keyword',
        help='Take on keyword',
        metavar='STR',
        type=str,
        required=True)

    parser.add_argument(
        '-o',
        '--output',
        help='Output filename',
        metavar='FILE',
        type=str,
        default='out.fa')

    return parser.parse_args()


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


# --------------------------------------------------
def main():
    """main"""
    args = get_args()
    input_file = args.input
    out_file = args.output
    keyword = args.keyword.lower()
    skip = set(map(str.lower, args.skip))

    if not os.path.isfile(input_file):
        die('"{}" is not a file'.format(input_file))

    print('Processing "{}"'.format(input_file))

    num_skipped = 0
    num_taken = 0
    with open(out_file, "w") as out_fh:
        for record in SeqIO.parse(input_file, "swiss"):
            annot = record.annotations
            if skip and 'taxonomy' in annot:
                taxa = set(map(str.lower, annot['taxonomy']))
                if skip.intersection(taxa):
                    num_skipped += 1
                    continue

            if 'keywords' in annot:
                kw = set(map(str.lower, annot['keywords']))

                if keyword in kw:
                    num_taken += 1
                    SeqIO.write(record, out_fh, 'fasta')
                else:
                    num_skipped += 1

    print('Done, skipped {} and took {}. See output in "{}".'.format(
        num_skipped, num_taken, out_file))


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

Chapter 48: Filter FASTA by Taxonomy

Given a FASTA file, filter for those sequences in the given tax list.

Solution

```
#!/usr/bin/env python3
"""
Author : kyclark
Date   : 2019-05-22
Purpose: Filter FASTA by taxons
"""

import argparse
import logging
import signal
import os
import re
import sys
from dire import die, warn
from Bio import Entrez, SeqIO


# --------------------------------------------------
def get_args():
    """get command-line arguments"""
    parser = argparse.ArgumentParser(
        description='Filter FASTA by taxons',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

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

    parser.add_argument('-t',
                        '--taxa',
                        help='Taxa ids/file',
                        metavar='str',
                        type=str,
                        required=True)

    parser.add_argument('-e',
                        '--email',
                        help='Email address for Entrez query',
                        metavar='str',
                        type=str,
                        default='kyclark@email.arizona.edu')

    parser.add_argument('-o',
                        '--outfile',
                        help='Output file',
                        metavar='str',
                        type=str,
                        default='seqs.fa')

    parser.add_argument('-d', '--debug', help='Debug', action='store_true')

    return parser.parse_args()


# --------------------------------------------------
def get_tax_names(taxa):
    """Get tax names from ids or string"""

    logging.debug('Checking tax inputs')

    def splitter(s):
        return re.split('\s*,\s*', s)

    tax_ids = []
    if os.path.isfile(taxa):
        for line in open(taxa):
            tax_ids.extend(splitter(line.rstrip()))
    else:
        tax_ids = splitter(taxa)

    tax_names = []
    for tax in tax_ids:
        logging.debug('Tax {}'.format(tax))

        if tax.isdigit():
            handle = Entrez.efetch(db='taxonomy', id=tax)
            results = Entrez.read(handle)
            if results:
                name = results[0].get('ScientificName')
                if name:
                    tax_names.append(name)
        else:
            tax_names.append(tax)

    return set(tax_names)


# --------------------------------------------------
def main():
    """Make a jazz noise here"""

    args = get_args()
    file = args.file
    out_file = args.outfile
    Entrez.email = args.email

    logging.basicConfig(
        filename='.log',
        filemode='w',
        level=logging.DEBUG if args.debug else logging.CRITICAL)

    ok_taxa = get_tax_names(args.taxa) or die('No usable taxa')
    logging.debug('OK tax = {}'.format(ok_taxa))

    logging.debug('Writing to "{}"'.format(out_file))
    out_fh = open(out_file, 'wt')
    num_checked, num_taken = 0, 0

    for rec in SeqIO.parse(args.file, 'fasta'):
        num_checked += 1
        print('{:4}: {}'.format(num_checked, rec.id))

        handle = Entrez.efetch(db='nucleotide',
                               id=rec.id,
                               rettype='gb',
                               retmode='text')

        for record in SeqIO.parse(handle, 'genbank'):
            tax = set(record.annotations.get('taxonomy'))
            tax_hit = ok_taxa.intersection(tax)
            if tax_hit:
                logging.debug('Taking {} ({})'.format(rec.id, tax_hit))
                num_taken += 1
                SeqIO.write(record, 'fasta', out_fh)

    print('Done, checked {}, wrote {} to "{}"'.format(num_checked, num_taken,
                                                      out_file))


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

Chapter 49: Filter Reads by Centrifuge Taxa Classification

Solution

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

import os
import argparse
import re
import sys
import gzip
from Bio import SeqIO

# --------------------------------------------------
def get_args():
    parser = argparse.ArgumentParser(description='Filter FASTA with Centrifuge')
    parser.add_argument('-f', '--fasta', help='fasta file',
            type=str, metavar='FILE', required=True)
    parser.add_argument('-s', '--summary', help='Centrifuge summary file',
            type=str, metavar='FILE', required=True)
    parser.add_argument('-e', '--exclude', metavar='IDS_NAMES', required=True,
            help='Comma-separated list of taxIDs/names to exclude')
    parser.add_argument('-o', '--out_dir', help='Output directory',
            type=str, metavar='DIR', default='filtered')
    parser.add_argument('-x', '--exclude_dir', metavar='DIR', required=True,
            help='File name to write excluded')
    return parser.parse_args()

# --------------------------------------------------
def read_split(s):
    return s.rstrip("\n").split("\t")

# --------------------------------------------------
def get_excluded(x, sum_file):
    if not sum_file.endswith('.sum'):
        print('sum_file ({}) does not end with ".sum"'.format(sum_file))
        sys.exit

    tsv_file = re.sub(r'\.sum$', '.tsv', sum_file)
    if not os.path.exists(tsv_file):
        print('Cannot find TSV file ({})'.format(tsv_file))
        sys.exit

    name_to_id = dict()
    with open(tsv_file) as tsv:
        hdr = read_split(tsv.readline())
        for line in tsv:
            rec = dict(zip(hdr, read_split(line)))
            name_to_id[ rec['name'].lower() ] = rec['taxID']

    exclude = set()
    for arg in re.split('\s*,\s*', x.lower()):
        if str.isdigit(arg):
            exclude.add(arg)
        else:
            if arg in name_to_id:
                exclude.add(name_to_id[arg])
            else:
                print('Cannot find name "{}" in {}'.format(arg, tsv_file))
                sys.exit

    return exclude

# --------------------------------------------------
def main():
    args    = get_args()
    exclude = get_excluded(args.exclude, args.summary)
    out_dir = args.out_dir

    if not os.path.isdir(out_dir):
        os.mkdir(out_dir)

    tax_id = dict()
    with open(args.summary, 'r') as sum_fh:
        sum_hdr = read_split(sum_fh.readline())
        for line in sum_fh:
            #info = dict(zip(sum_hdr, read_split(line)))
            #tax_id[ info['readID'] ] = info['taxID']
            dat = read_split(line)
            tax_id[ dat[0] ] = dat[2]

    took       = 0
    skipped    = 0
    basename   = os.path.basename(args.fasta)
    out_file   = os.path.join(out_dir, basename)
    exclude_fh = open(os.path.join(args.exclude_dir, basename), 'wt') \
                 if args.exclude_dir else None

    with open(out_file, 'w') as out_fh:
        for seq in SeqIO.parse(args.fasta, "fasta"):
            tax = tax_id[ seq.id ] if seq.id in tax_id else '0'
            if tax in exclude:
                skipped += 1
                if not exclude_fh is None:
                    SeqIO.write(seq, exclude_fh, "fasta")
            else:
                took += 1
                SeqIO.write(seq, out_fh, "fasta")

    print("Done, took {}, skipped {}, see output {}".format(
        took, skipped, out_file))

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

Chapter 50: Fetching and Parsing PubMed JSON

Oh yeah.

Solution

```
#!/usr/bin/env python3
"""
Author : kyclark
Date   : 2019-05-16
Purpose: Fetch PubMed info, cf. http://www.ncbi.nlm.nih.gov/books/NBK25499/
"""

import argparse
import json
import pprint
import requests
import sys


# --------------------------------------------------
def get_args():
    """get command-line arguments"""
    parser = argparse.ArgumentParser(
        description='Fetch PubMed info',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('pubmed_id',
                        metavar='int',
                        type=int,
                        nargs='+',
                        help='PubMed ID(s)')

    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()

    pubmed_url = ('https://eutils.ncbi.nlm.nih.gov/entrez/eutils/'
                  'esummary.fcgi?db=pubmed&retmode=json&id={}')

    for pubmed_id in args.pubmed_id:
        r = requests.get(pubmed_url.format(pubmed_id))
        if r.status_code == 200:
            data = json.loads(r.text)
            result = data.get('result')
            if result:
                info = result.get(str(pubmed_id))
                if info:
                    pprint.PrettyPrinter().pprint(info)
                    print(info['title'], info['lastauthor'])


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

Chapter 51: Find Unclustered Proteins with Python

Run `make data` to get the data you need for this exercise or manually download the data:

```
wget ftp://ftp.imicrobe.us/biosys-analytics/exercises/unclustered-proteins.tgz
```

Unpack the tarball with `tar xvf unclustered-proteins.tgz`.

Write a Python program called `find_unclustered.py` that will create a FASTA file of the unclustered proteins. The program will take a `-c|--cdhit` argument that is the name of the CD-HIT cluster file, a `-p|--proteins` FASTA file, and an `-o|--outfile` argument (default `unclustered.fa`) where to write the sequences.

```
$ ./find_unclustered.py
usage: find_unclustered.py [-h] -c str -p str [-o str]
find_unclustered.py: error: the following arguments are required: -c/--cdhit, -p/--proteins
$ ./find_unclustered.py -h
usage: find_unclustered.py [-h] -c str -p str [-o str]

Find unclustered proteins

optional arguments:
  -h, --help            show this help message and exit
  -c str, --cdhit str   Output file from CD-HIT (clustered proteins) (default:
                        None)
  -p str, --proteins str
                        Proteins FASTA (default: None)
  -o str, --outfile str
                        Output file (default: unclustered.fa)
```

If either of the CD-HIT or proteins files are not files, die with an error message:

```
$ ./find_unclustered.py -c foo -p unclustered-proteins/proteins.fa
--cdhit "foo" is not a file
$ ./find_unclustered.py -c unclustered-proteins/cdhit60.3+.clstr -p foo
--proteins "foo" is not a file
```

If successful, report the number of unclustered proteins written to the indicated output file:

```
$ ./find_unclustered.py -c unclustered-proteins/cdhit60.3+.clstr -p unclustered-proteins/pro
Wrote 204,262 of 220,520 unclustered proteins to "unclustered.fa"
$ ./find_unclustered.py -c unclustered-proteins/cdhit60.3+.clstr -p unclustered-proteins/pro
Wrote 204,262 of 220,520 unclustered proteins to "unprots.fa"
```

Discussion

The file `cdhit60.3+.clstr` contains all of the GI numbers for proteins that were clustered and put into HMM profiles. It is almost in FASTA format, but not quite.

```
$ 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 protein IDs are in the bit that looks like ">gi|317183610|gb|ADV…" where the ID is "317183610". A regex would be the perfect thing to extract this.

The file `proteins.fa` contains all proteins. The protein ID is usually the only thing in the header:

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

But not always:

```
$ 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]
```

I would recommend you first parse the CD-HIT file and get all the protein IDs that have been clustered. You only need to know that they were in some cluster, so a `set` is a good data structure although a dictionary is fine, too. Proteins may be clustered more than once, but you don't need to keep track of that.

Once you know which protein IDs were clustered, go through the `proteins.fa` file and print the unclustered proteins to the given outfile. You will need to remove anything from the ID starting with a `|` (pipe character). The `re` module has a `sub` function that "substitutes" some pattern with a replacement. Because the `|` is a metacharacter inside a regex, it must be backslash-escaped or put into a character class:

```
>>> re.sub('\|.*', '', 'foo|bar')
'foo'
>>> re.sub('[|].*', '', 'foo|bar')
```

```
'foo'
```

Solution

```
#!/usr/bin/env python3
"""
Author : Ken Youens-Clark <kyclark@gmail.com>
Date   : 2019-02-20
Purpose: Rock the Casbah
"""

import argparse
import os
import re
import sys
from Bio import SeqIO


# --------------------------------------------------
def get_args():
    """get command-line arguments"""
    parser = argparse.ArgumentParser(
        description='Find unclustered proteins',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

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

    parser.add_argument(
        '-c',
        '--cdhit',
        help='Output file from CD-HIT (clustered proteins)',
        metavar='str',
        type=str,
        required=True)

    parser.add_argument(
        '-p',
        '--proteins',
        help='Proteins FASTA',
        metavar='str',
        type=str,
        required=True)

    parser.add_argument(
        '-o',
        '--outfile',
        help='Output file',
        metavar='str',
        type=str,
        default='unclustered.fa')

    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()
    proteins_file = args.proteins
    cdhit_file = args.cdhit
    out_file = args.outfile

    for arg_name, file in [('--proteins', proteins_file), ('--cdhit',
                                                             cdhit_file)]:
        if not os.path.isfile(file):
            die('{} "{}" is not a file'.format(arg_name, file))

    clustered = set()
    for line in open(cdhit_file):
        matches = re.search(r'>gi\|(?P<gi_num>\d+)\|', line)
        if matches:
            clustered.add(matches.group('gi_num'))

        # if line.startswith('>'):
        #     continue
        #
        # flds = line.split()
        # prot_id = flds[2].split('|')[1]
        # if prot_id.isdigit():
```

```
        #     clustered.add(prot_id)

    out_fh = open(out_file, 'wt')
    num_total = 0
    num_unclustered = 0

    for rec in SeqIO.parse(proteins_file, 'fasta'):
        num_total += 1
        prot_id = re.sub(r'\|.*', '', rec.id)
        if not prot_id in clustered:
            num_unclustered += 1
            SeqIO.write(rec, out_fh, 'fasta')

    print('Wrote {:,d} of {:,d} unclustered proteins to "{}"'.format(
        num_unclustered, num_total, out_file))


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

Chapter 52: Expanding DNA IUPAC Codes with Regular Expression

Write a program called `iupac.py` that translates an IUPAC-encoded (https://www.bioinformatics.org/sms/iupac.html) string of DNA into a regular expression that will match all the possible strings of DNA that match.

IUPAC nucleotide code	Base
A	Adenine
C	Cytosine
G	Guanine
T	Thymine
U	Uracil
R	A/G
Y	C/T
S	G/C
W	A/T
K	G/T
M	A/C
B	C/G/T
D	A/G/T
H	A/C/T
V	A/C/G
N	any base
./-	gap

For instance, the pattern `AYG` would match both `ACG` and `ATG`, so the regular expression would be `^A[CT]G$`. We can use the REPL to verify that this works:

```
>>> import re
>>> re.search('^A[CT]G$', 'ACG')
<re.Match object; span=(0, 3), match='ACG'>
>>> re.search('^A[CT]G$', 'ATG')
<re.Match object; span=(0, 3), match='ATG'>
>>> 'OK' if re.search('^A[CT]G$', 'ACG') else 'NO'
'OK'
```

Your program should echo the given pattern and a translation to a regular expression. Then iterate through a sorted list of all possible combinations of the bases to test your regular expression, printing "OK" if there is a match and "NO" if not.

```
$ ./iupac.py AYG
pattern = "AYG"
```

```
regex   = "^A[CT]G$"
ACG OK
ATG OK
$ ./iupac.py MRY
pattern = "MRY"
regex   = "^[AC][AG][CT]$"
AAC OK
AAT OK
AGC OK
AGT OK
CAC OK
CAT OK
CGC OK
CGT OK
```

Solution

```
#!/usr/bin/env python3
"""
Author : kyclark
Date   : 2019-05-15
Purpose: Turn IUPAC DNA codes into regex
"""

import argparse
import re
import sys
from itertools import product


# --------------------------------------------------
def get_args():
    """get command-line arguments"""
    parser = argparse.ArgumentParser(
        description='Turn IUPAC DNA codes into regex',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('pattern', 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()
    pattern = args.pattern
    trans = dict([('A', ('A', )), ('C', ('C', )), ('G', ('G', )),
                  ('T', ('T', )), ('U', ('U', )), ('R', ('A', 'G')),
                  ('Y', ('C', 'T')), ('S', ('G', 'C')), ('W', ('A', 'T')),
                  ('K', ('G', 'T')), ('M', ('A', 'C')), ('B', ('C', 'G', 'T')),
                  ('D', ('A', 'G', 'T')), ('H', ('A', 'C', 'T')),
                  ('V', ('A', 'C', 'G')), ('N', ('A', 'C', 'G', 'T'))])

    bases = sorted(trans.keys())
    if not re.search('^[' + ''.join(bases) + ']+$', pattern):
        die('Pattern must contain only {}.'.format(', '.join(bases)))

    iupac = list(map(lambda base: trans[base], pattern))
    regex = '^' + ''.join(
        map(lambda t: '[' + ''.join(t) + ']' if len(t) > 1 else t[0],
            iupac)) + '$'

    print('pattern = "{}"'.format(pattern))
    print('regex   = "{}"'.format(regex))

    for possibility in sorted(product(*iupac)):
        dna = ''.join(possibility)
        print(dna, 'OK' if re.search(regex, dna) else 'NO')

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

Chapter 53: Parsing Date Formats with Regular Expressions

Write a Python program called `dates.py` that takes as a single, positional argument a string and attempt to parse it as one of the given date formats. If given no argument, it should print a usage statement. It does not need to respond to `-h|--help`, so you could use `new_py.py` without the argparse flag.

