EXCEPTION HANDLING AND FILE IO

EXCEPTION
HANDLING

When programming, errors happen. It's just a normal experience. Maybe the user entered bad input. Maybe a network resource was unavailable. Maybe the program ran out of memory. Or the programmer may have even made a mistake!

Programme's solution to errors are exceptions. You might have seen one before.

EXCEPTIONS **VERSUS** SYNTAX ERRORS

Syntax errors occur when the parser detects an incorrect statement. Observe the following example:

```
>>> print( o / o ))
SyntaxError: invalid syntax
```

The arrow indicates where the parser ran into the syntax error. In this example, there was one bracket too many.

EXCEPTIONS **VERSUS** SYNTAX ERRORS

```
>>> print( 0 / 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

RAISING an EXCEPTION

We can use ***raise*** to throw an exception if a condition occurs. The statement can be complemented with a custom exception.

If you want to throw an error when a certain condition occurs using raise, you could go about it like this:

```
x = 10

if x > 5:

    raise Exception('x should not exceed 5. The value of x was: {}'.format(x))
```

The program comes to a halt and displays the exception to screen, with clues about what went wrong.

THE AssertionError EXCEPTION

Instead of waiting for a program to crash midway, you can also start by making an assertion in Python. We assert that a certain condition is met. If this condition turns out to be True, then that is excellent! The program can continue. If the condition turns out to be False, you can have the program throw an AssertionError exception.

Have a look at the following example, where it is asserted that the code will be executed on a Linux system:

```
import sys

assert ('linux' in sys.platform), "This code runs on Linux only."
```

Exception Handling

The try statement works as follows.

- First, the *try clause* (the statement(s) between the try and except keywords) is executed.
- If no exception occurs, the *except clause* is skipped and execution of the try statement is finished.
- If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the except keyword, the except clause is executed, and then execution continues after the try statement.

Exception Handling

- If an exception occurs which does not match the exception named in the except clause, it is an *unhandled exception* and execution stops with a message as shown above.

A try statement may have more than one except clause, to specify handlers for different exceptions. At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding try clause.

EXAMPLE

```
Def method():
    try:
            a = 2/0
    except ZeroDivisionError as e:
            print(e)
```

THE else CLAUSE

In Python, using the else statement, you can instruct a program to execute a certain block of code only in the absence of exceptions.

```
def method():
try:
        a = 2/0
except ZeroDivisionError as e:
        print(e)
else:
        print('Exception did not occur')
```

CLEANING Up AFTER USING finally

Imagine that you always had to implement some sort of action to clean up after executing your code. Python enables you to do so using the finally clause.

```
def method():
try:
        a = 2/0
except ZeroDivisionError as e:
        print(e)
else:
        print('Exception did not occur')
finaly:
        print('Clean up the code')
```

POINTS to NOTE

- ***raise*** allows you to throw an exception at any time.
- ***assert*** enables you to verify if a certain condition is met and throw an exception if it isn't.
- In the ***try*** clause, all statements are executed until an exception is encountered.
- ***except*** is used to catch and handle the exception(s) that are encountered in the try clause.
- ***else*** lets you code sections that should run only when no exceptions are encountered in the try clause.
- ***finally*** enables you to execute sections of code that should always run, with or without any previously encountered exceptions.

PYTHON - FileIO

OVERVIEW

Python's built-in functions ***input()*** and ***print()*** perform read/write operations with standard IO streams. The *input()* function reads text into memory variables from keyboard and the print() function send data to display device.

Instead of standard output device, if data is saved in persistent computer files then it can be used subsequently. ***File is a named location in computer's non-volatile storage device such as hard disk used to permanently store data***.

Since, random access memory (RAM) is volatile which loses its data when computer is turned off, we use files for future use of the data.

Python's built-in function open() returns file object mapped to a file on the permanent storage like disk.

open() FUNCTION

File object is returned by open() function which needs name of file along with its path and file opening mode.

file = open(name, mode)

The mode parameter decides how the file is to be treated. Default mode is 'r' which means that it is now possible to read data from the file. To store data in it, mode parameter should be set to 'w'.

Other supported values of mode parameter and their significance are listed in following table:

open() FUNCTION

Character	Purpose
r	Opens a file for reading only. (default)
w	Opens a file for writing only, deleting earlier contents
a	Opens a file for appending.
t	opens file in text format (default)
b	Opens a file in binary format.
+	Opens a file for simultaneous reading and writing.
x	opens file for exclusive creation.

The open() function returns a file like object representing any stream such as file, byte stream, socket or pipe etc. The file object supports various methods for operations on underlying data stream.

write() METHOD

The following statement opens file.txt in write mode.

Step 1: f=open("file.txt", "w")

Next we have to put certain data in the file. The write() method stores a string in the file.

Step 2: f.write(("Digital dreams ICT Academy file IO python class"))

Make sure that you close the file object in the end as shown below.

Step 3: f.close()

The "file.txt" will now be created in current folder, if it doesn't exist. Try opening it using any text editor to confirm that it contains above text.

write() METHOD

Alternatively:

```
with open('C://input/write.txt', 'w') as file_handle:
    file_handle.write("This is just an alternative way of writing to a file in python")
    file_handle.close()
```

writelines() METHOD

The file object also has writelines() method to write items in a list object to a file. The newline characters ("\n) should be part of the string. Let's create a list of strings.

```
lines=["Fantabulous! You are doing well.\n", "Are you normal? Ooin! \n", "We want to write in lines.\n", "Complex is better than complicated.\n"]

f=open("file.txt","w")

f.writelines(lines)

f.close()
```

writelines() METHOD

Alternatively:

```
lines = ["Beautiful is better than ugly.\n",
        "Explicit is better than implicit.\n",
        "Simple is better than complex.\n",
        "Complex is better than complicated.\n"]

with open('C://input/with.txt', 'w') as file_handle:
    for list_item in lines:
        file_handle.writelines(list_item)
    file_handle.close()
```

read() METHOD

To read data from a file, we need to open it in ‘r’ mode.

```
f=open('python.txt','r')
```

The read() method reads certain number of characters from file's current reading position. To read first 15 characters in file:

```
f.read(15)
```

read() METHOD

Alternatively:

```
with open('C://input/write.txt', 'r') as file_handle:
    get = file_handle.read()
    print(get)
    file_handle.close()
```

readline() METHOD

This method reads current line till it encounters newline character.

```
f=open(file.txt', 'r')
f.readline()
f.close
```

To read file line by line until all lines are read,

```
f=open("file.txt","r")
while True:
    line=f.readline()
    if line==":break
    print (line)
f.close()
```

readlines() METHOD

This method reads all lines and returns a list object.

```
f=open('file.txt', 'r')
ln = f.readlines()
print(ln)
```

EXCEPTION in **FILE** HANDLING

File operation is subject to occurrence of exception. If file could not be opened, OSError is raised and if it is not found, FileNotFoundError is raised.

Let's try to read from a non-existing file

f=open("nofile.txt", "r")

We will have an exception below:

```
Traceback (most recent call last):
 File "<pyshell#8>", line 1, in <module>
  f=open("nofile.txt","r")
FileNotFoundError: [Errno 2] No such file or directory:
  'nofile.txt'
```

Such operations should always be provided an exception handling mechanism.

Let's see an example in the next slide:

EXCEPTION in **FILE** HANDLING

```
try:
    f=open("python.txt","r")
    while True:
        line=f.readline()
        if line==":break
        print (line, end=")
except FileNotFoundError:
    print ("File is not found")
else:
    f.close()
```

FILE **object** as ITERATOR

The file object is a data stream that supports next() method to read file line by line. When end of file is encountered, StopIteration exception is raised.

```
f=open("python.txt","r")
while True:
    try:
        line=next(f)
        print (line, end="")
    except StopIteration:
        break
f.close()
```

append MODE

When a file is opened with "w" mode its contents are deleted if it already exists. In order to add more data to existing file use "a" mode (append mode).

```
f=open("python.txt","a")
```

If additional data is now written, it gets added to the end of file.

For example:

append MODE

```
f=open('python.txt','a')
f.write("We are just trying to add more content to
   an existing file .\n")
f.close()
```

The file will have additional line at the end.

+ mode for simultaneous **READ/WRITE**

"w" mode or "a" mode allows data to be written into and cannot be read from. On the other hand "r" mode allows reading only but doesn't allow write operation. In order to perform simultaneous read/write operations, use "r+" or "w+" mode.

SOME PYTHON OS MODULE

THE **OS** MODULE

The Python OS module lets us work with files and directories. We have been using it a lot to get to the Desktop in our examples. But it is much more. Let's discuss the important functions/methods it offers.

Let's check the dir() on this module:

*print**(dir**(os))*

That will print out python's ***os*** modules

NB: Make sure you must have imported os

CREATING DIRECTORY

We can create a new directory using the mkdir() function from the OS module.

