

# **Fundamentals of Data Science 21CSS202T**

# Unit II

## **Unit-2: INTRODUCTION TO PYTHON DEBUGGING 9 hours**

Debug python scripts using PDB and IDE, Classify Errors, Develop Unit Tests , Create project Skeletons,

Implement Database using SQLite, Perform CRUD operations SQLite database,

JSON file – Read, Write and Parse JSON file - JSON Conversion – to dictionary, to JSON, to JSON String, JSON schema – Schema Validation, Resolving JSON Reference, Extending Validator Classes - Virtual Environment, Floating point Arithmetic – Issues and Limitations,

Implement Regular Expression and its Basic Functions - findall(),search(),split(),sub(), Use Classes, Objects, and Attributes, Develop applications based on Object Oriented Programming and Methods

**T4: Implement programs to handle JSON files**

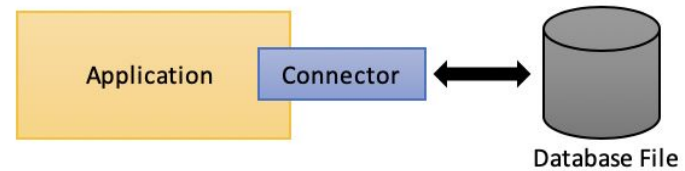
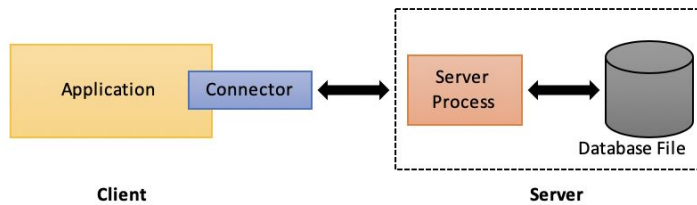
**T5: Implement programs to work with Regex functions and classes**

**T6: Implementing Debugging and creating projects in python IDE**

# SQLite

- SQLite is embedded relational database management system. It is self-contained, serverless, zero configuration and transactional SQL database engine.
- SQLite was designed originally on August 2000. It is named SQLite because it is very light weight (less than 500Kb size) unlike other database management systems like SQL Server or Oracle.
- SQLite does not have a separate server process. It reads and writes directly to ordinary disk files.

# SQL vs SQLite

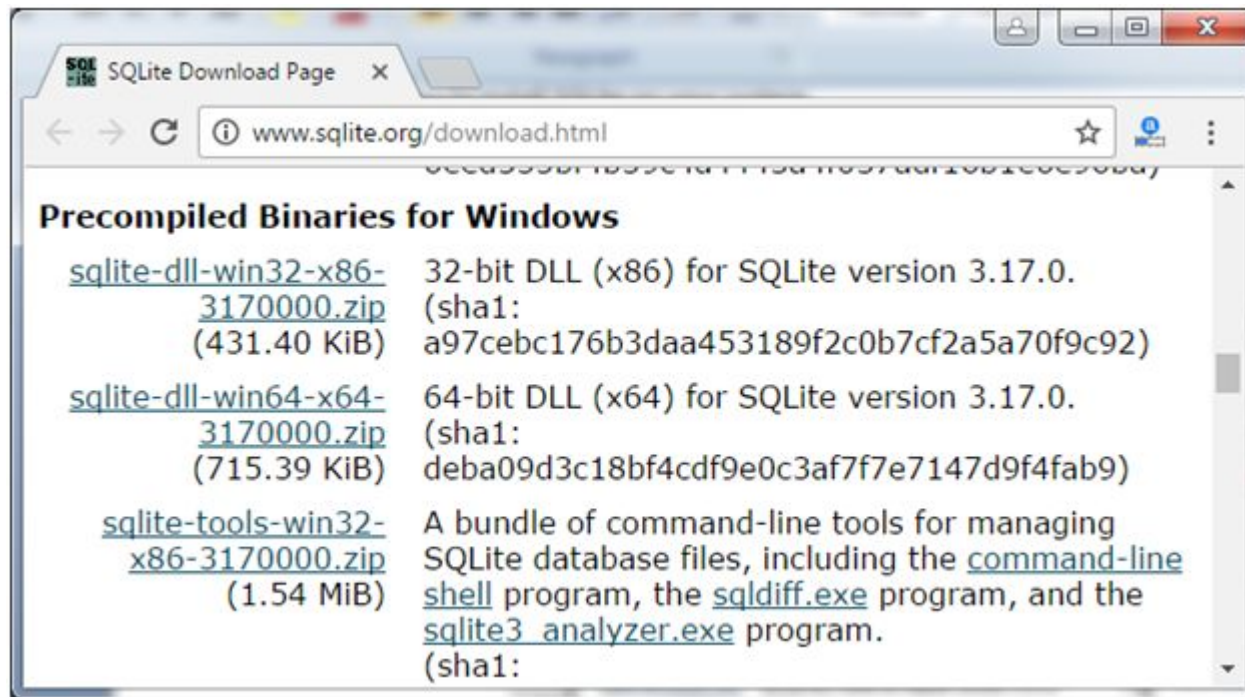


# SQLite

Year	Happenings
2000	SQLite was designed by D. Richard Hipp for the purpose of no administration required for operating a program.
2000	In August SQLite 1.0 released with GNU database manager.
2011	Hipp announced to add UNQL interface to SQLite db and to develop UNQLite (Document oriented database).

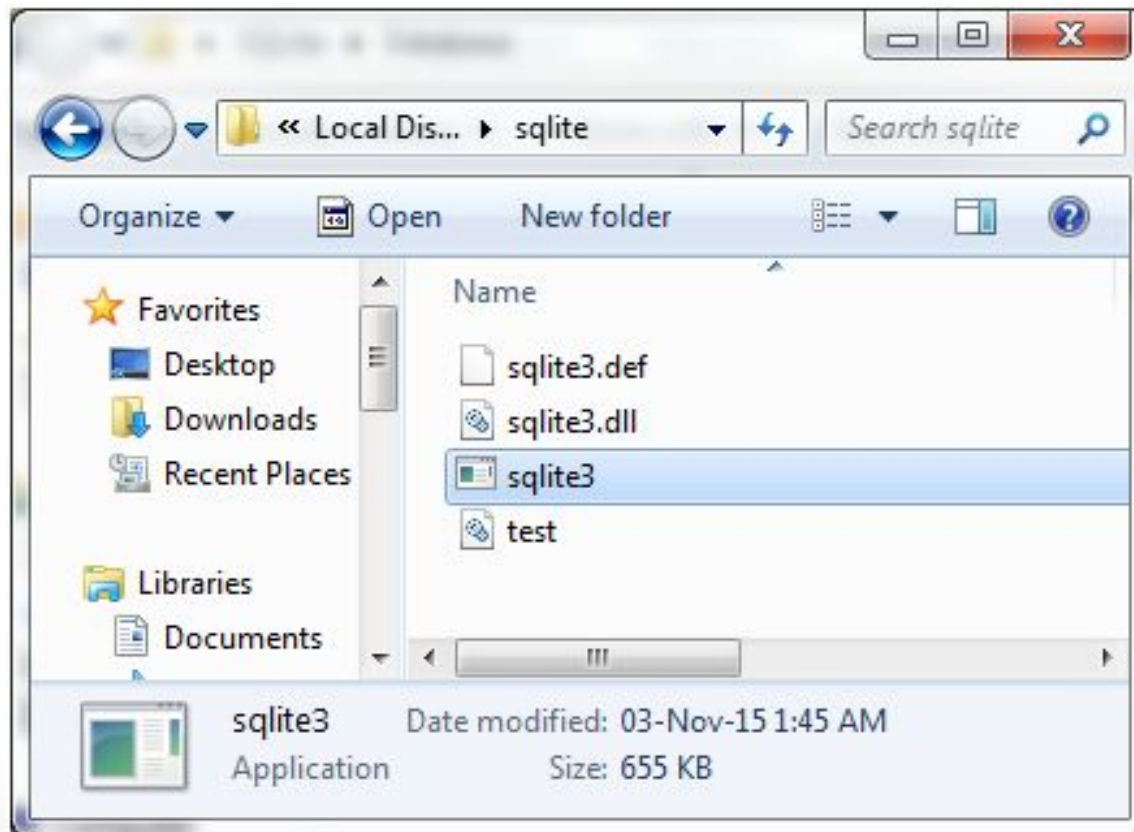
# Installation

- Go to SQLite official website download page <http://www.sqlite.org/download.html> And download precompiled binaries from Windows section.

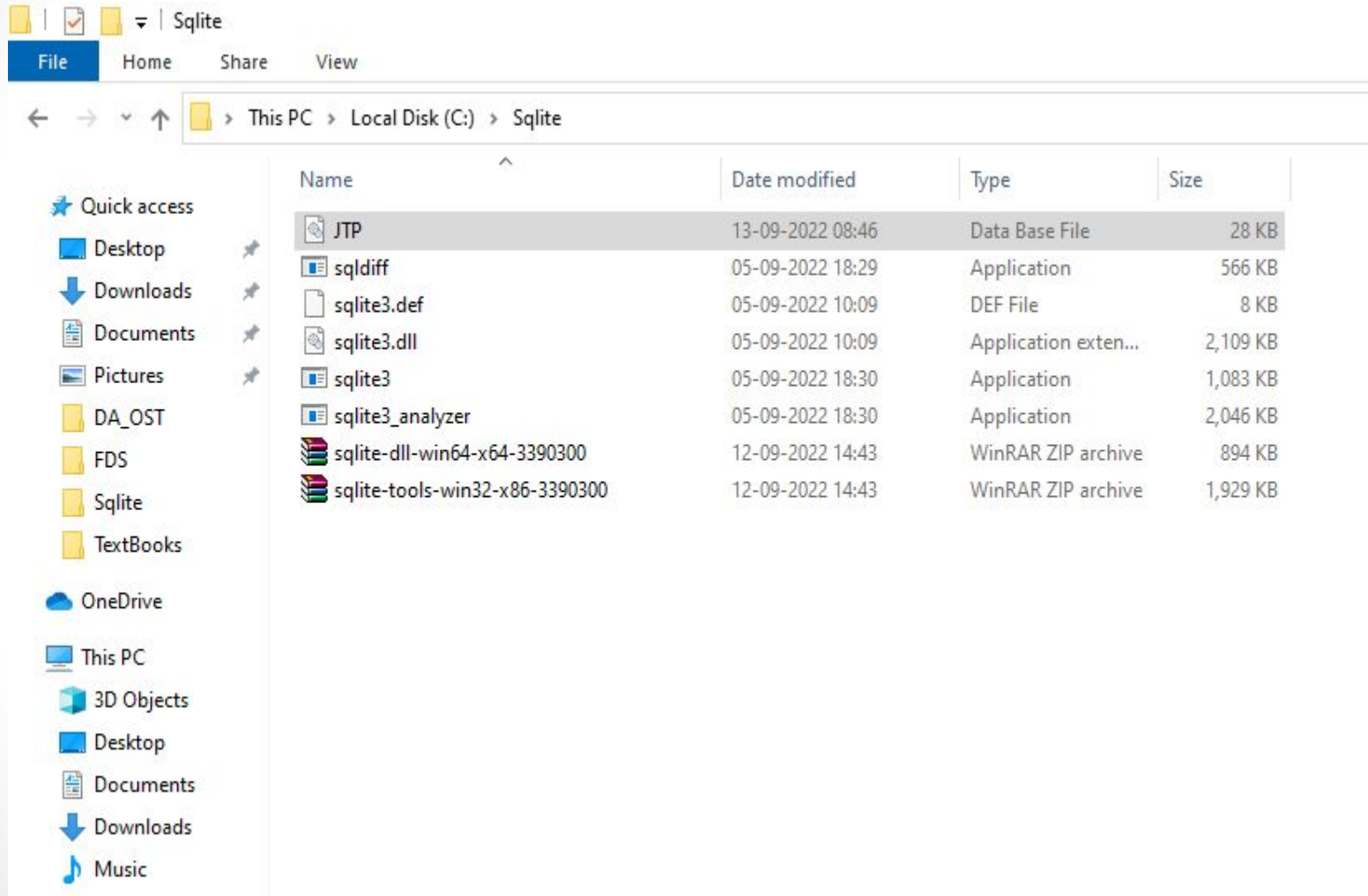


# Installation

Create a folder named sqlite in C directory and expand these files.

