Week 03 Tutorial Sample Answers

1. Assume that we are in a shell where the following shell variable assignments have been performed, and *ls* gives the following result:

```
$ x=2  y='Y Y'  z=ls
$ ls
    a       b       c
```

What will be displayed as a result of the following *echo* commands:

a. `$ echo a   b   c`

b. `$ echo "a   b   c"`

c. `$ echo $y`

d. `$ echo x$x`

e. `$ echo $xx`

f. `$ echo ${x}x`

g. `$ echo "$y"`

h. `$ echo '$y'`

i. `$ echo $($y)`

j. `$ echo $($z)`

k. `$ echo $(echo a b c)`

ANSWER:

The shell performs command and variable substitution before splitting the command line into separate words to make up the arguments.
Single-quotes and double-quotes perform a grouping function that overrides the normal word-splitting.

a.
```
$ echo a   b   c
a b c
```

Spaces between arguments are not preserved;
`echo` puts one space between each argument.

b.
```
$ echo "a   b   c"
a   b   c
```

Spaces *are* preserved,
because the quotes turn `"a  b  c"` into a single argument to `echo`.

c.
```
$ echo $y
Y Y
```

`$y` expands into two separate arguments.

d.
```
$ echo x$x
x2
```

`$x` expands to `2` and is appended after the letter `x`.

e.
```
$ echo $xx

```

`$xx` is treated as a reference to the shell variable `xx`:
since there is no such variable, it expands to the empty string.

f.
```
$ echo ${x}x
2x
```

`${x}` expands to `2` and the letter `x` is appended.

g.
```
$ echo "$y"
Y Y
```

`$y` expands into a single argument.

h.
```
$ echo '$y'
$y
```

Single quotes prevent variable expansion.

i.
```
$ echo $($y)
Y: command not found
```

`$y` expands to `Y Y`
which is then executed as a command because of the `$()`;
since there is no command `Y`, an error message follows.

j.
```
$ echo $($z)
a b c
```

`$z` expands to `ls`,
which is then executed as a command,
giving the names of the files in the current directory,
which are treated as three separate arguments.

k.
```
$ echo $(echo a b c)
a b c
```

The inner `echo` command is executed,
giving `a b c`,
which are passed as arguments to the outer `echo`.

2. The following C program and its equivalent in Python3
all aim to give precise information about their command-line arguments.

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("#args = %d\n", argc - 1);

    for (int i = 1; i < argc; i++) {
        printf("arg[%d] = \"%s\"\n", i, argv[i]);
    }

    return 0;
}
```

```python
#!/usr/bin/env python3
from sys import argv

def main():
    print(f"#args = {len(argv) - 1}")
    for index, arg in enumerate(argv[1:], 1):
        print(f'arg[{index}] = "{arg}"')

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

Assume that these programs are compiled in such a way that we may invoke them as `./args`.
Consider the following examples of how it operates:

```
$ ./args a b c
#args = 3
arg[1] = "a"
arg[2] = "b"
arg[3] = "c"
$ args "Hello there"
#args = 1
arg[1] = "Hello there"
```

Assume that we are in a shell where the following shell variable assignments have been performed,
and /ls gives the following result:

```
$ x=2  y='Y Y'  z=ls
$ ls
    a       b       c
```

What will be the output of the following:

a.
```
$ ./args x y   z
```

b.
```
$ ./args $(ls)
```

c.
```
$ ./args $y
```

d.
```
$ ./args "$y"
```

e.
```
$ ./args $(echo "$y")
```

f.
```
$ ./args $x$x$x
```

g.
```
$ ./args $x$y
```

h.
```
$ ./args $xy
```

ANSWER:

a.
```
$ ./args x y   z
#args = 3
arg[1] = "x"
arg[2] = "y"
arg[3] = "z"
```

Each of the letters is a single argument (separated by spaces).

b.
```
$ ./args $(ls)
#args = 3
arg[1] = "a"
arg[2] = "b"
arg[3] = "c"
```

The `ls` command is executed and its output is interpolated into the command line;
the shell then splits the command-line into arguments.

