PYTHON HAND WRITTEN NOTES

- **Comments** -

Starts with #

```
#This is a Comment
print("Hello, World!")
```

- Multiline Comments

```
# Comment 1
# Comment 2
# Comment 3
print("Hello, World!")
```

Or

```
"""
This is a Comment
written in
more than just 1 line
"""
print("Hello, World!")
```

- **Variables** - A variable is the name given to a memory location in a program. for example-

```
a = 30
b = "Tareen"
c = 52.25
```

Ex- x = 5
y = "cm" → 'cm' (same)
print(x)
print(y)

=> 5
cm

Variable name rules-
i) Must start with a letter or underscore
ii) Cannot start with number
iii) Contain (A-Z, 0-9, _)
iv) Case Sensitive

Assign Value to Multiple Variables-

```
x, y, z = "Tareen", "cm", "pc"
print(x)
print(y)
print(z)
```

Also -

```
x, y, z = "Tareen"
```

→ Same value assign

```
x = "Awesome"
print("Python is " + x)
```

→ Python is Awesome

Global Variable -

```
x = "awesome"

def myfunc():
    print("Python is " + x)

myfunc()
```

⇒ Python is Awesome

- Data Types -

1) Integers
2) Floating Point Numbers
3) Strings
4) Booleans
5) None

a = 52 → class <int>
b = 50.24 → class <float>
name = "CM" → class <str>

```
a = 5
print(type(a))
```

↳ <class 'int'>

- Numbers -

Random Number -

Python has a built-in module called random that can be used to make random numbers.

Ex -

```
import random
print(random.randrange(1,10))
```

⇒ 4
↳ Random Values

- Casting - There may be times when you want to specify a type on to a variable. This can be done with casting.

Ex -

```
x = int(1)
y = int(1.28)
z = int("3")
print(x)
print(y)
print(z)
```

⇒ 1
2
3

- **Strings** -

'hello' or "hello"

0 1 2 3 4

-5 -4 -3 -2 -1

Multiline String -

```
a = """ ____
    ____
    ____ """
print(a)
```

OR

```
a = ''' ____
    ____
    ____ '''
print(a)
```

Strings are Arrays -

0 1 2 3 4

-5 -4 -3 -2 -1

```
a = "Hello, World!"
print(a[1])
```

⇒ e

Slicing -

```
b = "Hello, World!"
print(b[2:5])
```

⇒ llo

```
a = "Hello, World!"
print(len(a))
```

⇒ 13

String Methods -

strip() → Removes any whitespace from the beginning or end

lower() → Returns the string in lower case

upper() → Returns the string in upper case

replace() → Replaces a string with another string.

split() → Splits the string into substrings if it finds instances of the separator

Check String - To check if a certain phrase or character is present in a string, use keywords **in** or **not in**

Ex -

```
txt = "The rain ____ "
x = "ain" in txt
print(x)
```

⇒ True

String Concatenation -

To Combine, two strings you can use the + operator

Ex-
```
a = "Hello"
b = "World"
c = a + b  ->  a + " " + b   (add space)
print(c)
```

⇒ HelloWorld

String format -

The format() method takes the passed arguments, formats them, and places them in the string where the placeholders {} are:

Ex-
```
age = 36
txt = "My age is {}"
print(txt.format(age))
```

⇒ My ~~name~~ age is 36

Escape Sequence Character - Sequence of characters after backslash '\' Ex- \n (newline), \t (tab)

• Booleans -

Two Values - True, False

Ex-
```
print(10 > 9)
print(10 == 9)
print(10 < 9)
```

⇒ True
False
False

• Operators -

- Arithmetic operators (+, -, *, / etc)
- Assignment operators (=, +=, -= etc)
- Comparision operators (==, >, >=, <, !=)
- Logical operators (and, or, not)
- Identity operators (is, is not)
- Membership operators (in, not in)

input() function -

This function allows the user to take input from the keyboard as a string.

a = input ("Enter Name")
↳ If a is "CM", the user entered CM

(It is important to note that the output of input is always a string)

- **Lists** -

There are four Collection data types in Python -

- List
- Set
- Tuple
- Dictionary

- List is a collection which is ordered and changeable. Allows duplicate members.

