write(): writes the data into the file.

- Data must be a string
- Returns the number of characters written in the file.

writelines(): write an iterable data into file.
(list, tuple or dictionary)

- Data must has string as its elements
- Returns none.

WRITING INTO FILES

- To write we need to use "w" mode.
- Syntax: file = open("demo.txt","w")

(or)

with open("demo.txt","w") as file:

- Functions to perform write operation
  - * write()
  - * writelines()

Examples write(), writelines()

```
import os
os.chdir(r"path")
Print(os.getcwd())
with open("example.txt","a") as file:          (or) "w"
    print(file.write("tomorrow is holiday\n"))
    file.writelines(["apple\n","is good for health\n","orange\n"])
with open("example.txt","r+") as file:
    file.write("all the best")
    for line in file:
        print(line)
```

tell() and seek()

- tell() - returns the current position of the cursor in the file.
- seek(pos) - navigates to the specified position.

Handling CSV files

- CSV (Comma Seperated Values) is a simple file format used to store tabular data, such as a spreadsheet or database.
- A CSV file stores tabular data (numbers and text) in plain text.
- Each line of the file is data record.
- Each record consists of one or more fields, seperated by commas.

For working CSV files in python, there is an inbuilt module named CSV.

Reading CSV files

There are 2 methods.

1. csv.reader(csvfile)
2. csv.DictReader(csv_file)

```
import csv.
#using reader()
with open(csv_file,"r") as file:
    rows = csv.reader(file)
```

rows → iterator object which holds each data record in the form of python list

using DictReader()

```
with open(filename, 'r') as csv_file:
    rows = csv.DictReader(csv_file)
```

rows → iterator object which holds each data record in the form of python dictionary.

Writing into csv files.

There are 2 methods,

1. csv.writer(csv_file)
2. csv.DictWriter(csv_file, fieldnames)

Note : Below are some supporting methods to write data into csv file,

1. writer_obj.writerow(): writes a single data into csv file. Data can be a list or dictionary.
2. writer_obj.writerows(): writes multiple data into csv file. Data should be list of iterables.
3. writer_obj.writeheader(): writes header in the file using the fieldnames specified.

```
import csv

Using writer():
with open(file_name, "w") as csv_file:
    csv_writer = csv.writer(csv_file)
    csv_writer.writerow([data])

Using DictWriter():
with open(file_name, 'w') as csv_file:
    csv_writer = csv.DictWriter(csv_file, [field_names])
    csv_writer.writeheader()
    csv_writer.writerow({'name': "xyz", 'age': 10})
```

writerows():

```
data = [('apple', 'google', 'yahoo'), ('microsoft', 'netflix'
                                       , 'gmail')]
with open(file_name, 'w') as csv_file:
    csv_writer = csv.writer(csv_file)
    csv_writer.writerows(data)
```

Example :-

```
import os
import csv
path = r"abcxyz"
with open(path) as file:
        obj = csv.reader(file)
        print(obj)
        for data in obj:
            print(data)

with open(path) as file:
        obj = csv.DictReader(file)
        print(obj)
        for data in obj:
            print(data)
```

writer(), writerow(), writerows()

```
Path1 = r"abcxyz"
with open(path1, "a", newline="") as file:
        w_obj = csv.writer(file)
        w_obj.writerow(["EmployeeName", "ID"])
        w_obj.writerow(["John", 256])
```

```
data = ["Apple", 456] ; ["Orange", 555]
w_obj.writerows (data)
```

default standard - "\r\n"

```
DictWriter()
os.chdir(r"abcxyz")
with open ("data.csv", "w", newline="") as file:
    obj = csv.DictWriter(file, ["en", "salary"])
    obj = writeheader()
    obj.writerow({'en': "John", "salary": 50})
    obj.writerows([{"en": "Bhavya", "salary":
        500000}, {"en": "santhosh", "salary":
        1000000}])
```

(or)

```
with open ("shashank.csv", "w", newline="") as file:
```

↳ Can create new file and we can write.

MID ASSESMENT.

1) write a program to print only even lines in a file (consider filename as Sample.txt)

```
import os
from itertools import islice.
from Collections import defaultdict.

OS.chdir(r"C:path")
def even_lines():
    with open("Sample.txt") as file:
        for line_no, line in enumerate(file):
            if line_no % 2 == 1:
                print(line)
even_lines()
```

2) write a program to print the nth fibonacci number.

```
n=10
a,b =0,1
for i in range(n-1):
    C=a+b
    a,b=b,c
Print(a)
```

3) write a function that returns the nth line and last n lines from a file.

```
os.chdir(r"C:path")
def n_lines(n):
    with open("sample.txt")as file:
```

```
    count=0
    for line in file:
        count+=1
with open("sample.txt") as file:
    for line_no, line in enumerate(file):
        if count-n <= line_no < count:
            print(line)
n_lines(3)
```

(or)

```
def n_lines1(n):
    with open("sample.txt") as file:
        count = 0
        for line in file:
            count+=1
        file.seek(0)
        lines = islice(file, count-n, count)
        return(list(lines))
Print(n_lines1(3))
```

b) Write a program to print longest and non repeated word in the sentence, "

```
s = "See and saw went to see a sea"
l = s.split()
l1=[]
for i in l:
    if l.count(i)==1:
        l1.append(i)
Print(sorted(l1)[-1])
```

(or)

```
l1=[i for i in l if l.count(i)==1]
print(sorted(l1)[-1])
```

5) Sort the dictionary based on the last character of the key.

```
Prices = {'ACME':45.23, 'AAPL':612.78, 'IBM':205.55,
          'HPQ':37.20, 'FB':10.75}
print(dict(sorted(prices.items(), key=lambda
          item: item[0][-1])))
```

6) Build a list with only even length string using filter function.

```
names = ['apple', 'google', 'Yahoo', 'facebook', 'Yelp'
         'flipkart', 'gmail', 'instagram', 'microsoft']
print(list(filter(lambda i: len(i)%2==0, names)))
```

