



MAPÚA UNIVERSITY

SCHOOL OF ELECTRICAL, ELECTRONICS, AND COMPUTER ENGINEERING

Experiment 1: Using Software Tools and Code Versioning System

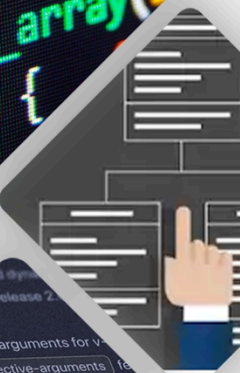
CPE106L (Software Design Laboratory)

JM Ajegram A. Esperanza

Ralph Jed N. Lam

Denise Kate L. Pagala

Group No.: 8
Section: E01



PreLab

Readings, Insights, and Reflection

Fundamentals of Python: First Programs (Lambert, 2018) Basic Python Programming

We worked through Chapter 1 and gained an understanding of the core principles of Python. It began with how Python runs code in two common ways: the interactive interpreter for quick trials and .py scripts for saved programs you can execute from the command line. It highlighted the readability of Python's syntax, where indentation creates code blocks and replaces braces, and where expressions are built from literals, variable names, and operators. The piece explained dynamic typing and showed that variables are just names bound to objects, so the same name can later point to a value of a different type.

The article introduced core data types and how to use them. It covered numbers (int, float), booleans, and strings, including indexing, slicing, and common string methods. It explained collections: lists for ordered, mutable data; tuples for ordered, immutable data; dictionaries for key-value pairs; and sets for unique elements. It also pointed out mutability differences, how copying works, and why that matters for function arguments. Truthiness and comparison operators came up as well, showing how nonzero numbers, nonempty strings, and nonempty containers evaluate in conditionals.

Control flow received a focused treatment. Conditionals use if, elif, and else to select behaviors. Loops are used to iterate over items in a container or a range, and while to repeat until a condition changes. The article showed how break exits a loop, continue skips to the next iteration, and pass acts as a placeholder. It encouraged tracing tiny examples and printing intermediate values to check logic during early learning. Functions were presented as the main unit of reuse. Parameters accept inputs, return sends results back, and docstrings record intent. The article touched on default parameters, simple input validation, and how exceptions surface errors. It showed try/except for handling predictable failures, like converting user input with int() or opening files. A short section on modules explained import, the standard library's value, and how to organize code across files. The article gave simple tips: use print() for quick checks and f-strings for readable output, keep code style consistent with PEP 8 naming and spacing, and add comments that explain "why," not "what." It suggested practicing with small scripts, experimenting in the interpreter, and reading built-in docs with help() to build comfort. Overall, it provided a clear path from running your first line to writing small, well-structured programs.

Member 1: JM Ajegram Esperanza

Package Management Tool: Anaconda & Pyenv

Working with Anaconda made package and environment setup less frustrating because conda created and curated distributions solved some of the problems. Pyenv helped me separate the Python runtime itself from packages, so I could pin exact Python versions per project and avoid the “works on my machine” problem. Together, they taught me to treat the interpreter and the libraries as two layers: pyenv picks the Python version; conda or pip manages packages inside an environment. The main challenge was path management on Windows since I primarily use a Mac OS device. But still then, I documented the install order and verified with `python --version`, and after that, I borrowed a laptop and which python, and switching contexts became predictable.

Working environment creation: Dos/ Linux commands

Using command-line tools gave me a faster, repeatable setup. Creating folders, navigating, and running scripts with `cd`, `mkdir`, `dir/ls`, and `python file.py` felt quicker than clicking around. I learned to think in terms of reproducible steps: a few commands in a README can rebuild the same environment on any machine. The habit of checking the current directory, reading errors carefully, and using history (`↑`) or tab completion saved time and lowered mistakes.

Code Versioning System: GitHub

GitHub showed me that saving code is not enough; tracking history and collaborating is the real value. Branches let me experiment without breaking main, and pull requests created a clear checkpoint for review. Writing meaningful commit messages and small changes helped me explain my work and made debugging easier. I also saw how issues, labels, and a simple README turn a folder of files into a project others can understand and contribute to.

Member 2: Ralph Jed Lam


Package Management Tool: Anaconda & Pyenv

Anaconda and Pyenv are important tools in managing Python environments and packages. During our laboratory, we also installed Anaconda and Pyenv wherein Anaconda streamlines the installation of commonly used libraries and provides an organized environment, particularly useful in data analysis and scientific computing. Meanwhile, Pyenv allows smooth handling of multiple Python versions, reducing conflicts between projects.

