UNIX 101

Gladys K. Andino, Daniel Dietz, Steve Kelley, Boyu Zhang, Xiao Zhu

Research Computing, ITaP
Purdue University

Contents

WHAT WE WILL LEARN

This is an introduction to using UNIX systems. In this workshop we will be talking about UNIX concepts, tools, and techniques for effectively using UNIX systems. We will be using Purdue systems to demonstrate these topics, but the skills learned are applicable for any UNIX system.

PREREQUISITES

Information Technology at Purdue (ITaP) provides shared cluster computing infrastructures for researchers at Purdue. There are many computing clusters that are available for Purdue faculty and students. The Radon cluster will be used during today's workshop.

To start using the Radon cluster you will need an account. All workshop attendees will have their account set up before the workshop. They can use their Purdue Career Account username and password to log in.

1 LOGGING IN

We will be using the Radon cluster: *https://www.rcac.purdue.edu/compute/radon/*

You can log onto its front-end/job-submission system (radon.rcac.purdue.edu) using your Purdue Career Account login and password. Logging into Radon requires an SSH client if you are using Windows, but Mac/Linux have these built into their OS. There are several Windows SSH clients available for download.

Microsoft Windows:

- PuTTY is an extremely small download of a free, full-featured SSH client.
- SecureCRT is a commercial SSH client which is freely available to Purdue students, faculty, and staff with a Purdue career account.
- As an exercise we will download PuTTY and log in to the cluster

1. Use Google to search for the word "putty"
2. Click the PuTTY Download Page
3. Download putty.exe
4. Wait for the download to finish then double click putty.exe
5. In the "Host Name" box type radon.rcac.purdue.edu
6. Click YES when the program asks if you want to cache the security key
7. Use your career account credentials to log in

Mac OS X:

- The SSH command is pre-installed. You may start a local terminal window from "Applications>Utilities".
- Log in using ssh *username@radon.rcac.purdue.edu*, where username is your Purdue Career Account login.

Linux / Solaris / AIX / HP-UX / Unix:

- The ssh command is pre-installed.

2 BASIC UNIX

UNIX is a text oriented operating system that has been around since the 70s, and is the primary operating system used at high performance computing facilities, as well as underlying the Mac OSX graphical operating system. You interact with the computer via a ***shell***.

A shell is a program that inputs Unix commands from the keyboard and relays them to the Unix system for execution. Shells typically include various shortcuts for users to use in stating their commands, and also a programming feature, in which users can make programs out of sets of their commands.

There are several UNIX shells, but the most common is probably ***bash*** (Bourne again shell), but differences between shells are not very important unless you are going to write ***shell scripts*** to automate your work. In this workshop we are only introducing the smallest possible set of commands needed to work in a UNIX environment – some of the references below give much more detail and an introduction to shell scripting.

2.1 UNIX BASICS REFERENCES

- Unix tutorial for beginners, *http://www.ee.surrey.ac.uk/Teaching/Unix*
- Unix tutorial, *http://evomics.org/learning/unix-tutorial*
- Introduction to Unix, *http://www.doc.ic.ac.uk/˜wjk/UnixIntro*
- A_quick_introduction_to_Unix, *http://en.wikibooks.org/wiki/A_Quick_Introduction_to_Unix*

2.2 FILES AND DIRECTORIES

Files and ***directories*** are two important constructs in UNIX (and most operating systems). They contain your documents, images, code, programs, OS, etc. Everything in UNIX is built on files and directories.

filesystem is a collection of files and directories stored on a single physical device. These are often called "drives" in Windows. The terms are interchangeable but filesystem is typically used in UNIX.

As an example, we will get our first look at files and directories by copying the workshop files. Type the command:

Let's run our first command and grab workshop files. Type the command:

```
$ cp -r /depot/itap/unix101-2016 .
```

Please note that the dot (.) exists at the end of the command. We will discuss files, directories, and the details of this command in more detail later.

2.2.1 FILES

Files simply store information. There are two basic types of files:

- Text (documents, code)

- Binary (images, executables)

Every file has *metadata* associated with it. The metadata contains the name, timestamps and permissions.

2.2.2 DIRECTORIES

Directories are collections of files and directories. They are analogous and interchangeable with "folders" typically used in Windows. Directories and folders are the same thing but the term "directories" is typically used in UNIX.

Much like files, directories have metadata associated with them. Similarly, the metadata contains the name, timestamps, and permissions.

2.2.3 FILE PATHS

In UNIX all files and directories have a ***path***. The "path" of directories you must follow in order to find the file. Directories in the path to a file are separated by a `/` in UNIX.

Examples:

```
/home/username
/home/username/
/home/username/file.txt
```

File extensions don't matter in UNIX. Unlike Windows, UNIX systems do not look at file extension to determine file type. It is not a bad convention, however, to give your files appropriate file extensions to help you and others identify the file type easily.

File paths may be written as ***relative*** or ***absolute*** paths.

Absolute paths are the paths to files starting at the root of the system. They begin with `/` to denote the path starts at the root. They are a guaranteed way to get you to your files.

Relative paths are the paths to files starting at your current location. You can indicate current directory with `.` (dot) and parent directory as `..` (dot dot). The path can break if you start in the wrong place, and thus, are relative paths.

Some examples comparing relative and absolute paths, assuming current location is `/home/ddietz`:

Relative Path	Absolute Path
`file.txt`	`/home/ddietz/file.txt`
`./file.txt`	`/home/ddietz/file.txt`
`files/file.txt`	`/home/ddietz/files/file.txt`
`../gandino/files`	`/home/gandino/files/`
`../../depot/`	`/depot/`
`../gandino/../../home/ddietz/file.txt`	`/home/ddietz/file.txt`

2.2.4 DIRECTORY STRUCTURE AND FILESYSTEMS

In UNIX the ***root*** of the system is /. Unfortunately, there is no strong analogy to this in Windows. In Windows you have distinct filesystems (C:, D:, F:, etc.) representing distinct sets of files and directories.

In UNIX all filesystems are all represented under /. Under here, filesystems may be *mounted* anywhere in the structure. On a simple UNIX system there may only be a single filesystem (/ then only contains a single filesystem), but on clusters and large systems there are multiple filesystems.

Examples:

Path	Filesystem
/	Root OS, single hard drive on login or compute node
/home/	Filesystem with hundreds of hard disks, home directories
/depot/	Large file system (couple terabytes) for research data
/scratch/radon/	Extremely fast filesystem for working/temporary data
/scratch/conte/	Extremely fast filesystem for working/temporary data

2.3 GETTING STARTED

The data files required for this workshop are located in a public directory in the ***Research Data Depot***. You need to have this in your home directory before you start. You can copy the directory by a simple command given below (just replace `username` with your Purdue career account). You need to first log into ***Radon***, e.g.

```
$ ssh username@radon.rcac.purdue.edu
```

then, follow the next steps:

1. Type in the following:
   ```
   $ cp -r /depot/itap/unix101-2016 ~username/BACKUP
   ```
2. Press enter.
   a. Note: the command is case sensitive.
3. Once your cursor (command prompt) comes back to the original position, type: `ls` (will explain this command later)
4. Press enter
5. You should see now:
   ```
   $ ls
   unix101-2016 BACKUP
   ```

2.4 NAVIGATION SHELL

This section will introduce you to some basic file/directory navigation and manipulation techniques.

Which directory/location are you in?

`pwd` (print working directory) command: prints working directory

Type in the following:

```
$ pwd
```

Now you should see **/home/username/**, where `username` should be your Purdue career account.

`pwd` command returns you the present working directory, this means, you are now working in the `username` directory, which is located in `home` directory. The directory that you will be in after logging in is your home directory.

Note: for further reference you can also avoid writing the full path to your home directory by using (~) tilde mark in front of your username.

~**username** same as **/home/username**

Present directory is represented as **.** (dot) and parent directory is represented as **..** (dot dot).

What files are here?

`ls` command: list files in current directory

`-l`	long listing, includes file date and size
`-h`	show file sizes in human readable terms
`-t`	show the newest files first
`-a`	includes hidden files, parent and current directories

Type in the following:

```
$ ls
BACKUP unix101-2016

$ ls -lht
drwxr-xr-x 5 gandino entm 201 Feb 16 17:05 BACKUP
drwxr-xr-x 5 gandino entm 201 Feb 16 17:02 unix101-2016
```

How do I move from one directory to another directory?

`cd` (change directory) command: jump from one directory to another.

Type in the following:

```
$ cd unix101-2016/basic_commands/
$ ls -lh
total 3.4G
-rw-r--r-- 1 gandino entm 2.1K Feb 16 17:02 basic-unix.txt
-rw-r----- 1 gandino entm 46M  Feb 17 13:59 sequences.fasta
-rwxr-xr-x 1 gandino entm 1.3G Feb 16 17:02 SP_R1.fastq
-rwxr-xr-x 1 gandino entm 197M Feb 16 17:02 SP_R1.list
-rwxr-xr-x 1 gandino entm 1.3G Feb 16 17:02 SP_R2.fastq
```

Changes your present location to the parent directory using:

```
cd ..
```

The directory which is up one level in the directory tree can be referred to as `..`

Type in the following:

```
$ cd ..
$ ls
basic_commands protein scripts Shakespeare
```

Few more ways to do the same thing:

```
$ cd unix101-2016/basic_commands/
$ ls -lh
total 3.4G
-rw-r--r-- 1 gandino entm 2.1K Feb 16 17:02 basic-unix.txt
-rw-r----- 1 gandino entm 46M  Feb 17 13:59 sequences.fasta
-rwxr-xr-x 1 gandino entm 1.3G Feb 16 17:02 SP_R1.fastq
-rwxr-xr-x 1 gandino entm 197M Feb 16 17:02 SP_R1.list
-rwxr-xr-x 1 gandino entm 1.3G Feb 16 17:02 SP_R2.fastq
$ cd ~ # going back to the home directory
$ ls
BACKUP unix101-2016
```

```
$ cd unix101-2016/basic_commands/
$ ls -lh
total 3.4G
-rw-r--r-- 1 gandino entm 2.1K Feb 16 17:02 basic-unix.txt
-rw-r----- 1 gandino entm 46M  Feb 17 13:59 sequences.fasta
-rwxr-xr-x 1 gandino entm 1.3G Feb 16 17:02 SP_R1.fastq
-rwxr-xr-x 1 gandino entm 197M Feb 16 17:02 SP_R1.list
-rwxr-xr-x 1 gandino entm 1.3G Feb 16 17:02 SP_R2.fastq
$ cd /home/gandino/  # going back to the home directory
$ ls
BACKUP unix101-2016
```

2.5 MAKING DIRECTORIES

How to create a directory?

`mkdir` command: (make directory) can be used.

Type in the followings:

```
$ unix101-2016/basic_commands/
$ ls -lh
total 3.4G
-rw-r--r-- 1 gandino entm 2.1K Feb 16 17:02 basic-unix.txt
-rw-r----- 1 gandino entm 46M  Feb 17 13:59 sequences.fasta
-rwxr-xr-x 1 gandino entm 1.3G Feb 16 17:02 SP_R1.fastq
-rwxr-xr-x 1 gandino entm 197M Feb 16 17:02 SP_R1.list
-rwxr-xr-x 1 gandino entm 1.3G Feb 16 17:02 SP_R2.fastq

$ mkdir NEW_DIRECTORY
$ ls -lh
total 3.4G
-rw-r--r-- 1 gandino entm 2.1K Feb 16 17:02 basic-unix.txt
drwxr-xr-x 2 gandino entm 0    Feb 18 11:31 NEW_DIRECTORY
-rw-r----- 1 gandino entm 46M  Feb 17 13:59 sequences.fasta
-rwxr-xr-x 1 gandino entm 1.3G Feb 16 17:02 SP_R1.fastq
-rwxr-xr-x 1 gandino entm 197M Feb 16 17:02 SP_R1.list
-rwxr-xr-x 1 gandino entm 1.3G Feb 16 17:02 SP_R2.fastq
```

A good practice in UNIX is to not use spaces in your directory name or any other special characters beside "_" or "-". We'll begin to see why over the next couple of chapters.

Alternatively, you can also directly specify the path where you want to create your new folder, e.g.,

Note: just replace `username` with your Purdue career account.

`mkdir /home/username/basic_commands/NEW_DIRECTORY`

2.6 COPYING FILES AND DIRECTORIES

How to copy a file or a directory?

`cp` (copy) command: copy a file. When using this command you have to provide both the source file and destination file.

`cp [OPTIONS] SOURCE DESTINATION`

Type in the following:

```
$ pwd
/home/gandino/unix101-2016/basic_commands/
$ cd unix101-2016/NEW_DIRECTORY
$ ls
# Note: this directory should be empty!
$ pwd
/home/gandino/unix101-2016/basic_commands/NEW_DIRECTORY
$ cp ../basic-unix.txt .
$ ls
basic-unix.tx
```

You can also specify the absolute path of the source and/or destination file. To know more about any command you can use `man` command, which opens the "manual" of the command you specify, e.g., `man cp`

This opens the manual for the `cp` command. Take a look at the manual of the cp command (use arrow keys to move top or bottom of the page). **To exit, press 'q'**.

OPTIONS are optional parameters that can be used to accomplish more from the same command e.g. by using option `-i` with the regular `cp` command, you can always make sure that you are not overwriting the existing file while copying.

When copying directories make sure to use the option `-R`, `-r`, or `--recursive` to copy directories recursively, what this means is that every file and subdirectories inside that directory will be copied.

Type in the following:

```
$ cp -R ../../../BACKUP BACKUP_FILES
$ ls -lh
total 28K
drwxr-xr-x 6 gandino entm 109 Feb 18 16:35 BACKUP_FILES
-rw-r--r-- 1 gandino entm 2.1K Feb 18 16:30 basic-unix.txt
```

2.7 MOVING DIRECTORIES

How to move a file or a directory?

`mv` (move) command is used to move a file or a directory. Like the cp command, you need to provide both the source file and destination file.

`mv SOURCE DESTINATION`

Absolute path also works fine. Some of the options used by `cp` command also work with `mv` command.

Type in the following:

```
$ pwd
/home/gandino/unix101-2016/basic_commands/NEW_DIRECTORY
$ cd ..
$ pwd
/home/gandino/unix101-2016/basic_commands/
$ mv NEW_DIRECTORY/basic-unix.txt .
$ ls NEW_DIRECTORY/
BACKUP_FILES
```

`mv` can also be used to rename files and directories

`mv OLDNAME NEWNAME`

```
$ pwd
/home/gandino/unix101-2016/basic_commands/
$ mv basic-unix.txt new_basic-unix.txt
$ ls -lh
total 3.4G
-rw-r--r-- 1 gandino entm 2.1K Feb 16 17:02 new_basic-unix.txt
drwxr-xr-x 2 gandino entm 30   Feb 18 17:19 NEW_DIRECTORY
-rw-r----- 1 gandino entm 46M  Feb 17 13:59 sequences.fasta
-rwxr-xr-x 1 gandino entm 1.3G Feb 16 17:02 SP_R1.fastq
-rwxr-xr-x 1 gandino entm 197M Feb 16 17:02 SP_R1.list
-rwxr-xr-x 1 gandino entm 1.3G Feb 16 17:02 SP_R2.fastq
```

2.8 CREATING AND EDITING FILES

How to create and edit a file on the server itself?