```
import os

os.mkdir("C://pythonfiles")
```

CREATING DIRECTORY

To be able create a directory recursively we make use of ***makedirs()***. Creating recursively means creating folders inside another. It is like ***mkdir()***, but ***mkdir()*** can only create one at a time. ***makedirs()*** mandates that all intermediate-level directories contain the leaf directory.

For example:

os.makedirs("C://pythonfiles/type/word")

CHANGING THE CURRENT WORKING DIRECTORY

We must first change the current working directory to a newly created one before doing any operations in it. This is done using the ***chdir()*** function.

This Python os module changes the current working directory to the path we specify.

```
os.chdir('C:\\input')
```

CHANGING THE CURRENT WORKING DIRECTORY

There is a getcwd() function in the OS module using which we can confirm if the current working directory has been changed or not.

In order to set the current directory to the parent directory use ".." as the argument in the chdir() function. E.g. **os.chdir("..")**

REMOVING a **DIRECTORY**

The **rmdir()** function in the OS module removes the specified directory either with an absolute or relative path.

NB: We can not remove the current working directory. Also, for a directory to be removed, it should be empty. To remove a current directory you must first change the directory to it's parent.

REMOVING a **DIRECTORY**

removedirs(path)

This Python os Module will remove directories recursively

LIST FILES AND SUB-DIRECTORIES

The listdir() function returns the list of all files and directories in the specified directory.

For example:

```
a = os.listdir('C://input')

print(a)
```

Access
(path,mode)

This method uses the real uid/gid to test for access to a path. If access is allowed, it returns True. Else, it returns False. The first argument is the path; the second is the mode. The mode can take one of four values:

- os.F_OK — Found
- os.R_OK — Readable
- os.W_OK — Writable
- os.X_OK — Executable

Access
(path,mode)

Example:

```
os.chdir('C://input/more')

a = os.access('change.txt', os.R_OK)

print(a)
```

WORKING WITH CSV FILES IN PYTHON

Python - CSV

CSV (Comma Separated Values) is a simple file format used to store tabular data, such as a spreadsheet or database. A CSV file stores tabular data (numbers and text) in plain text. Each line of the file is a data record. Each record consists of one or more fields, separated by commas. The use of the comma as a field separator is the source of the name for this file format.

For working CSV files in python, there is an inbuilt module called csv. We will have to import it.

WRITING to a CSV FILE

writer():

This function in csv module returns a writer object that converts data into a delimited string and stores in a file object. The function needs a file object created with ***open()*** function and with write permission as a parameter.

The writer class has following methods:

writerow():

This function writes items in a sequence (list, tuple or string) separating them by comma character. We use it to write the first row which is nothing but the field names.

writerows():

This function writes each sequence in a list as a comma separated line of items in the file. It's used to write multiple rows at once.

WRITING to a CSV FILE

To start writing CSV file create the writer class using following statement:

```
import csv
fields = ['Name', 'Branch', 'Year', 'CGPA']
rows = [ [‘Ifeanyi', ‘CEE', ‘3', ‘4.4'],
        [‘Felix', ‘MECH', '2', ‘3.1'],
        [‘Adanne', ‘EEE', '2', ‘4.3'],
        [‘Lucky', ‘MME', '1', ‘4.5'],
        [‘Ngozi', ‘CVE', '3', ‘2.8'],
        [‘Festus', ‘AGE', '2', ‘3.1']]
filename = "records.csv"
```

WRITING to a CSV FILE

```
with open(filename, 'w') as csv_file:
    try:
        csv_writer = csv.writer(csv_file)
        # writing the fields
        csv_writer.writerow(fields)
        # writing the data rows
        csv_writer.writerows(rows)
        print("File successfully created")
    except FileExistsError as e:
        print(e, "An exception occurred while creating the csv file")
    else:
        csv_file.close()
```

WRITING to a CSV FILE

If you open the file using the Excel App, you will notice empty rows added after each row, to correct this it is necessary to open the file with a newline attribute assigned as empty, such as:

Open(filename, 'w', newline=' ')

WRITING a **DICTIONARY** to a CSV FILE

Let's define our dictionary first of all:

```
mydict = [{'branch': 'CEE', 'cgpa': '9.0', 'name': 'Chisom',
'year': '2'}, {'branch': 'CVE', 'cgpa': '9.1', 'name': 'Sam',
'year': '2'}, {'branch': 'ICT', 'cgpa': '9.3', 'name': 'Adaobi',
'year': '2'}, {'branch': 'MECH', 'cgpa': '9.5', 'name':
'Sonia', 'year': '1'}, {'branch': 'MME', 'cgpa': '7.8', 'name':
'Prosper', 'year': '3'}, {'branch': 'EEE', 'cgpa': '9.1', 'name':
'Emma', 'year': '2'}]
```

WRITING a **DICTIONARY** to a CSV FILE

Let's define our dictionary first of all:

```
mydict = [{'branch': 'CEE', 'cgpa': '9.0', 'name': 'Chisom',
'year': '2'}, {'branch': 'CVE', 'cgpa': '9.1', 'name': 'Sam',
'year': '2'}, {'branch': 'ICT', 'cgpa': '9.3', 'name': 'Adaobi',
'year': '2'}, {'branch': 'MECH', 'cgpa': '9.5', 'name':
'Sonia', 'year': '1'}, {'branch': 'MME', 'cgpa': '7.8', 'name':
'Prosper', 'year': '3'}, {'branch': 'EEE', 'cgpa': '9.1', 'name':
'Emma', 'year': '2'}]
```

WRITING a **DICTIONARY** to a CSV FILE

```
fields = ['name', 'branch', 'year', 'cgpa']

filename = "C://input/records.csv"

# writing to csv file

with open(filename, 'w', newline=' ') as csvfile:

    # creating a csv dict writer object

    writer = csv.DictWriter(csvfile, fieldnames = fields)

    # writing headers (field names)

    writer.writeheader()

    # writing data rows

    writer.writerows(mydict)
```

READING a **CSV** file

```
filename = "C://input/records.csv“

# initializing the titles and rows list

fields = []

rows = []

# reading csv file

with open(filename, ‘r’, newline=‘ ’) as csvfile:

   # creating a csv reader object

   csvreader = csv.reader(csvfile)
```

READING a CSV file

Continuation from previous slide:

```
# extracting field names through first row
    fields = next(csv_reader)
    # extracting each data row one by one
    for row in csv_reader:
        rows.append(row)
    # get total number of rows
    print("Total no. of rows: %d"%(csv_reader.line_num))
```

READING a CSV file

Continuation from previous slide:

```
# printing the field names
print('Field names are:' + ', '.join(field for field in fields))
#  printing first 5 rows
print('\n First 5 rows are: \n')
for row in rows[:5]:
    # parsing each column of a row
    for col in row:
        print(" %8s " % col),
    print('\n')
```

READING from a CSV FILE as **DICTIONARY**

filename = "C://input/records.csv"

writing to csv file

with open(filename, 'w', newline=' ') as csv_file:

csv_read = csv.DictReader(csv_file)

The DictReader class provides fieldnames attribute. It returns the dictionary keys used as header of file.

print(csv_read. fieldnames)

Use loop over the DictReader object to fetch individual dictionary objects.

for row in csv_read:

print (row)

DIALECT class

The csv module also defines a dialect class. Dialect is set of standards used to implement CSV protocol. The list of dialects available can be obtained by *list_dialects()* function.

csv.list_dialects()

Dialect objects support following attributes:

DIALECT class

Attributes	Description
delimiter	A one-character string used to separate fields. It defaults to ','.
doublequote	Controls how instances of quotechar appearing inside a field should themselves be quoted.
escapechar	A one-character string used by the writer to escape the delimiter.
lineterminator	The string used to terminate lines produced by the writer. It defaults to '\r\n'.
quotechar	A one-character string used to quote fields containing special characters, such as the delimiter or quotechar, or which contain new-line characters. It defaults to '"'.
quoting	Controls when quotes should be generated by the writer and recognised by the reader. It defaults to QUOTE_MINIMAL.
strict	When True, raise exception Error on bad CSV input. The default is False.

DIALECT class

To register new dialect type, use register_dialect() function as shown:

```
csv.register_dialect(
    'mydialect',
    delimiter = ',',
    quotechar = '"',
    doublequote = True,
    skipinitialspace = True,
    lineterminator = '\r\n',
    quoting = csv.QUOTE_MINIMAL)
```

DIALECT class

Now, while defining a csv.reader or csv.writer object, we can specify the dialect like this:

csvreader = csv.reader(csvfile, dialect='mydialect')

DIALECT class

The csv module defines the following constants:

Constant	Description
csv.QUOTE_ALL	Instructs writer objects to quote all fields.
csv.QUOTE_MINIMAL	Instructs writer objects to only quote those fields which contain special characters such as delimiter, quotechar or any of the characters in lineterminator.
csv.QUOTE_NONNUMERIC	Instructs writer objects to quote all non-numeric fields. Instructs the reader to convert all non-quoted fields to type float.
csv.QUOTE_NONE	Instructs writer objects to never quote fields.

ASSIGNMENT

- Write a program that collects a users information and save it to a file marked by the users full name.
- Write a program that when a user supplies his name, you will read his file and print out his information's previously supplied.