Week 07 Tutorial Sample Answers

1. In shell we have been using the following hashbang:

```
#!/bin/dash
```

How should we modify this hashbang to use it with python?

> ANSWER:
>
> In python we want to use the following hashbang:
>
> ```
> #!/usr/bin/env python3
> ```
>
> First: this calls the *python3* interpreter, not the *dash* interpreter.
>
> Second: It does so indirectly, by using the *env* utility.
>
> The *env* utility will search the PATH for the *python3* interpreter.
>
> This means that we don't need to know the location of the *python3* interpreter.
>
> Common locations for the *python3* interpreter are:
>
> - /bin/python3
> - /usr/bin/python3
> - /usr/local/bin/python3
>
> As we don't know the location of the *python3* interpreter, we can't use it directly.
>
> Aditionally, using *env* is basically a requirement when using a *virtual environment*.
>
> Which we will be doing in *later weeks*.

2. What version of python should be used in this course?

What are the differences between different versions of python?

> ANSWER:
>
> CSE currently has python3.9 installed.
>
> > NOTE:
> >
> > A lot of the time when referring to the currently installed version of python, we will simply use the term **python**.
> >
> > As this is a lot simpler than specifying the version every time.
>
> You can find the version of python that is installed by running:
>
> ```
> $ python3 --version
> Python 3.9.2
> ```
>
> Python is (theoretically) foward compatible.
>
> This means that you can (theoretically) run any python3.*X* (where *X* is ≤ 9) code using python3.9
>
> But it is recommended that you use python3.9
>
> You should not use python3.10 features in your python code for this course as it will break when autotesting and automarking.
>
> python2 and python3 are *different languages*.
>
> python2 reached its end of life on January 1, 2020.
>
> This means that python2 is unsupported.
>
> Is is also very old and has a lot of subtle differences to python3, so should generally not be used.
>
> CSE still has python2 installed and this is the default version of python.

```
$ python --version
Python 2.7.18
```

Make sure you include the **3** at the end to get python3.

Many other computers simply don't have a *python* command.

```
$ python --version
Command 'python' not found
```

3. Where can I find the *python3* documentation?

ANSWER:

For `C` and `shell` the best way to find documentation is with the *man* command.

Python is a much bigger language and doesn't have it's full documentation in *man*.

But is does have a *man* for it's command line arguments.

```
$ man python3
...
```

Python has inbuilt documentation accessable with the `help` function

```
$ python3
>>> help(print)
Help on built-in function print in module builtins:

print(...)
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
[...]
>>> import os
>>> help(os)
Help on module os:

NAME
os - OS routines for NT or Posix depending on what system we're on.
[...]
>>> help(os.path)
Help on module posixpath:

NAME
posixpath - Common operations on Posix pathnames.
[...]
```

You can also start interactive help. by using the `help` function without any arguments.

```
$ python3
>>> help()
help> open
Help on built-in function open in module io:

open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True,
opener=None)
Open file and return a stream. Raise OSError upon failure.
[...]
```

You can also start interactive help directly from the command line.

```
$ python3 -c 'help()'
```

The `help` is useful but not the easiest to read or navigate.

Python has full online documentation available at docs.python.org.

Remember to select the correct version of python when using the online documentation.

The most useful parts of the documentation are:

- Library Reference for common functions
- Language Reference for python syntax
- Module Index for things you can import
- Search for finding things

4. What is a REPL?

   How do you start the python REPL?

   ANSWER:

   REPL stands for *read-eval-print loop*.

   A REPL is a way to interact with a programming language.

   We have already seen the REPL in the terminal.

   A terminal is a shell REPL.

   As it:

   1. **read**s the command line
   2. **eval**uates the command by executing it
   3. **print**s the stdout and stderr of the command
   4. **loop**s

   Shell is built REPL first, I.e. the top priority of a shell is to provide the REPL and the Shell language is built around it.

   This is why there are so many strange design decisions in the shell language.