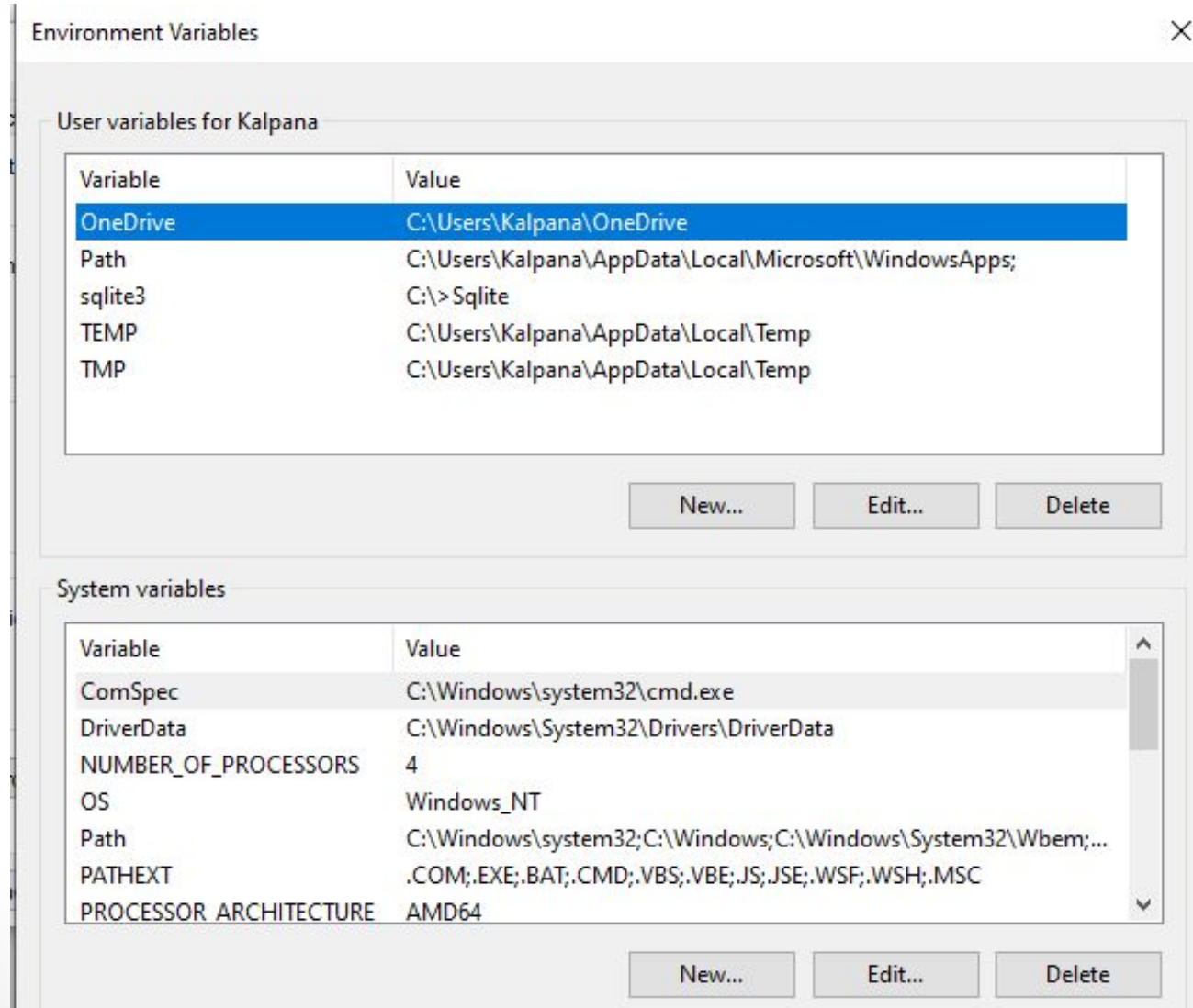


# Installation

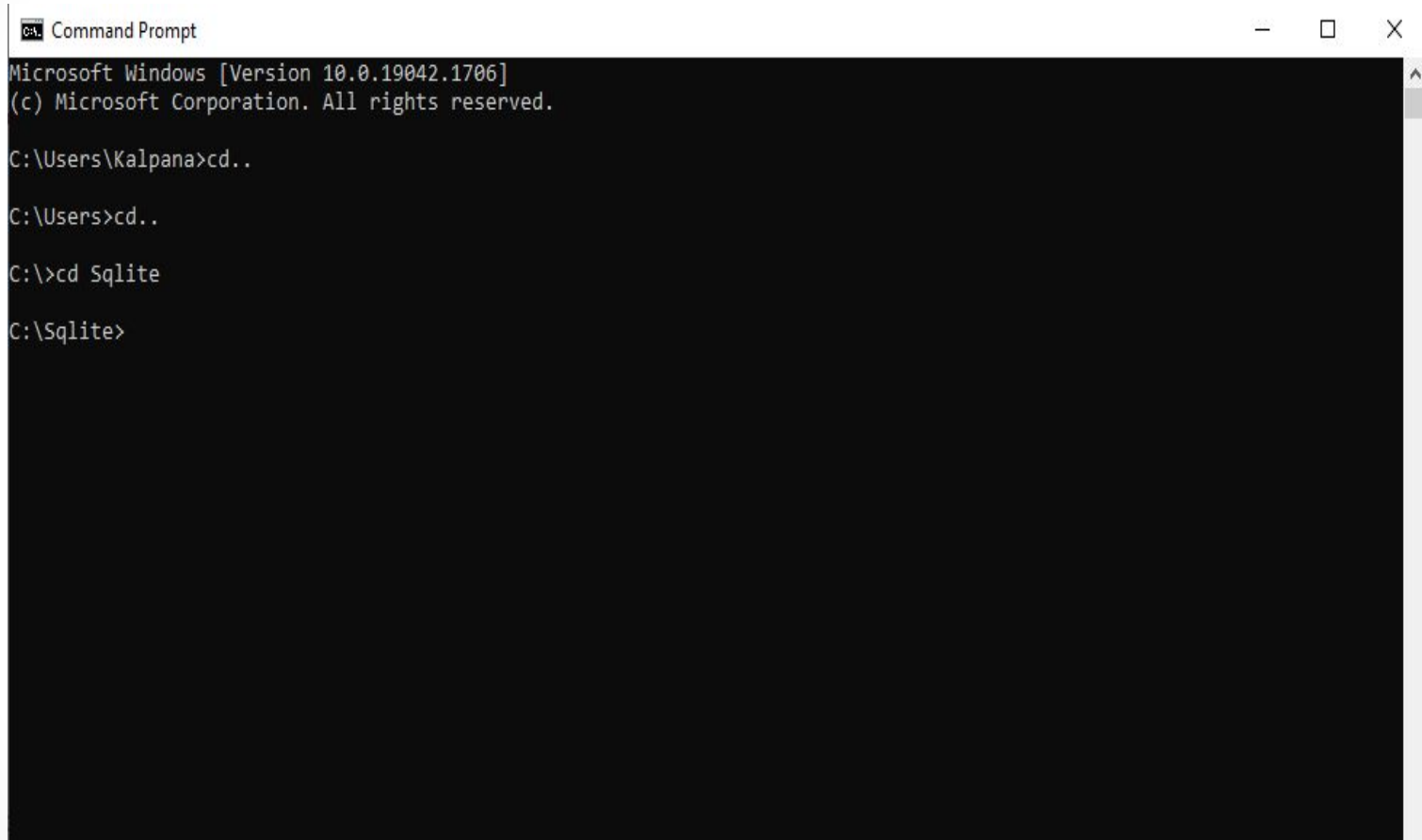




# Installation



# Installation



```
CA: Command Prompt
Microsoft Windows [Version 10.0.19042.1706]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Kalpana>cd..

C:\Users>cd..

C:\>cd Sqlite

C:\Sqlite>
```

The image shows a Windows Command Prompt window with a black background and white text. The title bar at the top reads "CA: Command Prompt" and includes standard window control buttons (minimize, maximize, close). The command history shows the user navigating from their home directory to the root of the drive and then to a subdirectory named "Sqlite".

# Create Database

- sqlite3 DatabaseName.db

Command Prompt

```
Microsoft Windows [Version 10.0.19042.1706]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Kalpana>cd..

C:\Users>cd..

C:\>cd Sqlite

C:\Sqlite>dir
Volume in drive C has no label.
Volume Serial Number is 7051-49B0

Directory of C:\Sqlite

13-09-2022  08:46    <DIR>          .
13-09-2022  08:46    <DIR>          ..
13-09-2022  08:46             28,672 JTP.db
05-09-2022  18:29             579,072 sqldiff.exe
12-09-2022  14:43             914,455 sqlite-dll-win64-x64-3390300.zip
12-09-2022  14:43            1,974,865 sqlite-tools-win32-x86-3390300.zip
05-09-2022  10:09              7,786 sqlite3.def
05-09-2022  10:09            2,159,616 sqlite3.dll
05-09-2022  18:30            1,108,992 sqlite3.exe
05-09-2022  18:30            2,095,104 sqlite3_analyzer.exe
               8 File(s)              8,868,562 bytes
               2 Dir(s)  63,075,438,592 bytes free

C:\Sqlite>_
```

# Create Database

- **sqlite3 Test.db**

Command Prompt - sqlite3 Test.db

Directory of C:\Sqlite

```
13-09-2022 08:46 <DIR> .
13-09-2022 08:46 <DIR> ..
13-09-2022 08:46      28,672 JTP.db
05-09-2022 18:29    579,072 sqldiff.exe
12-09-2022 14:43    914,455 sqlite-dll-win64-x64-3390300.zip
12-09-2022 14:43   1,974,865 sqlite-tools-win32-x86-3390300.zip
05-09-2022 10:09     7,786 sqlite3.def
05-09-2022 10:09   2,159,616 sqlite3.dll
05-09-2022 18:30   1,108,992 sqlite3.exe
05-09-2022 18:30   2,095,104 sqlite3_analyzer.exe
            8 File(s)      8,868,562 bytes
            2 Dir(s)  63,075,438,592 bytes free
```

C:\Sqlite>sqlite3 Test.db

SQLite version 3.39.3 2022-09-05 11:02:23

Enter ".help" for usage hints.

sqlite> █

# Create Database

- `sqlite> .databases`

```
C:\> Command Prompt - sqlite3 Test.db

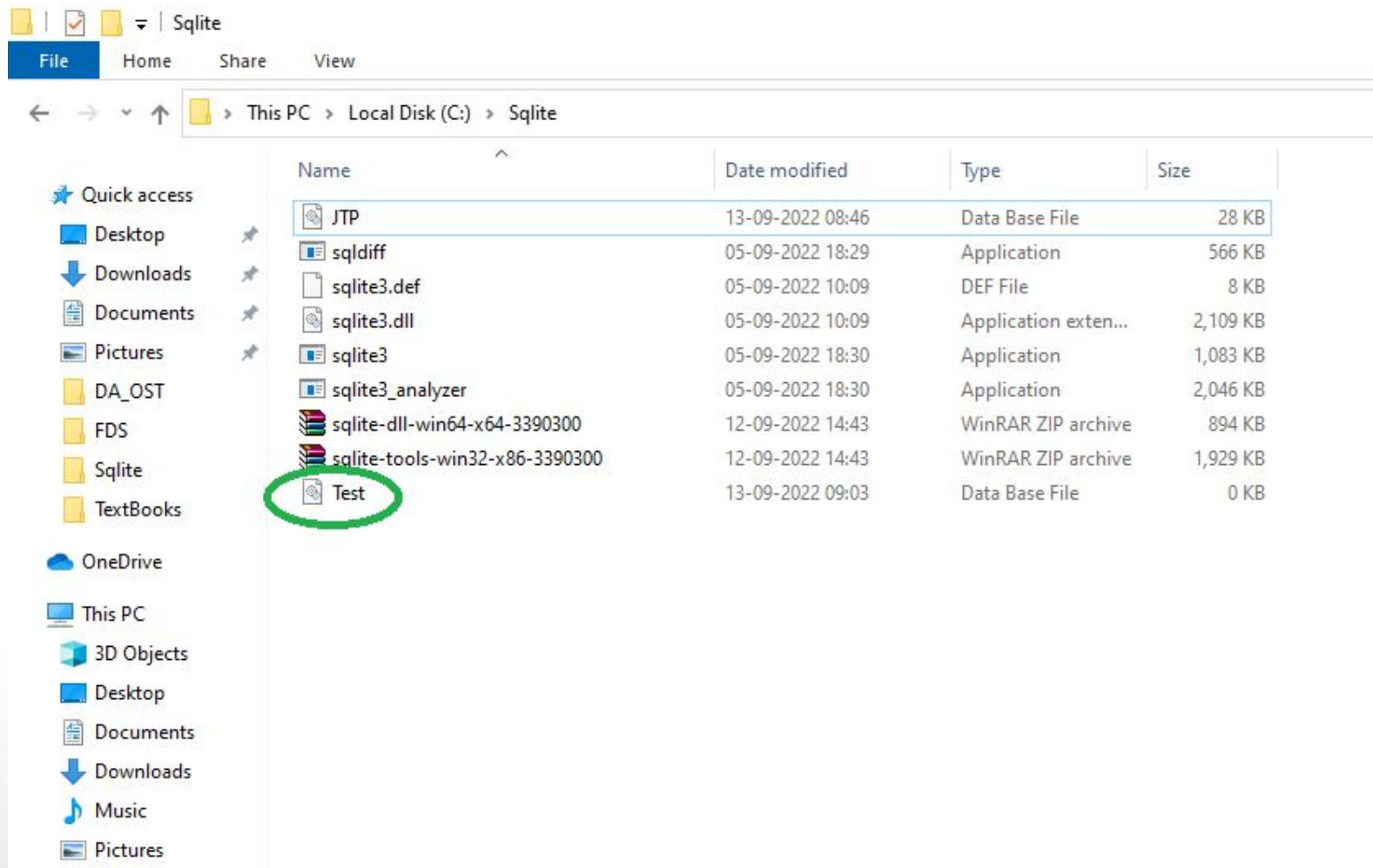
Directory of C:\Sqlite

13-09-2022  08:46    <DIR>          .
13-09-2022  08:46    <DIR>          ..
13-09-2022  08:46             28,672 JTP.db
05-09-2022  18:29             579,072 sqldiff.exe
12-09-2022  14:43             914,455 sqlite-dll-win64-x64-3390300.zip
12-09-2022  14:43            1,974,865 sqlite-tools-win32-x86-3390300.zip
05-09-2022  10:09              7,786 sqlite3.def
05-09-2022  10:09            2,159,616 sqlite3.dll
05-09-2022  18:30            1,108,992 sqlite3.exe
05-09-2022  18:30            2,095,104 sqlite3_analyzer.exe
             8 File(s)            8,868,562 bytes
             2 Dir(s)  63,075,438,592 bytes free

C:\Sqlite>sqlite3 Test.db
SQLite version 3.39.3 2022-09-05 11:02:23
Enter ".help" for usage hints.
sqlite> .databases
main: C:\Sqlite\Test.db r/w
sqlite> _
```

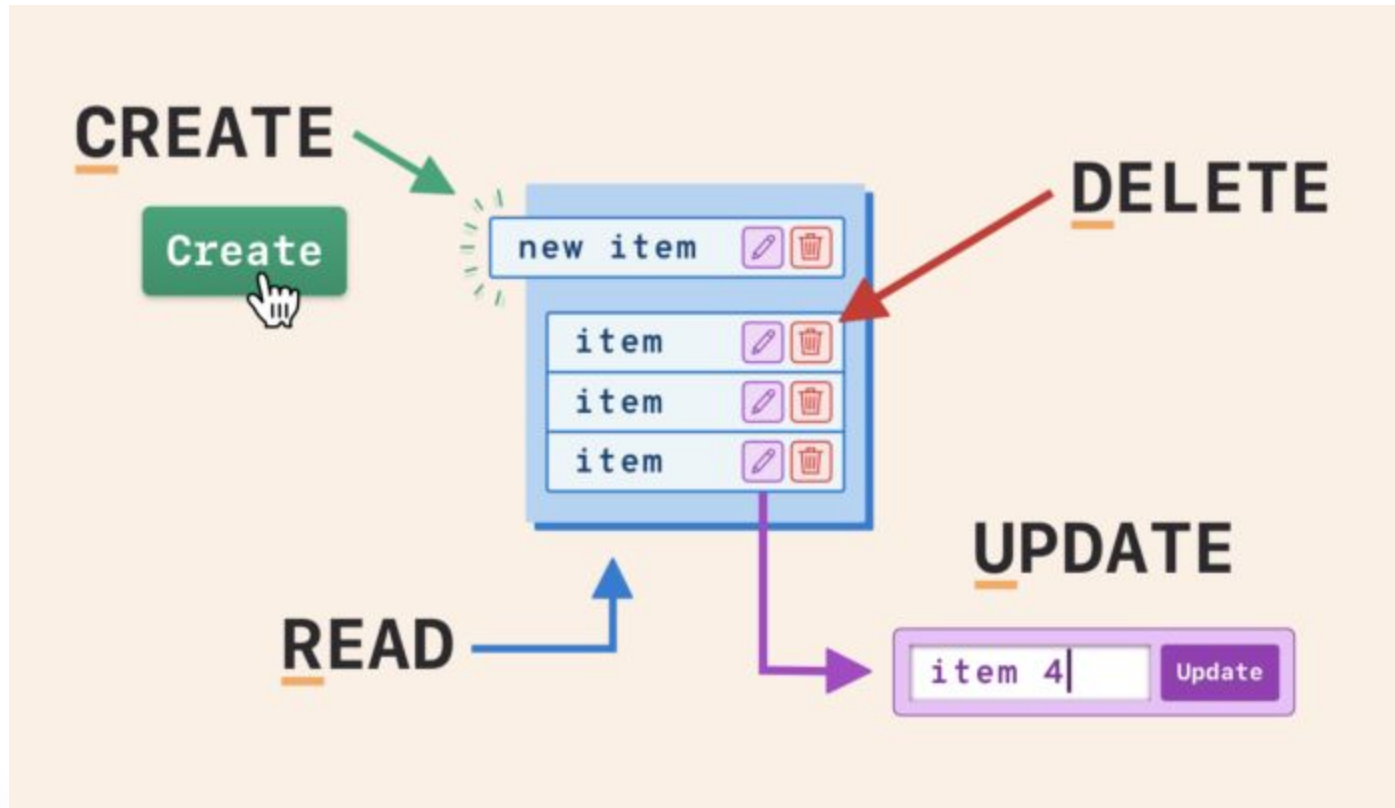
# Create Database

- **sqlite> .databases**





# CRUD Operations



# Create Table

- CREATE TABLE statement is used to create a new table.
- While creating the table, we name that table and define its column and data types of each column.

