IAT2 –Solution(Answer Key)

Subject:Python Programming(16MCA21)

1)a)List Creation:
The general form of a list expression is as follows:
[*«expression1»*, *«expression2»*, ... , *«expressionN»*]
>>> whales = [5, 4, 7, 3, 2, 3, 2, 6, 4, 2, 1, 7, 1, 3]
>>> whales
[5, 4, 7, 3, 2, 3, 2, 6, 4, 2, 1, 7, 1, 3]
Slicing and Cloning:
We can produce a new list taking a *slice* of the list:
>>> whales[4:10]
Cloning:

If we want to modify a list and also keep a copy of the original, we need to be able to make a copy of the list itself, not just the reference. This process is sometimes called **cloning**, to avoid the ambiguity of the word "copy."

The easiest way to clone a list is to use the slice operator:

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> print b
[1, 2, 3]
```

Taking any slice of `a` creates a new list. In this case the slice happens to consist of the whole list.

Modifying:
Modifying is based on index of list.
>>>Whales[3]=22
Traversing a list using for in:
b)The general form of a for loop over a list is as follows:
for *«variable»* **in** *«list»*:
«block»
A for loop is executed as follows:
• The loop variable is assigned the first item in the list, and the loop block—the *body* of the for loop—is executed.
• The loop variable is then assigned the second item in the list and the loop body is executed again.
b) for loop:
for *«variable»* **in** *«str»*:
«block»
As with a for loop over a list, the loop variable gets assigned a new value at

the beginning of each iteration. In the case of a loop over a string, the variable is assigned a single character.
For example, we can loop over each character in a string, printing the uppercase letters:

```
>>> country = 'United States of America'
>>> for ch in country:
... if ch.isupper():
... print(ch)
```

while loop:

for loops are useful only if you know how many iterations of the loop you need.In some situations, it is not known in advance how many loop iterations to execute. In a game program, for example, you can't know whether a player is going to want to play again or quit. In these situations, we use a while loop.
The general form of a while loop is as follows:

while *«expression»*:
«block»

The while loop expression is sometimes called the *loop condition*, just like the condition of an if statement. When Python executes a while loop, it evaluates the expression. If that expression evaluates to False, that is the end of the execution of the loop. If the expression evaluates to True, on the other hand,Python executes the loop body once and then goes back to the top of the loopand reevaluates the expression. If it still evaluates to True, the loop body is executed again. This is repeated—expression, body, expression, body—until the expression evaluates to False, at which point Python stops executing the loop.
Here's an example:

```
>>> rabbits = 3
>>> while rabbits > 0:
... print(rabbits)
... rabbits = rabbits - 1
...
```

2)a)Steps in opening a file:

1. Make a directory, perhaps called file_examples.
2. In IDLE, select File→New Window and type (or copy and paste) the following:

```
First line of text
Second line of text
Third line of text
```

3. Save this file in your file_examples directory under the name file_example.txt.
4. In IDLE, select File→New Window and type (or copy and paste) this program:

```
file = open('file_example.txt', 'r')
contents = file.read()
print(contents)
file.close()
```

5. Save this as file_reader.py in your file_examples directory.

b)Reading and Writing:

```
with open('file_example.txt', 'r') as file:
contents = file.read()
print(contents)
```
When called with no arguments, it reads everything from the current file cursor all the way to the end of the file and moves the file cursor to the end of the file. When called with one integer argument, it reads that many characters and moves the file cursor after the characters that were just read
This example reads the contents of a file into a list of strings and then prints that list:
```
with open('file_example.txt', 'r') as example_file:
lines = example_file.readlines()
print(lines)
```
Here is the output:
['First line of text.\n', 'Second line of text.\n', 'Third line of text.\n']
Take a close look at that list; you'll see that each line ends in \n characters. Python does not remove any characters from what is read; it only splits them into separate strings.
The "For Line in File" Technique
Use this technique when you want to do the same thing to every line from the file cursor to the end of a file. On each iteration, the file cursor is moved to the beginning of the next line.
This code opens file planets.txt and prints the length of each line in that file:
```
>>> with open('planets.txt', 'r') as data_file:
... for line in data_file:
... print(len(line))
...
with open('topics.txt', 'w') as output_file:
output_file.write('Computer Science')
```
In addition to writing characters to a file, method write returns the number of characters written.
To create a new file or to replace the contents of an existing file, we use write mode ('w'). If the filename doesn't exist already, then a new file is created; otherwise the file contents are erased and replaced. Once opened for writing, you can use method write to write a string to the file.
Rather than replacing the file contents, we can also add to a file using the append mode ('a'). When we write to a file that is opened in append mode, the data we write is added to the end of the file and the current file contents are not overwritten.
C)#matrix multiplication

```
r = raw_input("Enter the no of rows:")
c = raw_input("Enter the no of columns:")
print 'Enter numbers in array: '
matrix1=[[0 for x in range(int(r))] for y in range(int(c))]
for i in range(int(r)):
for j in range(int(c)):
```