c.
```
$ ./args $y
#args = 2
arg[1] = "Y"
arg[2] = "Y"
```

`$y` expands to the string `Y Y`;
when the shell splits the line into words,
these two characters becomes separate arguments.

d.
```
$ ./args "$y"
#args = 1
arg[1] = "Y Y"
```

`$y` expands to `Y Y` within the quotes,
so it is treated as a single word when the shell breaks the line into arguments.

e.
```
$ ./args $(echo $y)
#args = 2
arg[1] = "Y"
arg[2] = "Y"
```

The command within the backquotes expands to `Y Y`,
but since backquotes don't have a grouping function,
the two `Y`'s are treated as separate arguments

f.
```
$ ./args $x$x$x
#args = 1
arg[1] = "222"
```

`$x` expands into `2`,
which is concatenated with itself three times

g.
```
$ ./args $x$y
#args = 2
arg[1] = "2Y"
arg[2] = "Y"
```

`$x` expands to `2` and
`$y` expands to `Y Y`;
these two strings are concatenated to give `2Y Y` and,
when the shell splits the line into words,
the second `Y` becomes an argument in its own right.

h.
```
$ ./args $xy
#args = 0
```

There is no variable called `xy`,
so `$xy` expands to the empty string,
which vanishes when the shell splits the command line into words.

3. Imagine that we have just typed a shell script into the file `my_first_shell_script.sh` in the current directory.
We then attempt to execute the script and observe the following:

```
$ my_first_shell_script.sh
my_first_shell_script.sh: command not found
```

Explain the possible causes for this, and describe how to rectify them.

ANSWER:

- **Problem:**
  you might not have the current directory in your `$PATH`.

  Solution:
  either add `.` to the end of your `$PATH` (via `PATH=$PATH:.`),
  or type the command name as `./my_first_shell_script.sh`
- **Problem:**
  the `my_first_shell_script.sh` file may not be executable.

  Solution:
  either make the file executable, by running (e.g.,) `chmod +x my_first_shell_script.sh`,
  or execute it via the command `sh my_first_shell_script.sh` (also fixes the first problem)
- **Problem:**
  you might have gotten the `#!/bin/sh` line wrong.

  Solution:
  check the first line of the script,
  to make sure there are no spurious spaces or spelling mistakes,
  and then check that the shell is called `/bin/sh` on your system
- **Problem:**
  the `my_first_shell_script.sh` file has been transferred from a Windows-based computer in binary mode,
  and there is a carriage-return character,
  which is often rendered as `^M` (or `'\r'` in C) after `/bin/sh`.

  Solution:
  Run the command `dos2unix my_first_shell_script.sh`,
  which will remove the pesky `^M`s.

Note that some of these problems might result in a message like:
`my_first_shell_script.sh: Permission denied`,
depending on which shell you're using.

4. Implement a shell script called `seq.sh` for writing sequences of integers onto its standard output, with one integer per line.
The script can take up to three arguments, and behaves as follows:
   - `seq.sh` *`LAST`* writes all numbers from 1 up to *`LAST`*, inclusive. For example:

```
$ ./seq.sh 5
1
2
3
4
5
```

   - `seq.sh` *`FIRST LAST`* writes all numbers from *`FIRST`* up to *`LAST`*, inclusive. For example:

```
$ ./seq.sh 2 6
2
3
4
5
6
```

- `seq.sh` *`FIRST INCREMENT LAST`* writes all numbers from *`FIRST`* to *`LAST`* in steps of *`INCREMENT`*, inclusive; that is, it writes the sequence *`FIRST`*, *`FIRST + INCREMENT`*, *`FIRST + 2*INCREMENT`*, ..., up to the largest integer in this sequence less than or equal to *`LAST`*. For example:

```
$ ./seq.sh 3 5 24
3
8
13
18
23
```

ANSWER:

A sample solution for `seq.sh`:

