Python Date and Time:
In the real world applications, there are scenarios where we need to work with the **date** and **time**. In python, the **date** is **not** a **data type**, but we can work with the **date objects** by **importing** the **module** named with **datetime**, **time**, and **calendar**.

Tick:
In python, the **time instants** are **counted** since **12 AM, 1st January 1970**. The function **time ( )** of the **module time** returns the **total number of ticks** spent since 12 AM, 1st January 1970. A **tick** can be seen as the **smallest unit** to **measure** the **time**.

```
In [1]: import time          # Prints the number of ticks spent since 12 AM, 1st January 1970
        print(time.time())

        1589001786.975549
```

Python has a **module** named **datetime** to work with **dates** and **times**.

1) Get Current Date and Time:

```
In [4]: import datetime
        datetime_object = datetime.datetime.now()
        print(datetime_object)

        2020-05-09 10:58:57.813786
```

2) Get Current Date:

```
In [6]: import datetime
        dateobject = datetime.date.today()
        print(dateobject)

        2020-05-09
```

```
In [18]: from datetime import date
         dt = date.today()
         print("Current date =", dt)

         Current date = 2020-05-09
```

Note: In this program, we have used **today ( )** method defined in the **date class** to get a **date object** containing the **current local date**.

What's inside datetime?
We can use **dir ( )** function to **get a list containing all attributes** of a **module**.

```
In [8]: import datetime
        print(dir(datetime))

        ['MAXYEAR', 'MINYEAR', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec_
        _', 'date', 'datetime', 'datetime_CAPI', 'sys', 'time', 'timedelta', 'timezone', 'tzinfo']
```

1) https://docs.python.org/3/library/ **==> Standard Library**
2) https://pypi.org/ **==> Other Librarys**

Commonly used classes in the datetime module are:
date, **time** , **datetime** & **timedelta** Class

datetime.date Class:
You can **instantiate date objects** from the **date class**. A **date object** represents a **date (year, month and day).**

3) Date object to represent a date:

```
In [15]: import datetime
         dt = datetime.date(2020, 5, 9)
         print(dt)

         2020-05-09
```

If you are wondering, **date ( )** in the above example is a **constructor** of the **date class**. The **constructor** takes **three arguments**: **year, month and day**.

We can only **import date class** from the **datetime module**. Here's how:

```
In [16]: from datetime import date
         dt = date(2020, 5, 9)
         print(dt)

         2020-05-09
```

4) Get date from a timestamp:
We can also create **date objects** from a **timestamp**. **Unix timestamp** is the **number of seconds** between a **particular date** and **January 1, 1970** at UTC. You can **convert** a **timestamp** to **date** using **fromtimestamp ( )** method.

UTC: Universal Time Coordinated
Prior to 1972, this time was called **Greenwich Mean Time (GMT)** but is now referred to as **Coordinated Universal Time** or **Universal Time Coordinated** (UTC).

print(time.time())
1624249537.59069

```
In [20]: from datetime import date
         tmstmp = date.fromtimestamp(1326244364)
         print("Date =", tmstmp)

         Date = 2012-01-11
```

5) Print today's year, month and day:

We can get **year**, **month**, **day**, **day of the week** etc. from the **date object** easily.

```
In [22]: from datetime import date

         today = date.today()        # date object of today's date

         print("Current Year:", today.year)
         print("Current Month:", today.month)
         print("Current Day:", today.day)
```

```
Current Year: 2020
Current Month: 5
Current Day: 9
```

datetime.time:

A **time object instantiated** from the **time class represents** the **local time**.

6) Time object to represent time:

```
In [24]: from datetime import time
         a = time()              # time(hour = 0, minute = 0, second = 0)
         print("a =", a)

         b = time(11, 34, 56)  # time(hour, minute and second)
         print("b =", b)

         c = time(hour = 11, minute = 34, second = 56)  # time(hour, minute and second)
         print("c =", c)

         d = time(11, 34, 56, 234566)  # time(hour, minute, second, microsecond)
         print("d =", d)
```

```
a = 00:00:00
b = 11:34:56
c = 11:34:56
d = 11:34:56.234566
```

7) Print hour, minute, second and microsecond:

Once you create a **time object**, you can easily **print** its **attributes** such as **hour**, **minute**, **second, microsecond** etc.

```
In [26]: from datetime import time

         a = time(11, 34, 56)

         print("hour =", a.hour)
         print("minute =", a.minute)
         print("second =", a.second)
         print("microsecond =", a.microsecond)
```

```
hour = 11
minute = 34
second = 56
microsecond = 0
```

Note: Notice that we **haven't** passed **microsecond** argument. Hence, its **default** value **0** is printed.

datetime.datetime:

The **datetime module** has a **class** named **dateclass** that can contain information from both **date** and **time** objects.

8) Python datetime object:

```
In [28]: from datetime import datetime

         a = datetime(2018, 11, 28)  #datetime(year, month, day)
         print(a)

         b = datetime(2017, 11, 28, 23, 55, 59, 342380) # datetime(year, month, day, hour, minute, second, microsecond)
         print(b)
```

```
2018-11-28 00:00:00
2017-11-28 23:55:59.342380
```

Note: The **first three** arguments **year**, **month** and **day** in the **datetime ( )** constructor are mandatory.

9) Print year, month, hour, minute and timestamp:

```
In [30]: from datetime import datetime

         a = datetime(2017, 11, 28, 23, 55, 59, 342380)
         print("year =", a.year)
         print("month =", a.month)
         print("hour =", a.hour)
         print("minute =", a.minute)
         print("timestamp =", a.timestamp())
```

```
year = 2017
month = 11
hour = 23
minute = 55
timestamp = 1511893559.34238
```

10) Difference between two dates and times:

```
In [32]: from datetime import datetime, date

         t1 = date(year = 2018, month = 7, day = 12)
         t2 = date(year = 2017, month = 12, day = 23)
         t3 = t1 - t2
         print("t3 =", t3)

         t4 = datetime(year = 2018, month = 7, day = 12, hour = 7, minute = 9, second = 33)
         t5 = datetime(year = 2019, month = 6, day = 10, hour = 5, minute = 55, second = 13)
         t6 = t4 - t5
         print("t6 =", t6)

         print("type of t3 =", type(t3))
         print("type of t6 =", type(t6))

         t3 = 201 days, 0:00:00
         t6 = -333 days, 1:14:20
         type of t3 = <class 'datetime.timedelta'>
         type of t6 = <class 'datetime.timedelta'>
```

11) Difference between date & timedelta objects:

```
1  from datetime import date, timedelta
2  t1 = date(2021,6,21)
3  t2 = timedelta(15)
4  print(t1 - t2)
```

input

```
2021-06-06
```

Python format datetime

The way **date** and **time** is represented may be **different** in **different places**, **organizations** etc. It's more common to use **mm/dd/yyyy** in the **US**, whereas **dd/mm/yyyy** is more common in the **UK**.

12) Python strftime ( ) datetime object to string:

The **strftime ( ) method** is defined under **classes date**, **datetime** and **time**. The method **creates** a **formatted string** from a **given date**, **datetime** or **time** object.

```
1  from datetime import datetime
2  curr_time = datetime.now()
3  print(curr_time)
4  format_time1 = curr_time.strftime("%Y-%m-%d")
5  format_time2 = curr_time.strftime("%H:%M:%S")
6  print(format_time1)
7  print(format_time2)
8  print(format_time1, format_time2)
9  format_time3 = curr_time.strftime("%d-%m-%Y")
10 print("Indian Format Date:",format_time3)
```

input

```
2021-06-21 05:57:41.526460
2021-06-21
05:57:41
2021-06-21 05:57:41
Indian Format Date: 21-06-2021
```

Here, **%Y**, **%m**, **%d**, **%H**, **%M, %S** etc. are **format codes**. The **strftime ( )** method takes **one** or **more** format codes and **returns** a **formatted string** based on it.