`nano new_file`

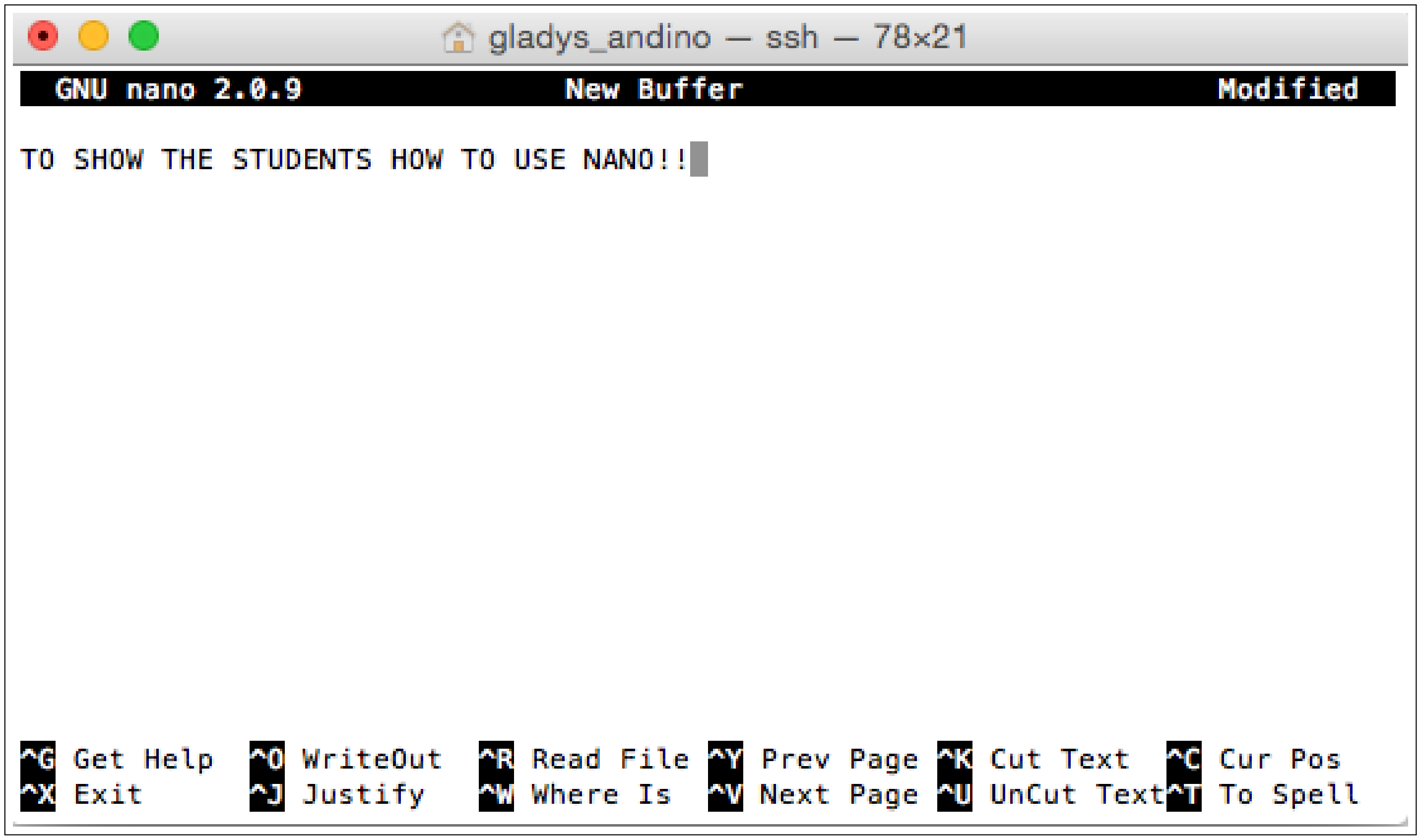
This text editor lets you edit a file. While you can prepare a file on your local computer or laptop and transfer it to the server, it is usually more convenient to create and edit the file on the server itself. There are many editors available (for instance `vi`, `emacs`, `pico` and `nano`), if you are familiar with an editor, feel free to use the one you are already familiar with.

If you aren't familiar with a UNIX editor, we recommend that you use `nano`. An advantage of `nano` are that the commands to "Exit" the editor and "WriteOut" a file are shown at the bottom of the screen (ˆ indicates Ctrl should be held down while typing the letter).

The basics of `nano` are simple (see figure):

- type `nano` at the command prompt to start the editor
- type your commands into the screen
- move around with the arrow keys if necessary

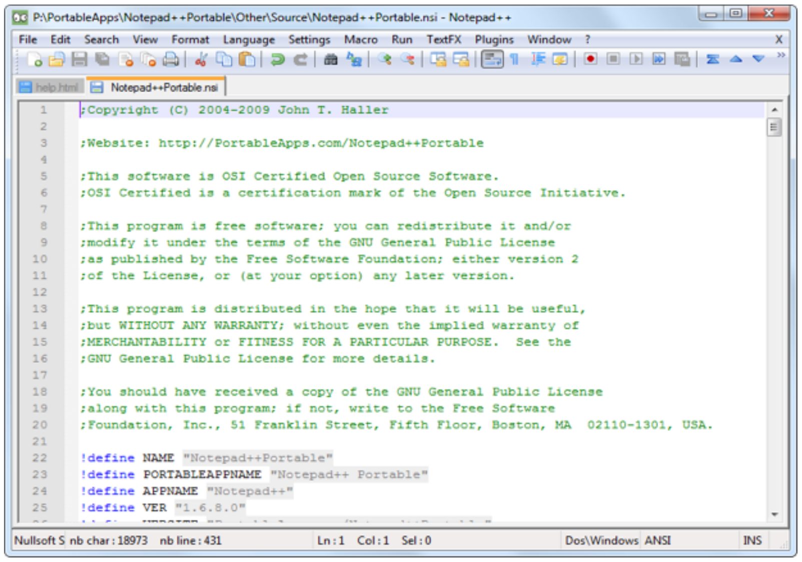
- save your file with the ˆo key, and provide a name for the file
- quit the editor with ˆx



```
gladys_andino — ssh — 78×21
  GNU nano 2.0.9                New Buffer                          Modified

TO SHOW THE STUDENTS HOW TO USE NANO!!

^G Get Help   ^O WriteOut   ^R Read File  ^Y Prev Page  ^K Cut Text   ^C Cur Pos
^X Exit       ^J Justify    ^W Where Is   ^V Next Page  ^U UnCut Text^T To Spell
```

How to create and edit a file using Notepad++?

Notepad++ is a text editor and source code editor for use with Microsoft Windows. Notepad++ supports tabbed editing, which allows working with multiple open files in a single window. It is also an open source editor with nice features (like syntax highlighting).


P:\PortableApps\Notepad++Portable\Other\Source\Notepad++Portable.nsi - Notepad++
File Edit Search View Format Language Settings Macro Run TextFX Plugins Window ?
help.html
Notepad++Portable.nsi

```
;Copyright (C) 2004-2009 John T. Haller

;Website: http://PortableApps.com/Notepad++Portable

;This software is OSI Certified Open Source Software.
;OSI Certified is a certification mark of the Open Source Initiative.

;This program is free software; you can redistribute it and/or
;modify it under the terms of the GNU General Public License
;as published by the Free Software Foundation; either version 2
;of the License, or (at your option) any later version.

;This program is distributed in the hope that it will be useful,
;but WITHOUT ANY WARRANTY; without even the implied warranty of
;MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
;GNU General Public License for more details.

;You should have received a copy of the GNU General Public License
;along with this program; if not, write to the Free Software
;Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301, USA.

!define NAME "Notepad++Portable"
!define PORTABLEAPPNAME "Notepad++ Portable"
!define APPNAME "Notepad++"
!define VER "1.6.8.0"
```

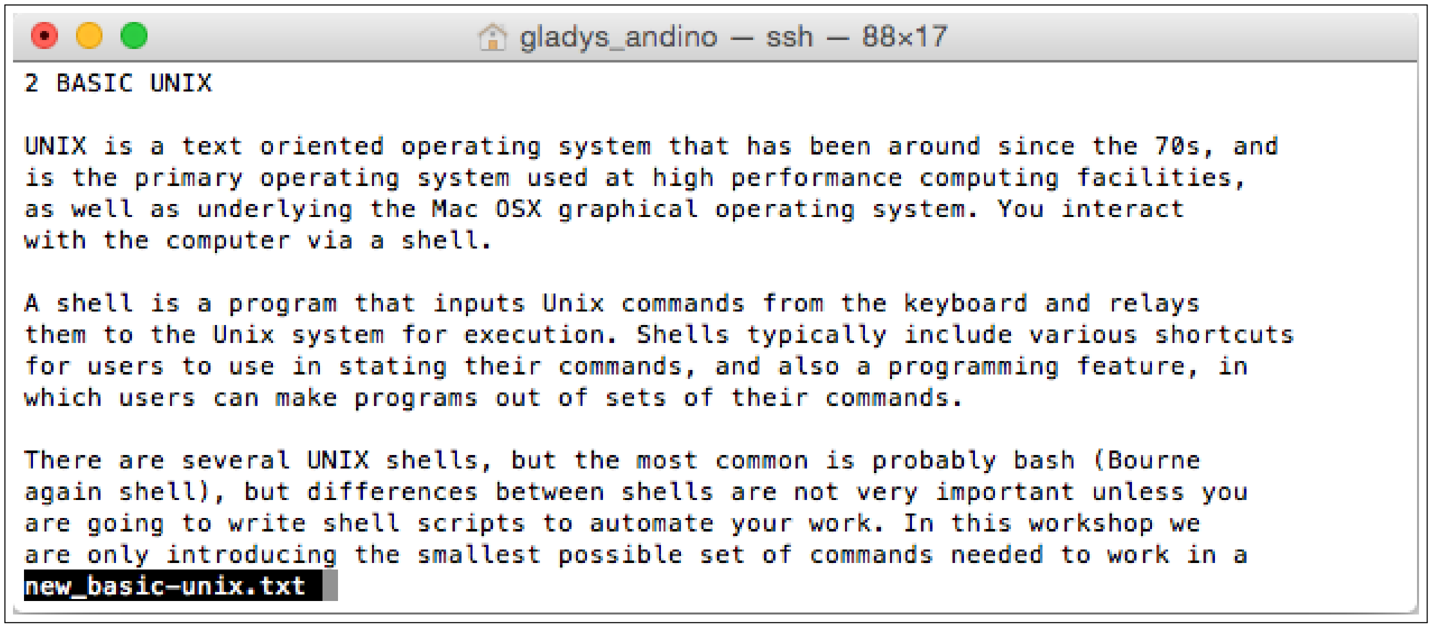
Nullsoft S nb char : 18973 nb line : 431 Ln : 1 Col : 1 Sel : 0 Dos\Windows ANSI INS

2.9 VIEWING CONTENTS OF THE FILES

There are various commands to print the contents of the file in bash. Most of these commands are often used in specific contexts. All these commands when executed with filenames display the contents on the screen. Most common ones are `less`, `more`, `cat`, `head` and `tail`.

2.9.1 `less`

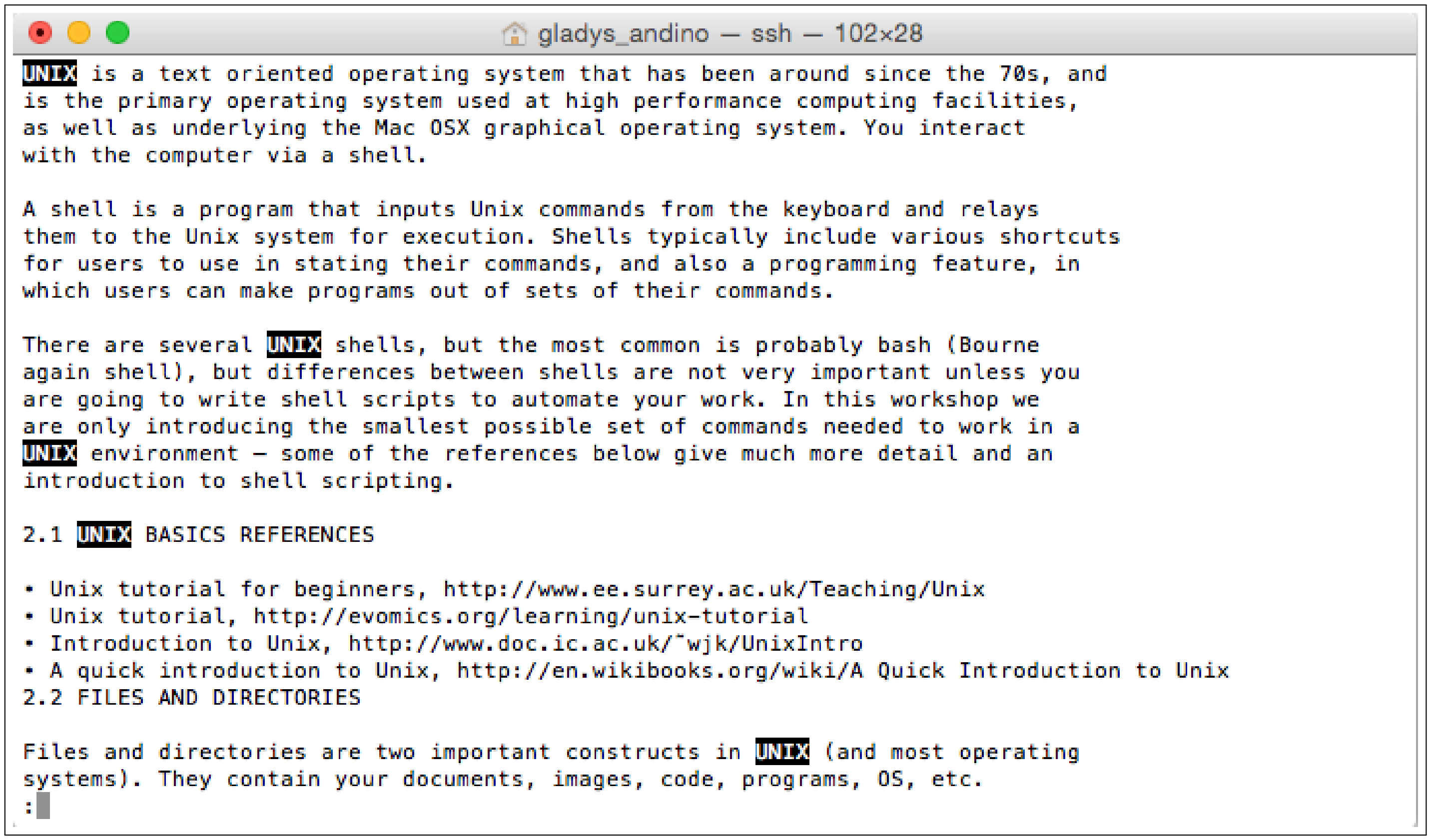
```
$ pwd
/home/gandino/unix101-2016/basic_commands/
$ ls -lh
total 3.4G
-rw-r--r-- 1 gandino entm 2.1K Feb 16 17:02 new_basic-unix.txt
drwxr-xr-x 2 gandino entm 30   Feb 18 17:19 NEW_DIRECTORY
-rw-r----- 1 gandino entm 46M  Feb 17 13:59 sequences.fasta
-rwxr-xr-x 1 gandino entm 1.3G Feb 16 17:02 SP_R1.fastq
-rwxr-xr-x 1 gandino entm 197M Feb 16 17:02 SP_R1.list
-rwxr-xr-x 1 gandino entm 1.3G Feb 16 17:02 SP_R2.fastq
$ less new_basic-unix.txt
```



```
2 BASIC UNIX

UNIX is a text oriented operating system that has been around since the 70s, and
is the primary operating system used at high performance computing facilities,
as well as underlying the Mac OSX graphical operating system. You interact
with the computer via a shell.

A shell is a program that inputs Unix commands from the keyboard and relays
them to the Unix system for execution. Shells typically include various shortcuts
for users to use in stating their commands, and also a programming feature, in
which users can make programs out of sets of their commands.

There are several UNIX shells, but the most common is probably bash (Bourne
again shell), but differences between shells are not very important unless you
are going to write shell scripts to automate your work. In this workshop we
are only introducing the smallest possible set of commands needed to work in a
new_basic-unix.txt
```

Displays file contents on the screen with line scrolling (to scroll you can use ‘arrow’ keys, ‘PgUp/PgDn’ keys, ‘space bar’ or ‘Enter’ key). **When you are done press ‘q’ to exit.**

You can search a pattern inside of the screen using /pattern. Try this: /UNIX



```
UNIX is a text oriented operating system that has been around since the 70s, and
is the primary operating system used at high performance computing facilities,
as well as underlying the Mac OSX graphical operating system. You interact
with the computer via a shell.

A shell is a program that inputs Unix commands from the keyboard and relays
them to the Unix system for execution. Shells typically include various shortcuts
for users to use in stating their commands, and also a programming feature, in
which users can make programs out of sets of their commands.

There are several UNIX shells, but the most common is probably bash (Bourne
again shell), but differences between shells are not very important unless you
are going to write shell scripts to automate your work. In this workshop we
are only introducing the smallest possible set of commands needed to work in a
UNIX environment - some of the references below give much more detail and an
introduction to shell scripting.

2.1 UNIX BASICS REFERENCES

• Unix tutorial for beginners, http://www.ee.surrey.ac.uk/Teaching/Unix
• Unix tutorial, http://evomics.org/learning/unix-tutorial
• Introduction to Unix, http://www.doc.ic.ac.uk/˜wjk/UnixIntro
• A quick introduction to Unix, http://en.wikibooks.org/wiki/A Quick Introduction to Unix
2.2 FILES AND DIRECTORIES

Files and directories are two important constructs in UNIX (and most operating
systems). They contain your documents, images, code, programs, OS, etc.
:
```

2.9.2 more

Like `less` command, also, displays file's content on the screen with line scrolling but uses 'space bar' or 'Enter' key to scroll. **When you are done press 'q' to exit.**

```
more FILENAME
```

2.9.3 cat

This is the simplest form of displaying contents. It catalogs the entire contents of the file on the screen. In case of large files, entire file will scroll on the screen without pausing.

```
cat FILENAME
```

2.9.4 head

Displays only the starting lines of a file. The default is first ten lines. But, any number of lines can be displayed using `-n` option (followed by required number of lines).

```
head FILENAME
```

2.9.5 `tail`

Similar to `head`, but displays the last 10 lines. `-n` option can be used to change this. More information about any of these commands can be found in the man pages.

```
tail FILNAME
```

2.9.6 grep

`grep` is one of the most commonly used commands in UNIX and it is commonly used to filter a file/input, line by line, against a pattern (e.g., to print each line of a file which contains a match for pattern).