Ex -

```
thislist = ["cm", "fm", "lm"]
print(thislist)
```

⇒ ['apple', 'fm', 'lm']
cm
↳0 ↳1 ↳2
(Index)

- Store a list of values of any data type

friendz = ["Affu", "Rohan", 7, False]
↓ str() ↓ int() ↓ bool()

(can store value of any datatype)

List Indexing -

A list can be indexed just like a string.

0 1 2 → +ve indexing
11 = [7, 5, "Tarun"]
-3 -2 -1 → -ve indexing

11[0] = 7
11[1] = 5
11[50] = Error
11[0:2] = [7, 5] ⇒ List Slicing

[:3] → Beginning to index[2]
[1:] → Index[1] to Last

To change item Value -

```
thislist = ["cm", "fm", "lm"]
thislist[1] = "TK"
print(thislist) ⇒ ['cm', 'TK', 'lm']
```

Loop through a list -

```
thislist = ["cm", "fm", "lm"]
for x in thislist:
  print(x)
```

⇒ cm
fm
lm

Check if item exists -

```
thislist = ["cm", "fm", "lm"]
if "cm" in thislist:
  print("yes, 'cm' is in the list")
```

⇒ yes, 'cm' is in the list

List length -

```
thislist = ["cm", "fm", "lm"]
print(len(thislist))
```

⇒ 3
↳ length of items in the list

List Methods -

$l_1 = [1, 15, 7, 21, 45]$

1] l_1.sort(): Updated the list to [1, 7, 15, 21, 45]
2] l_1.reverse(): Updated the list to [45, 21, 7, 15, 1]
3] l_1.append(4): Adds 4 at the end of the list
4] l_1.insert(3, 8): This will add 8 at 3 index (3 = index no., 8 = no.)
5] l_1.pop(2): Will delete element at index 2 and return its value
6] l_1.remove(21): Will remove 21 from the list
7] l_1.clear(): Empties the list
8] $l_2 = l_1$.copy(): Make a copy of the list

Join Two Lists -

```
list1 = ["a", "b", "c"]
list2 = [1, 2, 3]
list3 = list1 + list2
print(list3)
```

⇒ ['a', 'b', 'c', 1, 2, 3]

- **Tuples** -

A tuple is a collection which is ordered and unchangeable.
- written with round brackets.

Create a tuple:

```
thistuple = ("cm", "pm", "th")
print(thistuple)
```

```
          0     1     2
⇒ ('cm', 'pm', 'th')
   -3    -2    -1
```

a = () → Empty Tuple
a = (1,) → Tuple with only one element needs a comma
a = (1,5,7) → Tuples with more than 1 element

Change Tuple Values Using List -

```
x = ("cm", "pm", "th")
y = list(x) → Tuple converted into list
y[1] = "lm"
x = tuple(y) → List converted into Tuple.
print(x)
    ⇒ ('cm', 'lm', 'th')
```

Tuple Methods -

1) a.count(1): a.count(1) will return number of times 1 occurs in a
2) a.index(1): a.index(1) will return the index of first occurance of 1 in a

- **Lists** - A list is a collection which is unordered and unindexed.
- written with curly brackets

Create a list -

```
thislist = {"cm", "pm", "tm"}
print(thislist)
```

[∴ NoTe - The items appear in a random order]

⇒ {'pm', 'tm', 'cm'}

Properties -
- Lists are unordered
- Lists are unindexed
- There is no way to change items
- Lists cannot contain duplicate values

Operations on Sets -

S = {1, 8, 2, 3}

1] len(s) : Returns 4, the length of the set
2] S.remove(8) : Updates the set S and removes 8 from S
3] S.pop() : Removes an arbitrary element from the set and returns the element removed.
4] S.clear() : Empties the set S
5] S.union({8, 11}) : Returns a new set with all items from both sets ⇒ {1, 8, 2, 3, 11}
6] S.intersection({8, 11}) : Returns a set which contains only items in both sets.
⇒ {8}

- Dictionary -

A dictionary is a collection which is unordered, changable and indexed.
- written in curly brackets
- they have keys and values.

