File Handling in Python

Python too supports file handling and allows users to handle files i.e., to read and write files, along with many other file handling options, to operate on files. The concept of file handling has stretched over various other languages, but the implementation is either complicated or lengthy, but alike other concepts of Python, this concept here is also easy and short. Python treats file differently as text or binary and this is important. Each line of code includes a sequence of characters and they form text file. Each line of a file is terminated with a special character, called the EOL or End of Line characters like comma {,} or newline character. It ends the current line and tells the interpreter a new one has begun. Let's start with Reading and Writing files.

Working of open() function

We use **open ()** function in Python to open a file in read or write mode. As explained above, open ( ) will return 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. So, the syntax being: **open(filename, mode)**. There are three kinds of mode, that Python provides and how files can be opened:

- " **r** ", for reading.
- " **w** ", for writing.
- " **a** ", for appending.
- " **r+** ", for both reading and writing

One must 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 "geek", will be opened with the reading mode.
file = open('geek.txt', 'r')
# This will print every line one by one in the file
for each in file:
    print (each)
```

The open command will open the file in the read mode and the for loop will print each line present in the file.

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)
```

Creating a file using 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('geek.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

Let's see how the append mode works:

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

There are also various other commands in file handling that is used to handle various tasks like:

```
rstrip(): This function strips each line of a file off spaces from the
right-hand side.

lstrip(): This function strips each line of a file off spaces from the
left-hand side.
```

It is designed to provide much cleaner syntax and exceptions handling when you are working with code. That explains why it's good practice to use them with a statement where applicable. This is helpful because using this method any files opened will be closed automatically after one is done, so auto-cleanup.
Example:

```
# Python code to illustrate with()
with open("file.txt") as file:
    data = file.read()
# do something with data
```

Using write along with with() function

We can also use write function along with with() function:

```
# Python code to illustrate with() alongwith write()
with open("file.txt", "w") as f:
    f.write("Hello World!!!")
```

split() using file handling

We can also split lines using file handling in Python. This splits the variable when space is encountered. You can also split using any characters as we wish. Here is the code:

```
# Python code to illustrate split() function
with open("file.text", "r") as file:
    data = file.readlines()
    for line in data:
        word = line.split()
        print word
```

Open a File in Python

Python provides inbuilt functions for creating, writing and reading files. There are two types of files that can be handled in Python, normal text files and binary files (written in binary language, `0s` and `1s`).

- **Text files:** In this type of file, Each line of text is terminated with a special character called **EOL (End of Line)**, which is the new line character (`'\n'`) in Python by default.
- **Binary files:** In this type of file, there is no terminator for a line and the data is stored after converting it into machine-understandable binary language.

Refer the below articles to get the idea about basics of file handling.

- *Basics of file handling*
- *Reading and Writing to file*

Opening a file

Opening a file refers to getting the file ready either for reading or for writing. This can be done using the `open()` function. This function returns a file object and takes two arguments, one that accepts the file name and another that accepts the mode(Access Mode). Now, the question arises what is access mode?
Access modes govern the type of operations possible in the opened file. It refers to how the file will be used once it's opened. These modes also define the location of the **File Handle** in the file. **File handle** is like a cursor, which defines from where the data has to be read or written in the file. There are 6 access modes in python.

- **Read Only ('r'):** Open text file for reading. The handle is positioned at the beginning of the file. If the file does not exist, raises I/O error. This is also the default mode in which the file is opened.
- **Read and Write ('r+'):** Open the file for reading and writing. The handle is positioned at the beginning of the file. Raises I/O error if the file does not exist.
- **Write Only ('w'):** Open the file for writing. For existing file, the data is truncated and over-written. The handle is positioned at the beginning of the file. Creates the file if the file does not exist.
- **Write and Read ('w+'):** Open the file for reading and writing. For existing file, data is truncated and over-written. The handle is positioned at the beginning of the file.

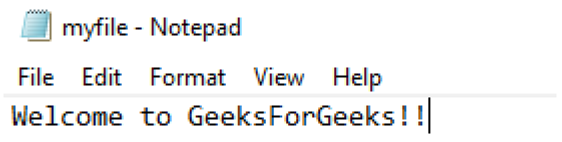
- **Append Only ('a'):** Open the file for writing. The file is created if it does not exist. The handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data.
- **Append and Read ('a+'):** Open the file for reading and writing. The file is created if it does not exist. The handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data.

Syntax:

```
File_object = open(r"File_Name", "Access_Mode")
```

Note: The file should exist in the same directory as the Python script, otherwise full address of the file should be written.

Example #1: Suppose the text file looked like this



```
Welcome to GeeksForGeeks!!
```

We want to read the content of the file using Python.

```
# Python program to demonstrate
# opening a file
# Open function to open the file "myfile.txt"
# (same directory) in read mode and store
# it's reference in the variable file1


file1 = open("myfile.txt")


# Reading from file
print(file1.read())


file1.close()
```

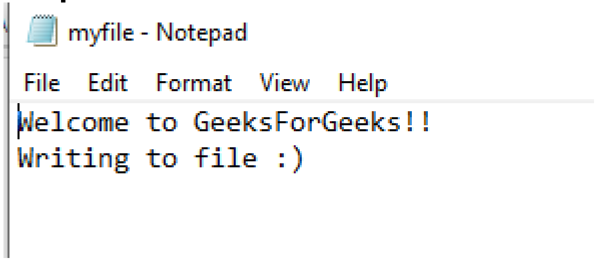
Output:

```
Welcome to GeeksForGeeks!!
```

Example #2: Suppose we want to write more data to the above file using Python.

```
# Python program to demonstrate
# opening a file
# Open function to open the file "myfile.txt"
# (same directory) in append mode and store
# it's reference in the variable file1
file1 = open("myfile.txt", "a")

# Writing to file
file1.write("\nWriting to file :)")

# Closing file
file1.close()
```

Output:



```
Welcome to GeeksForGeeks!!
Writing to file :)
```

How to read from a file in Python

Python provides inbuilt functions for creating, writing and reading files. There are two types of files that can be handled in python, normal text files and binary files (written in binary language, 0s and 1s).

- **Text files:** In this type of file, Each line of text is terminated with a special character called EOL (End of Line), which is the new line character ('\n') in python by default.
- **Binary files:** In this type of file, there is no terminator for a line and the data is stored after converting it into machine-understandable binary language.

Note: To know more about file handling click here.

Access mode

Access modes govern the type of operations possible in the opened file. It refers to how the file will be used once it's opened. These modes also define the location of the File Handle in the file. File handle is like a cursor, which defines from where the data has to be read or written in the file. Different access modes for reading a file are –

1. **Read Only ('r') :** Open text file for reading. The handle is positioned at the beginning of the file. If the file does not exists, raises I/O error. This is also the default mode in which file is opened.
2. **Read and Write ('r+') :** Open the file for reading and writing. The handle is positioned at the beginning of the file. Raises I/O error if the file does not exists.
3. **Append and Read ('a+') :** Open the file for reading and writing. The file is created if it does not exist. The handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data.

Note: To know more about access mode click here.

Opening a File

It is done using the `open()` function. No module is required to be imported for this function.
Syntax:

```
File_object = open(r"File_Name", "Access_Mode")
```

The file should exist in the same directory as the python program file else, full address of the file should be written on place of filename.

Note: The `r` is placed before filename to prevent the characters in filename string to be treated as special character. For example, if there is \temp in the file address, then \t is treated as the tab character and error is raised of invalid address. The r makes the string raw, that is, it tells that the string is without any special characters. The r can be ignored if the file is in same directory and address is not being placed.

```
# Open function to open the file "MyFile1.txt"
# (same directory) in read mode and
file1 = open("MyFile.txt", "r")


# store its reference in the variable file1
# and "MyFile2.txt" in D:\Text in file2
file2 = open(r"D:\Text\MyFile2.txt", "r+")
```

Here, file1 is created as object for MyFile1 and file2 as object for MyFile2.

Closing a file

`close()` function closes the file and frees the memory space acquired by that file. It is used at the time when the file is no longer needed or if it is to be opened in a different file mode.
Syntax:

```
File_object.close()
```

```
# Opening and Closing a file "MyFile.txt"

# for object name file1.

file1 = open("MyFile.txt", "r")

file1.close()
```

Reading from a file

There are three ways to read data from a text file.

- **read() :** Returns the read bytes in form of a string. Reads n bytes, if no n specified, reads the entire file.
- `File_object.read([n])`
- **readline() :** Reads a line of the file and returns in form of a string.For specified n, reads at most n bytes. However, does not reads more than one line, even if n exceeds the length of the line.
- `File_object.readline([n])`
- **readlines() :** Reads all the lines and return them as each line a string element in a list.
- `File_object.readlines()`

Note: `'\n'` is treated as a special character of two bytes.
Example:

```
# Program to show various ways to

# read data from a file.


# Creating a file

file1 = open("myfile.txt", "w")

L = ["This is Delhi \n", "This is Paris \n", "This is London \n"]


# Writing data to a file

file1.write("Hello \n")

file1.writelines(L)

file1.close()  # to change file access modes


file1 = open("myfile.txt", "r+")


print("Output of Read function is ")

print(file1.read())

print()
```

```
# seek(n) takes the file handle to the nth
# bite from the beginning.
file1.seek(0)

print("Output of Readline function is ")
print(file1.readline())
print()

file1.seek(0)

# To show difference between read and readline
print("Output of Read(9) function is ")
print(file1.read(9))
print()

file1.seek(0)

print("Output of Readline(9) function is ")
print(file1.readline(9))
print()

file1.seek(0)

# readlines function
print("Output of Readlines function is ")
print(file1.readlines())
print()
file1.close()
```

Output:

```
Output of Read function is
Hello
This is Delhi
This is Paris
This is London



Output of Readline function is

Hello



Output of Read(9) function is

Hello

Th


Output of Readline(9) function is

Hello



Output of Readlines function is

['Hello \n', 'This is Delhi \n', 'This is Paris \n', 'This is London
\n']
```

With statement

`with` statement in Python is used in exception handling to make the code cleaner and much more readable. It simplifies the management of common resources like file streams. Unlike the above implementations, there is no need to call `file.close()` when using with statement. The `with` statement itself ensures proper acquisition and release of resources.

