Python

A high-level programming lang. & it is a interpreted programming lang (script mode, interactive mode), exection of code is done line by line & processes output. by GuidoVanRossum in 1991.

Appli:- Youtube, Netflix, Dropbox .....

Comments:- used to explain the code, make it more readable or temporarily disable parts of code during debugging. Two types:-

* Single line Coment (#)
* Multi-line Comment (''' ... ''')

Variables:- In python, variables are used to store data values. You don't need to declare a variable before using it or specify its type. A var. created the moment you 1st assign a value to it.

```
x = 5                a,b,c = 50.5, 1, "Alice"
y = "John"           print(a)
print(x)             print(b)
print(y)             print(c)
```

chained assignment :- a=b=c=10 print(a,b,c) # 10 10 10

Var are dynamically typed so can change type of variable by assigning a value of diff. type of it.

```
var = 235            var = "Hello"
print(var) # 23.5    print(var) # Hello
```

Rules:- Must start with letter or underscore, cannot start with number, can only contain alphanumeric char & underscores, case-sensitive.

Global & local Variables:- L.V inside a fn & cannot be accessed out of fun. Global var. defined outside any fn. & can be accessed anywhere in the code.

```
x = global                    O/p:- local
def fun():                          local
    global x  # declaring x as global var.
    x = "local"
    print(x)
fun()
print(x)
```

Datatypes:- The converter which converts real world data into m/c level lang. or tells the interpreter type of data it is holding.

You don't need to declare the type explicitly.

* Numeric Types:-

Int - Integer values (5, 10, -5, 1000)

float - Decimal or floating point no. (3.14, -5.4)

complex - real & imaginary part (3+4j, 5-2j)

* Sequence Types:-

*str:- sequence of char. ("Hello", "Python")

*list:- Ordered, mutable coll. ([1,2,3], ['a','b'])

*tuple:- Ordered, immutable coll. ((1,2,3), ('x','y'))

*range:- a sequence of numbers (range(3) - 0,1,2)

* Mapping Types:-

* dict :- key-value pairs {"name":"Abi", "age":25}

* Set Types:-

*set: Unordered collection of unique elem. {1,2,3}

*frozenset: Immutable version of set.

* Boolean Type

*bool - Represents True or False.

* Binary Type

*Bytes: Immutable seques of bytes ("Hello").

*bytearray: Mutable sequence of bytes.

* memoryview: Provides memory access to binary data.

*None Type

None Type represents absence of value.

TYPE Casting:-

int("5")	str → Int
float(3)	int → float
str(5)	int → str
bool(0)	# Returns False.

& reduces (getters &

Operators:-
* Arithemetic (+, -, *, /, //, %, **)
* Comparison (==, !=, >, <, >=, <=)
* Logical (and, or, not)
* Assignment (=, +=, -=, *= ...)
* Bitwise (&, |, ^, <<, >>)
* Identity (is, is not)
* Membership (in, not in)

Control Flow:- → Decision-Making stmt
→ Looping stmt
→ Jumping stmt.

Decision-Making stmt:-

The if stmt:- when we a single condition to check

```
ic_cream = "chocolate"
if ic_cream == "chocolate":
    print("Yummy! I love it")
```

The if-else stmt:- if we have more than one & exactly 2 condi. to check

```
ice_cream = "vanilla"
if ice_cream == "chocolate":
    print("Yummy! I love it")
else:
    print("I don't like it")
```

The if-elif-else stmt:- more than 2 condi

```
ice_cream = "chocolate":
if ice_cream == "vanilla":
    print("I don't like it")
elif ice_cream == "Butterscotch":
    print("I like but not much")
elif ice_cream == "Pista":
    print("I love it")
else:
    print("Yummy! wow! love it")
```

Loops or Iteration statements:-
like merry go round for your code. let you repeat actions without writing sm code over & over.

The for loop:- When you know how many times to repeat.

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

The while loop:- keeps going as long as condi is true.

```
count = 0
while count < 5:
    print("count is:", count)
    count += 1
```

Prints 0 to 4.

Jump statements:-
Jumpstmts are like secret passages in a video game - they let you skip parts of your code or exit loops early.

The break stmt:- exits a loop immediately.

```
for i in range(10):
    if i == 5:
        print("found 5! exiting loop")
        break
    print(i)
```

Prints 0 to 4

The Continue stmt:- skips the rest of current iteration & move to next one.

```
for i in range(10):
    if i == 2:
        print("skipping")
        continue
    print(i)
```

all from 0 to 9 except 2.