7) write a python program to return a list of elements raised to the power of their indices using enumerate class.

```
numbers=[32,65,39,8,1]
l=[]
for i,j in enumerate(numbers):
    i.append(j**i)
print(i)
```

8) Write a program to create a dictionary with word and the number of occurances of word in the string without using inbuilt method.

```
Sentence = "hello world welcome to python hello
            hi hello hello"
l = Sentence.split()
```

```
d={}
for i in l.:
    c=0
    for j in l:
        if i==j :
            ct=1
    d[i]=c
print(d)
```

9) write a python program to Sum the Squares of the first 20 natural numbers.

```
Sum_=0
for i in range(21):
    Sum+= i**2
Print(Sum_)
```

10) write a dictionary comprehension to Create a dictionary with word as its Key and if the # word is of numeric type reverse it else add the word as it is in the Value,

```
S. '12 plus 18 equals to 30"
print {i:1[::-1]if i.isdigit()else ifor i in s.split()}
```

Part B

) write a function to group anagrams which are of Odd length.

```
words = ["silent", "sea", "case", "listen", "Pea", "ape"
         "fare", "fear"]
def anagram(words):
    d = defaultdict(list)
    for word in words:
        if len(word) % 2 == 1:
            key = "".join(sorted(word))
            d[key].append(word)
    return d.
print(anagram(words))
```

2) write a program to print the following pattern.

```
* * * * *
 * * * *
  * * *
   * *
    *
```

```
for i in range(5, 0, -1):
    print(f'{" " * i:^10}')
```

3) write a program to get the following output

```
S = 'AABBCCCDAACD"
C = 1
res = ""
for i in range(len(s)-1):
    if s[i] == s[i+1]:
        C+=1
```

```
else:
    res += str(c) + s[i]
    c = 1
print(res)
```

4) write a program to get the following output from the list

```
l = [1, 2, 3, 4, 5, 6, 7, 8, 9]
res = []
for i, j in enumerate(l):
    if i % 2 == 0:
        res.append(j)
    else:
        res.append(j)
        print(res)
        res = []
if len(l) % 2 == 1:
    print(res)
```

5) Find all max numbers from the below list.

```
numbers = [1, 2, 3, 4, 0, 3, 2, 4, 2, 1, 0, 4]
l = list(sorted(numbers))
Print([i for i in l if i == l[-1]])
```

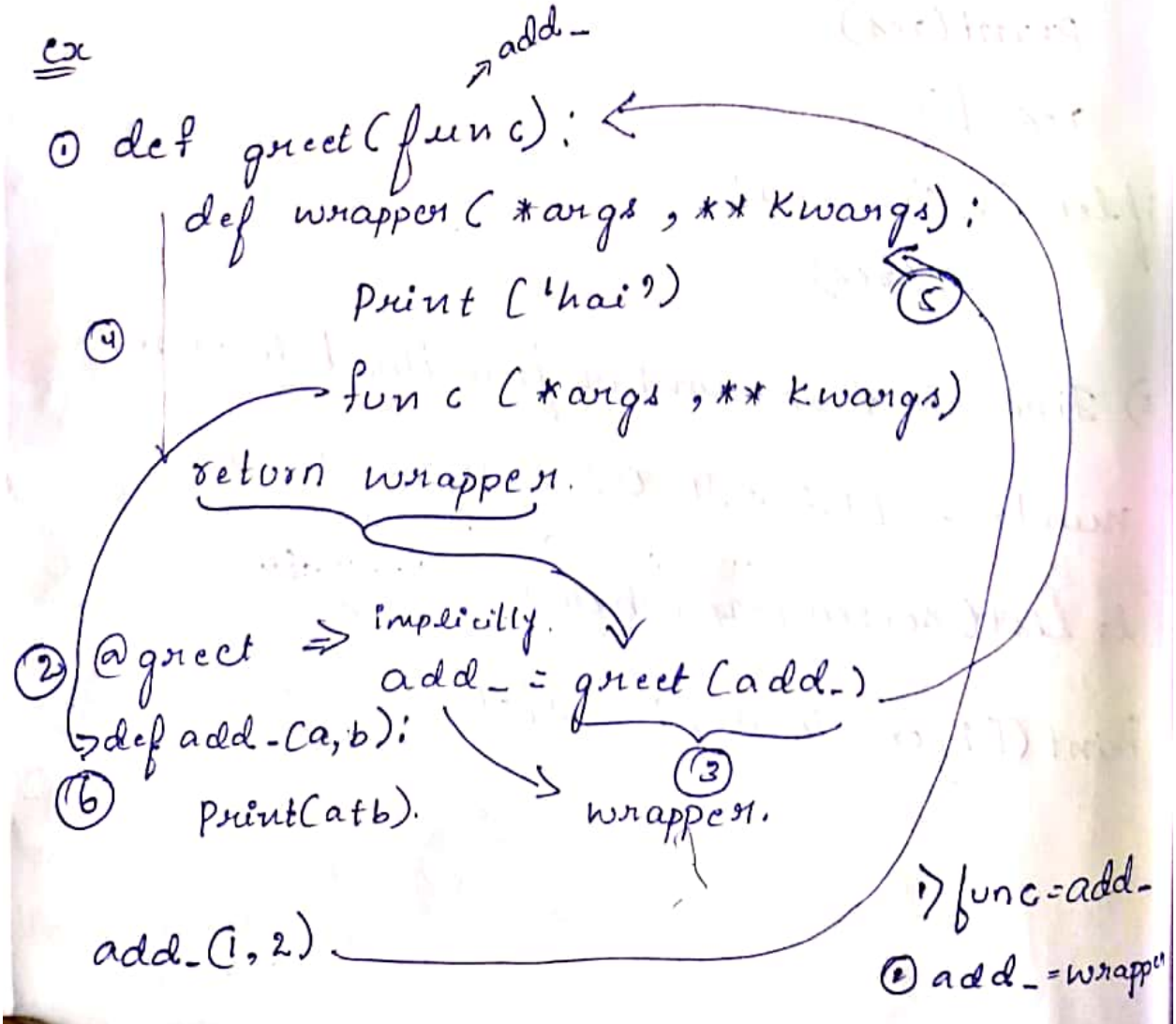
FIRST CLASS OBJECT

> First class objects are the one which is treated as an other object in python like strings, lists, dicts etc.

