S.NO	Index
1	Introduction to Python
2	Variable & data types
3	Comments, escape sequence & print statement.
4	operators
5	Control flow & loops
6	Strings
7	Functions in python.
8	Data structures
9	OOPs in Python
10	Dunder methods
11	Decorators
12	Args & Kwargs
13	List, Dict & Sets Comprehensions
14	Lambda function
15	Maps & filters.

Introduction to Python

It is the process of giving instructions to a computer to perform specific tasks.

→ writing code in a programming language that computer can understand & execute.

Python

↓

High level, Interpreted language, Simplicity & Redability.

↓

Widely used in : Web Dev [Django, flask]

↓

D.S & M.L [Pandas, Numpy, Tensorflow]

↓

Automation & Scripting.

↓

Game dev [Pygame]

↓

Large Community & Extensive library.

Extension :-

first_Program.Py.

Input :- Print ("Hello world");

output :- Hello world.

→ Print() is a built in function used to display output.

→ Python code is executed line by line.

→ understanding python Syntax & Basics :-

① Syntax rule :- Indentation :-

→ Python use indentation (spaces or tabs) to define block of code.

Ex:- if 5 > 2 :
    Print ("5 is greater than 2");
    # spaces before print are called Indentation

→ White space :-

→ Sensitive to white space.

→ Consistent indentation to avoid errors.

→ use 4 spaces for indentation.

→ Statement :-

→ Each line of code is a statement.

→ We can write multiple statements on one line using (;) But this is not recommended.

→ Comments :-

use [#] for Single line.

use ['''] or [" " "] for multi line comments.

② Variables & Data types :-

→ used to store Data that can be used and manipulated in a program.

→ Variable is created when you assign a value to it using (=) operator.

Ex :-
```
name = "Alice"
age = 25
height = 5.6.
```

→ Naming rule :-

→ Can contains a letters, numbers & underscore.

→ must start with letter or underscore.

→ Case sensitive.

→ Avoid using Python keywords as variable names.

[eg :- print, if]

→ Data types :- Supports several built in Data types.

① Integer (int) ⇒ whole numbers (Eg: 10, -5)

② Floats (float) ⇒ Decimal number (Eg: 3.14)

③ String (Str) ⇒ Data enclosed in quotes (eg: "Hello").

④ Booleans (bool) ⇒ Represents True or false.

⑤ Lists ⇒ ordered, mutable collections (eg: [1, 2, 3])

⑥ Tuples ⇒ ordered, immutable collections. (eg :- (1, 2, 3))

⑦ sets ⇒ unordered collections of unique elements. (eg :- {1, 2, 3})

⑧ Dictonaries ⇒ Key-Value pairs (eg :- {"name" : "Alice", "age" : 25})

→ Checking Data types :-

Use the type() function to check data types of a variable.

⇒ print (type (10)) # output : <class 'int'>

⇒ print (type ("Hello")) #output :- <class 'str'>

→ Type Casting :-

→ It is the process to convert one data type to another.

→ Provides built in functions for type casting.

- int() :- Converts to integer.
- float() :- Converts to float.
- str() :- Converts to string
- bool() :- Converts to boolean.

Ex :-

```
# Convert string to integer.
num_str = "10"
num_int = int(num_str)
print(num_int)    # output - 10

# Convert Integer to string.
num = 25
num_str = str(num)
print(num_str)    # output - "25"

# Convert float to integer.
pi = 3.14
pi_int = int(pi)
print(pi_int)    # output - 3
```

→ Taking user input :-

using input() function.

Ex:- name = input("Enter your name = ");

→ By defaullt input() returns a string. we can convert it to other datatypes as needed.

Ex:- age = int(input("Enter age = "))

print(f "Hello {name}, you are {age} years old.")

(4) Comments, Escape Sequences & Print statements..

① Single line comment (#)

② Multiline comments (''' or """)

→ Escape Sequence :-

\n → new line

\t → Tab

\\ → Backslash

\" → Double quote.

\' → Single quote.

③ Print Statement :-

Print ("Hello", "world", sep = ",", end = "!\n")

⑤ operators

① Arithmatic operator :- (+, −, *, /, %, **, //)

Ex:- print (10 + 5) # output : 15
print (10 ** 2) # output : 100

② Comparision operators :-

(==, !=, >, <, >=, <=)

Ex:- print (10 > 5) # True.
print (10 == 5) # false.