```
$ ./dates.py
Usage: dates.py DATE
```

If you are able to match one of the acceptable format strings below, print the date in a standard "YYYY-MM-DD" format. If only given year and month, e.g. "12/06," use "1" as the day. When there is a range of dates (e.g., "2015-01/2015-02"), only parse the first one.

These are the formats you will be given:

```
$ cat eg_dates.txt
2012-03-09T08:59
2012-03-09T08:59:03
2017-06-16Z
2015-01
2015-01/2015-02
2015-01-03/2015-02-14
20100910
12/06
2/14
2/14-12/15
2017-06-16Z
Dec-2015
Dec, 2015
March-2017
April, 2017
```

Here is the expected output:

```
$ while read -r DATE; do ./dates.py "$DATE"; done < eg_dates.txt
2012-03-09
2012-03-09
2017-06-16
2015-01-01
2015-01-01
2015-01-03
2010-09-10
2006-12-01
2014-02-01
2014-02-01
```

```
2017-06-16
2015-12-01
2015-12-01
2017-03-01
2017-04-01
```

If you are unable to parse the argument, print "No match":

```
$ ./dates.py foo
No match
$ ./dates.py 1999.12.31
No match
```

While there are date parsing modules, I do not want you to use those in your code. Please write your own regular expressions and parsing code.

Solution

```
#!/usr/bin/env python3
"""
Author : Ken Youens-Clark <kyclark@gmail.com>
Date   : 2019-03-24
Purpose: Rock the Casbah
"""

import os
import re
import sys


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

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

    date = args[0]

    re1 = re.compile('^(?P<year>\d{4})-(?P<month>\d{1,2})(?:-(?P<day>\d{1,2}))?')
    re2 = re.compile('^(?P<year>\d{4})(?P<month>\d{1,2})(?P<day>\d{1,2})$')
    re3 = re.compile('^(?P<month>\d{1,2})[/](?P<year>\d{2})')
    re4 = re.compile('^(?P<month>'
                     'Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec'
                     ')'
                     '[,-]'
                     '\s*'
                     '(?P<year>\d{4})')
    re5 = re.compile('^(?P<month>'
                     'January|February|March|April|May|June|July|August|'
                     'September|October|November|December'
                     ')'
                     '[,-]'
                     '\s*'
                     '(?P<year>\d{4})')

    match1 = re1.search(date) or re2.search(date)
    match2 = re3.search(date)
    match3 = re4.search(date) or re5.search(date)

    short_months = 'Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec'.split()
    short_mon2num = dict(map(reversed, enumerate(short_months, 1)))

    long_months = ('January February March April May June July August '
                   'September October November December').split()
    long_mon2num = dict(map(reversed, enumerate(long_months, 1)))

    if match1:
        day = match1.group('day') or '01'
        print('{}-{:02d}-{:02d}'.format(
            match1.group('year'), int(match1.group('month')), int(day)))

    elif match2:
        month = int(match2.group('month'))
        year = int(match2.group('year'))
        print('20{:02d}-{:02d}-01'.format(year, month))

    elif match3:
        month = match3.group('month')
        year = match3.group('year')
        month_num = short_mon2num[
            month] if month in short_mon2num else long_mon2num[month]
        print('{}-{:02d}-01'.format(year, month_num))

    else:
        print('No match')


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

Chapter 54: 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

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, 1
```

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., use this SQL:

~~~
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;
~~~

And you should see:

~~~
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 forei

````
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 wha

Obviously we're not going to manually enter our data by hand, so let's write a script to imp

First we're going to need to get our data, so do `make data` to download some TSV files fron
```

$ ./load_centrifuge.py *.tsv 1: Importing "YELLOWSTONE_SMPL_20717" (2) Loading "Synechococcus sp. JA-3-3Ab" (321327) Loading "Synechococcus sp. JA-2-3B'a(2-13)" (321332) 2: Importing "YELLOWSTONE_SMPL_20719" (3) Loading "Streptococcus suis" (1307) Loading "synthetic construct" (32630) 3: Importing "YELLOWSTONE_SMPL_20721" (4) Loading "Staphylococcus sp. AntiMn-1" (1715860) 4: Importing "YELLOWSTONE_SMPL_20723" (5) 5: Importing "YELLOWSTONE_SMPL_20725" (6) 6: Importing "YELLOWSTONE_SMPL_20727" (7) Done ""'

Here is the code that does that:

```
#!/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'])
    if not sort_order in sorting:
        print('"{}" not an allowed --sortorder, choose from {}'.format(
            order_by, ', '.join(sorting)))
        sys.exit(1)

    tax_ids = []
    for tax_id in re.split(r'\s*,\s*', args.taxid):
        if re.match(r'^\d+$', tax_id):
            tax_ids.append(tax_id)
        else:
            print('"{}" does not look like an NCBI tax id'.format(tax_id))

    if len(tax_ids) == 0:
        print('No tax ids')
        sys.exit(1)

    db = sqlite3.connect(dbname)
    cur = db.cursor()
    sql = """
        select   s.sample_name, t.tax_name, s2t.num_reads, s2t.abundance
        from     sample s, tax t, sample_to_tax s2t
        where    s.sample_id=s2t.sample_id
        and      s2t.tax_id=t.tax_id
        and      t.ncbi_id in ({})
        order by {} {}
    """.format(', '.join(tax_ids), order_by, sort_order)

    cur.execute(sql)

    samples = cur.fetchall()
    if len(samples) > 0:
        cols = [d[0] for d in cur.description]
        print(tabulate(samples, headers=cols))
    else:
        print('No results')

# --------------------------------------------------
if __name__ == '__main__':
    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 55: Finding Longhurst Province Using GeoJSON

Using shapes

Solution

```
#!/usr/bin/env python -u

import csv
import json
from shapely.geometry import shape, Point

# load GeoJSON file containing sectors
with open('longhurst.json', 'r') as f:
    js = json.load(f)

with open('lat-lon.tab', 'r') as f:
    reader = csv.reader(f, delimiter='\t')
    for sample_id,latitude,longitude in reader:
        if not latitude or not longitude:
            continue

        point = Point(float(longitude), float(latitude))

        # check each polygon to see if it contains the point
        for feature in js['features']:
            polygon = shape(feature['geometry'])
            if polygon.contains(point):
                print sample_id, feature['properties']['ProvCode']
```

Chapter 56: Finding K-mers in Text

```
$ ./kmer_tiler.py foobar
There are 4 3-mers in "foobar."
foobar
foo
 oob
  oba
   bar
```

Solution

```
#!/usr/bin/env python3

import os
import sys

args = sys.argv[1:]

if not 1 <= len(args) <= 2:
    print('Usage: {} WORD [SIZE]'.format(os.path.basename(sys.argv[0])))
    sys.exit(1)

word = args[0]
size = int(args[1]) if len(args) == 2 and args[1].isdigit() else 3
nkmer = len(word) - size + 1
verb = 'is' if nkmer == 1 else 'are'
plural = '' if nkmer == 1 else 's'

print('There {} {} {}-mer{} in "{}."'.format(verb, nkmer if nkmer > 0 else 0, size,

if nkmer > 0:
    print(word)
    for i in range(nkmer):
        print(' ' * i + word[i:i+size])
```

Chapter 57: Using De Bruijn Graphs to Assemble Sequences

We will find paths through sequences that could aid in assembly (cf http://rosalind.info/problems/grph/). For this exercise, we will only attempt to join any two sequences together. To do this, we will look at the last `k` characters of every sequence and find where the first `k` character of a *different* sequence are the same.

For example, in this file:

```
$ cat sample1.fa
>Rosalind_0498
AAATAAA
>Rosalind_2391
AAATTTT
>Rosalind_2323
TTTTCCC
>Rosalind_0442
AAATCCC
>Rosalind_5013
GGGTGGG
```

If `k` is 3, then the last 3-mer of sequence 498 is “AAA” which is also the first 3-mer of 2391 and 442. “TTT” ends 2391 and starts 2323, so the graphs we could create from 3-mers would be:

```
$ ./grph.py -k 3 sample1.fa
Rosalind_0498 Rosalind_2391
Rosalind_0498 Rosalind_0442
Rosalind_2391 Rosalind_2323
```

You will write a Python program called `grph.py` which will take a `-k|--overlap` option with a default value of `3` and a single positional argument of an input file which will be in FASTA format. I would recommend you read all the sequences and build two data structures that hold the k-mers at the beginnings and ends of your sequences. You should go through all the ending kmers and see if there are any sequences that begin with that string. It does not matter what order you emit the pairs as they will be sorted on the command line for the tests.

Solution

```
#!/usr/bin/env python3
"""
Author : Ken Youens-Clark <kyclark@email.arizona.edu>
Date   : 2019-04-08
Purpose: Graph through sequences
"""

import argparse
import logging
import os
import sys
from collections import defaultdict
from Bio import SeqIO


# --------------------------------------------------
def get_args():
    """get command-line arguments"""
    parser = argparse.ArgumentParser(
        description='Graph through sequences',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('file', metavar='str', help='FASTA file')

    parser.add_argument(
        '-k',
        '--overlap',
        help='K size of overlap',
        metavar='int',
        type=int,
        default=3)

    parser.add_argument(
        '-d', '--debug', help='Debug', 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 find_kmers(seq, k):
    """Find k-mers in string"""
    seq = str(seq)
    n = len(seq) - k + 1
    return list(map(lambda i: seq[i:i + k], range(n)))


# --------------------------------------------------
def main():
    """Make a jazz noise here"""
    args = get_args()
    file = args.file
    k = args.overlap

    if not os.path.isfile(file):
        die('"{}" is not a file'.format(file))

    if k < 1:
        die('-k "{}" must be a positive integer'.format(k))

    logging.basicConfig(
        filename='.log',
        filemode='w',
        level=logging.DEBUG if args.debug else logging.CRITICAL
    )

    beginning = defaultdict(list)
    end = defaultdict(list)
    for rec in SeqIO.parse(file, 'fasta'):
        kmers = find_kmers(rec.seq, k)
        beginning[kmers[0]].append(rec.id)
        end[kmers[-1]].append(rec.id)

    logging.debug('beginnings = {}'.format(beginning))
    logging.debug('ends = {}'.format(end))

    for kmer in end:
        if kmer in beginning:
            for seq_id in end[kmer]:
                for other in beginning[kmer]:
                    if seq_id != other:
                        print(seq_id, other)


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

Chapter 58: Find Sequences With Point Mutations (SNPs)

Write a Python program called `commoner.py` that takes exactly two positional arguments which should be text files that you will read and find words that are found to be in common. The program should also accept a `-m|--min_len` option (integer) which is the minimum length for a word to be included (so that we can avoid common short words like articles and "I", etc.) as well as a `-n|--hamming_distance` (integer) value that is the maximum allowed Hamming (edit) distance to consider two words to be the same. There should also be two options for debugging, one `-d|--debug` that turns on logging into a `-l|--logfile` option that defaults to `.log`. Lastly, the program should have a `-t|--table` option that indicates the output should be formatted into an ASCII table using the `tabulate` module (https://pypi.org/project/tabulate/); the default output (that is, without `-t`) should be tab-delimited text.

If there are no words found to be in common, print "No words in common." If there are words, print a header line with "word1," "word2", and "distance" as the column names. Then print each of the words, sorted by the pairs, along with their Hamming distances.

If either of the file inputs are not files, exit with an error and appropriate message. You could use the file handle `type` to `argparse` for the two file inputs as the test does not check for a specific error message. You also do not need to log the names of the input files, so there's that.

If the `--distance` is less than 0, exit with an error and message `--distance "{}" must be > 0`.

Logging

The logging tests do not look for specific messages, only that a non-empty log is created using a given name when `--debug` is present. You can use the same logging code from earlier assignments.

```
$ ./commoner.py
usage: commoner.py [-h] [-m int] [-n int] [-l str] [-d] [-t] FILE FILE
commoner.py: error: the following arguments are required: FILE
$ ./commoner.py -h
usage: commoner.py [-h] [-m int] [-n int] [-l str] [-d] [-t] FILE FILE

Find common words

positional arguments:
  FILE                  Input files
```

```
optional arguments:
  -h, --help            show this help message and exit
  -m int, --min_len int
                        Minimum length of words (default: 0)
  -n int, --hamming_distance int
                        Allowed Hamming distance (default: 0)
  -l str, --logfile str
                        Logfile name (default: .log)
  -d, --debug           Debug (default: False)
  -t, --table           Table output (default: False)
$ ./commoner.py data/fox.txt data/fox.txt
word1   word2   distance
brown   brown   0
dog dog 0
fox fox 0
jumps   jumps   0
lazy    lazy    0
over    over    0
quick   quick   0
the the 0
$ ./commoner.py data/fox.txt data/fox.txt -t
+---------+---------+------------+
| word1   | word2   |   distance |
|---------+---------+------------|
| brown   | brown   |          0 |
| dog     | dog     |          0 |
| fox     | fox     |          0 |
| jumps   | jumps   |          0 |
| lazy    | lazy    |          0 |
| over    | over    |          0 |
| quick   | quick   |          0 |
| the     | the     |          0 |
+---------+---------+------------+
$ ./commoner.py -t data/american.txt data/british.txt -m 5 -n 1
+-----------+-----------+------------+
| word1     | word2     |   distance |
|-----------+-----------+------------|
| about     | about     |          0 |
| analyze   | analyse   |          1 |
| faults    | faults    |          0 |
| forgot    | forgot    |          0 |
| generally | generally |          0 |
| improve   | improve   |          0 |
| license   | licence   |          1 |
| merits    | merits    |          0 |
| night     | night     |          0 |
| organize  | organise  |          1 |
| ourselves | ourselves |          0 |
| pretense  | pretence  |          1 |
| recognize | recognise |          1 |
| thoughts  | thoughts  |          0 |
| which     | which     |          0 |
| without   | without   |          0 |
+-----------+-----------+------------+
$ ./commoner.py data/american.txt data/british.txt -m 5 -n 1 | column -t
word1      word2      distance
about      about      0
analyze    analyse    1
faults     faults     0
forgot     forgot     0
generally  generally  0
improve    improve    0
license    licence    1
merits     merits     0
night      night      0
organize   organise   1
ourselves  ourselves  0
pretense   pretence   1
recognize  recognise  1
thoughts   thoughts   0
which      which      0
without    without    0
```

Testing

This test suite is going to mix unit tests *inside* your `commoner.py` program with integration tests in `test.py`. You will need to copy your `dist` function from `13-hamm` and add this (probably just after the `dist` function):

```
def test_dist():
    """dist ok"""

    tests = [('foo', 'boo', 1), ('foo', 'faa', 2), ('foo', 'foobar', 3),
             ('TAGGGCAATCATCCGAG', 'ACCGTCAGTAATGCTAC',
              9), ('TAGGGCAATCATCCGG', 'ACCGTCAGTAATGCTAC', 10)]

    for s1, s2, n in tests:
        d = dist(s1, s2)
        assert d == n
```

You will also need to define a function `def uniq_words(file, min_len):` that takes a file – or open file handle! – and a minimum length. Paste this test below your function definition:

```
def test_uniq_words():
    """Test uniq_words"""

    s1 = '?foo, "bar", FOO: $fa,'
    s2 = '%Apple.; -Pear. ;bANAna!!!'

    assert uniq_words(io.StringIO(s1), 0) == set(['foo', 'bar', 'fa'])

    assert uniq_words(io.StringIO(s1), 3) == set(['foo', 'bar'])

    assert uniq_words(io.StringIO(s2), 0) == set(['apple', 'pear', 'banana'])

    assert uniq_words(io.StringIO(s2), 4) == set(['apple', 'pear', 'banana'])

    assert uniq_words(io.StringIO(s2), 5) == set(['apple', 'banana'])
```

Note that this test is mocking the idea of a file handle; that is, the source for the words will be a file(handle), but for purposes of the test I just want to pass something that can pretend to be a filehandle. Notice how we can use a `for` loop over an `io.String` just like we can an `open` file:

```
>>> import io
>>> file = io.StringIO('foo\nbar baz\nquux!')
>>> for i, line in enumerate(file):
...   print(i, line, end='')
...
0 foo
1 bar baz
2 quux!
```

Lastly define a `def common(words1, words2, distance):` function that takes two lists of words and a maximum Hamming distance and returns a list of tuples containing the two words and the actual distance between the two words if that distance is less than or equal to the maximum allowed. Copy this function just below it.

```
def test_common():
    w1 = ['foo', 'bar', 'quux']
    w2 = ['bar', 'baz', 'faa']

    assert common(w1, w2, 0) == [('bar', 'bar', 0)]

    assert common(w1, w2, 1) == [('bar', 'bar', 0), ('bar', 'baz', 1)]

    assert common(w1, w2, 2) == [('bar', 'bar', 0), ('bar', 'baz', 1),
                                 ('bar', 'faa', 2), ('foo', 'faa', 2)]
```

Once you have written the above functions, I don't think it's helping too much to show you my logic:

```
words1 = uniq_words(fh1, args.min_len)
words2 = uniq_words(fh2, args.min_len)
common_words = common(words1, words2, distance)
```

I think it would help you to think about first getting a unique set of words of the correct length from each file. Then use those words to find the ones in common. Were we not concerned about the Hamming distance, we could use a `set` for the words and do `words1.intersection(words2)`, but we have to instead to a pair-wise comparison of every `word1` to every `word2`! That is most easily accomplished by using `itertools.product`.

