UNIVERSITY OF BIRMINGHAM DUBAI

LM Software Workshop 1 (34153, 34182, 34168, 36990)
Lab Exercise Sheet
Week 4 Chapter 5 (Lists and Dictionaries)

A. Apply (copy/paste) all the program examples of ppt slides into the Jupyter Notebook, remove errors (if any), and run the programs

B. The following 3 questions are mandatory for all students:

1. 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 Section 5.4. (screenshot as below) 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.

Mean:
The mean is also known as the average. It is calculated by adding up all the values in a dataset and dividing by the total number of values.
Example:
For example, in the dataset [3, 5, 1, 8, 6], the mean is (3 + 5 + 1 + 8 + 6) / 5 = 4.6.

Median:
The median is the middle value in a dataset when the values are arranged in ascending or descending order. If there is an even number of data points, the median is the average of the two middle values.
Example:
For the dataset [3, 5, 1, 8, 6], when arranged in ascending order [1, 3, 5, 6, 8], the median is 5 because it is the middle value.

Mode:
The mode is the value(s) in a dataset that occur with the highest frequency. In a dataset, there can be one mode (unimodal), more than one mode (multimodal), or no mode at all if all values occur with the same frequency.
Example:
In the dataset [3, 5, 1, 8, 6, 3, 1, 1, 8, 6], the modes are 1 and 3 because they occur more frequently than the other values.

Example: Using a List to Find the Median of a Set of Numbers

Researchers who do quantitative analysis are often interested in the **median** of a set of numbers. For example, the U.S. government often gathers data to determine the median family income. Roughly speaking, the median is the value that is less than half the numbers in the set and greater than the other half. If the number of values in a list is odd, the median of the list is the value at the midpoint when the set of numbers is sorted; otherwise, the median is the average of the two values surrounding the midpoint. Thus, the median of the list [1, 3, 3, 5, 7] is 3, and the median of the list [1, 2, 4, 4] is also 3. The following script inputs a set of numbers from a text file and prints their median:

```
"""
File: median.py
Prints the median of a set of numbers in a file.
"""

fileName = input("Enter the filename: ")
f = open(fileName, 'r')

# Input the text, convert it to numbers, and
# add the numbers to a list
numbers = []
for line in f:
    words = line.split()
    for word in words:
        numbers.append(float(word))

# Sort the list and print the number at its midpoint
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)
```

Note that the input process is the most complex part of this script. An accumulator list, **numbers**, is set to the empty list. The **for** loop reads each line of text and extracts a list of words from that line. The nested **for** loop traverses this list to convert each word to a number. The **list** method **append** then adds each number to the end of **numbers**, the accumulator list. The remaining lines of code locate the median value. When run with an input file whose contents are

```
3 2 7
8 2 1
5
```

the script produces the following output:

```
The median is 3.0
```

2. 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.

3. Modify the sentence-generator program of Case Study 5.3 (screenshot as below) 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.)

CASE STUDY: Generating Sentences

Can computers write poetry? We'll attempt to answer that question in this case study by giving a program a few words to play with.

Request

Write a program that generates sentences.

Analysis

Sentences in any language have a structure defined by a **grammar**. They also include a set of words from the **vocabulary** of the language. The vocabulary of a language like English consists of many thousands of words, and the grammar rules are quite complex. For the sake of simplicity our program will generate sentences from a simplified subset of English. The vocabulary will consist of sample words from several parts of speech, including nouns, verbs, articles, and prepositions. From these words, you can build noun phrases, prepositional phrases, and verb phrases. From these constituent phrases, you can build sentences. For example, the sentence "The girl hit the ball with the bat" contains three noun phrases, one verb phrase, and one prepositional phrase. Table 5-3 summarizes the grammar rules for our subset of English.

Phrase	Its Constituents
Sentence	Noun phrase + Verb phrase
Noun phrase	Article + Noun
Verb phrase	Verb + Noun phrase + Prepositional phrase
Prepositional phrase	Preposition + Noun phrase

Table 5-3 The grammar rules for the sentence generator

The rule for **Noun phrase** says that it is an **Article** followed by (+) a **Noun**. Thus, a possible noun phrase is "the bat." Note that some of the phrases in the left column of Table 5-3 also appear in the right column as constituents of other phrases. Although this grammar is much simpler than the complete set of rules for English grammar, you should still be able to generate sentences with quite a bit of structure.