Syntax:

```
with open filename as file:
```

```
# Program to show various ways to

# read data from a file.

L = ["This is Delhi \n", "This is Paris \n", "This is London \n"]

# Creating a file

with open("myfile.txt", "w") as file1:

    # Writing data to a file

    file1.write("Hello \n")

    file1.writelines(L)
    file1.close()  # to change file access modes
with open("myfile.txt", "r+") as file1:
    # Reading form a file
    print(file1.read())
```

Output:

```
Hello

This is Delhi

This is Paris

This is London
```

Writing to file in Python

Python provides inbuilt functions for creating, writing and reading files. There are two types of files that can be handled in python, normal text files and binary files (written in binary language, 0s and 1s).

- **Text files:** In this type of file, Each line of text is terminated with a special character called EOL (End of Line), which is the new line character ('\n') in python by default.
- **Binary files:** In this type of file, there is no terminator for a line and the data is stored after converting it into machine-understandable binary language.

Note: To know more about file handling click here.

Table of content

Access mode

Access modes govern the type of operations possible in the opened file. It refers to how the file will be used once it's opened. These modes also define the location of the File Handle in the file. File handle is like a cursor, which defines from where the data has to be read or written in the file. Different access modes for reading a file are –

1. **Write Only ('w') :** Open the file for writing. For an existing file, the data is truncated and over-written. The handle is positioned at the beginning of the file. Creates the file if the file does not exist.
2. **Write and Read ('w+') :** Open the file for reading and writing. For an existing file, data is truncated and over-written. The handle is positioned at the beginning of the file.

3. **Append Only (‘a’) :** Open the file for writing. The file is created if it does not exist. The handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data.

Note: To know more about access mode click here.

Opening a File

It is done using the `open()` function. No module is required to be imported for this function.

Syntax:

```
File_object = open(r"File_Name", "Access_Mode")
```

The file should exist in the same directory as the python program file else, full address of the file should be written on place of filename.

Note: The `r` is placed before filename to prevent the characters in filename string to be treated as special character. For example, if there is \temp in the file address, then \t is treated as the tab character and error is raised of invalid address. The r makes the string raw, that is, it tells that the string is without any special characters. The r can be ignored if the file is in same directory and address is not being placed.

```
# Open function to open the file "MyFile1.txt"
# (same directory) in read mode and
file1 = open("MyFile.txt", "w")


# store its reference in the variable file1
# and "MyFile2.txt" in D:\Text in file2
file2 = open(r"D:\Text\MyFile2.txt", "w+")
```

Here, file1 is created as object for MyFile1 and file2 as object for MyFile2.

Closing a file

`close()` function closes the file and frees the memory space acquired by that file. It is used at the time when the file is no longer needed or if it is to be opened in a different file mode.

Syntax:

```
File_object.close()
```

```
# Opening and Closing a file "MyFile.txt"
# for object name file1.
file1 = open("MyFile.txt", "w")
```

```
file1.close()
```

Writing to file

There are two ways to write in a file.

1. **write() :** Inserts the string str1 in a single line in the text file.
2. `File_object.write(str1)`
3. **writelines() :** For a list of string elements, each string is inserted in the text file. Used to insert multiple strings at a single time.
4. `File_object.writelines(L) for L = [str1, str2, str3]`

Note: `‘\n’` is treated as a special character of two bytes.
Example:

```
# Python program to demonstrate

# writing to file


# Opening a file

file1 = open('myfile.txt', 'w')

L = ["This is Delhi \n", "This is Paris \n", "This is London \n"]

s = "Hello\n"


# Writing a string to file

file1.write(s)


# Writing multiple strings

# at a time

file1.writelines(L)


# Closing file

file1.close()


# Checking if the data is

# written to file or not
```

```
file1 = open('myfile.txt', 'r')

print(file1.read())

file1.close()
```

Output:

```
Hello

This is Delhi

This is Paris

This is London
```

Appending to a file

When the file is opened in append mode, the handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data. Let's see the below example to clarify the difference between write mode and append mode.

```
# Python program to illustrate

# Append vs write mode

file1 = open("myfile.txt", "w")

L = ["This is Delhi \n", "This is Paris \n", "This is London \n"]

file1.writelines(L)

file1.close()

# Append-adds at last

file1 = open("myfile.txt", "a")  # append mode

file1.write("Today \n")

file1.close()


file1 = open("myfile.txt", "r")

print("Output of Readlines after appending")

print(file1.read())

print()

file1.close()


# Write-Overwrites

file1 = open("myfile.txt", "w")  # write mode

file1.write("Tomorrow \n")

file1.close()
```

```
file1 = open("myfile.txt", "r")

print("Output of Readlines after writing")

print(file1.read())

print()

file1.close()
```

Output:

```
Output of Readlines after appending

This is Delhi

This is Paris

This is London

Today




Output of Readlines after writing

Tomorrow
```

With statement

`with` statement in Python is used in exception handling to make the code cleaner and much more readable. It simplifies the management of common resources like file streams. Unlike the above implementations, there is no need to call `file.close()` when using with statement. The `with` statement itself ensures proper acquisition and release of resources.

Syntax:

```
with open filename as file:
```

```
# Program to show various ways to

# write data to a file using with statement

L = ["This is Delhi \n", "This is Paris \n", "This is London \n"]

# Writing to file

with open("myfile.txt", "w") as file1:

    # Writing data to a file

    file1.write("Hello \n")

    file1.writelines(L)

# Reading from file

with open("myfile.txt", "r+") as file1:

    # Reading form a file
    print(file1.read())
```

Output:

```
Hello

This is Delhi

This is Paris

This is London
```

Working with csv files in Python

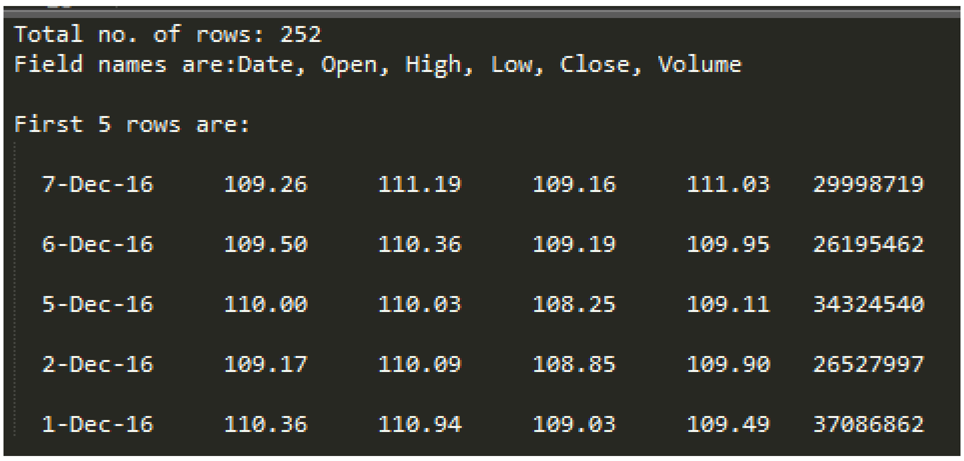
This article explains how to load and parse a CSV file in Python.

First of all, what is a 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.

Reading a CSV file

```
# importing csv module
import csv
# csv file name
filename = "aapl.csv"
# initializing the titles and rows list
fields = []
rows = []


# reading csv file
with open(filename, 'r') as csvfile:
    # creating a csv reader object
    csvreader = csv.reader(csvfile)


    # extracting field names through first row
    fields = next(csvreader)


    # extracting each data row one by one
    for row in csvreader:
        rows.append(row)
    # get total number of rows
    print("Total no. of rows: %d"%(csvreader.line_num))


# printing the field names
print('Field names are:' + ', '.join(field for field in fields))


#  printing first 5 rows
print('\nFirst 5 rows are:\n')
for row in rows[:5]:
    # parsing each column of a row
    for col in row:
        print("%10s"%col),
    print('\n')
```

The output of above program looks like this:



```
Total no. of rows: 252
Field names are:Date, Open, High, Low, Close, Volume

First 5 rows are:

  7-Dec-16     109.26     111.19     109.16     111.03   29998719

  6-Dec-16     109.50     110.36     109.19     109.95   26195462

  5-Dec-16     110.00     110.03     108.25     109.11   34324540

  2-Dec-16     109.17     110.09     108.85     109.90   26527997

  1-Dec-16     110.36     110.94     109.03     109.49   37086862
```

The above example uses a CSV file aapl.csv which can be downloaded from here.
Run this program with the aapl.csv file in same directory.
Let us try to understand this piece of code.

- `with open(filename, 'r') as csvfile:`
- `    csvreader = csv.reader(csvfile)`

  Here, we first open the CSV file in READ mode. The file object is named as **csvfile**. The file object is converted to csv.reader object. We save the csv.reader object as **csvreader**.
- `fields = csvreader.next()`

csvreader is an iterable object. Hence, .next() method returns the current row and advances the iterator to the next row. Since the first row of our csv file contains the headers (or field names), we save them in a list called **fields**.

```
for row in csvreader:
        rows.append(row)
```

Now, we iterate through remaining rows using a for loop. Each row is appended to a list called **rows**. If you try to print each row, one can find that row is nothing but a list containing all the field values.

```
print("Total no. of rows: %d"%(csvreader.line_num))
```

csvreader.line_num is nothing but a counter which returns the number of rows which have been iterated.