Current date in different formats:

```
In [56]: from datetime import date
today = date.today()

d1 = today.strftime("%d/%m/%Y")   # dd/mm/YY
print("d1 =", d1)

d2 = today.strftime("%B %d, %Y")  # Textual month, day and year
print("d2 =", d2)

d3 = today.strftime("%m/%d/%y")   # mm/dd/y
print("d3 =", d3)

d4 = today.strftime("%b-%d-%Y")   # Month abbreviation, day and year
print("d4 =", d4)
```

```
d1 = 09/05/2020
d2 = May 09, 2020
d3 = 05/09/20
d4 = May-09-2020
```

13) Python strptime ( ) - string to datetime:

The **strptime ( )** method **creates** a **datetime object** from a given **string** (representing **date** and **time**).

```
In [47]: from datetime import datetime

date_string = "21 June, 2018"
print("date_string =", date_string)

date_object = datetime.strptime(date_string, "%d %B, %Y")
print("date_object =", date_object)
```

```
date_string = 21 June, 2018
date_object = 2018-06-21 00:00:00
```

The **strptime ( )** method takes **two** arguments:

- a string representing date and time
- format code equivalent to the first argument

By the way, **%d**, **%B** and **%Y** format codes are used for **day**, **month** (full name) and **year**.

Here,
%d - Represents the day of the month. Example: 01, 02, ..., 31
%B - Month's name in full. Example: January, February etc.
%Y - Year in four digits. Example: 2018, 2019 etc.

14) Handling timezone in Python:

Suppose, you are working on a project and need to display **date** and **time** based on their **timezone**. Rather than trying to **handle timezone** yourself, we suggest you to use a third-party **pytz module**.

```
1  from datetime import datetime
2  import pytz
3
4  IST = datetime.now()
5  print("Current India is:", IST.strftime("%Y:%m:%d %H:%M:%S"))
6
7  NY_Time = pytz.timezone("America/New_York") #Europe/London
8  NY_TZ = datetime.now(NY_Time)
9  print("Current NY Time is:", NY_TZ.strftime("%Y:%m:%d %H:%M:%S"))
10
```

input

```
Current India is: 2021:06:21 06:29:22
Current NY Time is: 2021:06:21 02:29:22
```

15) sleep time:

The **sleep ( )** method of **time module** is used to **stop** the **execution** of the **script** for a **given amount of time**. The **output** will be **delayed** for the **number of seconds** given as **float**.

```
In [51]: import time
         for i in range(0,5):
             print(i)
             time.sleep(1)    # Each element will be printed after 1 second
```

```
0
1
2
3
4
```

The calendar module:

Python provides a **calendar object** that **contains** various **methods** to work with the **calendars**.

```
In [54]: import calendar;
         cal = calendar.month(2020, 4)
         print(cal)          # Printing the calendar of April 2020
```

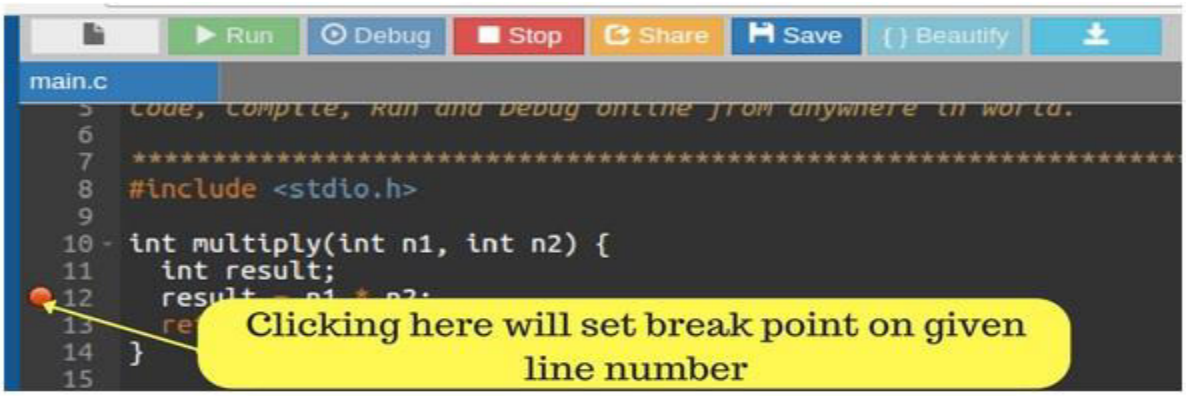
```
     April 2020
Mo Tu We Th Fr Sa Su
       1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30
```

Debugging:

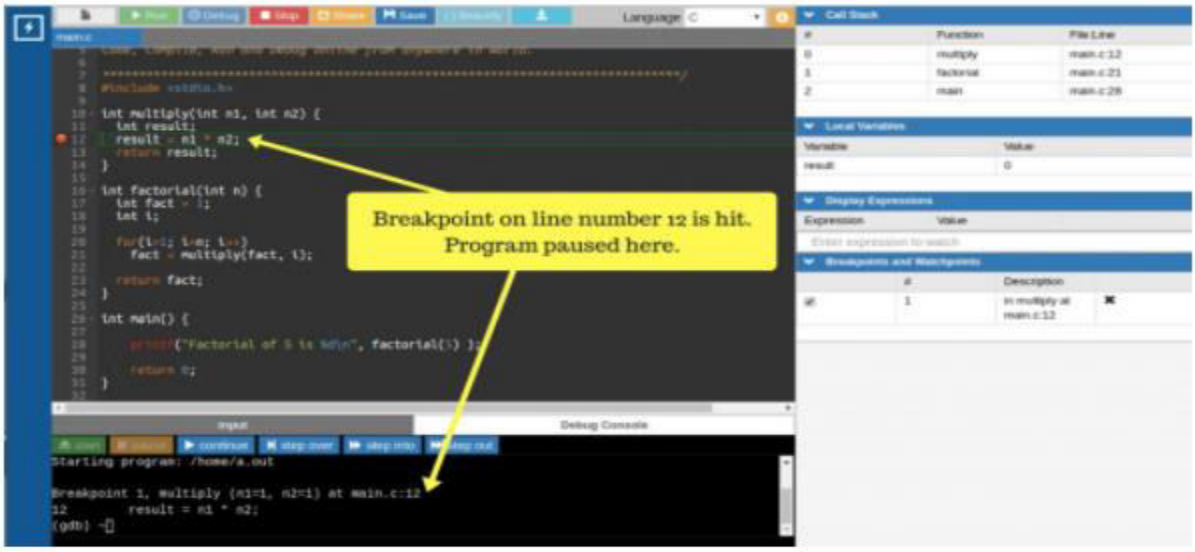
In general, **debugger** is **utility** that **runs target** program in **controlled environment** where you **can control execution** of **program** and see the **state of program** when **program** is **paused**.

How can I control execution of program?

We can tell **debugger** when to **pause** a program by setting **breakpoints**.
To set a **breakpoint**, click on **blank area** seen on **left side** of **line number** in **editor**.
When you **click** it, it should **display red circle**; which means **breakpoint** is **set** on that **line** number.



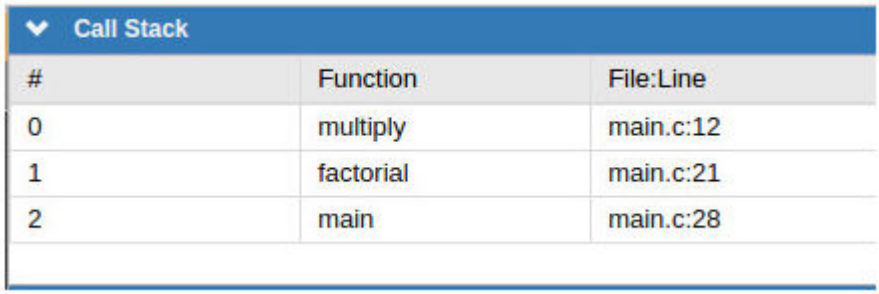
Once you **set breakpoint**, when you **start program** in **debug** mode, it will **pause execution** when **program reaches** the **line** where **breakpoint** is **set**.



