Week 10 Tutorial Sample Answers

1. How is the assignment going?

   Does anyone have hints or advice for other students?

   Has anyone discovered interesting cases that have to be handled?

   ANSWER:

   Discussed in tutorial.

2. Write a Python program, `until.py` which takes as an argument either a line number (e.g. `3`) or a regular expression (e.g. `/.[13579]/`) and prints all lines given to it by standard input until the given line number, or the first line matching the regular expression. For example:

```
$ seq 1 5 | ./until.py 3
1
2
3
```

```
$ seq 1 20 | ./until.py /.3/
1
2
3
4
5
6
7
8
9
10
11
12
13
```

   ANSWER:

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

from argparse import ArgumentParser
from re import compile
import sys

def main():
    # Overkill? yes.
    parser = ArgumentParser()
    parser.add_argument('address')
    args = parser.parse_args()

    try:
        # address is a line number
        address = int(args.address)
    except ValueError:
        # address is a regex (remove `/` from start and end)
        address = compile(args.address[1:-1])

    for line_number, line_content in enumerate(sys.stdin, start=1):
        # remove trailing newline if it exists
        if line_content[-1] == '\n':
            line_content = line_content[:-1]

        # address is an integer so check the line number
        if isinstance(address, int):
            if address == line_number:
                break
        # address is a regex so check the line content
        else:
            if address.search(line_content):
                break

        # if we aren't on a line that matches the address, print the line
        print(line_content)

    # once we've found a line that matches the address, print the line
    print(line_content)


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

3. Write a Python program that maps all lower-case vowels (a,e,i,o,u) in its standard input into their upper-case equivalents and, at the same time, maps all upper-case vowels (A, E, I, O, U) into their lower-case equivalents.
   The following shows an example input/output pair for this program:

Sample Input Data	Corresponding Output

`This is some boring text.`
`A little foolish perhaps?`
	`ThIs Is sOmE bOrIng tExt.`
	`a lIttlE fOOlIsh pErhAps?`

ANSWER:

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


import sys

VOWELS = "aeiou"

def main():
    trans = str.maketrans(VOWELS.upper() + VOWELS.lower(), VOWELS.lower() + VOWELS.upper())
    for line in sys.stdin:
        print(line.translate(trans), end="")


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

4. These commands both copy a directory tree to a CSE server.

```
$ scp -r directory1/ z1234567@login.cse.unsw.edu.au:directory2/
$ rsync -a directory1/ z1234567@login.cse.unsw.edu.au:directory2/
```

   What underlies them?

   How do they differ?

   Why are these differences important?

   ANSWER:

   Both command use `ssh` to connect to the remote machine.

   `rsync` will be more efficient if `directory2` already exists with similar contents. It will copy only files which are different and if only part of a file has changed it can copy only the changed bytes. This is is much more efficient when we are repeatedly making copies, e.g. for backups, particularly over a slow connection

   `rsync -a` also copies metadata such as file permissions and modification data which may be very important.

   `scp -rp` would copy metadata.

5. Assumes Linux kernel source tree containing thousand of source files can be found in `/usr/src/linux`

   Write a shell script that emails each of the 50,000 source (.c) files to Andrew, each as an attachment to a separate email.

   The source files may be anywhere in a directory tree than goes 10+ levels deep.

   Please don't run this script, and in general be very careful with such scripts. It is very embarassing to accidentally send thousands of emails.

   What assumptions does your script make?

   ANSWER:

```
#!/bin/sh

directory=/usr/src/linux

# depends on pathnames not containing white-space

for c_file in $(find "$directory" -type f -name '*.c')
do
    echo mutt -s "C for you"  -a "$c_file" -- andrewt@unsw.edu.au
done
```

6. The Perl programming language a handful of useful functions.
   Write a Python module `perl.py` that that contains the following functions:

   `chomp` - The Perl function `chomp` removes a single newline from the end of a string (if there is one).

   `qw` - The Perl function `qw` splits a string into a list of words.

   `die` - The Perl function `die` prints an error message and exits the program.

   ANSWER:

```
import sys

def chomp(string):
    if string[-1] == '\n':
        return string[:-1]
    else:
        return string

def qw(string):
    return string.split()

def die(message):
    print(sys.argv[0], "Error", message, sep=": ", file=sys.stderr)
    sys.exit(1)
```

Revision questions

The following questions are primarily intended for revision, either this week or later in session.
Your tutor may still choose to cover some of these questions, time permitting.

1. Write a *shell script* called `rmall.sh` that removes all of the files and directories below the directory supplied as its single command-line argument. The script should prompt the user with `Delete` *X* `?` before it starts deleting the contents of any directory *X*. If the user responds `yes` to the prompt, `rmall` should remove all of the plain files in the directory, and then check whether the contents of the subdirectories should be removed. The script should also check the validity of its command-line arguments.