Writing to a CSV file

```
# importing the csv module
import csv

# field names
fields = ['Name', 'Branch', 'Year', 'CGPA']

# data rows of csv file
rows = [ ['Nikhil', 'COE', '2', '9.0'],
         ['Sanchit', 'COE', '2', '9.1'],
         ['Aditya', 'IT', '2', '9.3'],
         ['Sagar', 'SE', '1', '9.5'],
         ['Prateek', 'MCE', '3', '7.8'],
         ['Sahil', 'EP', '2', '9.1']]

# name of csv file
filename = "university_records.csv"

# writing to csv file
with open(filename, 'w') as csvfile:
    # creating a csv writer object
    csvwriter = csv.writer(csvfile)

    # writing the fields
    csvwriter.writerow(fields)


    # writing the data rows
    csvwriter.writerows(rows)
```

Let us try to understand the above code in pieces.

- **fields** and **rows** have been already defined. fields is a list containing all the field names. **rows** is a list of lists. Each row is a list containing the field values of that row.
- `with open(filename, 'w') as csvfile:`
- `    csvwriter = csv.writer(csvfile)`

  Here, we first open the CSV file in WRITE mode. The file object is named as **csvfile**. The file object is converted to csv.writer object. We save the csv.writer object as **csvwriter**.
- `csvwriter.writerow(fields)`

  Now we use **writerow** method to write the first row which is nothing but the field names.
- ` csvwriter.writerows(rows)`

  We use **writerows** method to write multiple rows at once.

Writing a dictionary to a CSV file

```
# importing the csv module
import csv


# my data rows as dictionary objects
mydict =[{'branch': 'COE', 'cgpa': '9.0', 'name': 'Nikhil', 'year': '2'},
         {'branch': 'COE', 'cgpa': '9.1', 'name': 'Sanchit', 'year': '2'},
         {'branch': 'IT', 'cgpa': '9.3', 'name': 'Aditya', 'year': '2'},
         {'branch': 'SE', 'cgpa': '9.5', 'name': 'Sagar', 'year': '1'},
         {'branch': 'MCE', 'cgpa': '7.8', 'name': 'Prateek', 'year': '3'},
         {'branch': 'EP', 'cgpa': '9.1', 'name': 'Sahil', 'year': '2'}]


# field names
fields = ['name', 'branch', 'year', 'cgpa']


# name of csv file
filename = "university_records.csv"


# writing to csv file
```

```
with open(filename, 'w') 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)
```

In this example, we write a dictionary **mydict** to a CSV file.

-
  ```
  with open(filename, 'w') as csvfile:
  ```

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

  Here, the file object (**csvfile**) is converted to a DictWriter object.
  Here, we specify the **fieldnames** as an argument.

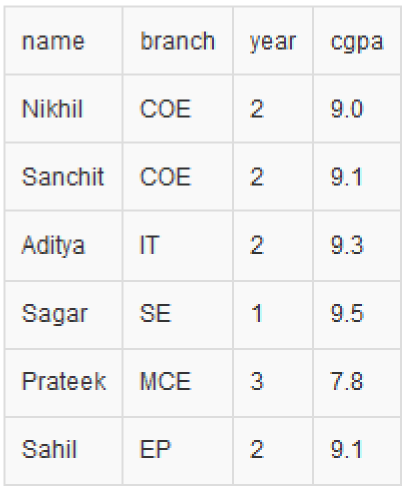
-
  ```
   writer.writeheader()
  ```

  writeheader method simply writes the first row of your csv file using the pre-specified fieldnames.

-
  ```
  writer.writerows(mydict)
  ```

  writerows method simply writes all the rows but in each row, it writes only the values(not keys).

So, in the end, our CSV file looks like this:



name	branch	year	cgpa
Nikhil	COE	2	9.0
Sanchit	COE	2	9.1
Aditya	IT	2	9.3
Sagar	SE	1	9.5
Prateek	MCE	3	7.8
Sahil	EP	2	9.1

Important Points:

- In csv modules, an optional *dialect* parameter can be given which is used to define a set of parameters specific to a particular *CSV format*. By default, csv module uses *excel* dialect which makes them compatible with excel spreadsheets. You can

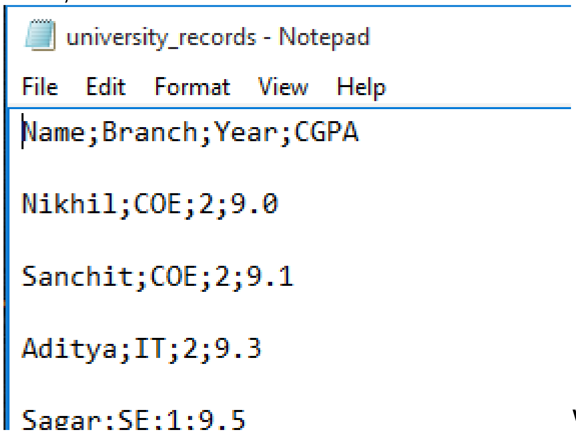
define your own dialect using **register_dialect** method.
Here is an example:

```
 csv.register_dialect(

'mydialect',

delimiter = ',',

quotechar = '"',

doublequote = True,

skipinitialspace = True,

lineterminator = '\r\n',

quoting = csv.QUOTE_MINIMAL)
```

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

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

- Now, consider that a CSV file looks like this in plain-text:



```
Name;Branch;Year;CGPA

Nikhil;COE;2;9.0

Sanchit;COE;2;9.1

Aditya;IT;2;9.3

Sagar:SE:1:9.5
```

We notice that the delimiter is not a comma but a semi-colon. Also, the rows are separated by two newlines instead of one. In such cases, we can specify the delimiter and line terminator as follows:

```
csvreader = csv.reader(csvfile, delimiter = ';', lineterminator = '\n\n')
```

So, this was a brief, yet concise discussion on how to load and parse CSV files in a python program.

Reading CSV files in Python

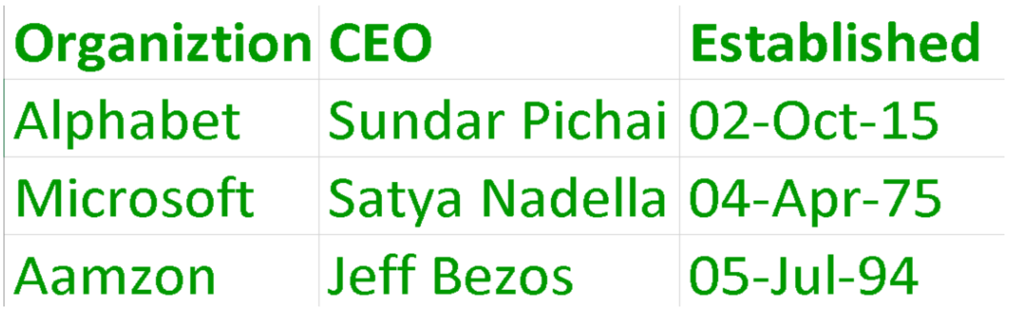
A **CSV (Comma Separated Values)** file is a form of plain text document which uses a particular format to organize tabular information. CSV file format is a bounded text document that uses a comma to distinguish the values. Every row in the document is a data log. Each log is composed of one or more fields, divided by commas. It is the most popular file format for importing and exporting spreadsheets and databases.

Reading a CSV File

There are various ways to read a CSV file that uses either the `csv` module or the `pandas` library.

- **csv Module:** The CSV module is one of the modules in Python which provides classes for reading and writing tabular information in CSV file format.
- **pandas Library:** The pandas library is one of the open-source Python libraries that provides high-performance, convenient data structures and data analysis tools and techniques for Python programming.

Reading a CSV File Format in Python:
Consider the below CSV file named 'Giants.CSV':



Organiztion	CEO	Established
Alphabet	Sundar Pichai	02-Oct-15
Microsoft	Satya Nadella	04-Apr-75
Aamzon	Jeff Bezos	05-Jul-94

- **USing csv.reader():** At first, the CSV file is opened using the `open()` method in 'r' mode(specifies read mode while opening a file) which returns the file object then it is read by using the `reader()` method of CSV module that returns the reader object that iterates throughout the lines in the specified CSV document.
  Note: The '`with`' keyword is used along with the open() method as it simplifies exception handling and automatically closes the CSV file.

```
import csv

# opening the CSV file

with open('Giants.csv', mode ='r')as file:


  # reading the CSV file

  csvFile = csv.reader(file)


  # displaying the contents of the CSV file

  for lines in csvFile:

        print(lines)
```

Output:

```
['Organiztion', 'CEO', 'Established']

['Alphabet', 'Sundar Pichai', '02-Oct-15']
```

```
['Microsoft', 'Satya Nadella', '04-Apr-75']
['Aamzon', 'Jeff Bezos', '05-Jul-94']
```

In the above program reader() method is used to read the Giants.csv file which maps the data into lists.

- **Using csv.DictReader() class:** It is similar to the previous method, the CSV file is first opened using the `open()` method then it is read by using the `DictReader` class of csv module which works like a regular reader but maps the information in the CSV file into a dictionary. The very first line of the file comprises of dictionary keys.

```
import csv


# opening the CSV file
with open('Giants.csv', mode ='r') as file:


        # reading the CSV file
        csvFile = csv.DictReader(file)


        # displaying the contents of the CSV file
        for lines in csvFile:
             print(lines)
```

Output:

OrderedDict([('Organiztion', 'Alphabet'), ('CEO', 'Sundar Pichai'), ('Established', '02-Oct-15')])
OrderedDict([('Organiztion', 'Microsoft'), ('CEO', 'Satya Nadella'), ('Established', '04-Apr-75')])
OrderedDict([('Organiztion', 'Aamzon'), ('CEO', 'Jeff Bezos'), ('Established', '05-Jul-94')])

- **Using pandas.read_csv() method:** It is very easy and simple to read a CSV file using pandas library functions. Here `read_csv()` method of pandas library is used to read data from CSV files.

```
import pandas


# reading the CSV file
csvFile = pandas.read_csv('Giants.csv')


# displaying the contents of the CSV file
print(csvFile)
```

Output:

```
Organiztion           CEO Established
```

```
0    Alphabet  Sundar Pichai   02-Oct-15
1   Microsoft  Satya Nadella   04-Apr-75
2      Aamzon     Jeff Bezos   05-Jul-94
```

In the above program, the csv_read() method of pandas library reads the Giants.csv file and maps its data into a 2D list.