Test Suite

The Makefile's `test` target is `pytest -v commoner.py test.py`. Notice how it's looking in both your `commoner.py` program for `test_` functions as well as the `test.py`. Again, the point here is to build small, testable functions inside your program and integrate the tests directly into the program. Then `test.py` is used to ensure that the *user interface* works; that is, your program generates a usage, it emits error codes on errors, it honors the expected flags and arguments, etc.

Solution

```
#!/usr/bin/env python3
"""
Author : Ken Youens-Clark <kyclark@gmail.com>
Date   : 2019-04-18
Purpose: Find common words
"""

import argparse
import io
import logging
import re
import sys
from itertools import product
from tabulate import tabulate


# --------------------------------------------------
def get_args():
    """get command-line arguments"""
    parser = argparse.ArgumentParser(
        description='Find common words',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument(
        'file',
        metavar='FILE',
        help='Input files',
        nargs=2,
        type=argparse.FileType('r', encoding='UTF-8'))

    parser.add_argument(
        '-m',
        '--min_len',
        help='Minimum length of words',
        metavar='int',
        type=int,
        default=0)

    parser.add_argument(
        '-n',
        '--hamming_distance',
        help='Allowed Hamming distance',
        metavar='int',
        type=int,
        default=0)

    parser.add_argument(
        '-l',
        '--logfile',
        help='Logfile name',
        metavar='str',
        type=str,
        default='.log')

    parser.add_argument('-d', '--debug', help='Debug', action='store_true')

    parser.add_argument(
        '-t', '--table', help='Table output', 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 dist(s1, s2):
    """Given two strings, return the Hamming distance (int)"""

    d = abs(len(s1) - len(s2)) + sum(
        map(lambda p: 0 if p[0] == p[1] else 1, zip(s1, s2)))

    logging.debug('s1 = {}, s2 = {}, d = {}'.format(s1, s2, d))

    return d


# --------------------------------------------------
def test_dist():
    """dist ok"""

    tests = [('foo', 'boo', 1), ('foo', 'faa', 2), ('foo', 'foobar', 3),
             ('TAGGGCAATCATCCGAG', 'ACCGTCAGTAATGCTAC',
              9), ('TAGGGCAATCATCCGG', 'ACCGTCAGTAATGCTAC', 10)]

    for s1, s2, n in tests:
        d = dist(s1, s2)
        assert d == n


# --------------------------------------------------
def uniq_words(file, min_len):
    """
    Given a file or filehandle, return a set of the unique words
    over a given minimum length
    """
    words = set()
    fh = open(file) if type(file) == str else file

    for line in fh:
        for word in line.lower().split():
            word = re.sub('[^a-zA-Z0-9]', '', word)
            if len(word) >= min_len:
                words.add(word)

    return words


# --------------------------------------------------
def test_uniq_words():
    """Test uniq_words"""

    s1 = '?foo, "bar", FOO: $fa,'
    s2 = '%Apple.; -Pear. ;bANAna!!!'

    assert uniq_words(io.StringIO(s1), 0) == set(['foo', 'bar', 'fa'])

    assert uniq_words(io.StringIO(s1), 3) == set(['foo', 'bar'])

    assert uniq_words(io.StringIO(s2), 0) == set(['apple', 'pear', 'banana'])

    assert uniq_words(io.StringIO(s2), 4) == set(['apple', 'pear', 'banana'])

    assert uniq_words(io.StringIO(s2), 5) == set(['apple', 'banana'])

```

```

# --------------------------------------------------
def common(words1, words2, distance):
    """Find the common words"""

    words = []
    for w1, w2 in sorted(product(words1, words2)):
        hamm = dist(w1, w2)
        if hamm <= distance:
            words.append((w1, w2, hamm))

    return words


# --------------------------------------------------
def test_common():
    w1 = ['foo', 'bar', 'quux']
    w2 = ['bar', 'baz', 'faa']

    assert common(w1, w2, 0) == [('bar', 'bar', 0)]

    assert common(w1, w2, 1) == [('bar', 'bar', 0), ('bar', 'baz', 1)]

    assert common(w1, w2, 2) == [('bar', 'bar', 0), ('bar', 'baz', 1),
                                 ('bar', 'faa', 2), ('foo', 'faa', 2)]


# --------------------------------------------------
def main():
    """Make a jazz noise here"""

    args = get_args()
    fh1, fh2 = args.file
    distance = args.hamming_distance

    if distance < 0:
        die('--distance "{}" must be > 0'.format(distance))

    logging.basicConfig(
        filename=args.logfile,
        filemode='w',
        level=logging.DEBUG if args.debug else logging.CRITICAL)

    words1 = uniq_words(fh1, args.min_len)
    words2 = uniq_words(fh2, args.min_len)
    common_words = common(words1, words2, distance)

    logging.debug('Found {} words in common'.format(len(common_words)))

    if not common_words:
        print('No words in common.')
    else:
        common_words.insert(0, ('word1', 'word2', 'distance'))
        if args.table:
            print(tabulate(common_words, headers='firstrow', tablefmt='psql'))
        else:
            for w1, w2, hamm in common_words:
                print('\t'.join([w1, w2, str(hamm)]))


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

Chapter 59: Sequence Similarity Using Shared k-mers

Another way to explore sequence similarity.

Solution

```
#!/usr/bin/env python3
"""Show shared kmers"""

import os
import sys
from collections import Counter

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

    if not 1 <= len(args) <= 3:
        print('Usage: {} WORD1 WORD2 [SIZE]'.format(os.path.basename(sys.argv[0])))
        sys.exit(1)

    word1 = args[0]
    word2 = args[1]
    len1 = len(word1)
    len2 = len(word2)
    size = int(args[2]) if len(args) == 3 and args[2].isdigit() else 3

    kmers1 = kmers(word1, size)
    kmers2 = kmers(word2, size)
    set1 = set(kmers1.keys())
    set2 = set(kmers2.keys())
    shared = set1.intersection(set2)
    num_shared = len(shared)
    plural = '' if num_shared == 1 else 's'
    msg = '"{}" and "{}" share {} {}-mer{}.'

    print(msg.format(word1, word2, num_shared, size, plural))

    if num_shared > 0:
        fmt = '{:' + str(size + 1) + '} {:>5} {:>5} {:>5} {:>5}'
        print(fmt.format('kmer', '#1', '%1', '#2', '%2'))
        print('-' * 50)
        t1, t2 = 0, 0
        for kmer in shared:
            n1 = kmers1[kmer]
            n2 = kmers2[kmer]
            t1 += n1
            t2 += n2
            p1 = int(n1 / len1 * 100)
            p2 = int(n2 / len2 * 100)
            print(fmt.format(kmer, n1, p1, n2, p2))
        print(fmt.format('tot', t1, int(t1/len1*100), t2, int(t2/len2*100)))

# --------------------------------------------------
def kmers(word, size):
    nkmer = len(word) - size + 1
    return Counter([word[i:i+size] for i in range(nkmer)])

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

Chapter 60: Species Abundance Bubble Plot

Centrifuge is a program that will make taxonomic assignments to short DNA reads. Write a program called `plot.py` that will read the `.tsv` output file from Centrifuge that gives a summary of the species and abundance for a given sample. The program should take the output directory containing a number of samples and use `matplotlib` to create a bubble plot showing the abundance of taxa at various `-r|--rank` assignments.

```
$ ./plot.py
usage: plot.py [-h] [-r str] [-m float] [-M float] [-x str] [-t str] [-o str]
               DIR
plot.py: error: the following arguments are required: DIR
$ ./plot.py -h
usage: plot.py [-h] [-r str] [-m float] [-M float] [-x str] [-t str] [-o str]
               DIR

Plot Centrifuge out

positional arguments:
  DIR                   Centrifuge output directory

optional arguments:
  -h, --help            show this help message and exit
  -r str, --rank str    Tax rank (default: species)
  -m float, --min float
                        Minimum percent abundance (default: 0.0)
  -M float, --multiplier float
                        Multiply abundance (default: 1.0)
  -x str, --exclude str
                        Tax IDs or names to exclude (default: )
  -t str, --title str   Figure title (default: )
  -o str, --outfile str
                        Output file (default: bubble.png)
```

Solution

```
#!/usr/bin/env python3
"""
Author : Ken Youens-Clark <kyclark@email.arizona.edu>
Date   : 2019-06-11
Purpose: Plot Centrifuge out
"""

import argparse
import csv
import os
import re
import pandas as pd
import matplotlib.pyplot as plt
from dire import die

# --------------------------------------------------
def get_args():
    """Get command-line arguments"""

    parser = argparse.ArgumentParser(
        description='Plot Centrifuge out',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('dir',
                        metavar='DIR',
                        type=str,
                        help='Centrifuge output directory')

    parser.add_argument('-r',
                        '--rank',
                        help='Tax rank',
                        metavar='str',
                        type=str,
                        choices=['species'],
                        default='species')

    parser.add_argument('-m',
                        '--min',
                        help='Minimum percent abundance',
                        metavar='float',
                        type=float,
                        default=0.)

    parser.add_argument('-M',
                        '--multiplier',
                        help='Multiply abundance',
                        metavar='float',
                        type=float,
                        default=1.)

    parser.add_argument('-x',
                        '--exclude',
                        help='Tax IDs or names to exclude',
                        metavar='str',
                        type=str,
                        default='')

    parser.add_argument('-t',
                        '--title',
                        help='Figure title',
                        metavar='str',
                        type=str,
                        default='')

    parser.add_argument('-o',
                        '--outfile',
                        help='Output file',
                        metavar='str',
                        type=str,
                        default='bubble.png')

    return parser.parse_args()


# --------------------------------------------------
def main():
    """Make a jazz noise here"""

    args = get_args()
    rank = args.rank
    min_pct = args.min
    exclude = re.split('\s*,\s*', args.exclude.lower())
    cent_dir = args.dir

    if not os.path.isdir(cent_dir):
        die('"{}" is not a directory'.format(cent_dir))

    tsv_files = list(filter(lambda f: f.endswith('.tsv'), os.listdir(cent_dir)))
    if not tsv_files:
        die('Found no ".tsv" files in "{}"'.format(cent_dir))

    assigned = []
    for i, file in enumerate(tsv_files, start=1):
        print('{:3}: {}'.format(i, file))

        with open(os.path.join(cent_dir, file)) as fh:
            reader = csv.DictReader(fh, delimiter='\t')
            for rec in filter(lambda r: r['taxRank'] == rank, reader):
                tax_id = rec['taxID']
                tax_name = rec['name']

                if tax_id in exclude or tax_name.lower() in exclude:
                    continue

                pct = float(rec.get('abundance'))
                if min_pct and pct < min_pct:
                    continue

                sample, _ = os.path.splitext(file)
                assigned.append({
                    'sample': sample,
                    'tax_id': tax_id,
                    'tax_name': tax_name,
                    'pct': pct,
                    'reads': int(rec['numReads'])
                })

    if not assigned:
        die('No data!')

    df = pd.DataFrame(assigned)
    plt.scatter(x=df['sample'],
                y=df['tax_name'],
                s=df['pct'] * args.multiplier,
                alpha=0.5)
    plt.xticks(rotation=45, ha='right')
    plt.yticks(rotation=45, ha='right')
    plt.gcf().subplots_adjust(bottom=.4, left=.3)
    plt.ylabel('Organism')
    plt.xlabel('Sample')
    if args.title:
        plt.title(args.title)

    plt.savefig(args.outfile)

    print('Done, see "{}"'.format(args.outfile))


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

Chapter 61: Writing Pipelines in Python

> Falling in love with code means falling in love with problem solving and being a part of a forever ongoing conversation. – Kathryn Barrett

You might be surprised at how far you can push humble `make` to write analysis pipelines. I'd encourage you to really explore Makefiles, reading the docs and looking at other people's examples. You'll save yourself many hours if you learn to use `make` well, even if you are just documenting how you ran your Python program. Beyond `make`, there are many other frameworks for writing pipelines such as Nextflow, Snakemake, Taverna, Pegasus and many more (cf https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5429012/), many of which are probably far superior to rolling your own in Python; however, we will do just that as you will learn many valuable skills along the way. After all, hubris is one of the three virtues of a great programmer:

> According to Larry Wall, the original author of the Perl programming language, there are three great virtues of a programmer:
>
> **Laziness**: The quality that makes you go to great effort to reduce overall energy expenditure. It makes you write labor-saving programs that other people will find useful and document what you wrote so you don't have to answer so many questions about it.
>
> **Impatience**: The anger you feel when the computer is being lazy. This makes you write programs that don't just react to your needs, but actually anticipate them. Or at least pretend to.
>
> **Hubris**: The quality that makes you write (and maintain) programs that other people won't want to say bad things about.
>
> *Programming Perl*, 2nd Edition, O'Reilly & Associates, 1996

A "pipeline" is chaining the output of one program or function as the input to the next as many times as necessary to arrive at an end product. Sometimes the whole pipeline can be written inside Python, but often in bioinformatics what we have is one program written in Java/C/C++ we install from source that creates some output that needs to be massaged by a program we write in bash or Python that gets fed to a Perl script you found on BioStars that produces some text file that we read into R to create some visualization. We're going to focus on how to use Python to take input, call external programs, check on the status, and feed the output to some other program.

Hello

In this first example, we'll pretend this "hello.sh" is something more interesting than it really is:

```
$ cat -n hello.sh
     1	#!/usr/bin/env bash
     2	
     3	if [[ $# -lt 1 ]]; then
     4	    printf "Usage: %s NAME\n" $(basename $0)
     5	    exit 1
     6	fi
     7	
     8	NAME=$1
     9	
    10	if [[ $NAME == 'Lord Voldemort' ]]; then
    11	    echo "Upon advice of my counsel, I respectfully refuse to say that name."
    12	    exit 1
    13	fi
    14	
    15	echo "Hello, $1!"
$ ./hello.sh
Usage: hello.sh NAME
$ ./hello.sh Jan
Hello, Jan!
$ ./hello.sh "Lord Voldemort"
Upon advice of my counsel, I respectfully refuse to say that name.
```

We'll write a Python program to feed names to the "hello.sh" program and monitor whether the program ran successfully.

```
$ cat -n run_hello.py
     1	#!/usr/bin/env python3
     2	"""
     3	Author : kyclark
     4	Date   : 2019-03-28
     5	Purpose: Run "hello.sh"
     6	"""
     7	
     8	import argparse
     9	import os
    10	import sys
    11	from subprocess import getstatusoutput
    12	
    13	
    14	# --------------------------------------------------
    15	def get_args():
    16	    """get command-line arguments"""
    17	    parser = argparse.ArgumentParser(
    18	        description='Simple pipeline',
    19	        formatter_class=argparse.ArgumentDefaultsHelpFormatter)
    20	
    parser.add_argument(
        'name', metavar='str', nargs='+', help='Names for hello.sh')

    parser.add_argument(
        '-p',
        '--program',
        help='Program to run',
        metavar='str',
        type=str,
        default='./hello.sh')

    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()
    prg = args.program

    if not os.path.isfile(prg):
        die('Missing expected program "{}"'.format(prg))

    for name in args.name:
        cmd = '{} "{}"'.format(prg, name)
        rv, out = getstatusoutput(cmd)
        if rv != 0:
            warn('Failed to run: {}\nError: {}'.format(cmd, out))
        else:
            print('Success: "{}"'.format(out))

    print('Done.')

```

```
67
68  # --------------------------------------------------
69  if __name__ == '__main__':
70      main()
```

In get_args we establish that we expect one or more positional arguments on the command line along with an optional -p|--program to run with those as arguments. One of the first items to check is if the program exists (we are expecting a full path with ./hello.sh being the default), so line 54 checks this and calls die if it does not exist.

The main event starts on line 57 where we loop through the name arguments. On line 58, we create a command by making a string with the name of the program and the argument. Then we use subprocess.getstatusoutput to run this command and give us the return value (rv) and the output from the command (both STDERR and STDOUT get combined). If the return value is not zero ("zero errors"), then we use warn to report on STDERR that there was a failure, else we print "Success" along with the output from hello.sh.

If we run this, we see it stops when given a bad program:

```
$ ./run_hello.py -p foo Ken
Missing expected program "foo"
```

And we see it correctly reports the results for our inputs:

```
$ ./run_hello.py Jan Marcia "Lord Voldemort" Cindy
Success: "Hello, Jan!"
Success: "Hello, Marcia!"
Failed to run: ./hello.sh "Lord Voldemort"
Error: Upon advice of my counsel, I respectfully refuse to say that name.
Success: "Hello, Cindy!"
Done.
```

If you were submitting this job to run on an HPC, it would be launched by the job scheduler sometime later than when you submit it and would be run in an automated fashion. You would quickly learn that it's better to capture errors to an error file rather than let them comingle with STDOUT.