③ Logical operator :- (and, or, not).

Ex:- print (True and false) # false.
print (True or false) # True.
print (not true) # false.

④ Assignment operators :-

(=, +=, -=, *=, /=, %=, **=, //=)

Ex:- x = 10
x += 5 # x = x+5
print (x) # 15

⑤ Membership operators :-

in , not in.

Ex:- fruits = ["apple", "banana", "cherry"].

print ("banana" in fruits) # True.

⑥ Identity operators :-

is , is not.

Ex:- x = 10

y = 10

print (x is y) # True.

⑥ Control flow & loops :-

→ Allow you to execute code based on certain conditions.

→ Python uses if, elif and else for decision making.

→ Syntax :-

```
if cond1 :
    # code
elif cond2 :
    # code
else :
    # code.
```

Example :-

```
age = 18;
if age < 18 :
    print ("you are minor")
elif age == 18 :
    print ("you just become adult")
else :
    print ("you are an adult");
```

→ Match Case (Python 3.10+)

→ New feature introduced in Python 3.10.

→ Simplifies complex conditional logic.

→ Syntax :-

```
match value :
    Case Pattern 1 :
        # Code
    Case Pattern 2 :
        # Code
    Case _ :
        # Default.
```

Ex :- Status = 404.

```
match status :
    Case 200 :
        print ("Success!")
    Case 404 :
        print ("Not found")
    Case _ :
        print ("unknown status")
```

Loops :-

① **for loop** :- used to iterate over a Sequence (list, string, range)

→ Execute a block of code repeatedly for each item in a Sequence.

Syntax :- for item in Sequence :
    #code.

Ex :- fruits = ["apple", "banana", "cherry"]
for fruit in fruits :
    print (fruit)

using range () :-

generates Sequence of numbers.

Ex :- for i in range (5) :
    print (i) # 0, 1, 2, 3, 4.

(2) While loop :-

```
While condition :
    # Code.
```

Ex :-
```
count = 0
While count < 5 :
    print (count)
    count += 1.
```

→ Infinite loop :-

```
While True :
    print ("run forever");
```

→ Break :-

This statement is used to exit a loop prematurely

Ex :-
```
for i in range (10) :
    if i == 5:
        break;
    print (i)  # 0, 1, 2, 3, 4.
```

→ Continue :-

This statement skips the rest of the code in the current iteration and moves to the next iteration

Ex:-
```
for i in range(5):
    if i == 2:
        continue
    print(i)        # 0, 1, 3, 4
```

→ Pass :-

The pass is a placeholder, that does nothing. It is used when syntax requires a statement but no action is needed.

Ex:-
```
for i in range(5):
    if i == 3:
        pass        # DO nothing
    print(i)        # 0, 1, 2, 3, 4.
```

⑦ Strings.

→ A string is a sequence of a characters enclosed with either single quotes (' '), double quotes (" ") or triple quote (''' ''' or """ """).

Eg:- a = "Hello".

→ String Indexing :-

```
text = "Python"
Print (text[0])  # P
print (text[1])  # Y
Print (text[-1])  # n. (last char)
```

→ String Slicing :-

Extract parts of a string using slicing

```
text = "Hello, Python!"
Print (text[0:5])   # Hello
Print (text[:5])    # Hello.
Print (text[7:])    # Python!
Print (text[::2])   # Hlo Pto!
```

→ String Methods :-

```
text = "Hello World"
print (text.upper())   # "HELLO WORLD"
print (text.lower())   # "hello world"
print (text.strip())   # "hello world"
print (text.replace ("world", "python"))
                       # "hello python"
print (text.split())   # ['hello', 'world'].
```

→ String formatting :-

```
name = "John"
age = 25.
#using format()
print ("my name is {} and iam {} years old." :
      .format (name, age));
#using f-strings (python 3.6+)
print (f "my name is {name} and I am
      {age} years old.").
```

→ Multiline String :-

```
message = '''
Hello,
This is a multi line string example.
Goodbye!
'''
Print (message)
```

Functions in Python.

→ user definition function :-

```
def hello():
    print("This is a hello function").
```

→ function calling :-

```
hello()
```

→ Block of reusable . we have to call it.

② Parameters & Arguments :-

Variable listed inside the definition.

ex:- def Sum (a, b):

print(f"the Sum of the numbers is {a+b}"):.

