Practice make your coding skill perfect. So, whenever you are given an exercise, please try and practice it.

Remember that computer coding skills need a systematic approach. Therefore, every week's learning is built on the current week and previous week learning.

Lab 9- Exception Handling

1. Exceptions and Exception Classes (Types)

a. What is an exception?
   Give an example to illustrate your answer.

b. In the Python interactive shell, enter the following and observe what the interpreter displays.

```
>>> 1 / 0
>>> int("abc")
>>> 1 + "abc"

>>> from math import log
>>> log(0)

>>> # run this with the file is missing >>>
infile = open("missing.txt")
```

c. Is **ZeroDivionError** a subclass of **ValueError**?
   How can you tell (without looking up some documentation)?

2. Potentially Raised Exceptions

Consider each of the programs below. Can the program raise an exception? If yes, how? What are the types of exceptions that may be raised?

a.
```
n = 10
print(n)
```

b.
```
name = input("Enter your name: ") 2 print(name)
```

c.
```
n = int(input("Enter an integer: ")) 2 print(n)
```

d.
```
n = int(input("Enter an integer: ")) 2 print(n, 100/n)
```

e.
```
infile = open("data.txt")
contents = infile.read() 3
print(contents)
```

f.
```
infile = open("data.txt")
line = infile.readline() 3
print(int(line))
```

g.
```
infile = open("data.txt")
line = infile.readline()
tokens = line.split() 4
numbers = [] 5 for token
in tokens:
numbers.append(int(token))
average = sum(numbers) / len(numbers)
print(average)
```

3. Handling Exceptions

Modify the programs in the previous question to capture and handle exceptions that may arise. Arrange the except blocks to capture each type of exception in a separate except block.

4. Multiple except Blocks

Consider this program:

```
try:
        infile = open("data.txt")
        n = int(infile.readline())
        print("inverse of ", n, "is ", 1/n)
        infile.close()
except FileNotFoundError:
        print("Error: data.txt does not exists!")
except ValueError:
      print("Error: data.txt has incorrect format!")
      print("Bye")
```

a. Under what conditions will block **except FileNotFoundError** handle the raised exception?

   What lines of the program, not counting line containing **try** and **except**, will be executed?

b. Repeat for block **except ValueError**

c. Is there a situation in which the program will crash? In this case, what lines of the program will be executed?

5. The Catch-All except Block

Consider this program:

```
try:
        infile = open("data.txt")
        n = int(infile.readline())
        print("inverse of ", n, "is ", 1/n)
```

```
            infile.close()
except FileNotFoundError:
        print("Error: data.txt does not exists!")
except ValueError:
            print("Error: data.txt has incorrect format!")
            infile.close()
except:
            print("Some error has occurred!") 13 print("Bye")
```

a. What is the difference between this program and the program in the previous question?

b. For the case in which the last except block handles the raised exception, what lines of the program, not counting line containing **try** and **except**, will be executed?

6. Exception Objects

What is the difference between the program below and the program in the previous question?

```
try:
        infile = open("data.txt")
        n = int(infile.readline())
        print("inverse of ", n, "is ", 1/n) 5        infile.close()
except FileNotFoundError:
    print("Error: data.txt does not exists!")
except ValueError:
    print("Error: data.txt has incorrect format!") 10
    infile.close()
except as err:
            print("Error type:", type(err))
            print("Error message:", err)
            print("Bye")
```

7. The finally Block

Test the following code to understand benefit of using finally block in the program below.

```
try:
        infile = open("data.txt")
        n = int(infile.readline())
        print("inverse of ", n, "is ", 1/n)
        ifile.close()
```

```
except Exception as err:
    print("Error type:", type(err)) 8 print("Error
    message:", err)
finally:
try:
    infile.close()
    except:
    pass
```

8. The Rule of 72

The Rule of 72 is a simple way to estimate how long an investment will take to double given a fixed annual rate of interest. By dividing 72 by the annual rate of return (as value 5 for 5%, for example), we obtain a rough estimate of how many years it will take for the initial investment to double itself.

