Date Page

## Iterators

* It is a programm that enables you to access each item in the collection, one at a time, without needing to know internal structure of collection.
* Iterable are those collections that can be iterated.
eg. list, tuples, strings etc..

* ~~In~~ for loop is a iterator which gives all the items of collection in one go but when we iter a collection, we get all the items one by one.

```
* mylist = [1,2,3]
  n = iter(mylist)
  next(n)
    1
  next(n)
    2
  next(n)
    3
  next(n)
  stop iteration
```

Date Page

## Generators

* It is a special type of function that allows you to create an iterator in python.
* Unlike other def() functions it gives one value at a time.
* It will require ~~gen~~ yield.
* def my_generator():
    yield 1
    yield 2
    yield 3
  gen = my_generator()
  next(gen)
    1
  next(gen)
    2
  next(gen)
    3
  next(gen)
    Stop iteration

Fibonacci series:
* It is sum of last two numbers (0, 1, 1, 2, 3...)

* def fib(n):
    a=0
    b=1
    for i in range(n):
    a,b=b,a+b

def fib(1000)=f
next(f)
0

Page

Date

## Lambda Functions

* It is Used to Create functions, known As lambda Expressions.
* Unlike def functions, lambda functions are used to create short functions.
* Syntax: lambda Argument : expression
* Eg: is_even = lambda x: x%2==0
  is_even(4)
  True

Key: Lambda Uses Key function to provide a simple and inline way to define behaviour.
* Syntax: Key=lambda x: expression
* Eg: a=["den","fibonacci","xi","add","door"]
  d=sorted(a, key=lambda a: len(a))
  d
  ['xi', 'den', 'add', 'door', 'fibonacci']

* Eg: Fibonacci Sequence
  fib=lambda n: n if n<=1 else fib(n-1)+fib(n-2)
  [fib(i) for i in range(10)]
  [0,1,1 . . . 34]

* Eg: Factorial of Number
  fact=lambda a: 1 if a<=0 else fact(a-1)* a
  fact(5)
  120

Data Page

# Map

* Map function applies a function to each of given iterables (like list, tuples or strings).
* Syntax:

```
map(function, iterables)
```

* Eg:

```
n = [1,2,3,4]
def sq(x):
    return x**2
list(map(sq,n))
[1,4,9,16]
```

* Using lambda:

```
n = [1,2,3,4]
list(map(lambda x: x**2, n))
[1,4,9,16]
```

* Eg:

```
n = "pwskills"
list(map(lambda x: x.upper(), n))
['P', 'W', ... 'S']
```

* Eg:

```
l1 = [100,200,300]
l2 = [1,2,3]
list(map(lambda(x,y: x+y, l1, l2)
[101,202,303]
```

map(function, iterables)

*Eg: n=[1,2,3,4]
def sq(x):
    return x**2
list(map(sq,n))
[1,4,9,16]

## Reduce

*Reduce function from functools module, Used to perform a cumulative operation on a list or sequence, which reduce it to single result.

*Syntax:
from functools import reduce
result = reduce(function, iterable, initializer)

*Eg: l=[2,1,3,4,4,5,6]
reduce(lambda x,y: x+y,l)
25

*Eg: words=["Data","Science","Courses"]
reduce(lambda x,y: x+" "+y, words)
'Data Science Courses'

*Eg: numbers=[1,2,3,100,1000,10000]
reduce(lambda x,y: x if x>y else y, numbers)
10000

Page

Date

Eg: def factorial(n):
    return reduce(lambda x,y: x*y, range(1,n+1))
factorial(5)
120

## Filter

*Filter function is used to create an iterator from iterable (like a list).
*Syntax: filter(function, iterable)
* Eg: list(filter(lambda x: x%2==0, l))
l=[2,1,3,4,4,5,6]
[2,4,4,6]

*Eg: l1=[-1,-2,3,4,5]
list(filter(lambda x: x>0, l1))
[3,4,5]

*Eg: s=["Ajay","pwskills"]
list(filter(lambda x: len(x)>4, s))
['pwskills']

ge(1,n+1)

Date Page

# OOPs

* It is a way of writing code that models real world things as "objects".
* Class Car:
    pass
c1 = Car()
type(c1)
<class '__main__.Car'>

* Eg: Class Car():
    def acc(self):
        print("car is accelarating")
c1 = Car()
c1.acce()
car is accelarating

* Eg: Class Bank:
    def deposit(self, amount):
        print("I am depositing money")
    def withdraw(self, withdraw):
        print("I am withdrawing money")
ajay = Bank()
ajay.deposit(₹100)
I am depositing money

Types of OOPs:
(i) Inheritance.
(ii) Abstraction
(iii) Polymorphism
(iv) Encapsulation

