LAB1:

This script begins with a line containing the #! character combination, which is commonly called hash bang or shebang and continues with the path to the interpreter.

#!/usr/bin/env python3 uses the operating system env command, which locates and executes Python by searching the PATH environment variable. Unlike Windows, the Python interpreter is usually already in the $PATH variable on linux, so you don't have to add it.
Now that you understand what the script does, and the functions within it, let's run the Python file using the following command:

```
./health_checks.py
```

We got a permission denied error.

```
gcpstagingeduit1658_student@linux-instance:~/scripts$ ./health_checks.py
-bash: ./health_checks.py: Permission denied
gcpstagingeduit1658_student@linux-instance:~/scripts$
```

This is because the above command tries to run your script directly as a program. The program is parsed by the interpreter specified in the first line of the script, i.e. shebang. If the kernel finds that the first two bytes are #! it uses the rest of the line as an interpreter and passes the file as an argument. So, to do this, the file needs to have execute permission.

To run this file, we need it to have execute permission (x). Let's update the file permissions and then try running the file. Use the following command to add execute permission to the file:

```
sudo chmod +x health_checks.py
```

Use a nano editor to open the file health_checks.py.

```
nano health_checks.py
```

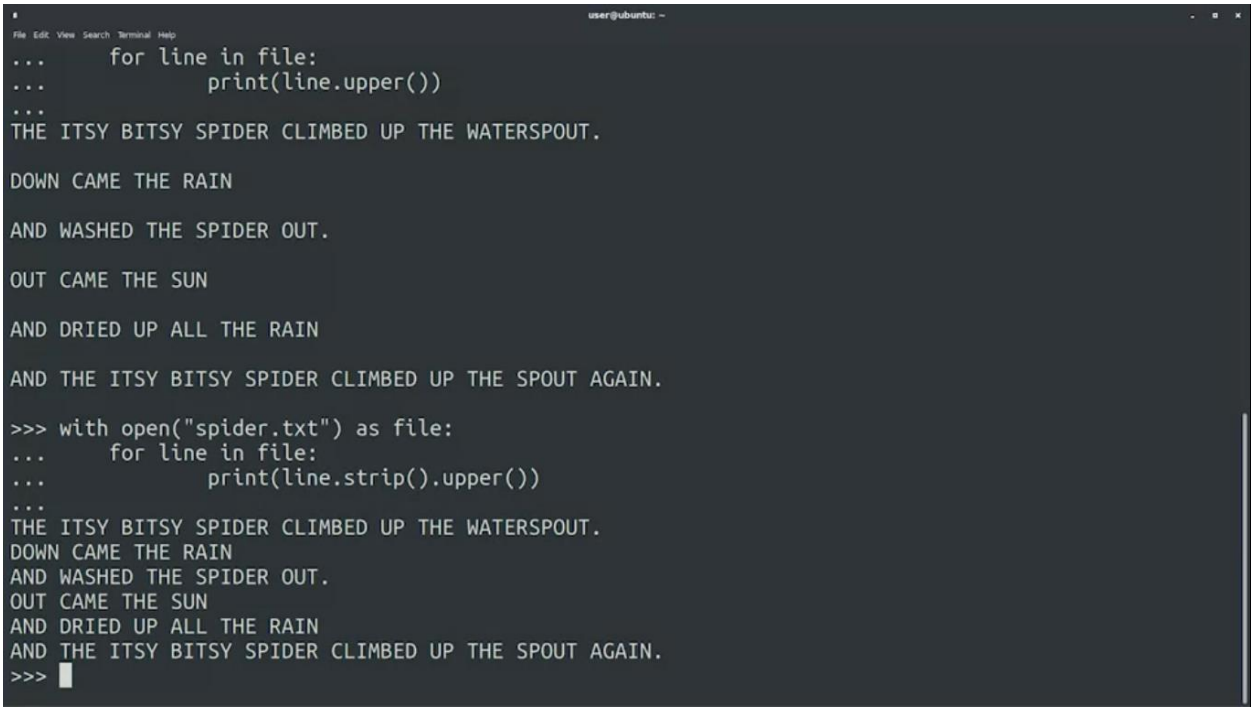
Make the necessary changes now. And once the changes are done, save the file by clicking Ctrl-o, enter key and Ctrl-x.

File Manipulation with Python:

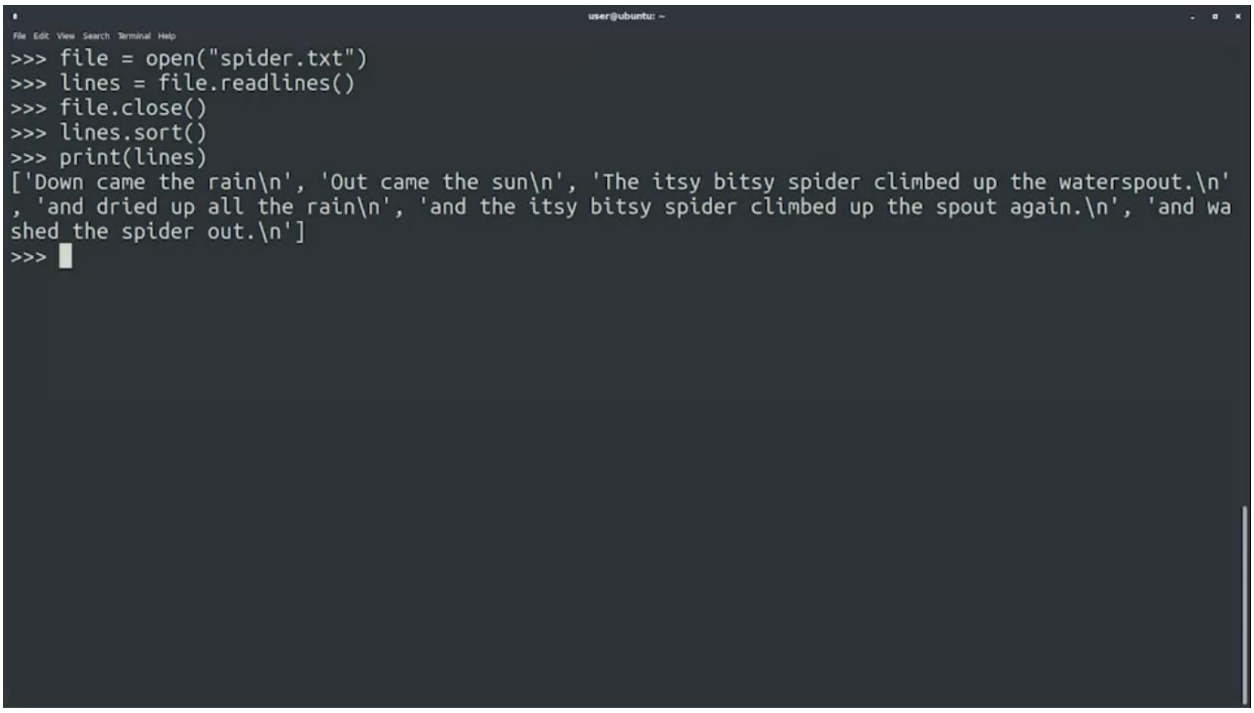
```
file=open(spider.txt)
print(file.readline())  //reads line by line
print(file.read()) //start reading from current position till end
file.close()
```

```
with open("spider.txt") as file:
    print(file.readline())
```

```
with open("spider.txt") as file:
    for line in file:
        print(line.upper())
        print(line.strip().upper()) //remove newlines after each line
```



Read File and put lines in a list:



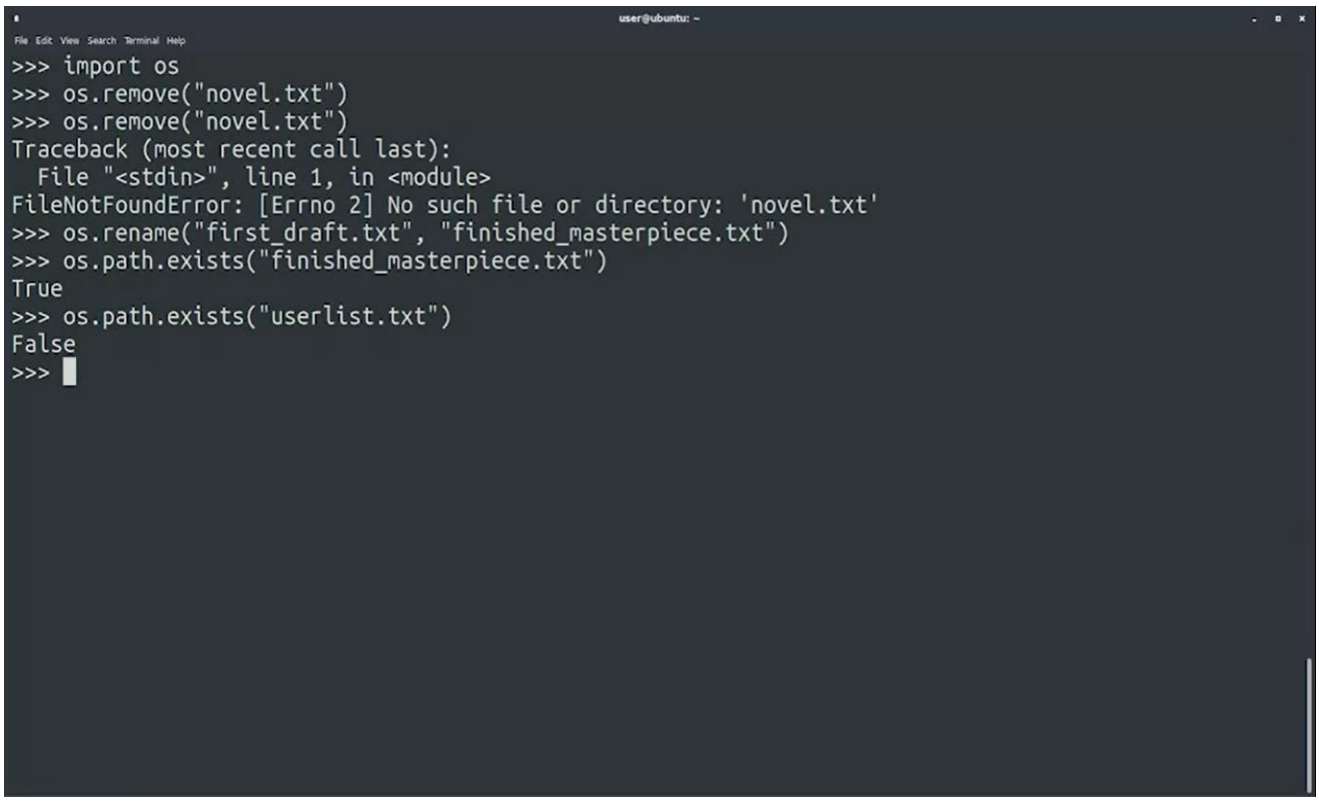
Writing Files:

Character	Meaning
`'r'`	open for reading (default)
`'w'`	open for writing, truncating the file first
`'x'`	open for exclusive creation, failing if the file already exists
`'a'`	open for writing, appending to the end of the file if it exists
`'b'`	binary mode
`'t'`	text mode (default)
`'+'`	open for updating (reading and writing)

```
guests = open("guests.txt", "w")
initial_guests = ["Bob", "Andrea", "Manuel", "Polly", "Khalid"]

for i in initial_guests:
    guests.write(i + "\n")
```

guests.close()

Working with Files



Directories:

```
>>> print(os.getcwd())
/home/user
>>> os.mkdir("new_dir")
>>> os.chdir("new_dir")
>>> os.getcwd()
'/home/user/new_dir'
>>> os.mkdir("newer_dir")
>>> os.rmdir("newer_dir")
>>>
```

List directories, check if file or directory:

```
user@ubuntu: ~
File Edit View Search Terminal Help
>>> import os
>>> os.listdir("website")
['images', 'index.html', 'favicon.ico']
>>> dir = "website"
>>> for name in os.listdir(dir):
...     fullname = os.path.join(dir, name)
...     if os.path.isdir(fullname):
...         print("{} is a directory".format(fullname))
...     else:
...         print("{} is a file".format(fullname))
...
website/images is a directory
website/index.html is a file
website/favicon.ico is a file
>>>
```

Practice Codes:

*** The create_python_script function creates a new python script in the current working directory, adds the line of comments to it declared by the 'comments' variable, and returns the size of the new file. Fill in the gaps to create a script called "program.py".

```
import os
def create_python_script(filename):
  comments = "# Start of a new Python program"
  file = open(filename,"w")
  file.write(comments)
  file.close()
  filesize = os.path.getsize(filename)
  return(filesize)

print(create_python_script("program.py"))
```

*** The new_directory function creates a new directory inside the current working directory, then creates a new empty file inside the new directory, and returns the list of files in that directory. Fill in the gaps to create a file "script.py" in the directory "PythonPrograms".

```
import os

def new_directory(directory, filename):
  # Before creating a new directory, check to see if it already exists
  if os.path.isdir(directory) == False:
    os.mkdir(directory)

  # Create the new file inside of the new directory
  os.chdir(directory)
  with open (filename,"w") as file:
    pass

  # Return the list of files in the new directory
  return os.listdir()

print(new_directory("PythonPrograms", "script.py"))
```

***The file_date function creates a new file in the current working directory, checks the date that the file was modified, and returns just the date portion of the timestamp in the format of yyyy-mm-dd. Fill in the gaps to create a file called "newfile.txt" and check the date that it was modified.

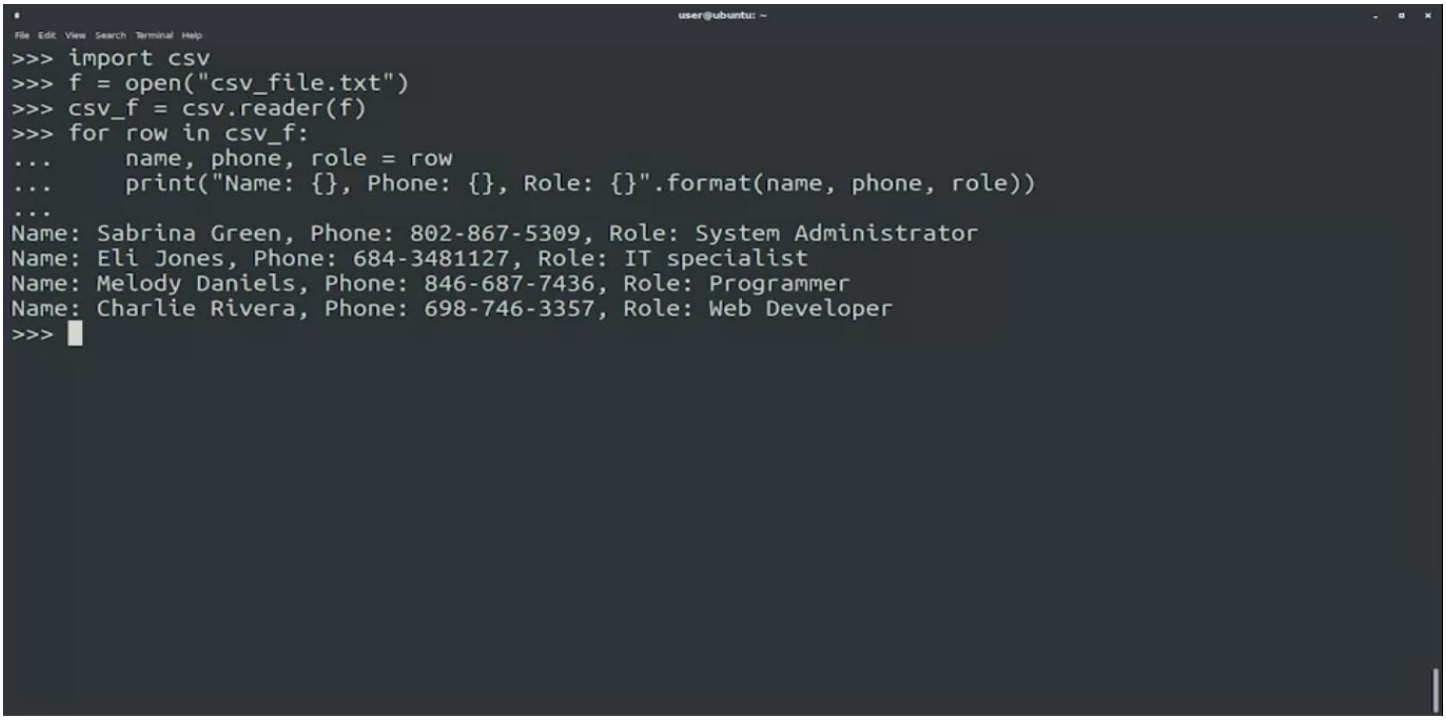
```
import os
import datetime

def file_date(filename):
  # Create the file in the current directory
  with open(filename, "w") as file:
    pass
  timestamp = os.path.getmtime(filename)
  # Convert the timestamp into a readable format, then into a string
  new_date = datetime.datetime.fromtimestamp(timestamp)
  # Return just the date portion
  # Hint: how many characters are in "yyyy-mm-dd"?
  return ("{}".format(new_date.date()))

print(file_date("newfile.txt"))
# Should be today's date in the format of yyyy-mm-dd
```