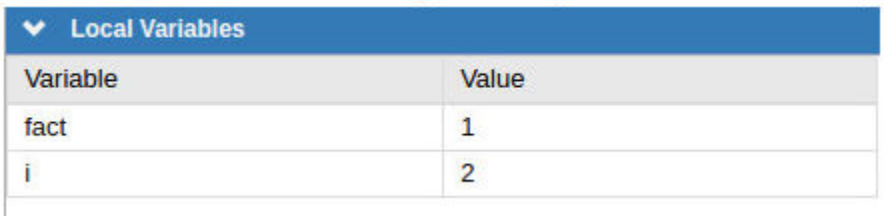
Now we can use **stepping commands** to **execute** program **line** by **line**.

1) **continue:** Resume program execution until next breakpoint is reached
2) **step into:** Execute program line by line stepping into function
3) **step over:** Execute program line by line but don't go inside function call
4) **step out:** Resume program execution until current function is finished

What can I see when program is paused?
You can see **call stack** and **values** of **local variables** of **current function.call** stack shows you **chain** of **function calls**.
As seen in below image, "**main**" function is calling function "**factorial**" and "**factorial**" is calling function "**multiply**".



Local variables window **shows** you **values** of **local variables** of **current function**. As shown in image below, value of "fact" is 1, "i" is 1.



Program: Remove Duplicates without using any function.

```
a = [5, 10, 9, ["Aniket", 25, "Saif"], 5, 5, 100]
i = 0
while i < len(a):
    if 5 == a[i]:
        a.remove(5)
        i = i - 1
    i = i + 1
print(a)
```

input

```
[10, 9, ['Aniket', 25, 'Saif'], 100]
```

Logging:

- Logging is a **means** of **tracking events** that **happen** when some **software runs**.
- Logging is important for software developing, debugging and running.
- If you **don't** have any **logging record** and your **program crashes**, there are very little **chances** that you **detect** the **cause** of the **problem**. And if you **detect** the **cause**, it will **consume** a **lot** of **time**.
- With logging, you can **leave** a **trail** of **location** so that if **something** goes **wrong**, we can **determine** the **cause** of the **problem**.
- Python has a **built-in** module **logging** which **allows** writing **status messages** to a **file** or any other **output streams**.

Levels of Log Message: There are two built-in levels of the log message.
1) Debug: These are used to give **detailed information** when **diagnosing** problems.
2) Info: These are used to **confirm** that **things** are **working** as **expected**
3) Warning: These are used as **indication** that **something unexpected happened**, or **indicative** of **some problem** in the **near future.**
4) Error: This tells that **due** to a **more serious problem**, the software has **not** been able to **perform** some function.
5) Critical: This tells **serious error**, indicating that the **program itself** may be **unable** to **continue** running.

Level	Numeric Value
NOTSET	0
DEBUG	10
INFO	20
WARNING	30
ERROR	40
CRITICAL	50

Note:
1) **Logging module** is packed with **several** features. It has several **constants**, **classes**, and **methods**.
2) The items with **all caps** are **constant**, the **capitalize** items are **classes** and the items which start with **lowercase** letters are **methods**.

There are several **logger objects** offered by the module itself.
1) Logger.info(msg): This will log a message with level INFO on this logger.
2) Logger.warning(msg): This will log a message with level WARNING on this logger.
3) Logger.error(msg): This will log a message with level ERROR on this logger.
4) Logger.critical(msg): This will log a message with level CRITICAL on this logger.
5) Logger.log(lvl, msg): This will Logs a message with integer level lvl on this logger.
6) Logger.exception(msg): This will log a message with level ERROR on this logger.
7) Logger.setLevel(lvl): This function sets the threshold of this logger to lvl. This means that all the messages below this level will be ignored.

import logging → Import logging
dir(logging) → check constants, classes & methods
****

Step 1: Creating a simple logger with info message.
****

```
import logging

#Create & Configure Logger: #Default logging level for basicConfig is set to 30
logging.basicConfig(filename="/home/saif/LFS/logs/pythonLogsTesting.log")
logger = logging.getLogger()

#Test the Logger:
logger.info("This is my info logger")

print(logger.level)
```

****

Output: File gets **created** but **no log msgs** are **written**.

```
-rw-rw-r-- 1 saif saif     0 Jun 21 12:54 my_python_log
```

Let's check the **level** of **logger**:

```
>>> print(logger.level)
30
```

info ➔ **20**
But:

```
basicConfig: Default log level is
             WARNING = 30
```

****

Step 2: Changing Log Level:

****

```
import logging

#Create & Configure Logger:
logging.basicConfig(filename="/home/saif/LFS/logs/pythonLogsTesting.log")
logger = logging.getLogger()

#Setting the threshold of logger to DEBUG:
logger.setLevel(logging.DEBUG)

#Test the Logger:
logger.info("This is my info logger")

print(logger.level)
```

****

Step 3: Add Time format to logs.

```
import logging

#Create & Configure Logger:
log_format = "%(levelname)s %(asctime)s - %(message)s"
logging.basicConfig(filename="/home/saif/LFS/logs/pythonLogsTesting.log",
                    format=log_format)
logger = logging.getLogger()

#Setting the threshold of logger to DEBUG:
logger.setLevel(logging.DEBUG)

#Test the Logger:
logger.info("This is my info logger")

print(logger.level)
```

****

Step 4: Overwriting the log data

```
import logging

#Create & Configure Logger:
log_format = "%(levelname)s %(asctime)s - %(message)s"
logging.basicConfig(filename="/home/saif/LFS/logs/pythonLogsTesting.log",
                    format=log_format,
                    filemode="w")
```

```
logger = logging.getLogger()

#Setting the threshold of logger to DEBUG:
logger.setLevel(logging.DEBUG)

#Test the Logger:
logger.info("This is my NEW INFO logger")

print(logger.level)
```

****

Step 5: Writing all Log Levels.

```
import logging              #importing module

#Create and configure logger:
logging.basicConfig(filename="/home/saif/LFS/logs/pythonLogsTesting.log",
                    format='%(asctime)s %(message)s',
                    filemode='w')

#Creating an object:
logger=logging.getLogger( )

#Setting the threshold of logger to DEBUG:
logger.setLevel(logging.DEBUG)      #ERROR

#Test messages:
logger.debug("Debug Message")
logger.info("Info Message")
logger.warning("Warning Message")
logger.error("Error Message")
logger.critical("Critical Message")

print("Logs written Successfully")
```

****

Exception Handling:

- Python has many **built-in exceptions** that **force** your **program** to **output** an **error** when **something** in the **program goes wrong**.
- When these **exceptions** occur, it **causes** the **current** process to **stop** and **passes** it to the **calling** process **until** it is **handled**. If **not handled**, our program will **crash**. **E.g.** If function A calls function B which in turn calls function C and an exception occurs in function C. If it is not handled in C, the exception passes to B and then to A.
- If **never handled**, an **error message** is **displayed** and our **program comes** to a sudden **unexpected** halt.