Page

Date

## __init__

* This method is used to setup function for class.
* When we create a class, __init__ runs automatically to setup the first values.
* Eg: Class ListOps: __init__

```
    def extract_even(self,l):
        self.l=l

    def extract_even(self,l):
        l1=[]
        for i in l:
            if i%2==0:
                l1.append(i)
        return l1

    def extract_odd(self,l):
        l1=[]
        for i in l:
            if i%2!=0:
                l1.append(i)
        return l1
ops1=ListOps()[1,2,3]
ops1.l
 [1,2,3]
ops1.extract_even
 [2]
```

Page

Date

## ① Inheritance

* It is when one class takes properties and behaviours of another class.
* this help reduce repeated codes.
* With this we get a parent class and a child class.
* Syntax:
```
class ParentClass:
    pass
class ChildClass:
    pass
```

## → Types of inheritance:

### ① Single Inheritance:

* It is when one child class gets its features and functions from just one parent class
* Syntax:
```
class Parent:
    def Parent_met(self):
        print("Parent")
class Child(Parent):
    def Child_met(self):
        print("Child")
Child=Child()
Child.parent_met()
Parent
```

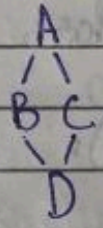
(ii) Multiple Inheritance:

* It means class can have more than one parent class.
* This allows child class to use features of both parent classes.
* Syntax:

```
Class Parent 1:
    pass
Class Parent 2:
    pass
Class Child(Parent 1, Parent 2):
    pass
```

→ Diamond Problem:

* It happens in multiple inheritance, when a class inherits from two classes that both inherit from same base class.
* Syntax:




* Eg:

```
Class A:
    def method(self):
        print("A")
Class B:
    def method(self):
        print("B")
Class C:
    def method(self):
        print("C")
Class D:
    def method(self):
        print("D")
```

```
d = D()
d.method()
B
```

Page

Date

→ Solution:
* Python uses MRO to set the order.
* So we call mro to get the ~~name of~~ order of inheritance.
* Eg: Print(D.~~mro~~ __mro__)
(<class '__main__.D'>, <class '__main__.B'>, ...,
<class 'object'>)

(iii) Multilevel Inheritance:
* It is when a class inherits from another class and that class inherits from another class, forming chain.
* In this Child Class 1 inherits from parent class and child Class 2 inherits from Child Class 1.
* Syntax: Class Grandparent:
pass

Class Parent(Grandparent):
pass

~~Class~~
Class Child(Parent):
pass

* Child Class Can access both parent and grandparent Class.

(iv) Hierarchical Inheritance:
* It is when multiple child classes inherit from single parent class.
* All child classes share the same properties from parent class.
* Syntax: Class Parent:
    pass

Class Child1(Parent):
    pass

Class Child2(Parent):
    pass

(v) Hybrid Inheritance:
* It is when a class uses more than one type of inheritance, such as multiple and multilevel at same time..
* Syntax: Class A:
    def met_A(self):
        print("A")
Class B(A):
    def met_B(self):
        print("B")
Class C(A):
    def met_C(self):
        print("C")
Class D(B,C):
    def met_D(self):
        print("D")
* This combination of inheritance is called hybrid inheritance.

Date Page

## (ii) ABSTRACTION:

* It means showing only the important parts and hiding the complex details.
* It helps you use only what you needs and hides unnecessary details.
* Syntax: from abc import ABC, abstractmethod

```
Class AbstractClassName(ABC):
    @abstractmethod
    def abstract_method(self):
        pass

Class Concrete.Class(AbstractClassName):
    def abstract_method(self):
        print("Abstract")
obj = Concrete.Class()
obj.abstractmethod()
```

*Eg:

```
Class shape:
    @abc.abstractmethod
    def calculate_area(self):
        pass
Class Rec(Shape):
    def calculate_area(self):
        return("len*bre")
Class Cir(Shape):
    def calculate_area(self):
        return("pi*r**2")
rect = Rec()
rect.calculate_area()
'len*bre'
```

→ # Class Method:
* It is a type of method that belongs to class itself, not to any specific object created from that class.
* It is useful when you work with class level not instance level.
* Syntax: Class ClassName:
@classmethod
def met_name(cls, parameter):

* e.g: Class student:
def __init__(self, name):
self.name = name
@classmethod
def student_details(cls, name):
return cls(name)
Student.student_details("Ajay")
'Ajay'

→ Add external function:
* To add extend function by class method, we use
* Class Student:
total_students = 0
def __init__(self, name):
self.name = name
student.total_student = student.total_student+1
@class method
def get_total_stud(cls):
return cls.total_students

