**National University of Computer and Emerging Sciences**

Logo

Description automatically generated with medium confidence

**Lab Manual 04 CL461-Artificial Intelligence Lab**

| Course Instructor | Dr. Aamir Wali |
| --- | --- |
| Lab Instructor (s) | Hira Ilyas  Saima Akhtar  Usama Khan  Usman Anwer |
| Section | D, B |
| Semester | Spring 2022 |

Department of Computer Science

FAST-NU, Lahore, Pakistan

Table of Contents

[Objectives](#_heading=h.gjdgxs) **3**

[Task Distribution](#_heading=h.30j0zll) **3**

[Python File Handling](#_heading=h.1fob9te) **3**

[Open & Close a file](#_heading=h.3znysh7) 3

[open(filename, mode)](#_heading=h.rkcvwdqvcm7v) 4

[Kinds of modes](#_heading=h.2et92p0) 4

[Working of read() mode](#_heading=h.tyjcwt) 5

[Working of write() mode](#_heading=h.3dy6vkm) 5

[Working of append() mode](#_heading=h.1t3h5sf) 5

[Python Iterators](#_heading=h.4d34og8) **5**

[Building Custom Iterators](#_heading=h.2s8eyo1) 6

[Classes and Objects](#_heading=h.6optqvttyh9y) **7**

[Defining a Class in Python](#_heading=h.ntlhef6njyvu) 7

[Creating Objects](#_heading=h.bto5p46svjm) 8

[Constructors in Python](#_heading=h.30bkx4jpna0t) 8

[Inheritance in Python](#_heading=h.y3ypz4ogwfiq) 9 [9](#_heading=h.etw2351o11mv)

[Python Stacks](#_heading=h.8wkd7ruy1ev4) **9**

[Stacks via Lists](#_heading=h.rzuo6go62loi) 9

[Stacks via collections.deque](#_heading=h.26in1rg) 10

[Stacks via queue.LifoQueue](#_heading=h.lnxbz9) 10

[Python Queues](#_heading=h.35nkun2) **11**

[Exercise](#_heading=h.17dp8vu) **11**

[Compute the probabilities of each word from text file](#_heading=h.3rdcrjn) 11

[Create a class for rectangle shape that calculates its area based upon the length and widt](#_heading=h.7m8ih1e1uqzs)h 13

[Form a queue such that it works in LIFO order](#_heading=h.y2qdfwreripc)  13

# Objectives

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

* Python file handling
* Python iterators
* Classes and objects
* Stacks
* Queues

# Task Distribution

| **Total Time** | **170 Minutes** |
| --- | --- |
| Python File Handling | 20 Minutes |
| Python Iterators | 10 Minutes |
| Classes and Objects | 10 Minutes |
| Stack and Queues | 10 Minutes |
| Exercise | 80 Minutes |

# 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](https://towardsdatascience.com/knowing-these-you-can-cover-99-of-file-operations-in-python-84725d82c2df)

## 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 builtin 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)

## 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)

## 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))

## 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.

## Working of append() mode

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

# 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)

## Building Custom Iterators

| \_\_iter\_\_() | and the | \_\_next\_\_() |
| --- | --- | --- |

Building an iterator from scratch is easy in Python. We just have to implement the methods.

| \_\_iter\_\_() |
| --- |

The method returns the iterator object itself. If required, some initialization can be performed.

| \_\_next\_\_() |
| --- |

The 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.

# 5. Classes and Objects

Python is an object oriented programming language. Almost everything in Python is an object, with its properties and methods. A Class is like an object constructor, or a "blueprint" for creating objects.

## 5.1 Defining a Class in Python

Like function definitions begin with the [def](https://www.programiz.com/python-programming/keyword-list#def) keyword in Python, class definitions begin with a [class](https://www.programiz.com/python-programming/keyword-list#class) keyword.

The first string inside the class is called docstring and has a brief description of the class. Although not mandatory, this is highly recommended.

Here is a simple class definition,

class MyNewClass:

'''This is a docstring. I have created a new class'''

pass

A class creates a new local [namespace](https://www.programiz.com/python-programming/namespace) where all its attributes are defined. Attributes may be data or functions.

As soon as we define a class, a new class object is created with the same name. This class object allows us to access the different attributes as well as to instantiate new objects of that class.

class Person:

"This is a person class"

age = 10

def greet(self):

print('Hello')

# Output: 10

print(Person.age)

# Output: <function Person.greet>

print(Person.greet)

# Output: "This is a person class"

print(Person.\_\_doc\_\_)

## 5.2 Creating Objects

The procedure to create an object is similar to a [function](https://www.programiz.com/python-programming/function) call.

harry = Person()

## 5.3 Constructors in Python

Class functions that begin with double underscore \_\_ are called special functions as they have special meaning.

Of one particular interest is the \_\_init\_\_() function. This special function gets called whenever a new object of that class is instantiated.