Let's see few errors:

a) SyntaxError:

```
for i in range(5)
  print("Python", i)
```

```
  File "main.py", line 1
    for i in range(5)
                   ^
SyntaxError: invalid syntax
```

b) ZeroDivisionError:

```
1/0
```

```
Traceback (most recent call last):
  File "main.py", line 1, in <module>
    1/0
ZeroDivisionError: division by zero
```

3) FileNotFoundError:

```
with open("file.txt") as f:
  readFile = f.read()
print(readFile)
```

```
Traceback (most recent call last):
  File "main.py", line 1, in <module>
    with open("file.txt") as f:
FileNotFoundError: [Errno 2] No such file or directory: 'file.txt'
```

4) TypeError:

```
1 + 2 + "Three"
```

```
Traceback (most recent call last):
  File "main.py", line 1, in <module>
    1 + 2 + "Three"
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

5) ValueError:

import math

print(math.sqrt(-1))

```
Traceback (most recent call last):
  File "main.py", line 2, in <module>
    print(math.sqrt(-1))
ValueError: math domain error
```

Syntax to find all Exceptions supported by Python:

print(dir(locals()['__builtins__']))

Exception Clauses:

```
try:
    # Runs first
    < code >
except:
    # Runs if exception occurs in try block
    < code >
else:
    # Executes if try block *succeeds*
    < code>
finally:
    # This code *always* executes
    < code >
```

Write a function that reads binary file & returns data & Measure the time required?

```
In [2]: import logging
        import time

        #Create logger:
        logging.basicConfig(filename="/home/saif/LFS/logs/exceptions.log")
        logger=logging.getLogger()

        #Setting the threshold of logger to DEBUG
        logger.setLevel(logging.DEBUG)
```

```
In [3]: #Function to read binary file:
        def read_file(path):
            """Return the contents of file at "path" & time taken"""
            start_time=time.time()
            try:
                f=open(path,mode="rb")
                data = f.read()
                return data
            except FileNotFoundError as err:
                logger.error(err)
            else:
                 f.close()
            finally:
                stop_time=time.time()
                dt=stop_time - start_time
                logger.info("Time required for {file} = {time}".format(file=path, time=dt))
        data=read_file("/home/saif/LFS/datasets/movies.csv") # partitions_txns
```

Output:

```
saif@smidsy-technologies:~/LFS/logs$ cat exceptions.log
INFO:root:Time required for /home/saif/LFS/datasets/movies.csv = 0.005500316619873047
```

Run by changing filename to:

movies1.csv

```
ERROR:root:[Errno 2] No such file or directory: '/home/saif/LFS/datasets/movies1.csv'
INFO:root:Time required for /home/saif/LFS/datasets/movies1.csv = 0.00017905235290527344
```

Use large file to see the read time difference:

```
INFO:root:Time required for /home/saif/LFS/datasets/partitions_txns = 5.20127534866333
```

Pip:

1) Pip is a tool for **installing Python packages**.
2) With pip, you can **search**, **download**, **install** and **upgrade** packages from **Python Package Index (PyPI)** and other **package indexes**.

When installing a Python module **globally**, it is highly **recommended** to **install** the module's **deb package** with the **apt tool** as they are **tested** to **work** properly on Ubuntu systems.

Python 3 packages are prefixed with **python3-** and **Python 2** packages are prefixed with **python2-**.

Use **pip** to **install** a module **globally** only if there is **no deb package** for that module.

Prefer using **pip** within a **virtual environment** only. Python Virtual Environments **allows** you to **install** Python modules in an **isolated** location for a **specific project**, rather than being **installed globally**. This way, you **do not** have to worry about **affecting** other Python projects.

Installing pip for Python 3:
To **install pip** for **Python 3** on **Ubuntu 20.04** run the following commands as **root** or **sudo user** in your **terminal**.
→ sudo apt update
→ sudo apt install python3-pip

The command above will also install all the dependencies required for building Python modules.

When the **installation** is **complete**, **verify** the **installation** by checking the **pip version**:
→ pip3 --version

The version number may vary, but it will look something like this:
pip 20.0.2 from /usr/lib/python3/dist-packages/pip (python 3.8)

How to Use Pip:
With pip, you can **install packages** from **PyPI, version control, local projects,** and from **distribution files**.

To view the list of all pip commands and options, type:
→ pip3 --help

You can get **more information** about a **specific command** using **pip <command> --help**.

For example, to get more information about the **install command**, type:
→ pip3 install --help

Installing Packages with Pip:
Let's say you want to **install** a **package** called **scrapy** which is used for **scraping** and **extracting data** from **websites**.

To install the **latest version** of the **package** you would **run** the following **command**:
→ pip3 install scrapy

To **check information** about **installed package**:
→ pip3 list
→ pip3 freeze | grep Scrapy **OR** pip3 freeze | cut -d = -f 1 | grep Scrapy
→ pip3 show scrapy

Uninstalling Packages with **Pip**:
→ pip3 uninstall Scrapy

To find **outdated** packages:
→ pip3 list --outdated
→ pip3 list -o

To install a **specific version** of the package **append ==** and the **version number** after the **package name**:
→ pip3 install scrapy==1.5
→ pip3 freeze | grep Scrapy
→ pip3 show Scrapy
→ pip3 list --outdated | grep Scrapy

Upgrade a **Package** with **Pip**:
→ pip3 install --upgrade Scrapy **OR** pip3 install -U Scrapy
→ pip3 list --outdated | grep Scrapy
→ pip3 show Scrapy

Installing Packages with **Pip** using the **Requirements Files:**
requirement.txt is a **text file** that **contains** a **list of pip packages** with their **versions** that are required to **run** a specific **Python project**.

Use the **following command** to **install** a **list of requirements** specified in a **file**:
→ pip3 install -r requirements.txt

Reading and Writing CSV Files in Python:

1) **CSV (Comma Separated Values)** format is the most common **import** and **export** format for **spreadsheets** and **databases**.
2) It is one of the most **common** methods for **exchanging data** between **applications** and popular **data format** used in **Data Science**.
3) It is **supported** by a **wide range** of applications. A **CSV** file stores **tabular** data in which each **data field** is separated by a ***delimiter*** (comma in most cases).
4) To represent a **CSV** file, it must be saved with the ***.csv* file extension**.

Reading CSV with 2 methods i.e. with & without csv:

Without CSV:

a)

```
In [2]: path = "/home/saif/LFS/datasets/Stock_Data.csv"
        lines = [line for line in open(path)]

In [3]: lines[0]
Out[3]: 'Date,Open,High,Low,Close,Volume,Adj Close\n'

In [4]: lines[1]
Out[4]: '8/19/2014,585.002622,587.342658,584.002627,586.862643,978600,586.862643\n'
```

Remove trailing spaces & dividing the string into smaller pieces:

```
In [5]: lines[0].strip()
Out[5]: 'Date,Open,High,Low,Close,Volume,Adj Close'

In [10]: lines[0].strip().split(",")
Out[10]: ['Date', 'Open', 'High', 'Low', 'Close', 'Volume', 'Adj Close']
```

Applying above logic in one go:

```
In [13]: data = [line.strip().split(',') for line in open(path)]
         print(data[0], '\n', data[1])

         ['Date', 'Open', 'High', 'Low', 'Close', 'Volume', 'Adj Close']
          ['8/19/2014', '585.002622', '587.342658', '584.002627', '586.862643', '978600', '586.8626
         43']
```

OR

b)

```
f = open("/home/saif/LFS/datasets/guido.txt")
text = f.read( )
f.close( )
print(text)
```

Note:

- What if something goes wrong before you close the file?
- You do not want to increase you computers memory with open files?

With CSV:

So, we have a better method to open the files using **with csv** option:

1) Python contains a module called **csv** for the handling of **CSV** files.

2) The **reader class** from the **module** is **used** for **reading data** from a **CSV** file.

3) 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.

Syntax:

csv.reader (csvfile, dialect='excel', **fmtparams)

Note: The **'with'** keyword is used along with the **open ( )** method as it **simplifies exception handling** and **automatically closes** the **CSV** file.