The program below gets the annual interest rate from the user. It then calculates and displays the estimate of number of years it takes to double the initial investment amount.

```
# Get annual interest rate
rate = float(input("Enter annual interest rate (5 for 5%):"))

# Calculate and display number of years it takes to double
# the investment account
years = 72 / rate
print("at the rate of" , rate, "%, it takes", years,
        "to double the initial investment")
```

a. Run the program with the following inputs:
   - 10
   - 0
   - ten

b. Under what conditions does the program have a normal run?

c. What are the exceptions that can occur and disrupt the normal run of the program?
   Demonstrate your answers with suitably chosen sample runs.

d. Modify the *given program* to have one except block that can capture and handle any exception that may arise and display a plain error message
   "Some error has been made. Please try again."

e. Modify the *given program* to have one except block that can capture and handle any exception that may arise and display an error message that differentiate between different types of exceptions."

f. Modify the *given program* so that it has different exception blocks to

capture and handle each type of the exception individually.

Exception Handling and Functional Calls
(To supplement the lecture notes)

– Sequences of Calls

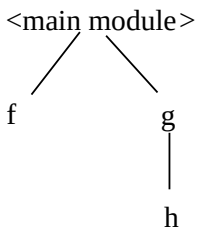
So far in this lab we have considered only programs that do not have any user defined functions (i.e. functions that we ourselves define). Of such a program, we say that it has a *flat structure*.

Exception handling mechanism can be extended to programs that has user defined functions. Of such a program, we say that it has a *tree structure*.

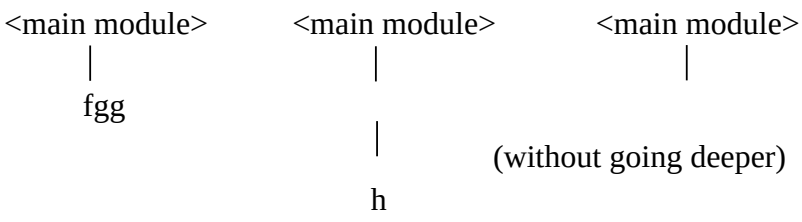
Consider for example a program shown below in skeleton form.

```
def f(): ...
def g():
... h() ...
def h():
    ...
#
main
... f()
... g()
...
```

The static structure of the program can be represented by a tree below, where Main Module denotes the program itself:



The program can give rise to several sequence of calls, as shown in diagram below:



– Exception handling for call sequences

The exception handling mechanism, when applied to sequence of calls, can be summarized as follows:

- An exception that is not handled at one call level is passed upward to the next level above it.
- An exception that is not handled at the main module level will cause the program to crash.

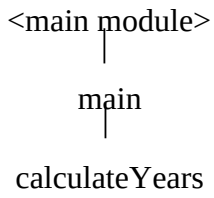
– How to Read a Traceback – An Example

Consider program

```
def calculateYears(rate):
    return 72 / rate

def main():
    rate = float(input("Enter interest rate (e.g. 5 for 5%): "))
years = calculateYears(rate) print("It will take", years, "years")
main()
```

The program has the structure:



Here is the result of a sample run:

```
1 Enter interest rate (e.g. 5 for 5%): 0 2
Traceback (most recent call last):
3     File "C:\ test \rule72.py", line 9, in <module>
4     main()
5     File "C:\ test \rule72.py", line 6, in main
6     years = calculateYears(rate)
7     File "C:\test\rule72.py", line 2, in calculateYears
8     return 72 / rate
9     ZeroDivisionError: float division by zero
```

The traceback (lines 3 - 8) of the sequence of calls shows that three levels are involved. They are extracted and presented in the table below, together with the comments on how to read them.

