Bash

Streams, Redirection, and Control Structures

Warm Up

- Write a simple bash script that takes in a file name as an argument, and does the following:
    - Sorts that file, and outputs the results to the screen
    - Paste that file to another file with the same name, but all o's replaced with e's, and outputs it to the screen

In [ ]:

```
./src/shell/demo1.sh data/noodles

#1=data/noodles
sort $1
paste $1 ${1//o/e}
```

Streams

- STDIN
- STDOUT
- STDERR

Output Redirection

- The greater than symbol (**>**), is used to redirect output
    - With no additional symbols, this redirects STDOUT to the specified location
    - **1>** also redirects STDOUT to the specified location, but this form is not normally used
    - **2>** redirects STDERR to the specified location
    - **&>** redirects both STDOUT and STDERR to the same specified location
    - **>>** appends STDOUT to the specified file

```
In [ ]: echo "Hello" > data/hello.txt
```

```
In [ ]: more data/hello.txt
```

```
In [ ]: echo "World" >> data/hello.txt
```

```
In [ ]: more data/hello.txt
In [ ]: gcc no_file.c
```

```
In [ ]: gcc no_file.c 2> data/gcc_errors.txt
```

```
In [ ]: more data/gcc_errors.txt
In [ ]: more src/python/out_and_err.py
```

```
In [ ]: ./src/python/out_and_err.py > out 2> err
```

```
In [ ]: more out
```

```
In [ ]: more err
```

/dev/null

- Unix has a special device that allows streams to be redirected to it but doesn't save any of the redirected text
- By redirecting to **/dev/null** you are throwing away that stream
    - Can be very useful to ignore errors, but many commands have a quiet option built in

In [ ]:

```
gcc no_file 2>/dev/null
```

Input Redirection

- The less than symbol (**<**) is used to redirect input to STDIN
  - Not many variations of this, but....
  - Two less than operators (**<<**) are used to create a here document, which will have its own slide

```
In [ ]: more src/python/simple.py
```

In [ ]:

```
./src/python/simple.py < data/numbers.txt
```

Here Documents

- A here document takes any string and allows it to be passed to a command as if it were coming from STDIN
    - For commands that take multiple arguments, you may see the dash (**-**) being used to explicitly indicate which argument should use STDIN
    - The **<<** must be followed by a delimiter that is used to mark the end of the HERE document
    - Using **<<-** will remove leading tabs, which can be useful for formatting nice looking scripts

Here Strings

- If all you want to redirect is a single line, you can use three less than symbols (**<<<**) with no delimiter to indicate a here string
    - Any variables in a here string (or here document) are expanded before being redirected

In [ ]:

```
more data/numbers.txt
```

In [ ]:

```
diff - data/numbers.txt <<EOF
40
1
2
3
EOF
```

In [ ]:

```
diff - data/numbers.txt <<< "Hello"
```

Pipes

- Many times the output of one command will function as the input to a second command
- Rather than redirect output to a tempoarary file and then use that file as input, use the pipe command (**|**)
    - The STDERR stream can be redirection *along with* the STDOUT stream using **|&**

In [ ]:

```
ls -lh | wc -l
```

In [ ]:

```
find ~/ -size +100M 2>/dev/null | head
```

Redirection and Pipe Practice

- Combine the `find` and `sort` commands to produce a sorted list of all files over 10M in a directory. Redirect the output to a file called big_files.txt

In [ ]:

```
find ~ -size +10M 2> /dev/null | sort > big_files.txt
more big_files.txt
```

Tee

- The `tee` command takes in a stream as input, and outputs that stream both to STDOUT and to the specified file
  - Used following a pipe operator

In [3]:

```
pip2 install -U asdfadsf |& tee scipy.log
```

```
Collecting asdfadsf
  Could not find a version that satisfies the requirement asdfadsf (from versi
ons: )
No matching distribution found for asdfadsf
```

In [4]:

```
more scipy.log
```

```
Collecting asdfadsf
  Could not find a version that satisfies the requirement asdfadsf (from versi
on
s: )
No matching distribution found for asdfadsf
```