```
import csv                          → Import CSV Module
# opening the CSV file:
with open('/home/saif/LFS/datasets/guido.txt', 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)
```

Read Data:

```
>>> with open("/home/saif/LFS/datasets/guido.txt") as fobj:
...     data = fobj.read()
...
>>> print(data)
```

What if we open a file that does not exist?

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: '/home/saif/LFS/datasets/saif.txt'
```

Error Handling:

```
>>> try:
...       with open("/home/saif/LFS/datasets/saif.txt") as fobj:
...          data = fobj.read()
... except FileNotFoundError:
...       data = None
...
>>> print(data)
None
```

Writing to CSV file:

1) **csv.writer** class is used to **insert data** to the **CSV** file.
2) This class **returns** a **writer object** which is responsible for **converting** the **user's data** into a **delimited string**.
3) A **CSV file object** should be **opened** with **newline=' '** else **newline characters inside** the **quoted fields** will **not be interpreted correctly**.

Syntax:

csv.writer (csvfile, dialect='excel', **fmtparams)

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

1) writerow( ): This method writes a single row at a time. Field row can be written using this method.

2) writerows( ): This method is used to write multiple rows at a time. This can be used to write rows list.

 # Python program to demonstrate writing to CSV:

```
In [10]: import csv

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

         # data rows of csv file:
         rows = [["Saif", 'COE', '2', '9.0'],
                 ["Ram", 'COE', '2', '9.1'],
                 ["Aniket", 'IT', '2', '9.3'],
                 ["Sagar", 'SDL', '1', '9.5'],
                 ["Swaroop", 'SDL', '3', '7.8']]

         # name of csv file:
         filename = "/home/saif/LFS/datasets/student_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)
```

Write Files:

```
>>> names = ["Saif", "Ram", "Aniket", "Mitali", "Tausif"]
>>> with open("/home/saif/LFS/datasets/names.txt", "w") as f:
...     for name in names:
...         f.write(name)
...
```

Add a new line:

```
>>> names = ["Saif", "Ram", "Aniket", "Mitali", "Tausif"]
>>> with open("/home/saif/LFS/datasets/names.txt", "w") as f:
...     for name in names:
...         f.write(name)
...         f.write("\n")
...
```

OR

```
>>> names = ["Saif", "Ram", "Aniket", "Mitali", "Tausif"]
>>> with open("/home/saif/LFS/datasets/names.txt", "w") as f:
...     for name in names:
...         print(name, file=f)
...
```

Append Data:

```
>>> with open("/home/saif/LFS/datasets/names.txt", "a") as f:
...     print(23*"=", file=f)
...     print("Appending Names", file=f)
...
```

Stock Market Example:

Step 1:

```
In [22]: import csv

path = "/home/saif/LFS/datasets/Stock_Data.csv"
file = open(path, newline='')
reader = csv.reader(file)

header = next(reader)
data = [row for row in reader]

print(header)
print(data[0])
```

```
['Date', 'Open', 'High', 'Low', 'Close', 'Volume', 'Adj Close']
['8/19/2014', '585.002622', '587.342658', '584.002627', '586.862643', '978600', '586.86264
3']
```

OR

```
with open("/home/saif/LFS/datasets/Stock_Data.csv", newline='') as f:
    reader = csv.reader(f)
    header = next(reader)
    data = [row for row in reader]

print(header)
print(data[0])
```

```
['Date', 'Open', 'High', 'Low', 'Close', 'Volume', 'Adj Close']
['8/19/2014', '585.002622', '587.342658', '584.002627', '586.862643', '978600', '586.86264
3']
```

Step 2: Formatting the data as per their data type:

In [30]:
```
import csv
from datetime import datetime

path = "/home/saif/LFS/datasets/Stock_Data.csv"
file = open(path, newline='')
reader = csv.reader(file)

header = next(reader)
# data = [row for row in reader]

# print(header)
# print(data[0])

data = []
for row in reader:
    date = datetime.strptime(row[0], "%m/%d/%Y")
    open_price = float(row[1])
    high = float(row[2])
    low = float(row[3])
    close = float(row[4])
    volume = int(row[5])
    adj_close = float(row[6])

    data.append([date, open_price, high, low, close, volume, adj_close])
print(data[0])
```

```
[datetime.datetime(2014, 8, 19, 0, 0), 585.002622, 587.342658, 584.002627, 586.862643, 978
600, 586.862643]
```

OR

In [31]:
```
with open("/home/saif/LFS/datasets/Stock_Data.csv", newline='') as f:
    reader = csv.reader(f)
    header = next(reader)
#     data = [row for row in reader]

# print(header)
# print(data[0])
    data = []
    for row in reader:
        date = datetime.strptime(row[0], "%m/%d/%Y")
        open_price = float(row[1])
        high = float(row[2])
        low = float(row[3])
        close = float(row[4])
        volume = int(row[5])
        adj_close = float(row[6])

        data.append([date, open_price, high, low, close, volume, adj_close])
    print(data[0])
```

```
[datetime.datetime(2014, 8, 19, 0, 0), 585.002622, 587.342658, 584.002627, 586.862643, 978
600, 586.862643]
```

Step 3: Computing Daily Stock Returns: [Add this step in above program]

```
In [33]: #Compute & Store daily stock returns:
         return_path = "/home/saif/LFS/datasets/d_stock_returns.csv"
         file = open(return_path, 'w')
         writer = csv.writer(file)
         writer.writerow(["Date", "Return"])

         for i in range(len(data) - 1):
             todays_row = data[i]
             todays_date = todays_row[0]
             todays_price = todays_row[1]
             yesterdays_row = data[i+1]
             yesterdays_price = yesterdays_row[-1]

             daily_return = (todays_price - yesterdays_price)/yesterdays_price
             writer.writerow([todays_date, daily_return])
```

Format the date part for above code in last line:

formatted_date = todays_date.strftime("%m/%d/%Y")
writer.writerow([formatted_date, daily_return])

End to End Code in Other Style:

```
import csv
from datetime import datetime

with open("/home/saif/LFS/datasets/Stock_Data.csv", newline='') as f:
    reader = csv.reader(f)
    header = next(reader)

    data = []
    for row in reader:
        date = datetime.strptime(row[0], "%m/%d/%Y")
        open_price = float(row[1])
        high = float(row[2])
        low = float(row[3])
        close = float(row[4])
        volume = int(row[5])
        adj_close = float(row[6])

        data.append([date, open_price, high, low, close, volume, adj_close])
    print(data[0])

#Compute & Store daily stock returns:
with open("/home/saif/LFS/datasets/d_stock_returns1.csv", 'w') as f:
    writer = csv.writer(f)
    writer.writerow(["Date", "Return"])


    for i in range(len(data) - 1):
        todays_row = data[i]
        todays_date = todays_row[0]
        todays_price = todays_row[1]
        yesterdays_row = data[i+1]
        yesterdays_price = yesterdays_row[-1]

        daily_return = (todays_price - yesterdays_price)/yesterdays_price
        formatted_date = todays_date.strftime("%m/%d/%Y")
        writer.writerow([formatted_date, daily_return])
```

```
[datetime.datetime(2014, 8, 19, 0, 0), 585.002622, 587.342658, 584.002627, 586.862643, 978600, 586.862643]
```

Reading and Writing JSON in Python:

JSON format has been one of, if not the most, popular ways to **serialize** data. Especially in the **web development** world, you'll likely encounter **JSON** through one of the many **REST APIs**, **application configuration**, or even **simple data storage**.
Given its prevalence and impact on programming, at some point in your development you'll likely want to learn how to read JSON from a file or write JSON to a file.