The Pass stmt:- like a placeholder, does nothing but useful when you need empty code blocks:

Functions:- A fn. is a block of code which only runs when it is called.
The values passed at the time of defining a fn. called parameter.
The values passed at the time calling the fn. is called arguments.
There are two types:-
* Arbitary → Positional arguments (args) (*args)
* Keyword arguments (kwargs) (**kwargs)

If we call the fn. without argument it uses default value.

```
def my_f(country = "Norway"):
    print("I am from" + country)
my_f("Sweden")
my_f() # Norway.
```

1. Positional-only arguments:- (Python3.8+)
must be passed only by position not by name. Declared before the / symbol in a fn. definition.

```
def func(a,b,/):
    Print(a,b)
func(20,30) # Valid
func(a=10, b=20) # Invalid
```

2. Keyword-only arguments:-
Passed using parameter names(keyword) Declared after * symbol in a fn. definition.

```
def func(*,a,b):
    Print(a,b)
func(a=10, b=20) # Valid
func(10,20) # Invalid
```

3. Both Positional & Keyword arguments:-
You can mix positional-only, positional or keyword & keyword-only arg. using / and *.

```
def func(x,y,/,z,*,a,b):
    Print(x,y,z,a,b)
func(1,2,3,a=3,b=5) # Valid
func(1,2,z=3,a=5,b=4) # Valid
func(x=1,y=2,z=3,a=4,b=5) # Invalid
```

x, y: Positional only
z: Positional or keyword
a, b: keyword-only.

/ - everything before me must be positional only - it is a formal way to declare that.
* - everything after me must be keyword-only

Lambda functions:-
Anonymous (unnamed) fn. Defined using the lambda key word.
syntax:-
lambda arguments: expression

```
add = lambda a,b: a+b
Print(add(2,3)) # 5
```

1. map() - Applies a fn. to each item in an iterable (list, tuple). Returns a map object.
syntax:-
map(function, iterable)
ex:-

```
nums = [1,2,3,4]
squared = list(map(lambda x:x**2, nums))
Print(squared)
```

2. filter() - Filters items from an iterable based on condi. only returns items where fn returns true.
syntax:- filter(function, iterable)
ex:-

```
nums = [1,2,3,4,5,6]
even = list(filter(lambda x: x%2==0, nums))
Print(even) # [2,4,6]
```

3. reduce() - Applies a fn. cumulatively to the items of a sequence, comes from functools module.

```
from functools import reduce
reduce(function, iterable)

nums = [1,2,3,4]
Product = reduce(lambda x,y: x*y, nums)
Print(Product) # 24
```

4. zip() - Combines multiple iterables element-wise. Returns tuples pairing items from each iterable.
syntax:
zip(iterable1, iterable2, ...)

```
names = ['Alice', 'Bob']
scores = [90, 85]
combined = list(zip(names, scores))
Print(combined) # output [('Alice', 90),
                          ('Bob', 85)]
```

Strings:- A series of char. that can include numbers or other here we treat them as char. strings in Python are surrounded by either single or double quotation marks.

```
a = "Hello World"
Print(a)
int("Tejaswini")

a = "Tejaswini
     is a
     goodgirl"
Print(a)
```

Python doesnot have a character data type, a single char. is simply a string with length 1.

Index value within square brackets helps in accessing elements of string.

To get length of a string use - len() fn.

a.lower() # tejaswin is a good girl

a.upper() # HELLO WORLD

.replace("world", "Python") # Hello Python

Slicing - range of characters by using slice syntax. specify start, stop, skip separated by colon (:)

a[p:5], a[::-1] a[: 5] a[5:]

include exclude

⟶ format()

```
Print(f"My name {a}")
age=16
text="My name John, I am {}"
Print(text.format(age)) # ..I am16
```

*strip() - removes whitespace in string.

List:- A datastructure that allows you to store a collection of items in a single variable. Ordered, mutable & allow duplicate values. index starts from 0.