```
$ ./run_hello.py Jan Marcia "Lord Voldemort" Cindy 2>err
Success: "Hello, Jan!"
Success: "Hello, Marcia!"
Success: "Hello, Cindy!"
Done.
$ cat err
Failed to run: ./hello.sh "Lord Voldemort"
Error: Upon advice of my counsel, I respectfully refuse to say that name.
```

Parallel Hello

This works fairly well, but what if there are potentially dozens, hundreds, or thousands of names to greet? We are processing these in a serial fashion, but it's common that even laptops have more than one CPU that could we could use. Even with just 2 CPUs, we'd accomplish the task 2X faster than using just one. It's common to have 60-90 CPUs (or "cores") on HPC machines. If you aren't using them, you're wasting time!

The GNU `parallel` program (https://www.gnu.org/software/parallel/) provides a simple way to use more than one CPU to complete a batch of jobs. It takes as input the commands that need to be run and spins them out to all available CPUs (or as many as you limit it to), watching for jobs that fail, starting up new jobs when older ones finish.

To see it in action, let's compare these two programs in the "examples/gnu_parallel" directory. The first one simply prints the number 1-30 in order:

```
$ cat -n run.sh
     1	#!/usr/bin/env bash
     2
     3	for i in $(seq 1 30); do
     4	    echo $i
     5	done
     6
     7	echo "Done."
```

The second one uses `parallel` to print them. While this is a trivial case, imagine something more intense like BLAST jobs.

```
$ cat -n run_parallel.sh
     1	#!/usr/bin/env bash
     2
     3	JOBS=$(mktemp)
     4
     5	for i in $(seq 1 30); do
     6	    echo "echo $i" >> "$JOBS"
     7	done
     8
     9	NUM_JOBS=$(wc -l "$JOBS" | awk '{print $1}')
    10
    11	if [[ $NUM_JOBS -gt 0 ]]; then
    12	    echo "Running $NUM_JOBS jobs"
    13	    parallel -j 8 --halt soon,fail=1 < "$JOBS"
    14	fi
    15
    16	[[ -f "$JOBS" ]] && rm "$JOBS"
```

```

echo "Done."
```

And here is they look like when they are run:

```
$ ./run.sh
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
Done.
$ ./run_parallel.sh
Running 30 jobs
7
9
8
10
11
12
13
14
```

```
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
6
5
30
4
3
2
1
Done.
```

The `parallel` version looks out of order because the jobs are run as quickly as possible in whatever order that happens.

Chapter 62: BLAST Pipeline

Everyone needs a thneed.

Chapter 63: CD-HIT Pipeline

Let's take the `cd-hit` cluster exercise and extend it to where we take the proteins FASTA, run cd-hit, and find the unclustered proteins all in one go. First things first, we need to ensure `cd-hit` is on our system. It's highly unlikely that it is, so let's figure out how to install it.

If you search on the Internet for `cd-hit`, you might end up at http://weizhongli-lab.org/cd-hit/ from which you go to the download page (http://weizhongli-lab.org/cd-hit/download.php) which directs you to the GitHub releases for the `cd-hit` repository (https://github.com/weizhongli/cdhit/releases). From there, we can download the source code tarball (`.tar.gz` file). For instance, I right-click on the link to copy the line address, then go to my HPC into my "downloads" directory and then use `wget` to retrieve the tarball. Next use `tar xvf` to "extract" in a "verbose" fashion the "file" (followed by the tarball). Finally you should have a directory like `cd-hit-v4.8.1-2019-0228` into which you should `cd`.

If you look at the README, you'll see the way to compile this is to just type `make`. On my Mac laptop, I needed to compile without multi-threading support, so I used `make openmp=no`. That will run for a few seconds and look something like this:

```
$ make openmp=no
g++  -DNO_OPENMP -DWITH_ZLIB -O2  cdhit-common.c++ -c
g++  -DNO_OPENMP -DWITH_ZLIB -O2  cdhit-utility.c++ -c
g++  -DNO_OPENMP -DWITH_ZLIB -O2  cdhit.c++ -c
g++  -DNO_OPENMP -DWITH_ZLIB -O2  cdhit.o cdhit-common.o cdhit-utility.o -lz -o cd-hit
g++  -DNO_OPENMP -DWITH_ZLIB -O2  cdhit-est.c++ -c
g++  -DNO_OPENMP -DWITH_ZLIB -O2  cdhit-est.o cdhit-common.o cdhit-utility.o -lz -o cd-hit-e
g++  -DNO_OPENMP -DWITH_ZLIB -O2  cdhit-2d.c++ -c
g++  -DNO_OPENMP -DWITH_ZLIB -O2  cdhit-2d.o cdhit-common.o cdhit-utility.o -lz -o cd-hit-2d
g++  -DNO_OPENMP -DWITH_ZLIB -O2  cdhit-est-2d.c++ -c
g++  -DNO_OPENMP -DWITH_ZLIB -O2  cdhit-est-2d.o cdhit-common.o cdhit-utility.o -lz -o cd-hi
g++  -DNO_OPENMP -DWITH_ZLIB -O2  cdhit-div.c++ -c
g++  -DNO_OPENMP -DWITH_ZLIB -O2  cdhit-div.o cdhit-common.o cdhit-utility.o -lz -o cd-hit-d
g++  -DNO_OPENMP -DWITH_ZLIB -O2  cdhit-454.c++ -c
g++  -DNO_OPENMP -DWITH_ZLIB -O2  cdhit-454.o cdhit-common.o cdhit-utility.o -lz -o cd-hit-4
```

Often Makefiles will include an `install` target that will copy the new programs into a directory like `/usr/local/bin`. This one does not, so you'll have to manually copy the programs (e.g., `cd-hit`, `cd-hit-2d`, etc.) to whatever location you like. On an HPC (like Ocelote), you will not have permissions to copy to `/usr/local/bin`, so I'd recommend you create a directory like `$HOME/.local/bin` which you add to your `$PATH` and copy the binaries to that location.

Ensure you have a `cd-hit` binary you can use:

```
$ which cd-hit
/Users/kyclark/.local/bin/cd-hit
$ cd-hit -h | head
        ====== CD-HIT version 4.8.1 (built on Apr  9 2019) ======

Usage: cd-hit [Options]

Options

   -i   input filename in fasta format, required, can be in .gz format
   -o   output filename, required
   -c   sequence identity threshold, default 0.9
    this is the default cd-hit's "global sequence identity" calculated as:
```

Now we can try out our new code:

Solution

```
#!/usr/bin/env python3
"""
Author : Ken Youens-Clark <kyclark@gmail.com>
Date   : 2019-02-20
Purpose: Run cd-hit, find unclustered proteins
"""

import argparse
import datetime
import logging
import os
import re
import signal
import sys
from subprocess import getstatusoutput
from shutil import which
from Bio import SeqIO


# --------------------------------------------------
def get_args():
    """get command-line arguments"""
    parser = argparse.ArgumentParser(
        description='Run cd-hit, find unclustered proteins',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument(
        '-p',
        '--proteins',
        help='Proteins FASTA',
        metavar='str',
        type=str,
        required=True)

    parser.add_argument(
        '-c',
        '--seq_id_threshold',
        help='cd-hit Sequence identity threshold',
        metavar='float',
        type=float,
        default=0.9)

    parser.add_argument(
        '-o',
        '--outfile',
        help='Output file',
        metavar='str',
        type=str,
        default='unclustered.fa')

    parser.add_argument(
        '-l',
        '--logfile',
        help='Log file',
        metavar='str',
        type=str,
        default='.log')

    parser.add_argument('-d', '--debug', help='Debug', action='store_true')

    return parser.parse_args()


# --------------------------------------------------
def die(msg='Something bad happened'):
    """log a critical message() and exit with error"""
    logging.critical(msg)
    sys.exit(1)


# --------------------------------------------------
def run_cdhit(proteins_file, seq_id_threshold):
    """Run cd-hit"""
    cdhit = which('cd-hit')

    if not cdhit:
        die('Cannot find "cd-hit"')

    out_file = os.path.basename(proteins_file) + '.cdhit'
    out_path = os.path.join(os.path.dirname(proteins_file), out_file)

    logging.debug('Found cd-hit "{}"'.format(cdhit))
    cmd = '{} -c {} -i {} -o {}'.format(cdhit, seq_id_threshold,
                                        proteins_file, out_path)
    logging.debug('Running "{}"'.format(cmd))
    rv, out = getstatusoutput(cmd)

    if rv != 0:
        die('Non-zero ({}) return from "{}"\n{}\n'.format(rv, cmd, out))

    if not os.path.isfile(out_path):
        die('Failed to create "{}"'.format(out_path))

    logging.debug('Finished cd-hit, found cluster file "{}"'.format(out_path))

    return out_file


# --------------------------------------------------
def get_unclustered(cluster_file, proteins_file, out_file):
    """Find the unclustered proteins in the cd-hit output"""

    if not os.path.isfile(cluster_file):
        die('cdhit "{}" is not a file'.format(cluster_file))

    logging.debug('Parsing "{}"'.format(cluster_file))

    clustered = set([rec.id for rec in SeqIO.parse(cluster_file, 'fasta')])

    # Alternate (longer) way:
    # clustered = set()
    # for rec in SeqIO.parse(cluster_file, 'fasta'):
    #     clustered.add(rec.id)

    logging.debug('Will write to "{}"'.format(out_file))
    out_fh = open(out_file, 'wt')
    num_total = 0
    num_unclustered = 0

    for rec in SeqIO.parse(proteins_file, 'fasta'):
        num_total += 1
        prot_id = re.sub(r'\|.*', '', rec.id)
        if not prot_id in clustered:
            num_unclustered += 1
            SeqIO.write(rec, out_fh, 'fasta')

    logging.debug(
        'Finished writing unclustered proteins'.format(num_unclustered))

    return (num_unclustered, num_total)


# --------------------------------------------------
def main():
    """Make a jazz noise here"""
    args = get_args()
    proteins_file = args.proteins
    out_file = args.outfile
    log_file = args.logfile

    if not os.path.isfile(proteins_file):
        die('--proteins "{}" is not a file'.format(arg_name, proteins_file))

    logging.basicConfig(
        filename=log_file,
        filemode='a',
        level=logging.DEBUG if args.debug else logging.CRITICAL)

    def sigint(sig, frame):
        logging.critical('INT: Exiting early!')
        sys.exit(0)

    signal.signal(signal.SIGINT, sigint)

    banner = '#' * 50
    logging.debug(banner)
    logging.debug('BEGAN {}'.format(str(datetime.datetime.today())))

    cluster_file = run_cdhit(proteins_file, args.seq_id_threshold)
    num_unclustered, num_total = get_unclustered(cluster_file, proteins_file,
                                                  out_file)

    msg = 'Wrote {:,d} of {:,d} unclustered proteins to "{}"'.format(
        num_unclustered, num_total, out_file)

    print(msg)
    logging.debug(msg)
    logging.debug('FINISHED {}'.format(str(datetime.datetime.today())))
    logging.debug(banner)


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

Chapter 64: Centrifuge Pipeline in Python

Solution

```
#!/usr/bin/env python3
"""Run Centrifuge"""

import argparse
import os
import re
import subprocess
import sys
import tempfile as tmp


# --------------------------------------------------
def get_args():
    """Get command-line args"""

    parser = argparse.ArgumentParser(
        description='Argparse Python script',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument(
        '-q',
        '--query',
        help='File or directory of input',
        metavar='str',
        type=str,
        action='append',
        required=True)

    parser.add_argument(
        '-r',
        '--reads_are_paired',
        help='Expect forward/reverse (1/2) reads in --query',
        action='store_true')

    parser.add_argument(
        '-f',
        '--format',
        help='Input file format',
        metavar='str',
        type=str,
        default='')

    parser.add_argument(
        '-i',
        '--index',
        help='Centrifuge index name',
        metavar='str',
        type=str,
        default='p_compressed+h+v')

    parser.add_argument(
        '-I',
        '--index_dir',
        help='Centrifuge index directory',
        metavar='str',
        type=str,
        default='')

    parser.add_argument(
        '-o',
        '--out_dir',
        help='Output directory',
        metavar='str',
        type=str,
        default=os.path.join(os.getcwd(), 'centrifuge-out'))

    parser.add_argument(
        '-x',
        '--exclude_tax_ids',
        help='Comma-separated list of tax ids to exclude',
        metavar='str',
        type=str,
        default='')

    parser.add_argument(
        '-T',
        '--figure_title',
        help='Title for the bubble chart',
        metavar='str',
        type=str,
        default='Species abundance by sample')

    parser.add_argument(
        '-t',
        '--threads',
        help='Num of threads per instance of centrifuge',
        metavar='int',
        type=int,
        default=1)

    parser.add_argument(
        '-P',
        '--procs',
        help='Max number of processes to run',
        metavar='int',
        type=int,
        default=4)

    return parser.parse_args()


# --------------------------------------------------
def main():
    """Start here"""

    args = get_args()
    out_dir = args.out_dir
    index_dir = args.index_dir
    index_name = args.index
    file_format = args.format

    if not index_dir:
        print('--index_dir is required')
        sys.exit(1)

    if not index_name:
        print('--index_name is required')
        sys.exit(1)

    if not os.path.isdir(index_dir):
        die('--index_dir "{}" is not a directory'.format(index_dir))

    valid_index = set(
        map(lambda s: re.sub(r'\.\d+\.cf$', '', os.path.basename(s)),
            os.listdir(index_dir)))

    if not index_name in valid_index:
        tmpl = '--index "{}" is not valid, please choose from: {}'
        die(tmpl.format(index_name, ', '.join(sorted(valid_index))))

    if not os.path.isdir(out_dir):
        os.makedirs(out_dir)

    input_files = find_input_files(args.query, args.reads_are_paired)

    if not file_format:
        exts = set()
        for direction in input_files:
            for file in input_files[direction]:
                base = re.sub(r'\.gz$', '', os.path.basename(file))
                _, ext = os.path.splitext(base)
                exts.add(re.sub(r'^\.', '', ext))

        guesses = set()
        for ext in exts:
            if re.match(r'f(?:ast|n)?a', ext):
                guesses.add('fasta')
            elif re.match(r'f(?:ast)?q', ext):
                guesses.add('fastq')

        if len(guesses) == 1:
            file_format = guesses.pop()
        else:
            msg = 'Cannot guess file format ({}) from extentions ({})'
            die(msg.format(', '.join(guesses), ', '.join(exts)))

    valid_format = set(['fasta', 'fastq'])
    if not file_format in valid_format:
        msg = '--format "{}" is not valid, please choose from {}'
        die(msg.format(file_format, ', '.join(valid_format)))

    msg = 'Files found: forward = "{}", reverse = "{}", unpaired = "{}"'
    print(
        msg.format(
            len(input_files['forward']), len(input_files['reverse']),
            len(input_files['unpaired'])))

    reports_dir = run_centrifuge(
        file_format=file_format,
        files=input_files,
        out_dir=out_dir,
        exclude_tax_ids=args.exclude_tax_ids,
        index_dir=index_dir,
        index_name=index_name,
        threads=args.threads,
        procs=args.procs)

    fig_dir = make_bubble(
        reports_dir=reports_dir, out_dir=out_dir, title=args.figure_title)

    print('Done, reports in "{}", figures in "{}"'.format(
        reports_dir, fig_dir))


# --------------------------------------------------
def warn(msg):
    """Print a message to STDERR"""

    print(msg, file=sys.stderr)


# --------------------------------------------------
def die(msg='Something went wrong'):
    """Print a message to STDERR and exit with error"""

    warn('Error: {}'.format(msg))
    sys.exit(1)


# --------------------------------------------------
def unique_extensions(files):
    exts = set()
    for file in files:
        _, ext = os.path.splitext(file)
        exts.add(ext[1:])  # skip leading "."

    return exts


# --------------------------------------------------
def find_input_files(query, reads_are_paired):
    """Find input files from list of files/dirs"""

    files = []
    for qry in query:
        if os.path.isdir(qry):
            for filename in os.scandir(qry):
                if filename.is_file():
                    files.append(filename.path)
        elif os.path.isfile(qry):
            files.append(qry)
        else:
            die('--query "{}" neither file nor directory'.format(qry))

    files.sort()  # inplace

    forward = []
    reverse = []
    unpaired = []

    if reads_are_paired:
        extensions = unique_extensions(files)
        re_tmpl = '.+[_-][Rr]?{}\.(?:' + '|'.join(extensions) + ')$'
        forward_re = re.compile(re_tmpl.format('1'))
        reverse_re = re.compile(re_tmpl.format('2'))

        for fname in files:
            if forward_re.search(fname):
                forward.append(fname)
            elif reverse_re.search(fname):
                reverse.append(fname)
            else:
                unpaired.append(fname)

        num_forward = len(forward)
        num_reverse = len(reverse)

        if num_forward and num_reverse and num_forward != num_reverse:
            msg = 'Number of forward ({}) and reverse ({}) reads do not match'
            die(msg.format(num_forward, num_reverse))

    else:
        unpaired = files

    return {'forward': forward, 'reverse': reverse, 'unpaired': unpaired}


# --------------------------------------------------
def line_count(fname):
    """Count the number of lines in a file"""

    n = 0
    for _ in open(fname):
        n += 1

    return n


# --------------------------------------------------
def run_job_file(jobfile, msg='Running job', procs=1):
    """Run a job file if there are jobs"""

    num_jobs = line_count(jobfile)
    warn('{} (# jobs = {})'.format(msg, num_jobs))

    if num_jobs > 0:
        cmd = 'parallel --halt soon,fail=1 -P {} < {}'.format(procs, jobfile)

        try:
            subprocess.run(cmd, shell=True, check=True)
        except subprocess.CalledProcessError as err:
            die('Error:\n{}\n{}\n'.format(err.stderr, err.stdout))
        finally:
            os.remove(jobfile)

    return True


# --------------------------------------------------
def run_centrifuge(**args):
    """Run Centrifuge"""

    file_format = args['file_format']
    files = args['files']
    exclude_ids = get_excluded_tax(args['exclude_tax_ids'])
    index_name = args['index_name']
    index_dir = args['index_dir']
    out_dir = args['out_dir']
    threads = args['threads']
    procs = args['procs']

    reports_dir = os.path.join(out_dir, 'reports')

    if not os.path.isdir(reports_dir):
        os.makedirs(reports_dir)

    jobfile = tmp.NamedTemporaryFile(delete=False, mode='wt')
    exclude_arg = '--exclude-taxids ' + exclude_ids if exclude_ids else ''
    format_arg = '-f' if file_format == 'fasta' else ''

    cmd_tmpl = 'CENTRIFUGE_INDEXES={} centrifuge {} {} -p {} -x {} '
    cmd_base = cmd_tmpl.format(index_dir, exclude_arg, format_arg, threads,
                               index_name)

    for file in files['unpaired']:
        basename = os.path.basename(file)
        tsv_file = os.path.join(reports_dir, basename + '.tsv')
        sum_file = os.path.join(reports_dir, basename + '.sum')
        tmpl = cmd_base + '-U "{}" -S "{}" --report-file "{}"\n'
        if not os.path.isfile(tsv_file):
            jobfile.write(tmpl.format(file, sum_file, tsv_file))

    for i, file in enumerate(files['forward']):
        basename = os.path.basename(file)
        tsv_file = os.path.join(reports_dir, basename + '.tsv')
        sum_file = os.path.join(reports_dir, basename + '.sum')
        tmpl = cmd_base + '-1 "{}" -2 "{}" -S "{}" --report-file "{}"\n'
        if not os.path.isfile(tsv_file):
            jobfile.write(
                tmpl.format(file, files['reverse'][i], sum_file, tsv_file))

    jobfile.close()

    run_job_file(jobfile=jobfile.name, msg='Running Centrifuge', procs=procs)

    return reports_dir


# --------------------------------------------------
def get_excluded_tax(ids):
    """Verify the ids look like numbers"""

    tax_ids = []

    if ids:
        for s in [x.strip() for x in ids.split(',')]:
            if s.isnumeric():
                tax_ids.append(s)
            else:
                warn('tax_id "{}" is not numeric'.format(s))

    return ','.join(tax_ids)


# --------------------------------------------------
def make_bubble(reports_dir, out_dir, title):
    """Make bubble chart"""

    fig_dir = os.path.join(out_dir, 'figures')

    if not os.path.isdir(fig_dir):
        os.makedirs(fig_dir)

    cur_dir = os.path.dirname(os.path.realpath(__file__))
    bubble = os.path.join(cur_dir, 'centrifuge_bubble.r')
    tmpl = '{} --dir "{}" --title "{}" --outdir "{}"'
    job = tmpl.format(bubble, reports_dir, title, fig_dir)
    warn(job)

    subprocess.run(job, shell=True)

    return fig_dir


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

Appendix 1: argparse

The `argparse` module will interpret all the command-line arguments to your program. I suggest you use `argparse` for every command-line program you write so that you always have a standard way to get arguments and present help.

Types of arguments

Command-line arguments come in a variety of flavors:

- Positional: The order and number of the arguments is what determines their meaning. Some programs might expect, for instance, a file name as the first argument and an output directory as the second.
- Named options: Standard Unix format allows for a "short" name like `-f` (one dash and a single character) or a "long" name like `--file` (two dashes and a string of characters) followed by some value like a file name or a number. This allows for arguments to be provided in any order or not provided in which case the program can use a reasonable default value.
- Flag: A "Boolean" value like "yes"/"no" or `True`/`False` usually indicated by something that looks like a named option but without a value, e.g., `-d` or `--debug` to turn on debugging. Typically the presence of the flag indicates a `True` value for the argument; therefore, it's absence would mean `False`, so `--debug` turns *on* debugging while no `--debug` flag means there should not no debugging.

Datatypes of values

The `argparse` module can save you enormous amounts of time by forcing the user to provide arguments of a particular type. If you run `new.py`, all of the above types of arguments are present along with suggestions for how to get string or integer values:

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

You should change the `description` to a short sentence describing your program. The `formatter_class` argument tells `argparse` to show the default values in the the standard help documentation.

The `positional` argument's definition indicates we expect exactly one positional argument. The `-a` argument's `type` must be a `str` while the `-i` option must be something that Python can convert to the `int` type (you can also use `float`). Both of these arguments have `default` values which means the user is not required to provide them. You could instead define them with `required=True` to force the user to provide values themselves.

The `-f` flag notes that the `action` is to `store_true` which means the value's default with be `True` if the argument is present and `False` otherwise.

The `type` of the argument can be something much richer than simple Python types like strings or numbers. You can indicate that an argument must be a existing, readable file. Here is a simple implementation in Python of `cat -n`:

