MAPÚA UNIVERSITY
SCHOOL OF ELECTRICAL, ELECTRONICS, AND COMPUTER ENGINEERING

Experiment 2:
Strings, Lists, Tuples, and Dictionaries

CPE106L (Software Design Laboratory)

Danikka Villaron
Jessa Abigaile Tongol
Mark Vincent De Villa
Matthew Benedict Nepomuceno

Group No.: **2**
Section: **E03**

PreLab

Readings, Insights, and Reflection

Insights and Reflections
Fundamentals of Python: First Programs
9780357881132

De Villa (Chapter 4.1 - Accessing Characters and Substrings in Strings)

Chapter 4.1 of the book began with Python string declaration, the characters and their position in the string data structure, and the syntax of using the subscript operator, enabling developers to access nth characters in a string. Following this was a discussion about printing characters with a loop and slicing substrings, which includes the syntax and effects of changing values in the line of code. Additionally, the book presented a sample code that extracts strings in a list with conditions.

Answers to Questions

Question 1

```
1  data = [10, 20, 30]
2
3  print(data[1])
```

```
20


...Program finished with exit code 0
Press ENTER to exit console.
```

B. 20

Question 2

```
1  data = [10, 20, 30]
2
3  print(data[1:3])
```

```
[20, 30]


...Program finished with exit code 0
Press ENTER to exit console.
```

B. [20, 30]

Question 3

```
1  data = [10, 20, 30]
2
3  print(data.index(20))
```

```
1


...Program finished with exit code 0
Press ENTER to exit console.
```

A. 1

Question 4

```
1  data = [10, 20, 30]
2
3  print(data + [40, 50])
```

```
[10, 20, 30, 40, 50]


...Program finished with exit code 0
Press ENTER to exit console.
```

B. [10, 20, 30, 40, 50]

Question 5

```
1  data = [10, 20, 30]
2  data[1] = 5
3  print(data)
```

```
[10, 5, 30]


...Program finished with exit code 0
Press ENTER to exit console.
```

B. [10, 5, 30]

Question 6

```
1  data = [10, 20, 30]
2  data.insert(1,15)
3  print(data)
```

```
[10, 15, 20, 30]


...Program finished with exit code 0
Press ENTER to exit console.
```

C. [10, 15, 20, 30]

Question 7

```
1  info = {
2      "name" : "Sandy",
3      "age" : 17
4  }
5
6  print(list(info.keys()))
```

```
['name', 'age']


...Program finished with exit code 0
Press ENTER to exit console.
```

B. [“name”, “age”]

Question 8

```
1  info = {
2      "name" : "Sandy",
3      "age" : 17
4  }
5
6  print(info.get("hobbies", None))
```

```
None


...Program finished with exit code 0
Press ENTER to exit console.
```

B. None

Question 9
B. pop

Question 10
B. Strings and Tuples

InLab

- **Objectives**
  Examples:
  1. Debug the sample programs on tuples, dictionary, etc.
  2. Use Anaconda (or Python Virtual Environment) and Linux terminal in running python statements.
  3. Use Visual Studio Code in debugging.
  4. Differentiate the structures of Python and C++

- **Tools Used**
  - Anaconda (or Python Virtual Environment)
  - Ubuntu Linux Virtual Machine
  - Visual Studio Code
  - Online GDB

- **Procedure**

For the InLab activity, the group chose Kenneth Lambert's binarytodecimal.py program and a group member's C++ version of binary-to-decimal conversion to present the difference between C++ and Python in their data structures and syntaxes.

```
"""
File: binarytodecimal.py
Converts a string of bits to a decimal integer.
"""

bstring = input("Enter a string of bits: ")
decimal = 0
exponent = len(bstring) - 1
for digit in bstring:
    decimal = decimal + int(digit) * 2 ** exponent
    exponent = exponent - 1
print("The integer value is", decimal)
```

Figure 1.1 binarytodecimal.py of Kenneth Lambert

```
Enter a string of bits: 10110111
The integer value is 183
```