Sum (12, 45) → Arguments. 24

Sum (45, 45) ↑ 90.

→ positional Arguments

↳ 1st example.

→ 3 types :-

ex:- def add (a, b):
    return a+b

② def introduce (name, age):
    print (f"I am {name} and I am {age} years old")

introduce (age = 25, name = "Suraj")

→ Keyword Argument

③ def greet (name = "guest"):
    print (f"Hello, {name}!")

greet()

greet ("BOB")

→ Default Argument

→ Palindrome function :-

```
def Palindrome(st):
    rev = ""
    for i in range(len(st), -1, -1):
        rev = rev + st[i]

    if rev == st:
        print(f"{st} is a Palindrome")

    else:
        print(f"{st} is not a Palindrome")
```

→ return :-

①
```
def hello():
    print("Hello how are you.");
```

②
```
def hello():
    return "Hello how are you"
```

→ Value returns back to call function.

ex:- print(hello())

Data Structures :-

→ Represent data in a structured way.

→ Save multiple data in a single variable.

→ 4 types.
- → list → []
- → Tuple → ()
- → Dictionary
- → Set.

① List :-

→ Mutable → Values can be changed after creation

→ Duplicates → used to store multiple values. So duplicates means same value occuring multiple times.

→ Ordered → It maintains ordered data structure maintains the sequence of elements they were inserted. We can access elements using their position. (index)

→ Heterogenous → It have heterogenous nature that means we can have multiple datatypes inside the list.

→ Syntax :-

a = [12, 13, 14, 15 ..., n]

fruits = ["Apple", "Banana", "grapes"]

→ Accessing :-

a[0] => 12.

a[1] => 13

→ List traversing & method :-

→ Looped using the index values and directly.

ex :-

```
a = [12, 13, 14, 15, 16, 17]
for i in range (len (a)):
    print (a[i])
```

✓

→ list access using index.

→ ② directly on values.

```
for i in a :
    print (i).
```

✗

→ List methods :-

methods are defined in classes. (Or) Also a kind of function.

→ print (dir (list)).

→ help (list)

⇒ num = [5, 2, 9, 1, 5, 6]

num. append (10) # Adds 10 at the end.

num. insert (2, 15) # Insert 15 at index 2.

num. extend ([20, 25, 30]) # Add multiple elements at the end.

num. remove (5) # Remove first occurance of 5.

popped_item = number. pop (3) # remove the element at index 3. and store

② Tuple :-

a = (1, 2, 3, 4, 5).

print (type (a)).

① Immutable → we cannot change the value of tuple.

② Duplicates → we can have duplicate values in tuple.

③ Ordered → Set are ordered and you can access them through indexing.

④ Hetrogenous → sets also have hetrogenous nature and can have different types of data structures in tuple.

→ Tuple traversal & methods :-

• Same as list (traversing) using for loop.

• Method of tuples are :-

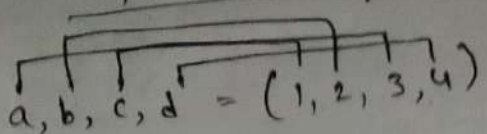
t = (1, 2, 3, 4, 5, 6, 7).

① index = a.index (5)
Print (index).

② count_5 = t.count (5).
Print (count_5).

} [only 2 methods.]

→ Tuple unpacking :-



Sets :-

S = {1, 2, 3, 4, 5}

- Mutable → Set are mutable. we can change the values.
- Duplicates → cannot have a duplicate values in Sets.
- unordered → we cannot access them through index values.
- Heterogenous → Set is semi-hetrogenous it can store some data types like string, numbers, tuples but not everything.

→ How Set store values :-

① Each value in a set is hashed using hash function.

② the hash is used as an index to store the elements in memory.

③ Since hashing does not maintain order, sets are unordered.

→ we cannot traverse on set.

→ Set methods :-

```
S = {1, 2, 3}
S.add(4) # Add elements to the set
S.~~remove~~(5) # remove 5 (No error if not found)
  discard.
```

→ Some of the operations performed between two Sets

A = {1, 2, 3}

B = {3, 4, 5}

→ union_set = A.union(B) # {1, 2, 3, 4, 5}

→ intersection_set = A.intersection(B) # 3

→ difference_set = A.difference(B) # {1, 2}

→ Symmetric_diff = A.symmetric_difference(B)

{1, 2, 4, 5}

Dictionary :-

```
d = {
    1 : "hello",
    2 : 56
    }
```

→ mutable → we can change, add & remove key value pairs after creation.