   The python REPL can be started by running:

   ```
   $ python3
   >>>
   ```

   in the python REPL you can run python code line by line

   ```
   >>> num = 5 * 9 + 8 ** 6 - 8 // 3
   >>> num += 96 ** (num // 30)
   ```

   Python will read, evaluate and print the result (if there is one (assigning a variable doesn't have a result so doesn't print anything)).

   A nice feature of the python REPL is that you can print a variable just by evaluating it.

   ```
   >>> num = 5 * 9 + 8 ** 6 - 8 // 3
   >>> num
   262187
   >>> print(num)
   262187
   ```

5. Write a simple version of the `head` command in Python, that accepts an optional command line argument in the form *–n*, where *n* is a number, and displays the first *n* lines from its standard input.

   If the *–n* option is not used, then the program simply displays the first ten lines from its standard input.

   ```
   # display first ten lines of file2
   $ ./head.py < file2
   # same as previous command
   $ ./head.py -10 < file2
   # display first five lines of file2
   $ ./head.py -5 < file2
   ```

   ANSWER:

   solution with for loop

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

import sys

n_lines = 10

if len(sys.argv) > 1 and sys.argv[1].startswith('-'):
    arg = sys.argv[1]
    arg = arg[1:]
    n_lines = int(arg)

n = 1
for line in sys.stdin:
    if n > n_lines: break
    sys.stdout.write(line)
    n += 1
```

solution reading all input into an array

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

import sys

n_lines = 10

if len(sys.argv) > 1 and sys.argv[1].startswith('-'):
    n_lines = int(sys.argv.pop(1)[1:])

# inefficient - reads entire file
sys.stdout.write("".join(sys.stdin.readlines()[0:n_lines]))
```

6. Modify the `head` program from the previous question so that, as well as handling an optional *–n* argument to specify how many lines, it also handles multiple files on the command line and displays the first *n* lines from each file, separating them by a line of the form `==>` *FileName* `<===` .

```
# display first ten lines of file1, file2, and file3
$ ./head.py file1 file2 file3
# display first three lines of file1, and file2
$ ./head.py -3 file1 file2
```

ANSWER:

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

import sys, itertools

n_lines = 10

if len(sys.argv) > 1 and sys.argv[1].startswith('-'):
    n_lines = int(sys.argv.pop(1)[1:])

if len(sys.argv) == 1:
    sys.argv.append("-")

for filename in sys.argv[1:]:
    try:
        print(f"==> {filename} <==")

        if filename == "-":
            stream = sys.stdin
        else:
            stream = open(filename)

        for line in itertools.islice(stream, n_lines):
            sys.stdout.write(line)

        if stream != sys.stdin:
            stream.close()

    except IOError as e:
```

```
        print(f"{sys.argv[0]}: can not open: {e.filename}: {e.strerror}")
```

7. The following is a Python version of the `cat` program.

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

import sys

if len(sys.argv) == 1:
    sys.argv.append("-")

for filename in sys.argv[1:]:
    try:
        if filename == "-":
            stream = sys.stdin
        else:
            stream = open(filename)

        for line in stream:
            sys.stdout.write(line)

        if stream != sys.stdin:
            stream.close()

    except IOError as e:
        print(f"{sys.argv[0]}: can not open: {e.filename}: {e.strerror}")
```

Write a new version of `cat` so that it accepts a `-n` command line argument and then prints a line number at the start of each line in a field of width 6, followed by two spaces, followed by the text of the line.

The numbers should constantly increase over all of the input files (i.e. don't start renumbering at the start of each file).

```
$ ./cat.py -n myFile
     1  This is the first line of my file
     2  This is the second line of my file
     3  This is the third line of my file
         ...
  1000  This is the thousandth line of my file
```

ANSWER:

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

from calendar import c
import sys

number = False

if len(sys.argv) > 1 and sys.argv[1].startswith('-'):
    arg = sys.argv.pop(1)
    arg = arg[1:]
    if arg == 'n':
        number = True

if len(sys.argv) == 1:
    sys.argv.append("-")

counter = 1
for filename in sys.argv[1:]:
    try:
        if filename == "-":
            stream = sys.stdin
        else:
            stream = open(filename)

        for line in stream:
            if number:
                sys.stdout.write(f"{counter:6}  {line}")
            else:
                sys.stdout.write(line)
            counter += 1

        if stream != sys.stdin:
            stream.close()

    except IOError as e:
        print(f"{sys.argv[0]}: can not open: {e.filename}: {e.strerror}")
```

using enumerate

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

from calendar import c
import sys

number = False

if len(sys.argv) > 1 and sys.argv[1].startswith('-'):
    arg = sys.argv.pop(1)
    arg = arg[1:]
    if arg == 'n':
        number = True

if len(sys.argv) == 1:
    sys.argv.append("-")

counter = 1
for filename in sys.argv[1:]:
    try:
        if filename == "-":
            stream = sys.stdin
        else:
            stream = open(filename)

        for counter, line in enumerate(stream, counter):
            if number:
                sys.stdout.write(f"{counter:6}  {line}")
            else:
                sys.stdout.write(line)

        counter += 1

        if stream != sys.stdin:
            stream.close()

    except IOError as e:
        print(f"{sys.argv[0]}: can not open: {e.filename}: {e.strerror}")
```

8. Modify the `cat` program from the previous question so that it also accepts a `-v` command line option to display *all* characters in the file in printable form.

   In particular, end of lines should be shown by a `$` symbol (useful for finding trailing whitespace in lines) and all control characters (ascii code less than 32) should be shown as `^X` (where *X* is the printable character obtained by adding the code for 'A' to the control character code). So, for example, tabs (ascii code 9) should display as `^I`.

```
$ ./cat -v myFile
This file contains a tabbed list:$
^I- point 1$
^I- point 2$
^I- point 3$
And this line has trailing spaces   $
which would otherwise be invisible.$
```

ANSWER:

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

import sys

number = False
verbose = False

while len(sys.argv) > 1 and sys.argv[1].startswith('-'):
    arg = sys.argv.pop(1)
    arg = arg[1:]
    if arg == 'n':
        number = True
    elif arg == 'v':
        verbose = True

if len(sys.argv) == 1:
    sys.argv.append("-")

counter = 1
for filename in sys.argv[1:]:
    try:
        if filename == "-":
            stream = sys.stdin
        else:
            stream = open(filename)

        for line in stream:

            if verbose:
                new_line = ""
                for char in line:
                    if ord(char) < 32 and ord(char) != 10:
                        new_line += "^" + chr(ord(char) + ord('A') - 1)
                    else:
                        new_line += char
                line = new_line.replace("\n", "$\n")

            if number:
                sys.stdout.write(f"{counter:6}  {line}")
            else:
                sys.stdout.write(line)
            counter += 1

        if stream != sys.stdin:
            stream.close()

    except IOError as e:
        print(f"{sys.argv[0]}: can not open: {e.filename}: {e.strerror}")
```

using str.translate()

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

import sys

number = False
verbose = False

while len(sys.argv) > 1 and sys.argv[1].startswith('-'):
    arg = sys.argv.pop(1)
    arg = arg[1:]
    if arg == 'n':
        number = True
    elif arg == 'v':
        verbose = True

if len(sys.argv) == 1:
    sys.argv.append("-")

counter = 1
for filename in sys.argv[1:]:
    try:
        if filename == "-":
            stream = sys.stdin
        else:
            stream = open(filename)

        for line in stream:

            if verbose:
                trans = str.maketrans({ i: "^" + chr(i + ord('A') - 1) for i in range(32) if i
!= 10 })
                line = line.translate(trans).replace('\n', '$\n')

            if number:
                sys.stdout.write(f"{counter:6}  {line}")
            else:
                sys.stdout.write(line)
            counter += 1

        if stream != sys.stdin:
            stream.close()

    except IOError as e:
        print(f"{sys.argv[0]}: can not open: {e.filename}: {e.strerror}")
```