Type() - type of datatype that list contains.

list() - Constructor returns a list a built in fn. fromiterables (string, list, tuple)

```
text = 'Python'
text_list = list(text)
Print(text_list) # ['P','y','t','h','O','n']
text = ('a','e','i','o','u') # tuple
Print(list(text)) # ['a','e','i','o','u']
text[1] = "O"
Print(list(text)) # ['a','o','i','o','u']
text[1:5] = ["u","o","i","e"]
Print(list(text)) # ["a","u","o","i","e"]
```

*insert() at a specific position ading anyelement.

```
text = ["A","B","C"]
Print(text.insert(2,"D")) # ["A","B","D"]
```

*Append() - to add element at end of list.

```
Print(text.append("E") # ["A","B","C","E"]
```

*Extend() - appends elements from another list to current list.

```
list1 = ["A","B","C"]
list2 = ["a","b","c"]
Print(list1.extend(list2)) # ["A","B","C","a","b","c"]
```

*remove() - To remove a specified element

```
Print(list1.remove("C")) # ["A","B"]
```

*pop() - removes specified index.

```
no print(list1.pop(1)) # ["A","C"]
```

* del - removes specified index.

```
del list1[0]
Print(del list1[0]) # B,C
```

*clear() - empties list.

```
Print(list1.clear()) # []
```

*sort() - sorts the list alphanumerically ascending by default.

```
list1 = [90, 54, 60, 75, 100]
Print(list1.sort()) # 100, 54, 60, 75, 90, 10
print(list1.sort(reverse=True))
                    # 100, 90, 75, 60, 54
```

```
def my-f(n,-50):
    return abs(n-50)
list1=[100,50,65,82,23]
print(list1.sort(key=my_f))
# [50, 65, 23, 82, 100]
```

reverse() - reverses current sorting order of elements..

```
print(list1.reverse()) # [23,82,65,50,100]
```

*copy() - copies the list values from other without referencens & don't not effect list 1 if changed to list2.

* Join() - use '+' to Join them. can also be done with extend().

List comprehension:- offers a shorter syntax when you want to create a new list based on values of existing list.

```
Newlist=[expr. for item in iterable if condi]
```

Tuple:- ()

Ordered, Immutable, Allow duplicates, elements can be of different types.

```
#Creation
t1=(1,2,3)
t2=tuple([4,5])   → constructor.
t_single=(5,)
#Access & Slicing
t1[0]
t1[-1]
t1[1:3]
# Methods
t1.count(2)
t1.index(3)
#Nested Tuples & Unpacking
t=(1,(2,3),4)
x,y,z=(10,20,30)
```

Sets:- {}

Unordered, No duplicates, Mutable, elements must be hashable (immutable types)

```
#Creation
s1={1,2,3}
s2=set([4,5])
empty_set=set() # a dict Not{}
#modification
s1.add(4)
s1.update([5,6])
s1.remove(2)
s1.discard(10)
s1.pop() # remove random item.
s1.clear() # empty the set.
#Set Operations
A={1,2,3}
B={4,5,6}
A|B   #union {1,2,3,4,5,6}
A&B   #intersection {3}
A-B   # difference {1,2}
A^B   # symmetric difference {1,2,4,5,6}
#Set Comparisons
A.issubset(B)
A.issuperset(B)
A.isdisjoint(B)
```

Dictionaries

Unordered, Mutable, Keys: unique & immutable, values can be any type.

```
#Creation
d1={'a':1,'b':2}
d2=dict(a=1,b=2)
d3=dict([('x',10),('y',20)])
empty_dict={}
```

...apsulation. ...urity & re... ...te-faces ... objec...

```
#Access & modify
d1['a']   # access
d1['a'] = 100  # modify
d1['c'] = 300  # Add
d1.get('z', 0)
#deletion
del d1['a']
d1.pop('b')
d1.clear()
#dictionary methods
d1.keys()
d1.values()
d1.items()
d1.update({'d': 4})
#Looping
    for k in d1:
        Print(k, d1[k])
    For k,v in d1.items():
        Print(k,v)
# Dictionary comprehension
squares = {x:x*x for x in range(1,6)}
#{1:1, 2:4 .... }
my_dict = {"name":"Alice", "age":25}
my_dict["city"] = "Bengaluru"
Print(my_dict)
#{"name":"Alice", "age":25, 'city':"Bengaluru"}
#adding multiple ele.
my_dict.update({"country": "India",
    "Profession":"Engineer"})
O/P:- {"name" : "Alice", "age":25,
"city":"Bengaluru', 'country': 'India',
'Profession': 'engineer'}
#update existing value
my_dict["age"] = 26
```

Pointer to existing

File handling:-

The process of performing operations like creating, opening, reading, writting & closing files in a program.

Basic file Operations:-

Python provides built-in fn. to handle files easily.