## Syntax:

```
CREATE TABLE database_name.table_name(  
    column1 datatype PRIMARY KEY(one or more columns),  
    column2 datatype,  
    column3 datatype,  
    ....  
    columnN datatype,  
);
```



# Create Table

```
CREATE TABLE STUDENT(  
    ID INT PRIMARY KEY NOT NULL,  
    NAME TEXT NOT NULL,  
    AGE INT NOT NULL,  
    ADDRESS CHAR(50),  
    FEES REAL  
);
```

# Create Table

Command Prompt - sqlite3 Test.db

```
Directory of C:\Sqlite

13-09-2022  08:46    <DIR>          .
13-09-2022  08:46    <DIR>          ..
13-09-2022  08:46             28,672 JTP.db
05-09-2022  18:29             579,072 sqldiff.exe
12-09-2022  14:43             914,455 sqlite-dll-win64-x64-3390300.zip
12-09-2022  14:43            1,974,865 sqlite-tools-win32-x86-3390300.zip
05-09-2022  10:09              7,786 sqlite3.def
05-09-2022  10:09            2,159,616 sqlite3.dll
05-09-2022  18:30            1,108,992 sqlite3.exe
05-09-2022  18:30            2,095,104 sqlite3_analyzer.exe
               8 File(s)              8,868,562 bytes
               2 Dir(s) 63,075,438,592 bytes free

C:\Sqlite>sqlite3 Test.db
SQLite version 3.39.3 2022-09-05 11:02:23
Enter ".help" for usage hints.
sqlite> .databases
main: C:\Sqlite\Test.db r/w
sqlite> CREATE TABLE STUDENT(
...>     ID INT PRIMARY KEY     NOT NULL,
...>     NAME           TEXT     NOT NULL,
...>     AGE             INT       NOT NULL,
...>     ADDRESS          CHAR(50),
...>     FEES             REAL
...> );
sqlite>
```

# Create Table

Command Prompt - sqlite3 Test.db

Directory of C:\Sqlite

```
13-09-2022 08:46 <DIR> .
13-09-2022 08:46 <DIR> ..
13-09-2022 08:46      28,672 JTP.db
05-09-2022 18:29    579,072 sqldiff.exe
12-09-2022 14:43    914,455 sqlite-dll-win64-x64-3390300.zip
12-09-2022 14:43   1,974,865 sqlite-tools-win32-x86-3390300.zip
05-09-2022 10:09     7,786 sqlite3.def
05-09-2022 10:09   2,159,616 sqlite3.dll
05-09-2022 18:30   1,108,992 sqlite3.exe
05-09-2022 18:30   2,095,104 sqlite3_analyzer.exe
            8 File(s)      8,868,562 bytes
            2 Dir(s)  63,075,438,592 bytes free
```

C:\Sqlite>sqlite3 Test.db

SQLite version 3.39.3 2022-09-05 11:02:23

Enter ".help" for usage hints.

sqlite> .databases

main: C:\Sqlite\Test.db r/w

sqlite> CREATE TABLE STUDENT(

...> ID INT PRIMARY KEY NOT NULL,

...> NAME TEXT NOT NULL,

...> AGE INT NOT NULL,

...> ADDRESS CHAR(50),

...> FEES REAL

...> );

sqlite> .tables

STUDENT

sqlite>

# Create Table

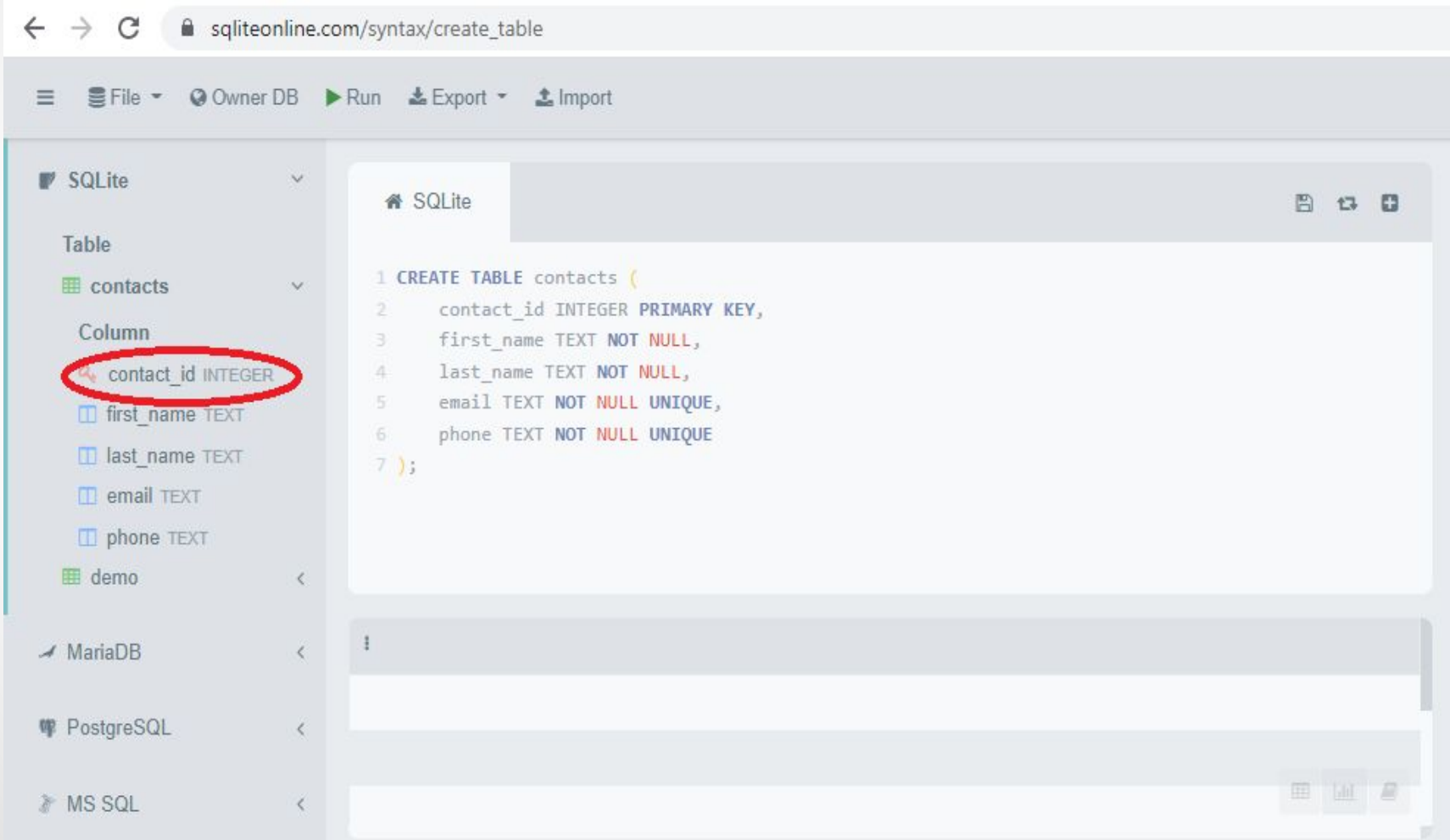
<https://sqliteonline.com/>

The screenshot displays the SQLiteOnline.com web application interface. The browser's address bar shows the URL `sqliteonline.com/syntax/create_table`. The application's top navigation bar includes a hamburger menu, a 'File' dropdown, 'Owner DB', 'Run', 'Export', and 'Import' buttons, along with a 'Sign in' link. On the left sidebar, a tree view shows the 'SQLite' database selected, with a 'Table' section containing 'contacts' and 'demo'. The main editor area shows a SQL query to create a 'contacts' table with columns: 'contact\_id' (INTEGER PRIMARY KEY), 'first\_name' (TEXT NOT NULL), 'last\_name' (TEXT NOT NULL), 'email' (TEXT NOT NULL UNIQUE), and 'phone' (TEXT NOT NULL UNIQUE). The query is as follows:

```
1 CREATE TABLE contacts (  
2   contact_id INTEGER PRIMARY KEY,  
3   first_name TEXT NOT NULL,  
4   last_name TEXT NOT NULL,  
5   email TEXT NOT NULL UNIQUE,  
6   phone TEXT NOT NULL UNIQUE  
7 );
```

On the right side, a 'History' panel shows a list of previous queries, with the current query highlighted. The timestamp '08:48:03' is visible at the bottom right of the history entry.

# Create Table



The screenshot shows the SQLiteOnline.com website interface for creating a table. The browser address bar displays `sqliteonline.com/syntax/create_table`. The top navigation bar includes icons for File, Owner DB, Run, Export, and Import. The left sidebar shows a tree view with 'SQLite' selected, containing a 'Table' section with 'contacts' and a 'Column' section with 'contact\_id INTEGER' circled in red. The main area displays the SQL code for creating the 'contacts' table:

```
1 CREATE TABLE contacts (  
2   contact_id INTEGER PRIMARY KEY,  
3   first_name TEXT NOT NULL,  
4   last_name TEXT NOT NULL,  
5   email TEXT NOT NULL UNIQUE,  
6   phone TEXT NOT NULL UNIQUE  
7 );
```

# Insert Table

- INSERT INTO statement is used to add new rows of data into a table.
- After creating the table, this command is used to insert data into the table.

## **Syntax1:**

**INSERT INTO** TABLE\_NAME [(column1, column2, column3,... columnN)] **VALUES** (value1, value2, value3,...valueN);

## **Syntax2:**

**INSERT INTO** TABLE\_NAME **VALUES** (value1,value2,value3,..valueN);

# Insert Table

- INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)  
VALUES (1, 'Paul', 32, 'California', 20000.00 );
- INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)  
VALUES (2, 'Allen', 25, 'Texas', 15000.00 );
- INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)  
VALUES (3, 'Teddy', 23, 'Norway', 20000.00 );
- INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)  
VALUES (4, 'Mark', 25, 'Rich-Mond ', 65000.00 );
- INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)  
VALUES (5, 'David', 27, 'Texas', 85000.00 );
- INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)  
VALUES (6, 'Kim', 22, 'South-Hall', 45000.00 );



# Insert Table

- **INSERT INTO STUDENT VALUES** (6, 'Kanchan', 21, 'Meerut', 10000.00 );

Command Prompt - sqlite3 Test.db

```
sqlite> .tables
STUDENT
sqlite> CREATE TABLE DEPARTMENT(
...>   ID INT PRIMARY KEY      NOT NULL,
...>   DEPT          CHAR(50) NOT NULL,
...>   EMP_ID        INT       NOT NULL
...> );
sqlite> INSERT INTO STUDENT (ID,NAME,AGE,ADDRESS,FEES)
...> VALUES (1, 'Ajeet', 27, 'Delhi', 20000.00);
sqlite> INSERT INTO STUDENT (ID,NAME,AGE,ADDRESS,FEES)
...> VALUES (2, 'Akash', 25, 'Patna', 15000.00 );
sqlite> INSERT INTO STUDENT (ID,NAME,AGE,ADDRESS,FEES)
...> VALUES (3, 'Mark', 23, 'USA', 2000.00 );
sqlite> INSERT INTO STUDENT (ID,NAME,AGE,ADDRESS,FEES)
...> VALUES (4, 'Chandan', 25, 'Banglore', 65000.00 );
sqlite> INSERT INTO STUDENT (ID,NAME,AGE,ADDRESS,FEES)
...> VALUES (5, 'Kunwar', 26, 'Agra', 25000.00 );
```



# Insert Table

- **SELECT \* FROM COMPANY;**

```
sqlite> SELECT * FROM COMPANY;  
1|Paul|32|California|20000.0  
2|Allen|25|Texas|15000.0  
3|Teddy|23|Norway|20000.0  
4|Mark|25|Rich-Mond |65000.0  
5|David|27|Texas|85000.0  
6|Kim|22|South-Hall|45000.0  
7|James|24|Houston|10000.0  
sqlite> _
```

# Insert Table

**INSERT INTO COMPANY VALUES (7, 'James', 24, 'Houston', 10000.00 );**

**ID NAME AGE ADDRESS SALARY**

-----

**1 Paul 32 California 20000.0**

**2 Allen 25 Texas 15000.0**

**3 Teddy 23 Norway 20000.0**

**4 Mark 25 Rich-Mond 65000.0**

**5 David 27 Texas 85000.0**

**6 Kim 22 South-Hall 45000.0**

**7 James 24 Houston 10000.0**

# Insert Table

sqliteonline.com

File Owner DB Run Export Import

SQLite

Table

- contacts
- demo

Column

- contact\_id INTEGER
- first\_name TEXT
- last\_name TEXT
- email TEXT
- phone TEXT

SQLite

```
1 INSERT INTO contacts VALUES(124, "Maria", "Thompson", "cbc@gmail.com", 9392222341);
2 SELECT * FROM contacts;
```

contact_id	first_name	last_name	email	phone
123	Alex	Thompson	abc@gmail.com	9394802341
124	Maria	Thompson	cbc@gmail.com	9392222341

# Update Table

- UPDATE query is used to modify the existing records in a table. It is used with WHERE clause to select the specific row otherwise all the rows would be updated.

## Syntax:

- **UPDATE** table\_name
- **SET** column1 = value1, column2 = value2..., columnN = valueN
- **WHERE** [condition];
-

# Update Table

```
sqlite> SELECT * FROM COMPANY;  
1|Paul|32|California|20000.0  
2|Allen|25|Texas|15000.0  
3|Teddy|23|Norway|20000.0  
4|Mark|25|Rich-Mond |65000.0  
5|David|27|Texas|85000.0  
6|Kim|22|South-Hall|45000.0  
7|James|24|Houston|10000.0  
sqlite> UPDATE COMPANY SET ADDRESS = 'Texas' WHERE ID = 6;  
sqlite> SELECT * FROM COMPANY;  
1|Paul|32|California|20000.0  
2|Allen|25|Texas|15000.0  
3|Teddy|23|Norway|20000.0  
4|Mark|25|Rich-Mond |65000.0  
5|David|27|Texas|85000.0
```

# Delete Table

- **DELETE** Query is used to delete the existing records from a table. You can use **WHERE** clause with **DELETE** query to delete the selected rows, otherwise all the records would be deleted.

**Syntax:**

**DELETE FROM table\_name**

**WHERE [condition];**

# Delete Table

```
sqlite> SELECT * FROM COMPANY;  
1|Paul|32|California|20000.0  
2|Allen|25|Texas|15000.0  
3|Teddy|23|Norway|20000.0  
4|Mark|25|Rich-Mond |65000.0  
5|David|27|Texas|85000.0  
6|Kim|22|Texas|45000.0  
7|James|24|Houston|10000.0  
sqlite> DELETE FROM COMPANY WHERE ID = 7;  
sqlite> SELECT * FROM COMPANY;  
1|Paul|32|California|20000.0  
2|Allen|25|Texas|15000.0  
3|Teddy|23|Norway|20000.0  
4|Mark|25|Rich-Mond |65000.0  
5|David|27|Texas|85000.0  
6|Kim|22|Texas|45000.0  
sqlite>
```

# Debugging

**Debug python scripts using PDB and IDE, Classify Errors, Develop Unit Tests , Create project Skeletons,**



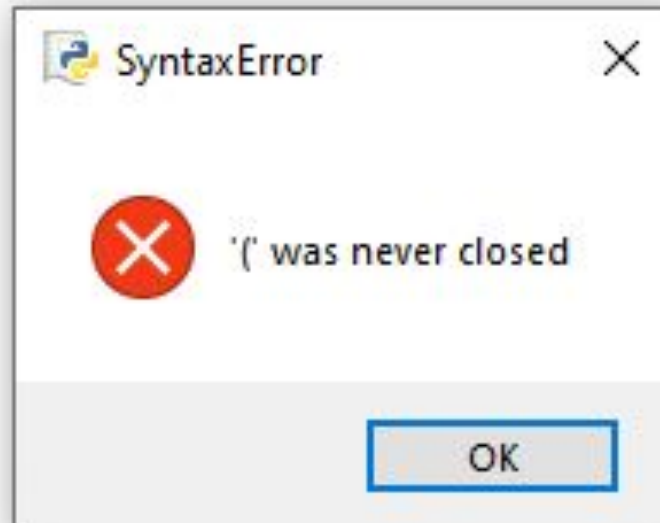
# Debugging

- Debugging a code is one of the most important yet tiresome tasks for any developer.
- When your code behaves weird, crashes, or just results in the wrong answers, that often means that your code contains a bug that resulted in those errors.

