

Activity No. 5.2 Files	
Course Code: CPE 103	Program: Computer Engineering
Course Title: Object Oriented Programming	Date Performed: 4/12/2025
Section: 1A	Date Submitted: 4/12/2025
Name: CATAHAN, JOSHUA A.	Instructor: ENGR. MARIA RIZETTE SAYO
1. Objective(s)	
This activity aims to demonstrate of creating a file, executing the file, updating the file, and deleting the file using python.	
2. Intended Learning Outcomes (ILOs)	
After this module, the students should: <ul style="list-style-type: none"> Demonstrate using files for different operations in python. 	
3. Discussion	
<p>Persistence</p> <p>So far, we have learned how to write programs and communicate our intentions to the <i>Central Processing Unit</i> using conditional execution, functions, and iterations. We have learned how to create and use data structures in the <i>Main Memory</i>. The CPU and memory are where our software works and runs. It is where all of the “thinking” happens.</p> <p>But if you recall from our hardware architecture discussions, once the power is turned off, anything stored in either the CPU or main memory is erased. So up to now, our programs have just been transient fun exercises to learn Python.</p> <p>Secondary Memory</p> <p>In this chapter, we start to work with <i>Secondary Memory</i> (or files). Secondary memory is not erased when the power is turned off. Or in the case of a USB flash drive, the data we write from our programs can be removed from the system and transported to another system.</p> <p>We will primarily focus on reading and writing text files such as those we create in a text editor. Later we will see how to work with database files which are binary files, specifically designed to be read and written through database software.</p> <p>Opening files</p> <p>When we want to read or write a file (say on your hard drive), we first must <i>open</i> the file. Opening the file communicates with your operating system, which knows where the data for each file is stored. When you open a file, you are asking the operating system to find the file by name and make sure the file exists. In this example, we open the file <i>mbox.txt</i>, which should be stored in the same folder that you are in when you start Python.</p> <pre>>>> fhand = open('mbox.txt') >>> print(fhand) <_io.TextIOWrapper name='mbox.txt' mode='r' encoding='cp1252'></pre> <p>If the open is successful, the operating system returns us a <i>file handle</i>. The file handle is not the actual data contained in the file, but instead it is a “handle” that we can use to read the data. You are given a handle if the requested file exists and you have the proper permissions to read the file.</p> <p>A File Handle</p>	

If the file does not exist, open will fail with a traceback and you will not get a handle to access the contents of the file:

```
>>> fhand = open('stuff.txt')
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

FileNotFoundError: [Errno 2] No such file or directory: 'stuff.txt'

Later we will use try and except to deal more gracefully with the situation where we attempt to open a file that does not exist.

Text files and lines

A text file can be thought of as a sequence of lines, much like a Python string can be thought of as a sequence of characters. For example, this is a sample of a text file which records mail activity from various individuals in an open source project development team:

From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008

Return-Path: <postmaster@collab.sakaiproject.org>

Date: Sat, 5 Jan 2008 09:12:18 -0500

To: source@collab.sakaiproject.org

From: stephen.marquard@uct.ac.za

Subject: [sakai] svn commit: r39772 - content/branches/

Details: <http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772>

These files are in a standard format for a file containing multiple mail messages. The lines which start with "From" separate the messages and the lines which start with "From:" are part of the messages.

To break the file into lines, there is a special character that represents the "end of the line" called the *newline* character.

In Python, we represent the *newline* character as a backslash-n in string constants. Even though this looks like two characters, it is actually a single character. When we look at the variable by entering "stuff" in the interpreter, it shows us the \n in the string, but when we use print to show the string, we see the string broken into two lines by the newline character.

```
>>> stuff = 'Hello\nWorld!'
```

```
>>> stuff
```

```
'Hello\nWorld!'
```

```
>>> print(stuff)
```

```
Hello
```

```
World!
```

```
>>> stuff = 'X\nY'
```

```
>>> print(stuff)
```

```
X
```

```
Y
```

```
>>> len(stuff)
```

```
3
```

You can also see that the length of the string X\nY is *three* characters because the newline character is a single character.

So when we look at the lines in a file, we need to *imagine* that there is a special invisible character called the newline at the end of each line that marks the end of the line.

So the newline character separates the characters in the file into lines.

Reading files

While the *file handle* does not contain the data for the file, it is quite easy to construct a for loop to read through and count each of the lines in a file:

```
fhand = open('mbox-short.txt')
count = 0
for line in fhand:
    count = count + 1
print('Line Count:', count)
```

Code: <https://www.py4e.com/code3/open.py>

We can use the file handle as the sequence in our for loop. Our for loop simply counts the number of lines in the file and prints them out. The rough translation of the for loop into English is, “for each line in the file represented by the file handle, add one to the count variable.”

The reason that the open function does not read the entire file is that the file might be quite large with many gigabytes of data. The open statement takes the same amount of time regardless of the size of the file. The for loop actually causes the data to be read from the file.

When the file is read using a for loop in this manner, Python takes care of splitting the data in the file into separate lines using the newline character. Python reads each line through the newline and includes the newline as the last character in the line variable for each iteration of the for loop.

Because the for loop reads the data one line at a time, it can efficiently read and count the lines in very large files without running out of main memory to store the data. The above program can count the lines in any size file using very little memory since each line is read, counted, and then discarded.

If you know the file is relatively small compared to the size of your main memory, you can read the whole file into one string using the read method on the file handle.