→ Duplicates → key must be unique but value can has duplicates.

→ order → Dictionary follows insertion order.

→ Hetrogeneous → Can store different types of key & values, like integers, strings, lists or even another dictionary.

→ Syntax :-

```
Student = {"name" : "John",
           "age" : 20
          }
print(student ["name"])
```

Exception handling :-

Errors :- occur due to mistakes in the code that prevent it from running. This can be Syntax errors or logical errors.

→ Exceptions :- This is an unexpected event or errors that occurs during the execution of a program, which disrupts the normal flow of program.

ex:- print (10/0) # raise zeroDivisionError

(1) Try → Wrap the block of code that might cause an exception.

(2) except → Handle the exception if it occurs.

(3) else → Run code only if no exception occurs.

(4) finally → Run code no matter what, whether there's an exception or not.

(5) raise → Manually throw an exception.

ex:- try:

    //statement:

except _____ : Exception as err:

    //statement

else:

    //statement

File handling :-

→ It means creating, Reading, updating & Deleting (CRUD) operations that we can perform in files.

→ open() to open the file.

Ex:- p = open('main.py')
print(p.read())

Modes :-

'r' → read

'w' → write

'a' → Append - adds to end of file.

'x' → Create - creates a new file, fails if it exists.

→ w

r = open("Superman.txt", 'w')
r.write("Hello") (append)
r.close().

OOPs

→ OOPs is a programming paridam based on the concept of 'objects', which can contain data (attributes) and code (methods).

→ reusable, multiple execution, Security, management system.

① Classes :- Blue print or template

Syntax :- class classname :
    // code.

→ class Factory :
    a = 12 → # Variable inside class. [Attribute].

    def hello(self):
        print("Hello") → # Method.

print(Factory().a)

print(Factory().hello())

- Objects → Access from class [All instances]

→ Syntax :- obj = Factory().

print (obj.a)

obj.hello()

Multiple objects → obj2 = Factory()

obj3 = Factory()

→ ~~Self Keyword~~ :-

→ Constructor :- Accepting parameters.

class factory: → dunder method

def __init__(self, material, zips, pockets):

self.material = material

self.zips = zips

self.pocket = pocket

reebok

= Fatory ("leather", 3, 2)

→ Self :- use for Capture location.

→ Attributes & methods :-

① What are attributes :

L> class attribute : Variable created inside the class

L> Instance attribute : Attribute created using an instance like self.name, self.age etc.

Ex:-

```
class Car:
    wheels = 4   # class attribute

    def __init__(self, color):
        self.color = color  # instance attribute

    def show(self): #instance method
        print("Hello").
```

② Types of methods :-

• Instance method :- An instance method works with instance object of the class. This method can access and modify instance attributes.

```
class myclass:
    def instance_method(self):
        print("This is an instance method")
```

→ Class method :- This method works with the class itself it will not target the object. We have to use @classmethod decorator for creating the class method and it takes cls as their first parameter.

```
class Myclass:
    @classmethod
    def class_method(cls):
        Print("This is a class method").
```

→ Static method :- It is a regular function placed inside a class.
→ uses a decorator @staticmethod

```
class Myclass:
    @staticmethod
    def static_method():
        print("this is a static method")
```

Inheritance

Syntax :- Class factory :

a = 12 # Attribute inside factory.

def hello(self):

print("method inside factory")

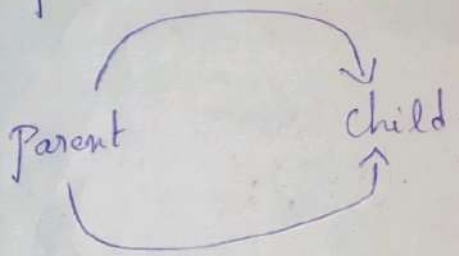
Inherit. Class factorypune(factory) : #child class.

pass

obj = factory ~~number~~()

obj = factorypune().

→ In general terms inheritance means property or any possession that comes to an heir.



→ It works between classes.

→ Inheritance allows a class (child class) to inherit properties and behaviors (attributes & methods) from another class (Parent class).

