

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:

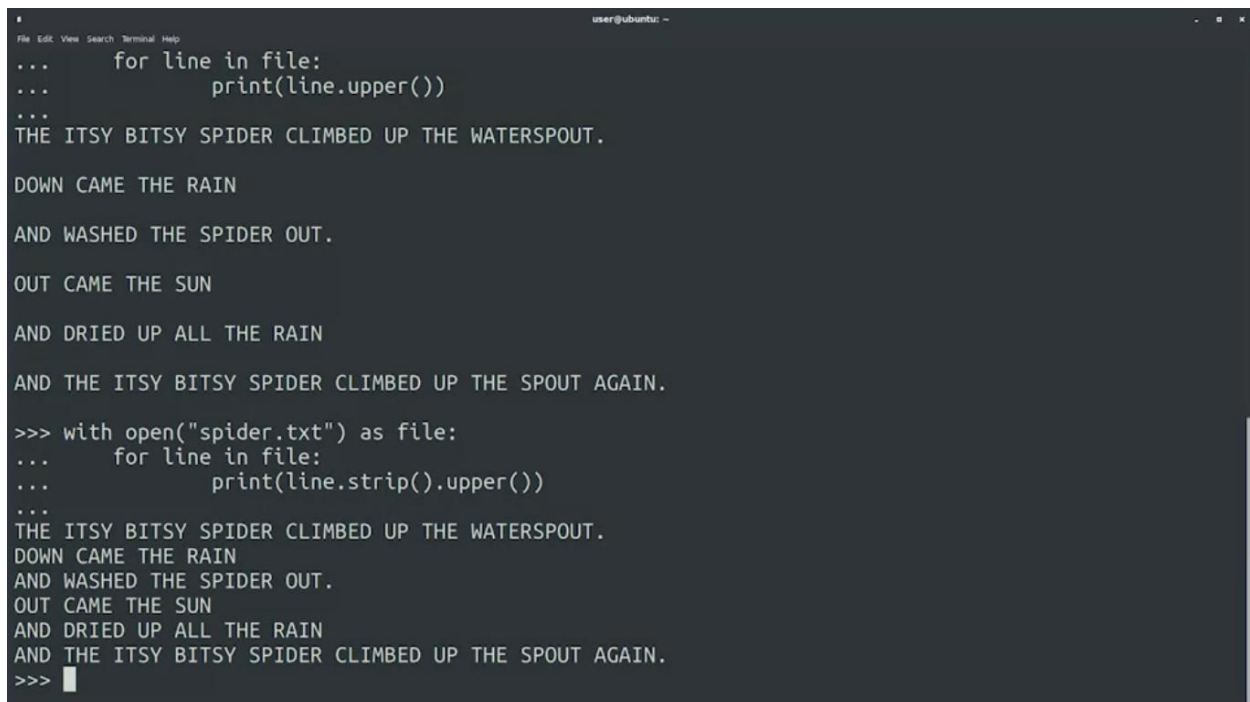
```
file=open(spider.txt)
print(file.readline()) //reads line by line
print(file.read()) //start reading from current position till end
file.close()
```

.....

```
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
```

A screenshot of a terminal window with a dark background. The window title is "user@ubuntu: ~". It shows the execution of Python code to read a file named "spider.txt". The first part of the code uses a for loop to print each line in uppercase. The second part uses a with statement to do the same, but also strips the newline characters. The output of the first code block shows the poem "The Itsy Bitsy Spider" with newlines. The output of the second code block shows the same poem but with all newlines removed.

```
File Edit View Search Terminal Help
...     for line in file:
...         print(line.upper())
...
THE ITSY BITSY SPIDER CLIMBED UP THE WATERSPOUT.

DOWN CAME THE RAIN

AND WASHED THE SPIDER OUT.

OUT CAME THE SUN

AND DRIED UP ALL THE RAIN

AND THE ITSY BITSY SPIDER CLIMBED UP THE SPOUT AGAIN.

>>> with open("spider.txt") as file:
...     for line in file:
...         print(line.strip().upper())
...
THE ITSY BITSY SPIDER CLIMBED UP THE WATERSPOUT.
DOWN CAME THE RAIN
AND WASHED THE SPIDER OUT.
OUT CAME THE SUN
AND DRIED UP ALL THE RAIN
AND THE ITSY BITSY SPIDER CLIMBED UP THE SPOUT AGAIN.
>>> █
```