Methods:

1) json.load(f) → Load JSON data from file
2) json.loads(s) → Load JSON data from string
3) json.dump(j, f) → Write JSON object to file
4) json.dumps(f) → Write JSON object as string

Reading JSON File:

```
In [28]: json_file = open("/home/saif/LFS/datasets/untold_story.txt", "r")
         movie = json.load(json_file)
         json_file.close()
```

```
In [31]: movie
Out[31]: {'title': 'The Untold Story',
          'release_year': 2016,
          'is_awesome': 'true',
          'won_oscal': 'false',
          'actos': ['Sushant Singh Rajput', 'Kiara Advani', 'Disha Patani', 'MS Dhoni'],
          'budget': None,
          'credits': {'director': 'Neeraj Pandey',
           'writer': 'Monica Ali',
           'composer': 'Ammal Mallik',
           'producer': 'Arun Pandey'}}
```

```
In [41]: movie["actors"]
Out[41]: ['Sushant Singh Rajput', 'Kiara Advani', 'Disha Patani', 'MS Dhoni']
```

ACSII Text:

By default, **json.dump** will ensure that **all** of your **text** in the given Python **dictionary** are **ASCII-encoded**.
If **non-ASCII** characters are **present**, then they're **automatically escaped**, as shown in the following example:

```
In [59]: import json
         data = {'item': 'Mobile', 'cost':'£150.00'}
         jstr = json.dumps(data, indent=4)
         print(jstr)

         {
             "item": "Mobile",
             "cost": "\u00a3150.00"
         }
```

This **isn't always acceptable**, and in **many** cases you may **want** to **keep** your **Unicode** characters **un-touched**. To do this, set the **ensure_ascii** option to **False**.

```
In [60]: import json
         data = {'item': 'Mobile', 'cost':'£150.00'}
         jstr = json.dumps(data, ensure_ascii=False, indent=4)
         print(jstr)

         {
             "item": "Mobile",
             "cost": "£150.00"
         }
```

Sorting:
In **JSON**, an object is **defined** as an **unordered set** of **name/value pairs**.

So the standard is saying that **key** order **isn't guaranteed**, but it's **possible** that you may **need** it for your **own purposes internally**. To achieve **ordering**, you can **pass True** to the **sort_keys option** when using **json.dump** or **json.dumps**.

```
In [61]: import json
         data = {'item': 'Mobile', 'cost':'£150.00'}
         jstr = json.dumps(data, sort_keys=True, ensure_ascii=False, indent=4)
         print(jstr)

         {
             "cost": "£150.00",
             "item": "Mobile"
         }
```

Loading JSON String data:

In [62]:
```
json_str_data = """{
"title":"The Untold Story",
"release_year":2016,
"is_awesome":"true",
"won_oscal":"false",
"actors":["Sushant Singh Rajput","Kiara Advani","Disha Patani","MS Dhoni"],
"budget":null}"""
strData = json.loads(json_str_data)
strData
```

Out[62]:
```
{'title': 'The Untold Story',
 'release_year': 2016,
 'is_awesome': 'true',
 'won_oscal': 'false',
 'actors': ['Sushant Singh Rajput',
  'Kiara Advani',
  'Disha Patani',
  'MS Dhoni'],
 'budget': None}
```

Write JSON data:

```
movieWrite = {}
movieWrite["title"]="The U\u0144told Story"
movieWrite["release_year"]=2016
movieWrite["actors"]=["Sushant Singh Rajput","Kiara Advani","Disha Patani","MS Dhoni"],
movieWrite["is_awesome"]="true"
movieWrite["won_oscar"]="false"
```

```
movieData = open("/home/saif/LFS/datasets/dhoniMovie.txt", "w", encoding="utf-8")
json.dump(movieWrite, movieData, ensure_ascii=False)
movieData.close()
```

View Data:

```
!cat /home/saif/LFS/datasets/dhoniMovie.txt
```

```
{"title": "The Uńtold Story", "release_year": 2016, "actors": [["Sushant Singh Rajput", "Kiara Advani", "Disha Pat
ani", "MS Dhoni"]], "is_awesome": "true", "won_oscar": "false"}
```

Reading and Writing XML in Python:

XML stands for **Extensible Markup Language**. It is similar to **HTML** in its **appearance** but, **XML** is used for **data presentation**, while **HTML** is used to **define what data** is being **used**. XML is exclusively designed to **send** and **receive data** back and forth between **clients** and **servers**.

Python XML Parsing Modules:

- Python allows **parsing** these **XML** documents using **two modules** namely, the **xml.etree.ElementTree** module and **Minidom (Minimal DOM Implementation)**.
- **Parsing** means to **read** information **from** a **file** and **split** it **into pieces** by **identifying parts** of that **particular XML** file.

xml.etree.ElementTree Module:

This module helps us **format XML data** in a **tree structure** which is the most natural representation of **hierarchical** data. **Element** type **allows storage** of **hierarchical** data structures in **memory** and has the following properties:

Property	Description
Tag	It is a string representing the type of data being stored
Attributes	Consists of a number of attributes stored as dictionaries
Text String	A text string having information that needs to be displayed
Tail String	Can also have tail strings if necessary
Child Elements	Consists of a number of child elements stored as sequences

There are **two** ways to **parse** the file using **'ElementTree'** module.

- The first is by using the **parse( ) function** and the second is **fromstring( ) function**.
- The **parse ( )** function **parses XML** document which is **supplied** as a **file** whereas, **fromstring ( ) parses XML** when supplied as a **string** i.e. **within triple quotes**.

Check Available Classes/Methods:

```
In [72]: import xml.etree.ElementTree as ET
         dir(ET)
```

Display classes in ET Module:

```
import xml.etree.ElementTree as ET
from inspect import getmembers, isclass, isfunction

for (name, member) in getmembers(ET, isclass):
    if not name.startswith("_"):
        print(name)
```

Element - ElementTree

```
import xml.etree.ElementTree as ET
from inspect import getmembers, isclass, isfunction

for (name, member) in getmembers(ET, isfunction):
    if not name.startswith("_"):
        print(name)
```

fromstring( ) - tostring( ) - parse( )

1) Using parse( ) function:

As mentioned earlier, this function takes **XML** in **file** format to **parse** it. Take a look at the following example:

```
In [77]: import xml.etree.ElementTree as ET
         mytree = ET.parse('/home/saif/LFS/datasets/food.xml')
         myroot = mytree.getroot()
         print(myroot)

         <Element 'metadata' at 0x7f2bc420dae0>
```

2) Using fromstring( ) function:

You can also use **fromstring( )** function to **parse** your **string** data. In case you want to do this, pass your XML as a string within triple quotes as follows:

```
In [78]: import xml.etree.ElementTree as ET
         data='''<?xml version="1.0" encoding="UTF-8"?>
         <metadata>
         <food>
             <item name="breakfast">Idly</item>
             <price>$2.5</price>
             <description>
            Two idly's with chutney
            </description>
             <calories>553</calories>
         </food>
         </metadata>
         '''
         myroot = ET.fromstring(data)
         #print(myroot)
         print(myroot.tag)

         metadata
```

Finding Elements of Interest:

The **root** consists of **child** tags as well. To **retrieve** the **child** of the **root** tag, you can use the following:

```
In [80]: print(myroot[0].tag)

         food
```

Now, if you want to **retrieve all first-child tags** of the **root**, you can **iterate** over it using **for loop** as follows:

```
In [81]: for x in myroot[0]:
             print(x.tag, x.attrib)

         item {'name': 'breakfast'}
         price {}
         description {}
         calories {}
```

All the **items** returned are the **child attributes** and **tags** of **food**.

To **separate out** the **text** from **XML** using **ElementTree**, you can make use of the **text attribute**. For example, in case I want to **retrieve all** the information about the **first food item**, I should use the following piece of code:

```
In [85]: for x in myroot[0]:
             print(x.text)

         Idly
         $2.5

            Two idly's with chutney

         553
```

As you can see, the **text** information of the **first item** has been **returned** as the **output**.