Writing CSV files in Python

CSV (Comma Separated Values) is a simple file format used to store tabular data, such as a spreadsheet or database. 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.
Python provides an in-built module called `csv` to work with CSV files. There are various classes provided by this module for writing to CSV:

- Using csv.writer class
- Using csv.DictWriter class

Using csv.writer class

`csv.writer` class is used to insert data to the CSV file. This class returns a writer object which is responsible for converting the user's data into a delimited string. A csvfile object should be opened with `newline=''` otherwise newline characters inside the quoted fields will not be interpreted correctly.
Syntax: *csv.writer(csvfile, dialect='excel', **fmtparams)*
Parameters:
csvfile: *A file object with write() method.*
dialect (optional): *Name of the dialect to be used.*
fmtparams (optional): *Formatting parameters that will overwrite those specified in the dialect.*

`csv.writer` class provides two methods for writing to CSV. They are `writerow()` and `writerows()`.

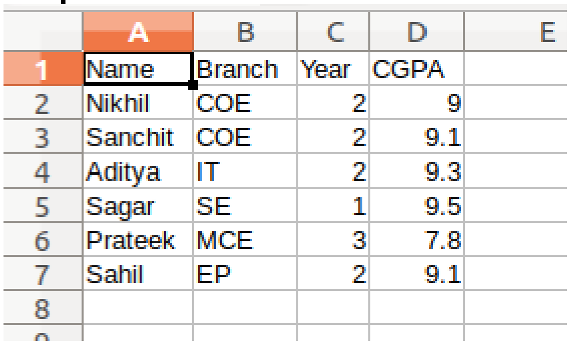
- **writerow():** This method writes a single row at a time. Field row can be written using this method.
  Syntax:
  ```
  writerow(fields)
  ```

- **writerows():** This method is used to write multiple rows at a time. This can be used to write rows list.
  Syntax:
  ```
  Writing CSV files in Python

  writerows(rows)
  ```

Example:

```
# Python program to demonstrate
# writing to CSV


import csv

# field names
fields = ['Name', 'Branch', 'Year', 'CGPA']

# data rows of csv file
rows = [ ['Nikhil', 'COE', '2', '9.0'],
         ['Sanchit', 'COE', '2', '9.1'],
         ['Aditya', 'IT', '2', '9.3'],
         ['Sagar', 'SE', '1', '9.5'],
         ['Prateek', 'MCE', '3', '7.8'],
         ['Sahil', 'EP', '2', '9.1']]

# name of csv file
filename = "university_records.csv"

# writing to csv file
with open(filename, 'w') as csvfile:
    # creating a csv writer object
    csvwriter = csv.writer(csvfile)

    # writing the fields
    csvwriter.writerow(fields)

    # writing the data rows
    csvwriter.writerows(rows)
```

Output:



	A	B	C	D	E
1	Name	Branch	Year	CGPA	
2	Nikhil	COE	2	9	
3	Sanchit	COE	2	9.1	
4	Aditya	IT	2	9.3	
5	Sagar	SE	1	9.5	
6	Prateek	MCE	3	7.8	
7	Sahil	EP	2	9.1	
8					

Using csv.DictWriter class

This class returns a writer object which maps dictionaries onto output rows.

Syntax: *csv.DictWriter(csvfile, fieldnames, restval=", extrasaction='raise', dialect='excel', *args, **kwds)*

Parameters:
csvfile: *A file object with write() method.*
fieldnames: *A sequence of keys that identify the order in which values in the dictionary should be passed.*
restval (optional): *Specifies the value to be written if the dictionary is missing a key in fieldnames.*
extrasaction (optional): *If a key not found in fieldnames, the optional extrasaction parameter indicates what action to take. If it is set to raise a ValueError will be raised.*
dialect (optional): *Name of the dialect to be used.*

csv.DictWriter provides two methods for writing to CSV. They are:

- **writeheader():** `writeheader()` method simply writes the first row of your csv file using the pre-specified fieldnames.
  Syntax:
  ```
  writeheader()
  ```
- `writerows():` `writerows` method simply writes all the rows but in each row, it writes only the values(not keys).
  Syntax:
  ```
  writerows(mydict)
  ```

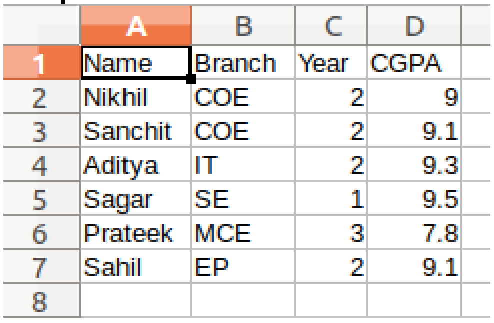
Example:

filter_none
brightness_4

```
# importing the csv module
import csv

# my data rows as dictionary objects
mydict =[{'branch': 'COE', 'cgpa': '9.0', 'name': 'Nikhil', 'year': '2'},
         {'branch': 'COE', 'cgpa': '9.1', 'name': 'Sanchit', 'year': '2'},
         {'branch': 'IT', 'cgpa': '9.3', 'name': 'Aditya', 'year': '2'},
         {'branch': 'SE', 'cgpa': '9.5', 'name': 'Sagar', 'year': '1'},
         {'branch': 'MCE', 'cgpa': '7.8', 'name': 'Prateek', 'year': '3'},
         {'branch': 'EP', 'cgpa': '9.1', 'name': 'Sahil', 'year': '2'}]

# field names
fields = ['name', 'branch', 'year', 'cgpa']

# name of csv file
filename = "university_records.csv"

# writing to csv file
with open(filename, 'w') 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)
```

Output:



	A	B	C	D
1	Name	Branch	Year	CGPA
2	Nikhil	COE	2	9
3	Sanchit	COE	2	9.1
4	Aditya	IT	2	9.3
5	Sagar	SE	1	9.5
6	Prateek	MCE	3	7.8
7	Sahil	EP	2	9.1
8				

Working With JSON Data in Python

Introduction of JSON in Python :

The full-form of JSON is JavaScript Object Notation. It means that a script (executable) file which is made of text in a programming language, is used to store and transfer the data. Python supports JSON through a built-in package called `json`. To use this feature, we import the json package in Python script. The text in JSON is done through quoted string which contains value in key-value mapping within `{ }`. It is similar to the dictionary in Python. JSON shows an API similar to users of Standard Library marshal and pickle modules and Python natively supports JSON features. For Example

```
# Python program showing

# use of json package


import json


# {key:value mapping}

a ={"name":"John",

   "age":31,

    "Salary":25000}


# conversion to JSON done by dumps() function

 b = json.dumps(a)


# printing the output

print(b)
```

Output:

```
{"age": 31, "Salary": 25000, "name": "John"}
```

As you can see, JSON supports primitive types, like strings and numbers, as well as nested lists, tuples and objects

```
# Python program showing that

# json support different primitive
# types

import json

# list conversion to Array
print(json.dumps(['Welcome', "to", "GeeksforGeeks"]))

# tuple conversion to Array
print(json.dumps(("Welcome", "to", "GeeksforGeeks")))

# string conversion to String
print(json.dumps("Hi"))

# int conversion to Number
print(json.dumps(123))

# float conversion to Number
print(json.dumps(23.572))

# Boolean conversion to their respective values
print(json.dumps(True))
print(json.dumps(False))

# None value to null
print(json.dumps(None))
```

Output:

```
["Welcome", "to", "GeeksforGeeks"]
["Welcome", "to", "GeeksforGeeks"]
"Hi"
123
23.572
true
false
```

```
null
```

Serializing JSON :
The process of encoding JSON is usually called **serialization**. This term refers to the transformation of data into a series of bytes (hence serial) to be stored or transmitted across a network. To handle the data flow in a file, the JSON library in Python uses `dump()` function to convert the Python objects into their respective JSON object, so it makes easy to write data to files. See the following table given below.

PYTHON OBJECT	JSON OBJECT
dict	object
list, tuple	array
str	string
int, long, float	numbers
True	true
False	false
None	null

Serialization Example :
Consider the given example of a Python object.

```
var = {
      "Subjects": {
                  "Maths":85,
                  "Physics":90
                   }
      }
```

Using Python's context managercontext manager, create a file named `Sample.json` and open it with write mode.

```
with open("Sample.json", "w") as p:

     json.dumps(var, p)
```

Here, the `dumps()` takes two arguments first, the data object to be serialized and second the object to which it will be written(Byte format).