Redirecting From Multiple Commands

- Sometimes you may need to combine the output of multiple commands and pass this on to a third or fourth command
- You could use temporary files, but process substitution fills this need nicely
- The syntax is **<(*command*)** (Known as process substitution)
    - This relies on certain operating system features, so isn't truly portable, but can be assumed to be

In [6]:

```
diff <(ls -lh .) <(ls -lh ~/Teaching/CMSC331)
```

```
1,22c1,13
< total 177M
< -rw-rw---- 1 bryan bryan    0 Feb 14 14:26 an_empty_file
< -rw-rw---- 1 bryan bryan  57K Feb 14 16:43 big_files.txt
< drwxr-x--- 2 bryan bryan 4.0K Feb 14 15:00 binder
< drwxr-x--- 2 bryan bryan 4.0K Feb 14 15:25 data
< -rwxr-x--- 1 bryan bryan 176M Sep 11 22:40 en.openfoodfacts.org.products.csv
< -rw-rw---- 1 bryan bryan   15 Feb 14 16:34 err
< -rwxrwx--- 1 bryan bryan 5.4K Feb 12 14:53 Git.ipynb
< drwxrwx--- 2 bryan bryan 4.0K Feb 10 13:17 helper_scripts
< drwxrwxr-x 2 bryan bryan 4.0K Feb 14 14:54 img
< -rwxr-x--- 1 bryan bryan 176K Nov 20 22:25 jupyter-php-installer.phar
< -rw-rw---- 1 bryan bryan 297K Feb 14 15:00 Lecture00.ipynb
< -rw-rw---- 1 bryan bryan  43K Feb 14 15:00 Lecture01.ipynb
< -rwxrwx--- 1 bryan bryan  43K Feb 12 14:53 Lecture02.ipynb
< -rwxrwx--- 1 bryan bryan  26K Feb 12 14:53 Lecture03.ipynb
< -rwxrwx--- 1 bryan bryan  71K Feb 14 15:06 Lecture04.ipynb
< -rw-rw---- 1 bryan bryan  28K Feb 14 16:46 Lecture05.ipynb
< -rw-rw---- 1 bryan bryan   15 Feb 14 16:34 out
< -rw-rw---- 1 bryan bryan   93 Feb 14 14:54 pngs
< -rw-rw---- 1 bryan bryan  149 Feb 14 16:45 scipy.log
< drwxr-x--- 6 bryan bryan 4.0K Feb  9 15:33 src
< lrwxrwxrwx 1 bryan bryan   26 Feb 12 15:26 upload -> ../teaching_scripts/upl
oad
---
> total 228K
> drwxrwx--- 2 bryan bryan 4.0K Feb 11 14:57 data
> drwxrwx--- 2 bryan bryan 4.0K Feb 11 19:51 img
> -rw-rw---- 1 bryan bryan  19K Feb 11 14:57 Lecture00.ipynb
> -rw-rw---- 1 bryan bryan  21K Feb 11 14:57 Lecture01.ipynb
> -rw-rw---- 1 bryan bryan  17K Feb 14 10:10 Lecture02.ipynb
> -rw-rw---- 1 bryan bryan  20K Feb 13 14:42 Lecture03.ipynb
> -rw-rw---- 1 bryan bryan  21K Feb 13 13:58 Lecture04.ipynb
```

In [7]:

```
head -n1 data/part1.tsv
```

```
1       Hydrogen        H       1.008   14.01
```

In [8]:

```
head -n1 data/part2.csv
```

```
H,1776
```

In [9]:

```
paste <(cut -f2 data/part1.tsv) <(cut -f2 data/part2.csv -d,)
```

```
Hydrogen        1776
Helium  1895
Lithium 1817
Beryllium       1797
Boron   1808
```