ANSWER:

Sample solution

```
#!/bin/sh

# check whether there is a cmd line arg
case $# in
1) # ok ... requires exactly one arg
    ;;
*)
    echo "Usage: $0 dir"
    exit 1
esac

# then make sure that it is a directory
if test ! -d $1
then
    echo "$1 is not a directory"
    echo "Usage: $0 dir"
    exit 1
fi

# change into the specified directory
cd $1

# for each plain file in the directory
for f in .* *
do
    if test -f "$f"
    then
        rm $f
    fi
done

# for each subdirectory
for d in .* *
do
    if test -d "$d" -a "$d" != .  -a "$d" != ..
    then
        echo -n "Delete $d? "
        read answer
        if test "$answer" = "yes"
        then
             rmall "$d"
        fi
     fi
done
```

Alternative solution using case

```
#!/bin/sh

# check whether there is a cmd line arg
case $# in
1) # ok ... requires exactly one arg
    ;;
*)
    echo "Usage: $0 dir"
    exit 1
esac

# then make sure that it is a directory
if test ! -d $1
then
    echo "$1 is not a directory"
    echo "Usage: $0 dir"
    exit 1
fi

# change into the specified directory
cd $1

# for each plain file in the directory
for f in .* *
do
    case $f in
    .|..) # ignore . and ..
        ;;
    *)
        if test -f $f
        then
            rm $f
        fi
        ;;
    esac
done

# for each subdirectory
for d in .* *
do
    case $d in
    .|..) # ignore . and ..
        ;;
    *)
        if test -d $d
        then
            echo -n "Delete $d? "
            read answer
            if test "$answer" = "yes"
            then
                rmall $d
            fi
        fi
        ;;
    esac
done
```

2. Write a *shell script* called `check` that looks for duplicated student ids in a file of marks for a particular subject. The file consists of lines in the following format:

```
2233445 David Smith 80
2155443 Peter Smith 73
2244668 Anne Smith 98
2198765 Linda Smith 65
```

The output should be a list of student ids that occur 2+ times, separated by newlines. (i.e. any student id that occurs more than once should be displayed on a line by itself on the standard output).

ANSWER:

Sample solution

```
#!/bin/sh

cut -d' ' -f1 < Marks | sort | uniq -c | egrep -v '^ *1 ' | sed -e 's/^.* //'
```

Explanation:

1. `cut -d' ' -f1 < Marks` ... extracts the student ID from each line
2. `sort | uniq -c` ... sorts and counts the occurrences of each ID
3. IDs that occur once will be on a line that begins with spaces followed by `1` followed by a TAB
4. `grep -v '^ *1 '` removes such lines, leaving only IDs that occur multiple times
5. `sed -e 's/^.* //'` gets rid of the counts that `uniq -c` placed at the start of each line

3. Write a *Python script* **revline.py** that reverses the fields on each line of its standard input.

   Assume that the fields are separated by spaces, and that only one space is required between fields in the output.

   For example

```
$ ./revline.py
hi how are you
i'm great thank you
Ctrl-D
you are how hi
you thank great i'm
```

ANSWER:

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


import sys


def main():
    for line in sys.stdin:
        words = line.split()
        words = reversed(words)
        print(" ".join(words))

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

4. Which one of the following regular expressions would match a non-empty string consisting only of the letters `x`, `y` and `z`, in any order?
   a. `[xyz]+`
   b. `x+y+z+`
   c. `(xyz)*`
   d. `x*y*z*`

ANSWER:

a. Correct

b. Incorrect ... this matches strings like `xxx...yyy...zzz...`

c. Incorrect ... this matches strings like `xyzxyzxyz...`

d. Incorrect ... this matches strings like `xxx...yyy...zzz...`

The difference between (b) and (d) is that (b) requires there to be at least one of each `x`, `y` and `z`.

5. Which one of the following commands would extract the student id field from a file in the following format:

```
COMP3311;2122987;David Smith;95
COMP3231;2233445;John Smith;51
COMP3311;2233445;John Smith;76
```

a. `cut -f 2`

b. `cut -d; -f 2`

c. `sed -e 's/.*;//'`

d. None of the above.

ANSWER:

a. Incorrect ... this gives the entire data file; the default field-separator is tab, and since the lines contain no tabs, they are treated as a single large field; if an invalid field number is specified, `cut` simply prints the first

b. Incorrect ... this runs two separate commands `cut -d` followed by `-f 2`, and neither of them makes sense on its own

c. Incorrect ... this removes all chars up to and including the final semicolon in the line, and this gives the 4th field on each line

d. Correct

6. Write a Python program **frequencies.py** that prints a count of how often each letter ('a'..'z' and 'A'..'Z') and digit ('0'..'9') occurs in its input. Your program should follow the output format indicated in the examples below exactly.

   No count should be printed for letters or digits which do not occur in the input.

   The counts should be printed in dictionary order ('0'..'9','A'..'Z','a'..'z').

   Characters other than letters and digits should be ignored.