Read File and put lines in a list:

```
user@ubuntu: ~  
File Edit View Search Terminal Help  
>>> file = open("spider.txt")  
>>> lines = file.readlines()  
>>> file.close()  
>>> lines.sort()  
>>> print(lines)  
['Down came the rain\n', 'Out came the sun\n', 'The itsy bitsy spider climbed up the waterspout.\n',  
, 'and dried up all the rain\n', 'and the itsy bitsy spider climbed up the spout again.\n', 'and wa  
shed the spider out.\n']  
>>> █
```

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

```
user@ubuntu: ~  
File Edit View Search Terminal Help  
>>> import os  
>>> os.remove("novel.txt")  
>>> os.remove("novel.txt")  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
FileNotFoundError: [Errno 2] No such file or directory: 'novel.txt'  
>>> os.rename("first_draft.txt", "finished_masterpiece.txt")  
>>> os.path.exists("finished_masterpiece.txt")  
True  
>>> os.path.exists("userlist.txt")  
False  
>>> █
```

```
user@ubuntu: ~  
File Edit View Search Terminal Help  
>>> os.path.getsize("spider.txt")  
192  
>>> os.path.getmtime("spider.txt")  
1578322923.89999994  
>>> import datetime  
>>> timestamp = os.path.getmtime("spider.txt")  
>>> datetime.datetime.fromtimestamp(timestamp)  
datetime.datetime(2020, 1, 6, 7, 2, 3, 899999)  
>>> os.path.abspath("spider.txt")  
'/home/user/spider.txt'  
>>> █
```

Directories:

```
File Edit View Search Terminal Help
>>> 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:

```
File Edit View Search Terminal Help user@ubuntu: ~
>>> 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
```

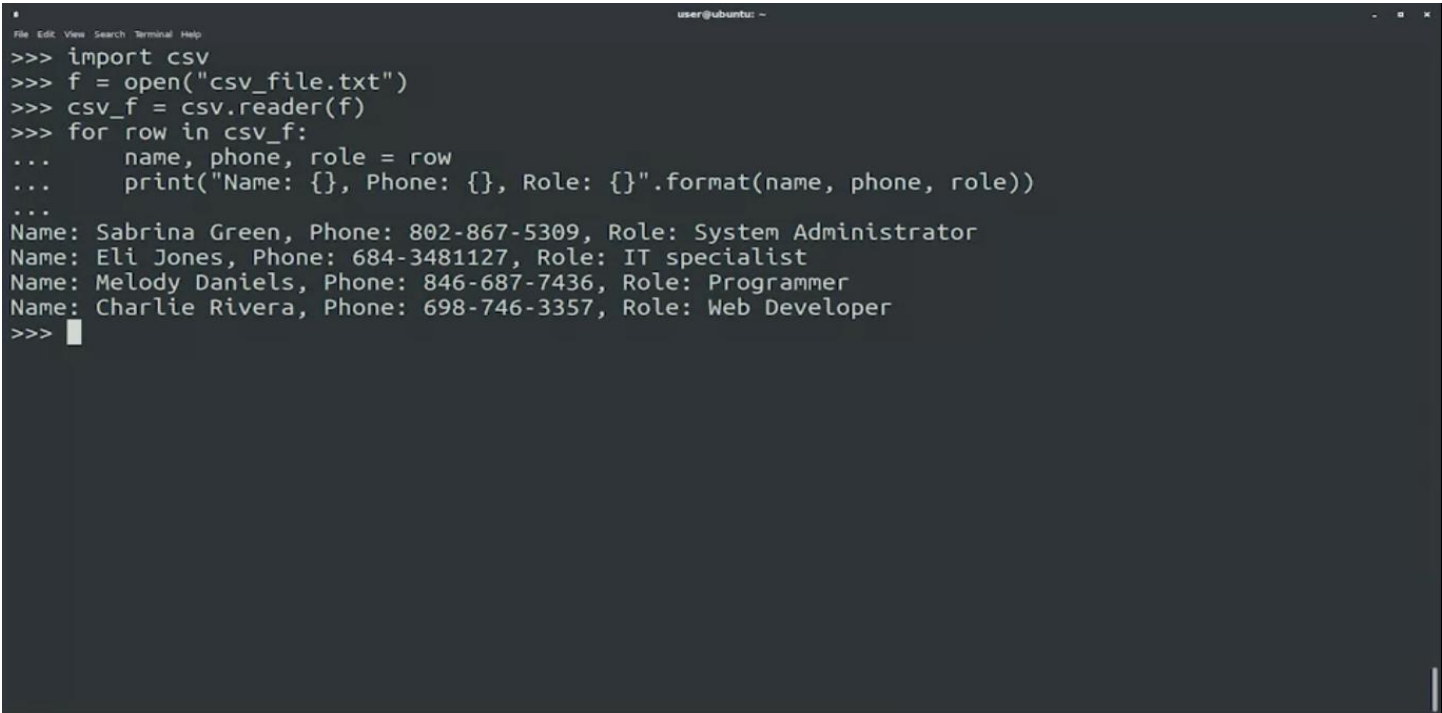
*** The parent_directory function returns the name of the directory that's located just above the current working directory. Remember that '..' is a relative path alias that means "go up to the parent directory". Fill in the gaps to complete this function.