*** The parent_directory function returns the name of the directory that's located just above the current working directory. Remember that '..' is a relative path alias that means "go up to the parent directory". Fill in the gaps to complete this function.

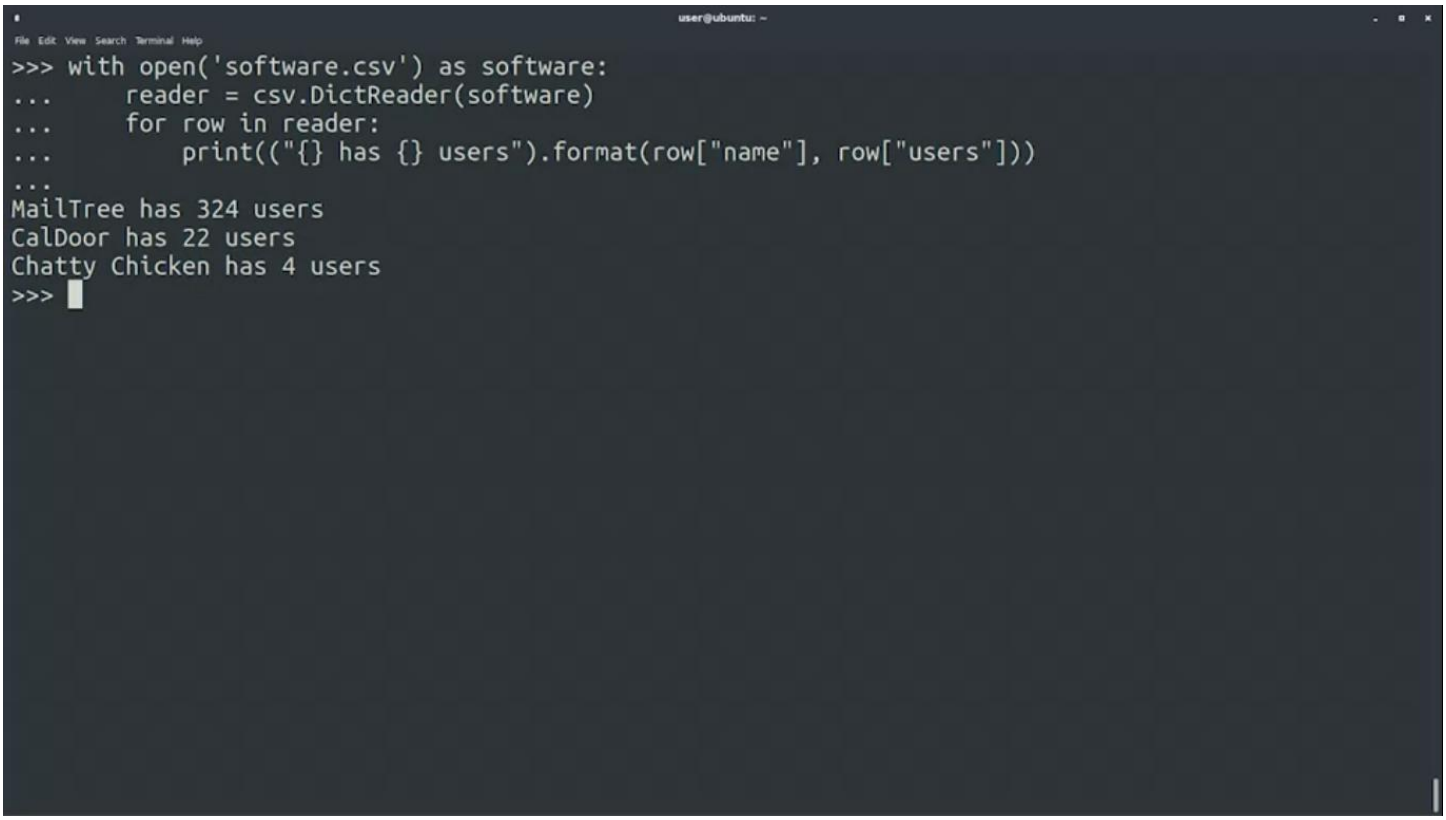
```
import os
def parent_directory():
  # Create a relative path to the parent
  # of the current working directory
  relative_parent = os.path.join('..',"w" )
```

