National University of Computer and Emerging Sciences

Lab Manual 03
CL461-Artificial Intelligence Lab

Course Instructor	Dr. Kashif Zafar
Lab Instructor (s)	Muhammad Saddam Ahmad Abdullah
Section	D
Semester	Spring 2021

Department of Computer Science
FAST-NU, Lahore, Pakistan

Table of Contents

1 Objectives

After performing this lab, students shall be able to understand Python data structures which include:

- ✓ Python sets
- ✓ Python exception handling
- ✓ Python file handling
- ✓ Python iterators

2 Task Distribution

Total Time	**170 Minutes**
Python Sets	20 Minutes
Python Exception Handling	20 Minutes
Python File Handling	20 Minutes
Python Iterators	10 Minutes
Exercise	90 Minutes
Online Submission	10 Minutes

3 Python Sets

Sets have following characteristics:

- Set in Python is a data structure equivalent to sets in mathematics.
- Sets are a mutable collection of distinct (unique) immutable values that are unordered.
- Any immutable data type can be an element of a set: a number, a string, a tuple.
- Mutable (changeable) data types cannot be elements of the set.
- In particular, list cannot be an element of a set (but tuple can), and another set cannot be an element of a set.
- You can perform standard operations on sets (union, intersection, difference).

3.1 Set Initialization Examples

You can initialize a set in the following ways:

```
# Initialize empty set
emptySet = set()

# Pass a list to set() to initialize it
dataScientist = set(['Python', 'R', 'SQL', 'Git', 'Tableau', 'SAS'])
dataEngineer = set(['Python', 'Java', 'Scala', 'Git', 'SQL', 'Hadoop']
)
```

```
# Direct initialization using curly braces
dataScientist = {'Python', 'R', 'SQL', 'Git', 'Tableau', 'SAS'}
dataEngineer = {'Python', 'Java', 'Scala', 'Git', 'SQL', 'Hadoop'}

# Curly braces can only be used to initialize a set containing values
emptyDict= {}
```

3.2 Set Modification Examples

Let's consider the following set for our add/remove examples:

```
# Initialize set with values
graphicDesigner = {'InDesign', 'Photoshop', 'Acrobat', 'Premiere', 'Br
idge'}

# Add a new immutable element to the set
graphicDesigner.add('Illustrator')

# TypeError: unhasable type 'list'
graphicDesigner.add(['Powerpoint', 'Blender'])

# Remove an element from the set
graphicDesigner.remove('Illustrator')

# Another way to remove an element. What is the difference?
graphicDesigner.discard('Premiere')

# Remove and return an arbitrary value from a set
graphicDesigner.pop()

# Remove all values from the set
graphicDesigner.clear()
```

3.3 Set Operations

Python sets have methods that allow you to perform these mathematical operations like union, intersection, difference, and symmetric difference.
Let's initialize two sets to work on our examples:

```
# Initialize sets
dataScientist = set(['Python', 'R', 'SQL', 'Git', 'Tableau', 'SAS'])
dataEngineer = set(['Python', 'Java', 'Scala', 'Git', 'SQL', 'Hadoop']
)

# set built-in function union
dataScientist.union(dataEngineer)

# Equivalent Result
```

```
dataScientist | dataEngineer

# Intersection operation
dataScientist.intersection(dataEngineer)

# Equivalent Result
dataScientist & dataEngineer

# These sets have elements in common so isdisjoint would return False
dataScientist.isdisjoint(dataEngineer)

# Difference Operation
dataScientist.difference(dataEngineer)

# Equivalent Result
dataScientist – dataEngineer

# Symmetric Difference Operation
dataScientist.symmetric_difference(dataEngineer)

# Equivalent Result
dataScientist ^ dataEngineer
```

3.4 Frozensets

You have encountered nested lists and tuples. The problem with nested sets is that you cannot normally have nested sets as sets cannot contain mutable values including sets.

- A frozenset is very similar to a set except that a frozenset is immutable.
- The primary reason to use them is to write clearer, functional code.
- By defining a variable as a frozen set, you're telling future readers: do not modify this.
- If you want to use a frozen set you'll have to use the function to construct it. No other way.

```
# Nested Lists and Tuples
nestedLists = [['the', 12], ['to', 11], ['of', 9], ['and', 7], ['that'
, 6]]
nestedTuples = (('the', 12), ('to', 11), ('of', 9), ('and', 7), ('that
', 6))

# Initialize a frozenset
immutableSet = frozenset()

# Initialize a frozenset
nestedSets = set([frozenset()])
```

A major disadvantage of a frozenset is that since they are immutable, it means that you cannot add or remove values.

```
# AttributeError: 'frozenset' object has no attribute 'add'
immutableSet.add("Strasbourg")
```

4 Python Exception Handling

An exception is an error that is thrown by our code when the execution of the code results in an unexpected outcome. Normally, an exception will have an error type and an error message. Some examples are as follows.

```
ZeroDivisionError: division by zero
TypeError: must be str, not int
```

ZeroDivisionError and TypeError are the error type and the text that comes after the colon is the error message. The error message usually describes the error type.