```
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

A terminal window titled 'user@ubuntu: ~' showing a Python script that reads a CSV file named 'csv_file.txt'. The script imports the 'csv' module, opens the file, creates a 'csv.reader' object, and iterates over each row. For each row, it prints the 'name', 'phone', and 'role' fields. The output shows four rows of data: Sabrina Green (System Administrator), Eli Jones (IT specialist), Melody Daniels (Programmer), and Charlie Rivera (Web Developer).

```
File Edit View Search Terminal Help
user@ubuntu: ~
>>> import csv
>>> f = open("csv_file.txt")
>>> csv_f = csv.reader(f)
>>> for row in csv_f:
...     name, phone, role = row
...     print("Name: {}, Phone: {}, Role: {}".format(name, phone, role))
...
Name: Sabrina Green, Phone: 802-867-5309, Role: System Administrator
Name: Eli Jones, Phone: 684-3481127, Role: IT specialist
Name: Melody Daniels, Phone: 846-687-7436, Role: Programmer
Name: Charlie Rivera, Phone: 698-746-3357, Role: Web Developer
>>> █
```

A terminal window titled 'user@ubuntu: ~' showing a Python script that writes a CSV file named 'hosts.csv'. The script creates a list of hostnames and IP addresses, opens the file in write mode, creates a 'csv.writer' object, and uses 'writer.writerow()' to write each row. After the script, the user runs 'cat hosts.csv' to display the contents of the file, which shows two lines: 'workstation.local,192.168.25.46' and 'webserver.cloud,10.2.5.6'.

```
File Edit View Search Terminal Help
user@ubuntu: ~
>>> hosts = [ ["workstation.local", "192.168.25.46"], ["webserver.cloud", "10.2.5.6"] ]
>>> with open('hosts.csv', 'w') as hosts_csv:
...     writer = csv.writer(hosts_csv)
...     writer.writerow(hosts)
...
>>>
user@ubuntu:~$ cat hosts.csv
workstation.local,192.168.25.46
webserver.cloud,10.2.5.6
user@ubuntu:~$ █
```