This type of function is also called constructors in Object Oriented Programming (OOP). We normally use it to initialize all the variables.

class ComplexNumber:

def \_\_init\_\_(self, r=0, i=0):

self.real = r

self.imag = i

def get\_data(self):

print(f'{self.real}+{self.imag}j')

# Create a new ComplexNumber object

num1 = ComplexNumber(2, 3)

We can even delete the object itself, using the del statement.

del c1

## 

## 5.4 Inheritance in Python

Inheritance is a powerful feature in object oriented programming.

It refers to defining a new [class](https://www.programiz.com/python-programming/class) with little or no modification to an existing class. The new class is called derived (or child) class and the one from which it inherits is called the base (or parent) class.

### Python Inheritance Syntax

class BaseClass:

Body of base class

class DerivedClass(BaseClass):

Body of derived class

# 6 Python Stacks

A stack is a data structure that stores items in a Last-In/First-Out manner. We will look at three different implementations of stacks in Python.

* Using list data structure
* Using collections.deque module
* Using queue.LifoQueue class

## Stacks via Lists

The built-in Python list object can be used as a stack. For stack **.push()** method, we can use **.append()** method of list. **.pop()** method of list can remove elements in LIFO order. Popping an empty list (stack) will raise an **IndexError.** To get the top most item (peek) in the stack, write **list[-1]**. Bigger lists (stacks) often run into speed issues as they continue to grow. **list** may be familiar, but it should be avoided because it can potentially have memory reallocation issues.

myStack = []

myStack.append('a')

myStack.append('b')

myStack.append('c')

myStack # ['a', 'b', 'c']

myStack.pop() # ‘c’

myStack.pop() # ‘b’

myStack.pop() # ‘a’

## Stacks via collections.deque

This method solves the speed problem we face in lists. The **deque** class has been designed as such to provide O(1) time complexity for append and pop operations. The **deque** class is built on top of a doubly linked list structure which provides faster insertion and removal. Popping an empty **deque** gives the same **IndexError**. Read more about it [here](https://docs.python.org/3/library/collections.html#collections.deque). Also, to know more about linked lists in Python, read [this](https://realpython.com/linked-lists-python/).

from collections import deque

myStack = deque()

myStack.append('a')

myStack.append('b')

myStack.append('c')

myStack # deque(['a', 'b', 'c'])

myStack.pop() # ‘c’

myStack.pop() # ‘b’

myStack.pop() # ‘a’

## Stacks via queue.LifoQueue

**LifoQueue** uses **.put()** and **.get()** to add and remove data from the stack. **LifoQueue** is designed to be fully thread-safe. But use it only if you are working with threads. Otherwise, **deque** works well. The **.get()** method by default will wait until an item is available. That means it waits forever if no item is present in the list. Instead, **get\_nowait()** method would immediately raise empty stack error. Read more about it [here](https://docs.python.org/3/library/queue.html).

from queue import LifoQueue

myStack = LifoQueue()

myStack.put('a')

myStack.put('b')

myStack.put('c')

myStack # <queue.LifoQueue object at 0x7f408885e2b0>

myStack.get() # ‘c’

myStack.get() # ‘b’

myStack.get() # ‘a’

# 7 . Python Queues

A queue is FIFO data structure. The insert and delete operations are sometimes called **enqueue** and **dequeue**. We can use list as a queue as well. To follow FIFO, use **pop(0)** to remove the first element of the queue. But as discussed before, lists are slow. They are not ideal from performance perspective.

We can use the **collections.deque** class again to implement Python queues. They work best for non-threaded programs. We can also use **queue.Queue** class. But it works well with synchronized programs.

If you are not looking for parallel processing, **collections.deque** is a good default choice.

from collections import deque

q = deque()

q.append('eat')

q.append('sleep')

q.append('code')

q # deque(['eat', 'sleep', 'code'])

q.popleft() # 'eat'

q.popleft() # 'sleep'

q.popleft() # 'code'

q.popleft() # IndexError: "pop from an empty deque"

# 8 Exercise (50 marks)

## 8.1 compute the probabilities of each word from text file (marks = 30)

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.

| **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 |

Probability can be computed as:

𝑐𝑜𝑢𝑛𝑡(𝑤𝑜𝑟𝑑)

Probability (word) =

𝑇𝑜𝑡𝑎𝑙 𝑊𝑜𝑟𝑑𝑠

Sample **“data.txt”:**

Hello to 3v3ry Body. Yours’s sincerely. hello I am “Ali”. I AM no body.

**Total words: 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 abovementioned 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**)

1. 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)
2. Now go to step-1 and read the next file and continue until all files are read
3. Call printWords function which displays the **Word, count** and its **probability** as above table

## 8.2 Create a class for rectangle shape that calculates its area based upon the length and width (10 marks)

## 8.3 Form a queue such that it works in LIFO order (10 marks)