4.1 Types of Exceptions

Here's a list of the common exceptions you'll come across in Python:

1. **ImportError**: It is raised when you try to import the library that is not installed or you have provided the wrong name
2. **IndexError:** Raised when an index is not found in a sequence. For example, if the length of the list is 10 and you are trying to access the 11th index from that list, then you will get this error
3. **IndentationError**: Raised when indentation is not specified properly
4. **ZeroDivisionError**: It is raised when you try to divide a number by zero
5. **ValueError**: Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified
6. **Exception**: Base class for all exceptions. If you are not sure about which exception may occur, you can use the base class. It will handle all of them.

4.2 Exception Handling with Try Except Clause

Python provides us with the try except clause to handle exceptions that might be raised by our code. The basic anatomy of the try except clause is as follows:

```
try:
    // some code
except:
    // what to do when the code in try raises an exception
```

In plain English, the try except clause is basically saying, "Try to do this, except (otherwise) if there's an error, then do this instead".

There are a few options on what to do with the thrown exception from the `try` block. Let's discuss them.

4.3 Re-raise the exception

Let's take a look at how to write the try except statement to handle an exception by re-raising it. First, let's define a function that takes two input arguments and returns their sum.

```
def myfunction(a, b):
    return a + b
```

Next, let's wrap it in a try except clause and pass input arguments with the wrong type so the function will raise the TypeError exception.

```
try:
    myfunction(100, "one hundred")
except:
    print('error')
    raise

  Output:

  raiseTraceback (most recent call last):
  File "<input>", line 2, in <module>
  File "<input>", line 2, in myfunction
  TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

4.4 Catch certain types of exception

Another option is to define which exception types we want to catch specifically. To do this, we need to add the exception type to the `except` block.

```
try:
    myfunction(100, "one hundred")
except TypeError:
    print("Cannot sum the variables. Please pass numbers only.")

Output:
Cannot sum the variables. Please pass numbers only.
```

To make it even better, we can actually log or print the exception itself.

```
try:
    myfunction(100, "one hundred")
except TypeError as e:
    print(f"Cannot sum the variables. The exception was: {e}")

Output:
Cannot sum the variables. The exception was: unsupported operand
type(s) for +: 'int' and 'str'
```

Furthermore, we can catch multiple exception types in one `except` clause if we want to handle

those exception types the same way. Let's pass an undefined variable to our function so that it will raise the `NameError`. We will also modify the `except` block to catch both `TypeError` and `NameError` and process either exception type the same way.

```
try:
    myfunction(100, a)
except (TypeError, NameError) as e:
    print(f"Cannot sum the variables. The exception was {e}")

Output:
Cannot sum the variables. The exception was name 'a' is not defined
```

4.5 Try….Finally

So far the try statement had always been paired with except clauses. But there is another way to use it as well. The try statement can be followed by a **finally** clause. Finally clauses are called clean-up or termination clauses, because they must be executed under all circumstances, i.e. a "finally" clause is always executed regardless if an exception occurred in a try block or not. A simple example to demonstrate the finally clause:

```
try:
    x = float(input("Your number: "))
    inverse = 1.0 / x
finally:
    print("There may or may not have been an exception.")
print("The inverse: ", inverse)
```

```
Your number: 34
There may or may not have been an exception.
The inverse:  0.029411764705882353
```

4.6 Try..except and finally

"finally" and "except" can be used together for the same try block, as it can be seen in the following Python example:

```
try:
    x = float(input("Your number: "))
    inverse = 1.0 / x
except ValueError:
    print("You should have given either an int or a float")
except ZeroDivisionError:
    print("Infinity")
```

```
finally:
    print("There may or may not have been an exception.")
```

```
Your number: 23
There may or may not have been an exception.
```

5 Python File Handling

Python allows users to handle files by supporting to read and write files, along with many other file handling options. More details can be learnt here

5.1 Open & Close a file

When you want to read or write a file, the first thing to do is to open the file. Python has a built-in function **open** that opens the file and returns a file object. To return a file object we use open() function along with two arguments, that accepts file name and the mode, whether to read or write.

The syntax is given below:

open(filename, mode)

5.2 Kinds of modes

There are three basic types of modes in which files can be opened in Python.

mode	meaning
r	open for reading (default)
r+	open for both reading and writing (file pointer is at the beginning of the file)
w	open for writing (truncate the file if it exists)
w+	open for both reading and writing (truncate the file if it exists)
a	open for writing (append to the end of the file if exists & file pointer is at the end of the file)

Always keep in mind that the mode argument is not mandatory. If not passed, then Python will assume it to be " **r** " by default.
Let's look at this program and try to analyze how the read mode works:

```
# a file named "book", will be opened with the reading mode.
file = open('book.txt', 'r')
# This will print every line one by one in the file
for each in file:
```

