bash

I/O, Processes, and Math

User Input

- User input is gotten by using the `read` command
- The general syntax is

```
read [OPTIONS] variable_name
```

- Common options are:
  - -p <text>: Prompt the user with text before getting input
  - -s: Do not display the text the user types (for passwords, etc)
  - -t <time>: Time out after the given number of seconds

In [ ]:

```
#Example Code Can't be Run in Browser/Jupyter
echo "Enter some text:"
read text
echo "You entered $text"
```

In [ ]:

```
#Example Code Can't be Run in Browser/Jupyter
read -p "Enter some more text: " more_text
echo "Now you are telling me $more_text"
```

In [ ]:

```
#Must be -sp, -ps means "s" is the argument of -p
read -sp "Enter the secret word: " secret

#Not printing characters means that we need to
#explicitly move to the next line
echo
echo "Was I supposed to keep $secret a secret?" ~
```

In [ ]:

```
echo -n "Enter something quickly!: "
read -t5 user_input
if [[ -n $user_input ]]; then
    echo "Congrats! You beat the clock"
else
    echo
    echo "Too Slow! Better luck next time"
fi
```

Mapfile

- The `mapfile` command reads STDIN into an array, breaking it up at newlines
- Even though it reads from STDIN, it primarily used with the pipe character or redicrects
    - Not used for user interaction
- The syntax is

```
mapfile [OPTIONS] array_variable
```

In [27]:

```
mapfile numbers<<HERE
1
2
3
4
5
HERE

for number in ${numbers[@]}; do
    echo -n "$number, "
done
echo
```

```
1, 2, 3, 4, 5,
```

Reading A File with a Loop

- The `mapfile` command is generally more efficient, but is a recent addition to bash
- If you want to do something more than just read the lines in, it can still be useful to use a loop
- Reading a file in a loop combines three techniques
  - A `while` loop
  - A `read` command
  - Input redirection

In [28]:

```
while read line; do
    echo $line
done < data/numbers.txt
```

```
40
1
2
3
```

Processing a File Practice

- Read in a file named data/words.txt, and find the longest word in the file

In [35]:

```
max=""
while read line; do
    for word in $line; do
        if [[ ${#word} -gt ${#max} ]]; then
            max=$word
        fi
    done;
done < data/lines.txt

echo $max;
```

```
interconnection
```

Formatted Output

- The `printf` command allows output to be formatted with more control than echo
- It uses a syntax similar to most formatted strings you are familiar with
  - Based on printf from C
- Newlines are not automatically added
- The variables to print are given as arguments to the `printf` command after the format string

In [36]:

```
printf "%d is a number\n" 30
printf "%10d is a number\n" 30
printf "%010d is a number\n" 30
printf "%-10d is a number\n" 30
printf "%d is a big number\n" 10000000000
printf "%'d is a big number that is easier to read" 10000000000
```

```
30 is a number
        30 is a number
0000000030 is a number
30         is a number
10000000000 is a big number
10,000,000,000 is a big number that is easier to read
```

In [37]:

```
printf "%f is a float\n" 30
printf "%f is a float\n" 30.1345
printf "%.2f is a truncated float\n" 30.12345
printf "%'.2f is a truncated , yet big, float" 3000000000.12345
```

```
30.000000 is a float
30.134500 is a float
30.12 is a truncated float
3,000,000,000.12 is a truncated , yet big, float
```

In [38]:

```
printf "%s is a string\n" "Hello there"
#All Arguments are always printed
printf "%s was passed as an argument\n" Hello there
printf "%3s doesn't truncate the string\n" "A long string"
printf "%.3s does truncate the string\n" "A long string"
printf "%10.3s truncates the string\
, but prints with a width of 10" "A long string"
```

```
Hello there is a string
Hello was passed as an argument
there was passed as an argument
A long string doesn't truncate the string
A l does truncate the string
       A l truncates the string, but prints with a width of 10
```

Other Uses of `printf`

- Two rather unique format types are
  - `%q` will escape your string into an appropriate format for bash
  - `%(fmt)T` converts seconds into a user specified date string
    - `fmt` is other format commands for dates, similar to `strftime` function in C

In [40]:

```
printf %q "A directoryname with spaces/"
printf "\n"
printf "%(%A the %d of %B, %Y, at %r)T\n" -1
printf "%(%A the %d of %B, %Y, at %r)T" 0
```

```
A\ directoryname\ with\ spaces/
Monday the 19 of February, 2018, at 04:47:10 PM
Wednesday the 31 of December, 1969, at 07:00:00 PM
```