General syntax:

```
grep [OPTIONS] PATTERN FILENAME
```

(if given no file, it reads from the standard input)

Like any other command there are various options available for this command. Most useful options include:

pattern	
–i	ignore case for the pattern matching.
-l	lists the file names containing the pattern (instead of match)
-c	counts the number of matches for a pattern
-w	forces the pattern to match an entire word

Some typical scenarios to use grep:

- Extracting specific line(s) from the simulation output
- Stripping header/footer/comments lines from an input file
- Selecting files of interest (with -l)
- Counting number of occurrences of a string or pattern in a file (with –c)

A handy trick for bioinformaticians: how many sequences are in a FASTA-formatted file? By definition, each sequence record in a FASTA file has one line of description that always starts with >, followed by multiple lines of sequence itself. Each sequence record ends when the next line starting with > appears:

So counting number of sequences in FASTA-file is as easy as
grep -c '^>' FILENAME

```
$ pwd
/home/gandino/unix101-2016/basic_commands/
$ grep -c '>' sequences.fasta
31925
```

2.10 DELETING FILES AND DIRECTORIES

How to remove a file or a directory?

2.10.1 `rmdir`

The `rmdir` (remove directory) command is used to delete directories from the system. `rmdir` DIRECTORY

The directory should be empty before you use the `rmdir` command.

2.10.2 `rm`

`rm`(remove) command is used to delete directories or files from the system.

`rm` FILE

Some useful options include:

`-r`	recursively delete files
`-f`	delete forcefully

```
$ pwd
/home/gandino/unix101-2016/basic_commands/
$ ls -lh
total 3.4G
-rw-r--r-- 1 gandino entm 2.1K Feb 16 17:02 new_basic-unix.txt
drwxr-xr-x 2 gandino entm 30   Feb 18 17:19 NEW_DIRECTORY
-rw-r----- 1 gandino entm 46M  Feb 17 13:59 sequences.fasta
-rwxr-xr-x 1 gandino entm 1.3G Feb 16 17:02 SP_R1.fastq
-rwxr-xr-x 1 gandino entm 197M Feb 16 17:02 SP_R1.list
-rwxr-xr-x 1 gandino entm 1.3G Feb 16 17:02 SP_R2.fastq

$ rm NEW_DIRECTORY/ # this directory it's not empty!
rm: cannot remove `NEW_DIRECTORY/': Is a directory
$ rm -r NEW_DIRECTORY/
$ ls -lh
total 3.4G
-rw-r--r-- 1 gandino entm 2.1K Feb 16 17:02 new_basic-unix.txt
-rw-r----- 1 gandino entm 46M  Feb 17 13:59 sequences.fasta
-rwxr-xr-x 1 gandino entm 1.3G Feb 16 17:02 SP_R1.fastq
-rwxr-xr-x 1 gandino entm 197M Feb 16 17:02 SP_R1.list
-rwxr-xr-x 1 gandino entm 1.3G Feb 16 17:02 SP_R2.fastq
```

2.11 FILE TRANSFER

2.11.1 SAMBA

Windows:

- Windows 7: Click Windows menu > Computer, then click Map Network Drive in the top bar.
- Windows 8.1: Tap the Windows key, type **computer**, select This PC, click Computer > Map Network Drive in the top bar.
- In the folder location enter the following information and click Finish:
- To access your home directory, enter

  \\samba.rcac.purdue.edu\myusername where **myusername** is your career account name.

- To access your scratch space on Radon, enter

  \\samba.rcac.purdue.edu\scratch. Once mapped, you will be able to navigate to **radon\m\myusername** where **m** is the first letter of your username and **myusername** is your career account name. You may also navigate to any of the other cluster scratch directories from this drive mapping.

- You may be prompted for login information. Enter your username as **onepurdue\myusername** and your account password. If you forget the **onepurdue** prefix it will prevent you from logging in.
- Your home or scratch directory should now be mounted as a drive in the computer window.

Mac OS X:

- In the Finder, click Go > Connect to Server
- In the Server Address enter the following information and click Connect:
- To access your home directory, enter

  smb://samba.rcac.purdue.edu/myusername
  where **myusername** is your career account name.

- To access your scratch space on Radon, enter

  smb://samba.rcac.purdue.edu\scratch. Once connected, you will be able to navigate to **radon\m\myusername** where **m** is the first letter of your username and **myusername** is your career account name. You may also navigate to any of the other cluster scratch directories from this mount.

- To access your Fortress long-term storage home directory, enter

  smb://fortress-smb.rcac.purdue.edu/myusername
  where **myusername** is your career account name.

- To access a shared Fortress group storage directory, enter

  smb://fortress-smb.rcac.purdue.edu/group/mygroupname
  where **mygroupname** is the name of the shared group space.

For more information about file transfer via Samba please see:
https://www.rcac.purdue.edu/compute/radon/guide/#storage_transfer_cifs

2.11.2 SCP

SCP (Secure CoPy) is a simple way of transferring files between two machines that use the SSH (Secure SHell) protocol. You may use SCP to connect to any system where you have SSH (login) access.

SCP is available as a protocol choice in some graphical file transfer programs and also as a command line program on most Linux, UNIX, and Mac OS X systems. SCP can copy single files, but will also recursively copy directory contents if given a directory name. scp can be used as follows:

```
scp sourcefile username@radon.rcac.purdue.edu:somedirectory/
```
(to a remote system from local)
```
scp username@radon.rcac.purdue.edu:somedirectory/sourcefile destinationfile
```
(from a remote system to local)
```
scp SourceDirectory/ username@radon.rcac.purdue.edu:somedirectory/
```
(recursive directory copy to a remote system from local)

Microsoft Windows:

- WinSCP is a full-featured and free graphical SCP and SFTP client.
- PuTTY also offers ”pscp.exe”, which is an extremely small program and a basic SCP client.
- Secure FX is a commercial SCP and SFTP client, which is freely available to Purdue students, faculty, and staff with a Purdue career account.

- As an exercise we will download WinSCP and copy a file to the clusters.
- If you do not have WinSCP installed, then follow these steps:

1. Use Google to search for the word “WinSCP”
2. Click the WinSCP download page, then click install package
    a. Install WinSCP
3. Double Click WinSCP
4. In the “Host Name” box type radon.rcac.purdue.edu
5. Click YES when the program asks if you want to cache the security key
6. Use your career account credentials to login
7. Now you can copy files back and forth from your desktop

Mac OS X:

- You will find the scp command pre-installed in your system. You may start a local terminal window from ”Applications>Utilities”.

Linux / Solaris / AIX / HP-UX / Unix:

- You will find the pre-installed ”scp” command in your system.

3 ADVANCED FILE MANIPULATION COMMANDS

In this section we’ll look at compressing and bundling files for storage and archiving, backing them up to the tape library for long-term storage, and discuss file access permissions.

3.1 FILE PERMISSIONS

All files in the UNIX system have a set of permissions, which define what actions can be done with that file and by whom. ***Read***, ***write***, and ***execute*** permissions are set on each file. Each file also has a ***user*** and ***group*** that own the file. They are denoted as:

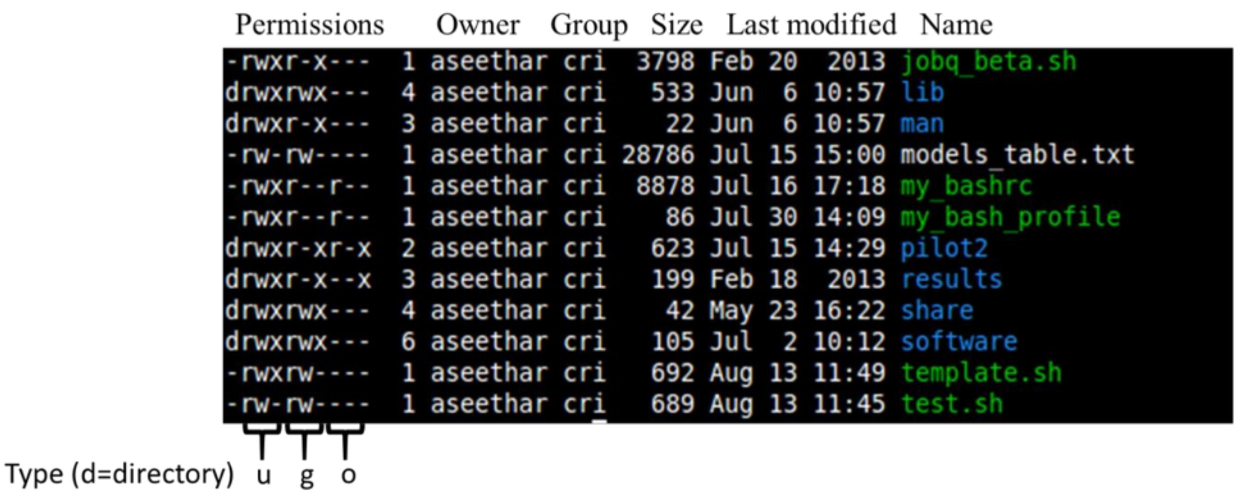
`read (r)`	Read or view contents
`write (w)`	Write to or modify contents
`execute (x)`	Run or execute script or program

These three levels of permissions can be applied to the following classifications of users:

`owner (u)`	User that owns the file
`group (g)`	Group (of users) that own the files
`others (o)`	Everyone who is not owner or in group
`all users (a)`	All users, including owner, group, and others

To look at the permissions for any file, you can list the files with `-l` option to ls:

```
ls -l
```



```
-rwxr-x---  1 aseethar cri  3798 Feb 20  2013 jobq_beta.sh
drwxrwx---  4 aseethar cri   533 Jun  6 10:57 lib
drwxr-x---  3 aseethar cri    22 Jun  6 10:57 man
-rw-rw----  1 aseethar cri 28786 Jul 15 15:00 models_table.txt
-rwxr--r--  1 aseethar cri  8878 Jul 16 17:18 my_bashrc
-rwxr--r--  1 aseethar cri    86 Jul 30 14:09 my_bash_profile
drwxr-xr-x  2 aseethar cri   623 Jul 15 14:29 pilot2
drwxr-x--x  3 aseethar cri   199 Feb 18  2013 results
drwxrwx---  4 aseethar cri    42 May 23 16:22 share
drwxrwx---  6 aseethar cri   105 Jul  2 10:12 software
-rwxrw----  1 aseethar cri   692 Aug 13 11:49 template.sh
-rw-rw----  1 aseethar cri   689 Aug 13 11:45 test.sh
```

3.1.1 chmod

To set or modify a file's permissions you need to use the `chmod` command. Only the owner of a file can alter a file's permissions.

General syntax:

```
chmod [OPTIONS] CLASSIFICATION[+/-]PERMISSIONS FILENAME
```

`OPTIONS` include

`-R` recursively (the permissions are applied to all files and directories inside the directory)

Add permissions:

```
chmod CLASSIFICATION+PERMISSIONS FILENAME
```

```
# grants read, write and execute permissions for group
$ chmod g+rwx FILENAME

# grants read permission for others
$ chmod o+r FILENAME

# makes the file publically readable, writable, and executable by anyone
$ chmod a+rwx FILENAME
```

Remove permissions:

`chmod CLASSIFICATION-PERMISSIONS FILENAME`

```
# removes write and execute permissions for others
$ chmod o-wx FILENAME

# removes all permissions for group
$ chmod g-rwx FILENAME

# removes execution permissions from all users
$ chmod a-x FILENAME
```

Task: Check the permissions for the files located in the Shakespeare directory.

Questions:

What permissions does the "others" have on these files?

Which group owns your files?

By default, your home directory and scratch space are readable only by you (do ls –la $HOME to check). Whether to give others read access or not is entirely up to you. We don't have any recommendation here (the choice of privacy vs. ease of collaboration with lab mates is yours), but if you do decide to change it, make sure you understand the consequences before you proceed.

3.1.2 chown

The `chown` command is used to change the owner and group of files or directories.

By default, the owner of a file or directory is the user that created it. The group is a set of users that share the same access permissions (i.e., read, write and execute) for that object.

General syntax:

`chown [OPTIONS] NEW_OWNER FILENAME`

`NEW_OWNER` is the username of the new owner and `FILENAME` is the name of the target file or directory. The ownership of any number of files or directories can be changed simultaneously.

`OPTIONS` include:

- `-R` recursively (the permissions are applied to all files and directories inside the directory)

```
# change the ownership of a file named file1 and a directory named dir1 to a new owner alice:

$ chown alice file1 dir1

# change the group ownership of a file named file1 to a group named group1:

$ chown :group1 file1

# change the owner of a file named file2 to user bob and change its group ownership to group2:

$ chown bob:group2 file2
```

You may find more information about `chown` here: *http://www.linfo.org/chown.html*

3.1.3 chgrp

The command `chgrp` changes group ownership of a file or files.

General syntax:

```
chgrp [OPTIONS]...  GROUP FILENAME...
chgrp [OPTIONS]...  --reference=RFILENAME FILENAME...
```

The first changes the group of each `FILENAME` to `GROUP`.

The second, with `--reference`, changes the group of each `FILENAME` to that of `RFILENAME`.

`OPTIONS` include:

`-R`	recursively (the permissions are applied to all files and directories inside the directory)

```
# change the group ownership of the file named file.txt to the group group1:

$ chgrp group1 file.txt

# change the group ownership of the directory named dir1, and all files and directories inside
   dir1, to the group group2:

$ chgrp -R group2 dir1
```

3.2 COMPRESSING FILES

There are several options for archiving and compressing groups of files or directories. Compressed files are not only easier to handle (copy/move) but also occupy less size on the disk (less than 1/3 of the original size). In ITaP Research Computing systems you can use `tar`, `gzip` or `zip` for packing and compressing files/directories.

3.2.1 tar ARCHIVING/EXTRACTION

`tar` utility (stands for 'tape archive') saves many files and directories together into a single archive file, and restores individual files from the archive. It also includes automatic archive compression/decompression options and special features for incremental and full backups. While zip can do this, too, tar is considered de-facto standard of packaging multiple files and directories on Unix.

General syntax:

```
tar ACTION [OPTIONS] files or directories
```

Common actions: `-c` (create), `-t` (test), `-x` (extract)
Common options:
`-v` verbose
`-z` use `gzip` compression
`-j` use `bzip2` compression
`-f` filename path to archive

tar -**c**vf OUTFILE.tar INFILE

archive single file INFILE (possible, but rarely used for single files)

tar -**c**zvf OUTFILE.tar.gz INFILE

archive and compress file INFILE (now, this starts making sense)

tar -**c**zvf OUTFILE.tar.gz DIRECTORY

archive and compress all files in a directory into one archive file *(most common #1)*

tar -**c**zvf OUTFILE.tar.gz *.txt

archive and compress all ".txt" files in current directory into one archive file

tar -**t**vf SOMEFILE.tar

list contents of archive SOMEFILE.tar (-tzvf if compressed)

tar -**x**vf SOMEFILE.tar

extract contents of SOMEFILE.tar

tar -**x**zvf SOMEFILE.tar.gz

extract contents of gzipped archive SOMEFILE.tar.gz *(most common #2)*

Task: Archive and compress the BACKUP_FILES directory you created earlier (you can name it backup.tar.gz or anything you want)

3.2.2 gzip COMPRESSION/EXTRACTION

gzip compression utility (note: not archiving! acts on a single file at a time) was designed as a replacement for compress, with better compression and no patented algorithms. This is a very common compression command for all UNIX systems.

gzip SOMEFILE

to compress SOMEFILE into SOMEFILE.gz (also removes original uncompressed file)

gunzip SOMEFILE.gz

to uncompress SOMEFILE.gz (also removes the compressed file)

Task: gzip the file Hamlet.txt and examine the size. gunzip it back so that you can use this file for the later exercises.

3.2.3 zip ARCHIVING/EXTRACTION

ZIP archiving and compression format is a commonly used archive format, so you may have a .zip file thrown at you occasionally. Linux has a pair of zip and unzip utilities to handle it.

zip OUTFILE.zip INFILE.txt

Compress and archive INFILE.txt

zip -r ARCHIVE.zip DIRECTORY

Compress and archive all files in a DIRECTORY into ARCHIVE.zip

zip -r ARCHIVE.zip . -i *.txt

Compress and archive all txt files in a '.' and below into ARCHIVE.zip

unzip -l ARCHIVE.zip

List the contents of an archive

unzip ARCHIVE.zip

Uncompress and extract file(s) from archive

Task: Archive and compress the BACKUP_FILES directory you created earlier (you can name it backup.zip or anything you want).