→ Benefits :- ① Code reusability
② Organized structure.
③ Easy to maintain & extend.

→ Syntax :-

```
class Parent :
    def Speak (self):
        print ("I can speak!")

class child (Parent):
    pass.
```

→ Constructor in inheritance :-

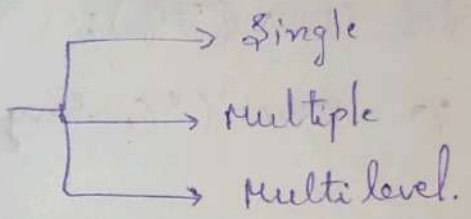
→ Constructor function of parent class will work for the child class as well.

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

class child (Parent):
    def display (self):
        print (f"my name is {self.name}")
```

→ Super() : This function targets the parent class.

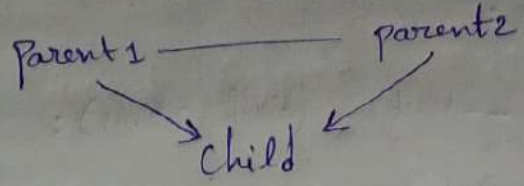
```
class parent:
    def __init__(self, name):
        self.name = name

class child(parent):
    def __init__(self, name, age):
        super().__init__(name)
        self.age = age
    def display(self):
        print(f"my name {self.name} and
I am {self.age} years old").
```

→ Types :



① Single Inheritance : All the inheritance we saw above was single level.

② Multiple inheritance :- 2 Parents 1 child.



→ Child class will inherits both parent classes.

→ MRO (Multimethod Resolution Order) followed by Python

Ex:-

```
class Father:
    def Skills(self):
        print("Coding")

class Mother:
    def Skills(self):
        print("Cooking")

class child(Father, mother):
    def show(self):
        Print("I have multiple
              Skills")
```

③ →

③ Multi level inheritance :-

→ grand parent class → parent class → child class.

```
class Grandparent :
    def heritage (self):
        print ("Heritage from grand parents")

class parent (Grandparent):
    pass

class child (parent):
    pass.
```

Polymorphism

→ One name but different forms or works.

→ Python doesnot support method overloading.

→ Method overloading means having same name method inside a class but parameters will be different but in Python the latest definition will overwrite the previous one.

→ Method overriding :-

• child class overrides a method of the parent class, and Python decides at runtime which method to call based on object type.

Ex:-

```
class Animal:
    def show(self):
        print("Animal makes a Sound")

Class Dog(Animal):
    def Sound(self):
        print("Dog barks")
```

→ Duck typing :-

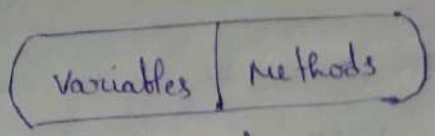
"If it walks like a duck and quacks like a Duck, it must be a duck."

Ex:-

```
Class Duck:
    def talk(self):
        print("Quack")

class Human:
    def talk(self):
        Print("Hello")
```

Encapsulation :-

→ It means putting data (variable) & code (function) together in one place. inside the class



capsule.

→ Hiding the internal details of how things work and only showing what is needed.

- ↳ Keeps data safe from being changed by mistake.
- ↳ Makes code clean & easy to use
- ↳ gives control over what others can access or change.

→ Access modifiers :-

- Means how we give access of our attributes and methods to the object or inherited class.

① Public attribute & methods :-

Till now we have created are public means the inherited classes and objects can access them no matter what.

• Protected Attributes & methods :-

→ Created using a single underscore (_).
→ Still we can access from outside of class.
→ ~~used~~ uses a naming convention to tell developers.

• Private Attributes & methods :-

→ It cannot accessed from outside the class.
→ Access only inside the class where it is defined.
→ we use two underscores (__) before the name to make it private.

Ex :-

```
class Demo:
    def __init__(self):
        self.name = "Public member"      # public
        self._age = 21                   # protected
        self.__salary = 50,000           # private

    def show(self):
        print("Inside the class")
        print("Public :", self.name)
        print("protected :", self._age)
        print("private :", self.__salary)
```

Abstraction :-

→ It is used to simplify complex system by focusing on essential features and hiding unnecessary details.

→ It is define a common interface for different subclasses.

⟶ Abstract classes and methods :-

→ Abstract classes are classes that contains one or more abstract methods.

→ A method that is defined but not implemented in the abstract class. Subclasses must provide the implementation.

Ex:-