Running Other Scripts

- Other scripts can always be run like other commands, simply by calling them
- If you want to have access to all the variables, including function definitions, use the `source` command
    - The single dot `.` is an alias for the `source` command

```
. lots_of_definitions
source other_definitions
```

In [41]:

```
more src/shell/definitions.sh
```

```
#!/bin/bash
pi=3.1415
e=2.7182
zero=0.0000
alphabet=(A B C D E F G H I J K L M N O P Q R S T U V W X Y Z)
```

In [1]:

```
./src/shell/definitions.sh
echo $pi
```

In [2]:

```
. src/shell/definitions.sh
echo ${alphabet[*]}
```

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

Process Management

- When calling other commands it is useful to know how to control processes
- Common process control commands are
  - `COMMAND &` - executes command in background
  - `bg JOB_SPEC` - sends command to background
  - `fg JOB_SPEC` - brings background command to foreground
- If you are using the shell interactively
  - `jobs` list all currently running processes launched from this shell
  - `ps` list all processes on the computer

ps Command

- When you have many processes running its useful to know how to query them
- The `ps` command by default displays the pids for processes launched from this shell
- Common options are
  - -A: display all processes on the system
  - -f: display more information, such as who started the process
  - -F: display even more information
  - -o<format>: customize the information displayed
  - -u<user>: display all processes launched by user

```
In [3]: ps
```

```
  PID TTY          TIME CMD
12325 pts/4    00:00:00 bash
12470 pts/4    00:00:00 ps
```

In [ ]:

```
ps -f -ubryan | more
```

```
UID         PID  PPID  C STIME TTY          TIME CMD
bryan      1384     1  0 Feb16 ?        00:00:00 /lib/systemd/systemd --user
bryan      1385  1384  0 Feb16 ?        00:00:00 (sd-pam)
bryan      1401  1232  0 Feb16 ?        00:00:00 -fish -c \/opt\/google\/chrome
-r
emote-desktop\/chrome-remote-desktop --start --child-process
bryan      1402  1401  0 Feb16 ?        00:00:00 /usr/bin/python2 /opt/google/c
hr
ome-remote-desktop/chrome-remote-desktop --config=/home/bryan/.config/chrome-r
em
ote-desktop/host#269f963d97dcad68f51b2a9fc1292735.json --start --child-process
bryan      1488  1402  0 Feb16 ?        00:00:06 Xvfb :20 -auth /home/bryan/.Xa
ut
hority -nolisten tcp -noreset -screen 0 1536x864x24
bryan      1780  1402  0 Feb16 ?        00:00:00 /bin/sh -c /home/bryan/.chrome
-r
emote-desktop-session
bryan      1782  1780  0 Feb16 ?        00:00:00 /bin/sh /etc/xdg/xfce4/xinitrc
-
- /etc/X11/xinit/xserverrc
bryan      1783  1402  0 Feb16 ?        00:00:01 /opt/google/chrome-remote-desk
to
p/chrome-remote-desktop-host --host-config=- --audio-pipe-name=/home/bryan/.co
nf
ig/chrome-remote-desktop/pulseaudio#269f963d97/fifo_output --server-supports-e
xa
ct-resize --ssh-auth-sockname=/tmp/chromoting.bryan.ssh_auth_sock --signal-par
en
t
bryan      1806  1782  0 Feb16 ?        00:00:00 xfce4-session
bryan      1807  1384  0 Feb16 ?        00:00:01 /usr/bin/dbus-daemon --session
-
-address=systemd: --nofork --nopidfile --systemd-activation
```

Kill

- Despite it's name `kill` is a more general command then just ended processes
- The `kill` command can send signals to running processes
    - The signal can be sent using either its numerical value or name
        - -9 or -SIGKILL
    - To see a full list use `kill -l`
- Syntax

```
kill SIGNAL PID
```

In [1]:

```
# Launch a random background job
htop &
```

```
[1] 12581
```

In [3]:

```
kill -15 12581
```

In [9]:

```
jobs
```

In [8]:

```
kill -9 12581
```

The nohup Command

- One signal sent to processes is `SIGHUP` which is sent when a terminal closes
  - Comes from hang up
  - This will generally kill processes
- If we have a long running background task that we want to continue after the terminal is close, use the nohup command

```
nohup COMMAND &
```