```
#!/usr/bin/env python3
"""Python version of `cat -n`"""

import argparse


# --------------------------------------------------
def get_args():
    """Get command-line arguments"""

    parser = argparse.ArgumentParser(
        description='Argparse Python script',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('file',
                        metavar='FILE',
                        type=argparse.FileType('r'),
                        help='Input file')

    return parser.parse_args()


# --------------------------------------------------
def main():
    """Make a jazz noise here"""

    args = get_args()
    fh = args.file

    print('Reading "{}"'.format(fh.name))
    for i, line in enumerate(fh):
        print(i, line, end='')


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

The `type` of the input `file` argument is an *open file handle* which we can directly read line-by-line with a `for` loop! Because it's a file *handle* and not a file *name*, I chose to call the variable `fh` to help me remember what it is. You can access the file's name via `fh.name`.

```
$ ./cat_n.py ../../inputs/the-bustle.txt
Reading "../../inputs/the-bustle.txt"
0 The bustle in a house
1 The morning after death
2 Is solemnest of industries
3 Enacted upon earth,--
4
5 The sweeping up the heart,
6 And putting love away
7 We shall not want to use again
8 Until eternity.
```

Number of arguments

If you want one positional argument, you can define them like so:

```
#!/usr/bin/env python3
"""One positional argument"""

import argparse

parser = argparse.ArgumentParser(
    description='One positional argument',
    formatter_class=argparse.ArgumentDefaultsHelpFormatter)

parser.add_argument('first', metavar='str', help='First argument')
args = parser.parse_args()
print('first =', args.first)
```

If the user provides anything other exactly one argument, they get a help message:

```
$ ./one_arg.py
usage: one_arg.py [-h] str
one_arg.py: error: the following arguments are required: str
$ ./one_arg.py foo bar
usage: one_arg.py [-h] str
one_arg.py: error: unrecognized arguments: bar
$ ./one_arg.py foo
first = foo
```

If you want two different positional arguments:

```
#!/usr/bin/env python3
"""Two positional arguments"""

import argparse

parser = argparse.ArgumentParser(
    description='Two positional arguments',
    formatter_class=argparse.ArgumentDefaultsHelpFormatter)

parser.add_argument('first', metavar='str', help='First argument')

parser.add_argument('second', metavar='int', help='Second argument')

return parser.parse_args()

print('first =', args.first)
print('second =', args.second)
```

Again, the user must provide exactly this number of positional arguments:

```
$ ./two_args.py
usage: two_args.py [-h] str str
two_args.py: error: the following arguments are required: str, str
$ ./two_args.py foo
usage: two_args.py [-h] str str
two_args.py: error: the following arguments are required: str
$ ./two_args.py foo bar
first = foo
second = bar
```

You can also use the `nargs=N` option to specify some number of arguments. It only makes sense if the arguments are the same thing like two files:

```
#!/usr/bin/env python3
"""nargs=2"""

import argparse

parser = argparse.ArgumentParser(
    description='nargs=2',
    formatter_class=argparse.ArgumentDefaultsHelpFormatter)

parser.add_argument('files', metavar='FILE', nargs=2, help='Two files')

args = parser.parse_args()

file1, file2 = args.files
print('file1 =', file1)
print('file2 =', file2)
```

The help indicates we want two files:

```
$ ./nargs2.py foo
usage: nargs2.py [-h] FILE FILE
nargs2.py: error: the following arguments are required: FILE
```

And we can unpack the two file arguments and use them:

```
$ ./nargs2.py foo bar
file1 = foo
file2 = bar
```

If you want one or more of some argument, you can use `nargs='+'`:

```
$ cat nargs+.py
#!/usr/bin/env python3
"""nargs=+"""
```

```
import argparse

parser = argparse.ArgumentParser(
    description='nargs=+',
    formatter_class=argparse.ArgumentDefaultsHelpFormatter)

parser.add_argument('files', metavar='FILE', nargs='+', help='Some files')

args = parser.parse_args()
files = args.files

print('number = {}'.format(len(files)))
print('files  = {}'.format(', '.join(files)))
```

Note that this will return a `list` – even a single argument will become a `list` of one value:

```
$ ./nargs+.py
usage: nargs+.py [-h] FILE [FILE ...]
nargs+.py: error: the following arguments are required: FILE
$ ./nargs+.py foo
number = 1
files  = foo
$ ./nargs+.py foo bar
number = 2
files  = foo, bar
```

Choices

Sometimes you want to limit the values of an argument. You can pass in a `list` of valid values to the `choices` option.

```
$ cat appendix/argparse/choices.py
#!/usr/bin/env python3
"""Choices"""

import argparse

parser = argparse.ArgumentParser(
    description='Choices',
    formatter_class=argparse.ArgumentDefaultsHelpFormatter)

parser.add_argument('color', metavar='str', help='Color', choices=['red', 'yellow', 'blue'])

args = parser.parse_args()
```

```
print('color =', args.color)
```

Any value not present in the list will be rejected and the user will be shown the valid choices:

```
$ ./choices.py
usage: choices.py [-h] str
choices.py: error: the following arguments are required: str
$ ./choices.py purple
usage: choices.py [-h] str
choices.py: error: argument str: invalid choice: 'purple' (choose from 'red', 'yellow', 'blu
```

Automatic help

The `argparse` module reserves the `-h` and `--help` flags for generating help documentation. You do not need to add these nor are you allowed to use these flags for other purposes. Using the above definition, this is the help that `argparse` will generate:

```
$ ./foo.py
usage: foo.py [-h] [-a str] [-i int] [-f] str
foo.py: error: the following arguments are required: str
[cholla@~/work/python/playful_python/article]$ ./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)
```

Notice how unhelpful a name like `positional` is?

Getting the argument values

The values for the arguments will be accessible through the “long” name you define and will have been coerced to the Python data type you indicated. If I change `main` to this:

```
# --------------------------------------------------
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, type(str_arg)))
    print('int_arg = "{}" ({})'.format(int_arg, type(int_arg)))
    print('flag_arg = "{}" ({})'.format(flag_arg, type(flag_arg)))
    print('positional = "{}" ({})'.format(pos_arg, type(pos_arg)))
```

And then run it:

```
$ ./foo.py -a foo -i 4 -f bar
str_arg = "foo" (<class 'str'>)
int_arg = "4" (<class 'int'>)
flag_arg = "True" (<class 'bool'>)
positional = "bar" (<class 'str'>)
```

Notice how we might think that `-f` takes the argument `bar`, but it is defined as a `flag` and the `argparse` knows that the program take

```
$ ./foo.py foo -a bar -i 4 -f
str_arg = "bar" (<class 'str'>)
int_arg = "4" (<class 'int'>)
flag_arg = "True" (<class 'bool'>)
positional = "foo" (<class 'str'>)
```

Appendix 2: 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
#!/usr/bin/env python3
"""Counts by tax name"""

import csv
import os
import sys
from collections import defaultdict

args = sys.argv[1:]

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

sum_file = args[0]

basename, ext = os.path.splitext(sum_file)
if not ext == '.sum':
    print('File extention "{}" is not ".sum"'.format(ext))
    sys.exit(1)

tsv_file = basename + '.tsv'
if not os.path.isfile(tsv_file):
    print('Cannot find expected TSV "{}"'.format(tsv_file))
    sys.exit(1)

tax_name = {}
with open(tsv_file) as csvfile:
    reader = csv.DictReader(csvfile, delimiter='\t')
    for row in reader:
        tax_name[row['taxID']] = row['name']

counts = defaultdict(int)
with open(sum_file) as csvfile:
    reader = csv.DictReader(csvfile, delimiter='\t')
    for row in reader:
        taxID = row['taxID']
```

```
            counts[taxID] += 1

    print('\t'.join(['count', 'taxID']))
    for taxID, count in counts.items():
        name = tax_name.get(taxID) or 'NA'
        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
```

Appendix 3: Bash 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"
```

Appendix 4: Common Patterns in Python

"To me programming is more than an important practical art. It is also a gigantic undertaking in the foundations of knowledge." - Grace Hopper

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
#!/usr/bin/env python3
"""Read a file argument"""

import os
import sys

args = sys.argv[1:]

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

filename = args[0]

if not os.path.isfile(filename):
    print('"{}" is not a file'.format(filename), file=sys.stderr)
    sys.exit(1)

for line in open(filename):
    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
```

```
import sys

args = sys.argv[1:]

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

dirname = args[0]

if not os.path.isdir(dirname):
    print('"{}" is not a directory'.format(dirname), file=sys.stderr)
    sys.exit(1)

for entry in os.listdir(dirname):
    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
```

```
#!/usr/bin/env python3

import os
import sys


args = sys.argv[1:]

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

file = args[0]

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

for i, line in enumerate(open(file)):
    if (i + 1) % 2 == 0:
        continue

    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
#!/usr/bin/env python3
"""Test for a directory and create if needed"""

import os
import sys

```

```
args = sys.argv[1:]

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

dirname = args[0]

if os.path.isdir(dirname):
    print('"{}" exists'.format(dirname))
else:
    print('Creating "{}"'.format(dirname))
    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
#!/usr/bin/env python3
"""Unpack dict"""

import os
import sys

albums = {
    "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,
    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	
```

```
import os
import sys

albums = {
    "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,
}

# Create a list of (value, key) tuples
# sorted in descending order by the values
pairs = sorted([(x[1], x[0]) for x in albums.items()])

for year, album in pairs:
    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
#!/usr/bin/env python3
"""Extract codons from DNA"""

import os
import sys

args = sys.argv[1:]
num_args = len(args)

if not 1 <= num_args <= 2:
    print('Usage: {} DNA'.format(os.path.basename(sys.argv[0])))
    sys.exit(1)

string = args[0]
k = 3
n = len(string) - k + 1

for i in range(0, n, k):
    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
#!/usr/bin/env python3
"""Extract k-mers from string"""

import os
import sys

args = sys.argv[1:]
```

```
num_args = len(args)

if not 1 <= num_args <= 2:
    print('Usage: {} STR [K]'.format(os.path.basename(sys.argv[0])))
    sys.exit(1)

string = args[0]
k = args[1] if num_args == 2 else '3'

# Guard against a string like "foo"
if not k.isdigit():
    print('k "{}" is not a digit'.format(k))
    sys.exit(1)

# Safe to convert now
k = int(k)

if len(string) < k:
    print('There are no {}-length substrings in "{}"'.format(k, string))
else:
    n = len(string) - k + 1
    for i in range(0, n):
        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
```

Make All Items in a List Uppercase

If you need to check all the strings in a list in a case-insensitive fashion, one way would be to upper- or lower-case all the strings. A very Pythonic way is to use a list comprehension, but we can also borrow an idea from the purely functional programming world where we use a "higher-order function," which is a function that takes one or more other functions as arguments. In this case, the `map` function expects as it's first argument some other function, here `str.upper`. Notice it's not `str.upper()` with parens! That is the syntax for **calling** the `str.upper` function. We want to pass **the function itself**, so we leave off the parens. The function is applied to each item in the list and returns a new list. The original list remains unchanged.