Now if you want to **display all** the **items** with their **particular price**, you can make use of the **get ( )** method. This method **accesses** the **element's attributes**.

```
In [97]: for x in myroot.findall('food'):
             item = x.find('item').text
             price = x.find('price').text
             print(item,":",price)

         Idly : $2.5
         Paper Dosa : $2.7
         Upma : $3.65
         Bisi Bele Bath : $4.50
         Kesari Bath : $1.95
```

Modifying XML files:
The **elements** present your **XML file** can be **manipulated**. To do this, you can use the **set ( ) function**.

Adding to XML:
The following example shows how you can **add** something to the **description** of **items**.

```
In [9]: for description in myroot.iter('description'):
            new_desc = str(description.text)+'wil be served'
            description.text = str(new_desc)
            description.set('updated', 'yes')

        mytree.write('/home/saif/LFS/datasets/food1.xml')
        !cat /home/saif/LFS/datasets/food1.xml | head -10
```

```
<metadata>
<food>
    <item name="breakfast">Idly</item>
    <price>$2.5</price>
    <description updated="yes">
   Two idly's with chutney
   wil be served</description>
    <calories>553</calories>
</food>
<food>
```

Note: The **write ( )** function helps create a **new xml file** and **writes** the **updated output** to the **same**.

To add a **new subtag**, you can make use of the **SubElement ( )** method. For example, if you want to **add** a **new specialty tag** to the **first item Idly**, you can do as follows:

```
In [10]: ET.SubElement(myroot[0], 'speciality')
         for x in myroot.iter('speciality'):
             new_desc = 'South Indian Special'
             x.text = str(new_desc)

         mytree.write('/home/saif/LFS/datasets/food2.xml')
         !cat /home/saif/LFS/datasets/food2.xml | head -10
```

```
<metadata>
<food>
    <item name="breakfast">Idly</item>
    <price>$2.5</price>
    <description updated="yes">
   Two idly's with chutney
   wil be served</description>
    <calories>553</calories>
<speciality>South Indian Special</speciality></food>
<food>
```

As you can see, a **new tag** has been **added** under the **first food tag**. You can **add** tags **wherever** you want by **specifying** the **subscript** within **[ ] brackets**.

Deleting from XML:

To **delete attributes** or **sub-elements** using **ElementTree**, you can make use of the **pop ( ) method**. This method will **remove** the **desired attribute** or **element** that is **not needed** by the **user**.

```
In [12]: myroot[0][0].attrib.pop('name', None)

         # create a new XML file with the results
         mytree.write('/home/saif/LFS/datasets/food3.xml')
         !cat /home/saif/LFS/datasets/food3.xml | head -10
```

```
<metadata>
<food>
    <item>Idly</item>
    <price>$2.5</price>
    <description updated="yes">
   Two idly's with chutney
   wil be served</description>
    <calories>553</calories>
<speciality>South Indian Special</speciality></food>
<food>
```

The above image shows that the **name attribute** has been **removed** from the **item tag**.

To **remove** the **complete tag**, you can use the same **pop ( )** method as follows:

```
In [13]: myroot[0].remove(myroot[0][0])

         mytree.write('/home/saif/LFS/datasets/food3.xml')
         !cat /home/saif/LFS/datasets/food3.xml | head -10
```

```
<metadata>
<food>
    <price>$2.5</price>
    <description updated="yes">
   Two idly's with chutney
   wil be served</description>
    <calories>553</calories>
<speciality>South Indian Special</speciality></food>
<food>
    <item name="breakfast">Paper Dosa</item>
```

The **output** shows that the **first sub-element** of the **food tag** has been **deleted**.

In case you want to **delete all tags**, you can make use of the **clear ( )** function as follows:

```
In [ ]: myroot[0].clear()
        mytree.write('/home/saif/LFS/datasets/food3.xml')
        !cat /home/saif/LFS/datasets/food3.xml | head -10
```

When the **above code** is **executed**, the **first child** of **food tag** will be **completely deleted including all** the **sub-tags**.

Python Internet Access using Urllib.Request and urlopen ( ):

What is urllib?

- **urllib** is a **Python module** that can be **used** for **opening URLs**.
- To read a **JSON response** there is a **widely used library** called **urllib** in python.
- This **library helps** to **open** the **URL** and **read** the **JSON response** from the **web**.
- To use this **library** in **python** and **fetch JSON response** we have to **import** the **json** and **urllib** in our code and the **json.loads ( )** method **returns JSON object**.

Python MySQL:
To build the real world applications, **connecting** with the **databases** is the **necessity** for the **programming languages**. However, python **allows us** to **connect** our **application** to the **databases** like **MySQL**, **SQLite**, **MongoDB**, and **many others**.

Environment Setup: Install mysql.connector
To connect the **python application** with the **MySQL database**, we must **import** the **mysql.connector module** in the **program**. The **mysql.connector** is **not** a **built-in module** that **comes** with the **python installation**.

Execute the following command to install it using pip installer:
python -m pip install mysql-connector
pip list | grep mysql

Database Connection:
Let's **discuss** the **steps** to **connect** the **python application** to the **database**.

Following are the **steps** to **connect** a **python application** to **database**.
1) Import **mysql.connector module**
2) Create the **connection object**.
3) Create the **cursor object**
4) **Execute** the **query**

Creating the connection:
- To create a **connection** between **MySQL database** and the **python application**, **connect ( )** method of **mysql.connector** module is used.
- Pass the **database details** like **HostName**, **username**, and the **database** password in the **method call**. The method **returns** the **connection object**.

Syntax:
Connection-Object= mysql.connector.connect (host = <host-name>, user = <username>, passwd = <password>)

import mysql.connector
 # Create the connection object:
myconn = mysql.connector.connect (host = "localhost", user = "root",
passwd = "Welcome@123")
 # Printing the connection object:
print (myconn)

```
>>> import mysql.connector
>>> myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "root")
>>> print(myconn)
<mysql.connector.connection_cext.CMySQLConnection object at 0x000001D885FE4190>
```

Here, we **can** also **specify** the **database name** in **connect ( ) method** if we want to **connect** to a **specific database**.

```
import mysql.connector
myconn = mysql.connector.connect (host = "localhost", user = "root",
                                  passwd = "Welcome@123", database = "retail_db")
print (myconn)
```

```
import mysql.connector
myconn = mysql.connector.connect (host = "localhost", user = "root",
                                  passwd = "Welcome@123", database = "retail_db")
print (myconn)
```

```
<mysql.connector.connection_cext.CMySQLConnection object at 0x7f6251669730>
```

Creating a cursor object:

- The **cursor object** can be **defined** as an **abstraction** specified in the **Python DB-API 2.0**. It **facilitates** us to have **multiple separate working environments** through the **same connection** to the **database**.
- We can **create** the **cursor object** by **calling** the **'cursor' function** of the **connection object**.
- The **cursor object** is an **important aspect** of **executing queries** to the **databases**.