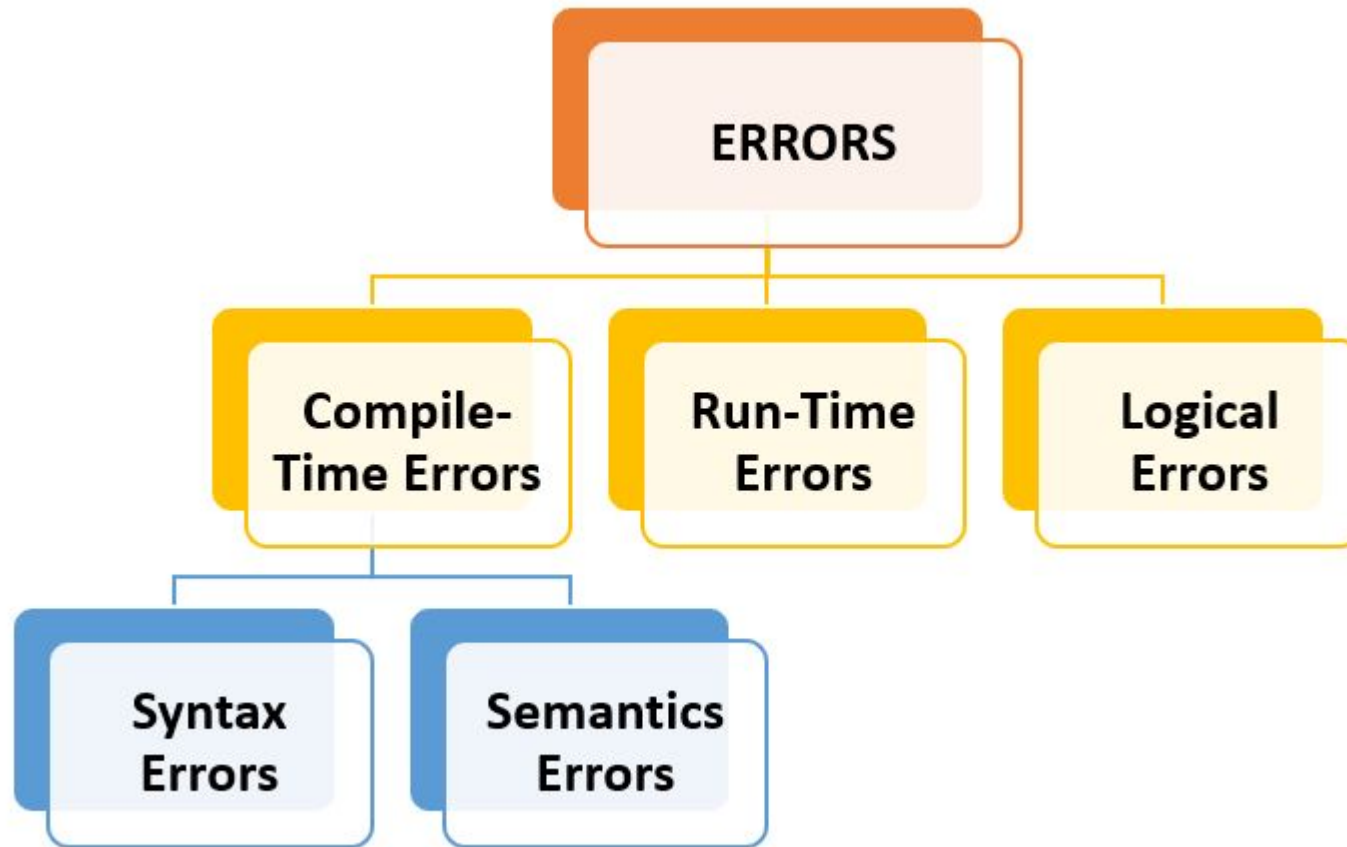
# Errors

- **Error** is some flaw in your code that stops the compilation of the code and doesn't allow the same code to run.

```
File Edit Format Run Options Window Help
x = float(input('Enter a number: '))
print (x)
```



# Classify Errors

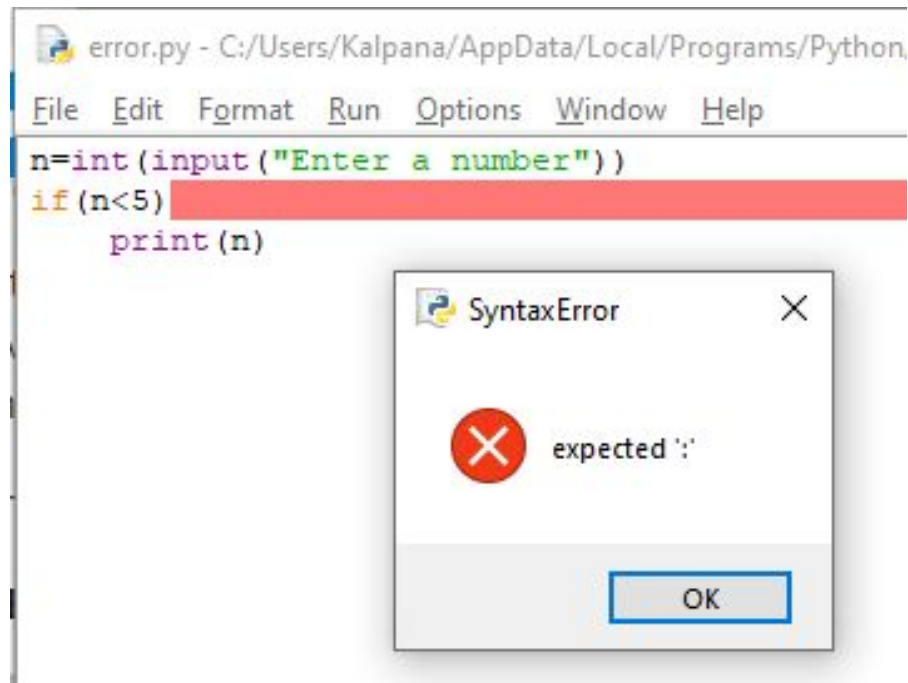


# Compile-time Errors

- The errors that occurs while compilation or at the compile-time are called compile-time errors.
- At the time of compilation, the system checks the source code. Now if there's any fault or violation of programming convention or you can say violation of language's rules then compilation stops and system gives us compilation error.
- There are 2 types of compile-time errors.
  - **Syntax Errors**
  - **Semantic Errors**

# Syntax Errors

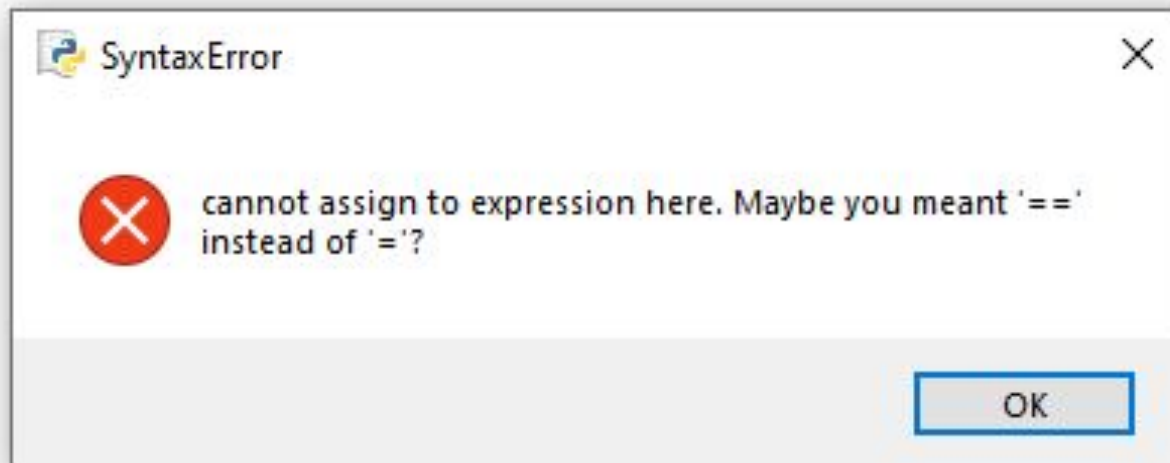
- Syntax refers to the formal rules those are used for writing valid statements in a programming language.
- You can understand it as a structure of the code and conventions that you have to follow to construct that structure.



# Semantic Errors

- Semantics refers to the meaning of a statement. So, when a statement doesn't make any sense and is not meaningful then we can say it is a Semantic error.

```
File Edit Format Run Options Window Help
n=int(input("Enter a number"))
sum = 0
sum + n = sum
```



# Run-time Errors

- This type of errors occurs after the compilation of the code; in-between when execution is going on. Due to when program is “crashed” or “abnormally abrupted”.
- These are also known as “Bugs” and are found during the process of debugging.
- For example:
  - **Infinite loop**
  - **Wrong value as Input**
  - **Invalid function call**
  - **Divide by Zero**

# Run-time Errors

## DivideByZero Error

```
1 my_value= 10/0
2 print(my_value)
```

## Output

```
Traceback (most recent call last):
  File "C:\Users\Dell\Desktop\test.py", line 1, in <module>
    my_value= 10/0
ZeroDivisionError: division by zero
```



# Logical Errors

- In many cases, programmers get demented and are not able to find the exact error. This is because errors are seen while compiling or at run-time. Sometimes programmer's analysis of the problem is wrong, in those cases the errors are logical errors.
- For example,
  - Incorrect implementation of an algorithm
  - Unmarked end of loop
  - Wrong parameters passed

# Logical Errors

error.py - C:/Users/Kalpana/AppData/Local/Programs/Python/Python310/error.py (3.10.7)

File Edit Format Run Options Window Help

```
x = float(input('Enter a number: '))
y = float(input('Enter a number: '))
z = x+y/2
print ('The result is:',z)
```

IDLE Shell 3.10.7

File Edit Shell Debug Options Window Help

```
== RESTART: C:/Users/Kalpana/AppData/Local/Programs/Python/Pythor
Enter a number: 2
Enter a number: 3
The result is: 3.5
>>> |
```

# Debugging

- The only way to discover where the bug in your code is and fix it to remove the error is debugging.
- Normally, use a bunch of print statements to track the execution of the code and to detect where the error occurs.
- **Python Standard Debugger (pdb)**
- **PyCharm**
- **Visual Studio Debugger**
- **Komodo**
- **Jupyter Visual Debugger**

# Python Standard Debugger(pdb)

- PDB is a **default debugger** that comes with all versions of Python, which means **no installation or hassle is needed**; you can just start using it if you already have any Python version installed on your machine.
- The pdb is a **command-line debugger** where you can insert breakpoints in your code and then run your code using the debugger mode.
- Using these breakpoints, you can inspect your code and the stack frames — it is very similar to using the print statement.
- Pdb is a very basic debugger, but various extensions can be added to make it more useful, such as rpdb and pdb++, which can make the debugging experience better ipdb if you're working with IPython.

# Python Standard Debugger(pdb)

- **Debugging in Python** is facilitated by **pdb module**(python debugger) which comes built-in to the Python standard library.
- It is actually defined as the class Pdb which internally makes use of bdb(basic debugger functions) and cmd(support for line-oriented command interpreters) modules.
- The major advantage of pdb is it runs **purely in the command line** thereby making it great for debugging code on remote servers when we don't have the privilege of a GUI-based debugger.
- pdb supports-
  - **Setting breakpoints**
  - **Stepping through code**
  - **Source code listing**
  - **Viewing stack traces**

# PDB module

- The PDB module in Python gives us gigantic highlights for compelling debugging of Python code. This incorporates:
  - Pausing of the program
  - Looking at the execution of each line of code
  - Checking the values of variables
- This module is already installed with installing of python. So, we only need to import it into our code to use its functionality.

# PDB module

- To import we simply use **import pdb** in our code.
- For debugging, we will use **pdb.set\_trace()** method. Now, in Python 3.7 **breakpoint()** method is also available for this.
- We run this on Python idle terminal (you can use any ide terminal to run).

# PDB module

- To **start debugging within the program** just insert `import pdb, pdb.set_trace()` commands.
- Run your script normally and execution will stop where we have introduced a breakpoint. So basically we are hard coding a breakpoint on a line below where we call `set_trace()`.
- With python 3.7 and later versions, there is a built-in function called **`breakpoint()`** which works in the same manner. Refer following example on how to insert `set_trace()` function.



# PDB module

```
import pdb
```

```
def addition(a, b):  
    answer = a + b  
    return answer
```

```
pdb.set_trace()  
x = input("Enter first number : ")  
y = input("Enter second number : ")  
sum = addition(x, y)  
print(sum)
```

# PDB module

```
import pdb
```

```
def addition(a, b):  
    answer = a + b  
    return answer
```

```
pdb.set_trace()  
x = input("Enter first number : ")  
y = input("Enter second number : ")  
sum = addition(x, y)  
print(sum)
```

\*IDLE Shell 3.10.7\*

File Edit Shell Debug Options Window Help

```
=== RESTART: C:/Users/Kalpana/AppData/Local/Programs/Python/Python310/add.py ===  
> c:\users\kalpana\appdata\local\programs\python\python310\add.py(10)<module>()  
-> x = input("Enter first number : ")  
(Pdb) n  
Enter first number : 22  
> c:\users\kalpana\appdata\local\programs\python\python310\add.py(11)<module>()  
-> y = input("Enter second number : ")  
(Pdb) n  
Enter second number : 23  
> c:\users\kalpana\appdata\local\programs\python\python310\add.py(12)<module>()  
-> sum = addition(x, y)  
(Pdb) n  
> c:\users\kalpana\appdata\local\programs\python\python310\add.py(13)<module>()  
-> print(sum)  
(Pdb) n  
2223  
--Return--
```

# PDB module

File Edit Format Run Options Window Help

```
import pdb
```

```
def addition(a, b):  
    answer = a + b  
    return answer
```

```
pdb.set_trace()  
x = input("Enter first number : ")  
y = input("Enter second number : ")  
sum = addition(x, y)  
print(sum)
```

\*IDLE Shell 3.10.7\*

File Edit Shell Debug Options Window Help

```
Python 3.10.7 (tags/v3.10.7:6cc6b13, Sep 5 2022, 14:08:36) [MSC v.1933 64 bit (AMD64)] on win32  
Type "help", "copyright", "credits" or "license()" for more information.
```

```
>>>
```

```
=== RESTART: C:/Users/Kalpna/AppData/Local/Programs/Python/Python310/add.py ===  
> c:\users\kalpana\appdata\local\programs\python\python310\add.py(10)<module>()  
-> x = input("Enter first number : ")  
(Pdb) n  
Enter first number : 22  
> c:\users\kalpana\appdata\local\programs\python\python310\add.py(11)<module>()  
-> y = input("Enter second number : ")  
(Pdb) n  
Enter second number : 23  
> c:\users\kalpana\appdata\local\programs\python\python310\add.py(12)<module>()  
-> sum = addition(x, y)  
(Pdb) n  
> c:\users\kalpana\appdata\local\programs\python\python310\add.py(13)<module>()
```

# PDB module

```
=== RESTART: C:/Users/Kalpana/AppData/Local/Programs/Python/Python310/add.py ===
> c:\users\kalpana\appdata\local\programs\python\python310\add.py(10)<module>()
-> x = input("Enter first number : ")
(Pdb) whatis x
*** NameError: name 'x' is not defined
(Pdb) n
Enter first number : 23
> c:\users\kalpana\appdata\local\programs\python\python310\add.py(11)<module>()
-> y = input("Enter second number : ")
(Pdb) whatis x
<class 'str'>
(Pdb) |
```

# PDB commands

Command	Function
---------	----------

help	To display all commands
------	-------------------------

where	Display the stack trace and line number of the current line
-------	---

next	Execute the current line and move to the next line ignoring function calls
------	--

step	Step into functions called at the current line
------	--

# PDB module

- **Post-mortem debugging** means entering debug mode after the program is finished with the execution process (failure has already occurred).
- pdb supports post-mortem debugging through the **pm()** and **post\_mortem()** functions. These functions look for active trace back and start the debugger at the line in the call stack where the exception occurred.

# PDB module

```
def multiply(a, b):  
    answer = a * b  
    return answer
```

```
x = input("Enter first number : ")  
y = input("Enter second number : ")  
result = multiply(x, y)  
print(result)
```

# PDB module

```
===== RESTART: C:/Users/Kalpana/AppData/Local/Programs/Python/Python310/mul.py =====
Enter first number : 2
Enter second number : 2
Traceback (most recent call last):
  File "C:/Users/Kalpana/AppData/Local/Programs/Python/Python310/mul.py", line 8, in <module>
    result = multiply(x, y)
  File "C:/Users/Kalpana/AppData/Local/Programs/Python/Python310/mul.py", line 2, in multiply
    answer = a * b
TypeError: can't multiply sequence by non-int of type 'str'

import pdb
pdb.pm()
> c:\users\kalpana\appdata\local\programs\python\python310\mul.py(2)multiply()
-> answer = a * b
```



# PDB module

- All the variables including variables local to the function being executed in the program as well as global are maintained on the stack.
- We can use **args**( or use **a**) to print all the arguments of function which is currently active.
- **p** command evaluates an expression given as an argument and prints the result.