Reading and Writing CSV Files with Dictionaries:

```
user@ubuntu: ~  
File Edit View Search Terminal Help  
>>> with open('software.csv') as software:  
...     reader = csv.DictReader(software)  
...     for row in reader:  
...         print("{} has {} users".format(row["name"], row["users"]))  
...  
MailTree has 324 users  
CalDoor has 22 users  
Chatty Chicken has 4 users  
>>> █
```

```
user@ubuntu: ~  
File Edit View Search Terminal Help  
>>> users = [ {"name": "Sol Mansi", "username": "solm", "department": "IT infrastructure"},  
... {"name": "Lio Nelson", "username": "lion", "department": "User Experience Research"},  
... {"name": "Charlie Grey", "username": "greyc", "department": "Development"}]  
>>> keys = ["name", "username", "department"]  
>>> with open('by_department.csv', 'w') as by_department:  
...     writer = csv.DictWriter(by_department, fieldnames=keys)  
...     writer.writeheader()  
...     writer.writerows(users)  
...  
>>>  
user@ubuntu:~$ cat by_department.csv  
name,username,department  
Sol Mansi,solm,IT infrastructure  
Lio Nelson,lion,User Experience Research  
Charlie Grey,greyc,Development  
user@ubuntu:~$ █
```


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`

```
user@ubuntu: ~$ python3
>>> import re
>>> result = re.search(r"aza", "plaza")
>>> print(result)
<re.Match object; span=(2, 5), match='aza'>
>>> result = re.search(r"aza", "bazaar")
>>> print(result)
<re.Match object; span=(1, 4), match='aza'>
>>> result = re.search(r"aza", "maze")
>>> print(result)
None
>>>
```

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

```
user@ubuntu: ~$ python3
>>> print(re.search(r"p.ng", "penguin"))
<re.Match object; span=(0, 4), match='peng'>
>>> print(re.search(r"p.ng", "clapping"))
<re.Match object; span=(4, 8), match='ping'>
>>> print(re.search(r"p.ng", "sponge"))
<re.Match object; span=(1, 5), match='pong'>
>>> print(re.search(r"p.ng", "Pangaea", re.IGNORECASE))
<re.Match object; span=(0, 4), match='Pang'>
>>>
```

```
user@ubuntu: ~$ python3
>>> print(re.search(r"^x", "xenon"))
<re.Match object; span=(0, 1), match='x'>
>>>
```

```
user@ubuntu: ~$ python3
>>> print(re.search(r"[a-z]way", "The end of the highway"))
<re.Match object; span=(18, 22), match='hway'>
>>> print(re.search(r"[a-z]way", "What a way to go"))
None
>>> print(re.search(r"cloud[a-zA-Z0-9]", "cloudy"))
<re.Match object; span=(0, 6), match='cloudy'>
>>> print(re.search(r"cloud[a-zA-Z0-9]", "cloud9"))
<re.Match object; span=(0, 6), match='cloud9'>
>>> print(re.search(r"^[^a-zA-Z]", "This is a sentence with spaces. "))
<re.Match object; span=(4, 5), match=' '>
>>> print(re.search(r"^[^a-zA-Z]", "This is a sentence with spaces. "))
<re.Match object; span=(30, 31), match='.'>
>>> print
```

***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.

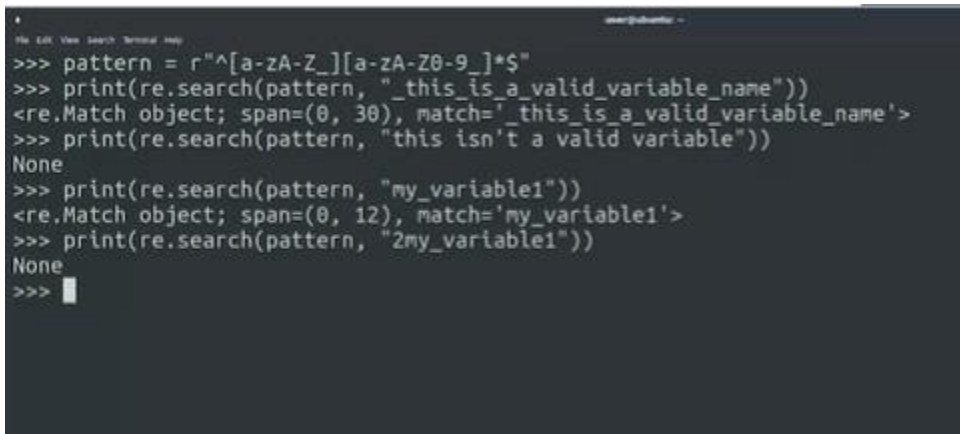
***square brace is called **Character Class**

