MCT-243 : COMPUTER PROGRAMMING-II
using Python 3.9

Prepared By:
Mr. Muhammad Ahsan Naeem

YouTube Playlist
https://www.youtube.com/playlist?list=PLWF9TXck7O_wMDB-VriREZ6EvwkWLNB7q

Lab 26: EXCEPTION HANDLING

Exception handling is a method to handle the exceptions (errors) of our code. We categorize errors in three types:

i. Syntax Error
ii. Logical Error
iii. Run-Time Error

When there is a syntax error in the program, it does not enter into the run mode. The interpreter will indicate the line number with syntax error on the output. While in case of Logical and Run-Time error, the program enters into run mode and will generate the error when that specific statement with logical or run-time error will be executed. The interpreter will execute Traceback function and will also indicate the name of the error and a little description. Syntax Errors are called as **Error** and the Logical/Run-Time errors are called as **Exception**. However, both the terms, Error and Exception are usually used interchangeably.

Exceptions must be handled carefully and intelligentially. We should consider the situation that could generate the exception and should bypass that as illustrated in following example:

Example:

Take two numbers from user and display the ratio.

The above simple task can be done as:

```
a,b=eval(input("Enter two numbers:"))
print(f"Ratio of two numbers is: {a/b}")
```

We know the above code will generate error when `b=0`. To avoid the error, we can use the following code:

```
a,b=eval(input("Enter two numbers:"))
if(b!=0):
    print(f"Ratio of two numbers is: {a/b}")
else:
    print("Second number cannot be zero")
```

The above approach of handling the possible error is known as **Look Before You Leap (LBYL)**. We are applying the checks before executing a statement that might cause an exception.

Second approach to handle the exceptions is known as **it's Easy to Ask Forgiveness than Permission (EAFTP)**.

In this approach we try to execute one statement, and if that executes successfully, we are good but if there is any exception then instead of generating that exception, we **catch** that. This is done with **try/except** blocks. In `try` block/clause we write statement(s) (usually just one or two that we think can generate an exception), and then is an `except` block/clause. If the statement(s) in `try` clause are executed successfully the interpreter will not execute the `except` clause. And if there is some exception while executing the `try` clause, the interpreter will not generate the error and instead it will leave the `try` clause and will enter into the `except` clause. We call this as **Catching the Exception**. The error inside the `try` clause can be any type of error.

The above simple example of the ratio of the two numbers, instead of division by zero, there can be some other possibility of the exception e.g., if instead of a number, user enter some letter and that will cause the exception for the `eval` function. So, here is the code with **try/except** approach that will handle both division by zero and non-numeric input:

```
try:
    a,b=eval(input("Enter two numbers:"))
    print(f"Ratio of two numbers is: {a/b}")
except:
    print("Ener numbers only and second number should be non-zero")
print("Thanks")
```

Three sample output of the above code is given as:

```
Enter two numbers:5,A
Ener numbers only and second number should be non-zero
Thanks
```

```
Enter two numbers:6,0
Ener numbers only and second number should be non-zero
Thanks
```

```
Enter two numbers:6,10
Ratio of two numbers is: 0.6
Thanks
```

A good thing about `try/except` is that we can target different errors by the error names. Every error has its associated exception name. For example, if we attempt to divide a number by 0, the exception generated has name `ZeroDivisionError`. And if some non-numeric value is entered in `eval()` function, the exception generated has name `NameError`. See the following code where the errors are handled by their names:

```
try:
    a,b=eval(input("Enter two numbers:"))
    print(f"Ratio of two numbers is: {a/b}")
except ZeroDivisionError:
    print("Second number cannot be zero!")
except NameError:
    print("Enter numeric values only!")
except:
    print("Something bad happened!")
    print("Report to the team.")
```

The two exception blocks are handling the errors by the names. If there is some error other than those two possibilities, the last anonymous except block is targeting that. The order of the exceptions is important.

In case of some exception inside the **`try`** clause, the exception is scanned in order against all except clauses. So, if the unnamed except clause is the first one that can catch any exception, then there is no use of the specific exceptions after that.

There is also the possibility to access the exception message (the one that is displayed by the Traceback function when exception is generated) by using the **`as`** keyword as shown here:

```
a,b=5,0
try:
    print(f'Ratio of two numbers is: {a/b}')
except ZeroDivisionError as e:
    print("Second number cannot be zero!")
    print(e)
print('Thanks')
```

With **`b`** as **`0`**, this will be the output:

```
Second number cannot be zero!
division by zero
Thanks
```

Try/Except (EAFP) vs. if condition (LBYL):

Firstly, the **`try/except`** approach is not the replacement of the **`if`** statement. Only when an **`if`** statement is used to handle some error, then we can have a **`try/except`** approach there. Otherwise, if you are using the **`if`** statement to decide a number **`n`** is even or odd like **`if(n%2==0)`** then of course it cannot be replaced with **`try/except`**.