3.3 BACKING UP FILES

Where do you put archived and/or compressed gigabytes and terabytes of data that you need to keep for a very long time? Flash drives, hard drives, DVDs, CDs... magnetic tape! In terms of dollars/gigabyte, tape is still the cheapest option for long-term archival storage of digital objects. ITaP maintains a large archival storage system called Fortress (*https://www.rcac.purdue.edu/storage/fortress/*).

Fortress has nearly unlimited capacity, no quotas and is free for Purdue researchers. It is a long-term, multi-tiered file caching and storage system consisting both of online disks and robotic tape drives. Fortress is fast on the uptake, but quite slow on the return (remember, it is an archive, not a working copy!). Fortress really shines when dealing with a relatively small number of large files (1 GB and up), as opposed to millions of small files.

Fortress is a standalone system that is not directly mounted on any of ITaP resources and it requires special tools to access it. Two most recommended tools are command-line utilities htar and hsi, as well as the *Globus* service.

3.3.1 ACCOUNTS AND DIRECTORIES ON FORTRESS

Accounts on the Fortress system come automatically with accounts on any of ITaP's Research Computing resources. Just as cluster account holders have cluster home directory (/home/myusername), each Fortress user has a Fortress home directory on the Fortress system. Your Fortress home directory also has the path of /home/myusername, but remember that this is a totally different filesystem than your home directory.

There are several ways of accessing Fortress from the command line. For the sake of simplicity, we will concentrate on just a handful commands for the most typical use cases.

See Fortress User Guide (*https://www.rcac.purdue.edu/storage/fortress/guide/*) and the hsi/htar homepage (*http://www.mgleicher.us/hsi/* and *http://www.mgleicher.us/htar/*) for more details.

3.3.2 USING hsi AND hatr TO ARCHIVE/EXTRACT FILES FROM FORTRESS

In this method, we will be using htar as the main archiving/extraction tool, and hsi for all auxiliary manipulations. htar is designed to closely mimic tar, and hsi mimics many of the UNIX shell commands.

Main htar commands (very much like tar, just no –z option):

- Create a Fortress archive from local files:

```
htar --cvf FORTRESSNAME.tar LOCALFILE(s)
```

- List contents of a Fortress archive:

```
htar --tvf FORTRESSNAME.tar
```

- Extract all files from a Fortress archive:

```
htar --xvf FORTRESSNAME.tar
```

- Extract specific files from a Fortress archive:

```
htar --xvf FORTRESSNAME.tar LOCALFILE(s)
```

Archive files created by htar are compatible with the regular tar and vice versa.

Main hsi commands (in the following examples you could also use hsi –q instead of hsi to cut on the clutter from extra header lines):

Task Backup the BACKUP directory you created in Section 2.

At the last command, note the auxiliary file backup.tar.idx along the side of the archive backup.tar. Do not remove it! This index file is a killer feature of htar: it is created automatically by htar and contains tape offsets for all files in the archive. This means that if you ever need a listing of the archive (-tvf), htar will not have to scan all archive from tape (i.e. it'll be fast!). Similarly, if you ever need to extract something, this index file will make tape library's life easier.

Task Restore the BACKUP directory in a different location from the Fortress archive you created in the last task.

`hsi ls [-l] NAME`	List the contents of remote directory (on Fortress). If no argument is given, your Fortress home directory is assumed.
`hsi mkdir DIRNAME`	make subdirectory on Fortress.
`hsi cp SOURCE DEST`	Copy/move files within Fortress. Note: both `SOURCE` and `DEST` are on Fortress (not between Fortress and your current machine!)
`hsi mv SOURCE DEST`	move files within Fortress. Note: both `SOURCE` and `DEST` are on Fortress (not between Fortress and your current machine!)
`hsi rm FILENAME`	Remove file on Fortress [CAREFUL! Forever!] Also `rmdir` to remove directories.
`hsi put LOCAL`	Put a local file (from your machine) on your Fortress home directory. Optionally, remote destination dir or name on Fortress can be specified after a `:` separator: `hsi put LOCAL.tar : FORTRESSNAME`
`hsi get FORTRESSFILE`	Get a remote file (from Fortress) to your current directory on your machine. Optionally, remote source directory or name on Fortress can be specified after a `:` separator: `hsi get LOCALNAME : FORTRESS.tar`

3.4 PIPES AND REDIRECTS

A common design philosophy of UNIX is to have a set of tools where each one does just one thing, but does it well. Another design decision is to allow programs to interact and pass data to each other. This lets users build complex processing pipelines from smaller building blocks.

Many UNIX commands use some input file (or text) and display the output on the screen. This is feasible when the data being displayed is small enough to fit the screen or if it is the endpoint of your analysis. For large data outputs, it is efficient to redirect to a file instead of screen (or to another program). This can be done very easily in UNIX using > (greater than) or < (lesser than), >> or | signs.

- < redirects the data to the command for processing
- > redirects the data from the command's output to a file. The file will be created if it does not exist and if present it will be overwritten with the new output data (you will lose the original file).
- >> similar to >, but appends to rather than overwrite the destination
- Another special operator | (called pipe) is used to pass the output from a command to another command as input before sending it to an output file or display. This is how we can daisy-chain simple tools for complex tasks.

Some examples:

cat FILE1 > FILE2

Creates a new file FILE2 with same contents as old file FILE1. What a fancy cp!

cat < FILE1 > FILE2

Even fancier cp!

cat FILE1 >> FILE2

Appends the contents of FILE1 to FILE2 (equivalent to opening FILE1, copying all the contents, pasting the copied contents to the end of the FILE2 and saving it… all in two keystrokes).

cat FILE1 | less

Here, cat command displays the contents of the FILE1, but instead of sending it to standard output (the screen) it sends it through the pipe to the next command less so that contents of the file are now displayed on the screen with line scrolling.

Task: The Shakespeare directory contains a number of files and each of these files contains a single tragedy text. Combine them all together to make a single file tragedies.txt using redirects.

cat *.txt >> tragedies.txt

this command will combine all .txt files into one.

4 INTRODUCTION TO BASH PROGRAMMING

In this section, we'll look at using text files containing commands to write our own simple programs in the command shell.

A shell script is simply a text file containing commands that are to be executed by the same shell that executes the commands you type at a prompt. We can do more complex things with a shell script, for instance, we can pass it parameters to operate on and we can define variables to use within it. Essentially, shell scripts are just a list of commands to execute. They allow you to write more complex programs in an easy to understand file - imagine trying to jam a complex program into a single line!

4.1 MAKE A SCRIPT EXECUTABLE

The first line of a script should define the program that will interpret the script. In our case it will be bash, the shell program. The first line of any bash shell script should be:

```
#!/bin/bash
```

You might also see the form #!/bin/bash –l to indicate the shell should treat this script exactly like a login session, and you might see the form #!/usr/bin/env bash, which is slightly more portable between computers, but is not necessary on the Purdue Clusters.

In any form, when you execute a bash script your shell will look at this first line to determine which program should be called that can understand and interpret the commands that follow. This can also be applied to other types of scripts such as Python and Perl scripts.

The second step in making a script executable is to change its permission using the chmod command:

```
chmod +x FILENAME
```

For instance, will make the script named FILENAME executable (but don't forget the #!/bin/bash inside the file, too).

4.2 A SIMPLE SHELL SCRIPT

Now let's look at a simple example:

```
#!/bin/bash

echo "Hithere"
echo "files␣in␣this␣directory␣are:"
ls -l
```

You can find this first example script in scripts/script1.

Use `cd` to get to your unix101-2015 directory and run it from there with:

¡¡¡¡¡¡ HEAD `cd steve/script1` ======= scripts/script1 ¿¿¿¿¿¿¿ faecb038b1b690bfe7cba31f1045bee86b8a8b30

You'll see it just executes each of its lines, one after the other.

4.3 SIMPLE VARIABLES

We can add some complexity by defining variables within the script and by using some variables that the system defines for us automatically. Note that we **define** variables as plain words, but we **use** variables by adding a $ as their first character.

Look at scripts/script2:

```
#!/bin/bash

# the '#' means this is a comment and won't be executed
# the 'USER' variable is automatically defined (and so are some others!)

GREETING="Hithere"
TEXT="files␣in␣this␣directory␣are:"

echo $GREETING
echo $USER
echo $TEXT
ls -l
```

Now we've introduced comments, and we use the variables GREETING and TEXT as placeholders for the text we want to echo, and we can use the variable USER which the system has defined for us (it's just our username).

4.4 COMMAND LINE ARGUMENTS

And in scripts/script3:

```
#!/bin/bash

# the '#' means this is a comment and won't be executed

GREETING="Hithere"
TEXT="Your␣first␣parameter␣is"
NUMPAR="You␣typed␣this␣many␣parameters:"

# $1 through $9 are parameters the script reads from its own command line


echo $GREETING

echo $TEXT
echo $1

echo $NUMPAR
echo $#

# '$# is a special variable that tells us how many parameters there are
```

we see that there are variables automatically defined based on the command line we use, so if we call script3 as:

¡¡¡¡¡¡ HEAD `steve/script3 hello there` ======= scripts/script3 hello there ¿¿¿¿¿¿¿ faecb038b1b690bfe7cba31f1045bee86b8

We will see it pick out the first parameter hello using the variable $1. Edit your copy of script3 to pick out a different parameter ($2 or $3, for instance) and try it with different command lines. Try it with:

¡¡¡¡¡¡ HEAD `steve/script3 ``hello there'' everyone` ======= scripts/script3 “hello there” everyone ¿¿¿¿¿¿¿ faecb038b1b690bfe7cba31f1045bee86b8a8b30

How many parameters was that? Why? See if you can discover any other parameter parsing rules.

Here are the contents of scripts/script3:

4.5 SCRIPT SUMMARY

It's impossible to fully describe the capabilities of bash scripts in this limited workshop. The bash language has the ability to do arithmetic, to compare things, and to execute different commands based on those comparisons. In short, it is a complete scripting language just like Python or Perl. We've only barely touched the surface on the capabilities of scripting.

5 REGULAR EXPRESSIONS AND ADVANCED COMMANDS

5.1 REGULAR EXPRESSIONS

5.1.1 INTRODUCTION

When working with text-based files (be that protein structures, DNA sequences or CFD flow simulation results) we are often interested to see if a particular feature is present or not, or pull these features out of the text. These could be various things, like a specific output line, text string, or pattern. In UNIX all strings of text that follow some pattern can be searched using a formula called a regular expression.

For example, if a bioinformatician looks for a particular pattern in large number of sequences, they can create a regular expression and search all the sequences having that pattern relatively easily. Regular expressions consists of a string of regular characters and metacharacters that describe the pattern of text you wish to match.

Some common commands that can be used to manipulate text using regular expressions are grep (filters input against a pattern), sed (applies transformation after searching a pattern) and awk (manipulates data arranged in lines and columns). We will discuss these commands in detail in a later section.

In developing regular expressions, it is helpful to understand how one is processed by the computer. A regular expression is a type of deterministic finite automaton (DFA), or a state machine. While the intricate details and understanding of these is beyond the scope of this document, it is helpful to understand the basic idea. A DFA takes an input and works through it step by step, following a flow chart of sorts. The machine starts at the first character of the input, and is presented with a set of statements to evaluate. Depending on the outcome of the evaluation, the machine then proceeds down that branch and continues to the next set of statements. We will introduce a few illustrations throughout this section to help illustrate and further clarify this concept.

In this section, all regular expressions will be delimited by a `/` at the beginning and end of the expression. This is a typical way to delimit regular expressions in the aforementioned commands. Matches in a sample text will be underlined and bolded.

5.1.2 SUPER SIMPLE EXPRESSIONS

In the following examples we will use the following input text, a quote from Shakespeare's *Hamlet*:

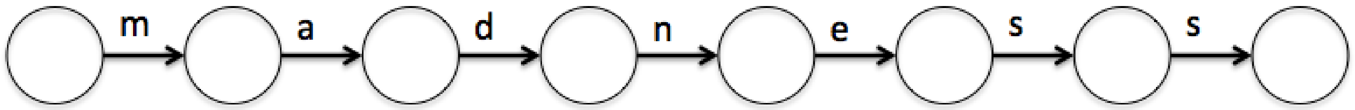
```
Though this be madness, yet there is method in it.
```

At the very simplest, a regular expression could simply be a string:

```
/madness/
```

Though this be **<u>madness</u>**, yet there is method in it.

As we see, this simple regular expression simple matches a plain string in the input. We can explain the expression with:



The processor starts with the circle on the left and the first character of the input string. Here it has one option to evaluate - is the character an ‘m’? The first character is a ‘T’, so no. This will fail to proceed down the expression, so it will move to the next character in the input string and start again.

Eventually it will reach the ‘m’ in “madness” and be able to proceed to the second circle. Then it evaluates the next input character - is it a ‘a’? Yes, and so it proceeds to the third circle, and so on. If the processor is able to reach the final circle, it will have found a match that passes the entire expression.

This is a very simple expression, but as we see expressions will match exactly what we tell it to match. Let’s try to match the word “is”:

/is/

Though th**is** be madness, yet there **is** method in it.

As we see, trying to search for the word “is” also matches the second half of the word “this”. This simple regular expression does not understand word boundaries and blindly matches all occurrences of “is”, no matter where they appear in the input text.

We can try and make the search more specific:

/ is /

Though this be madness, yet there** is **method in it.

This pattern is more stringent and requires a space before and after the word. This method is a bit better, but not perfect. This pattern would fail if the desired word fell at the end of a sentence, or the word is otherwise immediately followed by punctuation (a period or comma, and not a space like we required in the pattern). This pattern also matches the space characters, which may not be desirable if you are trying to extract only the words and not extra word boundary characters. We will try to refine these methods in later sections.

5.1.3 CHARACTER GROUPS

In the following examples we will use the following input text, a quote from Shakespeare’s *Hamlet*:

Though this be madness, yet there is method in it.

In the previous section we attempted to pick out a whole word and avoid partial matches. We attempted to define a whole word by requiring spaces be present on either side, however, we see some pitfalls if we use this method to match the word “madness”:

/ madness /

This pattern would fail to match anything in the input string. The word “madness” is present, however, it is immediately followed by a comma and not a space. We could replace the last space with a comma in the pattern, and while it would work on this particular input string it would fail if the word appeared normally in another sentence, i.e., not followed by a comma.

We could define a regular expression that requires the word be followed by one of several characters by using *character groups*, delimited by `[]`:

```
/ madness[,.  ]/
```

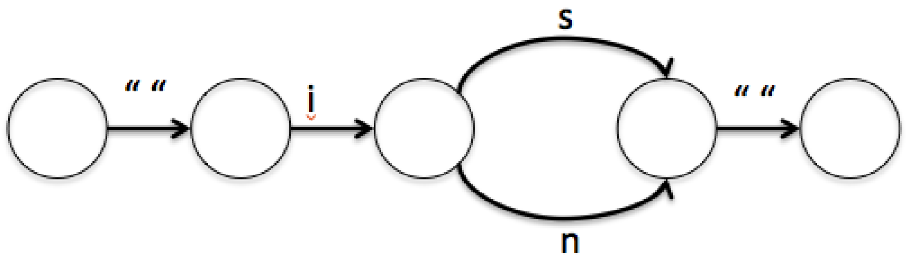
`Though this be `**`madness,`**` yet there is method in it.`

Now this pattern matches the word because it is followed by **one** of the characters inside the brackets. The expression will match if one of the characters is present. This expression also has the pitfall that it also matches and returns the punctuation. We'll refine this later.

Perhaps a more useful expression:

```
/ i[sn] /
```

To help explain the idea behind character groups:



After the parser matches "i", it is presented with two options to proceed to the next state. If it finds and 's' or an 'n' then it is able to proceed onwards. This results in the match:

`Though this be madness, yet there`** is **`method`** in **`it.`

This pattern matches both " is " and " in ". We specify the space, an "i" then either an "s" or an "n".

Character groups can also specify ranges of characters:

```
/[A-Z]/
```

`T``hough this be madness, yet there is method in it.`

This character groups specifies all letters between A and Z. Note that these are case sensitive so we are only matching capital letters.

Multiple ranges can also be specified:

```
/[A-Za-z]/
```

`Though` **`this`** **`be`** **`madness`**`,` **`yet`** **`there`** **`is`** **`method`** **`in`** `'`**`t`**`.`

These ranges are based on ASCII values, which are typically in a logical order (A through Z are in order, sequentially) by logical groups of characters. Care should be taken when trying to mix and match logical groups of characters. Specifying [A-z] will get you [A-Z] and [a-z], but it will also match a whole host of other characters that lie between in ASCII values. We'll cover some easier ways to specify logical groups of characters later on.