Command Substitution

- We've used it a few times, but formally command substitution runs a command and returns it's output
- You may encounter two forms
  - `` `command` ``
  - `$(command)`
- Always use `$(command)`
  - It is nestable
  - It is safer

In [10]:

```
html_files=$(find . -name "*.ipynb")
echo $html_files
```

```
./Git.ipynb ./Lecture03.ipynb ./Lecture00.ipynb ./Lecture06.ipynb ./Lecture02.
ipynb ./Lecture05.ipynb ./Lecture04.ipynb ./Lecture01.ipynb ./.ipynb_checkpoin
ts/Lecture05-checkpoint.ipynb ./.ipynb_checkpoints/Lecture06-checkpoint.ipynb
./.ipynb_checkpoints/Lecture04-checkpoint.ipynb
```

In [11]:

```
ps_out=$(ps)
```

In [12]:

```
echo ${ps_out::10}
```

```
PID TTY
```

In [13]:

```
nesting=$(echo $(ls))
echo $nesting
```

```
an_empty_file big_files.txt binder data en.openfoodfacts.org.products.csv err
Git.ipynb helper_scripts img jupyter-php-installer.phar Lecture00.ipynb Lectur
e01.ipynb Lecture02.ipynb Lecture03.ipynb Lecture04.ipynb Lecture05.ipynb Lect
ure06.ipynb out pngs scipy.log src test.sh upload words.txt
```

Command Substitution Practice

- Use command substitution to print all the `ipynb` files in the directory, with `ipynb` removed
    - Hint: Use `${var//pattern/substitute}`

In [16]:

```
var=$(ls *ipynb)
echo ${var//.ipynb/}
```

```
Git Lecture00 Lecture01 Lecture02 Lecture03 Lecture04 Lecture05 Lecture06
```

Chaining Commands

- The `&&`,`||`, and `;` operators are used to chain commands together
  - `command1 && command2` only executes command2 upon successful exit of command1
  - `command1 || command2` only executes command2 upon unsuccessful exit of command1
  - `command1 ; command2` always executes command2

In [17]:

```
rm /home 2> /dev/null || echo "You can't do that"
[[ 1 -eq 1 ]] && echo "That is true 1"
[[ 1 -eq 2 ]] && echo "That is true 2"
[[ 1 -eq 2 ]] || echo "That isn't true 2"
```

```
You can't do that
That is true 1
That isn't true 2
```

Subshells

- A subshell is a group of commands run in a separate shell from the current process
- Changes to variables in the subshell will not be reflected in the main script
- Can also be used to send an entire group of commands to the background
- Syntax is

```
( COMANDS )
```

In [18]:

```
echo $(pwd)
(
    cd ~
    echo $(pwd)
)
echo $(pwd)
```

```
/home/bryan/Teaching/CMSC433
/home/bryan
/home/bryan/Teaching/CMSC433
```

In [19]:

```
printf "%'d is a big number\n" 1000000
(
    LANG=es_ES.UTF-8
    printf "%'d is a big number\n" 1000000
)
printf "%'d is a big number\n" 1000000
```

```
1,000,000 is a big number
1,000,000 is a big number
1,000,000 is a big number
```

Parallel Execution

- Parallel execution can be achieved easily using subshells and backgrounding processes
- Bash has a builtin command `wait` that will pause the execution of the script until all child processes have returned
- For more complex parallel applications, we will look at the GNU parallel suite of tools

In [20]:

```
#Supress notification of completed background jobs
set +m

(
    for letter in {A..M}; do
        echo "$letter ";
        sleep 0.5;
    done;
) &

(
    for number in 1 2 3 4 5 6 7; do
        echo  "$number ";
        sleep 0.25;
    done
) &

wait
echo "EVERYTHING IS AWESOME"
```

```
[1] 13117
A
[2] 13119
1
B
2
3
C
4
5
D
6
7
E
F
```

GNU Parallel

- GNU parallel is a collection of utilities to manage processes executing in parallel
- The `parallel` command executes a command in parallel given a list of arguments separated by `:::`

```
parallel echo ::: A B C ::: 1 2 3
```

- `parallel --pipe` allows parallel processing of STDIN
- The `sem` command is useful to combine with backgrounded subprocesses to limit how many run at a time

In [21]:

```
parallel echo ::: A B C ::: 1 2 3
```

```
A 1
A 2
A 3
B 1
B 2
B 3
C 1
C 2
C 3
```