Process Substitution Practice

- Use process substitution to shuffle two files, concatenate them together, and shuffle the final results
  - data/numbers.txt - The list of numbers from before
  - data/letters.txt - A list of the letters of the alphabet, one per line

```
In [14]: cat <(shuf data/numbers.txt) <(shuf data/letters.txt) | shuf
```

```
2
40
1
3
z
```

xargs

- Theoretically, you could pass the `rm` command a long list of directories to delete
  - When this list of arguments becomes arbitarilaly too long, `rm` may break
  - It is better to call `rm` on each of the directories in turn
- xargs allows us to process a string, determine what the arguments are and how to split them up, and how many times to call a command
  - Very useful for calling a command on the output of `find`

```
In [15]: echo 1 2 3 4 | xargs ls
```

```
ls: cannot access '1': No such file or directory
ls: cannot access '2': No such file or directory
ls: cannot access '3': No such file or directory
ls: cannot access '4': No such file or directory
```

```
In [16]: ls *.ipynb | xargs file
```

```
Git.ipynb:       ASCII text
Lecture00.ipynb: UTF-8 Unicode text, with very long lines
Lecture01.ipynb: ASCII text, with very long lines
Lecture02.ipynb: UTF-8 Unicode text, with very long lines
Lecture03.ipynb: ASCII text, with very long lines
Lecture04.ipynb: UTF-8 Unicode text
Lecture05.ipynb: ASCII text
```

In [17]:

```
ls img/*.png | xargs -I{} convert {} {}.jpg
```

In [18]:

```
rm img/*.jpg
ls img/*.png > pngs
more pngs
xargs -IFILE convert FILE FILE.jpg < pngs
ls img/*.jpg
```

```
img/ajax-fig1.png
img/ajax-fig2.png
img/fb_messenger.png
img/fb_verify.png
img/registers.png
img/ajax-fig1.png.jpg  img/fb_messenger.png.jpg  img/registers.png.jpg
img/ajax-fig2.png.jpg  img/fb_verify.png.jpg
```

If-Then-Else

- The `if` block must end with `fi`
- The `then` keyword is required in bash
    - For both `elif` and `if`
    - Must be on a different line or follow on the same line after a semicolon

```bash
if CONDITIONAL; then
#CODE
elif CONDITIONAL; then
#CODE
else
#CODE
fi
```

If-Then-Else

- The `if` block must end with `fi`
- The `then` keyword is required in bash
    - For both `elif` and `if`
    - Must be on a different line or follow on the same line after a semicolon

      ```
      if CONDITIONAL
      then
      #CODE
      elif CONDITIONAL
      then
      #CODE
      else
      #CODE
      fi
      ```

Conditional Expression in Bash

- Binary expressions in bash are evaluated
  - Using the `test` command
  - Using the `[` command (an alias of `test`)
  - Using the `[[` syntax
- Results are stored as a return code
  - Not normally invoked on its own
- Whitespace is very important

[and test vs [[

- [and test are commands
- [[is part of bash syntax
  - Allows for easier composition of conditionals using && and ||
  - Parentheses don't have to be escaped
  - Can do pattern matching and regular expressions as a conditional

Conditional Operators

- Bash has three types of conditional operators
  - numeric operators
  - string operators
  - file operators
- You can always negate an comparison by using `!` in front of it

Conditionals on Numbers

- Equal: -eq
- Not Equal: -ne
- Greater Than: -gt
- Greater Than or Equal: -ge
- Less Than: -lt
- Less Than or Equal: -le

In [21]:

```
if [ 1 -eq 7 ]; then
echo "What math are you doing?"
else
echo "One is not equal to 7"
fi
```

```
One is not equal to 7
```

In [22]:

```
if [ 1 -ne 7 ]; then
echo "One is not equal to 7"
else
echo "What math are you doing?"
fi
```

```
One is not equal to 7
```

In [23]:

```
if [ ! 1 -eq 7 ]; then
echo "What math are you doing?"
else
echo "One is not equal to 7"
fi
```

```
What math are you doing?
```

In [24]:

```
a=1
b=2
if [ $a -lt $b ]; then
echo "$a is smaller than $b"
else
echo "$b is smallter than $a"
fi
```

```
1 is smaller than 2
```