```
>>> a = ['foo', 'bar', 'baz']
>>> [s.upper() for s in a]
['FOO', 'BAR', 'BAZ']
>>> list(map(str.upper, a))
['FOO', 'BAR', 'BAZ']
>>> a
['foo', 'bar', 'baz']
```

Another way to write the `map` is to use a `lambda` expression which is just a very short, anonymouse (unnamed) function:

```
>>> list(map(lambda s: s.upper(), a))
['FOO', 'BAR', 'BAZ']
```

Here is the code in action:

```
$ cat -n upper_list.py
     1	#!/usr/bin/env python3
     2	
     3	import os
     4	import sys
     5	
     6	args = sys.argv[1:]
     7	
     8	if len(args) < 1:
     9	    print('Usage: {} ARG [ARG...]'.format(os.path.basename(sys.argv[0])))
    10	    sys.exit(1)
    11	
    12	print('List comprehension')
    13	print(', '.join([x.upper() for x in args]))
    14	
    15	print('Map')
    16	print(', '.join(map(str.upper, args)))
$ ./upper_list.py foo bar baz
List comprehension
FOO, BAR, BAZ
Map
FOO, BAR, BAZ
```

Appendix 5: Logging in Python

So far we've use `print` statements that go to STDOUT and the `warn` function that makes is slightly more convenient to write to STDERR. The trouble with this approach to writing and debugging code is that you need to remove all the `print`/`warn` statements prior to releasing your code or running your tests. With the `logging` module (https://docs.python.org/3/library/logging.html), you can sprinkle messages to yourself liberally throughout your code and chose *at run time* which ones to see.

Like with `random.seed`, calls to the `logging` module affect the **global state** of how logging happens. First you need to set up how the logging will happen using the `basicConfig` (https://docs.python.org/3/library/logging.html#logging.basicConfig). Typically you will set log message to go to a `filename` (if you don't indicate a filename then messages go to STDERR) with the `filemode` of "w" (write, which will overwrite existing files; default is "a" for append) at some `level` like `logging.DEBUG` (default is `logging.NOTSET` so everything prints). Here is a script (in `examples`) that does that:

```
$ cat -n basic.py
     1	#!/usr/bin/env python3
     2	
     3	import logging
     4	import os
     5	import sys
     6	
     7	prg = sys.argv[0]
     8	prg_name, _ = os.path.splitext(os.path.basename(prg))
     9	logging.basicConfig(
    10	    filename=prg_name + '.log',
    11	    filemode='w',
    12	    level=logging.DEBUG
    13	)
    14	
    15	logging.debug('DEBUG!')
    16	logging.critical('CRITICAL!')
```

Before running the program, see that there is no log file:

```
$ ls
basic.py* long.py*
```

Run it, and see that `basic.log` has been created:

```
$ ls
basic.log  basic.py*  long.py*
$ cat basic.log
DEBUG:root:DEBUG!
```

```
CRITICAL:root:CRITICAL!
```

The key is to understand the hierarchy of the levels:

1. CRITICAL
2. ERROR
3. WARNING
4. INFO
5. DEBUG
6. NOTSET

The log level includes everything above the level you set. As in the above program, we set it to `logging.DEBUG` and so a call to `critical` was included. If you change the program to `logging.CRITICAL`, then `error` through `debug` calls are not emitted:

```
$ cat -n basic.py
     1	#!/usr/bin/env python3
     2	
     3	import logging
     4	import os
     5	import sys
     6	
     7	prg = sys.argv[0]
     8	prg_name, _ = os.path.splitext(os.path.basename(prg))
     9	logging.basicConfig(
    10	    filename=prg_name + '.log',
    11	    filemode='w',
    12	    level=logging.CRITICAL
    13	)
    14	
    15	logging.debug('DEBUG!')
    16	logging.critical('CRITICAL!')
$ ./basic.py
$ cat basic.log
CRITICAL:root:CRITICAL!
```

If you find yourself repeatedly debugging some program or just need to know information about how it is proceeding, then `logging` is for you. Maybe you have some functions or system calls that take a long time; sometimes you want to monitor how they are going and other times (e.g., running unattended on the HPC) you don't. Here is a program that logs random levels and then sleeps for one second. To see how this could be useful, open two terminals and navigate to the `examples` directory.

Here is the program:

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

```

import argparse
import logging
import os
import random
import sys
import time


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

    parser.add_argument(
        '-d', '--debug', help='Debug mode', action='store_true')

    return parser.parse_args()


# --------------------------------------------------
def main():
    """Make a jazz noise here"""
    args = get_args()

    prg = sys.argv[0]
    prg_name, _ = os.path.splitext(os.path.basename(prg))
    logging.basicConfig(
        filename=prg_name + '.log',
        filemode='a',
        level=logging.DEBUG if args.debug else logging.CRITICAL)

    logging.debug('Starting')
    for i in range(1, 11):
        method = random.choice([
            logging.info, logging.warning, logging.error, logging.critical,
            logging.debug
        ])
        method('{}: Hey!'.format(i))
        time.sleep(1)

    logging.debug('Done')

    print('Done.')
```

```python


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

Start running `long.py` in one terminal, then execute `tail -f long.log` in the other where `tail` is the program to show you the end of a file and `-f` tells `tail` to stay running and "follow" the file as it grows. (Use CTRL-C to stop following.) Following is what I see when I run `long.py`. Note that, since I didn't set the `-d|--debug` flag, my program will only log *critical* errors:

```
CRITICAL:root:5: Hey!
CRITICAL:root:8: Hey!
```

And when I run `long.py -d`, everything from "debug" on up is displayed:

```
DEBUG:root:Starting
WARNING:root:1: Hey!
ERROR:root:2: Hey!
DEBUG:root:3: Hey!
DEBUG:root:4: Hey!
CRITICAL:root:5: Hey!
INFO:root:6: Hey!
ERROR:root:7: Hey!
INFO:root:8: Hey!
DEBUG:root:9: Hey!
CRITICAL:root:10: Hey!
DEBUG:root:Done
```

Appendix 6: Writing Tests For Your Python Programs

> Much of the essence of building a program is in fact the debugging of the specification. – Fred Brooks

Let's start with a simple example of a `hello` program that should say "Hello, name!"

```
$ ./hello.py
Usage: hello.py NAME [NAME...]
$ ./hello.py Jan
Hello, Jan!
$ ./hello.py Bobby Peter Greg
Hello, Bobby!
Hello, Peter!
Hello, Greg!
```

Here is one way to write such a program *with an embedded test*:

```
#!/usr/bin/env python3

import os
import sys

def hello(name):
    return 'Hello, {}!'.format(name)

def test_hello():
    assert hello('World') == 'Hello, World!'
    assert hello('') == 'Hello, !'
    assert hello('my name is Fred') == 'Hello, my name is Fred!'

def main():
    args = sys.argv[1:]
    if not args:
        print('Usage: {} NAME [NAME...]'.format(os.path.basename(sys.argv[0])))
        sys.exit(1)

    for arg in args:
        print(hello(arg))

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

Specifically I've written this to use the PyTest (https://docs.pytest.org/en/latest/) framework that will search for function names starting with `test_` and will run them.

```
$ pytest -v hello.py
============================= test session starts ==============================
platform darwin -- Python 3.6.8, pytest-4.2.0, py-1.7.0, pluggy-0.8.1 -- /anaconda3/bin/pyth
cachedir: .pytest_cache
rootdir: /Users/kyclark/work/biosys-analytics/lectures/18-writing-tests/examples, inifile:
plugins: remotedata-0.3.1, openfiles-0.3.2, doctestplus-0.2.0, arraydiff-0.3
collected 1 item

hello.py::test_hello PASSED                                              [100%]

=========================== 1 passed in 0.03 seconds ===========================
```

The `assert` function you see in the `test_hello` function is a built-in Python function that evaluates some predicate and will throw an error if the predicate is false. For instance, we `assert` that `hello("World")` should return the string `Hello, World!`. If this does not happen, the test will fail:

```
>>> def hello(name):
...     return 'Hello, {}!'.format(name)
...
>>> assert hello('World') == 'foo'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

PyTest will find such errors and report them as failed tests:

```
$ cat -n hello_bad.py
     1	#!/usr/bin/env python3
     2	
     3	import os
     4	import sys
     5	
     6	def hello(name):
     7	    return 'Hello, {}!'.format(name)
     8	
     9	def test_hello():
    10	    assert hello('World') == 'Hello, World.'
    11	
    12	def main():
    13	    args = sys.argv[1:]
    14	    if not args:
    15	        print('Usage: {} NAME [NAME...]'.format(os.path.basename(sys.argv[0])))
    16	        sys.exit(1)
    17	
    18	    for arg in args:
```

```
    19          print(hello(arg))
    20
    21  if __name__ == '__main__':
    22      main()
$ pytest -v hello_bad.py
============================= test session starts ==============================
platform darwin -- Python 3.6.8, pytest-4.2.0, py-1.7.0, pluggy-0.8.1 -- /anaconda3/bin/pyth
cachedir: .pytest_cache
rootdir: /Users/kyclark/work/biosys-analytics/lectures/18-writing-tests/examples, inifile:
plugins: remotedata-0.3.1, openfiles-0.3.2, doctestplus-0.2.0, arraydiff-0.3
collected 1 item

hello_bad.py::test_hello FAILED                                          [100%]

=================================== FAILURES ===================================
__________________________________ test_hello __________________________________

    def test_hello():
>       assert hello('World') == 'Hello, World.'
E       AssertionError: assert 'Hello, World!' == 'Hello, World.'
E         - Hello, World!
E         ?             ^
E         + Hello, World.
E         ?             ^

hello_bad.py:10: AssertionError
=========================== 1 failed in 0.08 seconds ===========================
```

The error output highlights the differences between what was expected (`Hello, World.` ending in a period) and what the `hello` function actually returned (`Hello, World!` ending in an exclamation point).

I would recommend writing your tests for every function directly below the function being tested and calling the test `test_function`. Try to make a function do just one thing, then write tests to ensure it does that thing. Try to write tests that probe the edge cases, e.g., passing an empty string or a very long string. Here's a version where the `hello` function will only greet if the argument is a `str`; otherwise it will return an admonishment. This is an extremely contrived example because everything coming in via `sys.argv` is by definition a string, so I will intentionally convert anything that looks like a digit to an `int` so that we can see the error:

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

```

def hello(name):
    if type(name) is str:
        return 'Hello, {}!'.format(name)
    else:
        return 'Can only say hello to a string'


def test_hello():
    assert hello('World') == 'Hello, World!'
    assert hello('') == 'Hello, !'
    assert hello('my name is Fred') == 'Hello, my name is Fred!'

    err = 'Can only say hello to a string'
    assert hello(4) == err
    assert hello(None) == err
    assert hello(float) == err
    assert hello(str) == err

def main():
    args = sys.argv[1:]
    if not args:
        print('Usage: {} NAME [NAME...]'.format(os.path.basename(sys.argv[0])))
        sys.exit(1)

    for arg in args:
        if arg.isdigit(): arg = int(arg)

        print(hello(arg))

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

```
$ ./hello_fail.py Bob 3 Sue
Hello, Bob!
Can only say hello to a string
Hello, Sue!
$ pytest -v hello_fail.py
============================ test session starts =============================
platform darwin -- Python 3.6.8, pytest-4.2.0, py-1.7.0, pluggy-0.8.1 -- /anaconda3/bin/pyth
cachedir: .pytest_cache
rootdir: /Users/kyclark/work/biosys-analytics/lectures/18-writing-tests/examples, inifile:
plugins: remotedata-0.3.1, openfiles-0.3.2, doctestplus-0.2.0, arraydiff-0.3
collected 1 item

hello_fail.py::test_hello PASSED                                          [100%]
```

```
========================== 1 passed in 0.04 seconds ===========================
```

These types of tests that live *inside* each of your source files and test invididual functions are known as "unit tests". As your software grows, you may find yourself breaking your functions into logically grouped files or modules. We can also write tests that live *outside* our program files to ensure the proper integration of modules as well as the user interface we present. All the `test.py` programs that have been included in your assignments have these types of tests – ensuring, for instance, that your program will create a "usage" statement if passed no arguments or `-h|--help`, will print message to STDERR and `sys.exit()` with a non-zero value when there is an error, or will run to completion given good input and produce the expected STDOUT and/or output files.

Here is a `test.py` that tests for the usage statement and error code on no input and then tests for one, argument, more than one argument, and an argument that is more than one word:

```
$ cat -n test.py
     1	#!/usr/bin/env python3
     2	
     3	from subprocess import getstatusoutput
     4	
     5	prg = './hello.py'
     6	
     7	def test_usage():
     8	    rv, out = getstatusoutput('{}'.format(prg))
     9	    assert rv != 0
    10	    assert out.lower().startswith('usage')
    11	
    12	def test_runs_ok():
    13	    rv1, out1 = getstatusoutput('{} Carl'.format(prg))
    14	    assert rv1 == 0
    15	    assert out1 == 'Hello, Carl!'
    16	
    17	    rv2, out2 = getstatusoutput('{} Barbara McClintock'.format(prg))
    18	    assert rv2 == 0
    19	    assert out2 == 'Hello, Barbara!\nHello, McClintock!'
    20	
    21	    rv3, out3 = getstatusoutput('{} "Barbara McClintock"'.format(prg))
    22	    assert rv3 == 0
    23	    assert out3 == 'Hello, Barbara McClintock!'
```

I typically create a `Makefile` with a `test` target to show users how to run the tests:

```
$ cat -n Makefile
     1	.PHONY: test
     2	
```

```
test:
	pytest -v test.py
```

```
$ make test
pytest -v test.py
============================= test session starts ==============================
platform darwin -- Python 3.6.8, pytest-4.2.0, py-1.7.0, pluggy-0.8.1 -- /anaconda3/bin/pyth
cachedir: .pytest_cache
rootdir: /Users/kyclark/work/biosys-analytics/lectures/18-writing-tests/examples, inifile:
plugins: remotedata-0.3.1, openfiles-0.3.2, doctestplus-0.2.0, arraydiff-0.3
collected 2 items

test.py::test_usage PASSED                                                [ 50%]
test.py::test_runs_ok PASSED                                              [100%]

=========================== 2 passed in 0.19 seconds ===========================
```

Introduction to Regular Expressions in Python

The term "regular expression" is a formal, linguistic term 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 of characters that we want to find. Regexes are an entirely separate DSL (domain-specific language) that we use inside Python, just like in the previous chapter we use SQL statements to communite with SQLite. While it's a bit of a drag to have to learn yet another language, the bonus is that you can use regular expressions in many places besides Python including with command line tools like `grep` and `awk` as well as within other languages like Perl and Rust.

We can `import re` to use the Python regular expression module and use it to search text.

Matching a "number"

How do we match a string that looks like a "number"? Numbers can be reprented as many types of strings. Here are just a few:

- integer: 1, +10, -42
- float: +0.2, 3.14, 10., -.03
- scientific notation: 4.32e-30
- currency: $10,324,102.88

Let's start simple with a single digit:

```
print(re.match('1', '1'))
```

```
<_sre.SRE_Match object; span=(0, 1), match='1'>
```

But that only works for just "1"

```
print(re.match('2', '1'))
```

```
None
```

How do we match all the numbers from 0 to 9? We can create a character class that contains that range:

```
print(re.match('[0-9]', '1'))
```

```
<_sre.SRE_Match object; span=(0, 1), match='1'>
```

There is a short-hand for the character class `[0-9]` that is `\d` (digit)

```
re.match('\d', '1')
```

```
<_sre.SRE_Match object; span=(0, 1), match='1'>
```

But this only matches the first number we see:

```
re.match('\d', '123')
```

```
<_sre.SRE_Match object; span=(0, 1), match='1'>
```

We can use `{}` to indicate `{min,max}`, `{min,}`, `{,max}`, or `{exactly}`:

```
print(re.match('\d{1,4}', '8005551212'))
```

```
<_sre.SRE_Match object; span=(0, 4), match='8005'>
```

```
print(re.match('\d{1,}', '8005551212'))
```

```
<_sre.SRE_Match object; span=(0, 10), match='8005551212'>
```

```
print(re.match('\d{,5}', '8005551212'))
```

```
<_sre.SRE_Match object; span=(0, 5), match='80055'>
```

```
print(re.match('\d{8}', '8005551212'))
```

```
<_sre.SRE_Match object; span=(0, 8), match='80055512'>
```

match vs search

Note that we are using `re.match` which requires the regex to match **at the beginning of the string**:

```
print(re.match('\d{10}', 'That number to call is 8005551212!'))
```

```
None
```

If you want to match anywhere in the string, use `re.search`:

```
for s in ['123', 'abc456', '789def']:
    print(s, re.search('\d{3}', s))
```

```
123 <_sre.SRE_Match object; span=(0, 3), match='123'>
abc456 <_sre.SRE_Match object; span=(3, 6), match='456'>
789def <_sre.SRE_Match object; span=(0, 3), match='789'>
```

To anchor your match to the beginning of the string, use the `^`:

```
for s in ['123', 'abc456', '789def']:
    print(s, re.search('^\d{3}', s))
```

```
123 <_sre.SRE_Match object; span=(0, 3), match='123'>
abc456 None
789def <_sre.SRE_Match object; span=(0, 3), match='789'>
```

Use `$` for the end of the string:

```
for s in ['123', 'abc456', '789def']:
    print(s, re.search('\d{3}$', s))
```

```
123 <_sre.SRE_Match object; span=(0, 3), match='123'>
abc456 <_sre.SRE_Match object; span=(3, 6), match='456'>
789def None
```

And use both to say that the entire string from beginning to end must match:

```
for s in ['123', 'abc456', '789def']:
    print(s, re.search('^\d{3}$', s))
```

```
123 <_sre.SRE_Match object; span=(0, 3), match='123'>
abc123 None
123def None
```

Returning to our previous problem of trying to see if we got *exactly* one "X" or "O" for our tic-tac-toe player:

```
for player in ['X', 'O', 'XX', 'OO']:
    print(player, re.match('[XO]{1}', player))
```

```
X <_sre.SRE_Match object; span=(0, 1), match='X'>
O <_sre.SRE_Match object; span=(0, 1), match='O'>
XX <_sre.SRE_Match object; span=(0, 1), match='X'>
OO <_sre.SRE_Match object; span=(0, 1), match='O'>
```

The problem is that there is a match of `[XO]{1}` in the strings "XX" and "OO" – there *is* exactly one X or O at the beginning of those strings. Since `re.match` already anchors the match to the beginning of the string, we could just add `$` to the end of our pattern:

```
for player in ['X', 'O', 'XX', 'OO']:
    print(player, re.match('[XO]{1}$', player))
```

```
X <_sre.SRE_Match object; span=(0, 1), match='X'>
O <_sre.SRE_Match object; span=(0, 1), match='O'>
XX None
OO None
```

Or use `re.search` with `^$` to indicate a match over the entire string:

```
for player in ['X', 'O', 'XX', 'OO']:
    print(player, re.search('^[XO]{1}$', player))
```

```
X <_sre.SRE_Match object; span=(0, 1), match='X'>
O <_sre.SRE_Match object; span=(0, 1), match='O'>
XX None
OO None
```

Matching SSNs and Dates

What if we wanted to recognize a US SSN (social security number)? We will use `re.compile` to create the regex and use it in a `for` loop:

```
ssn_re = re.compile('\d{3}-\d{2}-\d{4}')
for s in ['123456789', '123-456-789', '123-45-6789']:
    print('{}: {}'.format(s, ssn_re.match(s)))
```

```
123456789: None
123-456-789: None
123-45-6789: <_sre.SRE_Match object; span=(0, 11), match='123-45-6789'>
```

SSNs always use a dash (-) as a number separator, but dates do not.