*** **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.



```
user@ubuntu: ~  
File Edit View Search Terminal Help  
>>> pattern = r"^[a-zA-Z_][a-zA-Z0-9_]*$"  
>>> print(re.search(pattern, "_this_is_a_valid_variable_name"))  
<re.Match object; span=(0, 30), match='_this_is_a_valid_variable_name'>  
>>> print(re.search(pattern, "this isn't a valid variable"))  
None  
>>> print(re.search(pattern, "my_variable1"))  
<re.Match object; span=(0, 12), match='my_variable1'>  
>>> print(re.search(pattern, "2my_variable1"))  
None  
>>> █
```

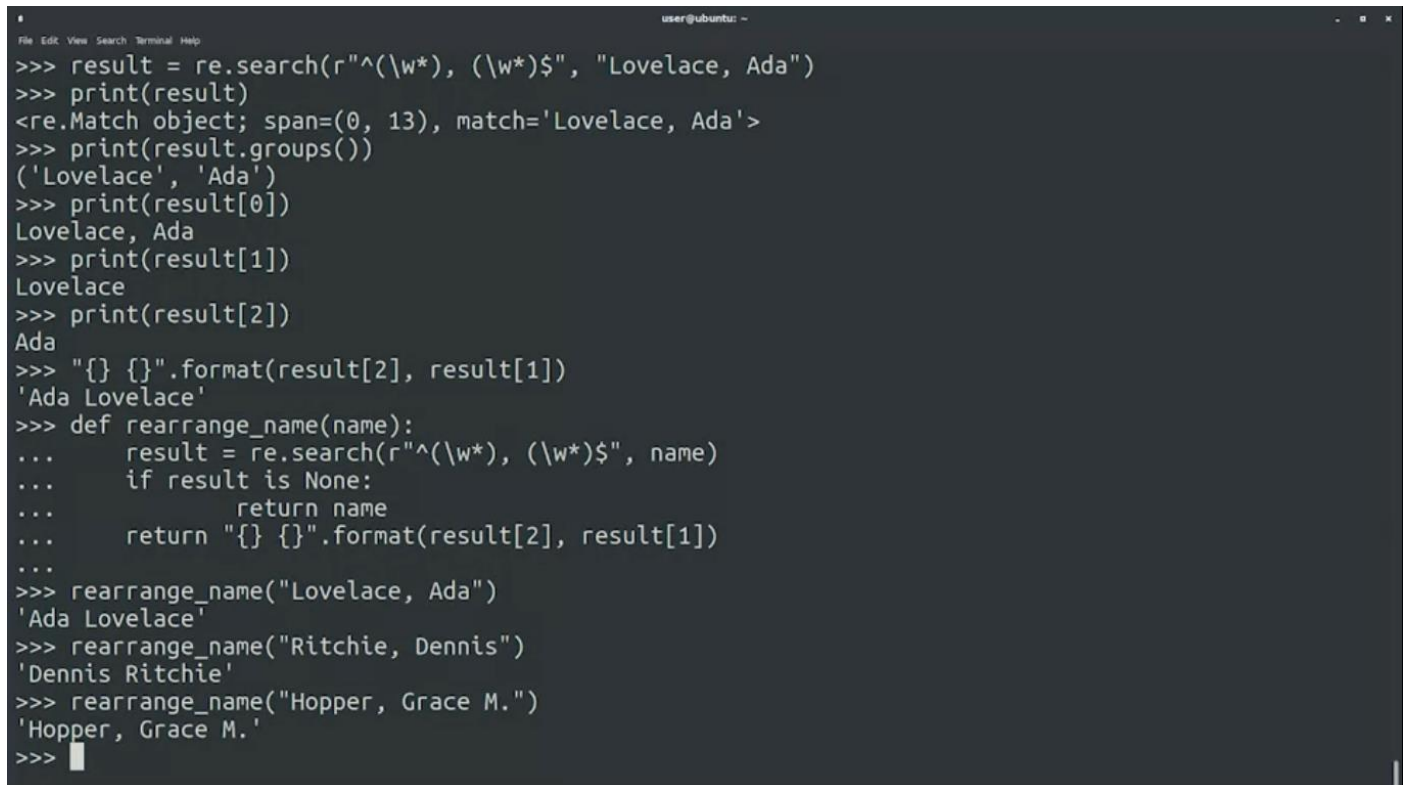
<https://regex101.com/>

<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)



```
user@ubuntu: ~  
File Edit View Search Terminal Help  
>>> result = re.search(r"^(\w*), (\w*)$", "Lovelace, Ada")  
>>> print(result)  
<re.Match object; span=(0, 13), match='Lovelace, Ada'>  
>>> print(result.groups())  
( 'Lovelace', 'Ada' )  
>>> print(result[0])  
Lovelace, Ada  