Deserializing JSON :
The Deserialization is opposite of Serialization, i.e. conversion of JSON object into their respective Python objects. The `load()` method is used for it. If you have used Json data from another program or obtained as a string format of Json, then it can easily be deserialized with `load()`, which is usually used to load from string, otherwise the root object is in list or dict.

```
with open("Sample.json", "r") as read_it:

     data = json.load(read_it)
```

Deserialization Example :

```
json_var ="""

{

    "Country": {

        "name": "INDIA",

        "Languages_spoken": [

            {

                "names": ["Hindi", "English", "Bengali", "Telugu"]

            }

        ]

    }

}

"""

var = json.loads(json_var)
```

Encoding and Decoding :
Encoding is defined as converting the text or values into an encrypted form that can only be used by the desired user through decoding it. Here encoding and decoding is done for JSON (object)format. Encoding is also known as Serialization and Decoding is known as Deserialization. Python have a popular package for this operation. This package is known as **Demjson**. To install it follow the steps below.
For Windows

```
pip install demjson
```

For Ubuntu

```
sudo apt-get update

sudo apt-get install python-demjson
```

Encoding : The `encode()` function is used to convert the python object into a JSON string representation. Syntax

```
demjson.encode(self, obj, nest_level=0)
```

Code 1: Encoding using demjson package

filter_none

brightness_4

```
# storing marks of 3 subjects

var = [{"Math": 50, "physics":60, "Chemistry":70}]

print(demjson.encode(var))
```

Output:

```
[{"Chemistry":70, "Math":50, "physics":60}]
```

Decoding: The `decode()` function is used to convert the JSON oject into python format type. Syntax

```
demjson.decode(self, obj)
```

Code 2: Decoding using demjson package

```
var = '{"a":0, "b":1, "c":2, "d":3, "e":4}'

text = demjson.decode(var)
```

Output:

```
{'a': 0, 'b': 1, 'c': 2, 'd': 3, 'e': 4}
```

Code 3: Encoding using iterencode package

```
# Other Method of Encoding

json.JSONEncoder().encode({"foo": ["bar"]})

'{"foo": ["bar"]}'


# Using iterencode(object) to encode a given object.

for i in json.JSONEncoder().iterencode(bigobject):

    mysocket.write(i)
```

Code 4: Encoding and Decoding using `dumps()` and `loads()`

```
# To encode and decode operations

import json

var = {'age':31, 'height'= 6}

x = json.dumps(var)

y = json.loads(var)

print(x)

print(y)


# when performing from a file in disk

with open("any_file.json", "r") as readit:

    x = json.load(readit)

print(x)
```

Real World Example :
Let us take a real life example on the implementation of the JSON in python. A good source for practice purpose is JSON_placeholder, it provides great API requests package which we will be using in our example. To get started, follow these simple steps. Open Python IDE or CLI and create a new script file, name it sample.py.

```
import requests

import json


# Now we have to request our JSON data through

# the API package

res = requests.get("https://jsonplaceholder.typicode.com / todos")

var = json.loads(res.text)


# To view your Json data, type var and hit enter

var


# Now our Goal is to find the User who have

# maximum completed their task !!

# i.e we would count the True value of a

# User in completed key.

# {

    # "userId": 1,
    # "id": 1,
    # "title": "Hey",
    # "completed": false,  # we will count
                           # this for a user.
# }

# Note that there are multiple users with
# unique id, and their task have respective
# Boolean Values.

def find(todo):
    check = todo["completed"]
    max_var = todo["userId"] in users
    return check and max_var

# To find the values.

Value = list(filter(find, todos))

# To write these value to your disk

with open("sample.json", "w") as data:
    Value = list(filter(keep, todos))
    json.dump(Value, data, indent = 2)
```

Connect MySQL database using MySQL-Connector Python

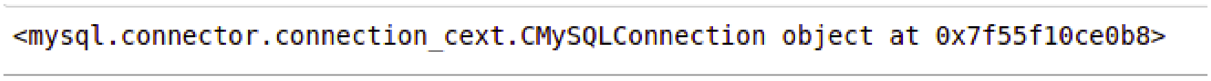
While working with Python we need to work with databases, they may be of different types like MySQL, SQLite, NoSQL, etc. In this article, we will be looking forward to how to connect MySQL databases using MySQL Connector/Python.

MySQL Connector module of Python is used to connect MySQL databases with the Python programs, it does that using the Python Database API Specification v2.0 (PEP 249). It uses the Python standard library and has no dependencies.

Connecting to the Database

In the following example we will be connecting to MySQL database using `connect()`
Example:

```
# Python program to connect
# to mysql databse


import mysql.connector


# Connecting from the server
conn = mysql.connector.connect(user = 'username',
                                host = 'localhost',
                               database = 'databse_name')

print(conn)

# Disconnecting from the server
conn.close()
```
Output:



```
<mysql.connector.connection_cext.CMySQLConnection object at 0x7f55f10ce0b8>
```

Also for the same, we can use `connection.MySQLConnection()` class instead of `connect()`:
Example:

```
# Python program to connect
# to mysql databse


from mysql.connector import connection


# Connecting to the server
conn = connection.MySQLConnection(user = 'username',
                                    host = 'localhost',
                                    database = 'database_name')

print(conn)

# Disconnecting from the server
conn.close()
```
Output:

```
<mysql.connector.connection.MySQLConnection object at 0x7f55e47a6358>
```

Another way is to pass the dictionary in the connect() function using ‘**’ operator:

Example:

```
# Python program to connect
# to mysql databse


from mysql.connector import connection


dict = {
  'user': 'root',
  'host': 'localhost',
  'database': 'College'
}

# Connecting to the server
conn = connection.MySQLConnection(**dict)

print(conn)

# Disconnecting from the server
conn.close()
```
Output:

```
<mysql.connector.connection.MySQLConnection object at 0x7f55e47a6630>
```

Python MySQL – Create Database

Python Database API (Application Program Interface) is the Database interface for the standard Python. This standard is adhered to by most Python Database interfaces. There are various Database servers supported by Python Database such as MySQL, GadFly, mSQL, PostgreSQL, Microsoft SQL Server 2000, Informix, Interbase, Oracle, Sybase etc. To connect with MySQL database server from Python, we need to import the `mysql.connector` interface.

Syntax:

```
CREATE DATABASE DATABASE_NAME
```

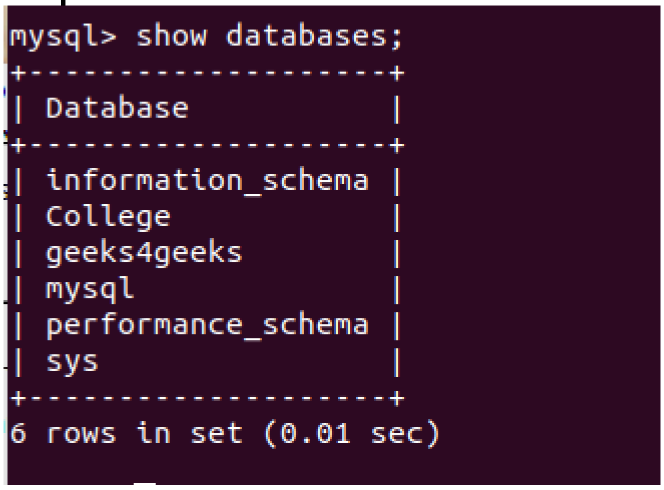
Example:

filter_none

brightness_4

```
# importing rquired libraries
import mysql.connector

dataBase = mysql.connector.connect(
  host ="localhost",
  user ="user",
  passwd ="gfg"
)

# preparing a cursor object
cursorObject = dataBase.cursor()

# creating database
cursorObject.execute("CREATE DATABASE geeks4geeks")
```

Output:



```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| College            |
| geeks4geeks        |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
6 rows in set (0.01 sec)
```

The above program illustrates the creation of MySQL database `geeks4geeks` in which host-name is `localhost`, the username is `user` and password is `gfg`.
Let's suppose we want to create a table in the database, then we need to connect to a database. Below is a program to create a table in the `geeks4geeks` database which was created in the above program.

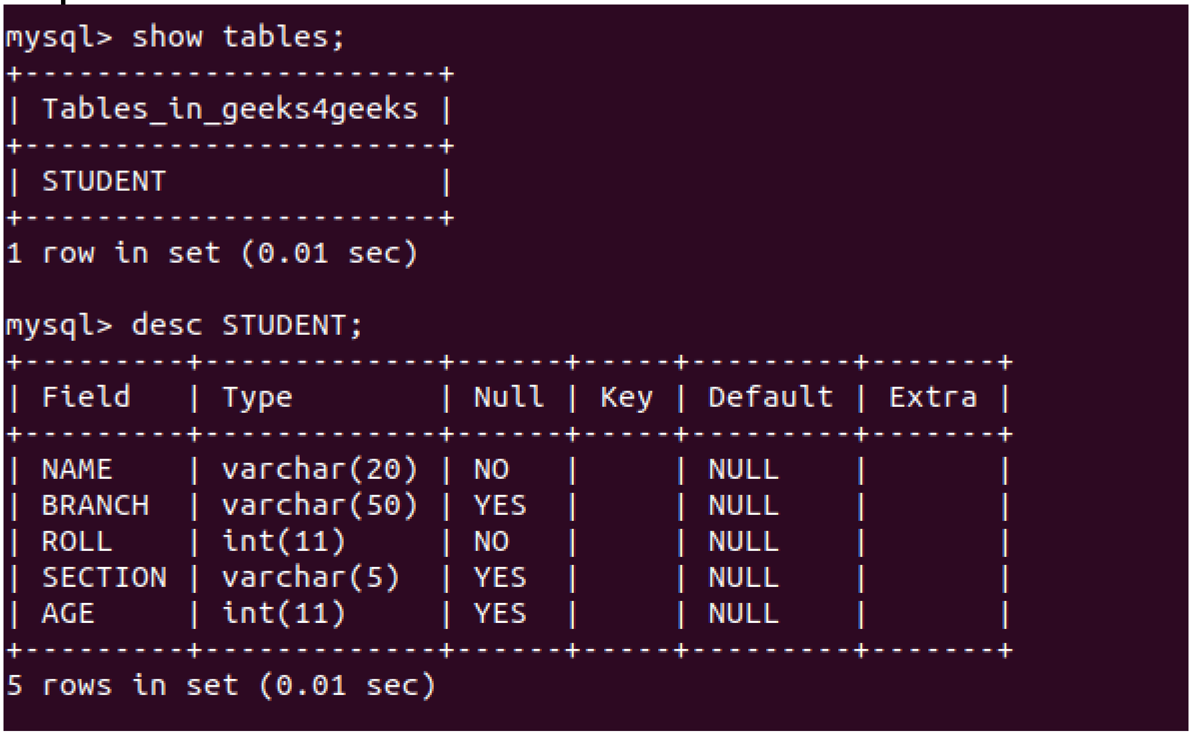
filter_none

brightness_4

```
# importing required library
import mysql.connector

# connecting to the database
dataBase = mysql.connector.connect(
                     host = "localhost",
                     user = "user",
                     passwd = "gfg",
                     database = "geeks4geeks" )
```

```
# preparing a cursor object
cursorObject = dataBase.cursor()

# creating table
studentRecord = """CREATE TABLE STUDENT (
                   NAME  VARCHAR(20) NOT NULL,
                   BRANCH VARCHAR(50),
                   ROLL INT NOT NULL,
                   SECTION VARCHAR(5),
                   AGE INT,
                   )"""

# table created
cursorObject.execute(studentRecord)

# disconnecting from server
dataBase.close()
```

Output:



```
mysql> show tables;
+-----------------------+
| Tables_in_geeks4geeks |
+-----------------------+
| STUDENT               |
+-----------------------+
1 row in set (0.01 sec)

mysql> desc STUDENT;
+---------+-------------+------+-----+---------+-------+
| Field   | Type        | Null | Key | Default | Extra |
+---------+-------------+------+-----+---------+-------+
| NAME    | varchar(20) | NO   |     | NULL    |       |
| BRANCH  | varchar(50) | YES  |     | NULL    |       |
| ROLL    | int(11)     | NO   |     | NULL    |       |
| SECTION | varchar(5)  | YES  |     | NULL    |       |
| AGE     | int(11)     | YES  |     | NULL    |       |
+---------+-------------+------+-----+---------+-------+
5 rows in set (0.01 sec)
```

Python Exception Handling

We have explored basic python till now from Set 1 to 4 (Set 1 | Set 2 | Set 3 | Set 4). Error in Python can be of two types i.e. Syntax errors and Exceptions. Errors are the problems in a program due to which the program will stop the execution. On the other hand, exceptions are raised when some internal events occur which changes the normal flow of the program.