Page

Date

```
def course_details(cls, course_name)
    print ("Details are:", course_name)

Student.course_details = classmethod(course_details)
#Adding new function

Student.course_details("Data Science")
'Details are: Data Science.'
```

→ Delete a function

```
del Student.course_details
#deleting new function

Student.course_details("Data Science")
#error
```

→ Another way to delete

```
delattr(Student, "get_total_students")

Student.get_total_students()
#error
```

Page

Date

## → Static Method:

* It is a method that belongs to specific object created from class.
* By this you can call a static method using the class name, without creating objects.
* It is a type of method that belongs to class itself rather than any instance of class.
* Syntax: Class ClassName:
    @staticmethod
    def static_method_name(Parameter):
        pass

* Eg: Class Calculator:
    @staticmethod
    def add(x,y):
        return x+y
    @staticmethod
    def subtract(x,y):
        return x-y

Calculator.add(5,6)
11

Date Page

## (iii) Polymorphism

* It is a OOPs concept that allows objects of different types to be treated as if they are of the same type.
* It allows same function to work in different ways based on object it is used with.
* Eg:

```
class teacher_lecture:
    def lec_function(self):
        print("Teacher's prespective.")
class student_lecture:
    def lec_function(self):
        print("Student's prespective.")
obj1=teacher_lecture()
obj2=student_lecture()
class_obj=[obj1,obj2]
def parcer(class_obj):
    for i in class_obj:
        i.lec_function()
parcer(class_obj)
```

'Teacher's prespective.'
'Student's prespective.'

→ Types of Polymorphism:
(i) Method overloading
(ii) Method overriding

Date Page

## (iii) Polymorphism

* It is a OOPs concept that allows objects of different types to be treated as if they are of the same type.
* It allows same function to work in different ways based on object it is used with.
* Eg:

```
class teacher_lecture:
    def lec_function(self):
        print("Teacher's prespective.")
class student_lecture:
    def lec_function(self):
        print("Student's prespective.")
obj1=teacher_lecture()
obj2=student_lecture()
class_obj=[obj1,obj2]
def parcer(class_obj):
    for i in class_obj:
        i.lec_function()
parcer(class_obj)
```

'Teacher's prespective.'
'Student's prespective.'

→ Types of Polymorphism:
(i) Method overloading
(ii) Method overriding

Page

Date

## ① Method overloading:

* It allows the class to have multiple methods with same name but different parameters.
* It is when class have more than one method with same name but different input parameters.
* It allows method to perform different tasks based on arguments it recieves.
* Syntax: ① Using default Arguments:

```
class Example:
    def display(self, a, b=None, c=None):
        if b is not None and c is not
                                None:
            print(a,b,c)
        elif b is not None:
            print(a,b)
        else:
            print(a)
obj.~~display~~Example()
obj.display(1)
    '1'
obj.display(2,3)
    '2,3'
obj.display(1,2,3)
    '1,2,3'
```

Date Page

*⑥ Using *args Arguments:

```
Class Example:
    def display(self, *args):
        if len(args) == 1
            print(args[0])
        elif len(args) == 2:
            print(args[0], args[1])
        elif len(args) == 3:
            print(args[0], args[1], args[2])
        else:
            print("Unsupported numbers")

obj = Example()
obj.display(1)
    '1'
obj.display(1,2)
    '1,2'
obj.display(1,2,3)
    '1,2,3'
```

*Eg:

```
Class Student:
    def student(self):
        print("Welcome to pwskills")
    def student(self, name=""):
        print("Welcome to pwskills", name)
    def student(self, name="", course="")
        print("Welcome to pwskills", name, course)

stud = Student()
stud.student()
  'Welcome....'
stud.student("Ajay")
  'Welcome.... Ajay'
```

Date Page

## (ii) Method Overriding:

* It is used when a child class needs to change or extend the behavior of a method from its parent class.
* It is when a child class has a method with the same name as one in its parent class.
* This allows the child class to give its own version of the method.
* Syntax:
```
class ParentClass:
    def met_name(self):
        pass
class ChildrenClass(ParentClass):
    def met_name(self):
        pass
```
* Eg:
```
class Animal:
    def sound(self):
        print("Animal Sound")
class Cat(Animal):
    def sound(self):
        print("Cat Sound")
anim = Animal()
anm.sound()
'Animal Sound'
cat = Cat()
cat.sound()
'Cat Sound'
```

Date _____ Page _____

## (iv) Encapsulation:

* It involves variables and functions that operates on data, called a class.
* The main goal is to protect the internal state of an object. ~~From~~
* It helps protect the data by hiding it from direct access and only allows changes through controlled methods.
* Syntax:

```
class ClassName:
    def __init__(self):
        self.__private_variable = value
    def __private_method(self):
        pass
    def get_private_variable(self):
        return self.__private_variable
    def set_private_variable(self, value):
        self.__private_variable = value

obj = ClassName()
print(obj.get_private_variable())
obj.set_private_variable(new_value)
```