> You can pass a function to another function, You can return a function from another function, just like any other functions.

> A decorator is a function, which takes another function as an argument, adds some extra functionality, and returns another function without altering the source code of original function.

ex



Decorators overview

- A decorator takes in a function, adds some functionality and returns it.
- This is also called metaprogramming because a part of the program tries to modify another part of the program at compile time.
- A single decorator can decorate any number of functions.

Types

- Built in decorators : @classmethod, @staticmeth
- user defined decorator : created by users.

General Structure of a decorator

```
def outer (func):
    def inner (*args, **kwargs):
        func (*args, **kwargs)
    return inner
```

* This process of calling a function using some other name or variable is called as monkey patching.

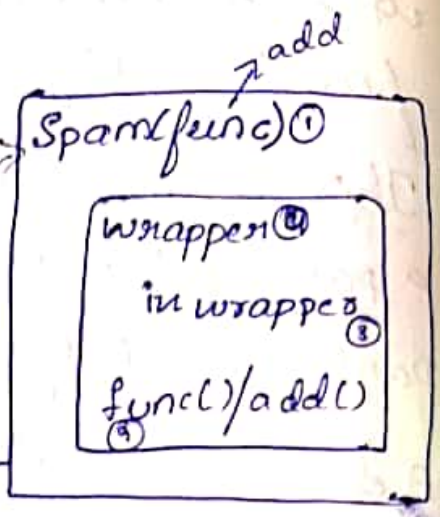
Note :- when a function is decorated with decorator function (@decoratorfunction) There will be 2 major changes happening

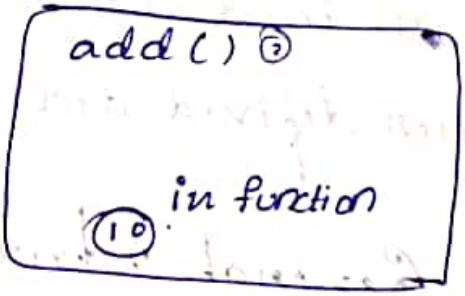
1) The parameter of outer function (func) will start pointing to main function.
2) The ~~main~~ variable pointing to main function starts pointing to wrapper function.

Working procedure of a Decorator.

```
def spam(func):
    def wrapper():
        print("in wrapper")
    func()
    return wrapper.

@spam
def add():
    print("in function")
add()
```





② → @spam → add = spam(add) // add-wrapper ⑥

⑦ → add()

10

1) Write a decorator function to log a message (function name) before executing any function.

Note __name__ → it is the magic method to get a function name,

dundeer method, which returns name of any Programmable entity attached to it.

logging function name decorator.

```
def log_(func):
    def wrapper(*args, **kwargs):
        print("function name:", func.__name__)
        res = func(*args, **kwargs)
        return res.
    return wrapper.

@log_    # multiply = log_(multiply)
def multiply(a,b):
    return a*b

print(multiply(7,3))
```

2) write a decorator function to input 5 seconds of delay before executing any function.

```
# delay decorator.
import time
def delay(func):
    def wrapper(*args, **kwargs):
        time.sleep(5)
        func(*args, **kwargs)

    return wrapper
@delay    # push = delay(push)
def push(element):
    l=[]
    l.append(element)
    print(l)
push("hello")
```

3) write a decorator function that executes any function for 3 times

```
def thrice_(func):
    def wrapper(*args, **kwargs):
        for i in range(3):
            func(*args, **kwargs)
    return wrapper.
@thrice_      # spam = ~~repeat~~ thrice_(spam)
def spam():
    print("in spam")
spam()
```

4) write a decorator function which calculates the execution time of any function.

```
import time.
def execution_time(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        func = (*args, **kwargs)
        end = time.time()
        return f'the function {func.__name__}
            is executed in {end-start} seconds'
    return wrapper.
@execution_time
def display():
    time.sleep(3)
    print('in display')
print(display())
```

3) write a decorator function which counts the number of arguments passed to a function.

```
def counts_(func):
    def wrapper(*args, **kwargs):
        print(f"number of args given to {func.__name__} is are:", len(args)+len(kwargs))
        func(*args, **kwargs)

    return wrapper.

@counts_
def add(a,b):
    print(a+b)

add(1, 2)
```

4) write a decorator that returns only the positive value on performing substrations.

```
def positive_(func):
    def wrapper(*args, **kwargs):
        return abs(func(*args, **kwargs))

    return wrapper.

@positive_
def sub(a,b):
    print(a-b)

print(sub(1,2))
```

7) write a decorator function to count the number of function call of the main function.

```
count=0
def no-of-function-calls(func):
    def wrapper(*args, **kwargs):
        global count
        count+=1
        func(*args, **kwargs)
    return wrapper.

@no_of_function_calls.
def sub(a,b):
    print(a-b)

Sub(1,2)
Sub(3,9)
Sub(7,2)
Print(Count)
```

Syntax

```
*** delay for n seconds ***
def outer(parameters of decorator):
    def decorator(main_func):
        def wrapper(parameters_of_mainfunc):
            ** additional functionality **
            mainfunc(parameters)
        return wrapper
    return decorator
```

```
@no_dec_fun
def push(element):
    l=[]
    l.append(element)
    print(l)
Push("hello")
Print(f"{c} functions decorated")
```