Type "help", "copyright", "credits" or "license()" for more information.

>>>

=== RESTART: C:/Users/Kalpana/AppData/Local/Programs/Python/Python310/add

> c:\users\kalpana\appdata\local\programs\python\python310\add.py(10)<modu

-> x = input("Enter first number : ")

(Pdb) n

Enter first number : 2

> c:\users\kalpana\appdata\local\programs\python\python310\add.py(11)<modu

-> y = input("Enter second number : ")

(Pdb) n

Enter second number : 3

> c:\users\kalpana\appdata\local\programs\python\python310\add.py(12)<modu

-> sum = addition(x, y)

(Pdb) step

--Call--

> c:\users\kalpana\appdata\local\programs\python\python310\add.py(4) addit:

-> def addition(a, b):

(Pdb) args

a = '2'

b = '3'

(Pdb) next

> c:\users\kalpana\appdata\local\programs\python\python310\add.py(5) addit:

-> answer = a + b

(Pdb) p b

'3'

(Pdb) |

# PDB module

```
(Pdb) help
```

```
Documented commands (type help <topic>):
```

```
=====
```

EOF	c	d	h	list	q	rv	undisplay
a	cl	debug	help	ll	quit	s	unt
alias	clear	disable	ignore	longlist	r	source	until
args	commands	display	interact	n	restart	step	up
b	condition	down	j	next	return	tbreak	w
break	cont	enable	jump	p	retval	u	whatis
bt	continue	exit	l	pp	run	unalias	where

```
Miscellaneous help topics:
```

```
=====
```

```
exec pdb
```

```
(Pdb) |
```

# PDB module

## Stepping Through Code

There are two commands you can use to step through code when debugging:

Command	Description
n (next)	Continue execution until the next line in the current function is reached or it returns.
s (step)	Execute the current line and stop at the first possible occasion (either in a function that is called or in the current function).

# PDB module

```
def debugger(a, b):  
    breakpoint( )  
    result = a / b  
    return result  
  
print(debugger(5, 0))
```

# Features provided by PDB Debugging

1. Printing Variables or expressions
2. Moving in code by steps
3. Using Breakpoints
4. Execute code until the specified line
5. List the code
6. Displaying Expressions
7. Frames up-down



# Develop Unit Test

- Unit Testing is a technique in which particular module is tested to check by developer himself whether there are any errors. The primary focus of unit testing is test an individual unit of system to analyze, detect, and fix the errors.
- Python provides the unittest module to test the unit of source code. The unittest plays an essential role when we are writing the huge code, and it provides the facility to check whether the output is correct or not.

# Manual Testing

- Manual Testing has another form, which is known as exploratory testing. It is a testing which is done without any plan. To do the manual testing, we need to prepare a list of the application; we enter the different inputs and wait for the expected output.
- Every time we give the inputs or change the code, we need to go through every single feature of the list and check it.
- It is the most common way of testing and it is also time-consuming process.



# Unit Testing

Unit testing is a smaller test, it checks a single component that it is working in right way or not. Using the unit test, we can separate what necessities to be fixed in our system.



# unittest

- **OOP concepts supported by unittest framework**
  - **Test fixture**
  - **Test case**
  - **Test suite**
  - **Test runner**

# unittest

- **test fixture:**

A test fixture is used as a baseline for running tests to ensure that **there is a fixed environment** in which tests are run so that results are repeatable.

Examples :

- creating temporary databases.
- starting a server process.

- **test case:**

A test case is a **set of conditions** which is used to determine whether a system under test works correctly.

- **test suite:**

Test suite is a **collection of testcases** that are used to test a software program to show that it has some specified set of behaviours by executing the aggregated tests together.

- **test runner:**

A test runner is a component which **set up the execution of tests** and provides the outcome to the user.

# Unit Testing

- Python contains many test runners. The most popular build-in Python library is called **unittest**. The unittest is portable to the other frameworks.
- Consider the following three top most test runners.
  - unittest
  - nose Or nose2
  - pytest

# Unittest module

- The unittest is built into the Python standard library since 2.1.
- The best thing about the unittest, it comes with both a test framework and a test runner.
- There are few requirements of the unittest to write and execute the code.
- The code must be written using the classes and functions.
- The sequence of distinct assertion methods in the **TestCase** class apart from the built-in asserts statements.

# unittest

```
import unittest
```

```
class TestingSum(unittest.TestCase):
```

```
    def test_sum(self):
```

```
        self.assertEqual(sum([2, 3, 5]), 10, "It should be 10")
```

```
    def test_sum_tuple(self):
```

```
        self.assertEqual(sum((1, 3, 5)), 10, "It should be 10")
```

```
if __name__ == '__main__':
```

```
    unittest.main()
```

# unittest

```
>>>
```

```
= RESTART: C:/Users/Kalpana/AppData/Local/Programs/Python/Python310/TestingSum.py
```

```
.F
```

```
=====
```

```
FAIL: test_sum_tuple (__main__.TestingSum)
```

```
-----
```

```
Traceback (most recent call last):
```

```
  File "C:/Users/Kalpana/AppData/Local/Programs/Python/Python310/TestingSum.py", line 7, in test_sum_tuple
    self.assertEqual(sum((1, 3, 5)), 10, "It should be 10")
```

```
AssertionError: 9 != 10 : It should be 10
```

```
-----
```

```
Ran 2 tests in 0.408s
```

```
FAILED (failures=1)
```

```
>>>
```

# Unit Testing

- **assert** sum([ 2, 3, 5]) == 10, "Should be 10"
- 

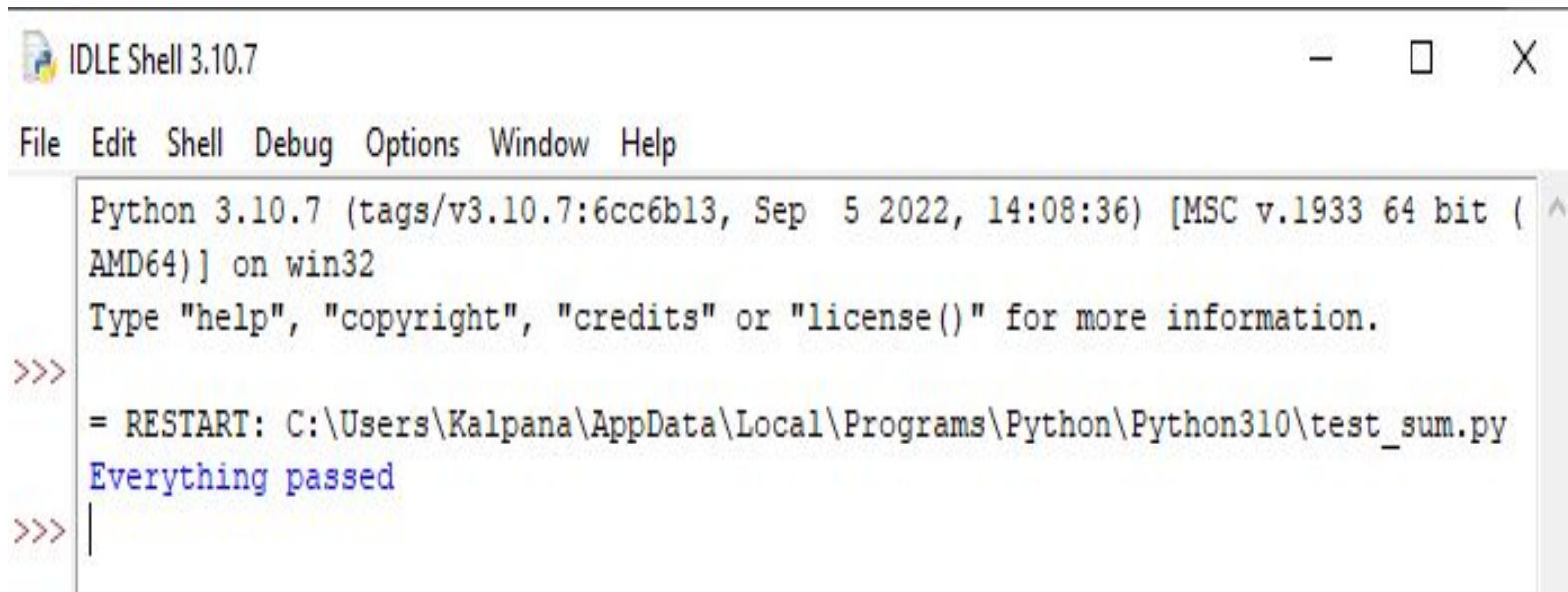
```
>>> assert sum([1, 3, 5]) == 10, "Should be 10"
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    assert sum([1, 3, 5]) == 10, "Should be 10"
AssertionError: Should be 10
>>> |
```



# Unit Testing

```
def test_sum():  
    assert sum([2, 3, 5]) == 10, "It should be 10"  
  
if __name__ == "__main__":  
    test_sum()  
    print("Everything passed")
```

# Unit Testing



```
Python 3.10.7 (tags/v3.10.7:6cc6b13, Sep 5 2022, 14:08:36) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:\Users\Kalpana\AppData\Local\Programs\Python\Python310\test_sum.py
Everything passed
>>> |
```

# Unit Testing

```
def test_sum2():  
    assert sum([2, 3, 5]) == 10, "It should be 10"
```

```
def test_sum_tuple():  
    assert sum((1, 3, 5)) == 10, "It should be 10"
```

```
if __name__ == "__main__":  
    test_sum2()  
    test_sum_tuple()  
    print("Everything is correct")
```

# Unit Testing

File Edit Shell Debug Options Window Help

```
Python 3.10.7 (tags/v3.10.7:6cc6b13, Sep 5 2022, 14:08:36) [MSC v.1933 64 bit (AMD64)] on win32
```

```
Type "help", "copyright", "credits" or "license()" for more information.
```

```
>>>
```

```
= RESTART: C:\Users\Kalpana\AppData\Local\Programs\Python\Python310\test_sum.py  
Everything passed
```

```
>>>
```

```
= RESTART: C:\Users\Kalpana\AppData\Local\Programs\Python\Python310\test_sum2.py  
Traceback (most recent call last):
```

```
  File "C:\Users\Kalpana\AppData\Local\Programs\Python\Python310\test_sum2.py",  
    line 9, in <module>
```

```
    test_sum_tuple()
```

```
  File "C:\Users\Kalpana\AppData\Local\Programs\Python\Python310\test_sum2.py",  
    line 5, in test_sum_tuple
```

```
    assert sum((1, 3, 5)) == 10, "It should be 10"
```

```
AssertionError: It should be 10
```

```
>>>
```

# Unit Testing

```
>>> = RESTART: C:\Users\Kalpana\AppData\Local\Programs\Python\Python310\test_sum2.py
Traceback (most recent call last):
  File "C:\Users\Kalpana\AppData\Local\Programs\Python\Python310\test_sum2.py",
line 9, in <module>
    test_sum_tuple()
  File "C:\Users\Kalpana\AppData\Local\Programs\Python\Python310\test_sum2.py",
line 5, in test_sum_tuple
    assert sum((1, 3, 5)) == 10, "It should be 10"
AssertionError: It should be 10

>>> = RESTART: C:/Users/Kalpana/AppData/Local/Programs/Python/Python310/unittest_demo.py
.
-----
Ran 1 test in 0.059s

OK
>>> |
```

# Project Skeleton in Python

- Skeleton code is a term that refers to a **project's basic layout** that does not include any data but is more than a blank template.
- By “structure,” we mean the decisions you make about how your project will best achieve its goal.
- We must evaluate how to make the most of Python's features in order to write clean, effective code.
- In practice, **“structure” entails writing clean code with clear logic and dependencies**, as well as organizing files and directories in the file system.

# Project Skeleton in Python

- **Requirements for Python**

- Packaging and inclusion of “modern” setup tools
- Capable of creating CLI entry points (for command-line tools; this should be able to be skipped when generating a library)
- Fixtures, `parametrize()`, and other pytest integration examples.
- From the start, the created project should be able to be built, tested, and have a good Python style.
- `README.md` contains instructions that are both legitimate and repeatable.
- `Setuptools_scm` allows for versioning based on version control. Instead of synchronizing a tag and a version string in the repo, this makes it simple to cut versions using tags.

# Project Directory in Python

- When you get the path of the root project structure, you'll get a string with the absolute path of the current project's root.
- Use the function **os.path.dirname(path)**, where the path is the path to any file in the project's top level.
- The outer folder, which contains all other project files, is at the top level of the project. This path can be found by calling **os.path.abspath(file)**, where a file is a file in the project's top level



# Project Directory in Python

```
import os  
ROOT_DIR =  
os.path.dirname(os.path.abspath("sample_doc.txt"))  
print(ROOT_DIR)
```

```
>>> = RESTART: C:/Users/Kalpana/AppData/Local/Programs/Python/Python310/pythondir.py  
C:\Users\Kalpana\AppData\Local\Programs\Python\Python310
```

**Implement Regular Expression and its Basic Functions - findall( ),search( ),split( ),sub( )**

# Regular Expressions (RegEx)

- A **Regular Expressions (RegEx)** is a special sequence of characters that uses a search pattern to find a string or set of strings.
- It can detect the presence or absence of a text by matching it with a particular pattern, and also can split a pattern into one or more sub-patterns.

# RegEx Module

- Python has a built-in package called re, which can be used to work with Regular Expressions.
- Import the re module:

```
import re
```

# RegEx Module

```
import re
```

```
#Check if the string starts with "The" and ends with "Spain":
```

```
txt = "The rain in Spain"
```

```
x = re.search("^The.*Spain$", txt)
```

```
if x:
```

```
    print("YES! We have a match!")
```

```
else:
```

```
    print("No match")
```

# RegEx Functions

Function	Description
<u><a href="#">findall</a></u>	Returns a list containing all matches
<u><a href="#">search</a></u>	Returns a <u><a href="#">Match object</a></u> if there is a match anywhere in the string
<u><a href="#">split</a></u>	Returns a list where the string has been split at each match

# findall()

- The `findall()` function returns a list containing all matches.

```
import re
```

```
txt = "The rain in Spain"  
x = re.findall("ai", txt)  
print(x)
```

**Output:**

```
['ai', 'ai']
```

# findall()

```
import re
```

```
txt = "The rain in Spain"  
x = re.findall("Portugal", txt)  
print(x)
```

**Output:**

```
[]
```



# search()

- The `search()` function searches the string for a match, and returns a Match object if there is a match.

```
import re
```

```
txt = "The rain in Spain"  
x = re.search("\s", txt)
```

```
print("The first white-space character is located in position:",  
x.start())
```

## **Output:**

The first white-space character is located in position: 3

# search()

- The search() function searches the string for a match, and returns a Match object if there is a match.

```
import re
txt = "The rain in Spain"
x = re.search("Portugal", txt)
print(x)
```

## **Output:**

Portugal found in: None

# split()

- The **split()** function returns a list where the string has been split at each match:

```
import re
```

```
txt = "The rain in Spain"  
x = re.split("\s", txt)  
print(x)
```

Output

```
['The', 'rain', 'in', 'Spain']
```

# split()

- Split the string only at the first occurrence:

```
import re
```

```
txt = "The rain in Spain"  
x = re.split("\s", txt, 1)  
print(x)
```

Output

```
['The', 'rain in Spain']
```

# sub()

- The sub() function replaces the matches with the text of your choice:

```
import re
```

```
txt = "The rain in Spain"  
x = re.sub("\s", "9", txt)  
print(x)
```

## Output

- The9rain9in9Spain

# sub()

- Replace the first 2 occurrences:

```
import re
```

```
txt = "The rain in Spain"  
x = re.sub("\s", "9", txt, 2)  
print(x)
```

**Output:**

The9rain9in Spain

# Match object

- A Match Object is an object containing information about the search and the result.
- The Match object has properties and methods used to retrieve information about the search, and the result:
  - `span()` returns a tuple containing the start-, and end positions of the match.
  - `string` returns the string passed into the function
  - `group()` returns the part of the string where there was a match

# Match object

- `import re`

```
txt = "The rain in Spain"  
x = re.search("ai", txt)  
print(x) #this will print an object
```



# Match object

- Print the position (start- and end-position) of the first match occurrence.