Working environment creation: Dos/ Linux commands

DOS and Linux commands are also essential in creating and managing a programming environment. Basic commands such as navigating directories, listing files, and executing scripts illustrate the direct interaction between the programmer and the system. This emphasizes the value of understanding command-line operations as a foundation for programming tasks, system management, and effective workflow organization.

Code Versioning System: GitHub



GitHub is a helpful tool for saving and organizing code. It allows changes to be tracked over time so that older versions can be restored if needed. It also makes teamwork easier, since multiple people can work on the same project without losing progress. GitHub shows how versioning systems keep projects organized and support collaboration in programming.

Member 3: Denise Kate Pagala

Package Management Tool: Anaconda & Pyenv

The Anaconda and Pyenv are tools used for Python versions and libraries to allow developers to create environments for different projects. In our case we use softDesLab to ensure that our projects are organized and placed in one environment only allowing for a smoother project development. I learned that these tools are essential as they can save time and handle different projects with proper package management.

Working environment creation: Dos/ Linux commands

I learned that through the DOS/Linux commands, I can directly interact with the operating system and perform tasks more quickly. At first, the commands felt overwhelming because there were many to remember, but with continued practice, I became more familiar with them, which made typing commands easier and faster. These commands are also essential for creating and managing working environments, as mentioned above.

Code Versioning System: GitHub

Through GitHub, I learned how to organize my code more effectively and keep track of my progress. Its features allowed me to collaborate smoothly with my peers, while the history function made it easier to review and monitor changes. Using GitHub also gave me a sense of what it's like to work as a professional developer, since this platform is widely used in the industry. Furthermore, GitHub plays an important role in version management and teamwork, making it an essential tool for programming projects.

InLab

- **Objectives**

1. Be familiarized with the basic Git functions, such as creating repository and branch, exploring the GitHub ecosystem, and practicing version control operations through the GitHub Learning Lab.
2. Apply fundamental Python programming concepts using the Anaconda prompt as the interactive terminal.

- **Tools Used**

- Git Terminal
- Anaconda & Visual Studio Code

- **Procedure**

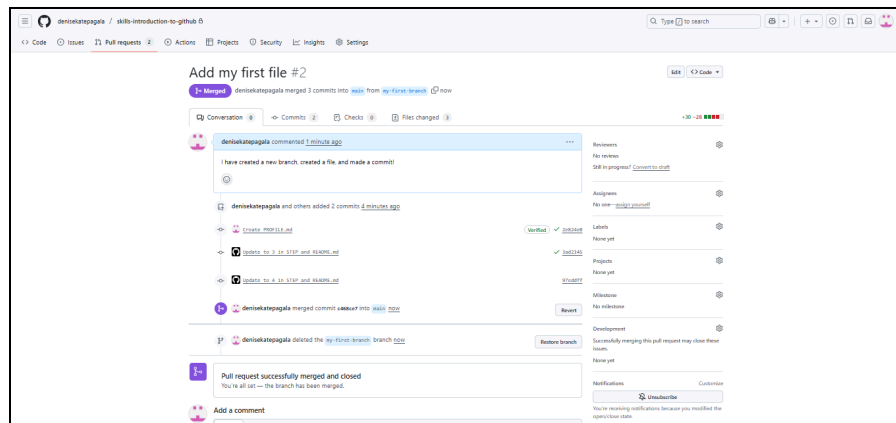
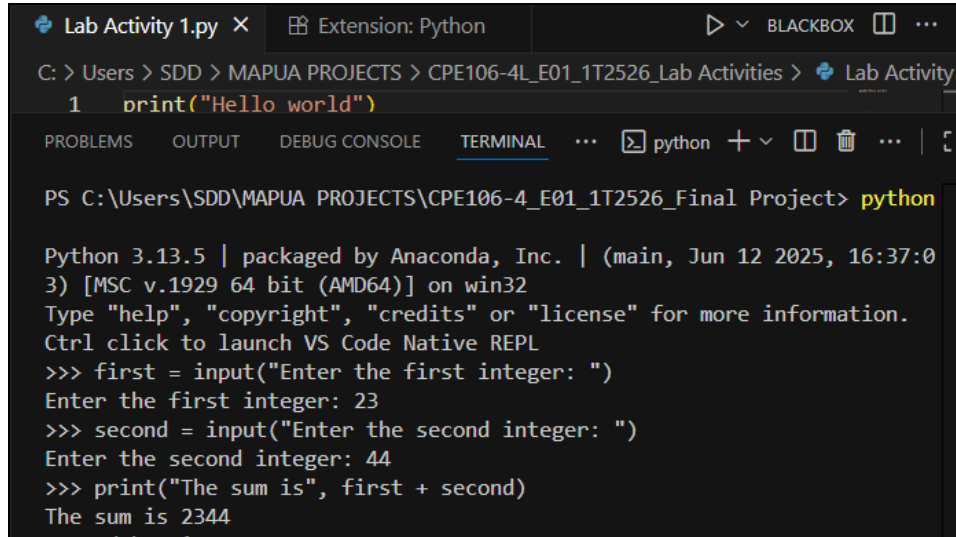