11. WADF to create a dictionary of function name and number of function call pair

```
d={}
def no_dec_fun(func):
    def wrapper(*args, **kwargs):
        if func.__name__ not in d:
            d[func.__name__]=1
        else:
            d[func.__name__]+=1
        func(*args, **kwargs)
    return wrapper.
@no_def_fun
def sub(a,b):
    Print(a-b)
Sub(5,7)
Sub(6,4)
Sub(8,3)
```

```
@ no_dec_fun
def push (element):
    l=[]
    l.append (element)
    Print (l)

Push ("hello")
Push ("world")
Print (d)
```

12. WADF to check if the number is positive, if positive perform addition else...

```
sm, ml = 0, 1
def n_positive (*arg):
    def positive (func):
        def wrapper (*args, **kwargs):
            global sm, ml
            for i in arg:
                if i > 0:
                    sm += i
                else:
                    ml *= i
            func (*args, **kwargs)
        return wrapper
    return positive.

@n_positive (1,4,6,-3,5,-3)
def operation ():
    print (sm, ml)

operation ()
```

(or)

```
sm, ml = 0, 1
def positive(func):
    def wrapper(*args, **kwargs):
        global sm, ml
        for i in args:
            if i > 0:
                sm += i
            else:
                ml *= i
        func(*args, **kwargs)
    return wrapper

@positive
def operation(a, b, c, d):
    print(sm, ml)

operation(4, 6, 9, -4)
```

(or)

```
def positive(func):
    def wrapper(*args, **kwargs):
        c = 0
        for i in args:
            if i > 0:
                c += 1
            else:
                return "Not positive"
        if c > 1:
            return func(*args, **kwargs)
    return wrapper
```

```
@positive
def add(a,b):
    return a+b
Print(add(1,2))
Print(add(-4,5))
Print(add(9,3))
```

13) WADF to check if the args are iterables or not, if it is iterable return length else 'not iterable'.

```
def iterable_(func):
    def wrapper(arg):
        if isinstance(arg, (str, list, tuple,
                            set, dict)):
            Print(len(arg))
        else:
            Print("not iterable")
        func(arg)
    return wrapper.

@iterable
def args(a):
    Print(a)
args("happy")
```

14) WADF to reverse a string using delay function.

```
import time

def delay_func():

    def wrapper(*args, **kwargs):
        time.sleep(3)
        func(*args, **kwargs)

    return wrapper
```

@ delay_
```
def reverse(string):
print(string[::-1])

reverse_("bhavvvyyya")
```

if i want to pass the parameter in decorator function then

```
def n_delay(n):
        sleep(n)

@n_delay(3)
```

} extrathing to add to a normal function.

1> GENERATOR

```
def func (a, b):
    yield a+b
    yield a*b
    yield a-b

res = func (1, 2)
Print (next (res))
Print (next (res))
Print (next (res))

res = func (2, 2)
Print(list(res))
```

2> Generate ~~list of~~ Square number of a given list using generator function.

```
l =[1, 2, 3, 4, 5]
def Square_(l1):
    for i in l1:
        yield i**2
res = Square_(l1)  } # print(list(Square_(l))
Print (list (res))

# l = (i**2 for i in l1)
Print (list (l1))
```

GENERATORS OVERVIEW

> Python generator is used to create python iterators
> This is done by defining a function but instead of the return statement returning from the function, use the "yield" keyword.
> A Generator is similar to a function returning an array.

Features of Generators

* A Generator is a function that returns an iterator it generates values using the 'yield' keyword.
2) They don't take memory of a list. They are LAZY iterables. Generators are used for saving memory.
3) When called on next() function, it raises StopIteration exception when there are no more values to generate.
4) 'yield' keyword suspends or pauses the execution of the function. But 'return' statement ends the function.

Yield v/s return.

Return	Yield
1. Stops the execution	1. Pauses the execution
2. cannot return to the function block	2. can return to the function block
3. Multiple return statements cannot be written in a function.	3. Multiple yield keyword can be written in a function.
4. can return multiple elements	4. can return multiple elements

1. write a generator function generate only the strings with odd length in the given list

```
names = ["bob", "steve", 'alex', "maya", 'john']
def odd_length(string):
    for i in string:
        if len(i) % 2 == 1:
            yield i
print(list(odd_len(names)))

# using generator expression.
l = (i for i in names if len(i) % 2 == 1)
print(l)
print(list(l))
```

2. Generate a tuple of only numeric values in the given list.

```
items = ["flipkart", 2021, "gmail", 1.2, [1,2,3],
         2+3j, True]
def num_value(values):
    for i in values:
        if isinstance(i, (int, float, complex))
            and not isinstance(i, bool):
            yield i
Print(tuple(num_value(items)))

#
g = (i for i in items if isinstance(i, (int, float,
                                        complex)))
print(tuple(g))
```

3) Generate 10 fibonacci numbers.

```
def fibo(n):
    a, b = 0, 1
    for _ in range(n):
        yield a
        c = a+b
        a, b = b, c
        yield f"{n}th fibo number is {a}"

Print(list(fibo(10)))
```

for nth fibonacci number.

```
def fibo(n):
    a, b = 0, 1
    for _ in range(n-1):
        c = a+b
        a, b = b, c
    yield f"{n}th fibo number is {a}"

Print(list(fibo(10)))
```

4) Generate a list → if individual datatype, reverse it else keep it as it is.

```
items = ["flipkart", 2021, "gmail", 1.2, [1,2,3]
         , 2+3j, True]
def rev_ind(lis):
    for i in lis:
        if isinstance(i, (int, float, complex, bool)):
```

```
            yield str(i)[::-1]
        else:
            yield i
print(list(rev_ind(items)))
```

#

```
l1 = (str(i)[::-1] if isinstance(i,(int,float,complex)) else i for i in item)
```

5) Generating list of PI values with increasing decimal point number.

```
From math import pi
def pi_(n):                    → (1, n, -1) → reversing.
    for i in range(n):
        yield round(pi,i)
print(list(pi_(4)))

a=(round(pi,i) for i in range(n))
print(a)
```