```
date_re = re.compile('\d{4}-\d{2}-\d{2}')
dates = ['1999-01-02', '1999/01/02']
for d in dates:
    print('{}: {}'.format(d, date_re.match(d)))
```

```
1999-01-02: <_sre.SRE_Match object; span=(0, 10), match='1999-01-02'>
1999/01/02: None
```

Just as we created a character class with `[0-9]` to represent all the numbers from 0 to 9, we can create a class to represent the separators "/" and "-" with `[/-]`. As regular expressions get longer, it makes sense to break each unit onto a different line and use Python's literal string expression to join them into a single string. As a bonus, we can comment on each unit of the regex.

```
date_re = re.compile('\d{4}'  # year
                     '[/-]'   # separator
                     '\d{2}'  # month
                     '[/-]'   # separator
                     '\d{2}') # day

dates = ['1999-01-02', '1999/01/02']
for d in dates:
    print('{}: {}'.format(d, date_re.match(d)))
```

```
1999-01-02: <_sre.SRE_Match object; span=(0, 10), match='1999-01-02'>
1999/01/02: <_sre.SRE_Match object; span=(0, 10), match='1999/01/02'>
```

You may notice that certain elements are repeated. If we followed DRY (Don't Repeat Yourself), we might want to make variables to hold each piece, but then we could not use the literal string joining trick above. In that case, just go back to using + to join strings:

```
sep = '[/-]'
four_digits = '\d{4}'
two_digits = '\d{2}'

date_re = re.compile(four_digits + # year
                     sep         + # separator
                     two_digits  + # month
                     sep         + # separator
                     two_digits)   # day

dates = ['1999-01-02', '1999/01/02']
for d in dates:
    print('{}: {}'.format(d, date_re.match(d)))
```

```
1999-01-02: <_sre.SRE_Match object; span=(0, 10), match='1999-01-02'>
1999/01/02: <_sre.SRE_Match object; span=(0, 10), match='1999/01/02'>
```

Dates are not always written YYYY-MM-DD where the month/day are zero-padded left, e.g., “01” instead of “1”. How could we handle that? Change our `two_digits` from `\d{2}` (exactly two) to `\d{1,2}` (one or two):

```
sep = '[/-]'
four_digits = '\d{4}'
two_digits = '\d{1,2}'

date_re = re.compile(four_digits + # year
                     sep         + # separator
                     two_digits  + # month
                     sep         + # separator
                     two_digits)   # day

dates = ['1999-01-01', '1999/01/02', '1999/1/2']
for d in dates:
    print('{}: {}'.format(d, date_re.match(d)))
```

```
1999-01-01: <_sre.SRE_Match object; span=(0, 10), match='1999-01-01'>
1999/01/02: <_sre.SRE_Match object; span=(0, 10), match='1999/01/02'>
1999/1/2: <_sre.SRE_Match object; span=(0, 8), match='1999/1/2'>
```

If we wanted to extract each part of the date (year, month, day), we can use parentheses `()` around the parts we want to capture into `groups`. The group “0” is the whole string that was match, and they are numbered sequentially after that for each group.

Can you change the regex to match all three strings?

```
date_re = re.compile('(\d{4})'    # capture year (group 1)
                     '[/-]'       # separator
                     '(\d{1,2})'  # capture month (group 2)
                     '[/-]'       # separator
                     '(\d{1,2})') # capture day (group 3)

dates = ['1999-01-02', '1999/1/2', '1999.01.01']
for d in dates:
    match = date_re.match(d)
    print('{}: {}'.format(d, 'match' if match else 'miss'))
    if match:
        print(match.groups())
        print('year:', match.group(1))
    print()
```

```
1999-01-01: match
('1999', '01', '01')
year: 1999

1999/01/01: match
('1999', '01', '01')
year: 1999

1999.01.01: miss
```

As we add more groups, it can be confusing to remember them by their positions, so we can name them with `?P<name>` just inside the opening paren.

```
date_re = re.compile('(?P<year>\d{4})'
                     '[/-]'
                     '(?P<month>\d{1,2})'
                     '[/-]'
                     '(?P<day>\d{1,2})')

dates = ['1999-1-2', '1999/01/02', '1999.01.01']

for d in dates:
    match = date_re.match(d)
    print('{}: {}'.format(d, 'match' if match else 'miss'))
    if match:
        print('{} = year "{}" month "{}" day "{}"'.format(d,
                                                          match.group('year'),
                                                          match.group('month'),
                                                          match.group('day')))
    print()
```

```
1999-1-2: match
1999-1-2 = year "1999" month "1" day "2"

1999/01/02: match
1999/01/02 = year "1999" month "01" day "02"

1999.01.01: miss
```

Matching US Phone Numbers

What if we wanted to match a US phone number?

```
phone_re = re.compile('(\d{3})'  # area code
                      ' '        # a space
                      '\d{3}'    # prefix
                      '-'        # dash
                      '\d{4}')   # line number
print(phone_re.match('(800) 555-1212'))
```

```
None
```

Why didn't that work?

What do those parentheses do again? They group!

So we need to indicate that the parens are literal things to match by using backslashes \ to escape them.

```
phone_re = re.compile('\('     # left paren
                      '\d{3}'  # area code
                      '\)'     # right paren
                      ' '      # space
                      '\d{3}'  # prefix
                      '-'      # dash
                      '\d{4}') # line number
print(phone_re.match('(800) 555-1212'))
```

```
<_sre.SRE_Match object; span=(0, 14), match='(800) 555-1212'>
```

We could also use character classes to make this more readable:

```
phone_re = re.compile('[(]'    # left paren
                      '\d{3}'  # area code
                      '[)]'    # right paren
                      ' '      # space
                      '\d{3}'  # prefix
                      '-'      # dash
                      '\d{4}') # line number

print(phone_re.match('(800) 555-1212'))
```

```
<_sre.SRE_Match object; span=(0, 14), match='(800) 555-1212'>
```

There is not always a space after the area code, and it may sometimes it may be more than one space (or a tab?). We can use the `\s` to indicate any type of whitespace and `*` to indicate zero or more:

```
phone_re = re.compile('[(]'    # left paren
                      '\d{3}'  # area code
                      '[)]'    # right paren
                      '\s*'    # zero or more spaces
                      '\d{3}'  # prefix
                      '-'      # dash
                      '\d{4}') # line number
phones = ['(800)555-1212', '(800) 555-1212', '(800)  555-1212']
for phone in phones:
    print('{}\t{}'.format(phone, phone_re.match(phone)))
```

```
(800)555-1212   None
(800) 555-1212  None
(800)  555-1212 <_sre.SRE_Match object; span=(0, 15), match='(800)  555-1212'>
```

When the parens around the area code are optional, usually there is a dash to separate the area code:

```
phone_re = re.compile('[(]?'   # optional left paren
                      '\d{3}'  # area code
                      '[)]?'   # optional right paren
                      '[-]?'   # optional dash
                      '\s*'    # zero or more whitespace
                      '\d{3}'  # prefix
                      '-'      # dash
                      '\d{4}') # line number

phones = ['(800)555-1212', '(800) 555-1212', '800-555-1212']
for phone in phones:
    print('{}\t{}'.format(phone, phone_re.match(phone)))
```

```
(800)555-1212   <_sre.SRE_Match object; span=(0, 13), match='(800)555-1212'>
(800) 555-1212  <_sre.SRE_Match object; span=(0, 14), match='(800) 555-1212'>
800-555-1212    <_sre.SRE_Match object; span=(0, 12), match='800-555-1212'>
```

This has the affect of matching a dash after parens which is generally not a valid format:

```
phone_re = re.compile('[(]?'   # optional left paren
                      '\d{3}'  # three digits
                      '[)]?'   # optional right paren
                      '[-]?'   # optional dash
                      '\s*'    # zero or more spaces
                      '\d{3}'  # three digits
                      '-'      # dash
                      '\d{4}') # four digits

phone_re.match('(800)-555-1212')
```

```
<_sre.SRE_Match object; span=(0, 14), match='(800)-555-1212'>
```

We really have to create two regexes to handle these cases:

```
phone_re1 = re.compile('[(]'
                       '\d{3}'
                       '[)]'
                       '\s*'
                       '\d{3}'
                       '-'
                       '\d{4}')

phone_re2 = re.compile('\d{3}'
                       '-'
                       '\d{3}'
                       '-'
                       '\d{4}')

phones = ['(800)555-1212', '(800) 555-1212', '800-555-1212', '(800)-555-1212']
for phone in phones:
    match1 = phone_re1.match(phone)
    match2 = phone_re2.match(phone)
    print('{}\t{}'.format(phone, 'match' if match1 or match2 else 'miss'))
```

```
(800)555-1212   match
(800) 555-1212  match
800-555-1212    match
(800)-555-1212  miss
```

I worked with a graphic artist who always insisted on using dots as the number separator, and sometimes there are no separators at all. The combination of these two regexes find the valid formats and skip the invalid one.

```
phone_re1 = re.compile('[(]'
                       '\d{3}'
                       '[)]'
                       '\s*'
                       '\d{3}'
                       '[.-]'
                       '\d{4}')

phone_re2 = re.compile('\d{3}'
                       '[.-]?'
                       '\d{3}'
                       '[.-]?'
                       '\d{4}')

phones = ['8005551212', '(800)555-1212', '(800) 555-1212',
          '800-555-1212', '(800)-555-1212', '800.555.1212']

for phone in phones:
    match = phone_re1.match(phone) or phone_re2.match(phone)
    print('{}\t{}'.format(phone, 'match' if match else 'miss'))
```

```
8005551212  match
```

```
(800)555-1212   match
(800) 555-1212  match
800-555-1212    match
(800)-555-1212  miss
800.555.1212    match
```

OK, now let's normalize the numbers by using parens to capture the area code, prefix, and line number and then create a standard representation.

```
phone_re1 = re.compile('[(]'
                       '(\d{3})'  # group 1
                       '[)]'
                       '\s*'
                       '(\d{3})'  # group 2
                       '[.-]'
                       '(\d{4})') # group 3

phone_re2 = re.compile('(\d{3})'  # group 1
                       '[.-]?'
                       '(\d{3})'  # group 2
                       '[.-]?'
                       '(\d{4})') # group 3

phones = ['8005551212', '(800)555-1212', '(800) 555-1212',
          '800-555-1212', '(800)-555-1212', '800.555.1212']

for phone in phones:
    match = phone_re1.match(phone) or phone_re2.match(phone)
    standard = '{}-{}-{}'.format(match.group(1),
                                 match.group(2),
                                 match.group(3)) if match else 'miss'
    print('{}\t{}'.format(phone, standard))
```

```
8005551212  800-555-1212
(800)555-1212   800-555-1212
(800) 555-1212  800-555-1212
800-555-1212    800-555-1212
(800)-555-1212  miss
800.555.1212    800-555-1212
```

And if we add named capture groups…

```
phone_re1 = re.compile('[(]'
                       '(?P<area_code>\d{3})'
                       '[)]'
                       '\s*'
                       '(?P<prefix>\d{3})'
                       '[.-]'
```

```
                         '(?P<line_num>\d{4})')

phone_re2 = re.compile('(?P<area_code>\d{3})'
                       '[.-]?'
                       '(?P<prefix>\d{3})'
                       '[.-]?'
                       '(?P<line_num>\d{4})')

phones = ['8005551212', '(800)555-1212', '(800) 555-1212',
          '800-555-1212', '(800)-555-1212', '800.555.1212']

for phone in phones:
    match = phone_re1.match(phone) or phone_re2.match(phone)
    standard = '{}-{}-{}'.format(match.group('area_code'),
                                 match.group('prefix'),
                                 match.group('line_num')) if match else 'miss'
    print('{}\t{}'.format(phone, standard))
```

```
8005551212  800-555-1212
(800)555-1212   800-555-1212
(800) 555-1212  800-555-1212
800-555-1212    800-555-1212
(800)-555-1212  miss
800.555.1212    800-555-1212
```

And if we add named capture groups and named groups in `format`:

```
phone_re1 = re.compile('[(]'
                       '(?P<area_code>\d{3})'
                       '[)]'
                       '\s*(?P<prefix>\d{3})'
                       '[.-]'
                       '(?P<line_num>\d{4})')

phone_re2 = re.compile('(?P<area_code>\d{3})'
                       '[.-]?'
                       '(?P<prefix>\d{3})'
                       '[.-]?'
                       '(?P<line_num>\d{4})')

phones = ['8005551212', '(800)555-1212', '(800) 555-1212',
          '800-555-1212', '(800)-555-1212', '800.555.1212']

for phone in phones:
    match = phone_re1.match(phone) or phone_re2.match(phone)
    tmpl = '{area_code}-{prefix}-{line_num}'
    standard = tmpl.format(prefix=match.group('prefix'),
                           area_code=match.group('area_code'),
                           line_num=match.group('line_num')) if match else 'miss'
    print('{}\t{}'.format(phone, standard))
```

```
8005551212  800-555-1212
(800)555-1212   800-555-1212
(800) 555-1212  800-555-1212
800-555-1212    800-555-1212
(800)-555-1212  miss
800.555.1212    800-555-1212
```

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":

```
print(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):

```
print(re.search('\d+\.', '27.83387'))
print(re.search('\d+\.', '27x83387'))
```

```
<_sre.SRE_Match object; span=(0, 3), match='27.'>
None
```

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:

```
print(re.search('\d+\.\d+', '27'))
print(re.search('\d+(\.\d+)?', '27'))
print(re.search('\d+(\.\d+)?', '27.83387'))
```

```
None
<_sre.SRE_Match object; span=(0, 2), match='27'>
<_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:

```
print(re.search('-?\d+(\.\d+)?', '-27.83387'))
print(re.search('-?\d+(\.\d+)?', '27.83387'))
print(re.search('-?\d+(\.\d+)?', '-27'))
print(re.search('-?\d+(\.\d+)?', '27'))
```

```
<_sre.SRE_Match object; span=(0, 9), match='-27.83387'>
<_sre.SRE_Match object; span=(0, 8), match='27.83387'>
<_sre.SRE_Match object; span=(0, 3), match='-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 `[+-]?`:

```
print(re.search('[+-]?\d+(\.\d+)?', '-27.83387'))
print(re.search('[+-]?\d+(\.\d+)?', '+27.83387'))
print(re.search('[+-]?\d+(\.\d+)?', '27.83387'))
```

```
<_sre.SRE_Match object; span=(0, 9), match='-27.83387'>
<_sre.SRE_Match object; span=(0, 9), match='+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.:

```
print(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+)?)','Lat is "-27.83387" and lon is "+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 is "-27.83387" and lon is "+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)
```

```

file = args[0]

float_re = r'[+-]?\d+\.*\d*'
ll1 = re.compile('(' + float_re + ')\s*[,_]\s*(' + float_re + ')')
ll2 = re.compile('(' + float_re + ')(?:\s*([NS]))?(?:\s*,)?\s+(' + float_re +
                 ')(?:\s*([EW])?)')
loc_hms = r"""
\d+\.\d+'\d+\.\d+"
""".strip()
ll3 = re.compile('(' + loc_hms + ')\s+(' + loc_hms + ')')

for line in open(file):
    line = line.rstrip()
    ll_match1 = ll1.search(line)
    ll_match2 = ll2.search(line)
    ll_match3 = ll3.search(line)

    if ll_match1:
        lat, lon = ll_match1.group(1), ll_match1.group(2)
        lat = float(lat)
        lon = float(lon)
        print('lat = {}, lon = {}'.format(lat, lon))
    elif ll_match2:
        lat, lat_dir, lon, lon_dir = ll_match2.group(
            1), ll_match2.group(2), ll_match2.group(
                3), ll_match2.group(4)
        lat = float(lat)
        lon = float(lon)

        if lat_dir == 'S':
            lat *= -1

        if lon_dir == 'W':
            lon *= -1
        print('lat = {}, lon = {}'.format(lat, lon))
    elif ll_match3:
        lat, lon = ll_match3.group(1), ll_match3.group(2)
        print('lat = {}, lon = {}'.format(lat, lon))
    else:
        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
```

```
import dateparser
for date in ['March 24, 2014',
             '2013-08-15',
             '20100910',
             '02-May-2012',
             'Jul-2009',
             '5/04/2012']:

    print('{:15}\t{}'.format(date, dateparser.parse(date)))
```

```
March 24, 2014  2014-03-24 00:00:00
2013-08-15      2013-08-15 00:00:00
20100910        2000-02-01 09:01:00
```

```
02-May-2012      2012-05-02 00:00:00
Jul-2009         2009-07-26 00:00:00
5/04/2012        2012-05-04 00:00:00
```

You can see it's not perfect, e.g., "20100910" should be "2010-09-10" and "Jul-2009" should not resolve to the 26th 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!

Exercises

Write the regular expressions to parse the year, month, and day from the following date formats found in SRA metadata. When no day is present, e.g., "2/14," use "01" for the day.