Figure 1. Exploring Introduction to Github

Figure 1 shows the first part of the experiment, wherein we explored how to use the basic functions of GitHub. In this part, we learned how to create a new branch to work on a separate task without affecting the main code, commit a file to save and track changes made in the repository, and open a pull request to propose merging updates. These features are important, especially when collaborating to track all the changes made.



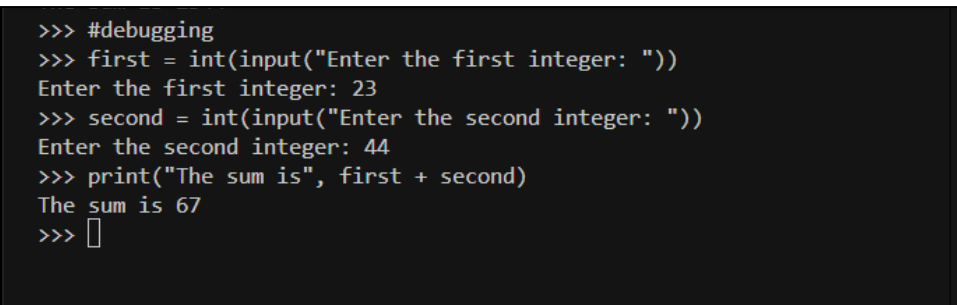
```
Lab Activity 1.py x Extension: Python BLACKBOX ...
C: > Users > SDD > MAPUA PROJECTS > CPE106-4L_E01_1T2526_Lab Activities > Lab Activity
1 print("Hello world")

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL python + - [ ] [ ] ... [ ]

PS C:\Users\SDD\MAPUA PROJECTS\CPE106-4_E01_1T2526_Final Project> python

Python 3.13.5 | packaged by Anaconda, Inc. | (main, Jun 12 2025, 16:37:0
3) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
Ctrl click to launch VS Code Native REPL
>>> first = input("Enter the first integer: ")
Enter the first integer: 23
>>> second = input("Enter the second integer: ")
Enter the second integer: 44
>>> print("The sum is", first + second)
The sum is 2344
```

Figure 2.1. Chapter 1 Debugging Problem in Fundamentals of Python: First Programs



```
>>> #debugging
>>> first = int(input("Enter the first integer: "))
Enter the first integer: 23
>>> second = int(input("Enter the second integer: "))
Enter the second integer: 44
>>> print("The sum is", first + second)
The sum is 67
>>> 
```

Figure 2.2. Chapter 1 Debugging Solution in Fundamentals of Python: First Programs

Figures 2.1 and 2.2 shows the Debugging Problem in Chapter 1 of the book *Fundamental of Python: First Programs*. The problem highlights the use of data types since in Figure 2.1, the output for the variables `first` and `second` is treated as a string, which causes the two values to be concatenated instead of added. In Figure 2.2, the data type `int` is applied, which produces the correct output by performing arithmetic addition rather than string concatenation.

```
PS C:\Users\SDD> python
Python 3.13.1 (tags/v3.13.1:0671451, Dec 3 2024, 19:06:28) [MSC v.1942 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> purchasePrice = float(input("Enter the purchase price as $: "))
Enter the purchase price as $: 5.50
>>> taxRate = int(input("Enter the tax rate as %: "))
Enter the tax rate as %: 5
>>> tax = purchasePrice * taxRate
>>> totalOwed = purchasePrice + tax
>>> print("Purchase price: ", purchasePrice, "\nTax: ", tax, "\nTotal Owed: ", totalOwed)
Purchase price: 5.5
Tax: 27.5
Total Owed: 33.0
```