#The regular expression looks for any words that starts with an upper case "S"

```
import re
```

```
txt = "The rain in Spain"  
x = re.search(r"\bS\w+", txt)  
print(x.span())
```

Output: (12,17)

# Match object

- Print the string passed into the function:

```
import re
```

```
txt = "The rain in Spain"  
x = re.search(r"\bS\w+", txt)  
print(x.string)
```

Output : The rain in Spain

# Match object

- Print the part of the string where there was a match.
- The regular expression looks for any words that starts with an upper case "S"

```
import re
#Search for an upper case "S" character in the beginning of a
word, and print the word:
txt = "The rain in Spain"
x = re.search(r"\bS\w+", txt)
print(x.group())
```

Output : Spain

**Use Classes, Objects, and Attributes,  
Develop applications based on Object  
Oriented Programming and Methods**



# Class

- A class is a **user-defined blueprint** or prototype from which objects are created.
- Classes provide a means of **bundling data and functionality** together.
- Creating a new class creates a **new type of object**, allowing new instances of that type to be made.
- Each class instance can have **attributes** attached to it for maintaining its state.
- Class instances can also have **methods** (defined by their class) for modifying their state.

# Class

- Class creates a **user-defined data structure**, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class.
- A class is like a **blueprint for an object**.

## Syntax:

#Class Definition

```
class ClassName:
```

```
    # Statement
```

## Example:

```
class MyClass:
```

```
    x = 15
```

# Class

- **Note:**
- Classes are created by keyword **class**.
- Attributes are the **variables** that belong to a class.
- Attributes are always **public** and can be accessed using the **dot (.)** operator.  
Eg.: Myclass.Myattribute



# Object

- An Object is an **instance of a Class**.
- A class is like a blueprint while an instance is a **copy of the class with *actual values***.
- It's not an idea anymore, it's an actual dog, like a dog of breed pug who's seven years old. You can have many dogs to create many different instances, but without the class as a guide, you would be lost, not knowing what information is required.

# Characteristics of Object

**A**

## State

Represents the data of an object.

## Behavior

represents the behavior of an object such as deposit, withdraw, etc.

**B**

**C**

## Identity

It is used internally by the JVM to identify each object uniquely.

# Object

An object consists of :

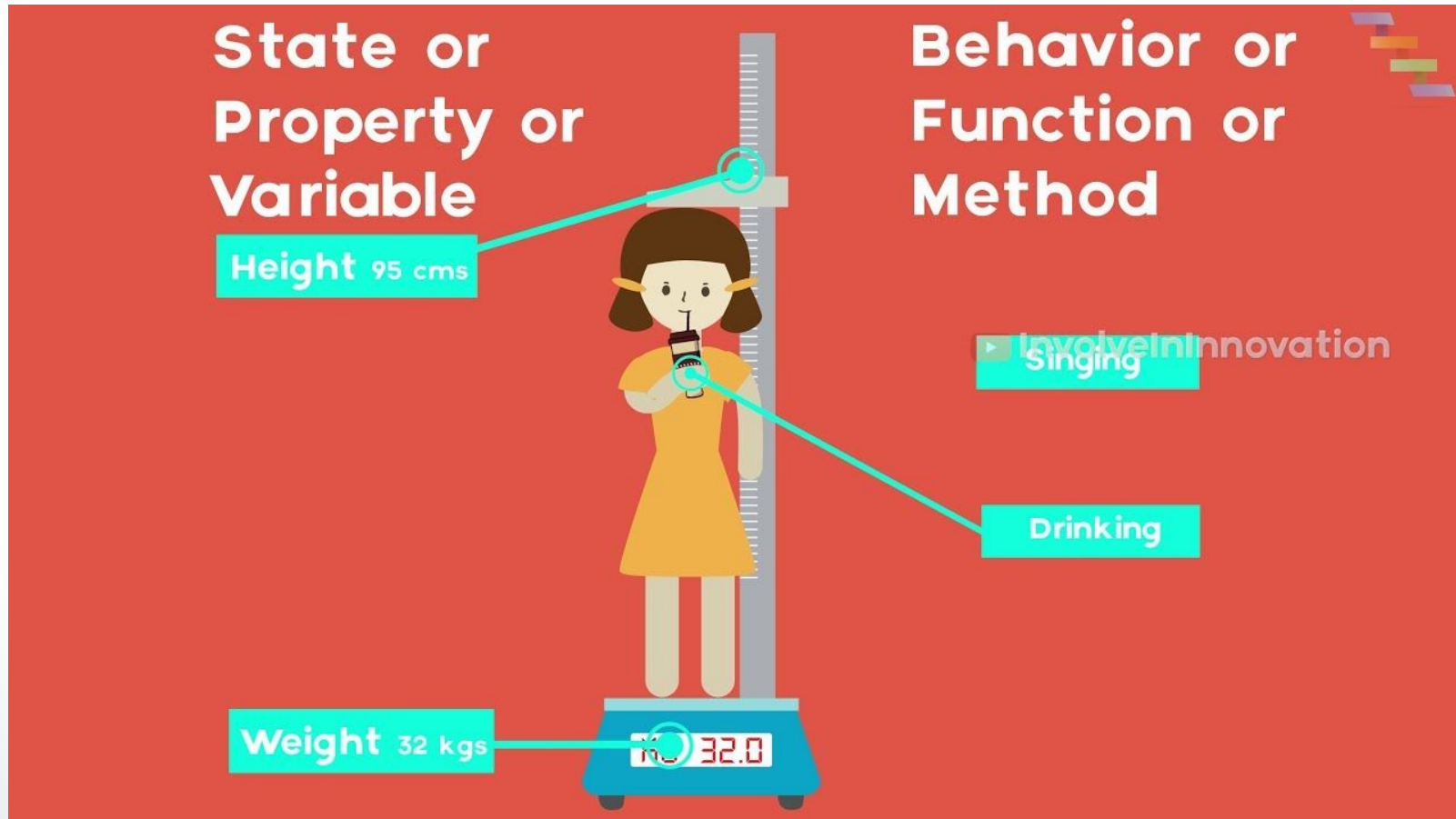
- **State:** It is represented by the **attributes** of an object. It also reflects the properties of an object.
- **Behaviour:** It is represented by the **methods** of an object. It also reflects the response of an object to other objects.
- **Identity:** It gives a **unique name to an object** and enables one object to interact with other objects.

# Object

## Examples of Objects

Object	Identity	Behaviour	State
A person.	'Hussain Pervez.'	Speak, walk, read.	Studying, resting, qualified.
A shirt.	My favourite button white denim shirt.	Shrink, stain, rip.	Pressed, dirty, worn.
A sale.	Sale no #0015, 16/06/02.	Earn loyalty points.	Invoiced, cancelled.
A bottle of ketchup.	<i>This</i> bottle of ketchup.	Spill in transit.	Unsold, opened, empty.

# Object

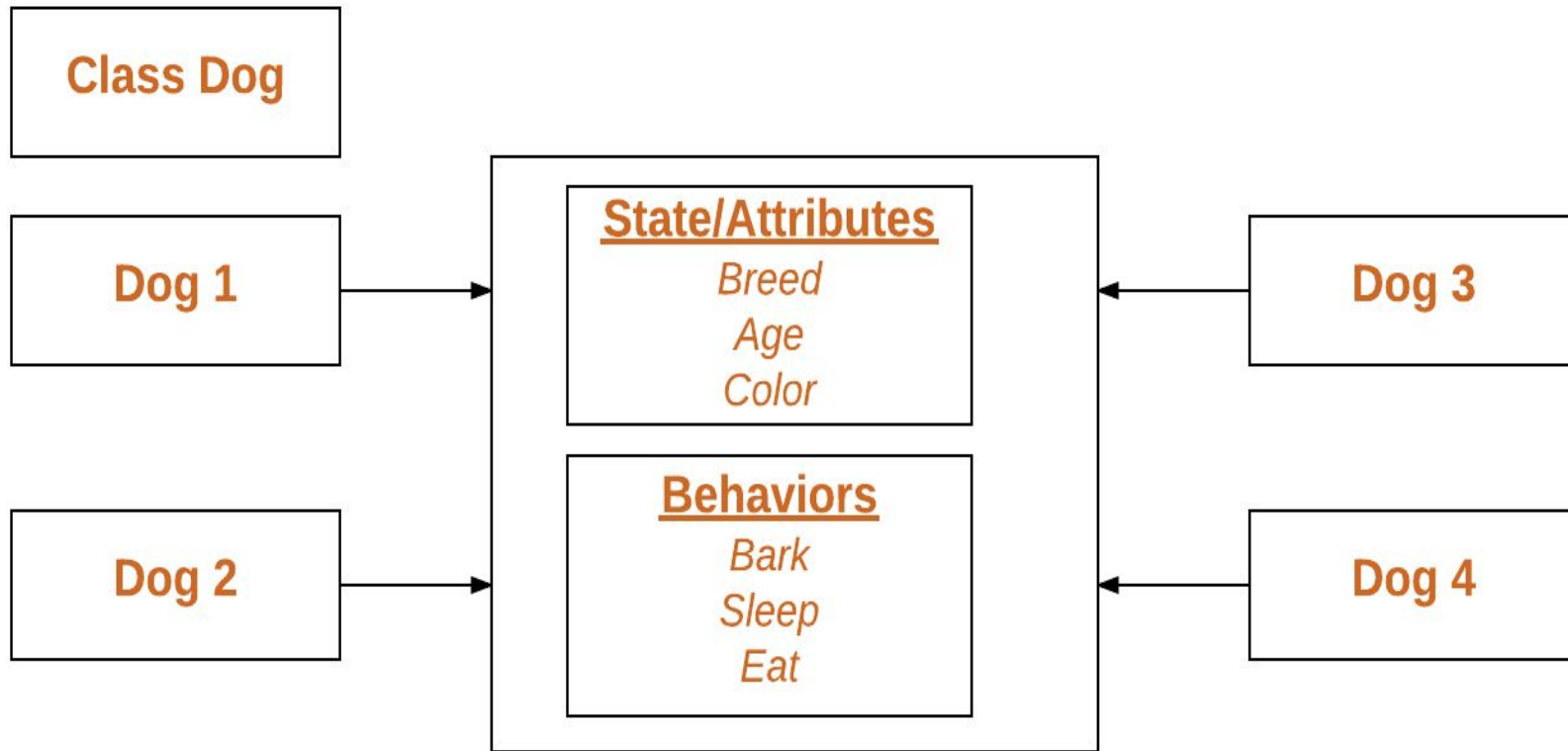


# Object

## Declaring Objects (Also called instantiating a class)

- When an object of a class is created, the class is said to be instantiated.
- All the **instances share the attributes and the behavior of the class**. But the values of those attributes, i.e. the state are unique for each object.
- A single class may have **any number of instances**.

# Object



# Object

## **Syntax:**

```
obj = ClassName()  
print(obj.attr)
```

## **Example:**

```
p1 = MyClass()  
print(p1.x)
```



# Class and objects

```
class MyClass:
```

```
    x = 15
```

```
p1 = MyClass()
```

```
print(p1.x)
```

**Output:**

**15**

# Class and objects

# Python3 program to demonstrate instantiating a class

class Dog:

attr1 = "mammal"

attr2 = "dog"

Rodger = Dog()

print(Rodger.attr1)

# The self

- Class methods must have an extra first parameter in the method definition. We do not give a value for this parameter when we call the method, Python provides it.
- If we have a method that takes no arguments, we still have one argument.
- This is similar to this pointer in C++ and this reference in Java.
- When we call a method of this object as `myobject.method(arg1, arg2)`, this is automatically converted by Python into `MyClass.method(myobject, arg1, arg2)` – this is all the special self is about.

# The `__init__`

- The `__init__` method is similar to constructors in C++ and Java.
- Constructors are used to **initialize the object's state**.
- Like methods, a constructor also contains a collection of statements(i.e. instructions) that are executed at the time of Object creation.
- It runs as soon as an object of a class is instantiated.
- The method is useful **to do any initialization** you want to do with your object.

# Attributes

- A **class attribute** is a variable that belongs to a certain class, and not a particular object. Every instance of this class shares the *same* variable. These attributes are usually defined outside the `__init__` constructor.
- An **instance/object attribute** is a variable that belongs to one (*and only one*) object. Every instance of a class points to its own attributes variables. These attributes are defined within the `__init__` constructor.

# Attributes

```
class Dog:
    dogs_count = 0
    def __init__(self, name, age):
        self.name = name
        self.age = age
        print("Welcome to this world {}".format(self.name))
        Dog.dogs_count += 1
    def __del__(self):
        print("Goodbye {}".format(self.name))
        Dog.dogs_count -= 1

a = Dog("Max", 1)
print("Number of dogs: {}".format(Dog.dogs_count))
b = Dog("Charlie", 7)
del a
c = Dog("Spot", 4.5)
print("Number of dogs: {}".format(Dog.dogs_count))
del b
del c
print("Number of dogs: {}".format(Dog.dogs_count))
```

# Attributes

## ***Output:***

Welcome to this world Max!

Number of dogs: 1

Welcome to this world Charlie!

Goodbye Max :(

Welcome to this world Spot!

Number of dogs: 2

Goodbye Charlie :(

Goodbye Spot :(

Number of dogs: 0

# Attributes

 initdel.py - C:/Users/Kalpana/AppData/Local/Programs/Python/Python310/initdel.py (3.10.7)

File Edit Format Run Options Window Help

```
class Dog:
    dogs_count = 0
    def __init__(self, name, age):
        self.name = name
        self.age = age
        print("Welcome to this world {}".format(self.name))
        Dog.dogs_count += 1
    def __del__(self):
        print("Goodbye {} :(".format(self.name))
        Dog.dogs_count -= 1

a = Dog("Max", 1)
print("Number of dogs: {}".format(Dog.dogs_count))
b = Dog("Charlie", 7)
del a
c = Dog("Spot", 4.5)
print("Number of dogs: {}".format(Dog.dogs_count))
del b
del c
print("Number of dogs: {}".format(Dog.dogs_count))
```



# Attributes

```
>>>
= RESTART: C:/Users/Kalpana/AppData/Local/Programs/Python/Python310/initdel.py =
Welcome to this world Max!
Number of dogs: 1
Welcome to this world Charlie!
Goodbye Max :(
Welcome to this world Spot!
Number of dogs: 2
Goodbye Charlie :(
Goodbye Spot :(
Number of dogs: 0
>>> |
```

# Class and objects

```
class Person:
```

```
    def __init__(self, name):  
        self.name = name
```

```
    def say_hi(self):  
        print('Hello, my name is', self.name)
```

```
p = Person('Nikhil')  
p.say_hi()
```

Output: Hello, my name is Nikhil

# Class and Instance Variables

- **Instance variables** - **data, unique to each instance** and **class variables** - **attributes and methods shared by all instances of the class.**
- **Instance variables** are **variables whose value is assigned inside a constructor** or method with self whereas **class variables** are **variables whose value is assigned in the class.**

# Class and Instance Variables

```
class Dog:

    animal = 'dog'

    def __init__(self, breed, color):

        self.breed = breed
        self.color = color

Rodger = Dog("Pug", "brown")
Buzo = Dog("Bulldog", "black")
```

```
print('Rodger details:')
print('Rodger is a',
      Rodger.animal)
print('Breed: ', Rodger.breed)
print('Color: ', Rodger.color)

print('\nBuzo details:')
print('Buzo is a', Buzo.animal)
print('Breed: ', Buzo.breed)
print('Color: ', Buzo.color)

print("\nAccessing class
variable using class name")
print(Dog.animal)
```

# Class and Instance

## Variables

**Rodger details:**

**Rodger is a dog**

**Breed: Pug**

**Color: brown**

**Buzo details:**

**Buzo is a dog**

**Breed: Bulldog**

**Color: black**

**Accessing class variable using class name dog**

# Defining instance variables using the normal method

```
class Dog:

    # Class Variable
    animal = 'dog'

    # The init method or
    # constructor
    def __init__(self, breed):

        # Instance Variable
        self.breed = breed

    # Adds an instance variable
    def setColor(self, color):
        self.color = color

    # Retrieves instance variable
    def getColor(self):
        return self.color

# Driver Code
Rodger = Dog("pug")
Rodger.setColor("brown")
print(Rodger.getColor())
```