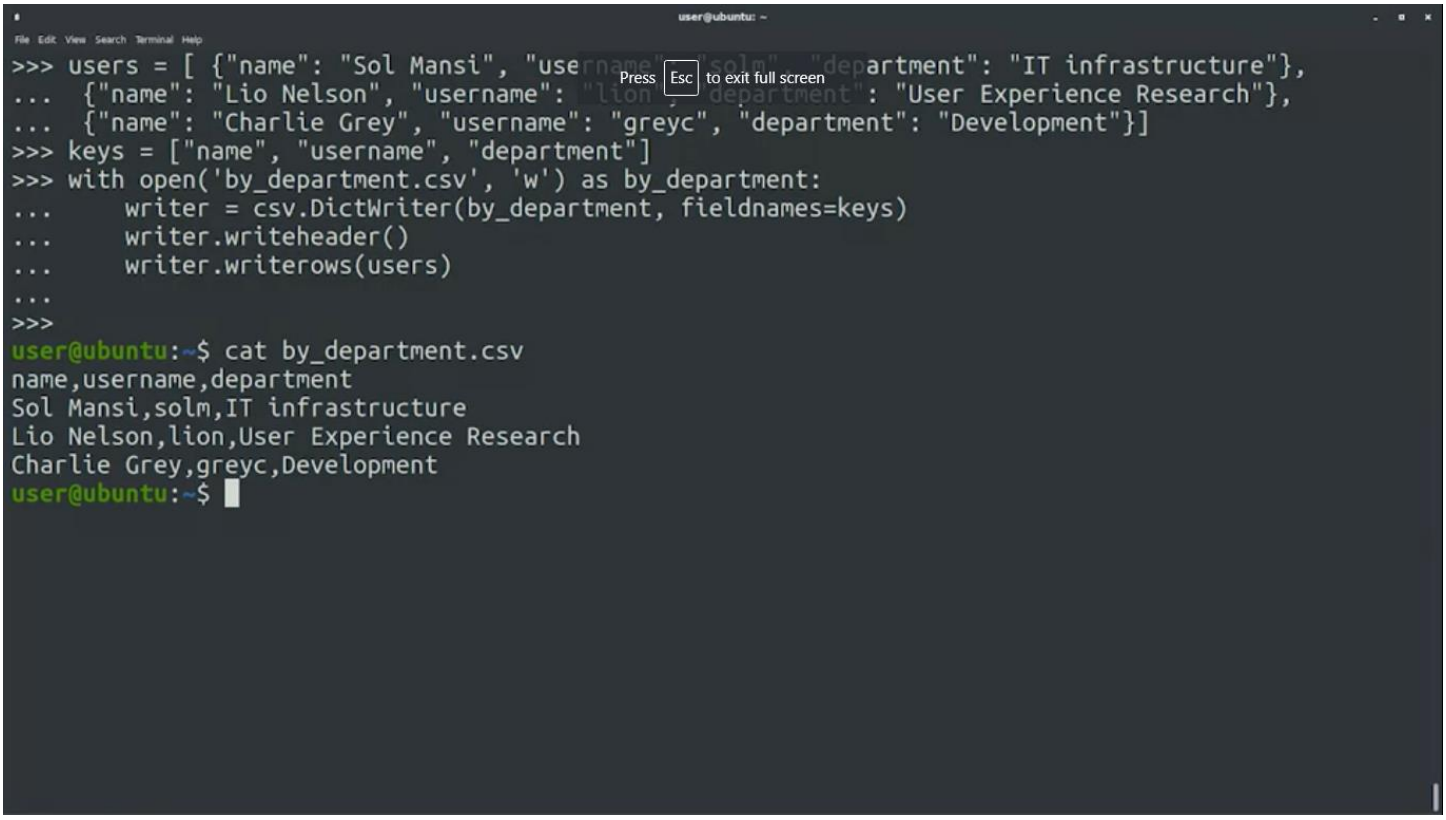
```
  # Return the absolute path of the parent directory
  return os.path.abspath('..')

print(parent_directory())
```

***Reading and Writing CSV Files



Reading and Writing CSV Files with Dictionaries:





Here's the `employee_birthday.txt` file:

CSV

```
name,department,birthday month
John Smith,Accounting,November
Erica Meyers,IT,March
```

Here's code to read it:

```
import csv

with open('employee_birthday.txt') as csv_file:
    csv_reader = csv.reader(csv_file, delimiter=',')
    line_count = 0
    for row in csv_reader:
        if line_count == 0:
            print(f'Column names are {", ".join(row)}')
            line_count += 1
        else:
            print(f'\t{row[0]} works in the {row[1]} department, and was born in {row[2]}.')
            line_count += 1
    print(f'Processed {line_count} lines.')
```

Here's the code to read it in as a dictionary this time:

```
import csv

with open('employee_birthday.txt', mode='r') as csv_file:
    csv_reader = csv.DictReader(csv_file)
    line_count = 0
    for row in csv_reader:
        if line_count == 0:
            print(f'Column names are {", ".join(row)}')
            line_count += 1
        print(f'\t{row["name"]} works in the {row["department"]} department, and was born in
{row["birthday month"]}.')
        line_count += 1
    print(f'Processed {line_count} lines.')
```

Writing CSV Files With csv

You can also write to a CSV file using a writer object and the .write_row() method:

```
import csv

with open('employee_file.csv', mode='w') as employee_file:
    employee_writer = csv.writer(employee_file, delimiter=',', quotechar='"',
quoting=csv.QUOTE_MINIMAL)

    employee_writer.writerow(['John Smith', 'Accounting', 'November'])
    employee_writer.writerow(['Erica Meyers', 'IT', 'March'])
```

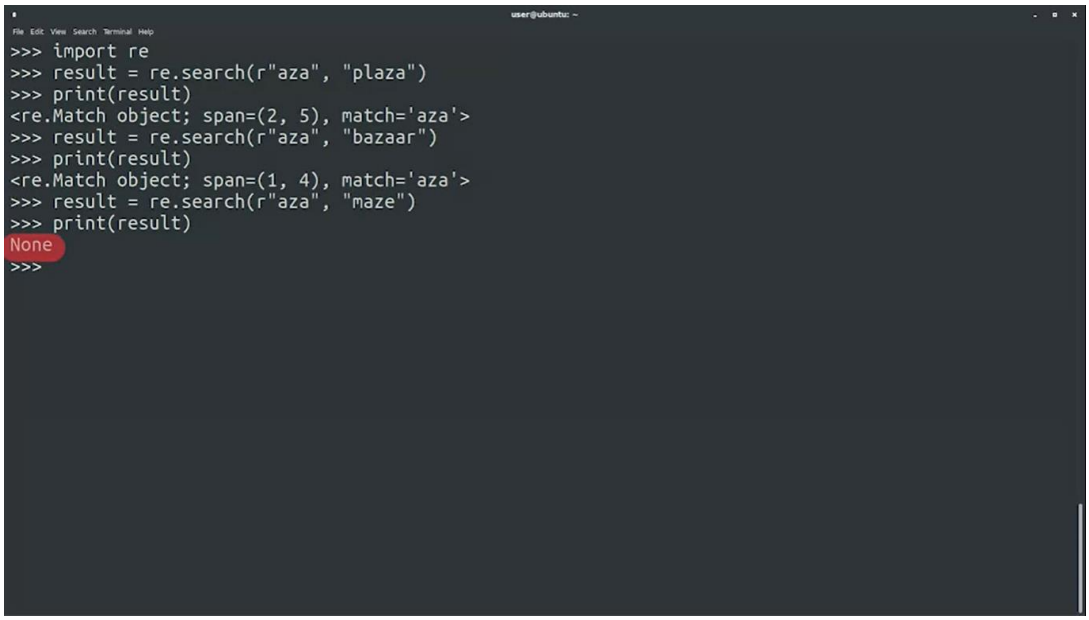
Writing CSV File From a Dictionary With `csv`

Since you can read our data into a dictionary, it's only fair that you should be able to write it out from a dictionary as well:

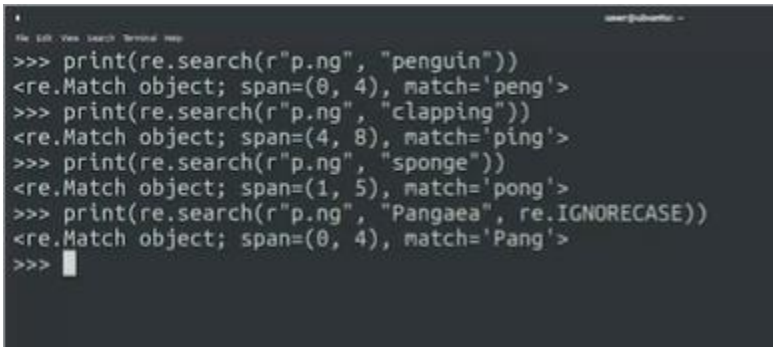
```
import csv

with open('employee_file2.csv', mode='w') as csv_file:
    fieldnames = ['emp_name', 'dept', 'birth_month']
    writer = csv.DictWriter(csv_file, fieldnames=fieldnames)

    writer.writeheader()
    writer.writerow({'emp_name': 'John Smith', 'dept': 'Accounting', 'birth_month': 'November'})
    writer.writerow({'emp_name': 'Erica Meyers', 'dept': 'IT', 'birth_month': 'March'})
```