>>> print(result[1])  
Lovelace  
>>> print(result[2])  
Ada  
>>> "{} {}".format(result[2], result[1])  
'Ada Lovelace'  
>>> def rearrange_name(name):  
...     result = re.search(r"^(\w*), (\w*)$", name)  
...     if result is None:  
...         return name  
...     return "{} {}".format(result[2], result[1])  
...  
>>> rearrange_name("Lovelace, Ada")  
'Ada Lovelace'  
>>> rearrange_name("Ritchie, Dennis")  
'Dennis Ritchie'  
>>> rearrange_name("Hopper, Grace M.")  
'Hopper, Grace M.'  
>>> █
```

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]
```

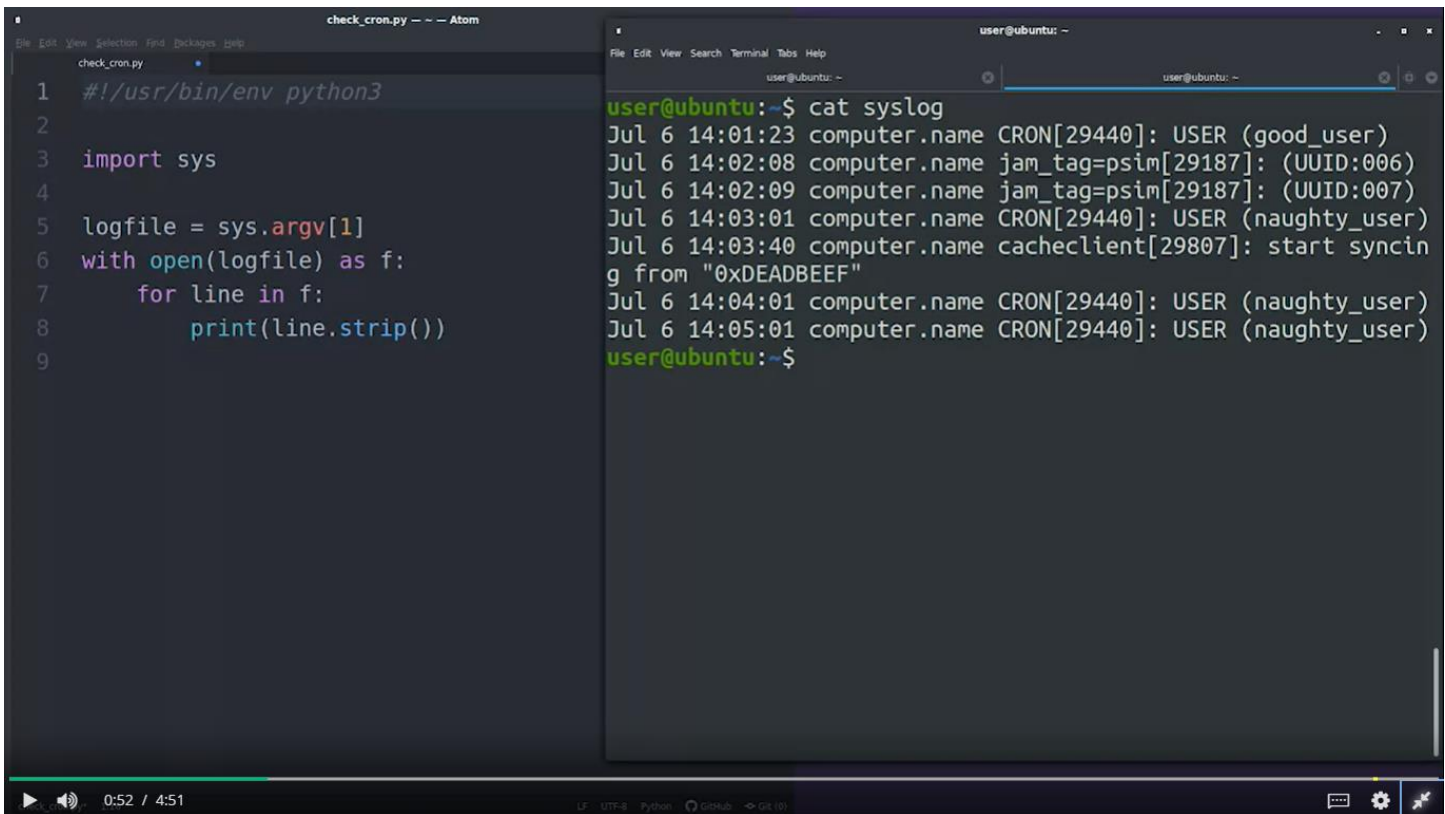
```
print(extract_pid("July 31 07:51:48 mycomputer bad_process[12345]: ERROR Performing package upgrade")) # 12345 (ERROR)
print(extract_pid("99 elephants in a [cage]")) # None
print(extract_pid("A string that also has numbers [34567] but no uppercase message")) # None
print(extract_pid("July 31 08:08:08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