Syntax:

<my_cur> = conn.cursor ( )

```
import mysql.connector
myconn = mysql.connector.connect (host = "localhost", user = "root",
                                  passwd = "Welcome@123", database = "retail_db")
print (myconn)
mycuror = myconn.cursor ()          # Creating the cursor object
print (mycuror)
```

```
import mysql.connector
myconn = mysql.connector.connect (host = "localhost", user = "root",
                                  passwd = "Welcome@123", database = "retail_db")
print (myconn)
mycuror = myconn.cursor ()        # Creating the cursor object
print (mycuror)
```

```
<mysql.connector.connection_cext.CMySQLConnection object at 0x7f6251669cd0>
CMySQLCursor: (Nothing executed yet)
```

Getting the list of existing databases:
We can get the **list** of **all** the **databases** by **using** the following **MySQL query**.
show databases;

```
import mysql.connector
myconn = mysql.connector.connect(host = "localhost", user = "root", passwd = "root")
cursor = myconn.cursor()
cursor.execute("SHOW DATABASES")
# 'fetchall()' method fetches all the rows from the last executed statement
databases = cursor.fetchall()
print(databases)
```

```
>>> import mysql.connector
>>> myconn = mysql.connector.connect(host = "localhost", user = "root", passwd = "root")
>>> cursor = myconn.cursor()
>>> cursor.execute("SHOW DATABASES")
>>> databases = cursor.fetchall() # 'fetchall()' method fetches all the rows from the last executed statement
>>> print(databases)
[('information_schema',), ('mysql',), ('performance_schema',), ('spark_mysql',), ('sys',)]
```

```
import mysql.connector
myconn = mysql.connector.connect (host = "localhost", user = "root",
                                  passwd = "Welcome@123", database = "retail_db")
cursor = myconn.cursor()
cursor.execute ("show databases")
for x in cursor:
  print(x)
```

```
import mysql.connector
myconn = mysql.connector.connect (host = "localhost", user = "root",
                                  passwd = "Welcome@123")
cursor = myconn.cursor()
cursor.execute ("show databases")
for x in cursor:
    print(x)
```

```
('information_schema',)
('metastore',)
('mysql',)
('performance_schema',)
('retail_db',)
('saif_db',)
('sys',)
```

Creating new databases:
Once a **database connection** is **established**, we are **ready** to **create tables** or **records** into the **database tables** using **execute method** of the **created cursor**.

import mysql.connector as mysql
db = mysql.connect(host = "localhost", user = "root", passwd = "Welcome@123")
cursor = db.cursor()
cursor.execute("create database test_db")

```
import mysql.connector as mysql
db = mysql.connect(host = "localhost", user = "root", passwd = "Welcome@123")
cursor = db.cursor()
cursor.execute("create database test_db")
```

```
cursor.execute("show databases")
for x in cursor:
    print(x)
```

```
('information_schema',)
('metastore',)
('mysql',)
('performance_schema',)
('saif_db',)
('sys',)
('test_db',)
```

Creating a Table:
To **create** a **table** in **MySQL**, use the "**CREATE TABLE**" statement. Make **sure** you **define** the **name** of the **database** when you **create** the **connection**.

import mysql.connector as mysql
db = mysql.connect(host = "localhost", user = "root",
passwd = "Welcome@123", database = "test_db")
cursor = db.cursor()
cursor.execute("CREATE TABLE users (name VARCHAR(255),
user_name VARCHAR(255))")

```
import mysql.connector as mysql
db = mysql.connect(host = "localhost", user = "root",
                   passwd = "Welcome@123", database = "test_db")
cursor = db.cursor()
cursor.execute("CREATE TABLE users (name VARCHAR(255), user_name VARCHAR(255))")
```

```
cursor.execute("show tables")
for x in cursor:
    print(x)
```

```
('users',)
```

OR

```
In [39]: import mysql.connector as mysql
         conn = mysql.connect(host = "localhost", user = "root",
                                              passwd = "Welcome@123", database = "test_db")
         cursor = conn.cursor()
         sql = """CREATE TABLE employee (
                  FIRST_NAME  CHAR(20) NOT NULL,
                  LAST_NAME  CHAR(20),
                  AGE INT,
                  SEX CHAR(1),
                  INCOME FLOAT)"""
         cursor.execute(sql)
```

Let's **see all** the **tables** present in the **database** using the **SHOW TABLES** statement.

```
In [43]: import mysql.connector as mysql
         conn = mysql.connect(host = "localhost", user = "root",
                              passwd = "Welcome@123", database = "test_db")
         cursor = conn.cursor()
         cursor.execute("show tables")

         # it returns list of tables present in the database:
         tables = cursor.fetchall()
         print(tables)

         [('employee',), ('users',)]
```

fetchall ( ) → Returns all tables in a list

```
In [44]: # showing all the tables one by one:
         for table in tables:
             print(table)

         ('employee',)
         ('users',)
```

Or use **for loop** to **iterate one** by **one**.

Desc tables in python:

```
>>> cursor.execute("desc employee")
>>> cursor.fetchall()
[('FIRST_NAME', 'char(20)', 'NO', '', None, ''), ('LAST_NAME', 'char(20)', 'YES', '', None, ''), ('AGE', 'int', 'YES',
', None, ''), ('SEX', 'char(1)', 'YES', '', None, ''), ('INCOME', 'float', 'YES', '', None, '')]
```

```
>>> cursor.execute("desc employee")
>>> t = cursor.fetchall()
>>> print(t)
[('FIRST_NAME', 'char(20)', 'NO', '', None, ''), ('LAST_NAME', 'char(20)', 'YES', '', None, ''), ('AGE', 'int', 'YES',
', None, ''), ('SEX', 'char(1)', 'YES', '', None, ''), ('INCOME', 'float', 'YES', '', None, '')]
```

Inserting Data
Use **INSERT INTO** statement to **insert** into the **table**.

Inserting a Single Row:
Let's see how to **insert** a **single row** into the **table**.

```
In [6]: query = "insert into users (name, user_name) values (%s, %s)"
        values = ("Saif", "saifshk")
        cursor.execute(query, values)     # executing the query with values
        conn.commit()
        print(cursor.rowcount, "record inserted")

        1 record inserted
```

Inserting Multiple Rows:
To **insert multiple rows** into the **table**, we use the **executemany** ( ) method. It takes a **list** of **tuples** containing the **data** as a **second parameter** and a **query** as the **first argument**.

```
In [12]: query = "insert into users (name, user_name) values (%s, %s)"
         values = [("Tausif", "Tausifshk"),
                   ("Ram", "ramshirali"),
                   ("Aniket", "aniketmishra")]
         cursor.executemany(query, values)
         conn.commit()
         print(cursor.rowcount, "records inserted")

         3 records inserted
```

Select Data:
To **retrieve** the **data** from a table we use, **SELECT** statement.

```
In [13]: query = "select * from users"
         cursor.execute(query)
         records = cursor.fetchall()
         print(records)

         [('Tausif', 'Tausifshk'), ('Ram', 'ramshirali'), ('Aniket', 'aniketmishra')]
```

```
In [14]: for rows in records:
             print(rows)

         ('Tausif', 'Tausifshk')
         ('Ram', 'ramshirali')
         ('Aniket', 'aniketmishra')
```

Formatting Output:

```
In [23]: print("Name:Username")
         for rows in records:
             print("%s|%s"%(rows[0],rows[1]))

         Name:Username
         Tausif|Tausifshk
         Ram|ramshirali
         Aniket|aniketmishra
```

Disconnecting Database:

To **disconnect db** connection use **close ( )** method. If the **connection** to a **database** is **closed** by the **user** with the **close ( )** method, any **outstanding transactions** are **rolled back** by the **DB**. However, instead of **depending** on any of **DB** lower level implementation details, your **application** would be **better off** calling **commit** or **rollback explicitly**.

Syntax: db.close ( )

```
In [28]: #Import Mysql Connector
         import mysql.connector as mysql

         #Create DB Connection:
         conn = mysql.connect(host="localhost", user="root",
                              passwd="Welcome@123", database="test_db")

         #Create a Cursor Object using Cursor Method:
         cursor = conn.cursor()

         #Prepare SQL Query to Delete Data:
         sql = "delete from users where name = '%s'" % ("Ram")

         try:
             #Execute SQL Query:
             cursor.execute(sql)

             #Commit Changes in DB:
             conn.commit()

         except:
             #Rollback for any Failures:
             conn.rollback()

             #Disonnect From Server:
         conn.close()
```