The program will prompt the user for the number of sentences to generate. The proposed user interface follows:

```
Enter the number of sentences: 3
THE BOY HIT THE BAT WITH A BOY
```

```
THE BOY HIT THE BALL BY A BAT
THE BOY SAW THE GIRL WITH THE GIRL

Enter the number of sentences: 2
A BALL HIT A GIRL WITH THE BAT
A GIRL SAW THE BAT BY A BOY
```

Design

Of the many ways to solve the problem in this case study, perhaps the simplest is to assign the task of generating each phrase to a separate function. Each function builds and returns a string that represents its phrase. This string contains words drawn from the parts of speech and from other phrases. When a function needs an individual word, it is selected at random from the words in that part of speech. When a function needs another phrase, it calls another function to build that phrase. The results, all strings, are concatenated with spaces and returned.

The function for **Sentence** is the easiest. It just calls the functions for **Noun phrase** and **Verb phrase** and concatenates the results, as in the following:

```
def sentence():
    """Builds and returns a sentence."""
    return nounPhrase() + " " + verbPhrase() + "."
```

The function for **Noun phrase** picks an article and a noun at random from the vocabulary, concatenates them, and returns the result. We assume that the variables **articles** and **nouns** refer to collections of these parts of speech and develop these later in the design. The function **random.choice** returns a random element from such a collection.

```
def nounPhrase() :
    """Builds and returns a noun phrase."""
    return random.choice(articles) + " " + random.choice(nouns)
```

The design of the remaining two phrase-structure functions is similar.

The **main** function drives the program with a count-controlled loop:

```
def main():
   """Allows the user to input the number of sentences to generate."""
   number = int(input("Enter the number of sentences: "))
   for count in range(number):
       print(sentence())
```

The variables **articles** and **nouns** used in the program's functions refer to the collections of actual words belonging to these two parts of speech. Two other collections, named **verbs** and **prepositions**, also will be used. The data structure used to represent a collection of words should allow the program to pick one word at random.

Because the data structure does not change during the course of the program, you can use a tuple of strings. Four tuples serve as a common pool of data for the functions in the program and are initialized before the functions are defined.

Implementation (Coding)

When functions use a common pool of data, you should define or initialize the data before the functions are defined. Thus, the variables for the data are initialized just below the `import` statement.

```
"""
Program: generator.py
Author: Ken
Generates and displays sentences using simple grammar
and vocabulary. Words are chosen at random.
"""

import random

# Vocabulary: words in 4 different parts of speech
articles     = ("A", "THE")
nouns        = ("BOY", "GIRL", "BAT", "BALL")
verbs        = ("HIT", "SAW", "LIKED")
prepositions = ("WITH", "BY")

def sentence():
    """Builds and returns a sentence."""
    return nounPhrase() + " " + verbPhrase()

def nounPhrase():
    """Builds and returns a noun phrase."""
    return random.choice(articles) + " " + random.choice(nouns)

def verbPhrase():
    """Builds and returns a verb phrase."""
    return random.choice(verbs) + " " + nounPhrase() + " " + \
           prepositionalPhrase()

def prepositionalPhrase():
    """Builds and returns a prepositional phrase."""
    return random.choice(prepositions) + " " + nounPhrase()

def main():
    """Allows the user to input the number of sentences
    to generate."""
    number = int(input("Enter the number of sentences: "))
```

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

# The entry point for program execution
if __name__ == "__main__":
    main()
```

Testing

Poetry it's not, but testing is still important. The functions developed in this case study can be tested in a bottom-up manner. To do so, you must initialize the data first. Then you can run the lowest-level function, **nounPhrase**, immediately to check its results, and you can work up to sentences from there.

On the other hand, testing can also follow the design, which took a top-down path. You might start by writing headers for all of the functions and simple **return** statements that return the functions' names. Then you can complete the code for the **sentence** function first, test it, and proceed downward from there. The wise programmer can also mix bottom-up and top-down testing as needed.

C. The following question is optional

A file concordance tracks the unique words in a file and their frequencies. Write a program that displays a concordance for a file. The program should output the unique words and their frequencies in alphabetical order. Variations are to track sequences of two words and their frequencies, or n words and their frequencies.