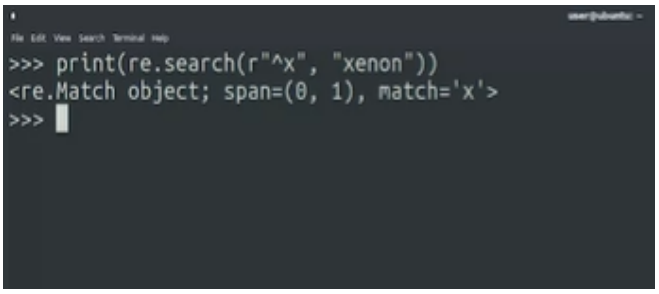
Regular Expression

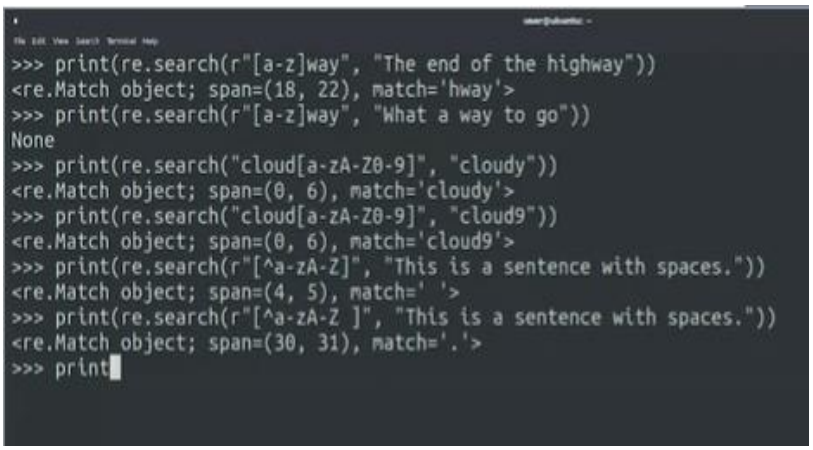
- The circumflex [^] and the dollar sign [$] are anchor characters. What do these anchor characters do in regex? Match the start and end of a line
- `grep word_to_search  /usr/home/Plabon/file`
- `grep -i word_to_search  /usr/home/Plabon/file //case insensitive`



****"r" indicates Raw String and tells python to not process any special characters.*







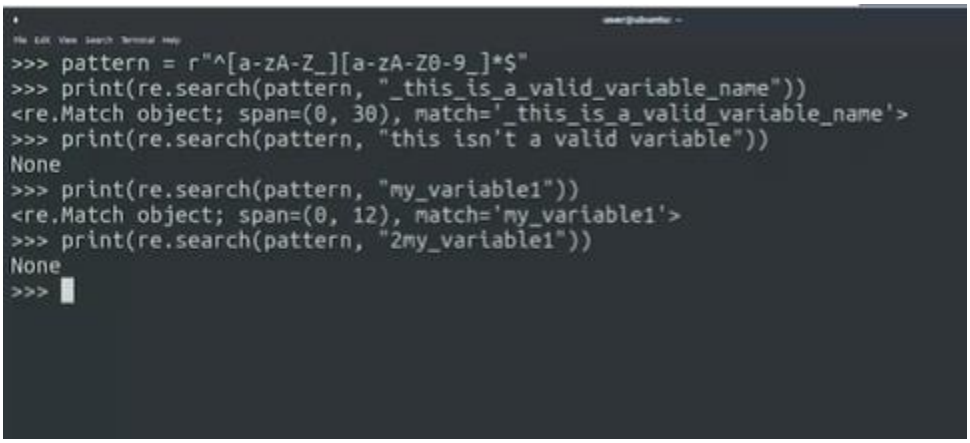
**inside square brackets are patterns.
** if we want to match something that is not inside the square bracket. A circumflex or caret (^) is used to denote that.
***square brace is called **Character Class**

*** **search()** function returns only the first match. If we want to find all the occurrences we can use the **findall()** function.

Regex for Python Variable:

Rules:

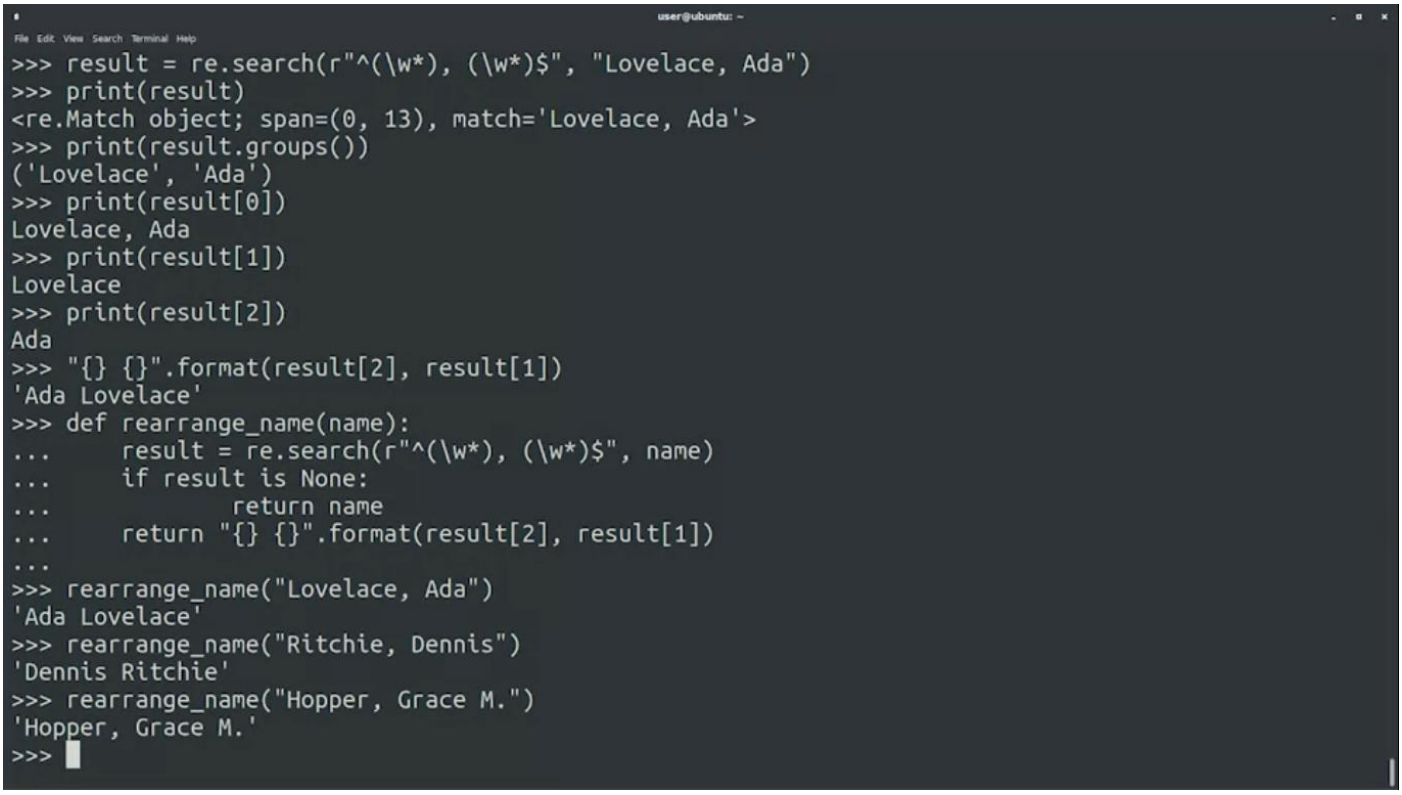
- variable name must start with letter or underscore (cannot start with number)
- variable name can contain only alphanumeric characters and underscore.
- Nothing else allowed
- (*) means the rule immediately before the star can occur 0 or more times.



https://regex101.com/
https://docs.python.org/3/howto/regex.html
https://docs.python.org/3/howto/regex.html#greedy-versus-non-greedy

Capturing Groups:

***(/w matches letters, numbers and underscore)



The correct regular expression for detecting middle names with dot and spaces should be:

```
"^([\w \.-]*), ([\w \.-]*)$"
```