*Opening a file:-

```
file = open("filename.txt", "mode")
```

- Modes- specifies purpose of opening file.

Modes:-	Description
'r'	Read, error if file not found
'w'	write, creates new file or overwrites.
'a'	append, adds to end of file
'x'	Create, error if file exists
'b'	Binary mode ('rb', 'wb')
't'	Text mode

*Reading from a file:-

```
file = open("sample.txt", "r")
content = file.read() # reads entire file
line = file.readline()  # reads one line
lines = file.readlines() # reads all lines as list
file.close()
```

*Writing to a file:-

```
file = open("sample.txt","w")
file.write("Hello, world!")
file.close()
```

To overwrite a file use 'a' to append instead of overwrite.

```
file = open("sample.txt", "a")
file.write("\n New line added.")
file.close()
```

4. Using with stmt:-

Automatically handles file closing:-

```
with open ('sample.txt', 'r') as file:
    content = file.read()
```

5. Checking if file exists

```
import os
if os.path.exists ("sample.txt"):
    Print("File exists")
else:
    Print ("File doesnot exists.")
```

Extra functions:-

```
file.seek(0) # move cursor to start of file
file.tell() # Returns current cursor Position.
```

Common errors to avoid:-

* Not closing the file
* Reading from a file that doesn't exist (use try-except or check existence).
* Using wrong mode ("w" will erase data if file exists).

Modules

A (.py) file with fn, variables or clases to reuse.

```
import module_name
from module_name import Ham
import module_name as alias.
```

Packages

A folder containing modules & a special __init__.py file.

```
from my_package import module1
from my_package.module2 import fun_name.
```

Built in module:-

math - Math fn; os - file & path operatin
random - Random numbers;
datetime - Dates & times; sys - Pyt hon runtime settings

* External modules

Install using PIP:-

```
Pip install module_name
```

Exception Handling:-

Errors:- A serious issues that a program should not try to handle.
Ex:- syntax error, EOF error....

Exception - less severe than errors & can be handled by the program. They occur due to situation like 'invalid i/p', missig files....

Handling - way of overcoming these errors/exceptions.

Exception Handling - the mechanism used for handling multiple exceptions is called exception handling.

```
try:
    # Code that might raise an exception
except SomeException:
    # Code to handle the exception
else:
    # Code to run if no exception occurs
finally:
    # Code to run regardless of whether an exception occurs.
```

Exception names:-

ArithmeticError, ZeroDivisionError, OverflowError, AssertionError, KeyError, NameError, OSError....

Raise an Exception:- is done using raise keyword followed by an instance of exception class that we want to trigger. we can choose from built-in excep or define our own custom exceptions by

reduces code

getters &

inheriting from Built in exception.

```
    raise ExceptionType("Error Message")
def setAge():
    if age<0:
        raise ValueError("Age can't be -ve")
    Print ("Age set to", Age)
try:
    set(-5)
except ValueError as e:
    Print(e)
```

OOPs:-

Object Oriented programming lang. It supports:-

* class & Objects
* Inheritance
* Polymorphism
* Encapsulation
* Abstraction

class :- Blueprint of creation of Objects. Combination of properties & methods. of a thing, person, object or anything is called as class.

Human

- Name
- Gender
- Occupation

→ Common Properties

- speaks
- Dowork
- sleeps

→ Common actions methods.

Object:- An object is an instance of a class. In OOP a class defines a blueprint & an object is a specific instance.

```
class Human:                          ← initializing class using self & other Parameters.
    def __init__(self, n, o):
Properties  self.name = n
            self.occupation = o
Methods     def do_work(self):
                if self.occupation == "Tennisplayer":
                    Print(self.name, "is a Tennis player")
                elif self.occuption == "Actor":
                    Print(self.name, "is an actor")
            def speaks(self):
                Print(self.name, "says how are you?")
tom = Human("tomcurise", "actor")     → here self Para. given by default
tom.do_work()
tom.speaks()                          } instance of cls (or) objects
```

o/p:-

Tomcursie is an actor.
Tom cursie says how are you?