<https://regexcrossword.com/>

Environment Variables:

```
user@ubuntu: ~  
File Edit View Search Terminal Help  
user@ubuntu:~$ echo $PATH  
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin  
user@ubuntu:~$ cat variables.py  
#!/usr/bin/env python3  
  
import os  
  
print("HOME: " + os.environ.get("HOME", ""))  
print("SHELL: " + os.environ.get("SHELL", ""))  
print("FRUIT: " + os.environ.get("FRUIT", ""))  
user@ubuntu:~$ ./variables.py  
HOME: /home/user  
SHELL: /bin/bash  
FRUIT:  
user@ubuntu:~$ export FRUIT=Pineapple  
user@ubuntu:~$ ./variables.py  
HOME: /home/user  
SHELL: /bin/bash  
FRUIT: Pineapple  
user@ubuntu:~$
```

***`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.



The image shows a split-screen view of a development environment. On the left, the Atom text editor is open with a file named `check_cron.py`. The code is a Python script that takes a filename as an argument, opens it, and prints each line. On the right, a terminal window shows the output of running `cat syslog`, displaying several log entries from the system log, including cron job executions for 'good_user' and 'naughty_user'.

```
check_cron.py -- Atom
1 #!/usr/bin/env python3
2
3 import sys
4
5 logfile = sys.argv[1]
6 with open(logfile) as f:
7     for line in f:
8         print(line.strip())
9
user@ubuntu: ~$ cat syslog
Jul 6 14:01:23 computer.name CRON[29440]: USER (good_user)
Jul 6 14:02:08 computer.name jam_tag=psim[29187]: (UUID:006)
Jul 6 14:02:09 computer.name jam_tag=psim[29187]: (UUID:007)
Jul 6 14:03:01 computer.name CRON[29440]: USER (naughty_user)
Jul 6 14:03:40 computer.name cacheclient[29807]: start syncin
g from "0xDEADBEEF"
Jul 6 14:04:01 computer.name CRON[29440]: USER (naughty_user)
Jul 6 14:05:01 computer.name CRON[29440]: USER (naughty_user)
user@ubuntu: ~$
```