Output: brown

**JSON file – Read, Write and Parse JSON file -  
JSON Conversion – to dictionary, to JSON, to  
JSON String, JSON schema – Schema  
Validation, Resolving JSON Reference,  
Extending Validator Classes - Virtual  
Environment, Floating point Arithmetic –  
Issues and Limitations**



- JSON stands for **J**ava**S**cript **O**bject **N**otation
- JSON is a **text format** for storing and transporting data
- JSON is "self-describing" and easy to understand

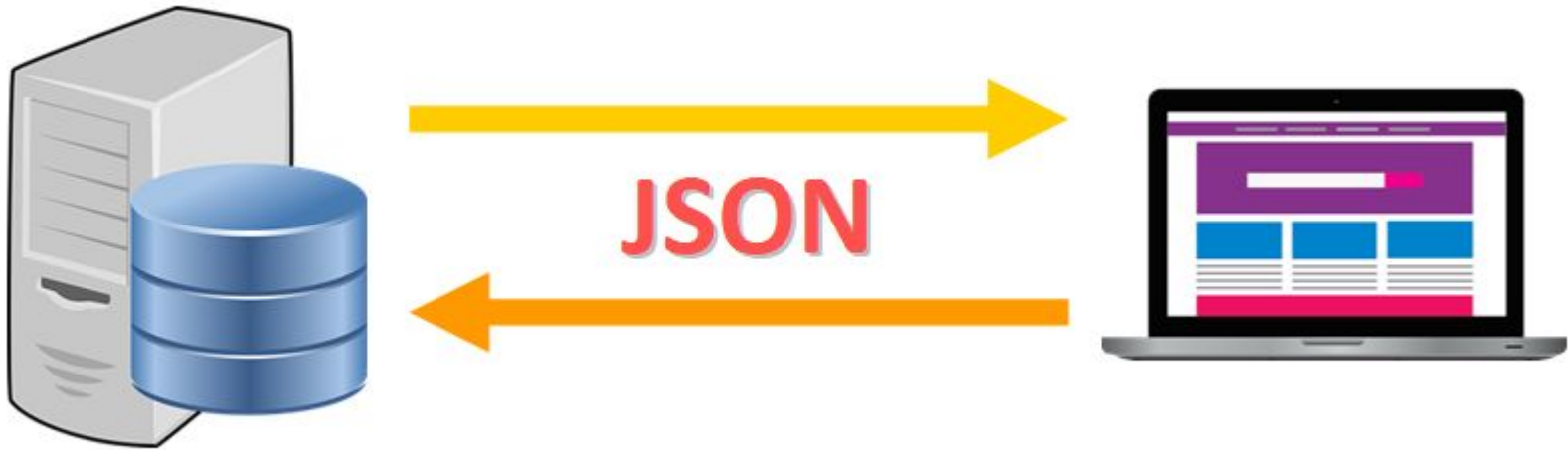


# Why JSON?

- The JSON format is syntactically similar to the code for **creating JavaScript objects**. Because of this, a JavaScript program can easily convert JSON data into JavaScript objects.
- Since the **format is text only**, JSON data can easily be sent between computers, and used by any programming language.

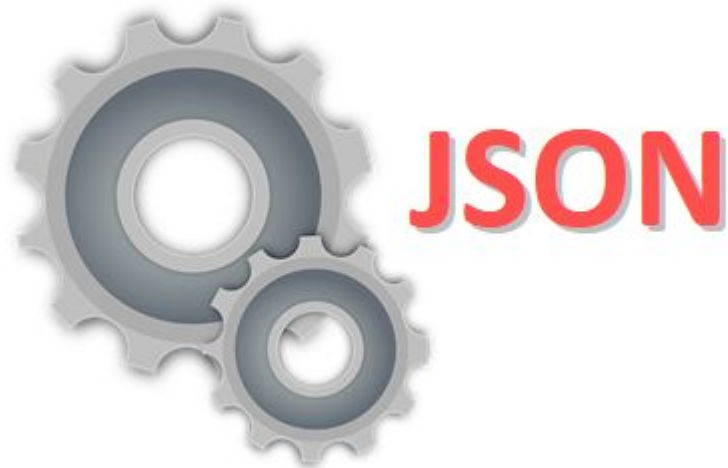
# JSON

- **Web Development:** JSON is commonly used to send data from the server to the client and vice versa in web applications.



# JSON

- **Configuration files:** JSON is also used to store configurations and settings.
- For example, to create a [Google Chrome App](#), you need to include a JSON file called **manifest.json** to specify the name of the app, its description, current version, and other properties and settings.



# JSON

```
{  
  "size": "medium",  
  "price": 15.67,  
  "toppings": ["mushrooms", "pepperoni", "basil"],  
  "extra_cheese": false,  
  "delivery": true,  
  "client": { "name": "Jane Doe", "phone": null, "email":  
    "janedoe@email.com" }  
}
```

# Characteristics of JSON format

- There is a sequence of key-value pairs surrounded by curly brackets {}.
- Each key is mapped to a particular value using this format:
  - "key": <value>
- **Tip:** The values that require quotes have to be surrounded by double quotes.
- Key-value pairs are separated by a comma. Only the last pair is not followed by a comma.

# Datatypes : Keys and Values

- JSON files have specific rules that determine which data types are valid for keys and values.
- **Keys** must be strings.
- **Values** can be either a string, a number, an array, a boolean value (true/ false), null, or a JSON object.
- **Note:**
- Keys in key/value pairs of JSON are always of the type str.  
When a dictionary is converted into JSON, all the keys of the dictionary are coerced to strings.

# Style Guide

- According to the [Google JSON Style Guide](#):
- Always choose meaningful names.
- Array types should have plural key names. All other key names should be singular. For example:  
use "orders" instead of "order" if the corresponding value is an array.
- There should be no comments in JSON objects.

# JSON

- `'{"name":"John", "age":30, "car":null}'`
- It defines an object with 3 properties:
  - **name**
  - **age**
  - **car**
- Each property has a value.
- If you parse the JSON string with a JavaScript program, you can access the data as an object:  
**let personName = obj.name;**  
**let personAge = obj.age;**

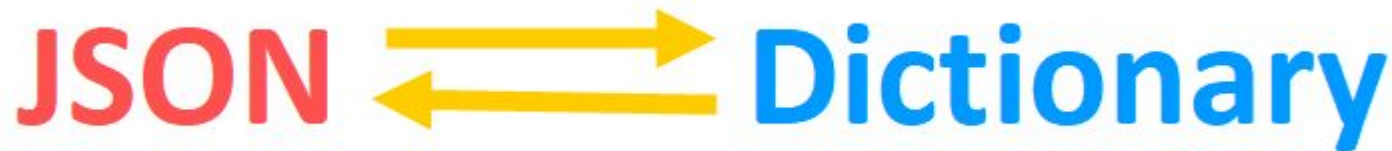


# JSON

- JSON stands for JavaScript **O**bject **N**otation
- JSON is a lightweight data-interchange format
- JSON is plain text written in JavaScript object notation
- JSON is used to send data between computers
- JSON is language independent \*
- The JSON format was originally specified by [Douglas Crockford](#).
-

# JSON vs. Python Dictionaries

- JSON and Dictionaries might look very similar at first (visually), but they are quite different.
- JSON is a file format used to represent and store data whereas a Python Dictionary is the actual data structure (object) that is kept in memory while a Python program runs.
- **How JSON and Python Dictionaries Work Together**



# JSON vs. Python Dictionaries

- When we work with JSON files in Python, we can't just read them and use the data in our program directly. This is because the entire file would be represented as a single string and we would not be able to access the key-value pairs individually.
- We use the key-value pairs of the JSON file to create a Python dictionary that we can use in our program to read the data, use it, and modify it (if needed).
- This is the main connection between JSON and Python Dictionaries.
- JSON is the string representation of the data and dictionaries are the actual data structures in memory that are created when the program runs.

# JSON String

- It is just a regular multi-line Python string that follows the JSON format.

```
data_JSON = """  
{  
    "size": "Medium",  
    "price": 15.67,  
    "toppings": ["Mushrooms", "Extra Cheese", "Pepperoni",  
        "Basil"],  
    "client": { "name": "Jane Doe", "phone": "455-344-234",  
        "email": "janedoe@email.com" }  
} """
```

- To define a multi-line string in Python, we use triple quotes.
- Then, we assign the string to the variable data\_JSON.

# The JSON Module

- Python comes with a built-in module called **json**. It is installed automatically when you install Python and it includes functions to help you work with JSON files and strings.
- **How to Import the JSON Module**
- To use json in our program, we just need to write an import statement at the top of the file.

```
import json
```

# The JSON Module

- **How to Import the JSON Module**
- To use json in our program, we just need to write an import statement at the top of the file.

```
import json
```



# Read, Write and Parse JSON

- JSON is a lightweight data format for data interchange which can be easily read and written by humans, easily parsed and generated by machines.
- It is a complete language-independent text format.
- To work with JSON data, Python has a built-in package called json.

# Read, Write and Parse JSON

- **Example:** `s = '{"id":01, "name": "Emily", "language": ["C++", "Python"]}'`
- **Name/Value pairs:** Represents Data, name is followed by ':' (colon) and the Name/Value pairs separated by, (comma).
- **Curly braces:** Holds objects.
- **Square brackets:** Hold arrays with values separated by, (comma).



# Read, Write and Parse JSON

- Keys/Name must be strings with double quotes and values must be data types amongst the following:
  - **String**
  - **Number**
  - **Object (JSON object)**
  - **array**
  - **Boolean**
  - **Null**

# Read, Write and Parse JSON

- Example:

```
{  
  "employee": [  
    {  
      "id": "01",  
      "name": "Amit",  
      "department": "Sales"  
    },  
    {  
      "id": "04",  
      "name": "sunil",  
      "department": "HR"  
    }  
  ]  
}
```

# Read, Write and Parse JSON

## Parse JSON (Convert from JSON to Python)

- `json.loads()` method can parse a json string and the result will be a Python dictionary.
- **Syntax:**  
**`json.loads(json_string)`**

# Read, Write and Parse JSON

- **JSON String to Python Dictionary**
- To do this, we will use the `loads()` function of the `json` module, passing the string as the argument.

```
json.loads(<JSON_string>)
```

Module

Function

String with JSON format

# Read, Write and Parse JSON

# Python program to convert JSON to Python

**import json**

# JSON string

**employee = '{"id": "09", "name": "Nitin",  
"department": "Finance"}'**

# Convert JSON string to Python dict

**employee\_dict = json.loads(employee)  
print(employee\_dict)**

**print(employee\_dict['name'])**

# Read, Write and Parse JSON

```
# Import the module
import json
# String with JSON format
data_JSON = """
{
    "size": "Medium",
    "price": 15.67,
    "toppings": ["Mushrooms", "Extra Cheese", "Pepperoni", "Basil"],
    "client": {
        "name": "Jane Doe",
        "phone": "455-344-234",
        "email": "janedoe@email.com"
    }
} """
# Convert JSON string to dictionary
data_dict = json.loads(data_JSON)
```

# Read, Write and Parse JSON

**Output:**

```
{'size': 'Medium', 'price': 15.67, 'toppings':  
['Mushrooms', 'Extra Cheese', 'Pepperoni', 'Basil'],  
'client': {'name': 'Jane Doe', 'phone': '455-344-234',  
'email': 'janedoe@email.com'}}
```

# Read, Write and Parse JSON

```
print(data_dict["size"])  
print(data_dict["price"])  
print(data_dict["toppings"])  
print(data_dict["client"])
```

## Output

Medium

15.67

['Mushrooms', 'Extra Cheese', 'Pepperoni', 'Basil']

{'name': 'Jane Doe', 'phone': '455-344-234', 'email':  
'janedoe@email.com'}



# Read, Write and Parse JSON

- **Python read JSON file**

`json.load()` method can read a file which contains a JSON object.

## **Syntax:**

- **`json.load(file_object)`**

# Read, Write and Parse JSON

- **Python read JSON file**
- If we want to read this file in Python, we just need to use a with statement:

Open orders.json in read mode

File Object

```
with open("orders.json") as file:  
    data = json.load(file)
```

Read the JSON file and create a dictionary

# Read, Write and Parse JSON

- `json.load(file)` creates and returns a new Python dictionary with the key-value pairs in the JSON file.
- Then, this dictionary is assigned to the data variable.
- **Tip:** Notice that we are using **`load()`** instead of **`loads()`**. This is a different function in the **`json`** module.

# Read, Write and Parse JSON

Consider a file named a.json which contains a JSON object.

```
import json
```

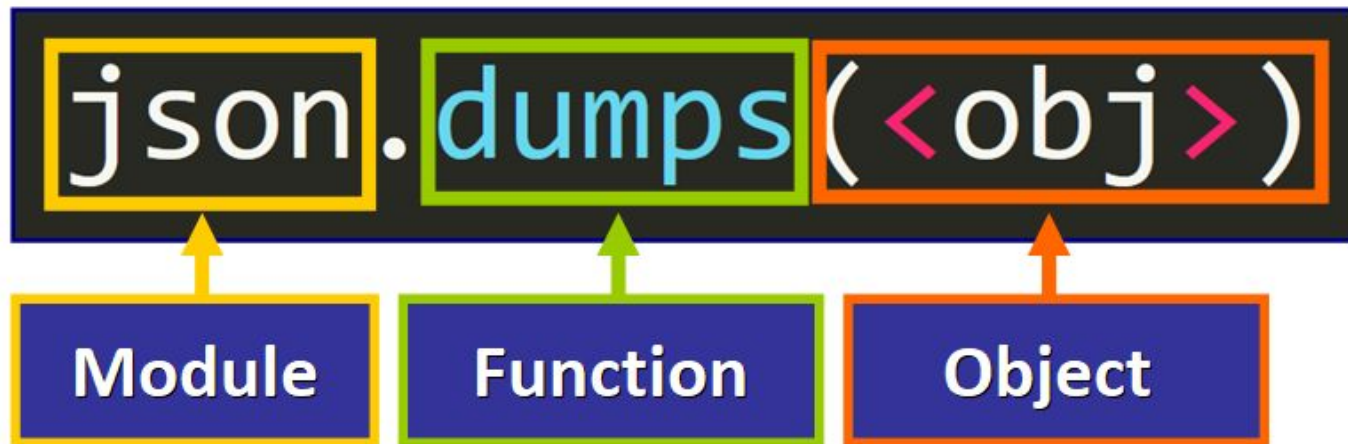
```
with open('C:/Users/Kalpana/Desktop/a.json', 'r') as f:  
    data = json.load(f)
```

```
print(data)
```

# Read, Write and Parse JSON

- To do that, we can use the dumps function of the json module, passing the object as argument:

- 



**Tip:** This function will return a string.

# Read, Write and Parse JSON

- **Python Convert to JSON string**
- You can convert a dictionary to JSON string using `json.dumps()` method.

```
import json
```

```
person_dict = {'name': 'Bob', 'age': 12, 'children':  
None }
```

```
person_json = json.dumps(person_dict)
```

```
# Output: {"name": "Bob", "age": 12, "children": null}
```

```
print(person_json)
```

# Read, Write and Parse JSON

## **JSON to Python: Type Conversion**

When you use `loads()` to create a Python dictionary from a JSON string, you will notice that some values will be converted into their corresponding Python values and data types.

# Read, Write and Parse JSON

A process of type conversion occurs as well when we convert a dictionary into a JSON string.