```
d1 = "2012-03-09T08:59"
print(d1, re.match('', d1))
```

```
2012-03-09T08:59 <_sre.SRE_Match object; span=(0, 0), match=''>
```

```
d2 = "2012-03-09T08:59:03"
```

```
d3 = "2017-06-16Z"
```

```
d4 = "2015-01"
```

```
d5 = "2015-01/2015-02"
```

```
d6 = "2015-01-03/2015-02-14"
```

```
d7 = "20100910"
```

```
d8 = "12/06"
```

```
d9 = "2/14"
```

```
d10 = "2/14-12/15"
```

```
d11 = "2017-06-16Z"
```

```
# "Excel" format! What is that?! Look it up.
d12 = "34210"
```

```
d13 = "Dec-2015"
```

```
d14 = "March-2017"
```

```
d15 = "May, 2017"
```

```
d16 = "March-April 2017"
```

```
d17 = "July of 2011"
```

```python
d18 = "2008 August"
```

Now combine all your code from the previous cell to normalize all the dates into the same format.

```python
dates = ["2012-03-09T08:59", "2012-03-09T08:59:03", "2017-06-16Z",
         "2015-01", "2015-01/2015-02", "2015-01-03/2015-02-14",
         "20100910", "12/06", "2/14", "2/14-12/15", "2017-06-16Z",
         "34210", "Dec-2015", "March-2017", "May, 2017",
         "March-April 2017", "July of 2011", "2008 August"]

for date in dates:
    year = '1999'
    month = '01'
    day = '01'
    print('{}-{}-{}\t{}'.format(year, month, day, date))
```

```
1999-01-01  2012-03-09T08:59
1999-01-01  2012-03-09T08:59:03
1999-01-01  2017-06-16Z
1999-01-01  2015-01
1999-01-01  2015-01/2015-02
1999-01-01  2015-01-03/2015-02-14
1999-01-01  20100910
1999-01-01  12/06
1999-01-01  2/14
1999-01-01  2/14-12/15
1999-01-01  2017-06-16Z
1999-01-01  34210
1999-01-01  Dec-2015
1999-01-01  March-2017
1999-01-01  May, 2017
1999-01-01  March-April 2017
1999-01-01  July of 2011
1999-01-01  2008 August
```

Appendix 8: Functional Programming Ideas in Python

"Gematria" is a system for assigning a number to a word by summing the numeric values of each of the letters as defined by the Mispar godol (https://en.wikipedia.org/wiki/Gematria). For English characters, we can use the ASCII table (https://en.wikipedia.org/wiki/ASCII). It is not necessary, however, to encode this table in our program as Python provides the `ord` function to convert a character to its "ordinal" (order in the ASCII table) value as well as the `chr` function to convert a number to its "character."

```
print('"{}" = "{}"'.format('A', ord('A')))
```

```
"A" = "65"
```

```
print('"{}" = "{}"'.format('a', ord('a')))
```

```
"a" = "97"
```

```
print('"{}" = "{}"'.format(88, chr(88)))
```

```
"88" = "X"
```

```
print('"{}" = "{}"'.format(112, chr(112)))
```

```
"112" = "p"
```

To implement an ASCII version of gematria in Python, we need to turn each letter into a number and add them all together. So, to start, note that Python can use a `for` loop to cycle through all the members of a list (in order):

```
for n in range(5):
    print(n)
```

```
0
1
2
3
4
```

```
for char in ['p', 'y', 't', 'h', 'o', 'n']:
    print(char)
```

```
p
y
t
h
o
n
```

A "word" is simply a list of characters, so we can iterate over it just like a list of numbers:

```
for char in "python":
    print(char)
```

```
p
y
t
h
o
n
```

Let's print the ordinal (ASCII) value instead:

```
for char in "python":
    print('"{}" = "{}"'.format(char, ord(char)))
```

```
"p" = "112"
"y" = "121"
"t" = "116"
"h" = "104"
"o" = "111"
"n" = "110"
```

Now let's create a variable to hold the running sum of the values:

```
word = "python"
total = 0
for char in word:
    total += ord(char)

print('"{}" = "{}"'.format(word, total))
```

```
"python" = "674"
```

Another way could be to create another list to hold the values and then use the `sum` function:

```
word = "python"
all = []
for char in word:
    all.append(ord(char))

print(all)
print('"{}" = "{}"'.format(word, sum(all)))
```

```
[112, 121, 116, 104, 111, 110]
"python" = "674"
```

Map

We can use a `map` function to transform all the characters via the `ord` function. This is interesting because `map` is a function that takes another function as its first argument. The second is a list of items to feed into the function. The result is the transformed list. For instance, we can use the `str.upper` function to turn each letter (e.g., "p") into the upper-case version ("P"). NB: it's necessary to force the results into a `list`.

```
list(map(str.upper, "python"))
```

```
['P', 'Y', 'T', 'H', 'O', 'N']
```

```
list(map(ord, "python"))
```

```
[112, 121, 116, 104, 111, 110]
```

Now we can `sum` those numbers:

```
sum(map(ord, "python"))
```

```
674
```

Now let's think about how we could apply this to all the words in a file. As above, we can use a `for` loop to iterate over all the lines in a file:

```
for line in open('gettysburg.txt'):
    print(line)
```

```
Four score and seven years ago our fathers brought forth on this

continent, a new nation, conceived in Liberty, and dedicated to the

proposition that all men are created equal.
```

The original is single-spaced, so why is this printing double-spaced? The `for` loop reads each "line" which is a string of text up to and including a newline. The `print` by default adds a newline, so we either need to `print(line, end='')` to indicate we don't want anything at the end:

```
for line in open('gettysburg.txt'):
    print(line, end='')
```

```
Four score and seven years ago our fathers brought forth on this
continent, a new nation, conceived in Liberty, and dedicated to the
proposition that all men are created equal.
```

Or we need to use the `rstrip` function to "strip" whitespace off the "r"ight side of the line:

```
for line in open('gettysburg.txt'):
    print(line.rstrip())
```

```
Four score and seven years ago our fathers brought forth on this
continent, a new nation, conceived in Liberty, and dedicated to the
proposition that all men are created equal.
```

We can use the `split` function to get all the words for each line and a `for` loop to iterate over those:

```
for line in open('gettysburg.txt'):
    for word in line.rstrip().split():
        print(word)
```

```
Four
score
and
seven
years
ago
our
fathers
brought
forth
on
this
continent,
a
new
nation,
conceived
in
Liberty,
and
dedicated
to
the
proposition
that
all
men
are
created
equal.
```

We want to get rid of anything that is not character like the punctuation. There is a function in the `str` library called `isalpha` that returns `True` or `False`:

```
for char in "a8,X.b!G":
    print('"{}" = "{}"'.format(char, str.isalpha(char)))
```

```
"a" = "True"
```

```
"8" = "False"
"," = "False"
"X" = "True"
"." = "False"
"b" = "True"
"!" = "False"
"G" = "True"
```

Each `char` in the loop is itself a string, so we can call the method directly on the variable:

```
for char in "a8,X.b!G":
    print('"{}" = "{}"'.format(char, char.isalpha()))
```

```
"a" = "True"
"8" = "False"
"," = "False"
"X" = "True"
"." = "False"
"b" = "True"
"!" = "False"
"G" = "True"
```

Filter

Similar to what we saw above with the `map` function, we can use `filter` to find all the characters in a string which are `True` for `isalpha`. `filter` is another "higher-order function" that takes another function for its first argument (called the "predicate") and a list as the second argument. Whereas `map` returns *all* the elements of the list transformed by the function, `filter` returns *only those for which the predicate is true.*

```
list(filter(str.isalpha, "a8,X.b!G"))
```

```
['a', 'X', 'b', 'G']
```

The first argument for `map` and `filter` is called the "lambda," and sometimes you will see it written out explicitly like so:

```
list(filter(lambda char: char.isalpha(), "a8,X.b!G"))
```

```
['a', 'X', 'b', 'G']
```

Here is a way to find only even numbers:

```
list(filter(lambda x: x % 2 == 0, range(10)))
```

```
[0, 2, 4, 6, 8]
```

Let's turn that list of characters back into a word with the `join` function:

```
''.join(filter(str.isalpha, "a8,X.b!G"))
```

```
'aXbG'
```

Aside (Regular expressions)

NB: This is not the way I would actually remove punctuation in my own code. I'd be more likely to use regular expressions, e.g., "anything not A-Z, a-z, and 0-9":

```
import re
print(re.sub('[^A-Za-z0-9]', '', 'a8,X.b!G'))
```

```
a8XbG
```

The `string` class actually defines "punctuation":

```
import string
print(string.punctuation)
```

```
!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
```

So we could use that to create a character class of punctuation if that was the only thing we intended to remove:

```
import string
print(re.sub('[' + string.punctuation + ']', '', 'a8,X.b!G'))
```

```
a8XbG
```

Combining map and filter

So, going back to our Gettysburg example, here is a list of all the words without punctuation:

```
for line in open('gettysburg.txt'):
    for word in line.rstrip().split():
        print(''.join(filter(str.isalpha, word)))
```

```
Four
score
and
seven
years
ago
our
fathers
brought
forth
```

```
on
this
continent
a
new
nation
conceived
in
Liberty
and
dedicated
to
the
proposition
that
all
men
are
created
equal
```

Now, rather let's print the `sum` of the `chr` values for each cleaned up word:

```
for line in open('gettysburg.txt'):
    for word in line.rstrip().split():
        clean = ''.join(filter(str.isalpha, word))
        print('"{}" = "{}"'.format(clean, sum(map(ord, clean))))
```

```
"Four" = "412"
"score" = "540"
"and" = "307"
"seven" = "545"
"years" = "548"
"ago" = "311"
"our" = "342"
"fathers" = "749"
"brought" = "763"
"forth" = "547"
"on" = "221"
"this" = "440"
"continent" = "978"
"a" = "97"
"new" = "330"
"nation" = "649"
"conceived" = "944"
"in" = "215"
"Liberty" = "731"
```

```
"and" = "307"
"dedicated" = "919"
"to" = "227"
"the" = "321"
"proposition" = "1222"
"that" = "433"
"all" = "313"
"men" = "320"
"are" = "312"
"created" = "728"
"equal" = "536"
```

Notice that we are calling `rstrip` for every line, so we could easily move that into a `map`, and the "cleaning" code can likewise be moved into a `map`:

```
for line in map(str.rstrip, open('gettysburg.txt')):
    for word in map(lambda w: ''.join(filter(str.isalpha, w)), line.split()):
        print('"{}" = "{}"'.format(word, sum(map(ord, word))))
```

```
"Four" = "412"
"score" = "540"
"and" = "307"
"seven" = "545"
"years" = "548"
"ago" = "311"
"our" = "342"
"fathers" = "749"
"brought" = "763"
"forth" = "547"
"on" = "221"
"this" = "440"
"continent" = "978"
"a" = "97"
"new" = "330"
"nation" = "649"
"conceived" = "944"
"in" = "215"
"Liberty" = "731"
"and" = "307"
"dedicated" = "919"
"to" = "227"
"the" = "321"
"proposition" = "1222"
"that" = "433"
"all" = "313"
"men" = "320"
"are" = "312"
```

```
"created" = "728"
"equal" = "536"
```

At this point, we have arguably sacrificed readability for the sake of using `map` and `filter` – another instance of "just because you can doesn't mean you should!"

We can improve readability, however, by creating our own functions with informative names. Also, since `onlychars` will get rid of the trailing newlines, we can remove the `line.rstrip()` call:

```python
def onlychars(word):
    return ''.join(filter(str.isalpha, word))

def word2num(word):
    return sum(map(ord, word))

for line in open('gettysburg.txt'):
    for word in map(onlychars, line.split()):
        print('"{}" = "{}"'.format(word, word2num(word)))
```

```
"Four" = "412"
"score" = "540"
"and" = "307"
"seven" = "545"
"years" = "548"
"ago" = "311"
"our" = "342"
"fathers" = "749"
"brought" = "763"
"forth" = "547"
"on" = "221"
"this" = "440"
"continent" = "978"
"a" = "97"
"new" = "330"
"nation" = "649"
"conceived" = "944"
"in" = "215"
"Liberty" = "731"
"and" = "307"
"dedicated" = "919"
"to" = "227"
"the" = "321"
"proposition" = "1222"
"that" = "433"
"all" = "313"
"men" = "320"
```

```
"are" = "312"
"created" = "728"
"equal" = "536"
```

Golfing

"Golfing" in code is when you try to express your code in fewer and fewer keystrokes. At some point you cross the line of cleanliness to absudity. Remember:

> It's such a fine line between stupid and clever. – David St. Hubbins (https://www.youtube.com/watch?v=wtXkD1BC564)

Here's a streamlined version that combines `open`, `read`, and `split` to read the entire file into a list of words which are `map`d into `word2num`.

NB: This version assumes you have enough memory to read an *entire file* and split it. The versions above which read and process each line consume only as much memory as any one line needs!

```python
def onlychars(word):
    return ''.join(filter(str.isalpha, word))

def word2num(word):
    return str(sum(map(ord, word)))

print(' '.join(map(word2num,
                   map(onlychars,
                       open('gettysburg.txt').read().split()))))
```

```
412 540 307 545 548 311 342 749 763 547 221 440 978 97 330 649 944 215 731 307 919 227 321 1
```

To mimic the above output:

```python
def onlychars(word):
    return ''.join(filter(str.isalpha, word))

def word2num(word):
    return str(sum(map(ord, onlychars(word))))

print('\n'.join(map(lambda word: '"{}" = "{}"'.format(word, word2num(word)),
                    map(onlychars,
                        open('gettysburg.txt').read().split()))))
```

```
"Four" = "412"
"score" = "540"
"and" = "307"
"seven" = "545"
"years" = "548"
```

```
"ago" = "311"
"our" = "342"
"fathers" = "749"
"brought" = "763"
"forth" = "547"
"on" = "221"
"this" = "440"
"continent" = "978"
"a" = "97"
"new" = "330"
"nation" = "649"
"conceived" = "944"
"in" = "215"
"Liberty" = "731"
"and" = "307"
"dedicated" = "919"
"to" = "227"
"the" = "321"
"proposition" = "1222"
"that" = "433"
"all" = "313"
"men" = "320"
"are" = "312"
"created" = "728"
"equal" = "536"
```

With this, I hope you're now understand what is meant by a "higher-order function" (functions that take other functions as arguments) and how they can streamline your code.

Exercise

Read your local dictionary (e.g., "/usr/share/dict/words") and find how many words share the same numeric representation. Which ones have the value "666"?

```python
from collections import defaultdict

def onlychars(word):
    return ''.join(filter(str.isalpha, word))

file = '/usr/share/dict/words'
num2word = defaultdict(list)
for line in map(str.rstrip, open(file)):
    for word in map(onlychars, line.split()):
        num = sum(map(ord, word))
        num2word[num].append(word)
```

```
satan = '666'
if satan in num2word:
    print('Satan =', num2word[satan])
else:
    print('No Satan!')

count_per_n = []
for n, wordlist in num2word.items():
    count_per_n.append((len(wordlist), n))

top10 = list(reversed(sorted(count_per_n)))[:10]
for num_of_words, n in top10:
    print('{} ({} words) = {} ...'.format(n, len(num2word[n]), ', '.join(num2word[n][:3])))
```

```
No Satan!
973 (623 words) = Actaeaceae, activator, actorship ...
969 (621 words) = abrotanum, acclivous, acidulous ...
965 (611 words) = abhorrent, acoumeter, acronymic ...
855 (607 words) = abuseful, acanthus, acronych ...
861 (601 words) = Absyrtus, acaulous, adjuvant ...
856 (597 words) = abrastol, accismus, acervose ...
971 (596 words) = aburabozu, acropathy, acuteness ...
974 (594 words) = ablastous, absolvent, abysmally ...
972 (592 words) = accessory, acropolis, acutiator ...
1078 (587 words) = absentness, acrogenous, actinozoan ...
```