```
>>> fhand = open('mbox-short.txt')
>>> inp = fhand.read()
>>> print(len(inp))
94626
>>> print(inp[:20])
```

From stephen.marquar

In this example, the entire contents (all 94,626 characters) of the file *mbox-short.txt* are read directly into the variable *inp*. We use string slicing to print out the first 20 characters of the string data stored in *inp*.

When the file is read in this manner, all the characters including all of the lines and newline characters are one big string in the variable *inp*. It is a good idea to store the output of read as a variable because each call to read exhausts the resource:

```
>>> fhand = open('mbox-short.txt')
>>> print(len(fhand.read()))
94626
>>> print(len(fhand.read()))
0
```

Remember that this form of the open function should only be used if the file data will fit comfortably in the main memory of your computer. If the file is too large to fit in main memory, you should write your program to read the file in chunks using a for or while loop.

Searching through a file

When you are searching through data in a file, it is a very common pattern to read through a file, ignoring most of the lines and only processing lines which meet a particular condition. We can combine the pattern for reading a file with string methods to build simple search mechanisms.

For example, if we wanted to read a file and only print out lines which started with the prefix "From:", we could use the string method *startswith* to select only those lines with the desired prefix:

```
fhand = open('mbox-short.txt')
for line in fhand:
    if line.startswith('From:'):
        print(line)
```

Code: <https://www.py4e.com/code3/search1.py>

When this program runs, we get the following output:

From: stephen.marquard@uct.ac.za

From: louis@media.berkeley.edu

From: zqian@umich.edu

From: rjlowe@iupui.edu

...

The output looks great since the only lines we are seeing are those which start with "From:", but why are we seeing the extra blank lines? This is due to that invisible *newline* character. Each of the lines ends with a newline, so the print statement prints the string in the variable *line* which includes a newline and then print adds *another* newline, resulting in the double spacing effect we see.

We could use line slicing to print all but the last character, but a simpler approach is to use the *rstrip* method which strips whitespaces from the right side of a string as follows:

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if line.startswith('From:'):
        print(line)
```

Code: <https://www.py4e.com/code3/search2.py>

When this program runs, we get the following output:

From: stephen.marquard@uct.ac.za

From: louis@media.berkeley.edu

From: zqian@umich.edu

From: rjlowe@iupui.edu

From: zqian@umich.edu

From: rjlowe@iupui.edu

From: cwen@iupui.edu

...

As your file processing programs get more complicated, you may want to structure your search loops using *continue*. The basic idea of the search loop is that you are looking for "interesting" lines and effectively skipping "uninteresting" lines. And then when we find an interesting line, we do something with that line.

We can structure the loop to follow the pattern of skipping uninteresting lines as follows:

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    # Skip 'uninteresting lines'
    if not line.startswith('From:'):
        continue
    # Process our 'interesting' line
    print(line)
```

Code: <https://www.py4e.com/code3/search3.py>

The output of the program is the same. In English, the uninteresting lines are those which do not start with “From:”, which we skip using continue. For the “interesting” lines (i.e., those that start with “From:”) we perform the processing.

We can use the find string method to simulate a text editor search that finds lines where the search string is anywhere in the line. Since find looks for an occurrence of a string within another string and either returns the position of the string or -1 if the string was not found, we can write the following loop to show lines which contain the string “@uct.ac.za” (i.e., they come from the University of Cape Town in South Africa):

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if line.find('@uct.ac.za') == -1: continue
    print(line)
```

Code: <https://www.py4e.com/code3/search4.py>

Which produces the following output:

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
X-Authentication-Warning: set sender to stephen.marquard@uct.ac.za using -f
From: stephen.marquard@uct.ac.za
Author: stephen.marquard@uct.ac.za
From david.horwitz@uct.ac.za Fri Jan  4 07:02:32 2008
X-Authentication-Warning: set sender to david.horwitz@uct.ac.za using -f
From: david.horwitz@uct.ac.za
Author: david.horwitz@uct.ac.za
...
```

Here we also use the contracted form of the if statement where we put the continue on the same line as the if. This contracted form of the if functions the same as if the continue were on the next line and indented.

Letting the user choose the file name

We really do not want to have to edit our Python code every time we want to process a different file. It would be more usable to ask the user to enter the file name string each time the program runs so they can use our program on different files without changing the Python code.

This is quite simple to do by reading the file name from the user using input as follows:

```
fname = input('Enter the file name: ')
```

```
fhand = open(fname)
count = 0
for line in fhand:
    if line.startswith('Subject:'):
        count = count + 1
print('There were', count, 'subject lines in', fname)
```

Code: <https://www.py4e.com/code3/search6.py>

We read the file name from the user and place it in a variable named `fname` and open that file. Now we can run the program repeatedly on different files.

```
python search6.py
```

Enter the file name: mbox.txt

There were 1797 subject lines in mbox.txt

```
python search6.py
```

Enter the file name: mbox-short.txt

There were 27 subject lines in mbox-short.txt

Before peeking at the next section, take a look at the above program and ask yourself, “What could go possibly wrong here?” or “What might our friendly user do that would cause our nice little program to ungracefully exit with a traceback, making us look not-so-cool in the eyes of our users?”

Using try, except, and open

I told you not to peek. This is your last chance.

What if our user types something that is not a file name?

