Seek and tell()

These functions are used to work with file objects and their positions within a file.

Seek function allows you to move to the current position within a file to a specific point.

with open("data.txt","r") as f:

    print(type(f))

    # move to the 8th byte of the file

    f.seek(8)

    # read next 4 bytes of a file

    data = f.read(4)

    print(data)

tell function returns the current position within the file, in bytes. This can be useful for keeping track of your location within the file.

with open("data.txt","r") as f:

    print(type(f))

    # move to the 8th byte of the file

    f.seek(8)

    # read next 4 bytes of a file

    data = f.read(4)

    # returns the current position which is 8th for now

    print(f.tell())

    print(data)

    # returns the current position after reading the data because you are printing tell() after read() function

    print(f.tell())

**truncate function** : truncate the file to a specific size.

with open("data.txt","w") as f:

    f.write("123456789")

    # truncate method basically cut out other data when you specify a number lets say i specified 5 it means truncate my file data to only 5 characters and remove other one's

    f.truncate(5)

with open("data.txt", "r") as f:

    print(f.read())

    # if you provide n argument it will take it as n not as n-1

Seek function runs to n-1 and truncate function runs to n. seek function expects to start from 0 where as truncate method runs from 1 like when you say truncate(5) it will include 5 characters in a file not 4 means (n-1) no truncate don’t do like that.

Lambda functions

# lambda functions used for write mini function like transforming a value taking cube or square

# my oldie function

# def double(x):

#     return x\*2

# my lambda buddy

double = lambda x: x\*2

print(double(5))

# you can write multiple inputs in lambda like x,z

avg = lambda x,y: (x+y) / 2

# you can write multiple inputs in lambda like x,y,z

avg2 = lambda x,y,z: (x+y+z) / 3

# but when you have multiple lines in your code, you don't wanna do that

# we use lambda functions when we want some minor changes on a value or expressions

print(avg(3,5))

print(avg2(3,5,10))

# we can also pass lambda functions inside another def function

def appl(fx,value):

    return 2 + fx(value)

# you can store it in a variable or you can directly pass lambda x:x\*x to appl() function

square = lambda x:x\*x

print(appl(square,4))

lambda function is a small anonymous function without a name.

lambda x:x\*x 🡪 see it has no name you can pass it directly to any function or any pipeline.

When you say square = lambda x:x\*x 🡪 you are basically assigning a lambda function to a variable.

Map, Filter, Reduce

Map,filter,reduce functions are built in functions that allow you to apply a function to a sequence of elements and return a new sequence. These functions are known as high-order functions because they take other functions as an argument.

When function takes another function as an argument then it is high order function. If function just operates on data not on other functions or return a function then it is said to be as low-order functions. Low-order isn’t a term widely used but based on how we define I can say that.

# map

l = [1,2,4,5,7]

def cube(x):

    return x\*x\*x

newl = list(map(cube,l)) # just map will return the map object for efficiency purposes so we need to convert it into a list

print(newl)

# filter

def filter\_function(a):

    return a>4

l = [1,2,3,4,5,6,7,8]

newl = list(filter(filter\_function,l))

print(newl) # filter elements according to your own desire and contain those elements that gets true for some condition

lets try filter on dictionary:

# filter

def filter\_function(a):

    return a>4

l = {

    1:"Number 01",

    2:"Number 02",

    3:"Number 03",

    4:"Number 04",

    5:"Number 05",

    6:"Number 06"

}

newl = list(filter(filter\_function,l))

print(newl) # filter elements according to your own desire and contain those elements that gets true for some condition

it will consider its keys and return you a list of that as we are converting it into a list. When you try to convert it into a dictionary, it will throw an error.

# map function on a dictionary

l = {

    1:"Number 01",

    2:"Number 02",

    3:"Number 03",

    4:"Number 04",

    5:"Number 05",

    6:"Number 06"

}

newl2 = list(map(lambda x:x\*2,l))

print(newl2) # [2, 4, 6, 8, 10, 12]

same goes for map function, you can’t convert map object into a dictionary. As it takes keys on account to perform operations on.