In [27]:

```
a=1
b=2
if [[ $a -lt $b && $b -gt $a ]]; then
echo "$a is smaller than $b"
else
echo "$b is smallter than $a"
fi
```

```
1 is smaller than 2
```

Conditionals on Strings

- Equal: =
- Not Equal: !=
- Is Empty: -z
- Is Not Empty: -n

In [28]:

```
string1="A string"
string2="Another string"
string3=
if [[ $string1 = $string1 ]]; then
echo "The strings are the same"
fi
```

```
The strings are the same
```

In [29]:

```
if [[ -z $string3 ]]; then
echo "The string is empty"
fi
```

```
The string is empty
```

In [30]:

```
if [[ -n $string2 ]]; then
echo "The string is not empty"
fi
```

```
The string is not empty
```

Conditionals on Files

- There are about 20 different tests that can be performed on a file
    - `man test` shows them all
- Some common ones are:
    - Existence: -e
    - Is a file: -f
    - Is a directory: -d
    - Is readable/writable/executable: -r/-w/-x
    - Isn't empty: -s

In [31]:

```
more data/a_missing_file
```

```
more: stat of data/a_missing_file failed: No such file or directory
```

In [32]:

```
if [[ ! -e 'a_missing_file' ]]; then
echo "Lets make a file" > data/a_missing_file
fi

more data/a_missing_file
```

```
Lets make a file
```

In [33]:

```
touch an_empty_file
if [[ -e 'an_empty_file' ]]; then
echo "An empty file exists"
fi
if [[ -s 'an_empty_file' ]]; then
echo "The file isn't empty"
fi
```

```
An empty file exists
```

In [34]:

```
if [ -f . ]; then
echo "This directory isn't a file...something is messed up"
else
echo "All is right in the world"
fi
```

```
All is right in the world
```

If Statement Practice

- Write a simple bash script that prints "Be Careful" if the argument passed to it is
  - A file and
  - Writable and
  - Not empty

In [38]:

```
a_name=data/numbers.txt
if [[ -f $a_name && -w $a_name && -s $a_name ]]; then
    echo "Be Careful"
fi
```

```
Be Careful
```

Switch Statements

- Switch statements start with the keyword `case` and end with the keyword `esac`
- Each clause is a pattern to match the expression against
    - The pattern in a clause ends with a right parentheses **)**
    - A clause must end with two semicolons (**;;**)

In [3]:

```
expression="This is a String"

case $expression in
    0)
        echo "The variable is 0"
        ;;
    *ing)
        echo "The variable ends in ing"
        ;;
    *String)
        echo "The variable ends in String"
        ;;
    *)
        echo "This is the default"
        ;;
esac
```

```
The variable ends in ing
```

For Loops

- Bash has traditionally used a foreach style loop (similar to Python)
- Can loop over any type of array
    - Can also loop over files
- Both loops have the general syntax of

```
for EXPRESSION(S); do
# CODE_GOES_HERE
done
```

Foreach Style Loop

- The foreach style loop uses the setup of

```
for variable in list; do
```

- list can be
  - a space seperated list
  - an expanded array
  - a shell-style regular expression (globbing)
  - the output of a command

In [4]:

```
for x in 1 2 3; do
    echo $x;
done
```

```
1
2
3
```

In [5]:

```
my_array=(1 2 3)
for y in ${my_array[@]}; do
    echo $y
done
```

```
1
2
3
```

In [6]:

```
for f in *.ipynb; do
    wc -l $f
done
```

```
176 Git.ipynb
687 Lecture00.ipynb
1580 Lecture01.ipynb
1515 Lecture02.ipynb
937 Lecture03.ipynb
2853 Lecture04.ipynb
1580 Lecture05.ipynb
972 Lecture06.ipynb
```

For Loop Practice

- Write a for loop that finds the most common line in each file in the data directory
    - Hint: use head to find **most** common