5.1.4 QUANTIFIERS

In the following examples we will use the following input text, a quote from Shakespeare's *Hamlet*:

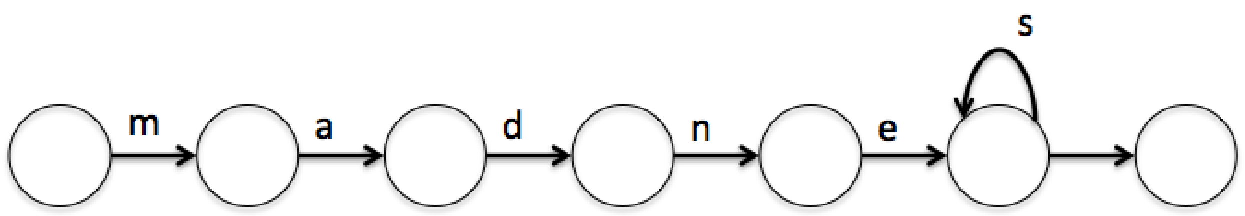
```
Though this be madness, yet there is method in it.
```

Regular expressions also have a set of special metacharacters that you can use to place *quantifiers* on parts of the expression. We will quickly outline several frequently used quantifiers.

We can use `*` to specify that the preceding part occurs *zero or more* times:

```
/ madnes*,/
```

To help explain this expression:



Here this expression operates as normal until it matches the 'e'. After matching the 'e', the machine is presented with two options: it can proceed to the end, or can match an 's', and end back in the same place. It can keep matching the 's' in a loop indefinitely until it is no longer able to match one. By default, these quantifiers are greedy - that is, it will attempt to "consume" as many characters as possible before moving on. So if an 's' is available, it will not be lazy and decide to skip to end of the expression. It will match as many 's' as possible before moving on. This shows us that the `*` operator lets us continue down the expression, with or without that character.

Though this be **madness,** yet there is method in it.

Here we request the "s" appears zero or more times. Here it appears two times and the pattern matches. We could specify something that doesn't appear at all (but perhaps does in another input):

```
/ madnessq*,/
```

Though this be **madness,** yet there is method in it.

Here we request the "q" appears zero or more times. The "q" doesn't appear at all, but this is still a valid match. These quantifiers can be applied to character groupings as well:

```
/ madness[,.  ]*/
```

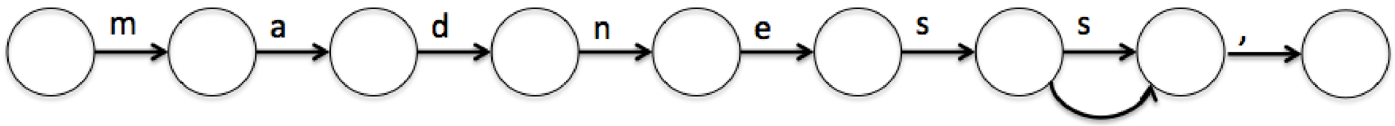
Though this be **madness,** yet there is method in it.

Here we match a comma *and* a space out of the character group. By itself, the character group will only match a single character one time, however, with quantifiers we can apply the character group multiple times and match several characters.

There are several other quantifiers that can be used. The `?` can be used to request the preceding part occurs *zero or one* times. It makes the preceding part optional:

`/ madness?,/`

To help explain this expression:



Here we see the machine operates like normal until the final ‘s’. Here it may match the ‘s’, or it may skip over it and proceed to searching for a comma. Again, the default behavior is to be greedy, so it will match the second ‘s’ (if one is present) and will not decide to be lazy and skip over it, then fail to find a comma (because the second ‘s’ it skipped is not a comma). This results in the match:

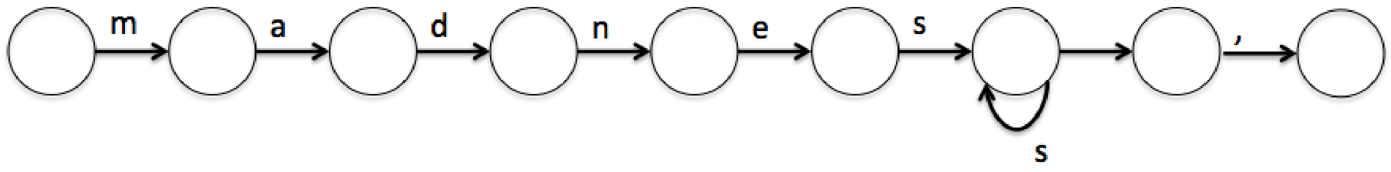
`Though this be` **madness,** `yet there is method in it.`

Here we make the final “s” optional, which of course matches.

We can use the + to request the preceding part appear *one or more* times. The preceding part will appear at least once:

`/ madnes+,/`

To help explain this expression:



This looks very similar to the * diagram, however, we see that we must proceed past at least one ‘s’. Then the machine can either loop over multiple ‘s’ or skip on to the next. This results in the match:

`Though this be` **madness,** `yet there is method in it.`

Finally, we can specify precisely how many of a thing we want.

`/s{2}/`

`Though this be madne`**ss**`, yet there is method in it.`

This matches occurrences of exactly two sequential “s” characters.

We can also specify a range:

`/s{1,2}/`

`Though thi`**s**` be madne`**ss**`, yet there i`**s**` method in it.`

This matches either one or two instances of the “s” character in a sequence.

The ranges can also be open-ended such as `{1,}` which is equivalent to the + (one or more) or `{,5}` which means up to 5, but not more.

5.1.5 CHARACTER CLASSES

In the following examples we will use the following input text, a quote from Shakespeare's *Hamlet*:

`Though this be madness, yet there is method in it.`

Most regular expression implementations have some sort of construct for defining a class of characters. These classes take the place of complex character groups, defining commonly used sets of characters. These include logical groupings such as "numbers", "word characters", or "whitespace".

If we wanted to match every word in our sample input, we could make a character group and list capital letters, lower-case letters, hyphens and other such characters that may be part of a word. However, this is somewhat cumbersome and prone to accidentally leaving out some range of characters. There is an easier way - character classes.

Some implementations of of regular expressions employ different syntax for character classes. The two main types are PCRE (Perl Compatible Regular Expressions) and GNU/POSIX. Most programming languages employ the PCRE types (or something similar), while GNU tools (which are covered in following section) use their own syntax. For these examples, we'll use PCRE, but the basic idea remains the same whether you are using PCRE or GNU tools. A conversion and reference table is provided at the end of this section.

A "word" character class is frequently used:

`/\w+/`

Though **this** **be** **madness**, **yet** **there** **is** **method** **in** **it**.

This is equivalent to:

`/[A-Za-z0-9_]+/`

Another commonly used class is the "space" class:

`/\s+/`

`Though_this_be_madness,_yet_there_is_method_in_it.`

This is equivalent to:

`/[ \t\r\n\f]+/`

Here we match all spaces, but we also match other space-like characters, like tabs, new lines and carriage returns.

We could combine these and try to match two words that are followed by a comma:

`/\w+\s\w+,/`

Though this **be madness,** yet there is method in it.

This looks for multiple word characters, followed by a space character, then followed by another series of word characters, and finally ended by a comma.

We can also combine character classes into character groups:

`/[\w,.']+/`

Though **this** **be** **madness,** **yet** **there** **is** **method** **in** **it.**

A special type of character class is the “dot”. This special character is a placeholder for **any** character. This special character is quite powerful and should be used with caution.

Let’s say we wanted to look for “madness” and then grab the rest of the line (because we like madness!):

`/madness.*/`

Though this be **madness, yet there is method in it.**

As soon as “madness” is matched it then matches everything else, regardless!

Here is a reference of commonly used character classes:

PCRE	POSIX	Description
\w	[:alnum:]	Matches alphanumeric characters
\s	[:space:]	Matches space characters
\d	[:digit:]	Matches digit characters

5.1.6 NEGATION AND ESCAPING

In the following examples we will use the following input text, a quote from Shakespeare’s *Hamlet*:

`Though this be madness, yet there is method in it.`

Sometimes we may want to match characters that are **not** certain characters. We use the caret, ^, at the beginning of a character to class to specify we want everything **NOT** in the following character class. This behavior only applies when the caret is used inside a character class.

Let’s try to find:

`/[^mad]/`

Though this be mad**ness, yet there is** m**etho**d **in it.**

This matches every single character except, ‘m’, ‘a’, and ‘d’.

Sometimes we want a regular expression to include one of the many special regular expression characters we have covered. If we type the special character directly into a regular expression, the regex parser will interpret that character as a special character and it’s special meaning. If we want to literally search for that character, then we need to *escape* the character. We just simply need to add a backslash in front of the character.

For example, if we wanted to search for a period, we can’t just type in a dot or it will match everything! But we can do:

`/\./`

Though this be madness, yet there is method in it**.**

The backslash instructs the interpreter to treat the following character it literally, in this case, it looks for a literal period.

5.1.7 ANCHORS

In the following examples we will use the following input text, a quote from Shakespeare’s *Hamlet*:

`Though this be madness, yet there is method in it.`

Often times in regular expressions, we want to look for a pattern in a specific part of an input. There are special characters that we can use to instruct the expression to look for a pattern in a specific place. The two most common ones are the beginning of line (ˆ) and end of line ($) anchors.

Let's look for the first word of the line:

`/^\w+/`

Though this be madness, yet there is method in it.

This regular expression will anchor at the beginning of the line and then begin it's search for word characters. This is not to be confused with the negation character used inside character classes. Similarly for the end of line anchor:

`/[\w]+[\s,\.!\?]+$/`

Though this be madness, yet there is method in **it.**

Here we look for a string of word characters or punctuation (don't forget those escapes!) that are immediately preceded by the end of line. This will not match any earlier words, because they are not at the end of line.

Going off the last example, and one our very first examples - we see a couple problems with including a list of punctuation and spaces as a way to look for the end of a word. First, did we include all possible punctuation? What about colons? Or semi-colons? The list becomes cumbersome. Secondly, this match then includes the punctuation and the spaces. What if we are just interested in the word itself and not the junk around it? A third type of anchor comes in very handy in this situation.

We can anchor our search around word boundaries with the **\b** anchor. In this way we can match entire words at once without including the extra junk in the returned result:

`/\b\w+\b/`

Though **this** **be** **madness**, **yet** **there** **is** **method** **in** '**t**.

Here we match whole words, not including the non-word characters in between. Here "non-word characters" is equivalent to [ˆ\w], without including those characters in our match!

5.1.8 GROUPING AND CAPTURING

In the following examples we will use the following input text, a quote from Shakespeare's *Hamlet*:

`Though this be madness, yet there is method in it.`

In regular expressions we can build more complex expressions through *grouping*. In this section we'll introduce grouping and the logical or operator.

We can create grouping simply by using parenthesis, and the logical or by a pipe `|`. Say we want to find one of several two-letter words:

`/\b(is|in|it|be)\b/`

Though this **be** madness, yet there **is** method **in** **it**.

Here we make a group, and say we want the word is, in, it, or be. Remember to anchor this match with word boundaries or we will have inadvertant partial matches inside other words.

Let's look at more complex example. Let's say we want to find occurrences of two sequential "i" words, such as "in it" in the test phrase. Let's walk through building it - let's first identify all i words:

```
/i\w+/
```

OK, but not quite. This matches the "is" in "this". Let's add word boundary:

```
/\bi\w+/
```

This finds all of our i words, but we are looking for occurrences of two of them in a row. So, can we say we want two of these like this?

```
/(\bi\w+){2}/
```

Well, not quite. We are forgetting about the space character in between the two. So we could try specifying our word must have a space at the end of it.

```
/(\bi\w+\s){2}/
```

Still, not quite. In our test input, the second word is followed by a period, not a space. Let's make the space optional instead, and not forget our word boundaries!

```
/\b(i\w+\b\s*){2}/
```

`Though this be madness, yet there is method` **`in it`**`.`

Here we moved the first word boundary anchor out of the grouping. We only want to define our word as having a single trailing word boundary. We only need to specify the first boundary anchor once (but still need it, or our two-word match might end up starting in the middle of a word).

Often times, when we are creating regular expressions we only want to extract part of the match. That is, we need to specify extra characters to drill down our search, but only want to extra part of the match. Perhaps we want to find and replace a word, or we want to find all words matching some criteria.

In most regular expression implementations what will be returned to you is the full match, along with an array containing any sub-matches. Each grouping creates a sub-match, and the **last** string to match that group will be returned in that sub-match.

What does this all mean? Let's take an earlier example and modify a bit to say that we want to find all the words followed by punctuation:

```
/\w+[,\.!\?]+/
```

`Though this be` **`madness,`** `yet there is method in` **`it.`**

This is not very useful if we want to extract just the words. Yes, it tells us that the input does contain words followed by punctuation, but that's it. What would be returned to you in a script or program would be "madness," and "it.". Let's create a group to get rid of the pesky punctuation (can't live with t, can't live without it!):

```
/(\w+)[,\.!\?]+/
```

The full match is exactly the same, however, now your regular expression implementation should also return to you the contents of what matched inside the parenthesis, \w+, the word! So you would have two matches, each with a single sub-match:

```
madness
```

```
it
```

We'll talk more about how to access these sub-matches in later section when we cover using these regular expressions in actual tools.

One last thing to note is that only the **last** match in a sub-group will be returned to you. So if you tried:

```
/(\w)+[,\.!\?]+/
```

Your full match would be identical, however, you'd only get the last character of each word in your sub-match! Matching a group of word characters vs groups of a word character. A very subtle difference!

5.1.9 PUTTING IT ALL TOGETHER

Regular expressions have far too many applications to list them all here, however, we will cover a few topics and uses for them.

If a bioinformatician looks for a particular motif in large number of sequences, they can create a regular expression and search all the sequences having that motif relatively easily. Here are some examples related to nucleotide/protein sequences:

Patterns	Matches
`^ATG`	Find a pattern starting with ATG
`TAG$`	Find a pattern ending with TAG
`^A[TGC]G`	Find patterns staring with either ATG, AGG or ACG
`TA[GA]$`	Find patterns ending with either TAG or TAA
`^A[TGC]G*TGAACT*TA[GA]$`	Find gene containing a specific motif
`[YXN][MPR]_[0-9]{4,9}`	Find patterns matching any NCBI RefSeq (eg XM_012345)
`(NP|XP)_[0-9]{4,9}`	Find patterns matching NCBI RefSeq proteins

5.1.10 REFERENCES

Here is a quick reference for your convenience.

Expression	Function
`.`	matches any single character
`$`	matches the end of a line
`^`	matches the beginning of a line
`*`	matches zero or more preceding item
`+`	matches one or more preceding item
`?`	match zero or one occurrence of the preceding item
`\`	escaping character, treat the next character followed by this as an ordinary character.
`[]`	matches one of the characters between the brackets
`[range]`	match any character in the range (eg., A-Z)
`[^range]`	match any character except those in the range
`{N}`	match N occurrences of the item preceding (sometimes simply +N) where N is a number.
`{N1,N2}`	match at least N1 occurrences of the item preceding but not more than N2
`|`	match two conditions together, (THIS|THAT) matches both THIS or THAT in the text

`()`	grouping - (A|Z){3} - matches A or Z three times sequentially, i.e. either AAA or ZZZ

Here a few web sites you may find helpful:

http://www.regular-expressions.info/ - Great resource for reference and everything you need to know about regular expressions.

https://regex101.com/ - Great tool for testing your regular expressions in various different environments.

5.2 TEXT SCULPTING

This section will deal with more complex commands and their useful options along with using multiple commands at a time. Make sure you understand all the commands from the previous sections.

In this section we will talk about commands and utilities that are sometimes referred to as “filters”, because they allow extraction or selective handling of certain lines or sections of the data. There are many of such “filters” (`grep`, `awk`, `sed`, `tr`, `cut`, `paste`, `join`, `sort`, `uniq` to name just a few most common ones). We will only cover basic use of some of them, and leave the rest for you to explore as you master your *Unix-fu*. Google and `man commandname` are your friends.

5.2.1 grep

The `grep` is one of the most commonly used commands in UNIX and it is commonly used to filter a file/input, line by line, against a pattern (e.g., to print each line of a file which contains a match for pattern).