Create a dictionary :

```
thisdict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1962
}
print(thisdict)
```

(keys ← "brand", "model", "year"; values → "Ford", "Mustang", 1962)

⇒ {'brand': 'Ford', 'model': 'Mustang', 'year': 1962}

Properties -
- It is unordered
- It is mutable
- It is indexed
- Cannot contain dupliate keys

Dictionary Methods -

```
a = { "name": "Tarun",
      "from": "India",
      "marks": [92, 98, 99] }
```

1] a.items() : Returns a list of (Key, Value) tuple
2] a.keys() : Returns a list containing dictionary's key

3] a.update({"friend": "Sam"}): Updated the dictionary with supplied key-value pairs

4] a.get("name"): Returns the value of the specified keys (and value is returned)

Adding Items -

```
a["color"] = "red"
print(a)
```

Removing Items -

```
a.pop("friend")
print(a)
```

Nested Dictionaries -

```
a = {
"a1": {"name": "Tarun",
       "age": 21
      },
"a2": {"name": "Sahil",
       "age": 20
      }
}
print(a)
```

If... Else Condition -

```
a = 34
b = 200
if b>a:
print("b is greater than a")
```

⇒ b is greater than a

Elif Condition -

```
a = 33
b = 33
if b>a:
  print("b is greater than a")
elif a==b:
  print("a and b are equal")
```

⇒ a and b are equal

Else Condition -

```
a = 200
b = 33
if b>a:
  print("b is greater than a")
elif a==b:
  print("a and b are equal")
```

else: ("a is greater than b")

⇒ a is greater than b

And Operator -

```
a = 200
b = 33
c = 500
if a > b and c > a:
  Print ("Both Conditions are True")
```

⇒ Both Conditions are true

Or Operator -

```
if a > b or a > c:
  Print ("At least one of the
Conditions is true")
```

Nested if -

```
x = 40
if x > 10:
  Print ("Above 10")
if x > 20:
  Print ("Above 20").
else: Print ("but not above 50")
```

⇒ but not above 50

The Pass Statement -

if statements cannot be empty, but if you for some reason have an if statement with no content, put it in the pass statement to avoid getting error msg.

While Loops -

```
i = 1
while i < 6:
  Print(i)
  i += 1
```

⇒ 1
2
3
4
5

The Break Statement -

We can stop the loop even if the while condition is true

```
i = 1
while i < 6:
  print(i)
  if (i == 3)
    break
  i += 1
```

⇒ 1
2
3

The Continue Statement -

```
i = 0
while i < 6:
  i += 1
  if i == 3:
    continue
  print(1)
```

⇒ 1
2 → 3 is missing
4
5
6

• For Loops -

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
```

⇒ apple
banana
cherry

The range() function -

```
for x in range(6):   → (0-5)
  print(x)
```

⇒ 0
1
2
3
4
5

range(2,6) → (2-5)
2
3
4
5
(output)

range(2,30,3)
2
5
8
:
29
(+3)
↓ increment value

Nested for loop -

```
a1 = ["a", "b", "c"]
a2 = ["d", "e", "f"]

for x in a1:
    for y in a2:
        print(x, y)
```

⇒
```
a d
a e
a f
b d
b e
b f
c d
c e
c f
```

For Loop With Else -

```
a = [1, 2, 3]
for x in a
print(x)
else
    print("Done")
```

→ This is printed when the loop exhausts!

⇒
```
1
2
3
Done
```

- **Function** - A function is a group of statements performing a specific task

Syntax -

```
def func1():
    print("Hello")
```

This function can be called any number of times, anywhere in the program.

Function Call -

func1() → This is called function call

Function Definition -

The part containing the exact set of instructions which are executed during the function call

Types of function -

1. Built in function → already present
2. User defined functions → defined by the user

Built in function -
len(), print(), range()
etc.

User defined functions -
func1() function is a user defined function

Functions With Arguments -

```
def greet(name):
    gr = "Hello" + name
    return gr

a = greet("Harry")
```

"Harry" is passed to greet in name

a will now Contain "Hello Harry"

Default parameter Value -

```
def greet(name = "Stranger"):
    # function_body
```

greet() → Name will be "Stranger" in function body (default)
greet("Harry") → Name will be "Harry" in function body (passed)

Arbitrary Arguments - *

If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition.

Ex-

```
def my_function(*kids):
    print("The youngest child is" +
          kids[2])
my_function("Tx", "Cm", "PC")
```

(0, 1, 2)

⇒ The youngest child is PC (2)

Recursion - Recursion is a function which calls itself.
It is used to directly use a mathematical formula as a function
for ex-

factorial(n) = n x factorial(n-1)
This function can be defined as follows:

```
def factorial(n):
    if i==0 or i==1:  → Base Condition which doesn't call the function any further
        return 1
    else:
        return n * factorial(n-1)
                    ↳ Function calling itself
```

This works as follows -

Factorial(4)
↳ [Function Called]
4 x Factorial(3)
4 x [3 x factorial(2)]
4 x 3 x [2 x factorial(1)]
4 x 3 x 2 x [1] [Function Returned]

- **Lambda function** :- A Lambda function can take any number of arguments, but can only have one expression.

Syntax -
lambda arguments : expression

Ex - x = lambda a: a + 10
print(x(5))

⇒ 15

- **Arrays** - Arrays are used to store multiple values in one single variable.

Ex - cars = ["Ford", "Volvo", "BMW"]
print(cars)

⇒ ['Ford', 'Volvo', 'BMW']
0 1 2

Access the elements of an Array -

cars = ["Ford", "Volvo", "BMW"]
x = cars[0]
print(x)

⇒ Ford

Modify the values of array items -

cars = ["Ford", "Volvo", "BMW"]
cars[0] = "Toyota"
print(cars)

⇒ ['Toyota', 'Volvo', 'BMW']

Length of an Array -

```
Cars = ["Ford", "Volvo", "BMW"]
x = len(cars)
print(x)
```

⇒ 3

Looping Array Elements -

```
Cars = ["Ford", "Volvo", "BMW"]
for x in cars:
  print(x)
```

⇒ Ford
Volvo
BMW

Removing Array Elements -

```
Cars = ["Ford", "Volvo", "BMW"]
cars.pop(1)
print(cars)
```

⇒ ['Ford', 'BMW']

Array Methods -

append() → Adds an element at the end of the list
clear() → Removes the elements from the list
copy() → Returns a copy of the list
count() → Returns the no of elements with the specified value
extend() → Add the elements of a list, to the end of the current list
index() → Returns the index of the first element with the specified value
insert() → Adds an element at the specified position
pop() → Removes the element at the specified position
remove() → Removes the first item with the specified value
reverse() → Reverses the order of the list
sort() → Sorts the list

• Classes and Objects -

Python is an Object Oriented Programming Language

To create a class, use the keyword **class**

Ex- class MyClass: → Class Name
x = 5
print(MyClass)

⇒ <class '__main__.MyClass'>

Create an Object -

class MyClass:
x = 5

Object ← [p1 = MyClass()
print(p1.x)

⇒ 5

The __init__() function -

Always executed when the class is being initiated.

Ex -

class Person:
def __init__(self, name, age):
self.name = name
self.age = age

p1 = Person("Tk", 20)

print(p1.name)
print(p1.age)

⇒ Tk
20

Note: The __init__() function is called automatically every time the class is being used to create a new object

Self parameter -

Self refered to the instance of the class. It is automatically passed with a function call from an object

harry.getSalary() → here self is harry
→ Equivalent to employee.getSalary(harry)

The function getSalary is defined as-

```
Class Employee:
    Company = "Google"
    def getSalary(self):
        print("Salary is not there")
```

Static Method -

Sometimes we need a function that doesn't use self parameter. We can define a static method like this:

```
@staticmethod  → Decorator to make greet as a static method
def greet():
    print("Hello")
```

Object Methods -

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def myfunc(self):
        print("Hello my name is " + self.name)

p1 = Person("Tk", 20)
p1.myfunc()
```

→ Hello my name is Tk

- Inheritance -

Inheritance allows us to define a class that inherits all the methods and properties from another class.

Create a Parent Class -

Any class can be Parent Class, so the syntax is the same as creating any other class.

Create a Child Class -

```
class Student(Person):   → Parent Class
    pass                 → Child Class
```

Note: The child's __init__() function overrides the inheritance of the parent's __init__() function.

To keep the inheritance of the parent's __init__() function, add a call to the parent's __init__() function.

Ex-

```
              child        Parent
class Student(Person):
    def __init__() (self, fname):
        Person.__init__(self, fname)
```

↳ Call the Parent's __init__() function

Use the super() function -

By using the super() function, you do not have to use the name of the parent element, it will automatically inherit the methods and properties from its parent.