   The following shows an example input/output pair for this program:

```
$ ./frequencies.py
The  Mississippi is
1800 miles long!
Ctrl-D
'0' occurred 2 times
'1' occurred 1 times
'8' occurred 1 times
'M' occurred 1 times
'T' occurred 1 times
'e' occurred 2 times
'g' occurred 1 times
'h' occurred 1 times
'i' occurred 6 times
'l' occurred 2 times
'm' occurred 1 times
'n' occurred 1 times
'o' occurred 1 times
'p' occurred 2 times
's' occurred 6 times
```

ANSWER:

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

import fileinput, collections

freq = collections.Counter()

for line in fileinput.input():
    chars = list(line)
    for c in chars:
        if c.isalnum():
            freq[c] += 1

for f in sorted(freq):
    print(f"'{f}' occurred {freq[f]} times")
```

7. A "hill vector" is structured as an *ascent*, followed by an *apex*, followed by a *descent*, where

   - the *ascent* is a non-empty strictly ascending sequence that ends with the apex
   - the *apex* is the maximum value, and must occur only once

- the *descent* is a non-empty strictly descending sequence that starts with the apex

For example, [1,2,3,4,3,2,1] is a hill vector (with apex=4) and [2,4,6,8,5] is a hill vector (with apex=8). The following vectors are not hill vectors: [1,1,2,3,3,2,1] (not strictly ascending and multiple apexes), [1,2,3,4] (no descent), and [2,6,3,7,8,4] (not ascent then descent). No vector with less than three elements is considered to be a hill.

Write a Python program **hill_vector.py** that determines whether a sequence of numbers (integers) read from standard input forms a "hill vector". The program should write "hill" if the input *does* form a hill vector and write "not hill" otherwise.

Your program's input will only contain digits and white space. Any amount of whitespace may precede or follow integers.

Multiple integers may occur on the same line.

A line may contain no integers.

You can assume all the integers are positive. The following shows example input/output pairs for this program:

Sample Input Data	Corresponding Output
`1 2 4 8 5 3 2`	`hill`
`1 2`	`not hill`
`1 3 1`	`hill`

```
  3
1   1
```
	`not hill`

`2 4 6 8 10 10 9 7 5 3 1`	`not hill`

ANSWER:

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


import re, sys


lines = sys.stdin.readlines()
content = " ".join(lines).strip()
nums = [int(x) for x in re.findall('\d+', content)]

i = 0
while i < (len(nums) - 1) and nums[i] < nums[i + 1]:
    i += 1

j = len(nums) - 1
while j > 0 and nums[j] < nums[j - 1]:
    j -= 1

if i != j or i == 0 or j == (len(nums) - 1):
    print("not ", end='')

print("hill")
```

8. A list a_1, a_2, ... a_n is said to be **converging** if

```
a₁ > a₂ > ... > aₙ
```

and

```
for all i aᵢ ₋ ₁ − aᵢ > aᵢ − aᵢ ₊ ₁
```

In other words, the list is strictly decreasing and the difference between consecutive list elements always decreases as you go down the list.

Write a Python program **converging.py** that determines whether a sequence of positive integers read from standard input is converging. The program should write "converging" if the input is converging and write "not converging" otherwise. It should produce no other output.

Sample Input Data	Corresponding Output

`2010 6 4 3`	`converging`

`20`
`15`
`9`
	`not converging`

`1000`
`    100   10`
`    1`
	`converging`

`  6`
`  5`
`2 2`
	`not converging`

`  1 2 4 8`	`not converging`

Your program's input will only contain digits and white space. Any amount of whitespace may precede or follow integers.

Multiple integers may occur on the same line.

A line may contain no integers.

You can assume your input contains at least 2 integers.

You can assume all the integers are positive.

ANSWER:

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


import re, sys


lines = sys.stdin.readlines()
content = " ".join(lines).strip()
nums = [int(x) for x in re.findall('\d+', content)]

for i in range(len(nums) - 1):
    if (nums[i] <= nums[i + 1]):
        print("not converging")
        sys.exit(0)

for i in range(len(nums) - 2):
    if (nums[i] - nums[i + 1] <= nums[i + 1] - nums[i + 2]):
        print("not converging")
        sys.exit(0)

print("converging")
```

9. The *weight* of a number in a list is its value multiplied by how many times it occurs in the list. Consider the list `[1 6 4 7 3 4 6 3 3]`. The number 7 occurs once so it has weight 7. The number 3 occurs 3 times so it has weight 9. The number 4 occurs twice so it has weight 8.

   Write a Python program **heaviest.py** which takes 1 or more positive integers as arguments and prints the heaviest.

   Your Python program should print one integer and no other output.

   Your Python program can assume it it is given only positive integers as arguments

```
$ ./heaviest.py 1 6 4 7 3 4 6 3 3
6
$ ./heaviest.py 1 6 4 7 3 4 3 3
3
$ ./heaviest.py 1 6 4 7 3 4 3
4
$ ./heaviest.py 1 6 4 7 3 3
7
```

ANSWER:

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


import sys
from collections import Counter


weights = Counter()
for n in sys.argv[1:]:
    weights[n] += int(n)

print(weights.most_common(1)[0][0])
```