General syntax:

```
grep [OPTIONS] PATTERN FILENAME
```

(if given no file, it reads from the standard input)

Like any other command there are various options available for this command. Most useful options include:

`-v`	inverts the match or finds lines NOT containing the pattern
`--color`	colors the matched text for easy visualization
`-F`	interprets the pattern as a literal string, not regexp (a.k.a. fgrep)
`-E`	interprets the pattern as an extended regular expression (more powerful, slightly different syntax, a.k.a. egrep)
`-H, -h`	print, don’t print the matched filename
`-i`	ignore case for the pattern matching
`-l`	lists the file names containing the pattern (instead of match)
`-n`	prints the line number containing the pattern (instead of match)
`-c`	counts the number of matches for a pattern
`-w`	forces the pattern to match an entire word
`-x`	forces patterns to match the whole line

Some typical scenarios to use `grep`:

- Extracting specific line(s) from the simulation output
- Stripping header/footer/comments lines from an input file
- Selecting files of interest (with `-l`)
- Counting number of occurrences of a string or pattern in a file (with `-c`)

Task: in file Hamlet.txt acts and scenes are denoted by 'ACT' and 'Scene', respectively, at the beginning of a line. What are the scene titles and how many acts are there in the tragedy?

```
grep '^Scene' Hamlet.txt
```

(also try the above with `--color`)

```
grep -c '^ACT' Hamlet.txt
```

A handy trick for bioinformaticians: how many sequences are in a FASTA-formatted file? By definition, each sequence record in a FASTA file has one line of description that always starts with >, followed by multiple lines of sequence itself. Each sequence record ends when the next line starting with > appears:

So counting number of sequences in FASTA-file is as easy as

```
grep -c '^>' FILENAME
```

A structural biologist may separate comments from the rest of PDB atomic coordinates file like this:

```
grep '^REMARK' myprotein.pdb > myprotein_comments.txt
```

```
grep -v '^REMARK' myprotein.pdb > myprotein_no_comments.pdb
```

Or even filter out everything but atomic coordinates:

```
grep '^\(ATOM\|HETATM\)' myprotein.pdb > myprotein_coord.pdb
```

(you have a sample PDB file 1UBQ.pdb in the "protein" subdirectory if you want to experiment).

Task: which tragedy has Iago as a character?

```
grep -i -w -l Iago *.txt
```

or

```
grep -i -w -c Iago *.txt
```

5.2.2 awk

Unlike other UNIX commands, `awk` is a structured language by itself. `awk` stands for the names of its authors Aho, Weinberger, and Kernighan. Many scientific programs generate rows and columns of information. `awk` is an excellent tool for processing these rows and columns, and it is easier than most conventional programming languages.

The syntax for awk is:

```
awk 'PATTERN {ACTION}' FILENAME
```

`awk` then works by reading the input file one line at a time, matching the given `PATTERN` and performing the corresponding `ACTION` for the matches. If there is no `PATTERN`, then the `ACTION` will be performed on each line. If there is no `ACTION` then the default `ACTION` (printing all lines) on the matching `PATTERN` will be performed (empty braces {} without any `ACTION` turns off default printing).

A simplest example would be:

`awk '{print;}' FILENAME` *try* `awk '{print;}' Hamlet.txt`

Here, since there is no `PATTERN`, the print `ACTION` will be performed on each line (equivalent to `cat FILENAME`). Now add a `PATTERN`:

```
awk '/Hamlet/ {print;}' Hamlet.txt
```

(what command is this equivalent to?) ______________________________

`awk` allows performing arbitrary actions with all or parts of input lines, using variables, conditionals, loops and functions (subroutines). Some built-in variables of `awk` include:

`FS`	Field Separator (default ANY WHITESPACE)
`OFS`	Output Field Separator (default SPACE)
`NF`	Number of Fields in the current input record (line)
`NR`	Number of Records (lines) in the input
`RS`	Record Separator (default NEWLINE)
`ORS`	Output Record Separator (default NEWLINE)
`FNR`	File line number
`$N`	Nth field of the line where N can be any number (eg. \$0 = entire line, \$1 = First field, \$2 = second field and so on).
`IGNORECASE`	If not zero, regexp matching is case insensitive (default=0)

`awk` accepts all standard patterns (regular expression and expression) plus some special patterns:

`BEGIN`	Special PATTERN that is executed before the INPUT is read
`END`	Special PATTERN that is executed after the INPUT is read
	empty (nonexistent) PATTERN matches every input record

Some simple examples using `awk` (you can try these commands with **Hamlet.txt** or better yet, with **protein/1wt.rmsd.txt** as `FILE`)

`awk 'NF' FILE`	Delete all blank lines
`awk 'NF>0' FILE`	Delete all blank lines
`awk 'NF>4' FILE`	Print all lines with more than 4 fields
`awk '$NF>4' FILE`	Print all lines with value of the last field >4 **(note the difference with above)**
`awk 'END {print $NF}' FILE`	Print value of the last field of last line
`awk 'NR==25,NR==100' FILE`	Print lines between 25 and 100
`awk 'NR==50' FILE`	Print 50^{th} line of input
`awk 'NR<26' FILE`	Print first 25 line
`awk 'NR>25' FILE`	Print file after 25^{th} line
`awk 'END {print}' FILE`	Print the last line of the file
`awk '{print NF ":" $0}' FILE`	Print number of fields in front of every line
`awk '{print FNR ":" $0}' FILE`	Print line number in front of every line
`awk '$5=="abc123"' FILE`	Print lines which have 'abc123' in 5^{th} field
`awk 'BEGIN {ORS=="\n\n"} {print}' FILE`	Double space the file
`awk '{print $1,$2}' FILE`	Print only 1^{st} and 2^{nd} field
`awk '{print $2,$1}' FILE`	Print only 2^{nd} and 1^{st} field (swapping columns)
`awk '{$2=""; print}' FILE`	Print the file without 2^{nd} column
`awk '/REGEX/' FILE`	Print all lines having REGEX
`awk '!/REGEX/' FILE`	Print all lines not having REGEX
`awk '/AAA|BBB|CCC/' FILE`	Print all lines having either AAA, BBB or CCC
`awk 'length>50' FILE`	Print lines having more than 50 characters
`awk '/POINTA/,/POINTB/' FILE`	Print section of file between POINTA and POINTB

Try to understand the following command lines (and record your results, where applicable):

```
awk 'END {print $NF}' Hamlet.txt
awk 'NR==30,NR==35' Hamlet.txt
awk 'NR==25' Hamlet.txt
awk 'NR<25' Hamlet.txt
awk 'END {print NR}' Hamlet.txt
awk '{print NF ":" $0}' Hamlet.txt > with_fields.txt
awk '{print NR ":" $0}' Hamlet.txt > with_Line_num.txt
awk '{print $1, $3}' Hamlet.txt
awk '{print $1"\t"$3"\t"$2}' protein/1wt.rmsd.txt
awk '{print $1,$2,$(NF-4),$(NF-3)}'protein/1wt.rmsd.txt
awk '/To be.*not to be/' Hamlet.txt
awk -v IGNORECASE=1 '/(to be).*(to be)/' Hamlet.txt
```

One of the awk's conveniences:

By default it splits lines into fields similarly as a human would do (i.e., on whitespace). In this way it can be used for processing of space or tab-delimited data just as well as for processing of free-flowing text. Alternative separators can also be specified.

`man awk` is a good reference material (although not as good of a tutorial). A lot of tutorial information can be found at *http://www.grymoire.com/Unix/Awk.html*. There is a good collection of useful awk one-liners at *http://www.pement.org/awk/awk1line.txt*.

5.2.3 sed

The `sed` command is a command that reads one or more text files line by line, makes changes according to the editing script and writes the results to standard output. The editing script can be defined to selectively add/delete/modify fragments of text as needed.

The most common use of `sed` for script editing is to substitute text that matches a pattern. The simple syntax for using sed in this fashion is as follows:

```
sed [OPTIONS] 's/REGEXP/REPLACEMENT/FLAGS' FILENAME
```

or

```
sed [OPTIONS] 'ANCHOR s/REGEXP/REPLACEMENT/FLAGS' FILENAME
```

Here, / is the delimiter. You can also use the _ (underscore), | (pipe), or : (colon) character as the delimiter as well as anything that's not part of `REGEXP` or `REPLACEMENT`.

`s` is the substitute operation that specifies the action to be performed.

`REGEXP` and `REPLACEMENT` specify search term and the substitution term respectively for the operation that is being performed.

`FLAGS` are additional parameters that control the operation. Some commo `FLAGS` include:

`g`	replace all instances of REGEXP with REPLACEMENT (globally)
`n`	(n=a number) replace nth instance of the REGEXP with REPLACEMENT
`i`	ignores case for matching REGEXP
`w filename`	if substitution was made, write out the result to the given file named filename
`p`	if substitution was made, then prints the new pattern space

`ANCHOR` is an optional condition that allows you to limit the scope of the action (e.g., only substitute on lines containing a specific label pattern, having specific line numbers, etc.) For brevity we only discuss the `sed` command with respect to its search and replace function. To do other things please refer to the man sed or the reference provided here *http://www.grymoire.com/Unix/Sed.html#uh-47*.

You may also find a useful collection of `sed` one-liners at *http://sed.sourceforge.net/sed1line.txt*.

```
# replace first Bob on any line
$ sed 's/Bob/Alice/' FILENAME

# replace 3rd Bob on any line
$sed 's/Bob/Alice/3' FILENAME

# replace all Bobs on all lines
sed 's/Bob/Alice/g' FILENAME

# replace all Bobs and ignore case
sed 's/Bob/Alice/gi' FILENAME

# replace with nothing (erase Bob)
sed 's/Bob//g' FILENAME

# pain of escaping!
sed 's/\/home\/Bob/\/home\/Alice/' FILENAME

# change delimiter!
sed 's@/home/Bob@/home/Alice@g' FILENAME
```

Some example of anchoring:

```
# only on lines with Mary
sed '/Mary/ s/Bob/Alice/' FILENAME

# only in the first two chapters
sed '/Chapter 1/, /Chapter 3/ s/Bob/Alice/' FILENAME

# every 10th line starting at 5th
sed '5~10 s/Bob/Alice/' FILENAME
```

Other useful `sed` operations besides 's':

```
# transliterate letter-by-letter (tr)
sed 'y/source/dest/' FILENAME

# delete lines with Bob (grep -v!)
sed '/Bob/d' FILENAME

# select lines with Bob (grep!)
sed -n '/Bob/p' FILENAME
```

`OPTIONS` include:

`-n`	do not print lines after processing (only print if an explicit 'p' flag or command was given)
`-i .bak`	inline processing (saves result under original file name, makes a copy of the original with `.bak` extension) [be careful here!]

5.3 ADVANCED COMMANDS

5.3.1 COUNTING LINES WITH 'wc'

The `wc` (word count) command simply counts the number of lines, words, and characters it sees in its standard input, and sends those totals to its standard output.

General syntax:

`wc [OPTIONS] FILENAME`

`OPTIONS` include:

`-l`	count lines only
`-w`	count words only
`-c`	count characters only

Let's look at the content of the file **wcdemo.txt**:

```
$ cat Shakespeare/wcdemo.txt
This is just a very simple
text file that we'll use
to demonstrate wc
```

Now what do we see if we run `wc wcdemo.txt`:

```
$ wc Shakespeare/wcdemo.txt
3 14 70 wcdemo.txt
```

This tells us that the file **wcdemo** has 3 lines, 14 words, and 70 characters. The word "we'll" in the file looks like one word to Unix text processing commands.

5.3.2 EXTRACTING SECTIONS FROM EACH LINE OF INPUT WITH 'cut'

The cut command is a simple command used to remove or "cut out" sections of each line of a file or files. General syntax:

`cut [OPTIONS] FILENAME`

`OPTIONS` include:

`-d`	use DELIM instead of TAB for field delimiter
`-f`	select only these fields; also print any line that contains no delimiter character

Try the following:

```
$ head protein/1UBQ.pdb
HEADER CHROMOSOMAL PROTEIN 02-JAN-87 1UBQ
TITLE STRUCTURE OF UBIQUITIN REFINED AT 1.8 ANGSTROMS RESOLUTION
COMPND MOL_ID: 1;
COMPND 2 MOLECULE: UBIQUITIN;
COMPND 3 CHAIN: A;
COMPND 4 ENGINEERED: YES
SOURCE MOL_ID: 1;
SOURCE 2 ORGANISM_SCIENTIFIC: HOMO SAPIENS;
SOURCE 3 ORGANISM_COMMON: HUMAN;
SOURCE 4 ORGANISM_TAXID: 9606

$ cut -f1 -d' ' protein/1UBQ.pdb | head
HEADER
TITLE
COMPND
COMPND
COMPND
COMPND
SOURCE
SOURCE
SOURCE
SOURCE
```

5.3.3 SORTING FILES WITH '`sort`'

The `sort` is a simple and very useful command which will order the lines in a text file so that they are sorted, numerically and alphabetically. By default, the rules for sorting are:

- Lines starting with a number will appear before lines starting with a letter.
- Lines starting with a letter that appears earlier in the alphabet will appear before lines starting with a letter that appears later in the alphabet.
- Lines starting with a lowercase letter will appear before lines starting with the same letter in uppercase.

General syntax:

`sort [OPTIONS] FILENAME`

`OPTIONS` include:

`-d`	consider only blanks and alphanumeric characters
`-r`	reverse the result of comparisons

```
# sort a specific field or key (the "k" in "-k" stands for "key") in the file:
$ sort -k[number] FILENAME

# sort the second field, and ignore the first:
$ sort -k2 FILENAME

# sort in reverse order:
$ sort -r FILENAME

# sort based on a field treated as a number, rather than text:
$ sort -n FILENAME

# sort and output only unique lines, i.e., remove duplicates from the sorted output:
$ sort -u FILENAME
```

Try the following exercise:

- Try the following: `sort` the **HamletWords.txt** file. This file contains a list of all words in Hamlet.txt (as a bonus, figure out how to create this list yourself!). Sort it with the `-u` option and pipe the output through `wc` to find out how many unique words there are in the play; `sort` it backwards. Find some other files to sort.
- What does this do?

```
$ cut -f1 -d' ' protein/1UBQ.pdb | sort -u | wc -l
```

5.3.4 SELECTING UNIQUE LINES IN A SORTED FILE WITH '`uniq`'

The `uniq` command simply takes a sorted file and outputs the unique lines in it. The input must be sorted – it can only find duplicate lines *if they are adjacent*. It is different from the sort `-u` option because it can also count the duplicates and tell you how many instances of a line there are with its `-c` option.

General syntax:

`uniq [OPTIONS] [INPUT] [OUTPUT]`

`OPTIONS` include:

`-c`	count how many times they occurred
`-d`	only print duplicated lines
`-u`	only print unique lines

Example:

`sort HamletWords.txt | uniq -c`

This will `sort` all the words in HamletWords.txt and feed them into `uniq -c` to count occurrences of each word.

Now `sort` by the word count (and consider the count as a numeric value – see what happens if you omit the `-n` option!):

```
sort HamletWords.txt | uniq -c | sort -n
```

Try it! See what the most common first word in a line of HamletWords.txt is.

5.3.5 ADVANCED PIPES AND REDIRECTS

In an earlier section we introduced the concepts of pipes and redirects. In this section we will take a look at some more advanced types of piping and redirecting.

First we must introduce a couple of new terms: **standard output (stdout)**, **standard error (stderr)**, and **standard input (stdin)**. These three are what are called **streams** that are created in the execution of every UNIX program.

These three are fairly self-explanatory. Standard out is where normal output from a program is sent, and by default is connected to your terminal screen. Standard error is typically where error messages from a program are sent, and is also by default connected to your terminal screen. Standard input is where input may come from. Not every program will connect to or read from (some may require an option to enable reading) the standard input stream, but if it does it will connect to your keyboard by default. It is possible to change where these different streams are connected to. This may be helpful if you have an input file you do not wish to type in by hand, or want to send output to a file but keep errors sent front and center to your screen. You may also want to send normal output to one file and errors to another making them easier to find later.

We have already covered standard input in the usage of the usage of < operator:

```
cat < FILE
```

This will instruct cat to take its input from `FILE` and read from the file, rather than take input from you on the keyboard. `cat` then takes `FILE` and prints it out to standard output, which is connected to your terminal screen. We can get fancy and reconnect standard output to something else with the > operator:

```
cat < FILE > OTHERFILE
```

Now let's imagine we are creating an automated script or a workflow that involves listing files, and we know that sometimes the files we are looking for are not present. We want to keep a log of when this happens, and what we were looking for:

```
$ ls -l somefile 2> error.log
$ cat error.log
ls: cannot access somefile: No such file or directory
```

Here we have sent error messages to the `error.log` file, and inspecting it we see the error from the listing.

We could redirect the standard output and error to two separate files as well:

```
$ ls -l * 2> error.log > out.log
$ cat error.log
ls: cannot open directory private: Permission denied
$ cat out.log
file1
file2
```

Oops, here we see we tried to do a listing in a directory we don't have permissions to look into. Here we see that logged the `error.log` file, along with all of our files in the `out.log` file.