Lab09 Iterator and generator

1- List comprehension: Write a program using List comprehension for the following tasks

```
squares_l = []
for x in range(10):
c=0
c=x**2
    squares_l.append(c)
print (squares_l)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
Multiply every element of a list by three and assign it to a new
 list.
List_1 = [3,4,5,6,8]
```

```
Print the first letter of each word
listWords = ["this","is","a","CSE5APG","Sem 2","Python", "Subjec
t"]
```

```
Convert list letter into lower case
l1=["C", "S", "E", "F", "A", "P", "G"]

Convert list letter into upper case letters
l2=["c", "s", "e", "f", "a", "p", "g"]
```

```
Check if the element is even, then raise that element to the pow
er of 2 and add into a new list
l=[1,2,3,4,5,6,7,8,9,10]
```

```
Convert kilometre to feet: 3280.8399 * kilometre
kilometre=[36.2, 37.4, 40.2, 55.7, 60.8]
```

```
Convert kilometre to feet if: 10 > kilometre < 40. feet 3280.839
9 * kilometre
kilometre= [10.2, 12.4, 14.2, 30.7, 33.6, 60.3, 41.2, 50]
```

```
Convert Celsius values into Fahrenheit and vice versa:
Fahrenheit= (1.8 + 32) * Celsius
Celsius= (Fahrenheit - 32) / 1.8

Celsius = [40.2, 33.4, 36.6, 38.4]
```

```
Cross product of two sets:
colours = [ "red", "green", "yellow", "blue"]
things = [ "house", "car", "tree"]
Output:
[('red', 'house'), ('red', 'car'), ('red', 'tree'), ('green', 'h
ouse'), ('green', 'car'), ('green', 'tree'), ('yellow', 'house')
, ('yellow', 'car'), ('yellow', 'tree'), ('blue', 'house'), ('bl
ue', 'car'), ('blue', 'tree')]
```

```
Use Nested Comprehensions to generate the following dictionary:

'Mleb':[1, 1, 1, 1, 1, 1, 1],
'Syd':[1, 1, 1, 1, 1, 1, 1],
'BNE':[1, 1, 1, 1, 1, 1, 1],
'Adal':[1, 1, 1, 1, 1, 1, 1],
'Perth':[1, 1, 1, 1, 1, 1, 1],
'Canb':[1, 1, 1, 1, 1, 1, 1]
```

2- Iterator

```
# Run this code and inspect the output
c=[1, 2, 3,4,5,6]
print (c)
l = iter(c)
print(l)
print (next(l))
print (next(l))
print (next(l))
print (next(l))
```

```
# Run this code and inspect the output
class CountDownN(object):
def __init__(self, start):
     self.counter = start + 5
def next(self):
     self.counter= self.counter-1
     if self.counter <= 0:
              raise StopIteration
return self.counter
def __iter__(self):
return self
x= CountDownN(10)
for i in x:
print(x)
```

3- Generator

```
# Run this code and inspect the output
def count_do(value):
for i in range(value, 0, -1):
yield i
c = count_do(6)
print(next(c))
print(next(c))
print(next(c))
print(next(c))
print(next(c))
print(next(c))
```

```
# Run this code and inspect the output
exp_gen = (x for x in range(6, 0, -1))
for i in exp_gen:
print(i)
```

```
# Run this code and inspect the output
def my_generator():
print("Inside my generator")
yield 'x'
yield 'y'
yield 'z'
for ch in my_generator():
print(ch)
```

```
# Run this code and inspect the output
s=sum([x*x for x in range(1,8)])
print (s)
g=sum(x*x for x in range(1,8))
print (g)
print (next(g))
print (next(g))
print (next(g))
print (next(g))
print (next(g))
```

```
# Run this code and inspect the output

def fib(value):
# Initialise the two
a, b = 1, 1
# One by one yield next Fibonacci Number
while a < value:
yield a
a, b = b, a + b
# Create a generator object
z = fib(6)
# Iterating over the generator object using next
print(next(z))
print(next(z))
print(next(z))
print(next(z))
print(next(z))
# Iterating over the generator object using for in loop.
print("\nfor-loop")
for i in fib(6):
print(i)
```

- Write a generator function that returns the cube of every other number in a given range starting from 1.
- Write a generator expression that returns the cube of every other number in a given range starting from 1.
- Write a generator function that converts lowercase to uppercase of a given string.
- Write a generator function that generates the first 10 perfect square numbers starting from 1