Figure 3.1. Chapter 1 Debugging Problem in Fundamentals of Python: First Programs

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
PS C:\Users\SDD> python
Python 3.13.1 (tags/v3.13.1:0671451, Dec 3 2024, 19:06:28) [MSC v.1942 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> #debugging
>>> purchasePrice = float(input("Enter the purchase price as $: "))
Enter the purchase price as $: 5.50
>>> taxRateDebug = int(input("Enter the tax rate as %: "))
Enter the tax rate as %: 5
>>> taxDebug = purchasePrice * taxRateDebug/100
>>> totalOwed = purchasePrice + taxDebug
>>> print("Purchase price: ", purchasePrice, "\nTax: ", taxDebug, "\nTotal Owed: ", totalOwed)
Purchase price: 5.5
Tax: 0.275
Total Owed: 5.775
>>> █
```

Figure 3.2. Chapter 1 Debugging Solution in Fundamentals of Python: First Programs

Figures 3.1 and 3.2 displays the Debugging Problem for Chapter 2 in the e-book. The problem demonstrates a logic error, as the expected output should be 0.275 for Tax and 5.775 for Total Owed, but the program instead produces 27.5 for Tax and 33.0 for Total Owed. This happens because the problem requires the tax rate to be expressed as a percentage, which means it must be divided by 100 to yield the correct tax value. Without this adjustment, the terminal interprets the code literally, resulting in the incorrect output.

PostLab

Programming Problem 1: ESPERANZA

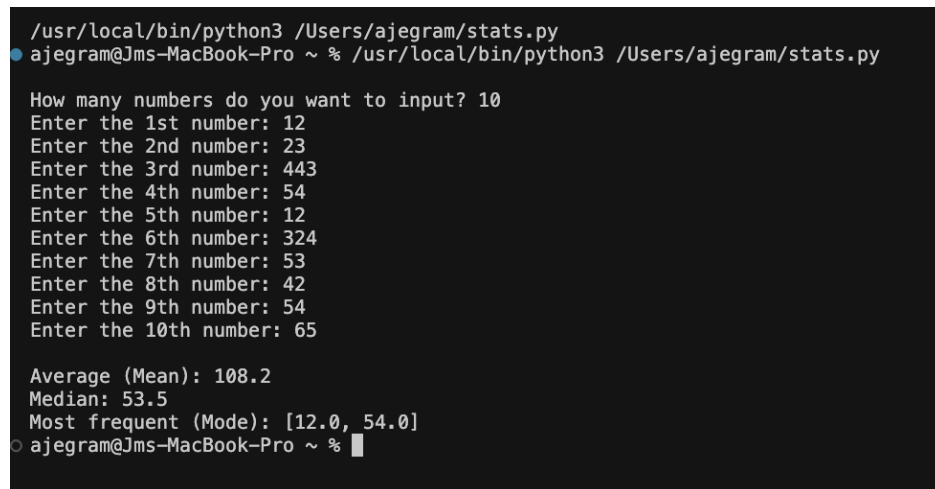
Statisticians would like to have a set of functions to compute the median and mode of a list of numbers. The median is the number that would appear at the midpoint of a list if it were sorted. The mode is the number that appears most frequently in the list. Define these functions in a module named stats.py. Also include a function named mean, which computes the average of a set of numbers. Each function expects a list of numbers as an argument and returns a single number.