```
python search6.py
```

Enter the file name: missing.txt

Traceback (most recent call last):

File "search6.py", line 2, in <module>

fhand = open(fname)

FileNotFoundError: [Errno 2] No such file or directory: 'missing.txt'

```
python search6.py
```

Enter the file name: na na boo boo

Traceback (most recent call last):

File "search6.py", line 2, in <module>

fhand = open(fname)

FileNotFoundError: [Errno 2] No such file or directory: 'na na boo boo'

Do not laugh. Users will eventually do every possible thing they can do to break your programs, either mistakenly or with malicious intent. As a matter of fact, an important part of any software development team is a person or group called *Quality Assurance* (or QA for short) whose very job it is to do the craziest things possible in an attempt to break the software that the programmer has created.

The QA team is responsible for finding the flaws in programs before we have delivered the program to the end users who may be purchasing the software or paying our salary to write the software. So the QA team is the programmer's best friend.

So now that we see the flaw in the program, we can elegantly fix it using the try/except structure. We need to assume that the open call might fail and add recovery code when the open fails as follows:

```

fname = input('Enter the file name: ')
try:
    fhand = open(fname)
except:
    print('File cannot be opened:', fname)
    exit()
count = 0
for line in fhand:
    if line.startswith('Subject:'):
        count = count + 1
print('There were', count, 'subject lines in', fname)

```

Code: <https://www.py4e.com/code3/search7.py>

The exit function terminates the program. It is a function that we call that never returns. Now when our user (or QA team) types in silliness or bad file names, we “catch” them and recover gracefully:

```
python search7.py
```

```
Enter the file name: mbox.txt
```

```
There were 1797 subject lines in mbox.txt
```

```
python search7.py
```

```
Enter the file name: na na boo boo
```

```
File cannot be opened: na na boo boo
```

Protecting the open call is a good example of the proper use of try and except in a Python program. We use the term “Pythonic” when we are doing something the “Python way”. We might say that the above example is the Pythonic way to open a file.

Once you become more skilled in Python, you can engage in repartee with other Python programmers to decide which of two equivalent solutions to a problem is “more Pythonic”. The goal to be “more Pythonic” captures the notion that programming is part engineering and part art. We are not always interested in just making something work, we also want our solution to be elegant and to be appreciated as elegant by our peers.

Writing files

To write a file, you have to open it with mode “w” as a second parameter:

```
>>> fout = open('output.txt', 'w')
```

```
>>> print(fout)
```

```
<_io.TextIOWrapper name='output.txt' mode='w' encoding='cp1252'>
```

If the file already exists, opening it in write mode clears out the old data and starts fresh, so be careful! If the file doesn’t exist, a new one is created.

The write method of the file handle object puts data into the file, returning the number of characters written. The default write mode is text for writing (and reading) strings.

```
>>> line1 = "This here's the wattle,\n"
```

```
>>> fout.write(line1)
```

```
24
```

Again, the file object keeps track of where it is, so if you call write again, it adds the new data to the end.

We must make sure to manage the ends of lines as we write to the file by explicitly inserting the newline character when we want to end a line. The print statement automatically appends a newline, but the write method does not add the newline automatically.

```
>>> line2 = 'the emblem of our land.\n'
```

```
>>> fout.write(line2)
```

```
24
```

When you are done writing, you have to close the file to make sure that the last bit of data is physically written to the disk so it will not be lost if the power goes off.

```
>>> fout.close()
```

We could close the files which we open for read as well, but we can be a little sloppy if we are only opening a few files since Python makes sure that all open files are closed when the program ends. When we are writing files, we want to explicitly close the files so as to leave nothing to chance.

Reference:

PY4E - Python for everybody. (n.d.). <https://www.py4e.com/html3/07-files>

4. Materials and Equipment

To properly perform this activity, the student must have:

- Python
- Spyder IDE
- Jupyter Notebook

5. Procedure

1. Open the Anaconda.
2. Use the jupyter notebook and follow the instructions below.
3. Provide a screenshot for every test case in each code and insert in the Output section with a corresponding description and observation.

Opening files

```
[ ] fhand = open('mbox.txt')  
    print(fhand)
```

```
[ ] fhand = open('stuff.txt')
```


Text files and lines

```
[ ] stuff = 'Hello\nWorld!'
    stuff
```

```
[ ] print(stuff)
```

```
[ ] stuff = 'X\nY'
    print(stuff)
```

```
[ ] len(stuff)
```

Reading files

```
[ ] fhand = open('mbox-short.txt')
    count = 0
    for line in fhand:
        count = count + 1
    print('Line Count:', count)
```

```
[ ] fhand = open('mbox-short.txt')
    inp = fhand.read()
    print(len(inp))
```

```
[ ] print(inp[:20])
```

```
[ ] fhand = open('mbox-short.txt')
    print(len(fhand.read()))
```

```
[ ] print(len(fhand.read()))
```

Searching through a file

```
[ ] fhand = open('mbox-short.txt')
    for line in fhand:
        if line.startswith('From:'):
            print(line)
```

```
[ ] fhand = open('mbox-short.txt')
    for line in fhand:
        line = line.rstrip()
        if line.startswith('From:'):
            print(line)
```

```
[ ] fhand = open('mbox-short.txt')
    for line in fhand:
        line = line.rstrip()
        # Skip 'uninteresting lines'
        if not line.startswith('From:'):
            continue
        # Process our 'interesting' line
        print(line)
```

```
[ ] fhand = open('mbox-short.txt')
    for line in fhand:
        line = line.rstrip()
        if line.find('@uct.ac.za') == -1: continue
        print(line)
```

Letting the user choose the file name

```
[ ] fname = input('Enter the file name: ')
    fhand = open(fname)
    count = 0
    for line in fhand:
        if line.startswith('Subject:'):
            count = count + 1
    print('There were', count, 'subject lines in', fname)
```

Using try, except, and open

```
[ ] fname = input('Enter the file name: ')
try:
    fhand = open(fname)
except:
    print('File cannot be opened:', fname)
    exit()
count = 0
for line in fhand:
    if line.startswith('Subject:'):
        count = count + 1
print('There were', count, 'subject lines in', fname)
```

Writing files