Ex-

```
class Student(Person):
    def __init__() (self, fname):
        super().__init__() (fname)
```

↳ Not using Parent Class Name

Add properties -

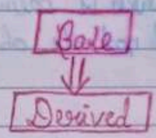
```
self.year = 2021
```

↳ Print (x.year)

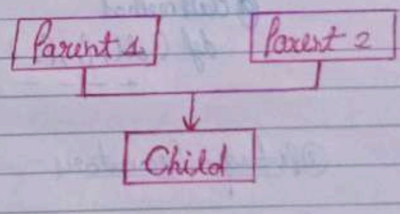
Types of Inheritance -

1. Single Inheritance
2. Multiple Inheritance
3. Multilevel Inheritance

1. Single Inheritance - It occurs when child class inherits only a single parent class.



2. Multiple Inheritance - It occurs when the child class inherits from more than one parent class.



3. Multilevel Inheritance - When a child class becomes a parent for another child class.

Parent
↓
Child 1
↓
Child 2

Class Methods -

A class method is a method which is bound to the class and not the object of the class.

@classmethod decorator is used to create a class method.

Syntax -

@classmethod
def (cls, p1, p2):

@Property Decorators -

Consider the following Class -

class Employee:

@property
def name (self):
return self.ename

if e = Employee() is an object of class employee, we can print (e.name) to print the ename/ call name() function.

@getters & @setters -

The method name with @property decorator is called getter method.
We can define a function + @name.setter decorator like below:

@name.setter
def name (self, value):
self.ename = value

Operator overloading in python -

Operators in python can be overloaded using dunder methods.
These methods are called when a given operator is used on the objects

Operators in Python can be overloaded using the following methods:

$p_1 + p_2 \longrightarrow p_1.\_\_add\_\_(p_2)$
$p_1 - p_2 \longrightarrow p_1.\_\_sub\_\_(p_2)$
$p_1 * p_2 \longrightarrow p_1.\_\_mul\_\_(p_2)$
$p_1 / p_2 \longrightarrow p_1.\_\_truediv\_\_(p_2)$
$p_1 // p_2 \longrightarrow p_1.\_\_floordiv\_\_(p_2)$

- Python Iterators -

An iterator is an object that contains a countable number of values.

Methods
__iter__()
__next__()

Iterator vs Iterable

Iterable ←

```
mytuple = ("TK", "LK", "MK")
myit = iter(mytuple)        → Iterator
print(next(myit))
print(next(myit))
print(next(myit))
```

⇒ TK
LK
MK

→ Strings are also iterable objects, containing a sequence of characters.

Ex-

```
mystr = "banana"
myit = iter(mystr)
print(next(myit))
"
"
"
"
"
```

⇒ b
a
n
a
n
a

Python Scope -

A variable is only available from inside the region it is created. This is called scope.

- Local Scope - Global Scope

Global Keyword -

Need to create a global variable in the local scope, use global keyword.

Ex-

```
def myfunc():
    global x
    x = 300

myfunc()
print(x)
```

⇒ 300

Also, use the global keyword if you want to make a change to a global variable inside a function.

Python Modules -

A file containing a set of functions you want to include in your application.

Create a Module -

To create a module just save the code you want in a file with the extension .py:

Ex-

Save this code in a file named mymodule.py

⇒
```
def greeting(name):
    print("Hello, " + name)
```

Use a Module -

Now import the module named mymodule, and call the greeting function: using import

```
import mymodule
mymodule.greeting("Tarun")
```

⇒ Hello, Tarun

Re-naming a Module -

Using as Keyword.

Ex-

import mymodule as mx

↓ New Module Name

dir() function -

Built-in function to list all the function names in a module

Import From Module =>

You can choose to import only parts from a module, by using the from keyword.

Ex-

from mymodule import person1

print(person1["age"])

<=

- Python Datetime -

module -> datetime

Ex-

import datetime

x = datetime.datetime.now()

print(x)

=> 2021-06-16 09:45:52.28764

year Month Date hour min Sec microsec.

Creating Date Objects -

x = datetime.datetime(2021, 06, 16)

- Python Math - Built-in functions

x = min(10, 52, 72)

y = max(50, 872, 2)

print(x)

print(y)

=> 10

872

The Math Module -

```
import math
```

Ex -

```
import math
x = math.sqrt(64)
print(x)
```

⇒ 8.0

- **Python PIP -**

PIP is a package manager for python packages, or modules.