using argparse

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

import sys
from argparse import ArgumentParser

from yaml import parse

number = False
verbose = False


parser = ArgumentParser()
parser.add_argument("-n", "--number", action="store_true", help="add line numbers to output")
parser.add_argument("-v", "--verbose", action="store_true", help="show control characters in
output")
parser.add_argument("files", nargs="*", default="-", help="files to concatenate")

args = parser.parse_args()

counter = 1
for filename in args.files:
    try:
        if filename == "-":
            stream = sys.stdin
        else:
            stream = open(filename)

        for line in stream:

            if args.verbose:
                trans = str.maketrans({ i: "^" + chr(i + ord('A') - 1) for i in range(32) if i
!= 10 })
                line = line.translate(trans).replace('\n', '$\n')

            if args.number:
                sys.stdout.write(f"{counter:6}  {line}")
            else:
                sys.stdout.write(line)
            counter += 1

        if stream != sys.stdin:
            stream.close()

    except IOError as e:
        print(f"{sys.argv[0]}: can not open: {e.filename}: {e.strerror}")
```

9. In Python, you can imitate a main function by using the `if __name__ == '__main__':` construct.

How does this work?

Why is this useful?

ANSWER:

Python is a scripting language.

Like most scripting languages, Python code is executed from the top down.

Unlike `C` where you have to write a main function.

The `if __name__ == '__main__':` construct is used to imitate a main function.

Usually, used like so:

```
def main():
    ...

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

This is a good way to make sure that your code is only executed when run as a script.

And not when imported as a module.

(more on that in later weeks)

How is works id that the `__name__` variable is set to the name of the current module.

If we are not in a module, then the `__name__` variable is set to `'__main__'`.

The most likely reason for this is that we are running the code as a script.

Or we are in the REPL

```
>>> __name__
'__main__'
```

Good style in python is to define a main function at the top of your script.

Then any other functions you want to use in your script can be defined below the main function.

And finally, you can call the main function at the end of your script using the `if __name__ == '__main__':` construct.

10. How can we use regular expressions in python?

ANSWER:

Python has a built-in regular expression library.

The `re` module.

To use the `re` module, we need to import it.

```
    import re
```

Or we can import the individual functions.

```
    from re import search, match, fullmatch
```

The 3 function are all used the same way:

```
    search(pattern, string [, flags])
```

where:
pattern is the regular expression pattern to search for
string is the string to search in
flags is an optional set of modifiers

```
    email = "cs2041@cse.unsw.edu.au"
    re.search(r'.+@.+\..+', email)
```

11. What is the difference between `search`, `match`, and `fullmatch`?

ANSWER:

`search` is the most like the *grep* command.

`search` will find a match anywhere in the string.

`match` will only find a match at the beginning of the string.

It is the same as using `search` with the `^` anchor.

`fullmatch` will only find a match at both the beginning and at the end of the string.

It is the same as using `search` with both the `^` and `$` anchors.

12. How are Python's regular expressions different from *grep*?

ANSWER:

Python's regular expressions are different from *grep* in a number of ways.

- When grep finds a match, it prints the line where the match was found.

```
$ grep '[aeiou]' <<EOF
a
b
c
d
e
f
g
EOF
a
e
```

When Python finds a match, it returns the match object.

```
>>> letters = """
... a
... b
... c
... d
... e
... f
... g
... """
>>> re.search(r'[aeiou]', letters)
<re.Match object; span=(1, 2), match='a'>
```

A match object has a number of useful attributes:

- `Match.span()` - the starting and ending index of the match
- `Match.re.pattern` - the regex pattern
- `Match.string` - the original string
- `Match.group(0)` - the match
- `Match.group(N)` - capture groups

- Grep finds all non-overlapping matches.

  Python finds only the first match.

- Grep works line by line

  Python works on the entire string.

- Grep can be given command-line options. like *grep -i*

  Python takes *flags* to do the same.

```
>>> re.search(r'[aeiou]', 'AbcdeFGhiJKLmnOp', re.IGNORECASE)
```