# reduce function: is used to reduce the sequence of elements by some operations

from functools import reduce

numbers = [1,2,3,4,5]

summ = reduce(lambda x,y:x+y,numbers)

print(summ)

Practice questions for map,filter and reduce:

1. **Basic map**: Given a list of numbers, use the map function to create a new list where each number is squared.
2. **Basic filter**: Given a list of numbers, use the filter function to create a new list containing only the even numbers.
3. **Basic reduce**: Given a list of numbers, use the reduce function to calculate the product of all the numbers in the list.
4. **Combining map and filter**: Given a list of numbers, use map and filter to create a new list containing the squares of only the even numbers.
5. **Using reduce with Strings**: Given a list of strings, use the reduce function to concatenate all the strings into a single string.
6. **Advanced map and filter**: Given a list of dictionaries representing people, use map and filter to create a list of names of people who are older than 18.
7. **Advanced reduce**: Given a list of dictionaries representing products, use the reduce function to calculate the total price of all products.
8. **Combining map, filter, and reduce**: Given a list of numbers, use map, filter, and reduce to calculate the sum of the squares of all even numbers.
9. **Custom map Function**: Implement your own version of the map function without using the built-in map. Call it my\_map.
10. **Custom filter Function**: Implement your own version of the filter function without using the built-in filter. Call it my\_filter.
11. **Custom reduce Function**: Implement your own version of the reduce function without using the built-in reduce. Call it my\_reduce.
12. **Real-World Problem**: Given a list of transactions (dictionaries with amount and type), use map, filter, and reduce to calculate the total balance.

Is and ==

Both of these are comparison operators

a = 3

b = "3"

print(a is b) # compares exact location of object in memory

print(a == b) # compares value

now lets try this with list:

a = [1,2,33]

b = [1,2,33]

print(a is b) # False

print(a == b) # True

on memory it will show false but on == it is showing that it is “True” this is because elements are same and python is smart enough that it created two different lists that user wants another list whether it has same values but it creates another list, as it is mutable so a is b will return false because in memory side, a is not referring to b or a and b are not same on the memory end.

a = (2,3,4)

b = (2,3,4)

print(a is b) # True

print(a == b) # True

now with tuple it is saying true because with immutable objects , python create them one time in memory because python knows that this item is unchangeable so why not just create it one time and if some variable contains same data , just refer it to same memory location because this item is created one time. Same goes for 3,”harry” 3.4, let me show you:

a = 3

b = 3

print(a is b) # True

print(a == b) # True

a = 3.2

b = 3.2

print(a is b) # True

print(a == b) # True

a = "Zain"

b = "Zain"

print(a is b) # True

print(a == b) # True

same goes for None

a = None

b = None

print(a is b) # True

print(a is None) # True

print(a == b) # True

“is” operator matches the identity

“==” operator matches the value and data type also, if value == value then it returns True

a = 2

b = "2"

print(a is b) # False

print(a == b) # False

Introduction to OOPs

All we did till today was procedural programming.

The basic idea of oop in python is to use classes and objects to represent real-world concepts and entities.

A class is a blueprint or template for creating objects. Objects we create through classes are called entities. It defines the properties and methods that an object of the class will have. Properties are the data or state of an object, and method are the actions or behaviors that an object can perform.

Railway form example is best.

An object is an instance of a class and it contains its own data and methods. For example, you could create a class called "Person" that has properties such as name and age, and methods such as speak() and walk(). Each instance of the Person class would be a unique object with its own name and age, but they would all have the same methods to speak and walk.

Oops increase program readability.

Classes and objects

The user-defined objects are created using the class keyword.

Creating a simple class:

class Person:

    name = "Zain"

    occupation = "Data scientist"

    networth = 250000

a = Person()

print(a.name) # Zain

now lets change this name:

class Person:

    name = "Zain"

    occupation = "Data scientist"

    networth = 250000

a = Person()

a.name = "Okasha"

print(a.name) # Okasha

this name only changed for that object or entity only for that, if I create another entity the name will be zain by default.

class Person:

    name = "Zain"

    occupation = "Data scientist"

    networth = 250000

    def info(self):

        print(f"{self.name} is a {self.occupation}")

a = Person()

a.name = "Okasha"

a.occupation = "Finance Accountant"

a.info() # Okasha is a Finance Accountant

I can do something like that but what is self ?

The self parameter is the reference to the current instance of the class, and is used to access variables that belongs to that class.

***Self ka mtlb woh object jiske liye yeh method call kiya jara hai!***

Right now we are giving default attributes to the class.