```
[ ] fout = open('output.txt', 'w')
    print(fout)
```

```
[ ] line1 = "This here's the wattle,\n"
    fout.write(line1)
```

```
[ ] line2 = 'the emblem of our land.\n'
    fout.write(line2)
```

```
[ ] fout.close()
```

6. Output

Provide an output of your work here. (include an analyzation for every screenshot or output)

The screenshot shows the PyCharm IDE interface. The top toolbar includes icons for file operations, search, and settings. The main editor window displays the file 'OPENING FILES.py' with the following code:

```
1 fhand = open('mbox.txt')
2 print(fhand)
3
4 fhand = open('stuff.txt')
```

Below the editor is the 'Run' console, which shows the execution command and the output:

```
C:\Users\ucc_e\PycharmProjects\PythonProject10\.venv\Scripts\python.exe "C:\Users\ucc_e\PycharmProjects\PythonProject10\OPENING FILES.py"
<_io.TextIOWrapper name='mbox.txt' mode='r' encoding='cp1252'>
Process finished with exit code 0
```

The status bar at the bottom indicates the project is 'PythonProject10', the file is 'OPENING FILES.py', and the Python version is '3.12'.

OBSERVATION:

- From the way the code was written, it wants to open some text file which is named: "mbox.txt" and "stuff.txt". Therefore, when I tried to run the code, it outputs an error. To fix this, I created a text file with the same name, thus, making the code run properly without an error.

The screenshot shows a code execution window titled 'TEXT FILES AND LINES'. It displays three code snippets and their corresponding outputs:

```
[6] stuff = 'Hello\nWorld!'
    stuff
'Hello\nWorld!'
```

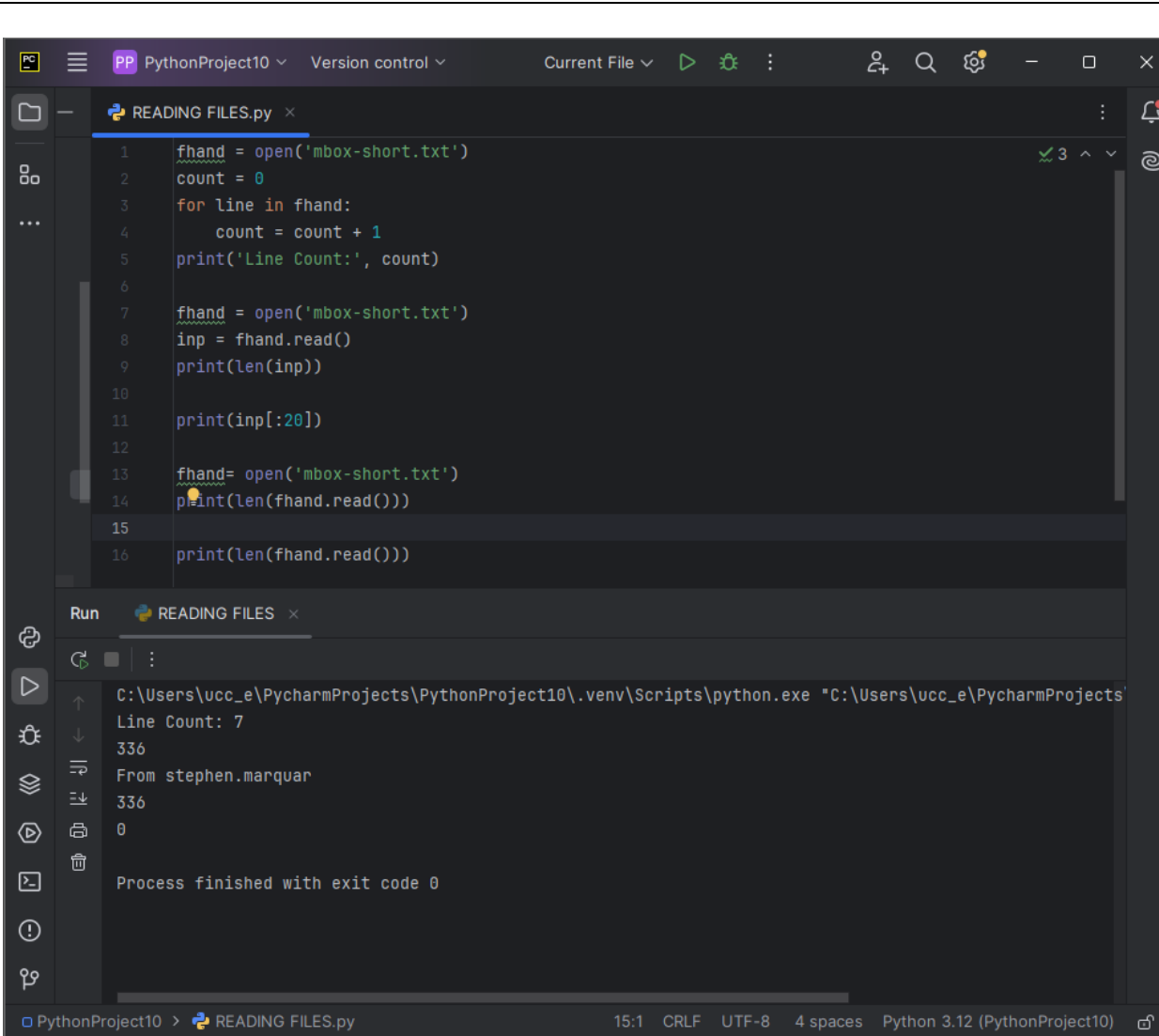
```
print(stuff)
Hello
World!
```