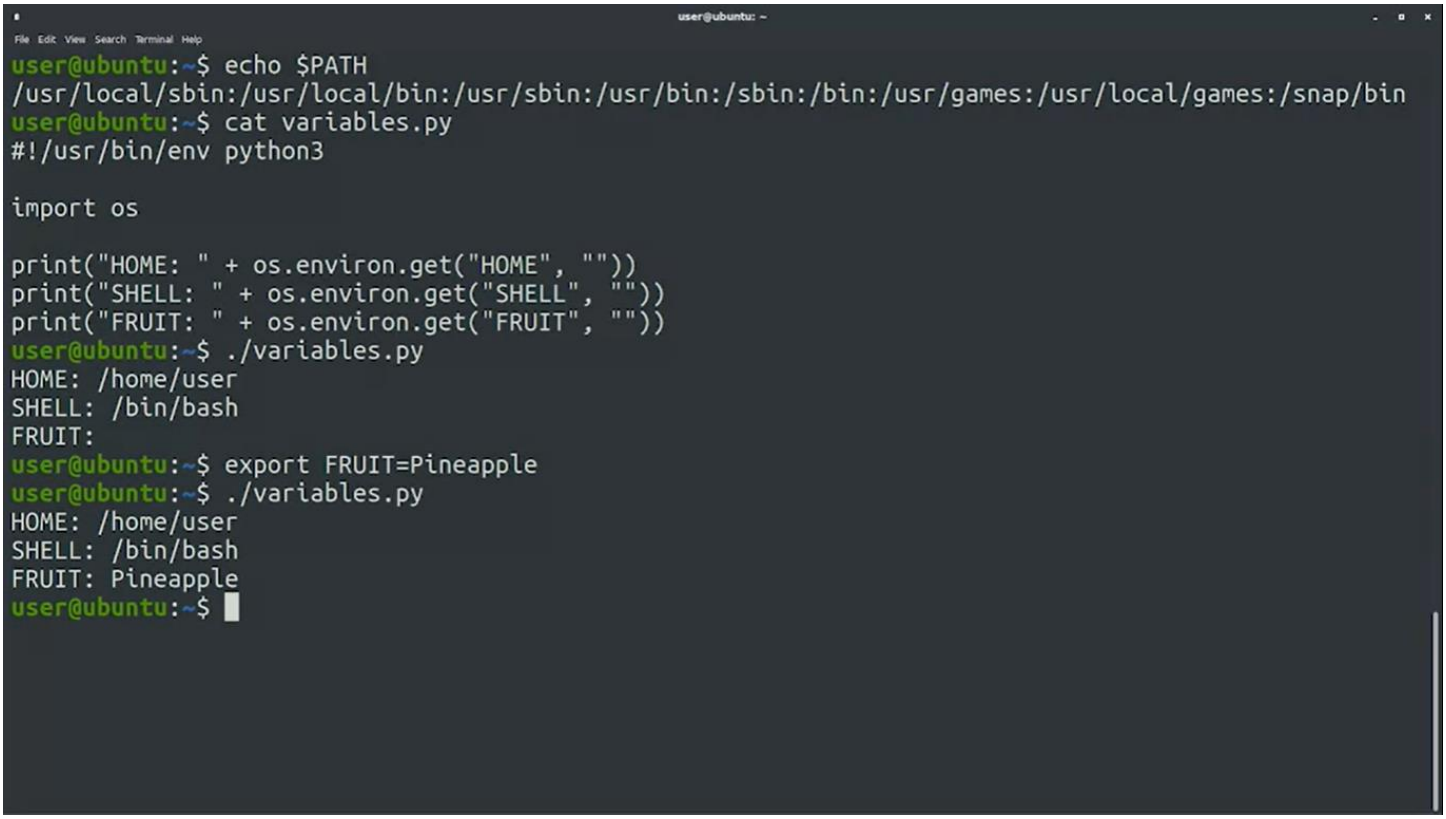
```
/************************************************

import re
def rearrange_name(name):
  result = re.search(r"^([\w \.-]*), ([\w \.-]*)$", name)
  if result == None:
    return name
  return "{} {}".format(result[2], result[1])

name=rearrange_name("Kennedy, John F.")
print(name)

/************************************************
```

Extracting PID from Log files

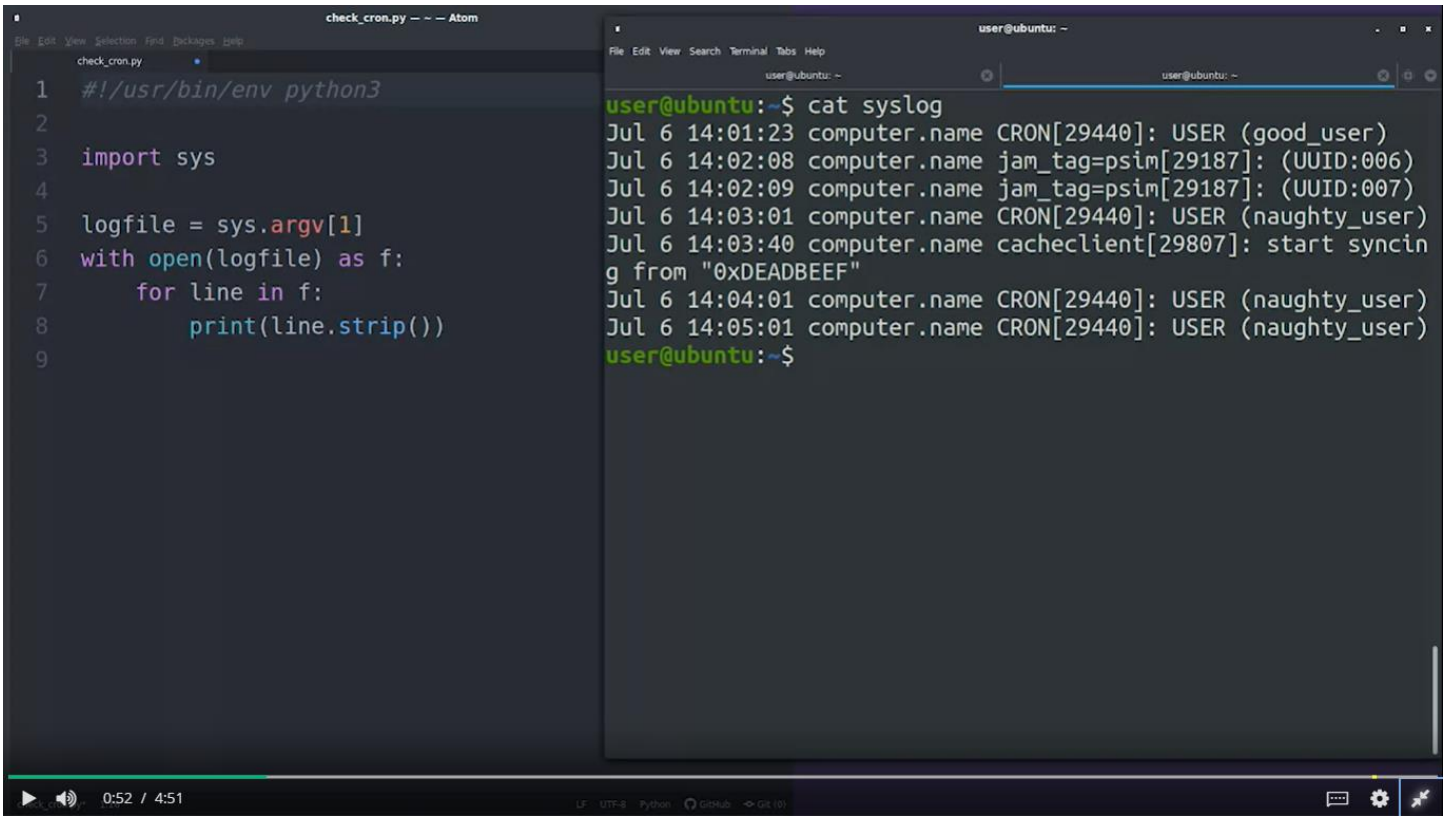
```
import re
def extract_pid(log_line):
    regex = r"\[(\d+)\]"
    result = re.search(regex, log_line)
    if result is None:
        return None
    return result[1]

print(extract_pid("July 31 07:51:48 mycomputer bad_process[12345]: ERROR Performing package
upgrade")) # 12345 (ERROR)
print(extract_pid("99 elephants in a [cage]")) # None
print(extract_pid("A string that also has numbers [34567] but no uppercase message")) # None
print(extract_pid("July 31 08:08:08 mycomputer new_process[67890]: RUNNING Performing backup")) #
67890 (RUNNING)
```

https://regexcrossword.com/

Environment Variables:



***os.environ returns dictionary. We could get the value in conventional way like **`os.environ["HOME"],`** but we would have got error if the key didn't exist. **`get()`** gives option to provide a default value if the key does not exist.