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

case $# in
    1)
        FIRST=1
        INCREMENT=1
        LAST=$1
        ;;
    2)
        FIRST=$1
        INCREMENT=1
        LAST=$2
        ;;
    3)
        FIRST=$1
        INCREMENT=$2
        LAST=$3
        ;;
    *)
        echo "Usage: $0 [FIRST [INCREMENT]] LAST" >& 2
        exit 1
esac

if [ "$FIRST" -eq "$FIRST" ] 2> /dev/null; then
    :
else
    echo "$0: Error <FIRST> must be an integer" >& 2
    exit
fi

if [ "$INCREMENT" -eq "$INCREMENT" ] 2> /dev/null; then
    if [ "$INCREMENT" -gt 0 ]; then
        :
    else
        echo "$0: Error <INCREMENT> must be positive" >& 2
        exit
    fi
else
    echo "$0: Error <INCREMENT> must be an integer" >& 2
    exit
fi

if [ "$LAST" -eq "$LAST" ] 2> /dev/null; then
    if [ "$LAST" -gt "$FIRST" ]; then
        :
    else
        echo "$0: Error <LAST> must be greater than <FIRST>" >& 2
        exit
    fi
else
    echo "$0: Error <LAST> must be an integer" >& 2
    exit
fi

CURRENT="$FIRST"
while [ "$CURRENT" -le "$LAST" ]; do
    echo "$CURRENT"
    CURRENT=$(( CURRENT + INCREMENT ))
done
```

5. What is **JSON**?

   Where might I encounter it?

   Why can **JSON** be difficult to manipulate with tools such as **grep**?

   How can a tool like jq help?

   ANSWER:

   JSON (JavaScript Object Notation) is a very widely used standard file format.

It is classicly used to pass data to web front-end applications but is very widely used.

The description of an object in JSON is often spread over multiple lines.

This makes it very hard to apply line-based tools such as `grep`.

jq can convert the data from JSON to a format wherewe can apply familar tools such `sort`, `uniq` and `wc`.

6. Write a shell script, `no_blinking.sh`, which removes all HTML files in the current directory which use the blink element:

```
$ no_blinking.sh
Removing old.html because it uses the <blink> tag
Removing evil.html because it uses the <blink> tag
Removing bad.html because it uses the <blink> tag
```

ANSWER:

```
#!/bin/sh
# Removes all HTML files in the current directory which use <blink>

for file in *.html
do
    # note use of -i to ignore case and -w to ignore white space
    # however tags containing newlines won't be detected
    if grep -Eiw '</?blink>' "$file" >/dev/null
    then
        echo  "Removing $file because it uses the <blink> tag"
        rm "$file"
    fi
done
```

7. Modify the `no_blinking.sh` shell script to instead take the HTML files to be checked as command line arguments and, instead of removing them, adding the suffix **.bad** to their name:

```
$ no_blinking.sh awful.html index.html terrible.html
Renaming awful.html to awful.html.bad because it uses the <blink> tag
Renaming terrible.html to terrible.html.bad because it uses the <blink> tag
```

ANSWER:

```
#!/bin/sh
# Removes all HTML files supplied as argument which use <blink>

for file in "$@"
do
    # note use of -i to ignore case and -w to ignore white space
    # however tags containing newlines won't be detected
    if grep -Eiw '</?blink>' "$file" >/dev/null
    then
        echo  "Renaming $file to $file.bad because it uses the <blink> tag"
        mv "$file" "$file.bad"
    fi
done
```

8. Write a shell script, **list_include_files.sh**, which for all the C source files (`.c` files) in the current directory prints the names of the files they include (`.h` files), for example

```
$ list_include_files.sh
count_words.c includes:
    stdio.h
    stdlib.h
    ctype.h
    time.h
    get_word.h
    map.h
get_word.c includes:
    stdio.h
    stdlib.h
map.c includes:
    get_word.h
    stdio.h
    stdlib.h
    map.h
```

ANSWER:

```
#!/bin/sh
# list the files included by the C sources files included as arguments

for file in *.c
do
    echo "$file includes:"
    grep -E '^#include' "$file" | # find '#include lines
    sed 's/[">][^">]*$//' |       # remove the last '"' or '>' and anything after it
    sed 's/^.*["<]/    /'         # remove the first '"' or '>' and anything before it
done
```

9. The following shell script emulates the *cat* command using the built-in shell commands *read* and *echo*:

```
#!/bin/sh
while read line
do
    echo "$line"
done
```

a. What are the differences between the above script and the real *cat* command?

b. modify the script so that it can concatenate multiple files from the command line, like the real *cat*

(Hint: the shell's control structures — for example, `if`, `while`, `for` — are commands in their own right, and can form a component of a pipeline.)

ANSWER:

a. Some differences:

- the script doesn't concatenate files named on the command line, just standard input
- it doesn't implement all of the *cat* options
- the appearance of lines may be altered: space at start of line is removed, and runs of multiple spaces will be compressed to a single space.

b. A shell script to concatenate multiple files specified on command line:

```
#!/bin/sh

for f in "$@"
do
    if [ ! -r "$f" ]
    then
        echo "No such file: $f"
    else
        while read line
        do
            echo "$line"
        done <"$f"
    fi
done
```

10. The *gzip* command compresses a text file, and renames it to `filename.gz` . The *zcat* command takes the name of a single compressed file as its argument and writes the original (non-compressed) text to its standard output.

    Write a shell script called `zshow` that takes multiple `.gz` file names as its arguments, and displays the original text of each file, separated by the name of the file.

    Consider the following example execution of `zshow` :

```
$ zshow a.gz b.gz bad.gz c.gz
===== a =====
... original contents of file "a" ...
===== b =====
... original contents of file "b" ...
===== bad =====
No such file: bad.gz
===== c =====
... original contents of file "c" ...
```

    ANSWER:

    A simple solution which aims to make things obvious:

```
#!/bin/sh

for f in "$@"   # for each command line arg
do

    f1=$(echo "$f" | sed -e 's/\.gz$//')

    echo "===== $f1 ====="
    if test ! -r "$f"   # is the arg readable?
    then
        echo "No such file: $f"
    else
        zcat "$f"
    fi
done
```

    A solution that aims to be more robust

```
#!/bin/sh
for f in "$@"  # iterates over command line args
do

    f1=$(echo "$f" | sed -e 's/\.gz$//')

    echo "===== $f1 ====="
    if test ! -r "$f"   # is the arg readable?
    then
        echo "No such file: $f"
    else
        ftype=$(file -b "$f" | sed 's/ .*//')

        if [ "$ftype" != "gzip" ]
        then
            echo "Incorrect file type: $f"
        else
            zcat "$f"
        fi
    fi
done
```

    Notice that robustness typically adds a significant amount of code: the extra code is definitely worth it.

    The *file* command tells you what kind of file its argument is; the call to *sed* adjusts its output.

11. Consider the marks data file from last week's tutorial; assume it's stored in a file called `Marks` :

```
2111321 37 FL
2166258 67 CR
2168678 84 DN
2186565 77 DN
2190546 78 DN
2210109 50 PS
2223455 95 HD
2266365 55 PS
...
```

Assume also that we have a file called `Students` that contains the names and student ids of for all students in the class, e.g.

```
2166258 Chen, X
2186565 Davis, PA
2168678 Hussein, M
2223455 Jain, S
2190546 Phan, DN
2111321 Smith, JA
2266365 Smith, JD
2210109 Wong, QH
...
```

Write a shell script that produces a list of names and their associated marks, sorted by name:

```
67 Chen, X
77 Davis, PA
84 Hussein, M
95 Jain, S
78 Phan, DN
37 Smith, JA
55 Smith, JD
50 Wong, QH
```

Note: there are many ways to do this, generally involving combinations of filters such as *cut*, *grep*, *sort*, *join*, etc. Try to think of more than one solution, and discuss the merits of each.

ANSWER:

One obvious strategy: iterate over the `Students` file using the `read` builtin. We iterate over `Students` rather than `Marks`, as it is already in the order we want; we could iterate the other way, but then we would have to sort the output afterwards. For each student, we can use *grep* and *cut* (or *sed* or *awk* or Perl) to extract their information from the `Marks` file.