```
[8] stuff = 'X\nY'
    print(stuff)
X
Y
```

```
len(stuff)
3
```

OBSERVATION:

- In the first part of the code, we can see that it outputs exactly what was written in the variable name "stuff". While if you write "print(stuff)", it understands the new line function, making the subsequent letters/words to be written in a new line.
- For "len(stuff)", we can see that it counts the "\n" as another letter.



The screenshot shows the PyCharm IDE interface. The top toolbar includes icons for file operations, version control, and running code. The main editor window displays a Python script named 'READING FILES.py'. The script contains three blocks of code: a loop to count lines, a full file read, and a second full file read after reopening the file. The Run window at the bottom shows the execution output, including the line count and the content of the file, confirming that the file was successfully reopened for the second read.

```
1 fhand = open('mbox-short.txt')
2 count = 0
3 for line in fhand:
4     count = count + 1
5 print('Line Count:', count)
6
7 fhand = open('mbox-short.txt')
8 inp = fhand.read()
9 print(len(inp))
10
11 print(inp[:20])
12
13 fhand= open('mbox-short.txt')
14 print(len(fhand.read()))
15
16 print(len(fhand.read()))
```

Run READING FILES

```
C:\Users\ucc_e\PycharmProjects\PythonProject10\.venv\Scripts\python.exe "C:\Users\ucc_e\PycharmProjects
Line Count: 7
336
From stephen.marquar
336
0
Process finished with exit code 0
```

PythonProject10 > READING FILES.py 15:1 CRLF UTF-8 4 spaces Python 3.12 (PythonProject10)

OBSERVATION:

- In the first part of the code, the loop goes through each line in the file and counts them one by one, showing the total number of lines at the end.
- Later, when the file is opened again and read fully using `.read()`, it returns the entire content as a single string—and we see that the file needs to be reopened before reading again, otherwise `.read()` will return 0 since the file pointer is already at the end.

The screenshot shows the PyCharm IDE with a file named `SEARCHING THROUGH A FILE.py`. The code consists of five blocks, each opening `mbox-short.txt` and iterating over its lines. The first block prints all lines starting with `'From: '`. The second block strips trailing spaces from each line before printing. The third block skips lines that do not start with `'From: '` using a `continue` statement. The fourth block further refines the search by skipping lines that do not end with `@uct.ac.za` using `line.find('@uct.ac.za') == -1`. The output window shows the results of these iterations: the first iteration prints four lines with trailing spaces, the second prints the same four lines without trailing spaces, the third prints the same four lines, and the fourth prints only the first two lines, which end with `@uct.ac.za`. The process finished with exit code 0.

```
1 fhand = open('mbox-short.txt')
2 for line in fhand:
3     if line.startswith('From:'):
4         print(line)
5
6 fhand = open('mbox-short.txt')
7 for line in fhand:
8     line = line.rstrip()
9     if line.startswith('From:'):
10        print(line)
11
12 fhand = open('mbox-short.txt')
13 for line in fhand:
14     line = line.rstrip()
15     #skip 'uninteresting lines'
16     if not line.startswith('From:'):
17         continue
18     #process our 'interesting' line
19     print(line)
20
21 fhand = open('mbox-short.txt')
22 for line in fhand:
23     line = line.rstrip()
24     if line.find('@uct.ac.za') == -1: continue
25     print(line)
```

Run SEARCHING THROUGH A FILE.py

C:\Users\ucc_e\PycharmProjects\PythonProject10\.venv\Scripts\python.exe "C:\Users\ucc_e\PycharmProjects\PythonProject10\SEARCHING THROUGH A FILE.py"

From: stephen.marquard@uct.ac.za

From: stephen.marquard@uct.ac.za

From: stephen.marquard@uct.ac.za

From: stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008

From: stephen.marquard@uct.ac.za

Process finished with exit code 0

OBSERVATION:

- In the beginning, the code prints out every line that starts with 'From:', but because it doesn't remove the newline characters, there's an awkward space between each printed line.
- As the code progresses, it becomes more refined by stripping off extra spaces with `rstrip()` and using `continue` to skip irrelevant lines, which helps in targeting specific patterns like email addresses ending with `@uct.ac.za` more accurately.

The screenshot shows the PyCharm IDE with a file named `LETTING THE USER CHOOSE THE FILE NAME.py`. The code prompts the user to enter a file name, opens the file, and counts the number of lines starting with `'Subject: '`. The output window shows the user entering `mbox.txt` and the program outputting `There were 2 subject lines in mbox.txt`. The process finished with exit code 0.

```
1 fname = input('Enter the file name: ')
2 fhand = open(fname)
3 count = 0
4 for line in fhand:
5     if line.startswith('Subject: '):
6         count = count + 1
7 print('There were', count, 'subject lines in', fname)
```

Run LETTING THE USER CHOOSE THE FILE NAME.py

C:\Users\ucc_e\PycharmProjects\PythonProject10\.venv\Scripts\python.exe "C:\Users\ucc_e\PycharmProjects\PythonProject10\LETTING THE USER CHOOSE THE FILE NAME.py"

Enter the file name: mbox.txt

There were 2 subject lines in mbox.txt

Process finished with exit code 0

OBSERVATION:

- The code asks the user to type in the name of a file, then opens it and looks through each line to count how many starts with 'Subject:'.

In the output, it prints a simple summary showing how many subject lines were found in the file the user specified, making it a flexible way to analyze different files.

The image shows a PyCharm IDE window with a Python script titled "USING TRY, EXCEPT, AND OPEN.py". The script is as follows:

```
1 fname = input('Enter the file name: ')
2 try:
3     fhand = open(fname)
4 except:
5     print('File cannot be opened:', fname)
6     quit()
7 count = 0
8 for line in fhand:
9     if line.startswith('Subject:'):
10         count = count + 1
11 print('There were', count, 'subject lines in', fname)
```

The Run console at the bottom shows the execution output:

```
C:\Users\ucc_e\PycharmProjects\PythonProject10\.venv\Scripts\python.exe "C:\Users\ucc_e\PycharmProjects\PythonProject10\USING TRY, EXCEPT, AND OPEN.py"
Enter the file name: stuff.txt
There were 4 subject lines in stuff.txt
Process finished with exit code 0
```

OBSERVATION:

- In this code, it checks whether the user inputs exist within the project in PyCharm. If it doesn't exist, it will output "File cannot be opened: (user inputs)". On the contrary, If the user inputs a valid text file name, it will output the number of subject lines written in it just like the previous code.

The screenshot shows the PyCharm IDE interface. The main editor window displays the file `WRITING FILES.py` with the following Python code:

```
1 fout = open('output.txt', 'w')
2 print(fout)
3
4 line1 = "This here's the wattle,\n"
5 fout.write(line1)
6
7 line2 = "the emblem of our land.\n"
8 fout.write(line2)
9
10 fout.close()
```

Below the editor, the Run console shows the execution of the script. The command executed is `C:\Users\ucc_e\PycharmProjects\PythonProject10\.venv\Scripts\python.exe "C:\Users\<_io.TextIOWrapper name='output.txt' mode='w' encoding='cp1252'>`. The output indicates that the process finished with exit code 0.

The screenshot shows the contents of the `output.txt` file. The text written to the file is:

```
1 This here's the wattle,
2 the emblem of our land.
3 |
```

OBSERVATION:

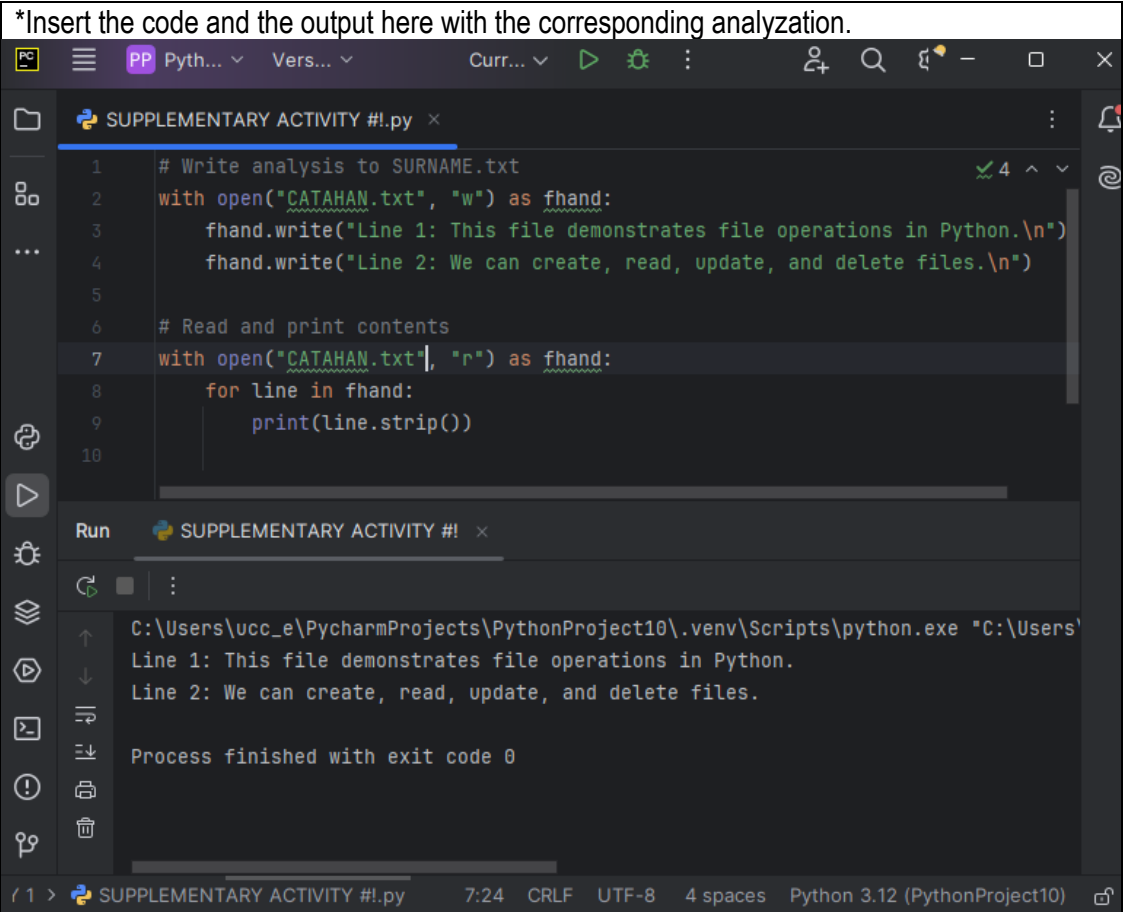
- In this code, it opens a new text file named "output.txt" when the user runs the program. Additionally, the code also includes the text to be written in this file per line using `fout.write(line1)`. Where `fout` is the variable to open the text file, while `line1` is the text to be included inside the file.

7. Supplementary Activity

Solve the following problems:

1. Create a SURNAME.txt to file write your analysis in the text file. Write a program to read through a file and print the contents of the file (line by line).

*Insert the code and the output here with the corresponding analyzation.



The screenshot shows the PyCharm IDE interface. The main editor window displays a Python script named 'SUPPLEMENTARY ACTIVITY #!.py'. The script consists of two parts: first, it writes two lines of text to a file named 'CATAHAN.txt' using the 'with open()' and 'fhand.write()' methods; second, it reads the contents of 'CATAHAN.txt' line by line using 'with open()' in read mode and a 'for' loop with 'fhand.readlines()' to print each line. Below the editor, the 'Run' window shows the execution output, which matches the text written to the file. The status bar at the bottom indicates the file encoding is UTF-8 and the Python version is 3.12.