Figure 1.2 binarytodecimal.py Output

The binarytodecimal.py program codes seen in Figure 1.1 by Kenneth Lambert convert a user-input binary string into its corresponding decimal equivalent. It begins by prompting the user for a binary string, then utilizes a for loop to simulate the formula to convert binary strings to a decimal, where the decimal variable obtains the sum of the nth bit multiplied to 2 and raised to n-1 then leaves the for loop and displays the calculated decimal. Figure 1.2 presents a sample conversion of the binary string 10110111 into its corresponding decimal value of 183.

```cpp
#include <iostream>
#include <stack>
#include <queue>
#include <cmath>
using namespace std;
int main()
{
    stack<int> bin1;
    int input, bigness, temp;
    int deci = 0;
    cout << "Enter a binary number and I will output the decimal form.\n\nEnter a number: ";
    cin >> input;
    temp = input;
    while(temp != 0){
        bin1.push(temp % 10);
        temp /= 10;
    }
    bigness = bin1.size();
    for(int i = bigness - 1; i > -1; i--){
        temp = bin1.top();
        if(temp == 1){
            deci += pow(2, i);
        }
        bin1.pop();
    }
    cout << "\nThe decimal form of " << input << " is " << deci;
    return 0;
}
```

Figure 1.3 Group's C++ Version of Binary-To-Decimal Conversion

In the C++ version, firstly we would need to include some libraries to make the code work such as iostream, stack, and cmath. These are needed to make the program work. Next, we initialize some variables to help us in storing some data.

input is used for the original Binary Input

bigness is used to store the length of the Binary number once it is put into a stack

temp is used to store the binary number as well but it is temporary since it will be manipulated

deci is initialized to be equal to 0 because this is where we will accumulate the value of the final decimal number

After the initialization of variables, temp is given the value of input (the binary number) so we can use temp to push the binary number into a stack. This is done using the while loop shown above by pushing the first digit, which is taken by getting the modulo value of the number and then dividing the int value by 10 to remove that digit and then repeating this process until we are left with 0. Then, once we have the binary number in a stack, we initialize a for loop so we can loop

through each digit in the stack and determine its value from binary to decimal form and adding it to the deci variable if the digit was 1 and not 0, this is done using the line, “deci += pow(2,i)”.

After the for loop is done, deci is now storing the value for the final decimal value.

```
Enter a binary number and I will output the decimal form.

Enter a number: 10110111

The decimal form of 10110111 is 183
```

Figure 1.4 Group’s C++ Version of Binary-To-Decimal Conversion Output

Figure 1.4 presents the output of the group's C++ version of the conversion, and the binary string used in Figure 1.2 is identical to test the correctness of the code's calculation, presenting the same results, which implies that the codes with the same input will have the same output despite differing in the program's code and language.

Comparing the data structures of Python and C++ based on the programs in Figure 1.1 and 1.3

The two languages differ in the data structures with converting a bit string to a decimal value. The C++ version utilizes a stack data structure to hold the bits of the user-input bit string. The Python version utilizes a string and a subscript operator to access and store the bit string. The C++ version uses a for loop to calculate and pop the values of the stack, ending when the loop completes n iterations. The Python version also utilizes a for loop to calculate the decimal equivalent of the bit string, ending when the number of iterations equals the length of the bit string.

PostLab

Programming Problems

1. Filename: stats.py

A group of statisticians at a local college has asked you to create a set of functions that compute the median and mode of a set of numbers, as defined in the below sample programs:

- mode.py
- median.py

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 should expect a list of numbers as an argument and return a single number. Each function should return 0 if the list is empty. Include a main function that tests the three statistical functions with a given list.

```
def mode():
    f = open(fileName, 'r')
    words = []
    for line in f:
        wordsInLine = line.split()
        for word in wordsInLine:
            words.append(word.upper())

    theDictionary = {}
    for word in words:
        number = theDictionary.get(word, None)
        if number == None:
            theDictionary[word] = 1
        else:
            theDictionary[word] = number + 1

    theMaximum = max(theDictionary.values())
    for key in theDictionary:
        if theDictionary[key] == theMaximum:
            print("The mode is", key)
            break


def median():
    f = open(fileName, 'r')

    numbers = []
    for line in f:
        words = line.split()
        for word in words:
            numbers.append(float(word))

    numbers.sort()
    midpoint = len(numbers) // 2
    print("The median is", end=" ")
    if len(numbers) % 2 == 1:
        print(numbers[midpoint])
    else:
        print((numbers[midpoint] + numbers[midpoint - 1]) / 2)


def mean():
    f = open(fileName, 'r')

    numbers = []
    for line in f:
        words = line.split()
        for word in words:
            numbers.append(float(word))
    print("The mean is", end=" ")
    print(sum(numbers)/(len(numbers)))

x = 1

while x <= 3:
    print("Trial",x)
    fileName = input("Enter the file name: ")
    mean()
    median()
    mode()
    print("\n")
    x+=1
```