```
#!/bin/sh

while read zid name init
do
    mark=$(grep -E "$zid" Marks | cut -d' ' -f2)
    echo "$mark $name $init"
done <Students
```

For minimalists (and Haskell lovers), a one-line solution:

```
#!/bin/sh

sort Students |
join Marks - |
sort -k4 |
cut -d' ' -f2,4,5
```

Note the use of `-` to make the second argument to *join* come from standard input: without this mechanism, we would need to create a temporary file containing a sorted copy of `Students`.

12. Implement a shell script, `grades.sh`, that reads a sequence of (studentID, mark) pairs from its standard input, and writes (studentID, grade) pairs to its standard output. The input pairs are written on a single line, separated by spaces, and the output should use a similar format. The script should also check whether the second value on each line looks like a valid mark, and output an appropriate message if it does not The script can ignore any extra data occurring after the mark on each line.

    Consider the following input and corresponding output to the program:

    Input **Output**

```
2212345 65
2198765 74
2199999 48
2234567 50 ok
2265432 99
2121212 hello
2222111 120
2524232 -1
```

```
2212345 CR
2198765 CR
2199999 FL
2234567 PS
2265432 HD
2121212 ?? (hello)
2222111 ?? (120)
2524232 ?? (-1)
```

To get you started, here is a framework for the script:

```
#!/bin/sh
while read id mark
do
    # ... insert mark/grade checking here ...
done
```

Note that the `read` shell builtin assumes that the components on each input line are separated by spaces. How could we use this script if the data was supplied in a file that used commas to separate the (studentID, mark) components, rather than spaces?

ANSWER:

Since “mapping a mark to a grade” is a standard problem in first year tutes, working out the algorithm should not pose any problems. Hopefully the only tricky thing is getting the shell syntax right. The main aim of the exercise is to write a multi-way selection statement.

We supply two solutions: one using `if`, the other using `case`. The `if` one is more natural for people who know how to program in languages like Java. The `case` version requires us to develop patterns to match all the possible inputs.

The `case` construct is a nice way for checking strings via patterns. The pattern used here catches both non-numbers and negative numbers: they start with a minus rather than a digit. Unfortunately, the *test* command does not support pattern-matching.

Shell supports a C-style `continue` construct for loops, which is used here to prevent processing non-numeric "mark" fields.

All of the bracket-style (`[...]`) syntax for tests could be replaced by the more conventional syntax for the *test* command — e.g., `test $mark -lt 50`.

Note that the `read` statement has 2 arguments to ensure that the mark is bundled in with the optional comment on each data line.

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

while read -r id mark _; do
    echo -n "$id "

    if [ "$mark" -eq "$mark" ] 2> /dev/null && [ "$mark" -ge 0 ] && [ "$mark" -le 100 ]; then
        if   [ "$mark" -lt 50 ]; then
            echo FL
        elif [ "$mark" -lt 65 ]; then
            echo PS
        elif [ "$mark" -lt 75 ]; then
            echo CD
        elif [ "$mark" -lt 85 ]; then
            echo DN
        else
            echo HD
        fi
    else
        echo "?? ($mark)" >& 2
    fi
done
```

Another possibility would be to use `case` patterns to match the correct ranges of values, but this assumes that all marks are integer values. Floating point values could also be handled, but at the cost of making the patterns more complex. Also, this approach would not scale up to arbitrary ranges of integers; it would become to messy to specify patterns for all possible numbers.

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

while read -r id mark _; do
    echo -n "$id "

    if [ "$mark" -eq "$mark" ] 2> /dev/null && [ "$mark" -ge 0 ] && [ "$mark" -le 100 ]; then
        case "$mark" in
            [0-9]|[1-4][0-9])
                echo "FL"
                ;;
            5[0-9]|6[0-4])
                echo "PS"
                ;;
            6[5-9]|7[0-4])
                echo "CR"
                ;;
            7[5-9]|8[0-4])
                echo "DN"
                ;;
            *)
                echo "HD"
                ;;
        esac
    else
        echo "?? ($mark)"
    fi
done
```

If the input file used comma as a separator, the easiest thing would be to run the input through *tr* to convert the commas to spaces, and pipe the output into the `grades` program:

```
tr ',' ' ' <data | grades
```

Alternatively, you could alter the “input field separator” for the shell, by setting `IFS=,` .