```
def mean(nums):.py stats.py x
Users > ajegram > stats.py > ...
1  def mean(data):
2      return sum(data) / len(data)
3
4  def median(data):
5      data.sort()
6      length = len(data)
7      mid_index = length // 2
8      if length % 2 == 0:
9          return (data[mid_index - 1] + data[mid_index]) / 2
10     else:
11         return data[mid_index]
12
13 def Mode(data):
14     if not data:
15         raise ValueError("List is empty.")
16
17     count = {}
18     for value in data:
19         count[value] = count.get(value, 0) + 1
20
21     highest_freq = max(count.values())
22     modes = [value for value, freq in count.items() if freq == highest_freq]
23
24     if len(modes) == len(set(data)):
25         return None
26
27     return modes if len(modes) > 1 else modes[0]
28
29 n_values = int(input("\nHow many numbers do you want to input? "))
30
31 data_list = []
32 for i in range(1, n_values + 1):
33     suffix = "th"
34     if i == 1:
35         suffix = "st"
36     elif i == 2:
37         suffix = "nd"
38     elif i == 3:
39         suffix = "rd"
40
41     number = float(input(f"Enter the {i}-{suffix} number: "))
42     data_list.append(number)
43
44
45 print("\nAverage (Mean):", mean(data_list))
46 print("Median:", median(data_list))
47 print("Most frequent (Mode):", Mode(data_list))
48
```

Figure 4.1 Programming Problem 1 on Visual Studio Code

In this screenshot, you can see the `stats.py` file that calculates basic statistics for a list of numbers. On the left, the functions are clearly defined: `mean()` computes the average by summing all the numbers and dividing by the total count, `median()` finds the median by sorting the list and picking the middle value (or averaging the two middle numbers if there's an even number of elements), and `mode()` calculates the most frequent value, handling multiple modes and returning `None` if all numbers appear equally often.

Below the functions, you can see the section where the user is prompted for input. It first asks how many numbers they want to enter, then uses a loop to collect each value into a list. Finally, at the bottom of the screenshot, the program prints the calculated mean, median, and mode. Looking at this in Visual Studio helps me see the code structure, variable names, and indentation clearly, making it easy to trace the flow of data from input to output. This setup also allows me to test different inputs quickly and visually confirm that the calculations work correctly.



```
/usr/local/bin/python3 /Users/ajegram/stats.py
ajegram@Jms-MacBook-Pro ~ % /usr/local/bin/python3 /Users/ajegram/stats.py

How many numbers do you want to input? 10
Enter the 1st number: 12
Enter the 2nd number: 23
Enter the 3rd number: 443
Enter the 4th number: 54
Enter the 5th number: 12
Enter the 6th number: 324
Enter the 7th number: 53
Enter the 8th number: 42
Enter the 9th number: 54
Enter the 10th number: 65

Average (Mean): 108.2
Median: 53.5
Most frequent (Mode): [12.0, 54.0]
ajegram@Jms-MacBook-Pro ~ %
```

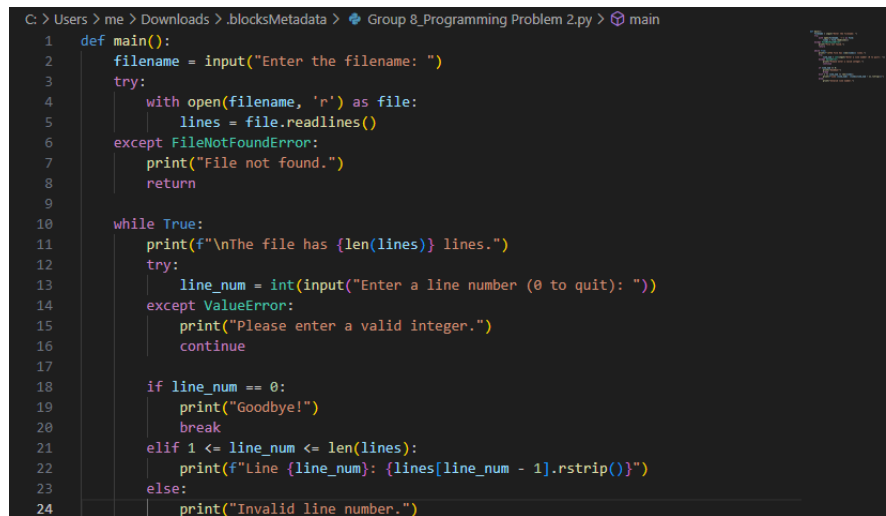
Figure 4.2 Programming Problem 2 Terminal Output

In this run of the program, the user entered a total of 10 numbers: 12, 23, 443, 54, 12, 324, 53, 42, 54, and 65. The program first calculated the average (mean), which was 108.2. This value represents the sum of all the numbers divided by the total count, giving a general measure of the central value of the data. Next, the median was calculated as 53.5. Since the dataset has an even number of elements, the median is the average of the two middle numbers when the data is sorted. This value provides a measure of the middle point of the dataset, which is less affected by extremely high values like 443 and 324. The mode was determined to be [12.0, 54.0]. This indicates that both 12 and 54 appear most frequently in the dataset, each occurring twice. The program correctly identifies multiple modes when there is a tie in frequency.

The output demonstrates that the code successfully calculates key statistical measures such as mean, median, and mode, which shows a clear summary of the data distribution, including handling multiple modes and accounting for outliers in the mean calculation.

Programming Problem 2: LAM

Write a program that allows the user to navigate through the lines of text in a file. The program prompts the user for a filename and inputs the lines of text into a list. The program then enters a loop in which it prints the number of lines in the file and prompts the user for a line number. Actual line numbers range from 1 to the number of lines in the file. If the input is 0, the program quits. Otherwise, the program prints the line associated with that number.

The image shows a screenshot of a Visual Studio Code editor window. The title bar at the top indicates the file path: 'C:\Users\me\Downloads\blocksMetadata\Group 8_Programming Problem 2.py' and the active tab is 'main'. The code is written in Python and is as follows:

```
1 def main():
2     filename = input("Enter the filename: ")
3     try:
4         with open(filename, 'r') as file:
5             lines = file.readlines()
6     except FileNotFoundError:
7         print("File not found.")
8         return
9
10    while True:
11        print(f"\nThe file has {len(lines)} lines.")
12        try:
13            line_num = int(input("Enter a line number (0 to quit): "))
14        except ValueError:
15            print("Please enter a valid integer.")
16            continue
17
18        if line_num == 0:
19            print("Goodbye!")
20            break
21        elif 1 <= line_num <= len(lines):
22            print(f"Line {line_num}: {lines[line_num - 1].rstrip()}")
23        else:
24            print("Invalid line number.")
```

Figure 5.1 Programming Problem 2 on Visual Studio Code

This Python program demonstrates basic file handling along with user interaction. It prompts the user for a filename, reads all lines from the specified file into a list, and allows the user to navigate through the file by entering line numbers. The program then displays the total number of lines and prints the content of the selected line. Input validation ensures that only valid line numbers are accepted, and the program handles errors such as missing files and invalid input gracefully. This exercise reinforces fundamental concepts such as loops, exception handling, and list indexing in Python.

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\me\Downloads\blocksMetadata> cd "c:\Users\me\Downloads\blocksMetadata"; & 'c:\Users\me\AppData\Local\Programs\Python\Python312\python.exe' 'c:\Users\me\.vscode\extensions\ms-python.debugpy-2025.10.0-win32-x64\bundled\libs\debugpy\launcher' '62536' '-' 'c:\Users\me\Downloads\blocksMetadata\Group 8_Programming Problem 2.py'
Enter the filename: Fox.txt

The file has 6 lines.
Enter a line number (0 to quit): 1
Line 1: "The quick brown fox jumps over the lazy dog"

The file has 6 lines.
The file has 6 lines.
Enter a line number (0 to quit): 2
Line 2: is an English-language pangram
The file has 6 lines.
Enter a line number (0 to quit): 2
The file has 6 lines.
Enter a line number (0 to quit): 2
The file has 6 lines.
Enter a line number (0 to quit): 2
Line 2: is an English-language pangram

The file has 6 lines.
Enter a line number (0 to quit): 3
Line 3: æ" a sentence that contains all the letters of the alphabet.

The file has 6 lines.
Enter a line number (0 to quit): 4
Line 4: The phrase is commonly used for touch-typing practice,

The file has 6 lines.
Enter a line number (0 to quit): 5
Line 5: testing typewriters and computer keyboards, displaying examples of fonts,

The file has 6 lines.
Enter a line number (0 to quit): 6
Line 6: and other applications involving text where the use of all letters in the alphabet is desired.

The file has 6 lines.
Enter a line number (0 to quit): 0
Goodbye!
PS C:\Users\me\Downloads\blocksMetadata> []
```

Figure 5.2 Programming Problem 2 Terminal Output

Here is an example of its output on Visual Studio Code. Generally, the output of the program will: (1) Prompt the user to enter a filename. (2) If the file exists, display how many lines are in the file. (3) Repeatedly ask the user for a line number. (4) If the user enters a valid line number, print that line's text. (5) If the user enters 0, print "Goodbye!" and exit. Lastly, (6) if the user enters an invalid number or non-integer, print an error message and prompt again.