* Eg:

```
class Student:
    def __init__(self, name, degree):
        self.name = name
        self.degree = degree
stud1 = Student("Ram", "Masters")
stud1.name
'Ram'
stud1.degree
'Masters'
```

Date Page

→ Private data in context of encapsulation:
* It restricts direct access to some variables within object.
* It means certain variables in a class are hidden from outside access.
* This data can only be accessed or changed using getters and setters.
*Syntax: class ClassName:
    def __init__(self):
        self.__private_var = value
*Eg: class Student:
    def __init__(self, name, degree):
        self.name = name
        self.__degree
    def show(self):
        print("name", self.name, "degree", self.__degree)

```
aj = Student("Ajay", "Masters")
aj.name
'Ajay'
aj.show()
'name Ajay degree Masters'
aj._Student__degree
'Masters'
```

Date Page

→ Protected data from Encapsulation:

* It is useful because it lets your data be accessible to related classes, while still keeping it hidden from others.
* This means you can create a base class and let others classes build on it safely.
* It allows controlled access and encourages inheritance

*Syntax: class Base:
    def __init__(self):
        self._protected_var = "I am protected"
class Derived(Base):
    def display(self):
        print(self._protected_var)

*Eg: class College:
    def __init__(self):
        self.college_name = "PWskills"
class Student(College):
    def __init__(self, name):
        self.name = name
        college.__init__(self)
    def show(self):
        print("name", self.name, "college", self.college_name)
stud = Student("Ajay")
stud.name
'Ajay'
stud.show()
name Ajay college PWskills
coll = college()
coll._college_name
'PWskills'

Data _____ Page _____

## → Super in concept of Encapsulation:

* It is a function that helps a child class use methods from its parent class.
* It's like a way for the child class to call and use the parent's methods without directly refferring to parent.
* It is a function that is used to call a method from a parent.
* It allows you to access methods in parent class from a child class without needing to refer to parent class by its name.
* Syntax: (i) By __init__ method:

```
class Parent:
    def __init__(self):
        print("Parent")
class Child(Parent):
    def __init__(self):
        super().__init__()
        print("Child")
```

(ii) By Child Class Method:

```
class Parent:
    def display(self):
        print("Parent")
class Child(Parent):
    def display(self):
        super().display()
        print("Child")
```

Date Page

*Eg: Class College:
    def __init__(self):
        self._college_name = "PWSkills"
class Student(College):
    def __init__(self, name):
        self.name = name
        super().__init__(self)
    def show(self):
        print("name", self.name, "College",
              self.~~name~~._college_name)
Stud = Student("Ajay")
stud.show()
'name Ajay college PWskills'

Data Page

## → DUNDER METHOD:

* They are special methods that starts and ends with double underscores.
* They let you customize how objects of your class behave with common operations such as printing, adding or comparing.
* It makes it possible to override or extend the behaviour of operations.
* Common dunder methods and their uses:

(i) __init__():
* This method initializes a new instance of a class.
* Setting initial values

(ii) __str__(self):
* It defines the string representation of an object when str() or print() is called.
* It is used for making output user friendly.

(iii) __rep__(self):
* It is to return a formal string representation of an object, ideally one that could be used to recreate the object.
* It is useful for debugging.

(iv) __add__(self, other):
* It customizes behaviour of + operator.
* It is used for adding or combining their properties.

Date Page

(v) __len__(self):
* It defines behavior for the len() function.
* It is used when you want to return the number of items in a custom object.

(vi) __eq__(self, other):
* It implements the behavior of the == operator.
* It is used for comparing objects for equality.

(vii) __lt__(self, others) and __gt__(self, others)
* It implements < and > operators
* It is used for customizing comparision logic.

(viii) __getitem__(self, key):
* It allows objects to be indexed using [].
* It is used for accessing elements in custom data structures.

(ix) __call__(self, ...):
* It makes an instance callable like a function.
* It is used for turning objects into callable entities.

(x) __setitem__(self, key, value):
* It defines item assignment (obj[key]=value)
* It is used for modifying items in custom objects.

(xi) ~~de~~ __del__(self):
* It is called when an object is deleted or goes out of scopes.
* It is used to cleanup operations.