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) | <code>a == b</code> | |
| assertNotEqual(a, b) | <code>a != b</code> | |
| assertTrue(x) | <code>bool(x)</code> is True | |
| assertFalse(x) | <code>bool(x)</code> is False | |
| assertIs(a, b) | <code>a</code> is <code>b</code> | 3.1 |
| assertIsNot(a, b) | <code>a</code> is not <code>b</code> | 3.1 |
| assertIsNone(x) | <code>x</code> is None | 3.1 |
| assertIsNotNone(x) | <code>x</code> is not None | 3.1 |
| assertIn(a, b) | <code>a</code> in <code>b</code> | 3.1 |
| assertNotIn(a, b) | <code>a</code> not in <code>b</code> | 3.1 |
| assertIsInstance(a, b) | <code>isinstance(a, b)</code> | 3.2 |
| assertNotIsInstance(a, b) | <code>not isinstance(a, b)</code> | 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.

```
1 $ python throwing.py
2 Traceback (most recent call last):
3   File "throwing.py", line 10, in <module>
4     main()
5   File "throwing.py", line 7, in main
6     throws()
7   File "throwing.py", line 4, in throws
8     raise RuntimeError('this is the error message')
9 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.

```
1 #!/usr/bin/env python
2
3 import sys
4
5 def throws():
6     raise RuntimeError('this is the error message')
7
8 def main():
9     try:
10         throws()
11         return 0
12     except Exception, err:
13         sys.stderr.write('ERROR: %sn' % str(err))
14         return 1
15
16 if __name__ == '__main__':
17     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.

```
1  #!/usr/bin/env python
2
3  import logging
4  import sys
5
6  def throws():
7      raise RuntimeError('this is the error message')
8
9  def main():
10     logging.basicConfig(level=logging.WARNING)
11     log = logging.getLogger('example')
12     try:
13         throws()
14         return 0
15     except Exception, err:
16         log.exception('Error from throws():')
17         return 1
18
19  if __name__ == '__main__':
20     sys.exit(main())
```

In this example, the logger is configured 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.

```

1 $ python logging_errors.py
2 ERROR:example:Error from throws():
3 Traceback (most recent call last):
4   File "logging_errors.py", line 13, in main
5     throws()
6   File "logging_errors.py", line 7, in throws
7     raise RuntimeError('this is the error message')
8 RuntimeError: this is the error message

```

Bash Scripting:

```

user@ubuntu: ~
File Edit View Search Terminal Help
user@ubuntu:~$ ./stdout_example.py > new_file.txt
user@ubuntu:~$ cat new_file.txt
Don't mind me, just a bit of text here...
user@ubuntu:~$ ./stdout_example.py >> new_file.txt
user@ubuntu:~$ cat new_file.txt
Don't mind me, just a bit of text here...
Don't mind me, just a bit of text here...
user@ubuntu:~$ cat streams_err.py
#!/usr/bin/env python3

data = input("This will come from STDIN: ")
print("Now we write it to STDOUT: " + data)
raise ValueError("Now we generate an error to STDERR")
user@ubuntu:~$ ./streams_err.py < new_file.txt
This will come from STDIN: Now we write it to STDOUT: Don't mind me, just a bit of text here...
Traceback (most recent call last):
  File "./streams_err.py", line 5, in <module>
    raise ValueError("Now we generate an error to STDERR")
ValueError: Now we generate an error to STDERR
user@ubuntu:~$ ./streams_err.py < new_file.txt 2> error_file.txt
This will come from STDIN: Now we write it to STDOUT: Don't mind me, just a bit of text here...
user@ubuntu:~$ cat error_file.txt
Traceback (most recent call last):
  File "./streams_err.py", line 5, in <module>
    raise ValueError("Now we generate an error to STDERR")
ValueError: Now we generate an error to STDERR
user@ubuntu:~$ echo "These are the contents of the file" > myamazingfile.txt

```

```
user@ubuntu: ~  
File Edit View Search Terminal Help  
user@ubuntu:~$ cat capitalize.py  
#!/usr/bin/env python3  
  
import sys  
  
for line in sys.stdin:  
    print(line.strip().capitalize())  
user@ubuntu:~$ cat haiku.txt  
advance your career,  
automating with Python,  
it's so fun to learn.  
user@ubuntu:~$ cat haiku.txt | ./capitalize.py  
Advance your career,  
Automating with python,  
It's so fun to learn.  
user@ubuntu:~$ ./capitalize.py < haiku.txt  
Advance your career,  
Automating with python,  
It's so fun to learn.  
user@ubuntu:~$
```

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)