CSV FILEHANDLING PROGRAMS

1. write a program to read all the names of the employees in employee.csv file.

```
import os
import csv
path = r"abcxyz"
with open(path) as file:
    rows = csv.reader(file)
    header = next(rows)
    for data in rows:
        print(data[0], end = " ")
```

```
with open(p) as file:
    rows = csv.DictReader(file)
for data in rows:
    print(data["name"])
```

2) write a program to print only the salaries that are >70000

```
with open(path) as file:
    rows = csv.reader(file)
    header = next(rows)
    for i in rows:
        if int(i[3]) > 70000:
            print(i[3])
```

using dictreader.

```
with open(p) as file:
    r = csv.DictReader(f)
    for data in r:
        if data["Pay"] > str(70000):
            print(data["Pay"], end=" ")
```

WAP to group male and female employees in the employees file :

```
male = []
female = []
with open (path) as file :
        rows = csv.reader (file)
        header = next (rows)
        for i in rows:
                if i[1] == "male":
                        male . append (i[0])
                else :
                        female :append (i[0])
        Print (f"male - employees : (male), \n\n
                female employees : {female}")
Print()
```

4) WAP to group employees based on their team

```
d = {}
with open (path) as file :
        rows = csv.reader (file)
        header = next (rows)
        for i in rows:
                if i[2] not in d:
                        d[i[2]] = [i[0]]
                else:
                        d[i[2]] . append (i[0])
        Print (d)
```

5) WAP to Sort the Shares in test.csv file based on the Share prices:

```
Path = r"C: "
with open (path) as file:
    l=[]
    rows = csv.reader(file)
    header = next (rows)
    for i in rows:
        if i[2].strip():
            l.append (float (i[2]))
Print (f'shares: {sorted(l)}')
```

6) WAP to add the all the shares in the test.csv file.

```
with open (path) as file:
    l = 0
    rows = csv.reader(file)
    header = next (rows)
    for i in rows:
        if i[1].strip():
            l+= int(i[1])
Print (f' total number of shares are{l}')
```

7) Analysing Vaccination details.

A. total Vaccination of all countries:

```
Path = r"C: "
with open (path) as file:
        tc = 0
        rows = csv.reader (file)
```

```
        header = next(rows)
        for i in rows:
            if i[5].strip():
                tot = int(i[5])
Print(f'total vaccinations of all countries:{tot}')
```

B. Total Vaccination by countries.

```
Path = r"C:
with open(path) as file:
    rows = csv.reader(file)
    header = next(rows)
    for i in rows:
        Print(f'{i[0]}:{i[5]}')
```

C. names of countries and WHO regions.

```
Path = r"C
with open(path) as file:
    rows = csv.reader(file)
    header = next(rows)
    for i in rows:
        Print(f'{i[0]}:{i[2]}')
```

D. country and person vaccinated and get top 3 countries with most vaccinated people

```
Path = r"C:
with open(path) as file:
    d = {}
    rows = csv.reader(file)
    header = next(rows)
    for i in rows:
        if i[0].strip() and i[6].strip():
```

```
    d[i[0]] = int(i[1])
res = sorted(d.items(), key = lambda  i: i[-1])
print()
Print()
Print(res[-3::])
```

5) countries with less than 10K Vaccinated people:

```
path =
with open(path) as file:
    rows = csv.reader(file)
    header = next(rows)
    for i in rows:
        if i[0].strip() and i[1].strip() and
            int(i[1]) < 10000:
                print(i[0], ":", i[1])
```

6) Get the latest updated Country with its total vaccinations and number of people Vaccinated.

```
from datetime import datetime.
Path = r"C[path",
with open(path) as file:
    dates = []
    rows = csv.reader(file)
    header = next(rows)
    for i in rows:
        if i[10].strip():
            dates.append(i[10])
    dates.sort(key = lambda date: datetime.strp
                time(date, "%y-%b-%d"))
Print(dates)
```

Decorator

WADF to check if the args are iterables or no, if it is iterable return length

```
def iterable_(func):
    def wrapper(arg):
        if isinstance(arg, (str, list, tuple, get, dict)):
            print(len(arg))
        else:
            print("not iterable")
        func(arg)
    return wrapper

def iter(func):
    def wrappen(arg):
        if isinstance(arg, str):
            print(arg[::-1])
        else:
            print("not a string")
        func(arg)
    return wrapper.

@iterable_
@filter
def args(a):
    print(a)
args("happy")
```

SERIALIZATION IN PYTHON

* The Serialization process is a way to convert a data structure into a linear form that can be stored or transmitted over a network.
* This process is also referred to as marshalling
* The reverse process, which takes a stream of bytes and converts it back into a data structure is called deserialization or unmarshalling.

(whatever the data stored will not be modified.)

JSON (Java Script Object Notation)

>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.

>JSON is most commonly used for client-server communication because;

→ It is human readable.

→ It stores data in Key-Value pair.

→ JSON is language-independent.

→ It transfers the data as it is over the network

Serializing the data

The json module provides the following two methods to serialize python objects into JSON format.

1. JSON.dump(object, file): used to write Python object as JSON formatted data into a file.
   - the file type can be anything including text, JSON or even binary file.

2. JSON.dumps(object): Serializes any python object into JSON formatted String.

De-Serializing the Data

The Json module provides the following two methods to de serialize JSON data into python objects.

1. json.load(file): used to read JSON object as python objects from a file.
   * The file type can be anything including text, JSON or even binary file.

2. json.loads(object) de serializes a JSON formatted String into python objects.