Inheritance:- Inheritance is a mechanism where one class (child class) acquire the properties & behaviors (method) of another class (Parent class). Benefits:- Code reusability, Extensibility, Readability.

```
class Vehicle:
    def general_usage(self):
        Print("General Usage")
class car(Vehicle):
    def __init__(self, wheels, has_roof):
        Print("I'm a Car")
        self.wheels = 4
        self.has_roof = True
    def specific_usage():
        self.general_usage()
        Print("specific_usage: commute to work, vacation")
```

```
class MotorCycle (Vehicle):
    def __init__(self, wheels,
                 has_roof):
        Print("I'm a motorcycle")
        self.wheels = 2
        self.has_roof = False
    def specific_usage(self):
        self.general_usage()
        print("specific use: racing,
              Road-trip")

C = Car()
C.specific_usage()

olp:- I'm a Car
      General usage
      specific usage: Commute to work,
                      vacation.
```

Multiple Inheritance:-

```
class Father():
    def skills(self):
        print("I enjoy gardening")
class Mother():
    def cooking(self):
        print("I love cooking")
class child(Father, Mother):
    def sports(self):
        print("I enjoy sports")

C = Child()
C.gskills()
C.cooking()
C.sports()
```

olp:- I enjoy garden.
I love cooking
I enjoy sports.

*super() - call parent methods

```
class Parent:
    def __init__(self):
        print("parent Constructor")
class child(Parent)
    def __init__(self):
        super().init__()
        Print("Child Constructor")
c = Child()
```

olp:- Parent Constructor
Child Constructor.

Polymorphism:- same method behaves de differently based on the object.

```
class Bird:
    def speak(self):
        print("Tweet")

class Parrot(Bird):
    def speak(self):
        print("squawk")

b = Bird()
P = Parrot()
b.speak()  # Tweet
P.speak()  # squawk
```

Abstraction:-

The process of hiding complex implementation details & showing only essential features to users. what to do but not how to do. It is achieved using abstract base classes (ABC).

```
from abc import ABC, abstractmethod
class Shape(ABC):
    @abstractmethod
    def area(self):
        Pass
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        return 3.14 * self.radius**2
circle = Circle(5)
Print(circle.area())  # 78.5
```

Encapsulation:-

The process of wrapping data (variables) and methods into a single unit (class) and restricting direct access to some of the object's components. Projects obj. internal state. Access via getter/setter methods. Achieved by making var. private using __ or __.

```
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance #Private var.
    def deposit(self, amount):
        self.__balance += amount
    def get_balance(self):
        return self.__balance
acc = BankAccount(1000)
acc.deposit(500)
Print(acc.get__balance()) # 1500
Print(acc.__balance()) #Attribute Error
```

* __balance is encapsulated (not accessed directly). Controlled via deposit() & get_balance().

Magic Methods:-

Special methods with double underscore (__name__) that allow customization of class behavior.

__init__	- Constructor
__str__	- stringRepresentation
__len__	- Length using len().
__add__	- Behavior of + operator.
__eq__	- equals (==) comporison

```
class Book:
    def __init__(self, title, pages):
        self.title = title
        self.pages = pages
    def __str__(self):
        return f"Book: {self.title}"
    def __len__(self):
        return self.pages
b = Book("Python Guide", 250)
Print(b) # Book: Python Guide
Print(len(b)) #250
```

* Generator, Iterator, Decorator
* List, set, Dictionary Comprehension
* Frozen sets

Iterators:-

An iterators is an object in python that allows traversal through a sequence (like lists, tuples or strings) using next() functions.

An iterator implement 2 methods:

* __iter__() → returns iterator object itself.
* __next__() → returns next item in the sequen

* Iterators allow efficient memory usage since they dont load all elements at once.
* Going through the elements one by one is also termed as iterators
* A for loop internally uses iterators for going through every element one by one.

```
a = ["hey", "bro", "youre", "awesome"]
itr = reversed(a)   #inbuilt method toget element
next(itr)                  fromlast.
next(itr)  └→# o/p:- 'awesome'
           →o/p: you're
```

Iterator using implemation using class:-

```
class Remotecontrol():
    def __init__(self):
        self.channels=["HBO","cnn","a","esp"]
        self.index=-1 # TV is off
    def __iter__(self):
        return self
    def __next__(self):
        self.index +=1
        if self.index==len(self.channels):
            raise StopIteration
        return self.channels[self.index]
r = Remotecontrol()
itr = iter(r)
print(next(itr))
print(next(itr))
print(next(itr))
print(next(itr))
```

o/p: HBO
cnn
a
esp

Ex:-
```
def remote_control_next():
    yield "cnn"
    yield "espn"

itr = remote_control_next()
next(itr)
next(itr)
```
} channels in my TV

o/p:-
```
cnn
espn
```

---

return :- stops fn. execution & returns a single final value.

yield :- Pauses fn. execution & returns a value, allowing resumption from same point.

example:- One-time Action (Instant result)

Think of return like ordering food at a restuarant. You place your order, the chef prepares everything, & the waiter brings full meal to you at once. Once meal is served, the kitchen forgets about your order. (return uses more memory)

yield - step-by-step Process (Lazy Evaluation) (low memory)

Think of yield like a vending mlc. You press a button, and it gives you one item at a time. You can keep pressing button to get more items, but the mlc doesn't hand you everything at once.