Programming Problem 3: PAGALA

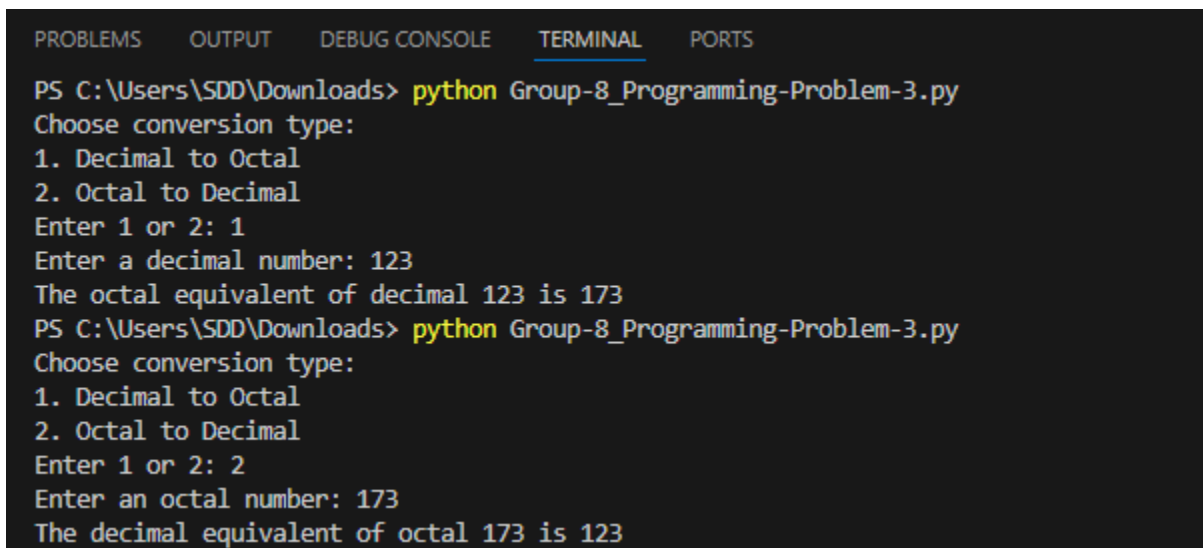
Chapter 4, Programming Problem # 4

VitalSource ebook: Lambert, K. A. (2023). Fundamentals of Python: First Programs (3rd ed.). Cengage Learning US. <https://bookshelf.vitalsource.com/books/9798214406732>

4. Octal numbers have a base of eight and the digits 0–7. Write the scripts

`octaltodecimal.py` and `decimaltooctal.py`, which convert numbers between the octal and decimal representations of integers. These scripts use algorithms that are similar to those of the `binaryToDecimal` and `decimalToBinary` scripts developed in [Section 4-3](#). (LO: 4.1, 4.3)

The Programming Problem 3 highlights the python programming fundamentals, number systems, and base conversions, which are all discussed in the Programming Logic and Design course. The task is to create a python program that converts numbers from octal to decimal or vice versa. Moreover, the program is written in one script instead of separated scripts for easier navigation. The problem helps in understanding number systems and applying programming logic to solve basic mathematical conversions.



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\SDD\Downloads> python Group-8_Programming-Problem-3.py
Choose conversion type:
1. Decimal to Octal
2. Octal to Decimal
Enter 1 or 2: 1
Enter a decimal number: 123
The octal equivalent of decimal 123 is 173
PS C:\Users\SDD\Downloads> python Group-8_Programming-Problem-3.py
Choose conversion type:
1. Decimal to Octal
2. Octal to Decimal
Enter 1 or 2: 2
Enter an octal number: 173
The decimal equivalent of octal 173 is 123
```

Figure 6. Programming Problem 3 Terminal Output

Figure 6 shows the terminal output of Programming Problem 3, wherein the user is asked to choose between the two types of conversion since the code is combined into one script. The decimal is base 10, which uses repeated division by 8 and collects the remainders. In the example above, the decimal number 123 is equal to the octal number 173 since when the numbers are divided by 8, all the remainders will result in 3, 7, and 1, which will be reversed, resulting in 173. On the other hand, the octal is base 8, which means that each digit represents a power of 8 depending on its position. For instance, in the number given above, 173, this will result in 123, which can be explained in the equation below:

$$(1 \times 8^2) + (7 \times 8^1) + (3 \times 8^0) = 123$$