The difference between Syntax Error and Exceptions

Syntax Error: As the name suggest this error is caused by wrong syntax in the code. It leads to the termination of the program.
Example

```
# initialize the amount variable
amount = 10000


# check that You are eligible to
#  purchase Dsa Self Paced or not
if(amount>2999)
    print("You are eligible to purchase Dsa Self Paced")
```

Output:

```
  File "/home/ac35380186f4ca7978956ff46697139b.py", line 4
    if(amount>2999)
                  ^
SyntaxError: invalid syntax
```

Exceptions: Exceptions are raised when the program is syntactically correct but the code resulted in an error. This error does not stop the execution of the program, however, it changes the normal flow of the program.
Example:

```
# initialize the amount variable
```

```
marks = 10000


# perform division with 0

a = marks / 0

print(a)
```

Output:

```
Traceback (most recent call last):
  File "/home/f3ad05420ab851d4bd106ffb04229907.py", line 4, in <module>
    a=marks/0
ZeroDivisionError: division by zero
```

In the above example raised the ZeroDivisionError as we are trying to divide a number by 0.

Note: Exception is the base class for all the exceptions in Python. You can check the exception hierarchy here.

Try and Except in Exception Handling

Let us try to access the array element whose index is out of bound and handle the corresponding exception.

```
# Python program to handle simple runtime error


a = [1, 2, 3]

try:

    print "Second element = %d" %(a[1])


    # Throws error since there are only 3 elements in array

    print "Fourth element = %d" %(a[3])


except IndexError:

    print "An error occurred"
```

Output:

```
Second element = 2

An error occurred
```

A try statement can have more than one except clause, to specify handlers for different exceptions. Please note that at most one handler will be executed.

```
# Program to handle multiple errors with one except statement
try :
    a = 3
    if a < 4 :

        # throws ZeroDivisionError for a = 3
        b = a/(a-3)

    # throws NameError if a >= 4
    print "Value of b = ", b

# note that braces () are necessary here for multiple exceptions
except(ZeroDivisionError, NameError):
    print "\nError Occurred and Handled"
```

Output:

```
Error Occurred and Handled
```

If you change the value of 'a' to greater than or equal to 4, the output will be

```
Value of b =

Error Occurred and Handled
```

The output above is so because as soon as python tries to access the value of b, NameError occurs.

Else Clause

In python, you can also use else clause on the `try-except` block which must be present after all the except clauses. The code enters the else block only if the try clause does not raise an exception.

```
# Program to depict else clause with try-except

# Function which returns a/b
def AbyB(a , b):
    try:
        c = ((a+b) / (a-b))
    except ZeroDivisionError:
        print "a/b result in 0"
    else:
        print c


# Driver program to test above function
AbyB(2.0, 3.0)
AbyB(3.0, 3.0)
```

The output for above program will be :

```
-5.0
a/b result in 0
```

Finally Keyword in Python

Python provides a keyword finally, which is always executed after try and except blocks. The finally block always executes after normal termination of try block or after try block terminates due to some exception.

Syntax:

```
try:
       # Some Code....


except:
       # optional block
       # Handling of exception (if required)


else:
       # execute if no exception


finally:
      # Some code .....(always executed)
```

Example:

```
# Python program to demonstrate finally


# No exception Exception raised in try block
try:
    k = 5//0 # raises divide by zero exception.
    print(k)


# handles zerodivision exception

except ZeroDivisionError:

    print("Can't divide by zero")


finally:

    # this block is always executed

    # regardless of exception generation.

    print('This is always executed')
```

Output:

```
Can't divide by zero

This is always executed
```

Raising Exception

The raise statement allows the programmer to force a specific exception to occur. The sole argument in raise indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from Exception).

```
# Program to depict Raising Exception


try:

    raise NameError("Hi there")  # Raise Error

except NameError:

    print "An exception"

    raise # To determine whether the exception was raised or not
```

The output of the above code will simply line printed as “An exception” but a Runtime error will also occur in the last due to raise statement in the last line. So, the output on your command line will look like

```
Traceback (most recent call last):

  File "003dff3d748c75816b7f849be98b91b8.py", line 4, in

    raise NameError("Hi there") # Raise Error

NameError: Hi there
```

Python Try Except

try() is used in **Error and Exception Handling**
There are two kinds of errors :

- **Syntax Error** : Also known as Parsing Errors, most basic. Arise when the Python parser is unable to understand a line of code.
- **Exception** : Errors which are detected during execution. eg – ZeroDivisionError.

List of Exception Errors :

- **IOError :** if file can't be opened
- **KeyboardInterrupt :** when an unrequired key is pressed by the user
- **ValueError :** when built-in function receives a wrong argument
- **EOFError :** if End-Of-File is hit without reading any data
- **ImportError :** if it is unable to find the module

Now, here comes the task to handle these errors within our code in Python. So here we need **try-except** statements.

```
Basic Syntax :
 try:
    // Code
 except:
    // Code
```

How try() works?

- First **try** clause is executed i.e. the code between **try** and **except** clause.
- If there is no exception, then only **try** clause will run, **except** clause is finished.
- If any exception occured, **try** clause will be skipped and **except** clause will run.
- If any exception occurs, but the **except** clause within the code doesn't handle it, it is passed on to the outer **try** statements. If the exception left unhandled, then the execution stops.
- A **try** statement can have more than one **except** clause

Code 1 : No exception, so **try** clause will run.

```
# Python code to illustrate
# working of try()
def divide(x, y):
    try:
        # Floor Division : Gives only Fractional Part as Answer
        result = x // y
        print("Yeah ! Your answer is :", result)
    except ZeroDivisionError:
        print("Sorry ! You are dividing by zero ")


# Look at parameters and note the working of Program
divide(3, 2)
```

Output :

```
('Yeah ! Your answer is :', 1)
```

Code 1 : There is an exception so only **except** clause will run.

```
# Python code to illustrate
# working of try()
def divide(x, y):
    try:
        # Floor Division : Gives only Fractional Part as Answer
        result = x // y
        print("Yeah ! Your answer is :", result)
    except ZeroDivisionError:
        print("Sorry ! You are dividing by zero ")


# Look at parameters and note the working of Program
divide(3, 0)
```

Output :

```
Sorry ! You are dividing by zero
```

Errors and Exceptions in Python

Errors are the problems in a program due to which the program will stop the execution. On the other hand, exceptions are raised when the some internal events occur which changes the normal flow of the program.

Two types of Error occurs in python.

1. Syntax errors
2. Logical errors (Exceptions)

Syntax errors

When the proper syntax of the language is not followed then syntax error is thrown.

Example

```
# initialize the amount variable
amount = 10000


# check that You are eligible to
#  purchase Dsa Self Paced or not
if(amount>2999)
```

```
    print("You are eligible to purchase Dsa Self Paced")
```

Output:

```
  File "/home/ac35380186f4ca7978956ff46697139b.py", line 4
    if(amount>2999)
                  ^
SyntaxError: invalid syntax
```

It returns a syntax error message because after if statement a colon `:` is missing. We can fix this by writing the correct syntax.

logical errors(Exception)

When in the runtime an error occurs after passing the syntax test is called exception or logical type. For example, when we divide any number by zero then `ZeroDivisionError` exception is raised, or when we import a module that does not exist then `ImportError` is raised.

Example 1:

```
# initialize the amount variable

marks = 10000


# perform division with 0

a = marks / 0

print(a)
```

Output:

```
Traceback (most recent call last):
  File "/home/f3ad05420ab851d4bd106ffb04229907.py", line 4, in <module>
    a=marks/0
ZeroDivisionError: division by zero
```

In the above example the ZeroDivisionError as we are trying to divide a number by 0.

Example 2: When indentation is not correct.

```
if(a<3):

print("gfg")
```

Output:

```
  File "/home/959e778cc1b15563df98d2e1e26f92e6.py", line 2
    print("gfg")
        ^
IndentationError: expected an indented block
```

Some of the common built-in exceptions are other than above mention exceptions are:

EXCEPTION	DESCRIPTION
IndexError	When the wrong index of a list is retrieved.
AssertionError	It occurs when assert statement fails
AttributeError	It occurs when an attribute assignment is failed.
ImportError	It occurs when an imported module is not found.
KeyError	It occurs when the key of the dictionary is not found.
NameError	It occurs when the variable is not defined.
MemoryError	It occurs when a program run out of memory.
TypeError	It occurs when a function and operation is applied in an incorrect type.

Note: For more information, refer to Built-in Exceptions in Python

Error Handling

When an error and an exception is raised then we handle that with the help of Handling method.