Data Page

~~*Syntax:~~

*Eg: (i) __init__:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
p = Person("Alice", 30)
print(p.name)
print(p.age)
'Alice'
'30'
```

(ii) __str__() and __repr__():

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author
    def __str__(self):
        return (self.title, "by",
                self.author)
    def __repr__(self):
        return ("Book(title=", self.title,
        ", ", "author=", self.author,
                ")")
```

~~'1984' by~~

```
b = Book("1984", "George Orwell")
print(b)
print(repr(b))
'1984' by George Orwell
'Book(title=1984, author=George
                Orwell)'
```

Date _____ Page _____

(iii) __add__():

```
class Number:
    def __init__(self, value):
        self.value = value
    def __add__(self, other):
        if isinstance(other, Number):
            return Number(self.value + other.value)
        return Not Implemented
    def __str__(self):
        return str(self.value)
num1 = Number(10)
num2 = Number(15)
result = num1 + num2
print(result)
```

'25'

(iv) __len__():

```
class Sentance:
    def __init__(self, text):
        self.text = text.split()
    def __len__(self):
        return len(self.text)
s = Sentance("Hello World")
print(len(s))
```

'4'

Data Page

(v) __eq__(), __lt__(), __gt__():

```
class Box:
    def __init__(self, volume):
        self.volume = volume
    def __eq__(self, other):
        self.volume == other.volume
    def __lt__(self, other):
        return self.volume < other.volume
    def __gt__(self, other):
        return self.volume > other.volume
box1 = Box(100)
box2 = Box(150)
print(box1 == box2)
print(box1 < box2)
print(box1 > box2)
```

'False'
'True'
'False'

(vii) __getitem__() and ~~set~~__setitem__()

```
class CustomList:
    def __init__(self):
        Self.items = {}
    def __getitem__(self, index):
        return self.items.get(index, "Item not found")
    def ~~set~~__setitem__(self, index, value):
        self.item[index] = value
c1 = CustomList()
c1[0] = "Python
c1[1] = "Programming"
print(c1[0])
'Python'
print(c1[1])
'Programming'
```

Date Page

(vii) __call__():

```
class Greeter:
    def __init__(self, name):
        self.name = name
    def __call__(self):
        return (print("Hello", self.name)
greet = Greeter("Adeel")
greet()
'Hello' Adeel'
```

(viii) __iter__() and __next__():

```
class Counter:
    def __init__(self, max_count):
        self.max_count = max_count
        self.current = 0
    def __iter__(self):
        print(self return(self)
    def __next__(self):
        if self.current < sel.max_count:
            self.current += 1
            return self.current
        raise StopIteration
counter = Counter(3)
for num in Counter:
    print(num)
'1 2 3'
```

Date Page

→ Decorators:

* It is a special tool that you can use to add extra features to a function without changing its main code.
* We use @ symbol for decorators
* They are powerful and flexible way to modify or enhance the behavior of functions without changing actual code.
* They are special functions that takes another function as an argument and returns a new function that modifies behavior of orignal.
* Eg:

```
def my_decorator(func):
    def wrapper():
        print("Before function")
        func()
        print("After function")
    return wrapper
@my_decorator
def say_Hello():
    print("Hello")
```

or

```
say_Hello = my_decorator(say_Hello)
```

'Before function'
'Hello'
'After function'

Date Page

→ Property Decorators:

* It makes it easy to manage how class attributes are accessed and updated.
* It like having a helper inside a class that checks or does something when you get or set an attribute, but you don't see helper when you use it.
* It allows you to create managed attributes.
* It enables you to define methods in a class and access them as an attributes.
* Syntax:

```
class ClassName:
    def __init__(self, attribute):
        self._attribute = attribute
    @property
    def attribute(self):
        return self._attribute
    @attribute.setter
    def attribute(self, value):
        if value < 0:
            raise ValueError("Must be +")
        self._attribute = value
    @attribute.deleter
    def attribute(self):
        del self._attribute
```

Date Page

*Eg: class Rectangle:
    def __init__(self, width, height):
        self._width = width
        self._height = height
    @property
    def width(self):
        return self._width
    @width.setter
    def width(self, value):
        if value <= 0:
            raise ValueError("Greater than 0")
        self.width = value
    @property
    def height(self):
        return self._height
    @height.setter
    def height(self, value):
        if value <= 0:
            raise ValueError("Height Greater than 0")
        self._height = value
    @property
    def area(self):
        "Area of rectangle"
        return self._width * self._height

rect = Rectangle(5,10)
rect.area
'50'
rect.width = 15
rect.area
'150'