Example

```
import json.

# Serialization - dump, dumps
data = {"username": "user1", "Password": "Pwd"}
Json_obj = json.dumps(data)
Print(json_obj)
Print(type(json_obj))
```

```
data = None.
json_obj = json.dumps(data)
Print (json_obj)
Print (type (json_obj))

data = {"username" : "user1", "Password" : "Pwd"}
data 2 = None
with open ("Sample.json", "w") as file :
    json.dump (data, file)
    json.dump (data 2, file)
    Print ("data dumped Successfully)

# de-serialization - load , loads

data = {"username" : "user 1", "Password" : "Pwd"}
json_obj = json.dumps (data)
Print (json_obj)
Print (type (json_obj))

Python_obj = json.loads (json_obj)
Print (python_obj)
Print (type (python_obj))

data = {"username": "User1", "Password": "Pwd"}
with open ("sample.json") as file:
    data = json.load (file)
    Print (data)
    Print (type (data)
```

PICKLE

> "Pickling" is the process whereby a python Object hierarchy is converted into a byte Stream, and "unpickling" is the inverse operation, whereby a byte stream (from a binary file or bytes-like object) is converted back into python object.

METHODS

1. dump () : pickle.dump(data, fp)
2. dumps() : pickle.dumps(data)
3. load () : Pickle.load (fp)
4. loads () : Pickle.loads (data)

Note :- The file should be opened in "binary" mode. (wb, rb, ab) and the file extension will be .pkl

Example

```
import Pickle.

# Serialize - dump, dumps
data = "hello"
Pck_obj = pickle.dumps(data)
Print (pck_obj)
Print ( type (pck_obj))
```

```
data = "hello"
with open("demo.pkl", "wb") as file:
    pickle.dump(data, file)

# deserialize - load, loads.
Python_obj = pickle.loads(pck_obj)
Print(python_obj)
Print(type(python_obj))

with open("demo.pkl", "rb") as file:
    data = pickle.load(file)
    Print(data).
```

OVERVIEW

> An event which causes the termination of the program.

> To handle unexpected termination of the program exception handling is done.

> To handle the exception, try and except block is used.

Types

1. Default except block
2. Specific except block.
3. Generic except block
4. Multiple except block

1) Default except block

Syntax

```
try:
    statements

except:
    statements
```

2) Specific except block

Handles specific exceptions only.

Syntax:

```
try:
    statements
except <exception_name>:
    statements
```

3) Generic except block

>Handles all types of exceptions in a single except block.

Syntax:

```
try:
    statements
except Exception/Base Exception:
    statements.
```

) Multiple except block

-A single try block can have multiple except blocks.

Syntax:

```
try:
    statements
except exception1:
    statements.
except exception2:
    statements
```

Nested try and except block

Syntax:

```
try :
    statements
        try:
            statements

        except :
            Statements.

except:
    nested try - except block
```

"as" Keyword

> Used to give alias name for the exception names written in the except block.

> Syntax : except <exception name> as alias_name:

Raise Keyword

> used to raise a specific exception whenever the condition is matched.

> Once an exception is raised, it searches for the specific except block and handles the exception

Syntax : raise error_name("message")

Finally block

> It is a block which will get executed even when the exception is raised or not.

> we can add try and except block inside finally

Syntax
```
try:
    statements
except:
    statements
finally:
    statements
```

Else block

It is a block which will get executed even when the exception is not raised.

Syntax:
```
try:
    statements.
except:
    statements.
else:
    statements.
```

User defined exceptions or custom exception

* custom exceptions can be created by inheriting exception class.

* Syntax:
```
class user-exception_name (Exception):
    pass
```

try - except, else, finally, raise.

1. Default except block

```
a = 1
b = 0
try:
    print (a/b)
    l.append (10)
except:
    print ("error handled")
```

2. Specific except block

```
except ZeroDivisionError as msg:
        print (msg)

except NameError:
    Print ("name error handled")
```

3) Generic except block.

```
except (ZeroDivisionError, Nameerror):
        print ("error handled")
```

```
l = [1, 2, 3]
try:
    print (l[3])
    l.remove (4)
```

4) Multiple except block

```
except IndexError as msg:
    Print (msg)
```

```
except ValueError as msg:
        Print (msg)
```

```
l=[1,2,3]
try:
   print(l[2])
    l.remove(1)
except (IndexError, ValueError) as Msg:
        Print (msg)
else:
   Print('no errors')
finally:
    Print ("executed")
```

```
a=1
b=0
```

```
try:
    if b>0:
      pass
    else:
       Print (a/b)
except ZeroDivisionError:
     raise zerodivision Error('error occured')
```

```
try:
  e.
```

custom exception or user defined exception

```
class UserNotPresent(Exception):
    pass
userinput = input("enter the name")
if userinput == "shashank":
    print(userinput)
else:
    raise Exception("user is invalid")
```

OOPS

1. A class is collection/Set of functions that carry out various operations of "Instances"

2. Instances are the actual objects./ data that your function manipulate on.

3) Different ways of storing data using built-in data structures.

```
a = [1, 2]
t = (1, 2)
d = {'a': 1, 'b': 2} #advantage of storing
```

data in dict is that you can

#performing different operations on the data stored in the list.

1. sort
2. reverse
3. __len__()
4. __getitem__()
5. __contains__()

want to perform other operations apart from built-in.

1. Adding a[0] = a[0] + 0.5, a[1] = a[1] + 0.5
2. Get the total of two co-ordinates total = a[0] + a[1]
3. Swap two co-ordinates temp = a[0], a[0] = a[1], a[1] = temp
4. Sorting two co-ordinates a.sort()
5. Resetting the co-ordinates a[0] = 0, a[1] = 0

User defined class or datatype.

```
class point:
    # data is being saved inside a dictionary.
    def __init__(self, a, b):
        self.a = a
        self.b = b

P1 = Point(1, 2)
P2 = Point(10, 20)
```

The values are internally stored in a dictionary
It is also called instance.

```
print(p1.__dict__)  # {"a":1, "b":2}
Print(p2.__dict__)  # {"a":10, "b":20}
```