Between the two LBYL and EAFP approaches, EAFP is better. Sometime we call that it is Pythonic approach and the LYBL is Non-Pythonic approach. See this example with LBYL approach where we have a dictionary and before printing the information inside the dictionary we are making sure that the key actually exists inside it, otherwise there will be **`KeyError`** exception.

```
a={
    'Name':'Ali Ahmad',
    'Reg':'MCT-UET-01',
    'Courses':['CP','LA','VCA']
    }
if('Name' in a and 'Reg' in a and 'Courses' in a):
    print(f'Hello {a["Name"]}-
({a["Reg"]}). Your courses are {a["Courses"]}.')
else:
    print('Some info is missing!')
```

Now see the EAFP approach here:

```
a={
    'Name':'Ali Ahmad',
    'Reg':'MCT-UET-01',
    'Courses':['CP','LA','VCA']
    }
try:
    print(f'Hello {a["Name"]}-
({a["Reg"]}). Your courses are {a["Courses"]}.')
except KeyError as e:
    print(f'Key {e} is missing!')
```

Observe the two approaches above and you can easily see that the EAFP is better than LBYL approach as:

- If there are 20 keys inside the dictionary, in case of LBYL, you will have to add all twenty checks inside the `if` statement condition. On the other hand, no change is required in EAFP approach.
- If all keys used inside the `print` statement are available inside the dictionary, in case of LBYL even then the dictionary is checked against each of the key. While in case of EAFP, simply the `print` statement is executed without any check tested on the dictionary. Hence EAFP is faster as compared to LBYL.
- In case of LBYL, someone reading your code will have to understand the condition carefully to know what is being tried to handle. On the other hand, in case of EAFP, it is very clear which exception you are trying to handle.

Else and Finally with Try/Except:

Although in many cases we use only the `try` and `except` clauses but there can be `else` and `finally` clauses with these as well which is explained here with an example.

If we want to read a text file named as `'test.txt'`, there is a chance that the file doesn't exists. We can handle this using try/except as:

```
try:
    f=open('test.txt')
    print(f.read())
    f.close()
except FileNotFoundError:
    print('No such file exists')
```

If the file exists, it will be opened, read and closed. And if it doesn't exist, it will print the message written at the end. In above example, we are considering the possibility that the file may not exist. The exception that can occur will be only because of line `f=open('test.txt')` and there are only two lines which are executed if file exists; which are:

```
print(f.read())
f.close()
```

In many cases there can be a lengthy code but the exception we want to handle is because of one or two lines only. For these cases it is good practice to keep, within the **try** block, only the lines that may cause error. The other code that is required to be executed when there is no error should be handled in **else** clause. The code in the **else** clause is executed when there is no exception in the **try** clause and if there is some exception there, **else** clause will not be executed. The above code with this approach is shown here:

```
try:
    f=open('test.txt')
except FileNotFoundError:
    print('No such file exists')
else:
    print(f.read())
    f.close()
```

Now, there might be error for some problem other than **FileNotFoundError** and we can handle that in general exception and print some message or even the error message as it is without causing the program to stop as shown here:

```
try:
    f=Open('test.txt')
except FileNotFoundError:
    print('No such file exists')
except Exception as e:
    print(e)
else:
    print(f.read())
    f.close()
```

Note that it is written **Open** instead of **open** in **try** block and it will get caught in second except block with the exception message printed. The program will continue after that (the code after **else** block) instead of getting terminated.

Why is it better to use else clause?

The purpose of **try/except** is to handle the exceptions and we want to be specific which error we are trying to handle. So, if we have a lengthy code in the **try** clause, it is hard to understand which line can cause the problem. So, a short **try** clause with just a couple of statements that might cause some exception, and the **except** clauses immediately after that clearly indicate what and how we want to handle an exception. And if everything is fine, then we have the **else** clause for further logic.

Now let's see the use of **finally** clause.

After **try**, **except** and the **else** clause, we can have the **finally** clause. The **else** clause is executed when there is no exception occurred in the **try** clause. But the **finally** clause will executed in any case; whether there occurred some exception in the **try** block or not. See this code:

```
try:
    f=open('test.txt')
```

```
except FileNotFoundError:
    print('No such file exists')
except Exception as e:
    print(e)
else:
    print(f.read())
finally:
    if 'f' in locals():
        f.close()
```

The file is being closed in the **finally** block because that is something we want to do in any case. It might be observed that if we do not have the **finally** clause and we just write the code of the **finally** clause at the end, that will get executed in any case.

```
try:
    f=open('test.txt')
except FileNotFoundError:
    print('No such file exists')
except Exception as e:
    print(e)
else:
    print(f.read())

if 'f' in locals():
    f.close()
```

There is one important point to understand in above code:

> Maybe we can have the file close statement inside the **else** clause since **else** will get executed only when there is no exception in the **try** clause, meaning the file opened successfully.