Running System Commands in Python

Use subprocess module
Run() function

- ⇨ Check videos of week-4

Unit Test

Unit Test Cheat-Sheet

Frankly, the unit testing library for Python is fairly well documented, but it can be a bit of a dry read. Instead, we suggest covering the core module concepts, and then reading in more detail later.
Best of Unit Testing Standard Library Module
Understand a Basic Example:

- https://docs.python.org/3/library/unittest.html#basic-example

Understand how to run the tests using the Command Line:

- https://docs.python.org/3/library/unittest.html#command-line-interface

Understand various Unit Test Design Patterns:

- https://docs.python.org/3/library/unittest.html#organizing-test-code
- Understand the uses of setUp, tearDown; setUpModule and tearDownModule
-

Understand basic assertions:

Method	Checks that	New in
assertEqual(a, b)	a == b	
assertNotEqual(a, b)	a != b	
assertTrue(x)	bool(x) is True	
assertFalse(x)	bool(x) is False	
assertIs(a, b)	a is b	3.1
assertIsNot(a, b)	a is not b	3.1
assertIsNone(x)	x is None	3.1
assertIsNotNone(x)	x is not None	3.1
assertIn(a, b)	a in b	3.1
assertNotIn(a, b)	a not in b	3.1
assertIsInstance(a, b)	isinstance(a, b)	3.2
assertNotIsInstance(a, b)	not isinstance(a, b)	3.2

Understand more specific assertions such as assertRaises

- https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertRaises

8.5. User-defined Exceptions

Programs may name their own exceptions by creating a new exception class (see Classes for more about Python classes). Exceptions should typically be derived from the `Exception` class, either directly or indirectly.

Exception classes can be defined which do anything any other class can do, but are usually kept simple, often only offering a number of attributes that allow information about the error to be extracted by handlers for the exception. When creating a module that can raise several distinct errors, a common practice is to create a base class for exceptions defined by that module, and subclass that to create specific exception classes for different error conditions:

```
class Error(Exception):
    """Base class for exceptions in this module."""
    pass

class InputError(Error):
    """Exception raised for errors in the input.

    Attributes:
        expression -- input expression in which the error occurred
        message -- explanation of the error
    """
```

```
    def __init__(self, expression, message):
        self.expression = expression
        self.message = message

class TransitionError(Error):
    """Raised when an operation attempts a state transition that's not
    allowed.

    Attributes:
        previous -- state at beginning of transition
        next -- attempted new state
        message -- explanation of why the specific transition is not allowed
    """

    def __init__(self, previous, next, message):
        self.previous = previous
        self.next = next
        self.message = message
```

Most exceptions are defined with names that end in “Error”, similar to the naming of the standard exceptions.

Many standard modules define their own exceptions to report errors that may occur in functions they define. More information on classes is presented in chapter Classes.

Python Exception Handling Techniques

https://doughellmann.com/blog/2009/06/19/python-exception-handling-techniques/

Error reporting and processing through exceptions is one of Python’s key features. Care must be taken when handling exceptions to ensure proper application cleanup while maintaining useful error reporting.

Error reporting and processing through exceptions is one of Python’s key features. Unlike C, where the common way to report errors is through function return values that then have to be checked on every invocation, in Python a programmer can raise an exception at any point in a program. When the exception is raised, program execution is interrupted as the interpreter searches back up the stack to find a context with an exception handler. This search algorithm allows error handling to be organized cleanly in a central or high-level place within the program structure. Libraries may not need to do any exception handling at all, and simple scripts can frequently get away with wrapping a portion of the main program in an exception handler to

print a nicely formatted error. Proper exception handling in more complicated situations can be a little tricky, though, especially in cases where the program has to clean up after itself as the exception propagates back up the stack.

Throwing and Catching

The statements used to deal with exceptions are `raise` and `except`. Both are language keywords. The most common form of throwing an exception with `raise` uses an instance of an exception class.

```
1  #!/usr/bin/env python
2
3  def throws():
4      raise RuntimeError('this is the error message')
5
6  def main():
7      throws()
8
9  if __name__ == '__main__':
10     main()
```

The arguments needed by the exception class vary, but usually include a message string to explain the problem encountered.

If the exception is left unhandled, the default behavior is for the interpreter to print a full traceback and the error message included in the exception.

```
$ python throwing.py
Traceback (most recent call last):
  File "throwing.py", line 10, in <module>
    main()
  File "throwing.py", line 7, in main
    throws()
  File "throwing.py", line 4, in throws
    raise RuntimeError('this is the error message')
RuntimeError: this is the error message
```

For some scripts this behavior is sufficient, but it is nicer to catch the exception and print a more user-friendly version of the error.

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

import sys

def throws():
    raise RuntimeError('this is the error message')

def main():
    try:
        throws()
        return 0
    except Exception, err:
        sys.stderr.write('ERROR: %sn' % str(err))
        return 1

if __name__ == '__main__':
    sys.exit(main())
```

In the example above, all exceptions derived from Exception are caught, and just the error message is printed to stderr. The program follows the Unix convention of returning an exit code indicating whether there was an error or not.