Let's say we don't care to split the outputs into two different files. We could combine them into a single file:

```
$ ls -l * >& out.log
$ cat out.log
file1
file2
ls: cannot open directory private: Permission denied
```

As advertised, the error message shows up in the output file along with all the normal output. There may be cases where separate files is more useful, or cases where combining them is appropriate. You will have to choose what is best for you, both options are available to you.

Similar options are also available to you when piping output of commands from one into another. Normally, when you pipe output only the standard output is piped into the next command and errors are sent to your screen. Typically, this makes sense. You don't want any error messages to get eaten up by the next program – you'd probably only notice if the next program got confused with parsing the error text and threw its own errors (hopefully to your screen this time!). You want to see them so you can fix whatever the problem is. However, there are times where you want to send error messages into the next command as well.

We can accomplish this by instructing that standard error messages get sent into the same place as standard out:

```
ls -l * 2>&1 | less
```

When we do this we get any error messages from the listing also piped into our `less` and we can page through the error messages as the occurred in the listing as well.

We can also do the opposite:

```
ls -l * 1>&2 | less
```

Perhaps we have a bunch of errors and just want to page through them.

What if we do have a bunch of errors and just don't care? We can throw away output as well! Unix systems have a special place called `/dev/null` we can send junk we don't care about:

```
ls -l * 2> /dev/null
```

We can, of course, redirect any or all of the outputs here:

```
ls -l * >& /dev/null
```

This special path will happily take any output you give and throw it away silently.

5.3.6 LOOPS

Bash provides a mechanism for looping and flow control. We will discuss the various types of flow control in a later section about Bash programming, but for now we will briefly discuss loops in the context of command line interaction.

The for loop is the most simple type of loop, and we will discuss in this section. In the simplest form, a for loop takes a list of items to iterate through. This could be a list of files or a sequential list of numbers.

A basic example:

```
$ for i in "one" "two" "three"; do echo $i; done
one
two
three
```

Here we give a variable name, "i", which will contain the value for an iteration of the loop, and then "in" our list of items. The "do" and "done" essentially mark the start and end of the loop block. This example just prints the variables, but we could give it a list of files to operate on.

We can also work on a sequence of numbers, say we had some numbered inputs we wanted to work on:

```
$ for i in {1..5}; do cp "file-$i.txt" backup/; done
```

Here we make a copy of files 1, 2, 3, 4, and 5 in our backup directory.

5.3.7 COMMAND SUBSTITUTION

Most shells allow running a command to fill in the input for another command. These are called *subshells* or *command substitution.* In Bash, we simply do this with the ` (backtick) character. This is the one on the key to the left of the number 1 key, and not an apostrophe.

Building off the last section about loops, we could do something like:

```
$ for i in `ls`; do cp "$i" backup/; done
```

This is executed by Bash by first executing the command between the backticks: `ls`. The output of this command is then substituted into the command around it. So the previous command could then become after the subshell execution:

```
$ for i in "one" "two" "three"; do cp "$i" backup/; done
```

Subshells are useful in many more ways in Bash programming, but they are still helpful from time to time in day to day command line usage. We will cover them in more detail in the following section on Bash programming.

6 BASH PROGRAMMING

In this section, we'll take a closer look at what we can do with the UNIX bash scripts. We'll discuss various aspects including defining and using variables, branches and loops, input and output. We recommend that you read ***Bash Guide for Beginner***s for more details (http://www.tldp.org/LDP/Bash-Beginners-Guide/html/).

6.1 SHELL BASICS

If we recall from Section 4, The UNIX shell program interprets user commands, which are either directly entered by the user, or which can be read from a file called the ***shell script*** or ***shell program***. Here let's revisit some of the important concepts and syntax of shell scripts.

Program Interpreting the Script: The first line of a shell script should define the program that will interpret the script. In our examples, this will be `bash`, the shell program. The way to specify this program is by putting it as the first line in the bash script:

```
#!/bin/bash
```

When your execute the bash script, bash will look at the first line and determine which program it should call that can understand and interpret the commands that follow. In this case, the program is `/bin/bash`.

End of a Command: A semicolon `;` or a newline indicate the end of a command. For example, the following script will execute `ls` and `pwd` one after another since the `;` indicates that `ls` is one command and `pwd` is another command although they are written in the same line. A semicolon right before a newline is superfluous, but sometimes people put them there for clarity.

```
#!/bin/bash

ls; pwd;
```

Making the Script Executable: To run a shell script, the script itself needs to be executable (i.e., with permission `x` set for the user). To make a script executable, you can change its permission using the `chmod` command:

`chmod +x FILENAME`

Executing a Shell Script: Once the script is executable, to run the shell script, simple type the path to the script in your terminal as following.

```
$ /path/to/script.sh
```

This example shows how to execute the script `script.sh` that can be found in the directory `/path/to/`. If the script is under your current working directory, simply type `./script.sh` to run. Note that you need to type the entire path of the executable `/path/to/script.sh` or `./script.sh` so that shell knows exactly where to find the `script.sh`. There are ways to tell the shell where to find the executable `script.sh` without typing the entire path as discussed below.

Putting it into PATH: On UNIX/Linux systems, when you type in the name of a command without specifying the path that leads to the command (in the previous example, the name of the command is `script.sh`), the shell searches a list of directories where executable files are kept to find the command. It also **only** searches the directories in that list. The list of the directories is called your `PATH` environment variable. If it does not find the program or command after searching, it will output the `command not found` error message. To avoid typing the full path to an executable every time, we can put the directory that contains the shell script in your `PATH` environment variable.

To view your `PATH`, type the following command:

```
$ echo $PATH
 /apps/rhel6/r/3.2.2/bin:/apps/rhel6/gcc/4.9.2/bin:/usr/lib64/qt-3.3/bin:/opt/moab/bin:/
    usr/local/bin:/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/sbin:/opt/hpss/bin:/opt/hsi/
    bin:/opt/ibutils/bin:/usr/pbs/bin:/opt/moab/bin:/opt/thinlinc/bin
```

This will output the list of directories shell searches when you type a command. This is a colon separated list of directories that will be searched if the full path of a command (or an executable, or a shell script) is not given.

To add the directory that contains the shell script (`/path/to/`) to your `PATH`, type the following command:

```
$ export PATH=$PATH:/path/to
$ echo $PATH
 /apps/rhel6/r/3.2.2/bin:/apps/rhel6/gcc/4.9.2/bin:/usr/lib64/qt-3.3/bin:/opt/moab/bin:/
    usr/local/bin:/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/sbin:/opt/hpss/bin:/opt/hsi/
    bin:/opt/ibutils/bin:/usr/pbs/bin:/opt/moab/bin:/opt/thinlinc/bin:/path/to
```

Note that the directory `/path/to` is added to the `PATH` environment variable, and now shell searches in this directory for executables.

In the following sections, when we discuss various shell concepts, commands, and operations, we mostly demonstrate them by directly entering the shell commands in the Bash shell. To use the set of commands collectively for a task, you can put the shell commands into a shell script or shell program, then execute the program.

6.2 SHELL TYPES

Just like people know different languages and dialects, your UNIX system will usually offer a variety of shell types:

- ***bash*** or ***Bourne Again shell***: the standard GNU shell, intuitive and flexible. Probably most advisable for beginning users while being at the same time a powerful tool for the advanced and professional user. On Linux, Bash is the standard shell for common users. This shell is a so-called superset of the Bourne shell, a set of add-ons and plug-ins. This means that the Bourne Again shell is compatible with the Bourne shell: commands that work in ***sh***, also work in ***bash***. However, the reverse is not always the case. In this workshop, all the examples use Bash.
- ***csh*** or ***C shell***: the syntax of this shell resembles that of the C programming language. Sometimes asked for by programmers.
- ***tcsh*** or ***TENEX C shell***: a superset of the common C shell, enhancing user-friendliness and speed. That is why some also call it the ***Turbo C shell***.
- ***sh*** or ***Bourne Shell***: the original shell still used on UNIX systems and in UNIX-related environments. This is the basic shell, a small program with few features. While this is not the standard shell, it is still available on every Linux system for compatibility with UNIX programs.

The file **/etc/shells** gives an overview of known shells on a Linux system:

```
 $ cat /etc/shells
/bin/sh
/bin/bash
/bin/ksh
/bin/tcsh
/bin/csh
/bin/zsh
/usr/local/bin/bash
/usr/local/bin/tcsh
/opt/acmaint/etc/message
/opt/acmaint/etc/nologin
/opt/acmaint/etc/disable
/opt/commsh/bin/commsh
```

To find out what is the current shell that you are using, use the following command.

```
$ echo $0
 -bash
```

To switch from one shell to another, just enter the name of the new shell in the active terminal. The system finds the directory where the name occurs using the `PATH` settings, and since a shell is an executable file (program), the current shell activates it and it gets executed. A new prompt is usually shown, because each shell has its typical appearance. The following example switches from the Bash shell to the tcsh.

```
$ tcsh
$ echo $0
 tcsh
```

6.3 VARIABLES

6.3.1 VARIABLE TYPES

Unlike many other programming languages, Bash does not segregate its variables by "type." Essentially, Bash variables are character strings, but depending on context, Bash permits arithmetic operations and comparisons on variables. The determining factor is whether the value of a variable contains only digits.

In the respect of the variable content, variables come in 4 types:

- String variables
- Integer variables
- Constant variables
- Array variables

To lighten the burden of keeping track of variable types in a script, Bash does permit declaring variables. We'll discuss this more in detail in the following sections.

6.3.2 CREATING VARIABLES

Variables are case sensitive. You are free to use the names you want or to mix cases. Variables can also contain digits, but a name starting with a digit is not allowed.

To create a variable in a shell, use:

```
$ VARNAME="value"
```

This creates a variable named `VARNAME`, and assigns the string `value` as the content of `VARNAME`. The syntax to dereference a variable is to place a dollar sign (`$`) in front of the variable name. The following example displays the variable content to the standard output.

```
$ VARNAME="value"
$ echo $VARNAME
 value
```

Note that there is no spaces around the equal sign. It is a good habit to quote content strings when assigning values to variables: this will reduce the chance that you make errors. We will discuss more details on quoting characters in Section 6.5.

Bash has reserved words that are not allowed to be used as variable names. For example, `if`, `else`, `for` are reserved words in Bash that control branches and loops. Different shells may have slightly different sets of reserved keywords, please refer to the documentation or manual of the shell that you use for a precise list. In addition to shell reserved words, shells have a list of environment variables. For example, `$HOME`, `$SHELL`, `$PATH`. One should avoid using them as variable names unless it is your intention to manipulate those system wide environment variables. To view the list of environment variable in your current shell, use the `printenv` command.

6.3.3 EXPORTING VARIABLES

In the example above, `VARNAME` is only available to the current shell. It is a local variable: child processes of the current shell will not be aware of this variable. In order to pass variables to a subshell, we need to export them using the `export` built-in command. Variables that are exported are referred to as environment variables. Setting and exporting is usually done in one step:

```
$ export VARNAME="value"
```

A subshell can change variables it inherited from the parent, but the changes made by the child don't affect the parent.

6.3.4 SPECIAL VARIABLES

The shell treats several variables specially. These variables may only be referenced; assignment to them is not allowed.

Table 5: Special Bash Variables

Character	Definition
`$*`	Expands to the positional parameters, starting from one. When the expansion occurs within double quotes, it expands to a single word with the value of each parameter separated by the first character of the `IFS` special variable.
`$@`	Expands to the positional parameters, starting from one. When the expansion occurs within double quotes, each parameter expands to a separate word.
`$#`	Expands to the number of positional parameters in decimal.
`$?`	Expands to the exit status of the most recently executed foreground pipeline.
`$-`	A hyphen expands to the current option flags as specified upon invocation, by the set built-in command, or those set by the shell itself (such as the `-i`).
`$$`	Expands to the process ID of the shell.
`$!`	Expands to the process ID of the most recently executed background (asynchronous) command.
`$0`	Expands to the name of the shell or shell script.
`$_`	The underscore variable is set at shell startup and contains the absolute file name of the shell or script being executed as passed in the argument list. Subsequently, it expands to the last argument to the previous command, after expansion. It is also set to the full pathname of each command executed and placed in the environment exported to that command. When checking mail, this parameter holds the name of the mail file.

The `$*` positional parameters are the words following the name of a shell script. They are put into the variables `$1`, `$2`, `$3` and so on. The variable `$0` stores the name of the executable, in our examples, the name of the executable is the name of the shell script. As long as needed, variables are added to an internal array. `$#` holds the total number of parameters, as is demonstrated with this simple script. This script (`positional.sh`) is also in the directory `scripts`.

```
#!/bin/bash

# positional.sh
# This script reads 3 positional parameters and prints them out.

POSPAR1="$1"
POSPAR2="$2"
POSPAR3="$3"

echo "$1␣is␣the␣first␣positional␣parameter,␣\$1."
echo "$2␣is␣the␣second␣positional␣parameter,␣\$2."
echo "$3␣is␣the␣third␣positional␣parameter,␣\$3."
echo
echo "The␣total␣number␣of␣positional␣parameters␣is␣$#."
```

6.3.5 CONSTATN VARIABLES

In Bash, constants are created by making a variable ***read-only***. The `readonly` built-in marks each specified variable as unchangeable. The syntax is:

```
readonly [OPTIONS] VARIABLE(s)
```

Commonly used `OPTIONS` include:

`-f`	refers to a shell function
`-a`	referes to an array of variables

The values of these variables can then no longer be changed by subsequent assignments. If no arguments are given, or if `-p` is supplied, a list of all read-only variables is displayed. Using the `-p` option, the output can be reused as input.

The return status is zero, unless an invalid option was specified, one of the variables or functions does not exist, or `-f` was supplied for a variable name instead of for a function name.

```
$ readonly TUX="penguinpower"
$ TUX="Mickeysoft"
 -bash: TUX: readonly variable
```

6.3.6 ARRAY VARIABLES

An array variable contains multiple values. Arrays are zero-based, the first element is indexed with the number 0. Indirect declaration is done using the following syntax:

```
MYARRAY[INDEXNR]=myvalue
```

The `INDEXNR` is treated as an arithmetic expression that must evaluate to a positive number.

Explicit declaration of an array is done using the declare built-in:

```
declare -a MYARRAY
```

Array variables may also be created using compound assignment in this format:

```
MYARRAY=(value1 value2 ...  valueN)
```

Adding missing or extra members in an array is done using the syntax:

```
MYARRAY[indexnumber]=value
```

Dereferencing the Variables in an Array: In order to refer to the content of an item in an array, use curly braces {}. This is necessary, as you can see from the following example, to bypass the shell interpretation of expansion operators. If the index number is `@` or `*`, all members of an array are referenced.

```
$ MYARRAY=(one two three)
$ echo ${MYARRAY[*]}
 one two three

$ echo $MYARRAY[*]
 one[*]

$ echo ${MYARRAY[2]}
 three

$ MYARRAY[3]=four
$ echo ${MYARRAY[*]}
 one two three four
```

In this example, `${MYARRAY[*]}` refers to all the elements in the array. Referring to the content of a member variable of an array without providing an index number is the same as referring to the content of the first element, the one referenced with index number zero. For example, `$MYARRAY` refers to the first element of the array, since the absence of the curly braces.

Deleting Array Variables: The `unset` built-in is used to destroy arrays or member variables of an array:

```
$ unset MYARRAY[1]
$ echo ${MYARRAY[*]}
 one three four

$ unset MYARRAY
$ echo ${MYARRAY[*]}
 <--no output-->
```

6.4 OPERATIONS ON VARIABLES

6.4.1 STRING OPERATIONS

Length of a Variable: Using the `${#VAR}` syntax will calculate the number of characters in a variable. The following examples demonstrate the usage.

```
$ echo $SHELL
 /usr/local/bin/bash

$ echo ${#SHELL}
 19

$ ARRAY=(one two three)
$ echo ${#ARRAY}
 3
```

In this example, `SHELL` is a variable with the value `/usr/local/bin/bash`. `${#SHELL}` calculates the length of `SHELL`, which is 19. `ARRAY` is an array variable with value `one two three`. The length of `ARRAY` is 3.