Date Page

# MATPLOTLIB

## LIBRARIES:

(i) Line Plot:
* It is to display trends over time or a continous set of data points.
* E.g: import matplotlib.pyplot as plt
```
x = [1,2,3,4,5]
y = [2,3,5,7,11]
plt.plot(x, y, marker='o', linestyle='-', color='b')
plt.xlabel('x-axis')
plt.ylabel('y-axis')
plt.title('Line plot')
plt.show()
```
(ii) Scatter Plot: plt.scatter(x,y)
(iii) Bar Plot: plt.bar(categories, values)
(iv) Histogram: plt.hist(data, bins=10)
(v) Box Plot: plt.boxplot(data)
(vi) Pie Chart: plt.pie(values, labels=categories)
(vii) Area Plot: plt.stackplot(x,y)
(viii) Heatmap: plt.imshow(matrix, cmap='viridis') or sns.heatmap(matrix)
(ix) Violin Plot: sns.violinplot(x=category, y=data)
(x) Stem Plot: plt.stem(x,y)
(xi) 3D Plot: import mpl_toolkits.mplot3d, ax=plt.axes(projection='3d') => ax.plot3D(x,y,z)
(xii) Contour Plot: plt.contour(X,Y,Z)
(xiii) Quiver Plot: plt.quiver(X,Y,U,V)
(xiv) Step Plot: plt.step(x,y)
(xv) Polar Plot: plt.polar(theta,r)
(xvi) Error Bar Plot: plt.errorbar(x,y,yerr=error)

Date Page

# NUMPY

(i) Line Plot:

```
import numpy as np
import matplotlib.pyplot as plt
x=np.linspace(0,10,100)
y=np.sin(x)
plt.plot(x,y)
plt.show()
```

(ii) Scatter Plot: plt.scatter(x,y)

(iii) Bar Plot: plt.bar(categories, values)

(iv) Histogram: plt.hist(data, bin=30)

(v) Box Plot: plt.boxplot(data)

(vi) Pie Chart: plt.pie(matrix, data)

(vii) Heatmap: plt.imshow(data, cmap='hot', interpolation=nearest)

(viii) 3D Plot: from mpl_toolkits.mplot3D import Axes3D, ax.plot_surface(X,Y,Z, cmap)

(ix) Contour Plot: plt.contour(X,Y,Z)

(x) Quiver Plot: plt.quiver(X,Y,U,V)

(xi) Step Plot: plt.step(x,y, where='mid')

(xii) Polar Plot: plt.polar(theta, r)

Date Page

# PANDAS

(i) Line Plot:

```
import pandas as pd
import matplotlib.pyplot as plt
data=pd.DataFrame({'Year':[2020,2021,2022,2023],
                   'Sales':[150,200,250,300]})
data.set_index('Year',inplace=True)
data.plot.line()
plt.title("Line plot")
plt.show()
```

(ii) Bar Plot: DataFrame().plot.bar()

(iii) Horizontal Bar Plot: DataFrame().plot.barh()

(iv) Histogram: DataFrame().plot.hist()

(v) Box Plot: DataFrame().plot.box()

(vi) Area Plot: DataFrame().plot.area()

(vii) Scatter Plot: DataFrame().plot.scatter()

(viii) Pie Chart: DataFrame().plot.pie()

(ix) Hexbin Plot: DataFrame().plot.hexbin()

(x) Density Plot: DataFrame().plot.kde()

# SEABORN

(i) ScatterPlot:

```
import seaborn as sns
import matplotlib.pyplot as plt
sns.set(style="whitegrid")
tips=sns.load_dataset("tips")
sns.scatterplot(x="total bill", y="tip", data=tips)
plt.show()
```

(ii) Line Plot: `sns.lineplot(x=x, y=y, data=data)`

(iii) Bar Plot: `sns.barplot(x=x, y=y, data=data)`

(iv) Count Plot: `sns.countplot(x=x, data=data)`

(v) Histogram: `sns.histplot(data=data, kde=True)`

(vi) Box Plot: `sns.boxplot(x=x, y=y, data=data)`

(vii) Voilin Plot: `sns.violinplot(x=x, y=y, data=data)`

(viii) Heatmap: `sns.heatmap(data)`

(ix) Pair Plot: `sns.pairplot(data)`

(x) FacetGrid: `sns.FacetGrid(data)=g, g.map(sns.scatterplot)`

(xi) Strip Plot: `sns.stripplot(x=x, y=y, data=data)`

(xii) Swarm Plot: `sns.swarmplot(x=x, y=y, data=data)`

(xiii) Displot: `sns.histplot(data, kde=True)` or `sns.kdeplot`

(xiv) Joint Plot: `sns.jointplot(x=x, y=y, data=data)`

(xv) Reg Plot: `sns.regplot(x=x, y=y, data=data)`

(xvi) Lm Plot: `sns.lmplot(x=x, y=y, data=data)`

(xvii) Box Plot with hue: `sns.barplot(x=x, y=y, data=data, hue=" ")`

(xviii) Point Plot: `sns.pointplot(x=x, y=y, data=data)`

(xix) Clustermap: `sns.clustermap(data)`

Date Page

# PLOTLY

(i) Line Plot:
import plotly.graph_objects as go
fig = go.Figure(data=go.Scatter(x=[1,2,3,4], y=[10,11,12,13], mode='lines'))
fig.update_layout(title="LinePlot", xaxis_title='X', yaxis_title='Y')
fig.show()