In [22]:

```
parallel jupyter-nbconvert {} --to html ::: *.ipynb
```

```
[NbConvertApp] Converting notebook Git.ipynb to html
[NbConvertApp] Writing 256385 bytes to Git.html
[NbConvertApp] Converting notebook Lecture00.ipynb to html
/usr/local/lib/python3.6/dist-packages/nbconvert/filters/datatypefilter.py:41:
UserWarning: Your element with mimetype(s) dict_keys([]) is not able to be rep
resented.
  mimetypes=output.keys())
[NbConvertApp] Writing 563949 bytes to Lecture00.html
[NbConvertApp] Converting notebook Lecture06.ipynb to html
[NbConvertApp] Writing 298314 bytes to Lecture06.html
[NbConvertApp] Converting notebook Lecture01.ipynb to html
[NbConvertApp] Writing 323885 bytes to Lecture01.html
[NbConvertApp] Converting notebook Lecture04.ipynb to html
[NbConvertApp] Writing 336999 bytes to Lecture04.html
[NbConvertApp] Converting notebook Lecture05.ipynb to html
[NbConvertApp] Writing 315353 bytes to Lecture05.html
[NbConvertApp] Converting notebook Lecture02.ipynb to html
[NbConvertApp] Writing 317691 bytes to Lecture02.html
[NbConvertApp] Converting notebook Lecture03.ipynb to html
[NbConvertApp] Writing 293097 bytes to Lecture03.html
```

In [23]:

```
time (grep -P "\d\d\d-\d\d\d-\d\d\d\d" ~/Research/Data/wackypediaFlat.slim | wc -l )
#grep -P "\d\d\d-\d\d\d-\d\d\d\d" ~/wackypediaFlat.slim | wc -l
```

```
257

real	0m2.294s
user	0m2.090s
sys	0m0.204s
```

In [24]:

```
time parallel --pipe --block 100M 'grep -P "\d\d\d-\d\d\d-\d\d\d\d" | wc -l' <  ~/Research/Data/wackypediaFlat.slim
```

```
11
18
20
16
13
11
17
10
9
7
16
14
21
15
8
12
13
10
9
12
2
```

In [ ]:

```
# There are better ways to do this, ie all in one search

for letter in {A..Z}; do
(
        n=$(grep -P "($letter)\1" ~/wackypediaFlat.slim | wc -l)
        echo "$n double $letter's found"
        sleep 0.5;
)&
done;

wait
```

In [ ]:

```
# There are better ways to do this, ie all in one search

for letter in {A..Z}; do
(

        n=$(sem --id $$ -j3 grep "${letter}${letter}" ~/wackypediaFlat.slim | wc -l)
        echo "$n double $letter's found"
        sleep 0.5;
)&
done;

sem --wait --id $$
```

Splitting a File

- Splitting a file comes in handy when doing parallel processing, if you don't want to or can't use `parallel --pipe`
- The split command will automatically split a file according to various metrics, and create new files with a suffix like "aa"
- Common options
    - -n: Split into N chunks
    - -l: Split into files with L lines
    - -b: Split into files with B bytes in them

In [ ]:

```
split -l1 numbers.txt numbers_aa
```

In [ ]:

```
ls x*
```

In [ ]:

```
more numbersaa
```

Arithmetic

- bash supports only integer arithmetic natively
- The syntax to indicate arithmetic is double parentheses **((EXPRESSION))**
- Variables do not need to be expanded inside the double parentheses (no $ needed)
- Standard operators are supported
    - % is the module operator
    - ** is used for exponentiation

In [ ]:

```
echo $((0 + 11))
echo $((10/6))
echo $((10 * 6))
echo $((10 % 6))
```

In [ ]:

```
x=10
((x++))
echo $((x += 1))
echo $((x += 1))
```

In [ ]:

```
echo $((3.14 + 11 ))
```

Floating Point Arithmetic

- In order to perform floating point math, the `bc` command is used
  - The input is STDIN
- The syntax is very similar to C
  - To determine the precision of the output, prefix the math with `scale=PRECISION;`
  - The default is to truncate all floating point numbers

In [ ]:

```
bc <<< "0+5"
bc <<< "scale=2;10/6"
bc <<< "scale=2;3.14 + 11"
bc <<< "scale=2; sqrt(9)"
echo "scale=2; c(0)" | bc -l
echo "scale=2; s(0)" | bc -l
```