- **Handling Exceptions with Try/Except/Finally**
  We can handle error by Try/Except/Finally method.we write unsafe code in the try, fall back code in except and final code in finally block.
  Example

```
# put unsafe operation in try block
try:
     print("code start")

     # unsafe operation perform
     print(1 / 0)

# if error occur the it goes in except block
except:
     print("an error occurs")

# final code in finally block
finally:
     print("GeeksForGeeks")
```

Output:

```
code start

an error occurs

GeeksForGeeks
```

- **Raising exceptions for a predefined condition**
  When we want to code for limitation of certain condition then we can raise an exception.
  Example

```
# try for unsafe code
try:
    amount = 1999
    if amount < 2999:

        # raise the ValueError
        raise ValueError("please add money in your account")
    else:
        print("You are eligible to purchase DSA Self Paced course")

# if false then raise the value error
except ValueError as e:
        print(e)
```

Output:

```
please add money in your account
```

Built-in Exceptions in Python

All instances in Python must be instances of a class that derives from **BaseException**. Two exception classes that are not related via subclassing are never equivalent, even if they have the same name. The built-in exceptions can be generated by the interpreter or built-in functions.
There are several built-in exceptions in Python that are raised when errors occur. These built-in exceptions can be viewed using the **local()** built-in functions as follows :

```
>>> locals()['__builtins__']
```

This returns a dictionary of built-in exceptions, functions and attributes.

Base Classes

The following exceptions are used mostly as base classes for other exceptions.

1. **exception BaseException**
   This is the base class for all built-in exceptions. It is not meant to be directly inherited by user-defined classes. For, user-defined classes, Exception is used. This class is responsible for creating a string representation of the exception using str() using the arguments passed. An empty string is returned if there are no arguments.
   - **args :** The args are the tuple of arguments given to the exception constructor.
   - **with_traceback(tb) :** This method is usually used in exception handling. This method sets tb as the new traceback for the exception and returns the exception object.
     Code :
     ```
     try:

         ...

     except SomeException:

         tb = sys.exc_info()[2]

         raise OtherException(...).with_traceback(tb)
     ```
2. **exception Exception**
   This is the base class for all built-in non-system-exiting exceptions. All user-defined exceptions should also be derived from this class.
3. **exception ArithmeticError**
   This class is the base class for those built-in exceptions that are raised for various arithmetic errors such as :
   - OverflowError

- ZeroDivisionError
- FloatingPointError

Example :

```
try:
    a = 10/0
    print a
except ArithmeticError:
        print "This statement is raising an arithmetic exception."
else:
    print "Success."
```

Output :

```
This statement is raising an arithmetic exception.
```

4. **exception BufferError**
   This exception is raised when buffer related operations cannot be performed.
5. **exception LookupError**
   This is the base class for those exceptions that are raised when a key or index used on a mapping or sequence is invalid or not found. The exceptions raised are :
   - KeyError
   - IndexError

   Example :

```
try:
    a = [1, 2, 3]
    print a[3]
except LookupError:
    print "Index out of bound error."
else:
    print "Success"
```

Output :

```
Index out of bound error.
```

Concrete exceptions

The following exceptions are the exceptions that are usually raised.

1. **exception AssertionError**
   An AssertionError is raised when an assert statement fails.
   Example :

```
assert False, 'The assertion failed'
```

Output :

```
Traceback (most recent call last):

  File "exceptions_AssertionError.py", line 12, in

    assert False, 'The assertion failed'

AssertionError: The assertion failed
```

2. **exception AttributeError**
   An AttributeError is raised when an attribute reference or assignment fails such as when a non-existent attribute is referenced.
   Example :

```
class Attributes(object):

    pass



object = Attributes()

print object.attribute
```

Output :

```
Traceback (most recent call last):

  File "d912bae549a2b42953bc62da114ae7a7.py", line 5, in

    print object.attribute

AttributeError: 'Attributes' object has no attribute 'attribute'
```

3. **exception EOFError**
   An EOFError is raised when built-in functions like input() hits an end-of-file condition (EOF) without reading any data. The file methods like readline() return an empty string when they hit EOF.
   Example :

```
while True:

    data = raw_input('Enter name : ')

    print 'Hello  ', data
```

Output :

```
Enter Name :Hello Aditi

Enter Name :Traceback (most recent call last):

  File "exceptions_EOFError.py", line 13, in

    data = raw_input('Enter name :')

EOFError: EOF when reading a line
```

4. **exception FloatingPointError**
   A FloatingPointError is raised when a floating point operation fails. This exception is always defined, but can only be raised when Python is configured with the–with-fpectl option, or the WANT_SIGFPE_HANDLER symbol is defined in the pyconfig.h file.
   Example :

```
import math


print math.exp(1000)
```

Output :

```
Traceback (most recent call last):

  File "", line 1, in

FloatingPointError: in math_1
```

5. **exception GeneratorExit**
   This exception directly inherits from BaseException instead of Exception since it is technically not an error. A GeneratorExit exception is raised when a generator or coroutine is closed.
   Example :

```
def my_generator():

    try:

        for i  in range(5):

            print 'Yielding', i

            yield i

    except GeneratorExit:

        print 'Exiting early'


g = my_generator()

print g.next()

g.close()
```

Output :

```
Yielding 0

0

Exiting early
```

6. **exception ImportError**
   An ImportError is raised when the import statement is unable to load a module or when the “from list” in from … import has a name that cannot be found.
   Example :

```
import module_does_not_exist
```

Output :

```
Traceback (most recent call last):

  File "exceptions_ImportError_nomodule.py", line 12, in
```

```
    import module_does_not_exist

ImportError: No module named module_does_not_exist
```

Example :

```
from exceptions import Userexception
```

Output :

```
Traceback (most recent call last):

  File "exceptions_ImportError_missingname.py", line 12, in

    from exceptions import Userexception

ImportError: cannot import name Userexception
```

7. **exception ModuleNotFoundError**
   This is the subclass of ImportError which is raised by import when a module could not be found. It is also raised when None is found in sys.modules.
8. **exception IndexError**
   An IndexError is raised when a sequence is referenced which is out of range.
   Example :

```
array = [ 0, 1, 2 ]

print array[3]
```

Output :

```
Traceback (most recent call last):

  File "exceptions_IndexError.py", line 13, in

    print array[3]

IndexError: list index out of range
```

9. **exception KeyError**
   A KeyError is raised when a mapping key is not found in the set of existing keys.
   Example :

```
array = { 'a':1, 'b':2 }

print array['c']
```

Output :

```
Traceback (most recent call last):

  File "exceptions_KeyError.py", line 13, in

    print array['c']

KeyError: 'c'
```

10. **exception KeyboardInterrupt**
    This error is raised when the user hits the interrupt key such as Control-C or Delete.

Example :

```
try:

    print 'Press Return or Ctrl-C:',

    ignored = raw_input()

except Exception, err:

    print 'Caught exception:', err

except KeyboardInterrupt, err:

    print 'Caught KeyboardInterrupt'

else:

    print 'No exception'
```

Output :

```
Press Return or Ctrl-C: ^CCaught KeyboardInterrupt
```

11. **exception MemoryError**
    This error is raised when an operation runs out of memory.
    Example :

```
def fact(a):

    factors = []

    for i in range(1, a+1):

        if a%i == 0:

            factors.append(i)

    return factors


num = 600851475143

print fact(num)
```

Output :

```
Traceback (most recent call last):

  File "4af5c316c749aff128df20714536b8f3.py", line 9, in

    print fact(num)

  File "4af5c316c749aff128df20714536b8f3.py", line 3, in fact

    for i in range(1, a+1):

MemoryError
```

12. **exception NameError**
    This error is raised when a local or global name is not found. For example, an unqualified variable name.
    Example :

```
def func():
    print ans


func()
```

Output :

```
Traceback (most recent call last):
  File "cfba0a5196b05397e0a23b1b5b8c7e19.py", line 4, in
    func()
  File "cfba0a5196b05397e0a23b1b5b8c7e19.py", line 2, in func
    print ans
NameError: global name 'ans' is not defined
```

13. **exception NotImplementedError**
This exception is derived from RuntimeError. Abstract methods in user defined classed should raise this exception when the derived classes override the method.
Example :

```
class BaseClass(object):
    """Defines the interface"""
    def __init__(self):
        super(BaseClass, self).__init__()
    def do_something(self):
        """The interface, not implemented"""
        raise NotImplementedError(self.__class__.__name__ + '.do_something')


class SubClass(BaseClass):
    """Implementes the interface"""
    def do_something(self):
        """really does something"""
        print self.__class__.__name__ + ' doing something!'


SubClass().do_something()
BaseClass().do_something()
```

Output :

```
Traceback (most recent call last):
  File "b32fc445850cbc23cd2f081ba1c1d60b.py", line 16, in
    BaseClass().do_something()
  File "b32fc445850cbc23cd2f081ba1c1d60b.py", line 7, in
do_something

    raise NotImplementedError(self.__class__.__name__ +
'.do_something')

NotImplementedError: BaseClass.do_something
```

14. **exception OSError([arg])**
    The OSError exception is raised when a system function returns a system-related error, including I/O failures such as “file not found” or “disk full” errors.
    Example :

```
def func():
    print ans


func()
```

Output :

```
Traceback (most recent call last):

  File "442eccd7535a2704adbe372cb731fc0f.py", line 4, in

    print i, os.ttyname(i)

OSError: [Errno 25] Inappropriate ioctl for device
```

15. **exception OverflowError**
    The OverflowError is raised when the result of an arithmetic operation is out of range. Integers raise MemoryError instead of OverflowError. OverflowError is sometimes raised for integers that are outside a required range. Floating point operations are not checked because of the lack of standardization of floating point exception handling in C.
    Example :

```
import sys


print 'Regular integer: (maxint=%s)' % sys.maxint

try:

    i = sys.maxint * 3

    print 'No overflow for ', type(i), 'i =', i

except OverflowError, err:

    print 'Overflowed at ', i, err


print

print 'Long integer:'

for i in range(0, 100, 10):
    print '%2d' % i, 2L ** i


print
print 'Floating point values:'
try:
    f = 2.0**i
    for i in range(100):
        print i, f
        f = f ** 2
except OverflowError, err:
    print 'Overflowed after ', f, err
```

Output :

```
Regular integer: (maxint=9223372036854775807)
No overflow for   i = 2767011611056432742l

Long integer:
 0 1
10 1024
20 1048576
30 1073741824
40 1099511627776
50 1125899906842624
60 1152921504606846976
70 1180591620717411303424
80 1208925819614629174706176
90 1237940039285380274899124224

Floating point values:
0 1.23794003929e+27
1 1.53249554087e+54
2 2.34854258277e+108
3 5.5156522631e+216
Overflowed after  5.5156522631e+216 (34, 'Numerical result out of
range')
```

16. **exception RecursionError**
    The RecursionError is derived from the RuntimeError. This exception is raised when the interpreter detects that the maximum recursion depth is exceeded.
17. **exception ReferenceError**
    The ReferenceError is raised when a weak reference proxy is used to access an attribute of the referent after the garbage collection.