Figure 2.1. Definition of mode.py, median.py, and mean function as a python module

```
Trial 1
Enter the file name: list1.txt
The mean is 36.46153846153846
The median is 25.0
The mode is 10


Trial 2
Enter the file name: list2.txt
The mean is 43.38461538461539
The median is 61.0
The mode is 68


Trial 3
Enter the file name: list3.txt
The mean is 0.0
The median is 0.0
The mode is 0
```

Figure 2.2. Output of main program testing three lists of numbers in separate files

Figure 2.1 presents the module that defines the given mode.py and median.py with a defined function obtaining the listed numbers' mean. The mode function begins by reading the written numbers in a file and sorting them in a Dictionary that counts the frequency of mentioning numbers in a file before returning the number with the highest frequency, essentially the list's mode. The median function also reads the file and sorts the numbers in a list in ascending order before obtaining the value at the middle, wherein the value at the midpoint is returned if there is an odd number of values and returns the lower middle when there are even numbers of values in the list. The mean function also reads the file, counting the number of values in the list, obtaining their sum, and dividing by the number of values present. Additionally, all these functions return the value of 0 when given an empty set.

Figure 2.2 presents the code's output and the values obtained after calling the module looped thrice. Trials 1 and 2 provide the mean, median, and mode of the list of numbers, whereas trial 3 had no values other than zero present.

2. Filename: LR2_2.py

Write a program that allows the user to navigate the lines of text in a file. The program should prompt the user for a filename and input 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.

```
"""
Filename: LR2_2.py

Allows users to navigate and view lines from a file
"""

filename = input("Enter the filename: ")

try:
    with open(filename, 'r') as file:
        lines = file.readlines()
        no_of_lines = len(lines)
    print(f"There are '{no_of_lines}' lines in the file...")

except FileNotFoundError:
    print("File cannot be found")

except Exception as e:
    print(f"Error: {e}")

else:
    while True:
        try:
            target_line = int(input("Enter a line number to read, enter 0 to exit the program: "))

        except ValueError:
            print(f"Please enter an integer\n")
            continue

        if target_line < 0:
            print("Please input a positive integer\n")
            continue

        elif target_line > no_of_lines:
            print(f"Please try a number below '{no_of_lines}'\n")
            continue

        elif target_line == 0:
            print("Leaving the program")
            break

        else:
            print(lines[target_line-1])
            continue
```

Figure 3.1. Code that enables users to navigate and view lines in a separate file

```
Enter the filename: text
File cannot be found
```

Figure 3.2. Output when filename is invalid

```
Enter the filename: text.txt
There are '3' lines in the file...
Enter a line number to read, enter 0 to exit the program: a
Please enter an integer

Enter a line number to read, enter 0 to exit the program: 1
Hello, how are you?

Enter a line number to read, enter 0 to exit the program: 2
Hallo, wie geht's?

Enter a line number to read, enter 0 to exit the program: 3
こんにちは，元気?

Enter a line number to read, enter 0 to exit the program: 4
Please try a number below '3'

Enter a line number to read, enter 0 to exit the program: -1
Please input a positive integer

Enter a line number to read, enter 0 to exit the program: 0
Leaving the program
```

Figure 3.3. Output with valid and invalid line numbers

```
Hello, how are you?
Hallo, wie geht's?
こんにちは，元気?
```

Figure 3.4 Content of text.txt

Figure 3.1 presents the program's code beginning with prompting the user for a valid file to read, then verified by a try-except block for the file's accessibility, followed by the code that only runs when the filename is valid or leaves the program when invalid, ensured by the else statement

included in the try-except blocks. The looping statement that follows prompts the user for an integer in the range of zero to the max number of lines in the provided file where another try-except block verifies the validity of the input - testing whether the input is an integer or not - continuously looping until zero becomes an input in the prompt, and displays the line of the corresponding line number when the numbers are valid and presents an error message before prompting the user again when a value is not valid.

Figure 3.2 presents the corresponding error message when the filename is invalid before leaving the program. Figure 3.3 presents the outputs of both valid and invalid values in a prompt, and Figure 3.4 presents the texts in the text.txt file read by the program in the sample.