```
n = raw_input("num :")
matrix1[i][j]=int(n)
for i in range(int(r)):
for j in range(int(c)):
print(matrix1[i][j])
r = raw_input("Enter the no of rows:")
c = raw_input("Enter the no of columns:")
print 'Enter numbers in array: '
matrix2=[[0 for x in range(int(r))] for y in range(int(c))]
for i in range(int(r)):
for j in range(int(c)):
n = raw_input("num :")
matrix2[i][j]=int(n)
for i in range(int(r)):
for j in range(int(c)):
print(matrix2[i][j])
# iterate through rows of X
for i in range(int(r)):
# iterate through columns of Y
for j in range(len(matrix1[0])):
# iterate through rows of Y
for k in range(int(c)):
result1[i][j] += matrix1[i][k] * matrix2[k][j]
for r in result1:
print(r)
# iterate through rows of X
for i in range(len(X)):
# iterate through columns of Y
for j in range(len(Y[0])):
# iterate through rows of Y
for k in range(len(Y)):
result[i][j] += X[i][k] * Y[k][j]
3)a)
def addstudent():
    No=input(“enter student no”)
    Name=input(“enter student name”)
    Stu_list.append(no,name)
def deletestudent():

   Stu_list.delete(no,name)
```

```
N=input(“enter the no of students”)
Stu_list=[[0 for x in range(2))] for y in range(int(n))]
For I in range(n):
Stu_list[0][i],stu_list[0][i]=input(“enter the roll no and student”)
Op_dict={1:addstudent,2:deletestudent,3:Display,4:Exit}
ch = 0
while (ch != 5):
print("1. Student insert")
print("2. Student delete")
print("3. Display")
print("4. Exit")
Selection = int(input("enter your option: "))
if (ch >= 1) and (ch <4):
op_dict[Selection]()
```

b)

Storing Data Using Sets

A *set* is an unordered collection of distinct items. *Unordered* means that items aren't stored in any particular order. Something is either in the set or it's not, but there's no notion of it being the first, second, or last item. *Distinct* means that any item appears in a set at most once; in other words, there are no duplicates.

Python has a type called `set` that allows us to store mutable collections of unordered, distinct items. (Remember that a *mutable* object means one that you can modify.) Here we create a set containing these vowels:

```
>>> vowels = {'a', 'e', 'i', 'o', 'u'}
>>> vowels
{'a', 'u', 'o', 'i', 'e'}
```

```
>>> ten = set(range(10))
>>> lows = {0, 1, 2, 3, 4}
>>> odds = {1, 3, 5, 7, 9}
>>> lows.add(9)
>>> lows
{0, 1, 2, 3, 4, 9}
>>> lows.difference(odds)
{0, 2, 4}
>>> lows.intersection(odds)
{1, 3, 9}
>>> lows.issubset(ten)
True
>>> lows.issuperset(odds)
False
>>> lows.remove(0)
>>> lows
{1, 2, 3, 4, 9}
>>> lows.symmetric_difference(odds)
{2, 4, 5, 7}
>>> lows.union(odds)
{1, 2, 3, 4, 5, 7, 9}
```

4a)
i)>>>list(range(int))
List(Range(int,int))
List(Range(int,int,int))

ii)List=[10,20,30,40]
For item in list :
  Item+=20
Print list
b)Tuples:

Python also has an immutable sequence type called a *tuple*. Tuples are written using parentheses instead of brackets; like strings and lists, they can be subscripted, sliced, and looped over:

```
>>> bases = ('A', 'C', 'G', 'T')
>>> for base in bases:
... print(base)
...
A
C
G
T
```

Assigning to Multiple Variables Using Tuples
You can assign to multiple variables at the same time:

```
>>> (x, y) = (10, 20)
>>> x
10
>>> y
20
```

c)output:Infinte loop printing odd numbers

5a)
Dictionary:

The right approach is to use another data structure called a *dictionary*. Also known as a *map*, a dictionary is an unordered mutable collection of key/value pairs. In plain English, Python's dictionaries are like dictionaries that map words to definitions. They associate a key (like a word) with a value (such as a definition). The keys form a set. Any particular key can appear once at most in a dictionary; and like the elements in sets, keys must be immutable (though the values associated with them don't have to be).

Dictionaries are created by putting key/value pairs inside braces (each key is followed by a colon and then by its value):

Method	Description
D.clear()	Removes all key/value pairs from dictionary D.
D.get(k)	Returns the value associated with key k, or None if the key isn't present. (Usually you'll want to use D[k] instead.)
D.get(k, v)	Returns the value associated with key k, or a default value v if the key isn't present.
D.keys()	Returns dictionary D's keys as a set-like object—entries are guaranteed to be unique.
D.items()	Returns dictionary D's (key, value) pairs as set-like objects.
D.pop(k)	Removes key k from dictionary D and returns the value that was associated with k—if k isn't in D, an error is raised.
D.pop(k, v)	Removes key k from dictionary D and returns the value that was associated with k; if k isn't in D , returns v.
D.setdefault(k)	Returns the value associated with key k in D.
D.setdefault(k, v)	Returns the value associated with key k in D; if k isn't a key in D, adds the key k with the value v to D and returns v.

D.values()	Returns dictionary D's values as a list-like object—entries may or may not be unique.
D.update(other)	Updates dictionary D with the contents of dictionary other; for each key in other, if it is also a key in D, replaces that key in D's value with the value from other; for each key in other, if that key isn't in D, adds that key/value pair to D.

b)i) list(range(33,50))
ii) >>>list(range(10,1,-1))

c)#print matrix