```
1 # Write analysis to SURNAME.txt
2 with open("CATAHAN.txt", "w") as fhand:
3     fhand.write("Line 1: This file demonstrates file operations in Python.\n")
4     fhand.write("Line 2: We can create, read, update, and delete files.\n")
5
6 # Read and print contents
7 with open("CATAHAN.txt", "r") as fhand:
8     for line in fhand:
9         print(line.strip())
10
```

Run SUPPLEMENTARY ACTIVITY #! x

C:\Users\ucc_e\PycharmProjects\PythonProject10\.venv\Scripts\python.exe "C:\Users\ucc_e\PycharmProjects\PythonProject10\SUPPLEMENTARY ACTIVITY #!.py"

Line 1: This file demonstrates file operations in Python.

Line 2: We can create, read, update, and delete files.

Process finished with exit code 0

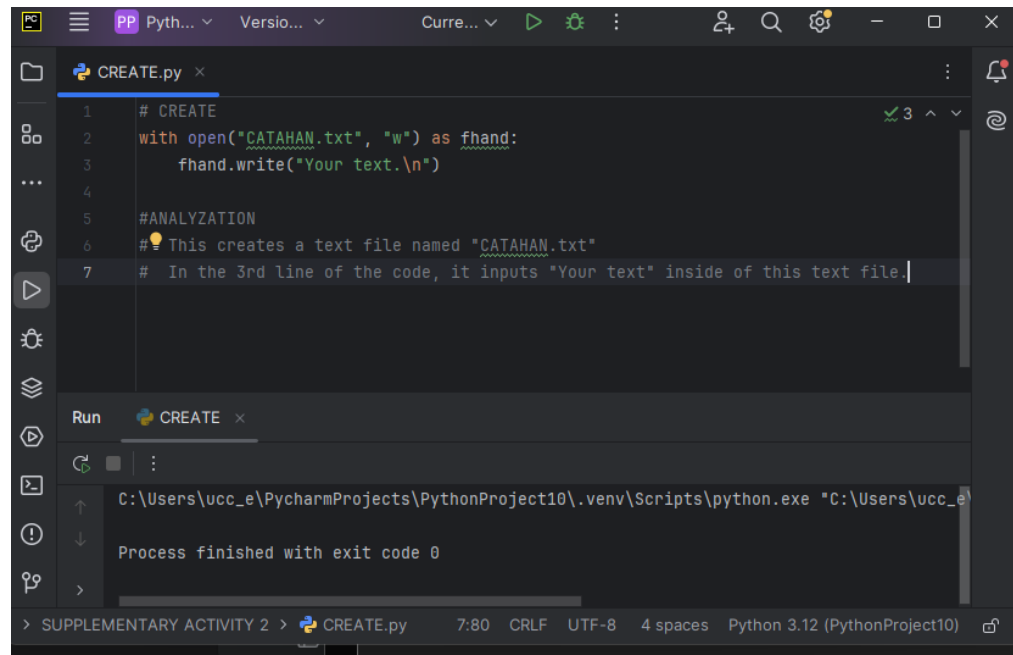
1 > SUPPLEMENTARY ACTIVITY #!.py 7:24 CRLF UTF-8 4 spaces Python 3.12 (PythonProject10)

ANALYZATION:

- The first part of the code writes two lines of text into a file named CATAHAN.txt using with open(CATAHAN.txt, "w"), which ensures the file is properly handled and closed after writing.
- In the second part, the file is reopened in read mode, and strip() is used to remove any trailing newline characters, so the output is clean, and lines are printed nicely one after the other.

2. Write a python program that demonstrate the CRUD (Create, Read, Update and Delete). The txt file name SURNAME.txt.

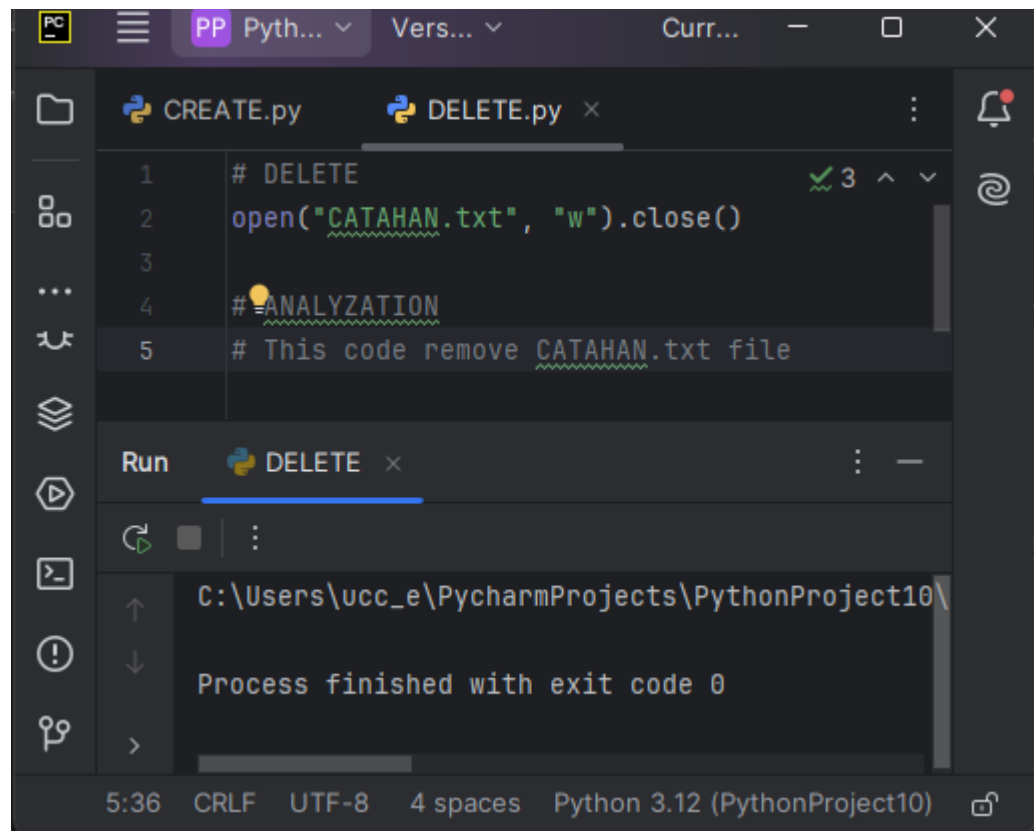
*Insert the code and the output here with the corresponding analyzation.



The screenshot shows the PyCharm IDE with a file named 'CREATE.py' open. The code in the file is as follows:

```
1 # CREATE
2 with open("CATAHAN.txt", "w") as fhand:
3     fhand.write("Your text.\n")
4
5 # ANALYZATION
6 # This creates a text file named "CATAHAN.txt"
7 # In the 3rd line of the code, it inputs "Your text" inside of this text file.
```

Below the code editor, the 'Run' window shows the execution command: `C:\Users\ucc_e\PycharmProjects\PythonProject10\.venv\Scripts\python.exe "C:\Users\ucc_e\PycharmProjects\PythonProject10\CREATE.py"`. The output indicates that the process finished with exit code 0.



The screenshot shows the PyCharm IDE with a file named 'DELETE.py' open. The code in the file is as follows:

```
1 # DELETE
2 open("CATAHAN.txt", "w").close()
3
4 # ANALYZATION
5 # This code remove CATAHAN.txt file
```

Below the code editor, the 'Run' window shows the execution command: `C:\Users\ucc_e\PycharmProjects\PythonProject10\DELETE.py`. The output indicates that the process finished with exit code 0.

The screenshot shows the PyCharm IDE with the 'READ.py' file open. The code in the editor is as follows:

```
1 # READ
2 with open("CATAHAN.txt", "r") as fhand:
3     print("Read:", fhand.read())
4
5 # ANALYZATION
6 # This code open the text file and print the text inside by reading it.
```

The Run tool window at the bottom shows the execution of the script:

```
C:\Users\ucc_e\PycharmProjects\PythonProject10\.venv\Scripts\python.exe "C:\Users\ucc_e\Py
Read:
Process finished with exit code 0
```

The status bar at the bottom indicates the file is 'SUPPLEMENTARY ACTIVITY 2 > READ.py', 6:72, CRLF, UTF-8, 4 spaces, Python 3.12 (PythonProject10).

The screenshot shows the PyCharm IDE with the 'UPDATE.py' file open. The code in the editor is as follows:

```
1 # UPDATE (Append)
2 with open("CATAHAN.txt", "a") as fhand:
3     fhand.write("Appended text.\n")
4
5 # ANALYZATION
6 # This one add another text in the text file.
```

The Run tool window at the bottom shows the execution of the script:

```
C:\Users\ucc_e\PycharmProjects\PythonProject10\.venv\Scripts\python.exe "C:\Users\ucc_e\Py
Process finished with exit code 0
```

The status bar at the bottom indicates the file is '10 > SUPPLEMENTARY ACTIVITY 2 > UPDATE.py', 6:46, CRLF, UTF-8, 4 spaces, Python 3.12 (PythonProject10).

```
1 # READ again to verify update
2 with open("CATAHAN.txt", "r") as fhand:
3     print("Updated Read:\n", fhand.read())
4
5 # ANALYZATION
6 # This code read the file again to check for additional text included in the file
```

PEP 8: W292 no newline at end of file
Reformat the file Alt+Shift+Enter

Run READ AGAIN x

C:\Users\ucc_e\PycharmProjects\PythonProject10\.venv\Scripts\python.exe "C:\Users\ucc_e\Py
Updated Read:
Your text.
Appended text.
Process finished with exit code 0

SUPPLEMENTARY ACTIVITY 2 > READ AGAIN.py 6:59 CRLF UTF-8 4 spaces Python 3.12 (PythonProject10)

8. Conclusions/Observations

By doing this activity, I learned more about how Python deals with files. Opening and writing data to reading, modifying, and deleting content. I understood how to employ functions such as `open()`, `read()`, `write()`, and how to correctly deal with exceptions using `try` and `except` in order to make the program more efficient. It was also fascinating to know how file pointers function and why re-opening a file is needed when reading one again after having already read it. Generally speaking, this task demonstrated how file handling provides a program with persistence so that the program can store and deal with data even when the program finishes.

9. Assessment Rubric