Example :

```
import gc

import weakref


class Foo(object):


    def __init__(self, name):

        self.name = name


    def __del__(self):

        print '(Deleting %s)' % self


obj = Foo('obj')

p = weakref.proxy(obj)


print 'BEFORE:', p.name

obj = None

print 'AFTER:', p.name
```

Output :

```
BEFORE: obj

(Deleting )

AFTER:


Traceback (most recent call last):

  File "49d0c29d8fe607b862c02f4e1cb6c756.py", line 17, in

    print 'AFTER:', p.name

ReferenceError: weakly-referenced object no longer exists
```

18. **exception RuntimeError**
    The RuntimeError is raised when no other exception applies. It returns a string indicating what precisely went wrong.
19. **exception StopIteration**
    The StopIteration error is raised by built-in function next() and an iterator‘s __next__() method to signal that all items are produced by the iterator.
    Example :

```
Arr = [3, 1, 2]

i=iter(Arr)



print i

print i.next()

print i.next()

print i.next()

print i.next()
```

Output :

```

3

1

2


Traceback (most recent call last):

  File "2136fa9a620e14f8436bb60d5395cc5b.py", line 8, in

    print i.next()

StopIteration
```

20. **exception SyntaxError**
    The SyntaxError is raised when the parser encounters a syntax error. A syntax error may occur in an import statement or while calling the built-in functions exec() or eval(), or when reading the initial script or standard input.
    Example :

```
try:

    print eval('geeks for geeks')

except SyntaxError, err:

    print 'Syntax error %s (%s-%s): %s' % \

        (err.filename, err.lineno, err.offset, err.text)

    print err
```

Output :

```
Syntax error  (1-9): geeks for geeks
```

```
invalid syntax (, line 1)
```

21. **exception SystemError**
    The SystemError is raised when the interpreter finds an internal error. The associated value is a string indicating what went wrong.
22. **exception SystemExit**
    The SystemExit is raised when sys.exit() function is called. A call to sys.exit() is translated into an exception to execute clean-up handlers (finally clauses of try statements) and to debug a script without running the risk of losing control.
23. **exception TypeError**
    TypeError is raised when an operation or function is applied to an object of inappropriate type. This exception returns a string giving details about the type mismatch.
    Example :

```
arr = ('tuple', ) + 'string'

print arr
```

Output :

```
Traceback (most recent call last):

  File "30238c120c0868eba7e13a06c0b1b1e4.py", line 1, in

    arr = ('tuple', ) + 'string'

TypeError: can only concatenate tuple (not "str") to tuple
```

24. **exception UnboundLocalError**
    UnboundLocalError is a subclass of NameError which is raised when a reference is made to a local variable in a function or method, but no value has been assigned to that variable.
    Example :

```
def global_name_error():

    print unknown_global_name



def unbound_local():

    local_val = local_val + 1

    print local_val



try:

    global_name_error()

except NameError, err:

    print 'Global name error:', err
```

```
try:
    unbound_local()
except UnboundLocalError, err:
    print 'Local name error:', err
```

Output :

```
Global name error: global name 'unknown_global_name' is not defined
Local name error: local variable 'local_val' referenced before
assignment
```

25. **exception UnicodeError**
    This exception is a subclass of ValueError. UnicodeError is raised when a Unicode-related encoding or decoding error occurs.
26. **exception ValueError**
    A ValueError is raised when a built-in operation or function receives an argument that has the right type but an invalid value.
    Example :

    ```
    print int('a')
    ```

    Output :

    ```
    Traceback (most recent call last):
      File "44f00efda935715a3c5468d899080381.py", line 1, in
        print int('a')
    ValueError: invalid literal for int() with base 10: 'a'
    ```
27. **exception ZeroDivisionError**
    A ZeroDivisionError is raised when the second argument of a division or modulo operation is zero. This exception returns a string indicating the type of the operands and the operation.
    Example :

    ```
    print 1/0
    ```

    Output :

    ```
    Traceback (most recent call last):
      File "c31d9626b41e53d170a78eac7d98cb85.py", line 1, in
        print 1/0
    ZeroDivisionError: integer division or modulo by zero
    ```

User-defined Exceptions in Python with Examples

Prerequisite- This article is an extension to Exception Handling.
Python throws errors and exceptions, when there is a code gone wrong, which may cause program to stop abruptly. Python also provides exception handling method with the help of try-except. Some of the standard exceptions which are most frequent include IndexError, ImportError, IOError, ZeroDivisionError, TypeError and FileNotFoundError. A user can create his own error using exception class.

Creating User-defined Exception

Programmers may name their own exceptions by creating a new exception class. Exceptions need to be derived from the Exception class, either directly or indirectly. Although not mandatory, most of the exceptions are named as names that end in **"Error"** similar to naming of the standard exceptions in python. For example:

```
# A python program to create user-defined exception


# class MyError is derived from super class Exception
class MyError(Exception):


    # Constructor or Initializer
    def __init__(self, value):
        self.value = value


    # __str__ is to print() the value
    def __str__(self):
        return(repr(self.value))


try:
    raise(MyError(3*2))


# Value of Exception is stored in error
except MyError as error:
    print('A New Exception occured: ',error.value)
```

Output:

```
('A New Exception occured: ', 6)
```

Knowing all about Exception Class

To know more about about class Exception, run the code below

```
help(Exception)
```

Deriving Error from Super Class Exception

Super class Exceptions are created when a module needs to handle several distinct errors. One of the common way of doing this is to create a base class for exceptions defined by that module. Further, various subclasses are defined to create specific exception classes for different error conditions.

```
# class Error is derived from super class Exception
class Error(Exception):


    # Error is derived class for Exception, but
    # Base class for exceptions in this module
    pass


class TransitionError(Error):


    # Raised when an operation attempts a state
    # transition that's not allowed.
    def __init__(self, prev, nex, msg):
        self.prev = prev
        self.next = nex


        # Error message thrown is saved in msg
        self.msg = msg
try:
    raise(TransitionError(2,3*2,"Not Allowed"))


# Value of Exception is stored in error
except TransitionError as error:
    print('Exception occured: ',error.msg)
```

Output:

```
('Exception occured: ', 'Not Allowed')
```

How to use standard Exceptions as base class?
Runtime error is a class is a standard exception which is raised when a generated error does not fall into any category. This program illustrates how to use runtime error as base class and network error as derived class. In a similar way, any exception can be derived from the standard exceptions of Python.

```
# NetworkError has base RuntimeError

# and not Exception

class Networkerror(RuntimeError):

    def __init__(self, arg):

        self.args = arg



try:

    raise Networkerror("Error")



except Networkerror as e:

    print (e.args)
```

Output:

```
('E', 'r', 'r', 'o', 'r')
```

NZEC error in Python

While coding in various competitive sites, many people must have have encountered NZEC error. NZEC (non zero exit code) as the name suggests occurs when your code is failed to return 0. When a code returns 0 it means it is successfully executed otherwise it will return some other number depending on the type of error.
When the program ends and it is supposed to return “0” to indicate if finished fine and not able to do so it cause NZEC. Of course there are more cases associated with NZEC.

Why does NZEC occur?(one example)

In python, generally multiple inputs are separated by commas and we read them using input() or int(input()), but most of the online coding platforms while testing gives input separated by space and in those cases int(input()) is not able to read the input properly and shows error like NZEC.

How to resolve?

For Example, Think of a simple program where you have to read 2 integer and print them(in input file both integer are in same line). Suppose you have two integers as shown below:
23 45
Instead of using :

```
n = int(input())

k = int(input())
```

Use:

```
n, k = raw_input().split(" ")

n = int(n)

k = int(k)
```

to delimit input by white spaces.

Wrong code

```
n = int(input())

k = int(input())

print n," ",k
```

Input:
2 3
When you run the above code in IDE with above input you will get error:-

```
Traceback (most recent call last):

  File "b712edd81d4a972de2a9189fac8a83ed.py", line 1, in

    n = int(input())

  File "", line 1

    2 3

      ^

SyntaxError: unexpected EOF while parsing
```

The above code will work fine when the input are in 2 two different lines. You can test yourself. To overcome this problem you need to use split.

Correct code

```
n, k = raw_input().split(" ")

n = int(n)

k = int(k)

print n," ",k
```

Input:

```
7 3
```

Output:

```
7   3
```

Some prominent reasons for NZEC error

1. Infinite Recursion or if you have run out of stack memory.
2. Input and output both are NOT exactly same as the test cases.
3. As the online platforms, test your program using a computer code which matches your output with the specified outputs exactly.
4. This type of error is also shown when your program is performing basic programming mistakes like dividing by 0.
5. Check for the values of your variables, they can be vulnerable to integer flow.

There could be some other reasons also for occurrence NZEC error, i have listed the frequent ones.