Constructors

class Person:

    name = "Harry"

    occupation = "Developer"

    def info(self):

        print(f"{self.name} is a {self.occupation}")

a = Person()

a.name = "Diviya"

a.occupation = "HR"

a.info()

we have talked about this right, but lets take a second to thing that why are we giving Harry and developer as default data inside the class, like why? Isn’t classes are used as blueprints? There should be a method in which I just give my data and according to that data it will fill out the attributes that are declared in class.

Now constructors are created using dender method \_\_init\_\_()

class Person:

    def \_\_init\_\_(self):

        print("Im a constructor")

    def info(self):

        print(f"{self.name} is a {self.occupation}")

a = Person() # Im a constructor

whenever you will create a new object , whatever is in the constructor will run automatically.

class Person:

    def \_\_init\_\_(self,n,o):

        print("Im a constructor")

        self.name = n

        self.occupation = o

    def info(self):

        print(f"{self.name} is a {self.occupation}")

a = Person("Zain","Data scientist") # Im a constructor

a.info() # Zain is a Data scientist

A constructor is a special method in a class used to create and initialize an object of a class. There are different types of constructors. Constructor is invoked automatically when an object of a class is created.

A constructor is a unique function that gets called automatically when an object is created of a class. The main purpose of a constructor is to initialize or assign values to the data members of that class. It cannot return any value other than None.

Two types of constructor:

1. Parameterized constructor (take parameters like name and occupation)
2. Default constructor (takes nothing)

Decorators

# decorators

def hello():

    print("Hello world!")

hello()

that is a simple function right, lets say I want this function to first say good morning or greet the user then perform its functionality.

# decorators

def greet(fx):

    def mfx():

        print("Greeting user!, ")

        fx()

        print("Thanks for using this function.")

    return mfx

@greet

def hello():

    print("Hello world!")

hello()

it will give following output:

*Greeting user!,*

*Hello world!*

*Thanks for using this function.*

# decorators

def greet(fx):

    def mfx():

        print("Greeting user!, ")

        fx()

        print("Thanks for using this function.")

    return mfx

# @greet

def hello():

    print("Hello world!")

# hello()

# this is also can be written as without @greet

greet(hello)() # this will throw same result as used with @greet

# but writing @greet is pretty simple way so we invoke function like normally

# greet(hello)() is kinda complex than writing this hello()

Query: Why it is not working when i use **greet(hello)()** then it is working but when im using **greet(hello)** it is not working why? dont we pass arguments like this in normal functions we write?

Answer:

greet(hello) returns mfx (the wrapped version of hello).

() after it calls the returned function (mfx), which then:

Prints "Greeting user!,"

Calls hello(), which prints "Hello world!"

Prints "Thanks for using this function.”

Passing an argument part:

# decorators

def greet(fx):

    def mfx():

        print("Greeting user!, ")

        fx()

        print("Thanks for using this function.")

    return mfx

def addDecorator(fx):

    def mfx(\*args,\*\*kwargs):

        # \*args is a way that takes arguments as a tuple

        # \*\*kwargs is a way that takes arguments as a dictionary

        print(\*args, " ", \*\*kwargs)

        print("welcome to add function")

        addo = fx(\*args,\*\*kwargs)

        print(addo)

        print("Exiting add function.")

    return mfx

@greet

def hello():

    print("Hello world!")

# now lets say i have a add function that takes 2 parameters

@addDecorator

def add(a,b):

    return a+b

add(2,3)

Getters and setters

Getters in Python are methods that are used to access the values of an object's properties. They are used to return the value of a specific property, and are typically defined using the @property decorator.

# getters

class MyClass:

    def \_\_init\_\_(self,value):

        self.\_value = value

    def show(self):

        print(f"Value is {self.\_value}")

    @property

    def Ten\_value(self):

        return 10 \* self.\_value

    @Ten\_value.setter

    def Ten\_value(self,new\_value):

        self.\_value = new\_value

obj = MyClass(10)

print(obj.Ten\_value)

# obj.Ten\_value = 67 # i cant use that when i have no setter defined in the class

obj.show()

obj.\_value = 20

obj.show()

# setters

obj.Ten\_value = 77

obj.show()

# like we created a method Ten\_value but we are treating it as an attribute or a property

It is important to note that the getters do not take any parameters and we cannot set the value through getter method.For that we need setter method which can be added by decorating method with @property\_name.setter