3. Filename: generator_modified.py

Modify the sentence-generator program of Case Study 5.3:

· METIS book: 9781337671019, page 150. · Python source code: generator.py

so that it inputs its vocabulary from a set of text files at startup. The filenames are nouns.txt, verbs. txt, articles.txt, and prepositions.txt. (Hint: Define a single new function, getWords. This function should expect a filename as an argument. The function should open an input file with this name, define a temporary list, read words from the file, and add them to the list. The function should then convert the list to a tuple and return this tuple. Call the function with an actual filename to initialize each of the four variables for the vocabulary.)

```python
import random

with open("nouns.txt",'r') as file:
    nouns = [line.strip() for line in file]

with open("verbs.txt", 'r') as file:
    verbs = [line.strip() for line in file]

with open("articles.txt",'r') as file:
    articles = [line.strip() for line in file]

with open("prepositions.txt", 'r') as file:
    prepositions = [line.strip() for line in file]

def sentence():
    return nounPhrase() + " " + verbPhrase()

def nounPhrase():
    return random.choice(articles) + " " + random.choice(nouns)

def verbPhrase():
    return random.choice(verbs) + " " + nounPhrase() + " " + prepositionalPhrase()

def prepositionalPhrase():
    return random.choice(prepositions) + " " + nounPhrase()

def main():
    while True:

        try:
            number = int(input("Enter the number of sentences: "))

        except ValueError:
            print(f"Please enter an integer\n")
            continue

        if number < 0:
            print(f"Please input a positive integer\n")
            continue

        elif number == 0:
            print("No sentences will be formed since input is 0")
            break

        else:
            break

    for count in range(number):
        print(sentence())

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

Figure 4.1. Modified generator.py code

```
Enter the number of sentences: a
Please enter an integer

Enter the number of sentences: -1
Please input a positive integer

Enter the number of sentences: 0
No sentences will be formed since input is 0
```

Figure 4.2. Possible messages for inputs that don't generate sentences

```
Enter the number of sentences: 5
a city bit the girl at a city
the street brought a washerman aboard the teacher
the city began a boy before a boy
the country caught the doctor aboard a teacher
a boy was the student within a man
```

Figure 4.3. Sample generation of five sentences

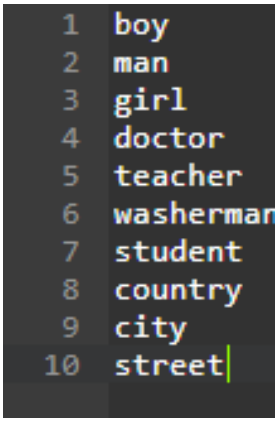

Figure 4.4. Content of nouns.txt

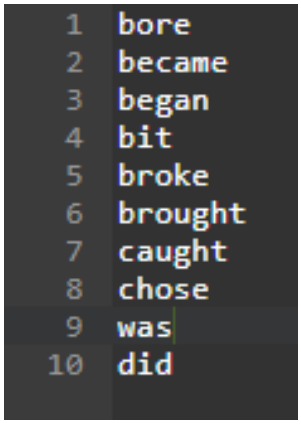

Figure 4.5.Content of verbs.txt

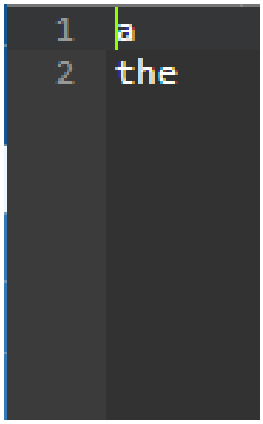

Figure 4.6. Content of articles.txt

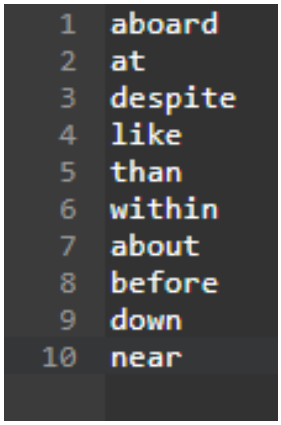

Figure 4.7. Content of prepositions.txt

Figure 4.1 presents Kenneth Lambert's sentence generator with modified sources of the four lists to a .txt file. The program begins by importing the random module and list declarations similar to the original, differing from the source of the list values. Followed by the declaration of functions that form the sentences and the main() function, which contains a try-except block to ensure that the inputs are valid. Figure 4.2 presents the possible inputs caught by the try-except block and an if statement to end the program without unnecessary generation when zero becomes the input. When the input value is greater than zero, the looping statement from the original runs to generate the n number of sentences as requested by the user, with Figure 4.3 being an example of five sentences generated by the program. Figures 4.4, 4.5, 4.6, and 4.6 present the possible words that can be part of the randomly generated sentences.