It looks justified but imagine there is some exception inside the **else** clause. Meaning that there was no exception inside the **try** clause and hence the file is opened, and the **else** clause is being executed and some exception occurs inside the **else** clause before the file close statement. In such case, the interpreter will generate the exception since there is no mechanism to catch the exception inside the **else** clause and the file opened in the **try** clause will not get closed. That is generally not a good thing since it is really important to close a file or database to release the system resources.

Now consider the case where we have the **finally** clause with file close statement. The **finally** clause gets executed whether there was exception or not in the **try** clause. Not just that, even if the exception occurs inside the **else** clause and there is no mechanism to catch that, and the exception will occur definitely, and even in such case interpreter will first execute the **finally** clause and then will generate the exception.

So, we put the important statement like closing a file or database inside the **`finally`** clause to make sure it gets executed in any worst case scenario.

Another scenario where the **`finally`** clause can make a difference is the use of **`try/except`** inside a user-defined function. See one example code without the **`else`** clause but having the **`finally`** clause:

```
def test():
    try:
        # Some statement that can generate exception
    except:
        return None
    finally:
        # Some statements
```

Suppose some exception occurs in the **`try`** clause of the above user-defined function. We know, the interpreter will execute the **`except`** clause and over there is a **`return`** statement. We know whenever **`return`** statement is execute by a function, the value is returned to the point where the function was invoked. But in above case where we have the **`finally`** clause and the **`return`** statement is to be executed, the interpreter will first execute the **`finally`** clause and then will execute the **`return`** statement. Hence the **`finally`** clause get executed no matter what happens.

Tasks:

[1] Suppose there is a Dictionary with 8 key-value pairs as shown here:

```
d={
    'element1':5,
    'element2':2,
    'element3':0,
    'element4':'Computer',
    'element5':3.2,
    'element6':False,
    'element7':0,
    'element8':9
}
```

Suppose that we want to print the reciprocal of each value inside the dictionary. So this is what we can do:

```
for k,v in d.items():
    print(f'Reciprocal of {v} is {1/v}')
```

It is going to generate errors since we can divide 0 or string or Boolean data types. Write the code in a way that if reciprocal is possible, it will print that as:

```
Reciprocal of 5 is 0.2.
```

And if reciprocal is not possible, it will print that it is not possible along with the description of the exception. For example, in case of 0, this should be displayed:

```
Reciprocal of 0 is not possible. Exception : (division by zero)
```

[2] A file read operation is applied using the try/except/else/finally clause:

```
try:
    f=open('sample.txt','r')
except FileNotFoundError:
    print('File does not exist!')
else:
    print(f.readlines())
finally:
    if 'f' in locals():
        f.close()
```

This works fine but suppose in the **`else`** clause, instead of reading the file, there is a statement to write content as:

```
f.write('hello')
```

This is going to generate error since file is opened in the read mode. We can open the file in read+write mode, but for some reasons we want the file to be opened in read mode always and writing should not be allowed. Mange the code with nested **`try`** clauses such that if write operation is attempted, an appropriate message is printed instead of an exception.

Raising an Exception:

We can also raise/generate exceptions at certain condition. Why would we raise exceptions? Mostly such situations arises when we will be writing classes which are meant to be used by other programmers. But generally, it can be any simple situation e.g., if we want user to enter a number between 1 to 10 only and want to terminate the code with error if it is not between 1 to 10, we can do it like:

```
x=eval(input("Enter a number (1 to 10): "))
if (x<1 or x>10):
    raise Exception

#If correct number is entered
print(f"You entered: {x}.")
```

We can also print customized message with the exception as well as shown here:

```
x=eval(input("Enter a number (1 to 10): "))
if (x<1 or x>10):
    raise Exception(f"1-10 allowed. You entered {x}.")

#If correct number is entered
print(f"You entered: {x}.")
```

In above examples, we are generating a new exception by our own. The other exceptions that we see have names also. If we want to raise some specific exception, we can do that by specifying that exception name. For example, in above case if numbers 1-10 are the keys of some dictionary within our program and we want to generate **`KeyError`** when other than 1-10 is entered (**`KeyError`** is a specific error generated when we try to access an unavailable key from a dictionary), we can do it as:

```
x=eval(input("Enter a number (1 to 10): "))
if (x<1 or x>10):
    raise KeyError

#If correct number is entered
print(f"You entered: {x}.")
```

We can also specify some customized message with the error as:

```
x=eval(input("Enter a number (1 to 10): "))
if (x<1 or x>10):
    raise KeyError(f'1-10 allowed. You entered {x}.')

#If correct number is entered
print(f"You entered: {x}.")
```