- **Python User Input -**

```
username = input("Enter username:")
print("Username is: " + username)
```

⇒ Enter Username: Tarun
Username is: Tarun

- **Python String formatting -**

format() Method

String format()

The format() method allows you to format selected parts of a string.

Add {} in the text and run the values through the format() method:

Ex - ①

```
price = 52
txt = "The price is {} dollars"
print(txt.format(price))
```

⇒ The price is 52 dollars

②

```
quantity = 3
item = 600
price = 49
myorder = "I want {} pieces of item number {} for {:.2f} dollars"
print(myorder.format(quantity, itemno, price))
```

⇒ I want 3 pieces of item number 600 for 49.00 dollars

- Exception Handling -

There are many built-in exceptions which are raised in python when something goes wrong.

Exceptions in python can be handled using a try statement.

The code that handles the exception is written in the except clause.

```
try:
    # Code                    → Code which might throw Exception
except Exception as e:
    print(e)
```

When the exception is handled, the code flow continues without program interruption.

Raise Exceptions -

We can raise custom exceptions using the raise keyword in python.

Ex -

```
def increment(num):
    try:
        return int(num)+1
    except:
        raise ValueError("Not Good")

a = increment('ab')
print(a)
```

→ raise an Custom ValueError

Try with Finally -

Python offers a finally clause which ensures execution of a piece of code irrespective of the exception.

```
try:
    # some code
except:
    # some code
finally:
    # some code → Executed regardless of error
```

Else - You can use the else keyword to define a block of code

To be executed if no errors were raised.

Ex-

```
try:
    Print("Hello")
except:
    Print("Something went wrong")
else:
    Print("Nothing went wrong")
```

- Python File Handling -

The key function for working with files in python is the open() function.

It takes 2 parameters:
filename, Mode

4 different Methods for opening a file

"r" - Read
"a" - Append
"w" - Write
"x" - Create

"t" - Text Mode
"b" - Binary Mode (.jpg, .dat)

Syntax -

```
f = open("filename.txt")
```
↓ Same
```
f = open("filename.txt", "rt")
```
(rt: "r" → read, "t" → text)

Opening a file -

```
open("this.txt", "r")
```
open is a built-in function | filename | Read Mode

Reading a file -

```
f = open("this.txt", "r")
text = f.read()      → Read its Content
print(text)          → Print its Content
f.close()            → close the file
```

We can also specify the number of characters in read() function:

```
f.read(2)
```
↓
Reads first 2 characters

We can also use f.readline() function to read one full line at a time.

f.readline() → Reads one line from the file

Writing files in Python -

In order to write to a file, we first open it in write or append mode after this, we use the Python's f.write() method to write to the file!

```
f = open("this.txt", "w")
f.write("This is nice")
f.close()
```
↳ Can be called multiple times

With Statement -

The best way to open and close the file automatically is the with statement.

```
with open("this.txt") as f:
    f.read()
```
↳ Don't need to write f.close() as it is done automatically

Python MySQL -

Create Connection -

```
import mysql.connector

mydb = mysql.connector.connect(
    host = "localhost",
    user = "my_username",
    password = "mypassword"
)

print(mydb)
```

Creating a Database -

To create a Database in MySQL, use the "CREATE DATABASE" statement:

Ex-

```
import mysql.connector

mydb = mysql.connector.connect(
    host = "localhost",
    user = "myusername",
    password = "mypassword"
)
```

```
mycursor = mydb.cursor()

mycursor.execute ("CREATE DATABASE
                   mydatabase")
```

Creating a Table -

To create a Table in MySQL, use the "CREATE TABLE" statement

Ex -

```
import mysql.connector

database = "mydatabase"
)
mycursor = mydb.cursor()

mycursor.execute ("CREATE TABLE
    customers (name VARCHAR(255),
    address VARCHAR(255))")
```

↳ Table Created with name "customers" in the database named "mydatabase"

Check if Table Exists -

You can check if a table exist by listing all tables in your database with the "SHOW TABLES" statement:

Ex -

```
mycursor.execute ("SHOW TABLES")

for x in mycursor:
    print(x)
```

⇒ ('customers') → Table name

Primary Key -

We use the statement -
"INT AUTOINCREMENT PRIMARY KEY"

Ex -

```
mycursor.execute ("CREATE TABLE
    customers (id INT AUTO_INCREMENT
    PRIMARY KEY, name VARCHAR(255),
    address VARCHAR(255))")
```

↳ Table Created - "customers" with unique id

If the Table already exists, use the ALTER TABLE keyword:

Ex-

```
mycursor.execute("ALTER TABLE customers ADD COLUMN id INT AUTO_INCREMENT PRIMARY KEY")
```

↳ Unique i'd created

Insert into Table -

To Fill a table in MySql, use the "INSERT INTO" statement.