In [13]:

```
for f in data/*; do
    sort $f | uniq -c | sort -n --key=2 | head -n1
done
```

```
      1 Aalborg Aalborg Airport AAL     Denmark Europe
sort: write failed: 'standard output': Broken pipe
sort: write error
      1 Lets make a file
      1 1.2G    Downloads
      1 1 Hydrogen
      1 0% Fat Greek Style Yogurt With Honey    04/08/2017      France  0.0
0.5     6.5     0.0     11.8    0.0     0.0     0.0     0.0     70.8661417323
0.0
      1 code    url     creator created_t       created_datetime        last_m
odified_t       last_modified_datetime  product_name    generic_name    quanti
ty      packaging       packaging_tags  brands  brands_tags     categories
categories_tags categories_en   origins origins_tags    manufacturing_places
manufacturing_places_tags       labels  labels_tags     labels_en       emb_co
des     emb_codes_tags  first_packaging_code_geo        cities  cities_tags
purchase_places stores  countries       countries_tags  countries_en    ingred
ients_text      allergens       allergens_en    traces  traces_tags     traces
_en     serving_size    no_nutriments   additives_n     additives       additi
ves_tags        additives_en    ingredients_from_palm_oil_n     ingredients_fr
om_palm_oil     ingredients_from_palm_oil_tags  ingredients_that_may_be_from_p
alm_oil_n       ingredients_that_may_be_from_palm_oil   ingredients_that_may_b
e_from_palm_oil_tags    nutrition_grade_uk      nutrition_grade_fr      pnns_g
roups_1 pnns_groups_2   states  states_tags     states_en       main_category
```

C-Style Loop

- Support for the C-style loop is widespread in bash, but not all shell scripts
- The syntax for the C-style loop is:

```
for (( START ; END ; CHANGE)); do
```

- The variable isn't prefixed with the dollar sign ($) inside the loop definition

In [14]:

```
for ((x = 1; x < 4; x++)); do
    echo $x
done
```

```
1
2
3
```

In [15]:

```
for ((x = 1; x < 4; x += 2)); do
    echo $x
done
```

```
1
3
```

seq Command

- There are many other ways to do a c-style loop while using the traditional syntax
- One option is the `seq` command, which returns a list of numbers
- The syntax of the `seq` command is

```
seq START INCREASE? END
```

In [16]:

```
for i in $(seq 1 3); do
    echo $i
done
```

```
1
2
3
```

In [17]:

```
for i in $(seq 0 2 10); do
    echo $i
done
```

```
0
2
4
6
8
10
```

Brace Expansion

- Another feature of bash that is often, but not exclusively used, with loops is brace expansion
- Bash will expand anything in braces into a list
- Braces can take two forms:

  ```
  {A_LIST,OF,OPTIONS}
  ```

  or

```
{START..END}
```

In [18]:

```
echo Lecture0{0,1,2,3,4,5}.ipynb | xargs ls -lh | cut -f6,7,8  -d' '
```

```
Feb 14 15:00
43K Feb 14
43K Feb 12
26K Feb 12
71K Feb 14
39K Feb 19
```

In [19]:

```
for i in {0..5}; do
    ls -lh Lecture0$i.ipynb | cut -f6,7,8 -d' '
done
```

```
Feb 14 15:00
Feb 14 15:00
Feb 12 14:53
Feb 12 14:53
Feb 14 15:06
Feb 19 16:18
```

While Loops

- While loops also use the `do` expression after the condition
- The syntax for a while loop is

```
while CONDITION; do
  #CODE_HERE
done
```

In [20]:

```
string='Some Characters'
while [[ -n $string ]]; do
    echo ${string:0:1}
    string=${string:1}
done
```

```
S
o
m
e
 
C
h
a
r
a
c
t
e
r
s
```

Until Loops

- The `until` loop is almost identical to the `while` loop, but continues until the statement is True
- The `until` is still places at the top of the loop and checked before entering it
- The syntax of `until` is

```
until CONDITIONAL; do
  #CODE GOES HERE
done
```

In [21]:

```
string='Some Characters'
until [[ -z $string ]]; do
    echo ${string:0:1}
    string=${string:1}
done
```

```
S
o
m
e

C
h
a
r
a
c
t
e
r
s
```