```
    print (each)
```

5.3 Working of read() mode

There is more than one way to read a file in Python. If you need to extract a string that contains all characters in the file then we can use **file.read()**. The full code would work like this:

```
# Python code to illustrate read() mode
file = open("file.text", "r")
print (file.read())
```

Another way to read a file is to call a certain number of characters like in the following code the interpreter will read the first five characters of stored data and return it as a string:

```
# Python code to illustrate read() mode character wise
file = open("file.txt", "r")
print (file.read(5))
```

5.4 Working of write() mode

Let's see how to create a file and how write mode works:
To manipulate the file, write the following in your Python environment:

```
# Python code to create a file
file = open('book.txt','w')
file.write("This is the write command")
file.write("It allows us to write in a particular file")
file.close()
```

The close() command terminates all the resources in use and frees the system of this particular program.

5.5 Working of append() mode

```
# Python code to illustrate append() mode
file = open('book.txt','a')
file.write("This will add this line")
file.close()
```

6 Python Iterators

An iterator is an object that contains a countable number of values. It is an object that can be iterated upon, meaning that you can traverse through all the values. Technically, in Python, an iterator is an object which implements the iterator protocol, which consist of the methods iter() and next().

Every time you ask an iterator for the **next** item, it calls its `__next__` method. If there is another value available, the iterator returns it. If not, it raises a `StopIteration` exception. More information about iterators can be found here.

This behavior (only returning the next element when asked to) has two main advantages:

1. Iterators need less space in memory. They remember the last value and a rule to get to the next value instead of memorizing every single element of a (potentially very long) sequence.
2. Iterators don't check how long the sequence they produce might get. For instance, they don't need to know how many lines a file has or how many files are in a folder to iterate through them.

(One important note: don't confuse iterators with iterables. Iterables are objects that can create iterators by using their `__iter__` method)

6.1 Building Custom Iterators

Building an iterator from scratch is easy in Python. We just have to implement the `__iter__()` and the `__next__()` methods.
The `__iter__()` method returns the iterator object itself. If required, some initialization can be performed.

The `__next__()` method must return the next item in the sequence.

Return an iterator from a tuple, and print each value:

```
mytuple = ("apple", "banana", "cherry")
myit = iter(mytuple)

print(next(myit))
print(next(myit))
print(next(myit))
```

To iterate the characters of a string:

```
mystr = "banana"

for x in mystr:
  print(x)
```

The `for` loop actually creates an iterator object and executes the next() method for each loop.

7 Exercise (50 marks)

7.1 compute the probabilities of each word from text file

You are given four text files; your task is to compute the probabilities of each word from the corpus/data. The "data.txt" can have any type of text in it, we only want to compute the probability of the words that have English alphabets only.

Probability can be computed as:

$$\text{Probability (word)} = \frac{count(word)}{Total\ Words}$$

Sample **"data.txt":**

Hello to 3v3ry Body. Yours's sincerely. hello I am "Ali". I AM no body.

Total words: 13

Word	Count	Probability
hello	2	2/13
to	1	1/13
body	2	2/13
yours	1	1/13
sincerely	1	1/13
i	2	2/13
am	2	2/13
ali	1	1/13
no	1	1/13

Apply these four filters to the text.

1. Convert every word to lowercase.
2. Ignore the words with numeric character/s.
3. Remove special characters (e.g., ' , " " .) it means remove every character other than English alphabets.
4. Remove stop words (), a separate file of stop words is given "StopWords".

Note: In the above example 4th condition is not applied, but you must use all four conditions.

To keep record of word and its count you are required to use an appropriate data structure that you have previously studied.

Implement a function ***getWordsList*** that receives text file name and returns a list of words after preprocessing the text file. Preprocessing means to filter the data according to the above-mentioned criteria. This function opens the file, reads the data, apply filters on data and store in a list.

Implement a function ***printWord*** which displays the **Word, count** and its **probability** as above table (top 100 words, having higher probabilities).

Work Flow:

1- Read the file, getWordsList(filename)
   Apply 4 conditions
   i- Convert text to lowercase (use text.**lowercase() function**)
   ii- Split the text into words (use text.**split(separater) function**)
   iii- Remove words having numeric character/s
   iv- Apply 3rd filter/condition
   v- Remove stop words (use **set operation**)
2- Get list of words from getWordsList function, store the words and their counts in appropriate data structure (this data structure will store the words from all text files)
3- Now go to step-1 and read the next file and continue until all files are read
4- Call printWords function which displays the **Word, count** and its **probability** as above table

8 Submission Instructions

Always read the submission instructions carefully.

- Rename your Jupyter notebook to your roll number and download the notebook as **.ipynb** extension.
- To download the required file, go to **File->Download .ipynb**
- Only submit the **.ipynb** file. DO NOT **zip** or **rar** your submission file
- Submit this file on Google Classroom under the relevant assignment.
- Late submissions will not be accepted