Ex-

```
mycursor = mydb.cursor()
sql = "INSERT INTO customers (name, address) VALUES (%s, %s)"
val = ("John", "Highway 21")
mycursor.execute(sql, val)
mydb.commit()
print(mycursor.rowcount, "record inserted.")
```

→ Insert data in table

→ Make the changes to the table

↳ Return the no. of row Printed

Insert Multiple Rows -

To insert multiple rows into a table, use the executemany() method

Ex-

```
sql = "INSERT INTO customers (name, address) VALUES (%s, %s)"
val = [
  ('Peter', 'Grovendhan'),
  ('Tk', 'Sp'),
  ('Lk', 'Tp'),
  ('Ml', 'La'),
  ('Cm', 'Gm')
]
mycursor.executemany(sql, val)
mydb.commit()
print(mycursor.rowcount, "record was inserted.")
```

→ insert multiple rows

→ Make changes

⇒ 5 record was inserted.

Note - If you insert more than 1 row, the id of the last inserted row is returned

⇒ mycursor.lastrowid

Select From a Table -

To select from a Table in MySQL, use the "SELECT" statement:

Ex-

```
mycursor.execute("SELECT * FROM customers")   → Select the data
myresult = mycursor.fetchall()                → Fetch all the data
for x in myresult:
  print(x)
```

Selecting Columns

```
mycursor.execute("SELECT name, address FROM customers")
```

→ Select address column from Table

Using the fetchone() Method -

The fetchone() method will return the first row of the result.

⇒ `myresult = mycursor.fetchone()`

Select with a Filter -

You can filter the selection by using the "WHERE" statement:

Ex -

```
sql = "SELECT * FROM customers WHERE address = 'TK'"
mycursor.execute(sql)
myresult = mycursor.fetchall()
```

Order By -

Use the ORDER BY statement to sort the result in ascending or descending order.

Ex-

```
sql = "SELECT * FROM customers ORDER BY name"
```

→ Data sort in ascending order

Use the DESC keyword to sort the result in a descending order.

Ex- `sql = "SELECT * FROM customers ORDER BY name DESC"`

→ Data sort in descending order

Delete from -

You can delete records from an existing table by using the "DELETE FROM" statement.

Ex-

```
sql = "DELETE FROM customers WHERE
      address = 'TK' "
```

↓ Specifies which record should be deleted

Delete a Table -

You can delete an existing table by using the "DROP TABLE" statement.

Ex-

```
sql = "DROP TABLE customers"
```

↳ Table named customers should be deleted

OR

```
"DROP TABLE IF EXISTS customers"
```

↳ Table deleted if exists

Update Table -

you can update existing records in a table by using the "UPDATE" statement

Ex-

```
sql = "UPDATE customers SET address
      = 'TK' WHERE address = 'CM' "
```

↳ Address changed from 'CM' to 'TK'

Limit the Result -

You can limit the number of records returned from the query, by using the "LIMIT" statement.

Ex-

```
sql = mycursor.execute ("SELECT * FROM
      customers LIMIT 5")
```

Join -

Combine rows from two or more tables, based on a related column b/w them, by using JOIN statement

Ex-

```
sql = "SELECT \ users.name AS user,
      \ products.name AS favourite \
      FROM users \
      INNER JOIN products ON
      users.fav = products.id "
```

- Left Join -

If you want to show all users, even if they do not have a favourite product, use the LEFT JOIN statement:

Ex-

```
sql = "SELECT \ users.name AS user,
      \ products.name AS favourite \
      FROM users \ LEFT JOIN
      products ON users.fav = products.id"
```

⇒ Full result shown

- Right Join -

Select all records from Table A along with Table B along for which the Join condition is met.

- Full Join -

Select all records from Table A and Table B, regardless of whether the Join condition is met or not