<b>Python</b>	<b>JSON Equivalent</b>
dict	object
list, tuple	array
str	string
int, float, int	number
True	true
False	false
None	null



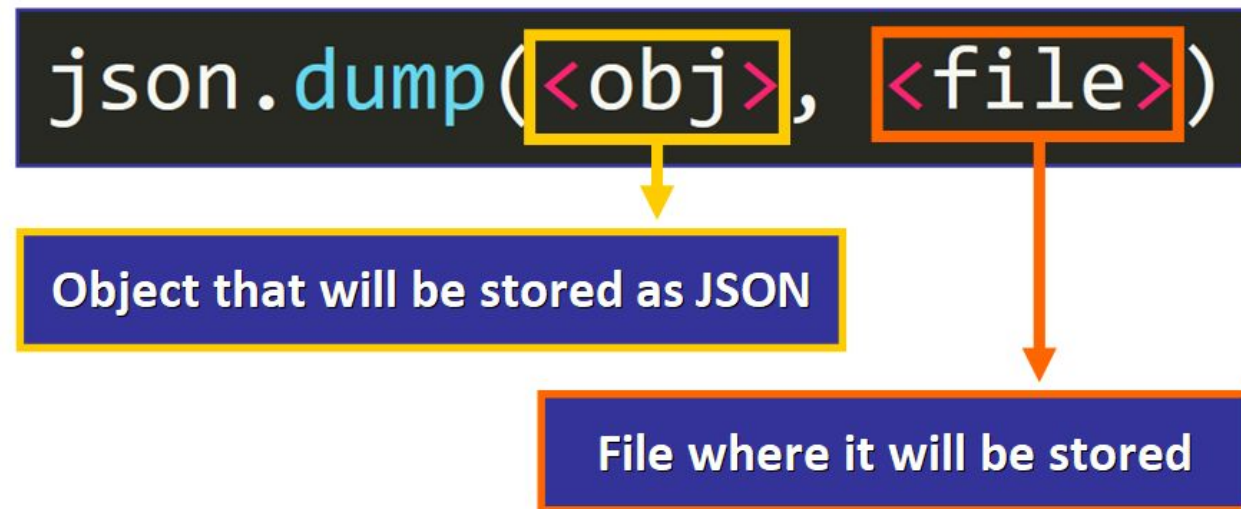
# Read, Write and Parse JSON

- **Tip:** If the file doesn't exist already in the current working directory (folder), it will be created automatically. By using the '**w**' mode, we will be replacing the entire content of the file if it already exists.
- There are two alternative ways to write to a JSON file in the body of the with statement:
  - **dump**
  - **dumps**

# Read, Write and Parse JSON

## First Approach: dump

- This is a function that takes two arguments:
- The object that will be stored in JSON format (for example, a dictionary).
- The file where it will be stored (a file object).



# Read, Write and Parse JSON

- **# Open the orders.json file with open("orders.json") as file: # Load its content and make a new dictionary data = json.load(file) # Delete the "client" key-value pair from each order for order in data["orders"]: del order["client"] # Open (or create) an orders\_new.json file # and store the new version of the data. with open("orders\_new.json", 'w') as file: json.dump(data, file)**

# Read, Write and Parse JSON

- The first line of the **with** statement is very similar.
- The only change is that you need to open the file in '**w**' (write) mode to be able to modify the file.

Open orders.json in write mode

File Object

```
with open("orders.json", 'w') as file:  
    # Write to the JSON file
```

# Read, Write and Parse JSON

- **Writing JSON to a file**

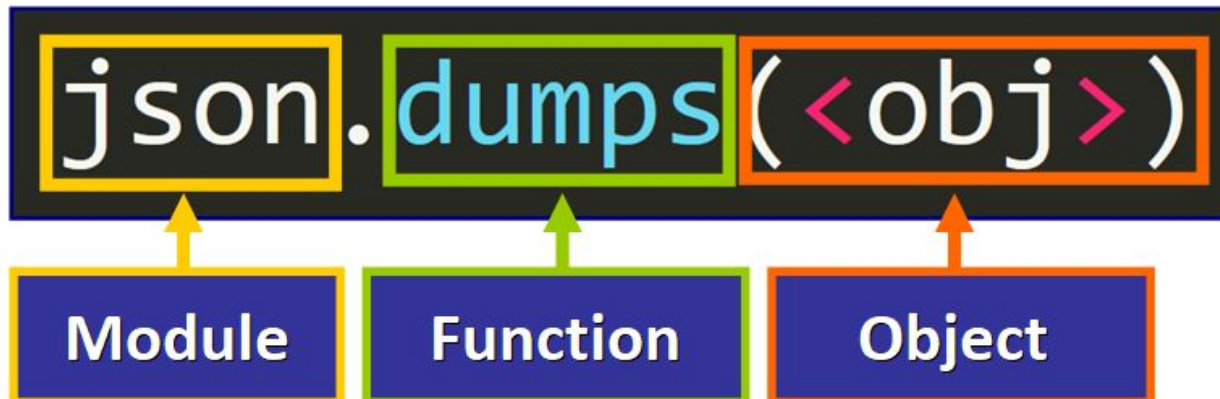
```
import json
```

```
person_dict = {"name": "Bob",  
               "languages": ["English", "French"],  
               "married": True,  
               "age": 32  
               }
```

```
with open('person.txt', 'w') as json_file:  
    json.dump(person_dict, json_file)
```

# Read, Write and Parse JSON

- **Python Dictionary to JSON String**
- create a Python dictionary from a string with JSON format.
- But sometimes we might need to do exactly the opposite, creating a string with JSON format from an object (for example, a dictionary) to print it, display it, store it, or work with it as a string.
- To do that, we can use the dumps function of the json module, passing the object as argument:



# Read, Write and Parse JSON

**# Python Dictionary**

```
client = {  
    "name": "Nora",  
    "age": 56,  
    "id": "45355",  
    "eye_color": "green",  
    "wears_glasses": False  
}
```

**# Get a JSON formatted string**

```
client_JSON = json.dumps(client)
```

# Read, Write and Parse JSON

**`client_JSON = json.dumps(client)`**

- `json.dumps(client)` creates and returns a string with all the key-value pairs of the dictionary in JSON format.
- Then, this string is assigned to the `client_JSON` variable.
- **`{"name": "Nora", "age": 56, "id": "45355", "eye_color": "green", "wears_glasses": false}`**



# Read, Write and Parse JSON

- How to Print JSON With Indentation

```
{"name": "Nora", "age": 56, "id": "45355", "eye_color":  
"green", "wears_glasses": false}
```

- We can improve the readability of the JSON string by adding **indentation**. To do this automatically, we just need to pass a second argument to specify the number of spaces that we want to use to indent the JSON string:

```
json.dumps(<obj>, indent=<spaces>)
```

Number of spaces for indentation

# Read, Write and Parse JSON

- How to Print JSON With Indentation

```
{"name": "Nora", "age": 56, "id": "45355", "eye_color":  
"green", "wears_glasses": false}
```

- We can improve the readability of the JSON string by adding **indentation**. To do this automatically, we just need to pass a second argument to specify the number of spaces that we want to use to indent the JSON string:

```
json.dumps(<obj>, indent=<spaces>)
```

Number of spaces for indentation

# Read, Write and Parse JSON

**Tip:** the second argument has to be a non-negative integer (number of spaces) or a string. If indent is a string (such as `"\t"`), that string is used to indent each level.

Now, if we call `dumps` with this second argument:

- **`client_JSON = json.dumps(client, indent=4)`**

The result of printing `client_JSON` is:

- **`{ "name": "Nora", "age": 56, "id": "45355", "eye_color": "green", "wears_glasses": false }`**

# Read, Write and Parse JSON

- **How to Sort the Keys**
- You can also sort the keys in alphabetical order if you need to. To do this, you just need to write the name of the parameter **sort\_keys** and pass the value **True**:

```
json.dumps(<obj>, sort_keys=True)
```

Sort names alphabetically or not

**Tip:** The value of `sort_keys` is `False` by default if you don't pass a value.

# Read, Write and Parse JSON

For example:

- **`client_JSON = json.dumps(client, sort_keys=True)`**

Returns this string with the keys sorted in alphabetical order:

- **`{"age": 56, "eye_color": "green", "id": "45355", "name": "Nora", "wears_glasses": false}`**

# Read, Write and Parse JSON

## How to Sort Alphabetically and Indent (at the same time)

- To generate a JSON string that is sorted alphabetically and indented, you just need to pass the two arguments:

In this case, the output is:

- `{ "age": 56, "eye_color": "green", "id": "45355", "name": "Nora", "wears_glasses": false }`

**Tip:** You can pass these arguments in any order (relative to each other), but the object has to be the first argument in the list.

# Read, Write and Parse JSON

	load()	loads()
Purpose	Create a Python object from a JSON file	Create a Python object from a string
Argument	JSON File	String
Return Value	Python object	Python object

# Read, Write and Parse JSON

	<b>dump()</b>	<b>dumps()</b>
<b>Purpose</b>	Write an object in JSON format to a file	Get a JSON string from an object
<b>Arguments</b>	Object + File	Object
<b>Return Value</b>	None	String



# Summary

- JSON (JavaScript Object Notation) is a format used to represent and store data.
- It is commonly used to transfer data on the web and to store configuration settings.
- JSON files have a .json extension.
- You can convert JSON strings into Python objects and vice versa.
- You can read JSON files and create Python objects from their key-value pairs.
- You can write to JSON files to store the content of Python objects in JSON format.

# JSON Schema

- **JSON Schema is a contract for your JSON document that defines the expected data types and format of each field in the response**



JSON Schema

# Why JSON Schema Validation required?

- Using JSON Schema to construct a model of your API response makes it easier to validate your API is returning the data it should.
- Monitor your API responses, ensuring they adhere to a specified format.
- Get alerted when breaking changes occur.

# JSON schema – Schema Validation

- JSON Schema is a specification for JSON based format for defining the structure of JSON data. It was written under IETF draft which expired in 2011. JSON Schema –
  - **Describes your existing data format.**
  - **Clear, human- and machine-readable documentation.**
  - **Complete structural validation, useful for automated testing.**
  - **Complete structural validation, validating client-submitted data.**

# JSON schema – Schema Validation

## **JSON Schema Validation Libraries**

There are several validators currently available for different programming languages.

Currently the most complete and compliant JSON Schema validator available is JSV.

# JSON schema – Schema Validation

Languages	Libraries
C	WJElement (LGPLv3)
Java	json-schema-validator (LGPLv3)
.NET	Json.NET (MIT)
ActionScript 3	Frigga (MIT)
Haskell	aeson-schema (MIT)
Python	Jsonschema
Ruby	autoparse (ASL 2.0); ruby-jsonschema (MIT)
PHP	php-json-schema (MIT). json-schema (Berkeley)
JavaScript	Orderly (BSD); JSV; json-schema; Matic (MIT); Dojo; Persevere (modified BSD or AFL 2.0); schema.js.

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "Product",
  "description": "A product from Acme's catalog",
  "type": "object",

  "properties": {

    "id": {
      "description": "The unique identifier for a product",
      "type": "integer"
    },

    "name": {
      "description": "Name of the product",
      "type": "string"
    },

    "price": {
      "type": "number",
      "minimum": 0,
      "exclusiveMinimum": true
    }
  },

  "required": ["id", "name", "price"]
}
```

# Keywords in Schema

Sr.No.	Keyword & Description
1	<b>\$schema</b> The \$schema keyword states that this schema is written according to the draft v4 specification.
2	<b>title</b> You will use this to give a title to your schema.
3	<b>description</b> A little description of the schema.
4	<b>type</b> The type keyword defines the first constraint on our JSON data: it has to be a JSON Object.
5	<b>properties</b> Defines various keys and their value types, minimum and maximum values to be used in JSON file.
6	<b>required</b> This keeps a list of required properties.
7	<b>minimum</b> This is the constraint to be put on the value and represents minimum acceptable value.



# Keywords in Schema

Sr.No.	Keyword & Description
8	<b>exclusiveMinimum</b> If "exclusiveMinimum" is present and has boolean value true, the instance is valid if it is strictly greater than the value of "minimum".
9	<b>maximum</b> This is the constraint to be put on the value and represents maximum acceptable value.
10	<b>exclusiveMaximum</b> If "exclusiveMaximum" is present and has boolean value true, the instance is valid if it is strictly lower than the value of "maximum".

# Keywords in Schema

Sr.No.	Keyword & Description
11	<b>multipleOf</b> A numeric instance is valid against "multipleOf" if the result of the division of the instance by this keyword's value is an integer.
12	<b>maxLength</b> The length of a string instance is defined as the maximum number of its characters.
13	<b>minLength</b> The length of a string instance is defined as the minimum number of its characters.
14	<b>pattern</b> A string instance is considered valid if the regular expression matches the instance successfully.

# Floating Point Arithmetic: Issues and Limitations

<https://docs.python.org/3/tutorial/floatingpoint.htm>

1

# Floating-point numbers

- Floating-point numbers are represented in computer hardware as base 2 (binary) fractions. For example, the decimal fraction
- 0.125
  - has value  $1/10 + 2/100 + 5/1000$ , and in the same way the binary fraction
- 0.001
  - has value  $0/2 + 0/4 + 1/8$ .
- These two fractions have identical values, the only real difference being that the first is written in base 10 fractional notation, and the second in base 2.

# Floating-point numbers

- Unfortunately, most decimal fractions cannot be represented exactly as binary fractions.
- A consequence is that, in general, the decimal floating-point numbers you enter are only approximated by the binary floating-point numbers actually stored in the machine.
- The problem is easier to understand at first in base 10.

# Floating-point numbers

Consider the fraction  $1/3$ . You can approximate that as a base 10 fraction:

**0.3**

or, better,

**0.33**

or, better,

**0.333**

and so on.

No matter how many digits you're willing to write down, the result will never be exactly  $1/3$ , but will be an increasingly better approximation of  $1/3$ .

# Floating-point numbers

```
>>> import math
>>> format(math.pi, '.12g')
'3.14159265359'
>>> format(math.pi, '.2f')
'3.14'
>>> repr(math.pi)
'3.141592653589793'
>>> .1 + .1 + .1 == .3
False
>>> round(.1, 1) + round(.1, 1) + round(.1, 1) == round(.3, 1)
False
>>> round(.1 + .1 + .1, 10) == round(.3, 10)
True
>>> 3602879701896397 / 2 ** 55
0.1
>>> x = 3.14159
>>> x.as_integer_ratio()
(3537115888337719, 1125899906842624)
>>> x.hex()
'0x1.921f9f01b866ep+1'
>>> sum([0.1] * 10) == 1.0
False
>>> math.fsum([0.1] * 10) == 1.0
True
```

# Floating-point numbers

- In the same way, no matter how many base 2 digits you're willing to use, the decimal value 0.1 cannot be represented exactly as a base 2 fraction. In base 2,  $1/10$  is the infinitely repeating fraction
- 0.0001100110011001100110011001100110011001100110011001100110011...
- Stop at any finite number of bits, and you get an approximation. On most machines today, floats are approximated using a binary fraction with the numerator using the first 53 bits starting with the most significant bit and with the denominator as a power of two. In the case of  $1/10$ , the binary fraction is  $3602879701896397 / 2^{55}$  which is close to but not exactly equal to the true value of  $1/10$ .

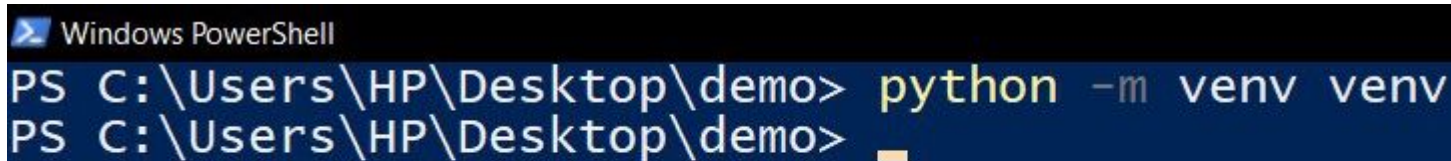


# **Virtual Environments**

# Step 1: Create a virtual environment

- Open the directory where you want to create your project. open cmd/powershell and navigate to the same directory and run the following commands to create a virtual environment.

**python -m venv venv**

A screenshot of a Windows PowerShell terminal window. The title bar at the top says "Windows PowerShell". The terminal has a dark blue background with white text. The first line shows the prompt "PS C:\Users\HP\Desktop\demo>" followed by the command "python -m venv venv". The second line shows the prompt "PS C:\Users\HP\Desktop\demo>" followed by a single underscore character "\_".

```
Windows PowerShell
PS C:\Users\HP\Desktop\demo> python -m venv venv
PS C:\Users\HP\Desktop\demo> _
```