"Point" class with a some methods.

```
class point:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    # takes data from instance dictionary
    def move(self, dx, dy):
        self.at = dx
        self.bt = dy
```

```
# resets the value of self.a and self.b to Zero.
    def reset(self):
        self.a = 0
        self.b = 0

# Method that sorts points.
    def sort(self):
        if self.a < self.b:
            return (self.a, self.b)
        return (self.b, self.a)

# Method that swaps values of 'a' and 'b'
    def swap_points(self):
        temp = self.a
        self.a = self.b
        self.b = temp
        return (self.a, self.b)

    def total(self):
        return self.a + self.b

    P1 = Point(1, 2)
    P2 = Point(1.4, 1.2)

# The information about data is present in
  instance dictionary
# The information about methods is present in
      class dictionary.
```

```
class Calculator:
    def __init__(self, x, y):
        self.a = x
        self.b = y

    def add(self):
        return self.a + self.b

    def mul(self):
        return self.a * self.b

C1 = Calculator(1,2)
C2 = Calculator(4,5)
C3 = Calculator(10,20)

# Employee class.
Class Employee:
    def __init__(self, fname, lname, pay):
        self.fname = fname
        self.lname = lname
        self.pay = pay

    def email(self):
        return f'{self.fname}.{self.lname}@Company.com'

C1 = Employee("Steve", "Jobs", 1000)
C2 = Employee("Bill", "Gates", 2000)
```

```
class player:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.health = 100
    def move(self, dx, dy):
        self.x += dx
        self.y += dy.
    def attack(self, pts):
        self.health -= pts.

P1 = player(1, 2)
P2 = player(3, 4)
P3 = player(5, 6)

print(P1.__dict__)
Print(P1.__class__.__dict__)
Print(player.__dict__)

# please note that __dict__ attribute is available
    only for custom classes.

class point:
    # a and b with default values.
    def __init__(self, a=0, b=0):
        self.a = a
        self.b = b
P1 = Point()
P2 = Point()
```

```
class Employee:
    def __init__(self, fname, lname, pay, *args):
        self.fname = fname
        self.lname = lname
        self.pay = pay
        self.args = args
e1 = Employee('steve', 'jobs', 1000, 'Python', 96,
              '2200 valley view lane')

# Overloading Constructor using optional arguments
class point:
    def __init__(self, a=0, b=0, c=0):
        self.a = a
        self.b = b
        self.c = c
P1 = Point()
P2 = Point(1)
P3 = Point(1,2)
P4 = Point(1, 2, 3)

# We can have multiple __init__ method's
  but the latest implementation of

class point:
    def __init__(self, a, b):
        self.a = a
        self.b = b.
```

Re-defining __init__ method (new implementation)

```
def __init__(self, a, b, c):
    self.a = a
    self.b = b
    self.c = c
```

* Python maintains the information about methods in class dictionary.
* we can access the dictionary using point.__dict__
* Method name will be the key of the dictionary and the reference of the method.

e.g.

```
>>> point.__dict__
>>> mappingproxy({'__module__': '__main__', '__init__'
                  : <function point.__init__
# '__weakref__' : <attribute '__weakref__' of
    'point' objects>, '__doc__' : None.
```

Practise in laptop.

```
t = (1, 2)
d1 = {"a" : 1 , "b" : 2}
d2 = {"x" : 4 , "y" : 5}

a = [1, 2]
b = [3,4]
```

low level operations (indexing)

```
total = a[0] + a[1]
a[0] = a[0] + 0.5
a[1] = a[1] + 0.5
```

want to swap the co-ordinates.

```
x = a[0]
y = a[1]
a[0] = y
a[1] = x
```

rest the co-ordinates to [0, 0]

```
a[0] = 0
a[1] = 0
```

```
class point:
    def __init__(self, a, b):
        self.a = a
        self.b = b
    def move(self, dx, dy):
        self.a = self.a + dx
        self.b = self.b + dy
    def reset(self):
        self.a = 0
        self.b = 0
    def sort(self):
        if self.a > self.b:
            temp = self.a
            self.a = self.b
            self.b = temp
```

```
def total (self):
    return self.a + self.b
```

__init__ method is helping us to do that
internally the data is stored inside the dict.
which will be created once the __init__ method is called
__init__ is also called as constructor. It will be automatically called.
when you call the class to save the data or when you create an instance of the class.

Saving the data inside the class.

```
P1 = Point (1, 2)
P2 = Point (3, 4)
P3 = Point (5, 6)
```

manupulating the data contained inside a dictionary.

```
P1.move(0.1, 0.1)    # Point.move(P1, 0.1, 0.1)
P2.move(0.5, 0.5)    # Point.move(P2, 0.5, 0.5)
P3.move(1, 2)        # Point.move(P3, 1, 1)
```

See the dictionary where the data is stored.

instance dictionaries.

```
Print(P1.__dict__)
Print(P2.__dict__)
Print(P3.__dict__)
```

```
class Calculator:
    def __init__(self, a, b):
        Print(f"Calling __init__ method and
Saving {a} and {b}")
        Self.a = a
        Self.b = b

    def add(Self):
        return Self.a + Self.b

    def Sub(Self):
        return Self.a - Self.b

    def mul(Self):
        return Self.a * Self.b

    def div(Self):
        return Self.a / Self.b

# Saving the date inside calculator object
  (inside dict)

C1 = Calculator(10, 20)
C2 = Calculator(1, 2)
C3 = Calculator(15, 25)
```

```
c1 = {"fname": "steve", "lname": "jobs", "pay":1000}
c2 = {"fname": "bill": "lname": "gates", "pay": 2000}

class Employee:
    def __init__(self, fname, lname, pay):
        self.fname = fname
        self.lname = lname
        self.pay = pay

    def email(self):
        return f"{self.fname}.
{self.lname}@Company.com"

    def pay_hike(self, percentage_hike):
        hike_amount = self.pay * percentage_
                                    hike
        self.pay = self.pay + hike_amount.


e1 = Employee("steve", "jobs", 1000)
e2 = Employee('bill', 'gates', 2000)

e1.pay_hike(0.1)      # Employee.pay_hike(e1)
e2.pay_hike(0.05)     # Employee.pay_hike(e2)

class point:
    def __init__(self, a=0, b=0, c=0):
        self.a = a
        self.b = b
        self.c = c
```