String Concatenation: The following example concatenates values of `str1` and `str2` and stores it in the third variable `str3`.

```
$ str1="Hello"
$ str2="World"
$ str3="$str1␣$str2"
$ echo $str3
 Hello World
```

Substring Extraction: To strip a substring from a string variable, starting from `OFFSET` location, with length `LENGTH`, use the following syntax:

`{VAR:OFFSET:LENGTH}`

If you want to assign the substring as the value to a new variable `NEWVAR`, use the following syntax:

`NEWVAR=${VAR:OFFSET:LENGTH}`

The `OFFSET` parameter defines the index of the character that the substring starts from. The index of the first character is 0. The `LENGTH` parameter defines how many characters to keep, starting from the first character after the offset point. If `LENGTH` is omitted, the remainder of the variable content is taken. The following examples show how to use this operation.

```
$ MYSTRING="thisisaverylongname"
$ echo ${MYSTRING:4}
 isaverylongname
$ echo ${MYSTRING:6:5}
 avery
```

Substring Replacement: To replace the first match of `substring` with `replacement` in `STRING`, use the following syntax:

`{STRING/substring/replacement}`

To replace all matches of `substring` with `replacement` in `STRING`, use the following syntax:

`{STRING//substring/replacement}`

The following example shows how to use this operation.

```
$ stringZ=abcABC123ABCabc
$ echo ${stringZ/abc/xyz} # Replaces first match of 'abc' with 'xyz'.
 xyzABC123ABCabc
$ echo ${stringZ//abc/xyz} # Replaces all matches of 'abc' with # 'xyz'.
 xyzABC123ABCxyz
$ echo "$stringZ"
 abcABC123ABCabc
```

The `substring` and `replacement` can also be variables.

```
$ substring=abc
$ repl=000
$ echo ${stringZ/$substring/$repl}
 000ABC123ABCabc
$ echo ${stringZ//$match/$repl}
 000ABC123ABC000
```

If `replacement` string is not specified, the `substring` is removed from the original string.

```
$ echo ${stringZ/abc}
 ABC123ABCabc
$ echo ${stringZ//abc}
 ABC123ABC
```

6.4.2 ARITHMETIC OPERATIONS

Arithmetic operations allow the evaluation of an arithmetic expression and the substitution of the result. The format for arithmetic expansion is:

`(( EXPRESSION ))`

Evaluation of arithmetic expressions is done in fixed-width integers with no check for overflow, although division by zero is trapped and recognized as an error.

Table 6 summaries the arithmetic operations.

Table 6: Arithmetic Operations

Operator	Meaning
`VAR++` and `VAR--`	Variable post-increment and post-decrement.
`++VAR` and `--VAR`	Variable pre-increment and pre-decrement.
`+` and `-`	Addition, subtraction.
`!` and ∼	Logical and bitwise negation.
`**`	Exponentiation.
`*`, `/` and `%`	Multiplication, division, and remainder.
<=, >=, < and >	Comparison operations.
== and `!=`	Equality and inequality.
`&`	Bitwise AND.
`|`	Bitwise OR.
`^`	Bitwise exclusive OR.
`&&`	Logical AND.
`||`	Logical OR.
`expr ? expr : expr`	Conditional evaluation.
=, `*=`, /=, `%=`, +=, −=, <<=, >>=, `&=` and `|=`	Assignments.
`,`	Separator between expressions.

Shell variables are allowed as operands. The value of a variable is evaluated as an arithmetic expression when it is referenced. A shell variable does not need to be explicitly declared as an integer variable to be used in an expression. Within the double parentheses, variable dereferencing is optional. As shown in the following example, both operations are correct.

```
$ x=1
$ y=$((x+2))
$ echo $y
 3

$ y=$(($x+2))
$ echo $y
 3
```

An alternative way of performing arithmetic operations is to use the `let` construction.

```
$ x=1
$ let y=x+2
$ echo $y
 3

$ let y=$x+2
$ echo $y
 3
```

In addition, both string and integer variables support comparison operations which are mainly used in `IF` statements that will be discussed in Section 6.6.1. Note that the comparison operations for string and integer can take different forms.

6.5 QUOTING CHARACTERS

A lot of words have special meanings in some context or other. Quoting is used to remove the special meaning of characters or words. Quotes can disable special treatment for special characters. They can prevent reserved words from being recognized as such and they can disable parameter expansion.

6.5.1 ESCAPE CHARACTERS

Escape characters are used to remove the special meaning from one single character. A non-quoted backslash, \, is used as an escape character in Bash. It preserves the literal value of the next character that follows, with the exception of newline. If a newline character appears immediately after the backslash, it marks the continuation of a line when it is longer that the width of the terminal; the backslash is removed from the input stream and effectively ignored.

The following example shows the use of \ to escape the `$`.

```
$ year=2016
$ echo $year
 2016

$ echo \$year
 $year
```

In this example, the variable `year` is created and set to hold the value `2016`. The first `echo` displays the value of the variable. But the second `echo`, since the `$` is escapted, it displays the literal value of `$year`.

6.5.2 SINGLE QUOTES

Single quotes (‘’) are used to preserve the literal value of each character enclosed within the quotes. A single quote may not occur between single quotes, even when preceded by a backslash.

We continue with the previous example to show the use of single quotes.

```
$ echo '$year'
 $year
```

6.5.3 BACKTICKS

Commands between backticks (backward single quotes, `` ` ` ``) are replaced by the output of the command, minus the trailing newline characters. This is called ***Command Substitution*** because context is substituted with the output of the command.

The following example substitutes the content of `x` with the command output of `date`.

```
$ date
 Tue Feb 16 14:02:35 EST 2016 #the output of date
$ x=`date`
$ echo $x
 Tue Feb 16 14:02:39 EST 2016
```

6.5.4 DOUBLE QUOTES

Double quotes preserve the literal value of all characters enclosed, except for the dollar sign (`$`), the backticks (backward single quotes, `` ` ` ``) and the backslash.

The dollar sign (`$`) and the backticks (`` ` ` ``) retain their special meaning within the double quotes.

A double quote (`""`) may be quoted within double quotes (`""`) by preceding it with a backslash (\).

The following examples show the usage of double quotes.

```
$ year=2016
$ echo "$year"
 2016

$ echo "`date`"
 Tue Feb 16 14:20:23 EST 2016

$ echo "I'd␣say:␣\"Go for it! \""
 I'd say: "Go␣for␣it!"

$ echo "\\"
\
```

6.6 CONDITIONALS AND LOOPS

One of the things that defines a complete programming language, like the Bash shell language, is the ability to make decisions, and the ability to repeat actions, possibly with different inputs. Now we are going to explore some of the ways to do these things in Bash.

Conditionals

We can make decisions based on the answers to logical questions. We can ask things like "does a file with this name exist?" or "is this number larger than that one?" or "is this string the same as that one?". It's not possible to enumerate all the different possible questions and answers we might find useful, but let's explore a few.

6.6.1 test

`test, [` returns a true or false value for some condition that it tests. For instance:

`test 5 -gt 3`

would return true, and

`test 5 -lt 3`

would return false.

The `test` command can also be called by enclosing the question in square brackets, so the same examples could be written as: `[ 5 -gt 3 ]` and `[ 5 -lt 3 ]`

Note the spaces next to the brackets – those are important!

`test`, or `[]`, supports a large number of questions that can be asked. Some of the most useful are:

`STRING1 = STRING2`	The strings are equal
`STRING1 != STRING2`	The strings are not equal
`INTEGER1 -eq INTEGER2`	The integers are equal.
`INTEGER1 -ne INTEGER2`	The integers are not equal.
`INTEGER1 -gt INTEGER2`	The first integer is greater than the second.
`INTEGER1 -lt INTEGER2`	The first integer is less than the second.
`-f FILE`	The file named `FILE` exists and is a normal file.
`-e FILE`	The file named `FILE` exists and may be any type of entry in the filesystem.
`-s FILE`	The file named `FILE` exists and has a size greater than zero.
`-d FILE`	The directory named `FILE` exists.
`-r FILE`	The file named `FILE` is readable.
`-w FILE`	The file named `FILE` is writeable.

6.6.2 if

Once we can determine whether a condition is true or false, we can take action based on that determination. We can do that with the `if` command.

Basic syntax:

```
if condition; then do-this; fi
if condition; then do-this; else do-that; fi
if condition; then do-this; elif other-condition; then do-that; else do-other-thing; fi
```

We can also test for the success of a command. For example if you simply want to know whether the string "AAGCTT" appears in a file of DNA sequences, you can do something like:

Example:

```
if grep -q "AAGCTT" filename; then
   echo "Yes,␣it's␣in␣there"
else
   echo "No,␣didn't␣find␣it␣this␣time"
fi
```

In that example, the `grep` command will return a success status if the string is found, and so the `if` command will print the "Yes" message; if the string is not there, `grep` returns a failure status, and `if` will print the "No" message.

One simple way we can use the `if` and `test` commands is to check the number of parameters passed to a script, and if the number is not what we expect, we write out a usage reminder and exit.

Example:

```
#!/bin/bash

echo $#

if [ $# -ne 2 ]; then
   echo "Should␣have␣2␣parameters"
   exit;
fi

echo "You've␣got␣it␣right"
```

Our simple script just counts the number of parameters, and insists there should be 2 of them. If there are, it compliments the user.

6.6.3 for

We can also repeat actions, usually with different inputs each time. One of the common ways we can do this in bash is with the `for` command.

Basic Syntax:

```
for arg in [list]
do
   commands ...
done
```

We can enumerate the elements in the list that we want the argument to use:

```
for name in "Bill" "Joe" "Mary"
do
   mail $name < message.txt
done
```

We can use a variable for the list as well:

```
#!/bin/bash

FILES="set1.fa␣set2.fa␣set3.fa"

for filename in $FILES
do
   if ! grep -q "^>" $filename; then
      echo "$filename␣has␣no␣sequences"
   fi
done
```

We can also commands such as `ls` to generate the list of variables:

```
#!/bin/bash

for filename in `ls`
do
   if ! grep -q "^>" $filename; then
      echo "$filename␣has␣no␣sequences"
   fi
done
```

The above script will check every file in the current working directory if it contains sequences.

We can also use filename expansion:

```
#!/bin/bash

for filename in *.fa
do
   if ! grep -q "^>" $filename; then
      echo "$filename␣has␣no␣sequences"
   fi
done
```

6.6.4 while

We can also run commands while a condition is true. One place this is very useful is when we want to read values from an input stream as long as the input keeps coming, and then gracefully quit. For example we can display line numbers when we `cat` a file:

```
#!/bin/bash

count="1"

cat $1 |
   while read line
   do
      echo "${count}:␣"$line
      (( count++ ))
   done
```

6.6.5 break, continue

What if we decide in the middle of a loop that we don't want to be running a loop anymore? For instance, what if we want to find the first file in a list that contains a particular item? If we have a large list of files, and we find the first one fairly early, it would be inefficient to keep looking.

Here's where **break** and **continue** come to our assistance.

`break` will completely quit a loop, and proceed to the next command after the loop body.
`continue` will skip to the next item in the loop, then continue the loop as usual.

Let's look at a couple of simple examples:

```
#/bin/bash

for i in {1..25}
do
   if [ $i -eq 12 ]
   then break;
   fi

   echo $i
done
```

```
#/bin/bash

for i in {1..25}
do
   if [ $i -eq 12 ]
   then continue;
   fi

   echo $i
done
```

Type them into your terminal shell, and see what the difference is in the output. Look closely, the second one may be hard to spot.

6.7 INPUT AND OUTPUT

There are shell scripts that work interactively with the users by prompting for some user input information, such as the user's name. In other cases, shell scripts read input from a file, and write output to a file. In this section, we'll take a closer look at how Bash scripts deal with input and output.

6.7.1 STDIN, STDOUT, AND STDERR

If we recall from Section 5.3.5, programs or commands can read input from standard input `stdin`, write output to standard output `stdout`, and write error messages to standard error `stderr`.

Bash scripts as programs can read user input from `stdin` using the command `read`. The built-in command `read` reads one line from the standard input. This gives the opportunity for the script to collect user input in an interactive manner. The syntax of `read` is:

```
read [OPTIONS] NAME1 NAME2 ...  NAMEN
```

This reads one line from the standard input, and assigns the first word of the line to variable `NAME1`, the second word of the line to variable `NAME2`, etc. Bash uses its internal variable `IFS` (Internal Field Separator) as the separator to split lines into words (or tokens). The default value of `IFS` is whitespace (space, tab, and newline). You can change the value of `IFS` to be something else. For example, you can change `IFS` to be **:** to parse colon-separated words. It is good practice to save the original `IFS` and restore it afterwards. The following script snippet helps you do that.

```
#!/bin/bash

 OIFS=$IFS #save the original IFS into OIFS
 IFS=":" #change IFS to colon

 commands; # do your thing

 IFS=$OIFS #restore the IFS after your are done
```

Some commonly used `OPTIONS` include:

`-a NAME`	The words are assigned to sequential indexes of the array variable `ANAME`, starting at `0`. All elements are removed from `ANAME` before the assignment. Other `NAME` arguments are ignored.
`-d DELIM`	The first character of `DELIM` is used to terminate the input line, rather than newline.
`-e`	`readline` is used to obtain the line.
`-n NCHARS`	`read` returns after reading `NCHARS` characters rather than waiting for a complete line of input.
`-t TIMEOUT`	Cause `read` to time out and return failure if a complete line of input is not read within `TIMEOUT` seconds. This option has no effect if `read` is not reading input from the terminal or from a pipe.
`-u FD`	Read input from file descriptor `FD`.

If the number of supplied variables `NAMEN` is less than the number of words in the line, the leftover words and their intervening separators are assigned to the last variable `NAMEN`. If the number of supplied variables `NAMEN` is more than the number of words in the line, the remaining variables are assigned empty values.

The return value of `read` command is zero, unless an end-of-file character is encountered. We'll see how to use this to read an entire file line by line for processing in next section.

The following script demonstrates how to use `read` to ask the user to enter his or her name and gender.

```
#!/bin/bash

# name_gender.sh
# This script asks the user to input name and gender, and echoes that information back to the
    user.

echo -n "Enter␣your␣name␣and␣press␣[ENTER]:␣"
read name

echo -n "Enter␣your␣gender␣and␣press␣[ENTER]:␣"
read gender

echo "You␣have␣entered␣name:␣$name␣and␣gender:␣$gender"
```

The script prints `Enter your name and press [ENTER]:` to your screen, you need to enter your name and press `ENTER`. Then the script prints `Enter your gender and press [ENTER]:`, you need to enter your gender. Then the script prints back the name and gender that you just entered. You can find this script in **/scripts/name_gender.sh**. If you `cd` to the directory, and run `./name_gender.sh`, you will see how the script guide you through. This is just one simple example to show how `read` can collect user input.

Similarly, Bash scripts can write output and error messages to `stdout` and `stderr`. We have been suing the `echo` command to see the output on our terminal (`stdout`). Another built-in command `printf` also prints output to `stdout`. Refer to the man page for the complete usage.

6.7.2 FILE I/O

It happens quite often that the scripts read input from one or multiple files instead of from `stdin`. For example, when you want to perform specific commands to lines of a file. There are multiple ways to read input from a file. One of them is to use the built-in `exec` and `read`.

Before we discuss `exec` in more details, we need to revisit the concept of ***redirection*** and ***file descriptors***. In your current shell, when executing a command, you can redirect the input and output of a command to a file, or to the input of another command. Redirection can also happen in a script, so that it can receive input from a file, or send output to a file. Bash tracks all open files for a given process using numeric values. These values are also known as integer handles, or ***file descriptors***. The best known file descriptors are `stdin`, `stdout`, and `stderr`, with file descriptors as `0`, `1`, and `2`, respectively. These numbers are reserved. Reading from a file instead of from `stdin` can be achieve by redirecting `stdin` to a file. File descriptors are widely used in redirection operations.

Now let's take a look at the command `exec`. `exec` is used to alter the file descriptors of the current shell. For example, this command: `exec < FILENAME` redirects `stdin` to the file `FILENAME`. From then on, all `stdin` comes from `FILENAME`, rather than its normal source the keyboard. This provides a method of reading `FILENAME` line by line and parsing each line using advanced commands such as `sed` or `awk`.

The following script reads from a file specified by user line by line, then simply prints each line back to the screen. You can find this script in **/scripts/simple_read_file.sh**. To run the script, type command:

```
./simple_read_file.sh data-file
```

```
#!/bin/bash

# simple_read_file.sh
# This script reads line by line from file specified by user, then prints each line on screen

INFILE=$1
exec < $INFILE

echo "The␣content␣of␣the␣file:␣"

while read line; do
      echo $line
done
```

Similarly, you can use the command `exec > FILENAME` in a script to redirect `stdout` to a designated file. This sends all command output that would normally go to `stdout` to `FILENAME`.

There are more ways for a script to read and write using files, the above example is a simple way to do so. We encourage you to explore more with your Bash scripts.

7 ACKNOWLEDGEMENTS

The material in this workshop was prepared by the Purdue University ITaP Research Computing team. We have drawn from documentation provided by the Purdue Bioinformatics Core used in the "*UNIX for Biologists*" workshop and *Next-generation Transcriptome Analysis Workshop Manual* provided by Professor Michael Gribskov and Professor Esperanza Torres. We have used The Linux Documentation Project (http://www.tldp.org) as a reference in this document.