```
$ python catching.py


ERROR: this is the error message
```

Logging Exceptions

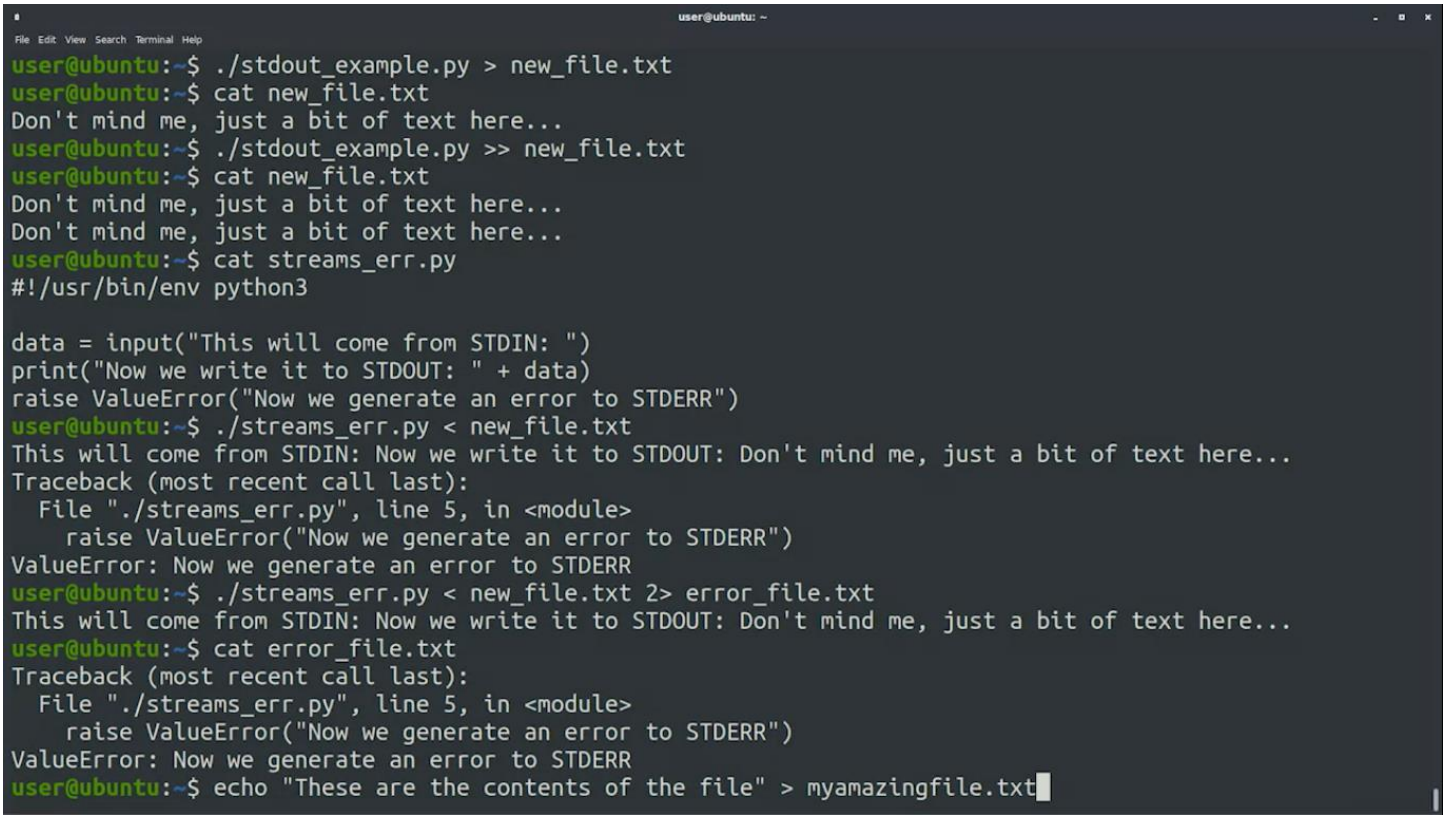
For daemons or other background processes, printing directly to stderr may not be an option. The file descriptor might have been closed, or it may be redirected somewhere that errors are hard to find. A better option is to use the logging module to log the error, including the full traceback.

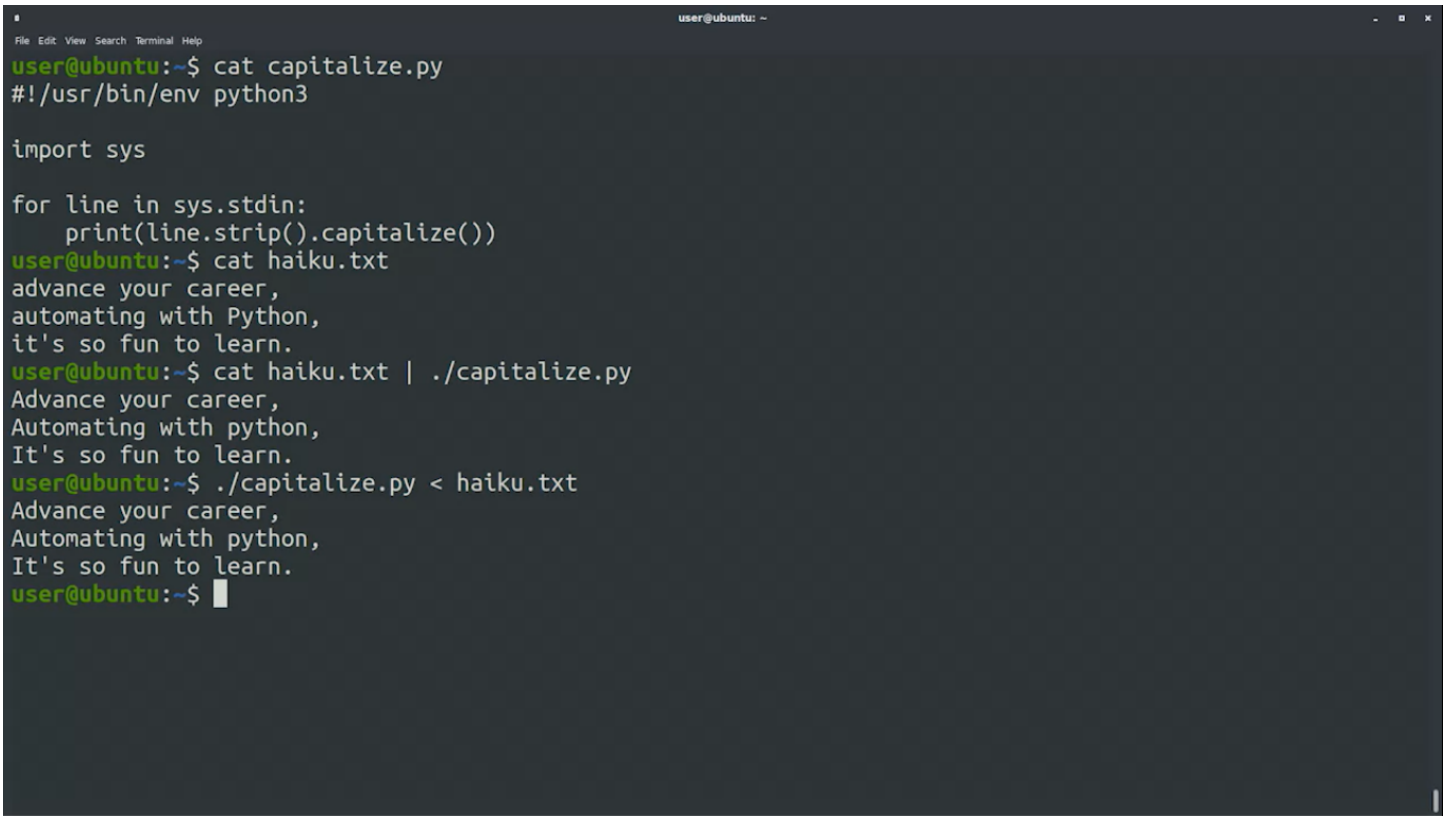
```
#!/usr/bin/env python

import logging
import sys

def throws():
    raise RuntimeError('this is the error message')

def main():
    logging.basicConfig(level=logging.WARNING)
    log = logging.getLogger('example')
    try:
        throws()
        return 0
    except Exception, err:
        log.exception('Error from throws():')
        return 1

if __name__ == '__main__':
    sys.exit(main())
```

In this example, the logger is configured to to use the default behavior of sending its output to stderr, but that can easily be adjusted. Saving tracebacks to a log file can make it easier to debug problems that are otherwise hard to reproduce outside of a production environment.

```
$ python logging_errors.py
ERROR:example:Error from throws():
Traceback (most recent call last):
  File "logging_errors.py", line 13, in main
    throws()
  File "logging_errors.py", line 7, in throws
    raise RuntimeError('this is the error message')
RuntimeError: this is the error message
```

Bash Scripting:





Redirections, Pipes and Signals

Managing streams

These are the redirectors that we can use to take control of the streams of our programs

- command **>** file: redirects standard output, overwrites file
- command **>>** file: redirects standard output, appends to file
- command **<** file: redirects standard input from file
- command **2>** file: redirects standard error to file
- command1 **|** command2: connects the output of command1 to the input of command2

Operating with processes

These are some commands that are useful to know in Linux when interacting with processes. Not all of them are explained in videos, so feel free to investigate them on your own.

- **ps:** lists the processes executing in the current terminal for the current user
- **ps** ax: lists all processes currently executing for all users
- **ps** e: shows the environment for the processes listed

- **kill** PID: sends the SIGTERM signal to the process identified by PID
- **fg**: causes a job that was stopped or in the background to return to the foreground
- **bg:** causes a job that was stopped to go to the background
- **jobs:** lists the jobs currently running or stopped
- **top:** shows the processes currently using the most CPU time (press "q" to quit)