P1 = Point()

P2 = Point(1)

P3 = Point(1, 2)

P4 = Point(1, 2, 3)

P5 = Point(a=10, b=20, c=30)

P6 = Point(10, 20, c=30)

```
class point {
    Public void Point() {
        This.Point(0, 0, 0)
    }
    Public void Point(a) {
        this.point(a, 0, 0)
    }
    Public Void Point(a, b) {
        this.point(a, b, 0)
    }
    Public Void Point(a, b, c) {
        This.a = a
        this.b = b
        this.c = c
    }
}
```

```
class point:
    # overloaded Constructor can be achieved by.
# having default values to the arguments.
    def __init__(self, a=0, b=0, c=0):
        self.a = a
        self.b = b
        self.c = c

    def __init__ (self, a, b):
        self.a = a
        self.b = b

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

def add (a, b, c):
    return a + b+c

# Point P1 = new point()
# Point P2 = new point(1)
# Point P3 = new point (1, 2)
# Point P4 = new Point (1, 2, 4)
```

2nd day

```
class BankAccount:
    # shared by all the customers on instances
of BankAccount class
    interest_rate = 0.05   # class variable
    def __init__(self, name, balance=0):
        self.name = name
        self.balance = balance
        self.transactions = []
        self.transactions.append(f"****
inital deposit : {balance} *****")

    def deposit(self, amount):
        self.balance = self.balance + amount
        self.transactions.append(f"{amount}
deposited : {amount}")

    def withdraw(self, amount):
        if amount > self.balance:
            raise Exception("Insufficient fundi")

        self.balance = self.balance - amount
        self.transactions.append(f"{amount}
withdrawn : {amount}")
```

```
    def statement(self):
        for item in self.transactions:
            print(item)

        print(f"**** total balance {self.balance} ****")
```

```
    def transfer_funds(self, to_account, amount):
        if amount > self.balance:
            raise Exception("Insufficient funds !!!")

        to_account.deposit(amount)
        self.balance -= amount
```

```
    def roi(self):
        interest_amount = self.balance * BankAccount.interest_rate

        self.balance = self.balance + interest_amount

c1 = BankAccount("steve", 1000)
c2 = BankAccount("bill", 2000)
c3 = BankAccount("Amy", 3000)
c4 = BankAccount("Alex")
```

afternoon

```
class ShoppingCart:
    # class Variable.
    Prices = {"iPhone": 800, "iMac": 2500, "Iwatch":
      3000, "iPad": 3500}

    Products = {"iPhone": 5, "iMac": 3, "Iwatch": 2,
      "iPad" 4}

    def __init__(self):
        self.cart = []

    def add_item(self, name, quantity):
        if name not in ShoppingCart.products:
            raise Exception("Product not
                available")

        elif quantity > ShoppingCart.products
                [name]:
            raise Exception("Product out
        of stock")

        else:
            self.cart.append({"name": name,
"quantity": quantity, "Price": shoppingcart.
    prices[name]})

            Shoppingcart.products[name] -=
quantity
```

```
def remove_item (self, name):
    for item in self.cart:
        If name == item ['name']:
            if item['quantity']> 1:
                item ['quantity']=
item['quantity'] - 1
            else:
                self.cart .remove (item)


Def total_cost(self ):
    total =0.00
    for item in self.cart:
        total += item ['quantity'] *
item['price']
    return total

c1 = ShoppingCart()
c2 = ShoppingCart()
```

day-3

INHERITENCE

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

    def apple(self):
        Print(f"Executing Parent Apple
{self.value}")

    def google(self):
        Print("Executing Parent Google")
        self.apple()
```

Case1: child class having Independent Method.

```
class child1(Parent):
    def __init__(self, value):
        # calling parent class Constructor
        super().__init__(value)

    def yahoo(self):
        Print("Child 1 Yahoo")
```

Case2: child class overriding parent class Method

```
class child 2 (Parent):
    def __init__ (self, value):
        super().__init__(value)

# Method Overriding
    def apple (self):
        Print (f" Executing child 2 Apple
{self.value} ")
```

Case 3: child class overriding parent, but re-using the parent class functionality

```
child child 3 (Parent):
    def __init__ (self, value):
        super().__init__(value)

    def apple (self):
        # this is the extra functionality
        Print ("Executing child 3 Apple")
        # Re-using the Parent class.
functionality.
        super().apple()
```

Case 4: Child class having a separate attribute in __init__ method.

```
class Child4(Parent):
    def __init__(self, value, extra_value):
    # super().__init__(value)
    Parent.__init__(self, value)
    self.extra_value = extra_value
```

Case 5: child inheriting from Multiple Parents.

```
Class Parent2:
    def __init__(self, a):
        self.a = a

    def facebook(self):
        Print("Executing Parent2 facebook")


class child5(Parent, Parent2):
    def __init__(self, x, y):
        Parent.__init__(self, x)
        Parent2.__init__(self, y)
```

```
    def watsappC(self):
        Print("Executing child5 watsapp")

# Multi-level Inheritance
class A:
    def demo(self):
        Print("A")

class B(A):
    def demo(self):
        Print("B")
        # Super().demo()
        A.demo(self)

class C(B):
    def demo(self):
        Print("C")
        # Super().demo()
        B.demo(self)

class Parent:
    def spam(self):
        Print("Parent spam")

class child1(Parent):
    def spam(self):
        Print("child1 spam")
        Super().spam()
```

```
class child 2 (parent):
    def spam (self)
        Print ("child 2 spam")
        super().spam ()

class child 3 (child 1, child 2):
    pass
```

__mro__ (Method Resolution Order) is the order in which python looks up for an attribute in inheritence hierarchy.

```
# child 3.__mro__

class Spam:
    a=10 # class Variable
    def apple (self):
        Print (f"Spam Apple
{self.__class__.a}")

class Demo (Spam):
    # overriding class Variable
    a=20 # class Variable
    def google (self):
        Print ("Google")
```