Imagine a Restaurant with a buffet & a waiter serving Instant meal.

Scenario	return(Instantmeal)	yield(Buffetstyle)
Process	servers full meal at once.	serves one dish at one
example	return gives final bill after eating	yield allows taking one dish at a time
Whentouse?	when you need everything upfront.	when you want to Processstepbystep.

When to use

* Use return when you need all results immediately (return - Once done it forgets all previous steps (like chef forgetting your order after serving)

* Use yield when handling large data or streams. (Pauses and resumes - The fn. remembers where it left off & continues).

Generator:-

A Generator is a special type of fn. that produces a sequence of values lazily, meaning it does not stores all values in memory at once. Instead of return, it uses yield to return values one at a time.

How Generators work :-

* Uses the yield keyword to return data without terminating the function.

* Remembers its state & resumes execution from where it last stopped.

* More memory efficient than fn. that use return especially for large datasets.

```
def my-generator():
    yield 1
    yield 2
    yield 3
gen = my_generator()  # creating a generator object
Print (next (gen))
Print (next (gen))
Print (next (gen))
```

o/p:- 1
2
3

∵ next() retrieves values one at a time instead of returning all at once.

```
def fib():
    a,b = 0,1
    while True:
        yield a
        a, b = b, a+b
for f in fib():
    if f > 50:
        break
    Print(f)
```

olp:-
0
1
1
2
3
5
8
13
21
34

*Benefits:-

- You don't need to define iter() & next() metho
- You don't need to raise stopIteration exception

List, set, Dictonary Comprehension:-

List Comprehension:-

When you need an ordered collection with dupli cate values. A way to create a list from an iterable.

[expression for item in iterable if condition]

Ex:-

```
numbers = [x for x in range[5)]
Print(numbers)   # olp:[0,1,2,3,4]
squares = [x**2 for x in range(5)]
Print(squares) # olp:- [0, 1, 4, 9, 16]
```

Set Comprehension:-

When you need unique values only, Used to create a set (removes duplicates automatically).

{expression for item in iterable if condition}

Ex:-

```
numbers = {x for x in range(5)}
Print(numbers)  # olp: {0,1,2,3,4} (unordered
unique_squares = {x**2 for i in[1,2,2,3,3,4,4]}
Print(unique_squares) # {1,4,9,16}
```

Dictionary Comprehension:- When you need key-value pairs used to create a dictionary in a concise way.

{key_expr : value_expr for item in iterable if condition}

Ex:-

```
square_dict = {x : x**2 for x in range(5)}
print(square_dict) # {0:0, 1:1, 2:4, 3:9, 4:16}
char_count = {char: ord(char) for char in "hello"}
print(char_count)
# {'h': 104, 'e': 101, 'l': 108, 'o': 111}
```

These are shortcuts to create lists, sets & dictionaries using a single line of code instead of loops.

They are used for:-

- Saves space (one-liner instead of multiple lines)
- faster execution than loops in many cases.
- Easy to read when used properly.

Sets & Frozensets :-

Set - A mutable (changeable) collection of unique elements. Un ordered - No specific order of elements. Mutable. can add or remove elements. No duplica-tes (Automatically removes duplicates).

Operation:- add, remove, update.

Ex:-

```
my_set = {1,2,3,4,4,5}
print(my_set)  # o/p: {1,2,3,4,5}
my_set.add(6)
print(my_set) # {1,2,3,4,5,6}
myset.updates([9,7,8])
print(my_set) # {1,2,3,4,5,6,9,7,8}
```

Frozen Set:-

An immutable version of set (unchangeable). There is no specific order (unordered) ; Cannot add or remove elements after creation (Immutable) ; No duplicates.

Operation :- Only read mode.

Ex:-

```
my_fs = frozenset([1,2,3,3,4,5])
print(my_fs)  # o/p: {frozenset({1,2,3,4,5})
my_fs.add(6) # Attribute error:- frozenset hobject has no attri. "add"
```