(ii) Scatter Plot: go.Figure(data=go.Scatter(x=x, y=y))

(iii) Bar Plot: go.Figure(data=go.Bar(x=x, y=y))

(iv) Histogram: import numpy as np, go.Figure(data=go.Histogram(x=x))

(v) Box Plot: import numpy as np, go.Figure(data=go.Box(y=y))

(vi) Pie Chart: go.Figure(data=go.Pie(x=x, y=y))

(vii) Area Plot: go.Figure(data=go.Scatter(x=x, y=y))

(viii) Heatmap: import numpy as np, go.Figure(data=go.Heatmap(z=z))

(ix) 3D Scatter Plot: import numpy as np, go.Figure(data=go.Scatter3d(x=x, y=y, z=z))

(x) 3D Surface Plot: import numpy as np, go.Figure(data=go.Surface(z=z, x=x, y=y))

(xi) Contour Plot: import numpy as np, go.Figure(data=go.Contour(z=z, x=x, y=y))

(xii) Bubble Chart: go.Figure(data=go.Scatter(x=x, y=y))

(xiii) Violin Plot: import numpy as np, import pandas as pd, px.violin(x=x, y=y)

(xiv) Radar Chart: go.Figure(data=go.Scatterpolar(r=x, theta=y))

(xv) Sunburst Chart: go.Figure(go.Sunburst(labels, x=x, y=y))

(xvi) Funnel Chart: go.Figure(go.Funnel(x=x, y=y))

Date

# BOKEH

(i) Line Plot:

```
from bokeh.plotting import figure, show
p = figure(title="Line Plot", x_axis_label='X', y_axis_label
    ='Y',)
p.line([1,2,3,4], [10,20,30,40], legend_label='Trend',
    line_width=2)
show(p)
```

(ii) Scatter Plot: p.scatter(x=x, y=y)

(iii) Bar Plot: p.vbar(x=x, y=y)

(iv) Histogram: p.quad(x=x)

(v) Box Plot: from bokeh.io import output_notebook, p.boxplot(data)

(vi) Heatmap: p.image(image=[data], x=0, y=0, dw=10, dh=10)

(vii) Area Plot: p.patch([x=x], [y=y])

(viii) Candlestick Chart: from bokeh.models import (ColumnDataSource, p.segment(x0,y0,x1,y1) p.vbar(x=x, y=y)

(ix) 3D Scatter Plot: go.Figure(data=[go.Scatter3d(x=x, y=y, z=z)])

(x) Heatmap with Hover Tools: p.image(image=[data], x=0, y=0, dw=10, dh=10)

(xi) Bubble Chart: p.scatter(x, y)

(xii) Timeline Plot: DataFrame.circle(x=x, y=y)

(xiii) Polar Plot: DataFrame.line(theta=, r)

(xiv) Radar Plot: Dataframe.patch(x=[], y=[]), DataFrame.line(x=[], y=[])

(xv) Sankey Diagram:

```
import holoviews as hv
hv.extension('bokeh')
hv.Sankey(DataFrame)
show(hv.render(Sankey))
```

Date Page

# FILES, Exceptional handling and Memory Management

## → Files Handling Basics:

### (i) Opening a File:

* You need to use open() function.
* Common Modes:
  - r: Read Mode
  - w: Open file for writing.
  - a: Append Mode: Opens new file and creates new file if file doesnot exists.
  - b: Binary Mode: Used for binary files (images, non text files). Combine with other modes (rb, wb)
  - x: Exclusive creation: Creates a new file and returns an error if the files exists.
* Syntax: file = open (example.txt, 'r')

### (ii) Reading from a File:

* Once a file is opened, you can read its content using different methods:
  - read(): Reads the entire content of file
  - readline(): Reads one line from the file at a time.
  - readlines(): Reads all lines ~~fro~~ and returns them as a lists of strings.
* Syntax: content = file_object.read()
  line = file_object.readline()
  lines = files_object.readlines()

Date Page

(iii) Writing to a File:

* To write data, you must use the file in write ('w') or append ('a') mode.
* • Write(): Writes a single string to the file.
  • writelines(): Writes a list of of strings to file.
* Syntax: file_object.write('Your text here')
  file_object.writelines(['Line 1\n', 'Line 2\n'])
* Eg: file=open('example.txt', w)
  file.write('This is new line.\n')
  file.write('Writing another line')
  file.close()

(iv) Closing a File:

* Always close a file after operations to free up resources.
* Syntax: file_object.close()
* Eg: file = open ('example.txt', 'r')
  content = file.read()
  print (content)
  file.close()
  file.closed()
  'True'
  file.name
  'file.txt'
  file.mode
  'r'

Data Page

## (v) Using with Statement:

* With statement automatically handles file closure, even if an exception occurs.
* Syntax: with open('filename.txt', 'mode') as file_object:

* Eg: with open('example.txt', 'r') as file:
    content = file.read()
    print(content)

## (vi) Handling Exceptions:

* Use try-except to manage potential errors such as file not found.
* ~~Syntax:~~ To handle errors that might occur (eg. file not found), you can use try-except blocks:
* Syntax: try:
    with open('filename.txt', 'mode') as file_object:
  except FileNotFoundError:
    print("File not Found")

* Eg: try:
    with open('nonexistent.txt', 'r') as file:
      Content = file.read()
  except FileNotFoundError:
    print("File does not exists")

Date Page

## (vii) File Pointer Operations:

* You can use tell() to get the current position of the pointer and seek() to move the pointer.
* When you open a file for reading, the file pointer is at the beginning. After reading or writing, you may need to move the pointer using:
  - seek(): Move the file pointer to specific location.
  - tell(): Returns the current position of file pointer.
* Syntax: position = file_object.tell()
  file_object.seek(offset, whence)
* ~~Eg: or~~
* Eg: with open ('example.txt', 'r') as file:
  print (file.tell())
  '0'
  content = file.read(5)
  print (file.tell())
  '5'
  file.seek(0)
  print(file.tell())
  '0'
  print(file.read())

Date Page

## (viii) Reading and Writing Binary Files:

* Use 'rb' or 'wb' for reading and writing binary files, such as image.
* When handling these files, you use the binary mode ('rb', 'wb', 'ab') in the open() function:
  - 'rb': Read a file in binary mode.
  - 'wb': Write to a file in binary mode (creates a new file or overwrites if it exists).
  - 'ab': Append to a file in binary mode.
* Syntax for Reading Binary Files:

```
with open('filename', 'rb') as file_object:
    content = file_object.read()
```

* Example (Reading an Image File):

```
with open('text.txt', 'rb') as file:
    text_data = file.read()
    print(text_data[:20])
'b'this is f line\n this ''
```

* Syntax for Writing Binary File:

```
with open('filename', 'wb') as file_object:
    file_object.write(byte_data)
```

* Example (Writing Binary Data):

```
data = b'this is binary string'
with open('output.bin', 'wb') as file:
    file.write(data)
```

Date

## → Logging:

* Logging is the process of recording runtime events, such as information about the state of the application, which can later be reviewed.
* Python ~~is a log~~ has a logging module.
* Uses of Logging:
  - Error Tracking: Logs can capture errors and provide insights into why and where it is.
  - Execution Flow: Helps track the flow of your program and monitor its behavior.
  - Post-Mortem Analysis: Logs can be reviewed after the application crashes to know what happened before the failure.
  - Performance Monitoring: Track execution times for different parts of your program.
* Logging Levels:
  - DEBUG: detailed info., useful for diagnosing issue.
  - INFO: General info. about execution.
  - WARNING: Something unexpected happened, but the program is still running.
  - ERROR: A more serious issue that prevents part of program from working.
  - CRITICAL: A very serious issue, causing program to stop.

Date

*To use logging, first import logging module.

Syntax:

```
import logging
logging.basicConfig(
        level = logging.DEBUG,
        format = '%(asctime)s - %(levelname)s - %(message)s',
        handlers = [logging.FileHandler('app.log'),
        logging.StreamHandler()])
logging.debug('debug message')
logging.info('info message')
logging.warning('warning message')
logging.error('error message')
logging.critical('critical message')
logging.shutdown()
```

*Eg:

```
import logging
logging.basicConfig(
        level = logging.DEBUG,
        format = '%(asctime)s - %(levelname)s
        - %(message)s',
        filename = 'example.log',
        filemode = 'w' )
logging.debug('Debug message')
logging.info('Info message')
logging.warning('warning message')
logging.error('Error message')
logging.critical('critical message')
```

```
'Debug message'
'Info message'
'warning message'
'error message'
'critical message'
```

Date

## → Debugging:

* It is the process of identifying and fixing errors or issues in your code.
* It involves checking logic errors, syntax problems, or runtime issues that cause the program to behave or fail to run.

### (i) Using Print Statement:

* It is simplest way for debug using print() to display the values of variables and see where your code may be going wrong.
* Pros: (a) Quick and Easy (b) No additional setup.
* Cons: (a) cluttered for larger projects.
  (b) Requires manual removal of print after debugging.
*