```
From abc import ABC, abstractmethod

class Animal (ABC):
    @abstractmethod
    def make_sound (self):
        pass

class Dog (Animal):
    def make_sound (self):
        print ("Dog says woof!")
```

```
Clars Cat (Animal):
    def make_sound (self):
        Print ("Cat Says meow!")
```

Dunder methods

→ Dunder methods are special method in Python that starts and end with double underscore.

Ex:- __init__ , __str__ , __add__ etc.

→ They automatically get called when you perform certain actions on object.

→ It helps in :-

- ↳ Customize behavior of your class.
- ↳ Make your class object behave like built in data types like string, list etc.

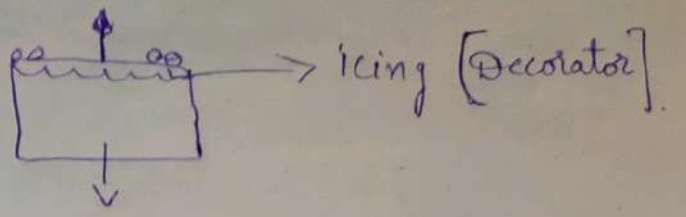
Ex:-

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

P = Person("Suraj")
print(P.name)
```

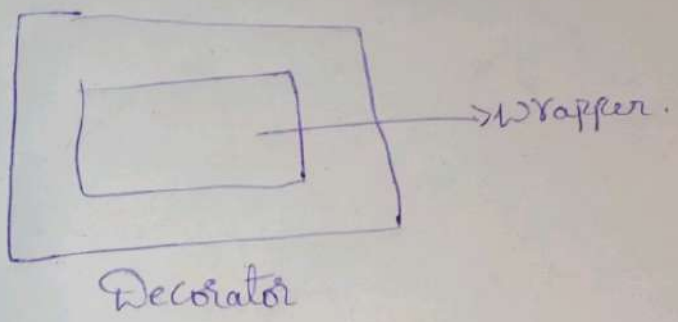
Decorators :-

→ A Decorator is just a function that modifies another function without changing its actual code.

for example :-



→ Makes it better, prettier.



Ex:-

```
def my_decorator (func):
    def wrapper():
        Print ("something before the function
                runs")
        func()
        print ("Something after the function
                runs")
    return wrapper
```

```
@my_decorator
def say_hello():
    print("Hello")

say_hello()
```

Args & Kwargs :-

→ Args use for capturing multiple Arguments.

Ex:-
```
def addition (*args):
    sum = 0
    for i in args:
        sum = sum + i
    print (sum)
addition (12, 13, 14, 15)
```

→ output :-

→ Kwargs → Keyword Arguments.

Ex:-
```
def information (**kwargs):
    print ("your information \n")
    for i in range kwargs:
        print (f" {i} : {kwargs[i]}")

information (name = "Suraj", age = "19", designation =
"Data Scientist")
```

List & Dictionary & Set Comprehension :-

→ Avoiding multiple line of codes.

Ex:- labels = ["Even" if x % 2 == 0 else "odd" for x in range(5)]

② evens = {x : x * x for x in range(10) if x % 2 == 0}

③ unique_even_squares = {x * x for x in range(10) if x % 2 == 0}

Lambda function :-

→ A lambda function is an anonymous, inline function defined using the lambda keyword.

→ It is often used for short, simple functions that are used only once or temporarily.

→ We can have multiple arguments but there will be only one expression.

Ex :- Square = lambda x : x ** 2

print (square(4)) # 16.

② check_even = lambda x : "Even" if x % 2 == 0 else "odd"

print (check_even (7)) # odd.

Map & Filter :-

→ Map is used for applying a function to multiple items.

Ex:- numbers = [1, 2, 3, 4]

doubled = map (lambda x : x * 2, numbers)

print (list(doubled))

→ use map() when you want to transform every item in a list.

→ Filter :-

→ Filter as the name suggest is used to filter out the stuff.

→ Takes a list (or other sequence)

→ checks each item using a function (a test)

→ Keeps only the items that pass the test (return True)

Ex:-

```
numbers = [1, 2, 3, 4, 5]
evens = filter(lambda x: x % 2 == 0,
               numbers)
print(list(evens))  # [2,4]
```