```
r = raw_input("Enter the no of rows:")
c = raw_input("Enter the no of columns:")
print 'Enter numbers in array: '
matrix1=[[0 for x in range(int(r))] for y in range(int(c))]
for i in range(int(r)):
for j in range(int(c)):
n = raw_input("num :")
matrix1[i][j]=int(n)
```

6a)Fibonacci using dictionary:

A good solution is to keep track of values that have already been computed by storing them in a dictionary. A previously computed value that is stored for later use is called a **hint**. Here is an implementation of `fibonacci` using hints:

```
previous = {0:1, 1:1}

def fibonacci(n):
  if previous.has_key(n):
    return previous[n]
  else:
    newValue = fibonacci(n-1) + fibonacci(n-2)
    previous[n] = newValue
    return newValue
```

The dictionary named `previous` keeps track of the Fibonacci numbers we already know. We start with only two pairs: 0 maps to 1; and 1 maps to 1.

b)

```
def intersect(a, b):
```

```
    """ return the intersection of two lists """
    return list(set(a) & set(b))
a = [0,1,2,0,1,2,3,4,5,6,7,8,9]
    b = [5,6,7,8,9,10,11,12,13,14]


    print intersect(a, b)
```

7a)

The following function copies a file, reading and writing up to fifty characters at a time. The first argument is the name of the original file; the second is the name of the new file:

```
def copyFile(oldFile, newFile):
  f1 = open(oldFile, "r")
  f2 = open(newFile, "w")
  while True:
    text = f1.read(50)
    if text == "":
      break
    f2.write(text)
  f1.close()
  f2.close()
  return
```

b)

```
def linear_search(lst, value):
    """ (list, object) -> int

    Return the index of the first occurrence of value in lst, or return
    -1 if value is not in lst.

    >>> linear_search([2, 5, 1, -3], 5)
    1
    >>> linear_search([2, 4, 2], 2)
    0


  >>> linear_search([2, 5, 1, -3], 4)
  -1
  >>> linear_search([], 5)
  -1
  """
```

```
def mystery_function(values):
    result = []
    for sublist in values:
        result.append([sublist[0]])
        for i in sublist[1:]:
            result[-1].insert(0, i)

    return result
```

Docstring for the above function:

“”” (list)-> (list)

8a) def openread():
global fo
print("File is opening in Read mode : ")
fo = open("temp.txt", "r+")
str = fo.read(10);
print "Read String is : ", str
with open("temp.txt", "r") as fin:
print fin.read()
def openwrite():
global fo
print("File is opening in Write mode : ")
fo = open("temp.txt", "w+")
fo.write("nex text is written in temp file")
fo.write("and another line")
def getposition():
global fo
print("Gettting Current File pointer position : ")
position = fo.tell();
print "Current file position : ", position
def setatbegin():
global fo
print("Reposition the pointer at the beginning of the File : ")
position = fo.seek(0, 0);
def errhandler ():
print("Your input has not been recognised")

MenuSelect = {
1: openread,
2: openwrite,
3: getposition,

4: setatbegin
}

Selection = 0
while (Selection != 5):
print("1. Open in Read Mode")
print("2. Open in Write Mode")
print("3. Current Position")
print("4. Set at the Beginning")
print("5. Quit")
Selection = int(input("Select a Menu option: "))
if (Selection >= 1) and (Selection < 5):
MenuSelect[Selection]()
b)

Tuples as return values

Functions can return tuples as return values. For example, we could write a function that swaps two parameters:

```
def swap(x, y):
  return y, x
```

Then we can assign the return value to a tuple with two variables:

```
a, b = swap(a, b)
```

In this case, there is no great advantage in making **swap** a function. In fact, there is a danger in trying to encapsulate **swap**, which is the following tempting mistake:

of the whole list.

```
# two dictionaries
# blow this up to 1000 to see the difference
some_dict = { 'zope':'zzz', 'python':'rocks' }
another_dict = { 'python':'rocks', 'perl':'$' }

# bad way
# two lots of "in"
intersect = []
for item in some_dict.keys():
  if item in another_dict.keys():
    intersect.append(item)

print "Intersects:", intersect

# good way
# use simple lookup with has_keys()
```

```
intersect = []
for item in some_dict.keys():
  if another_dict.has_key(item):
    intersect.append(item)

print "Intersects:", intersect
```