Write a program to check the number entered by user is even or odd. Program should accept integer digits only

try:

    def even\_odd(num):

        if num % 2 == 0:

            print("even")

        else:

            print("odd")

    even\_odd(int(input("Enter a number: ")))

except ValueError:

    print("Invalid input")

Illustrate types of arguments used in python function with example

In Python, there are several types of arguments that can be used in a function:

1. **Positional Arguments**: These are arguments that are passed to a function in the order they are defined in the function signature.

Example:

def greet(name, greeting):

return f"{greeting}, {name}!"

print(greet("Alice", "Hello")) # Output: Hello, Alice!

1. **Keyword Arguments**: These are arguments preceded by their parameter names when passed to a function. This allows you to specify arguments in any order.

Example:

def greet(name, greeting):

return f"{greeting}, {name}!"

print(greet(greeting="Hello", name="Bob")) # Output: Hello, Bob!

1. **Default Arguments**: These are arguments that have default values specified in the function signature. If a value is not provided when the function is called, the default value is used.

Example:

def greet(name, greeting="Hello"):

return f"{greeting}, {name}!"

print(greet("Alice")) # Output: Hello, Alice!

print(greet("Bob", "Hi")) # Output: Hi, Bob!

1. **Variable-Length Positional Arguments (\*args)**: These are arguments that allow you to pass a variable number of positional arguments to a function. They are captured as a tuple inside the function.

Example:

def sum\_numbers(\*args):

total = 0

for num in args:

total += num

return total

print(sum\_numbers(1, 2, 3)) # Output: 6

1. \*\*Variable-Length Keyword Arguments (**kwargs)**: These are arguments that allow you to pass a variable number of keyword arguments to a function. They are captured as a dictionary inside the function.

Example:

def print\_info(\*\*kwargs):

for key, value in kwargs.items():

print(f"{key}: {value}")

print\_info(name="Alice", age=30, city="New York")

# Output:

# name: Alice

# age: 30

# city: New York

Discuss the polymorphism concept in python create suitable python classes

Polymorphism is a fundamental concept in object-oriented programming that allows objects of different classes to be treated as objects of a common superclass. This means that a single interface can be used to represent different types of objects, enabling code to be written in a more generic and reusable way.

In Python, polymorphism is achieved through method overriding and method overloading.

1. **Method Overriding**: This occurs when a subclass provides a specific implementation of a method that is already defined in its superclass. The method in the subclass overrides the method in the superclass.
2. **Method Overloading**: Although Python doesn't support method overloading in the traditional sense (having multiple methods with the same name but different signatures), polymorphism can still be achieved through a single method that behaves differently depending on the type or number of arguments passed to it.

Example:

class IND():

    def capital(self):

        print("The New Delhi is capital of indeia")

    def language(self):

        print("Hindi is the most widely spoken language of india")

    def type(self):

        print("India is a developing country")

class USA():

    def capital(self):

        print("Washington, D.C. is the capital of USA")

    def language(self):

        print("English is the primary language of USA")

    def type(self):

        print("USA is a developed country")

a = IND()

b = USA()

for country in (a,b):

    country.capital()

    country.language()

    country.type()

Write a python program for the following.

1. Create list of fruits
2. Add new fruit in list.
3. sort the list.
4. delete last fruit name from list

fruit = ["Mango","Banana","Orange","Apple","Graps"]

print(fruit)

*# Add a new fruit in list*

fruit.append("Pineapple")

print(fruit)

*# short the list*

fruit.sort()

print(fruit)

*# Delete the last fruit in the list*

fruit.pop()

print(fruit)

*# Delete the fruit in the list*

fruit.remove("Apple")

print(fruit)

Write a python function to check the given number is even or odd. Handle suitable exceptions

try:

    def even\_odd(num):

        if num % 2 == 0:

            print("even")

        else:

            print("odd")

    even\_odd(int(input("Enter a number: ")))

except ValueError:

    print("Invalid input")

Explain constructor concept in python with example

In Python, there is only one type of constructor, which is the **\_\_init\_\_()** method. This method is called automatically when an object is instantiated from a class, and it is used to initialize the object's attributes.

The **\_\_init\_\_()** method is the most commonly used constructor in Python. It allows you to perform any necessary setup when creating an instance of a class by initializing the instance variables.

Here's an example demonstrating the **\_\_init\_\_()** constructor in Python:

class Person:

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

def display\_info(self):

print(f"Name: {self.name}, Age: {self.age}")

# Creating an instance of the Person class

person1 = Person("Alice", 30)

# Accessing attributes and calling methods

person1.display\_info() # Output: Name: Alice, Age: 30

**Types of constructors :**

* **default constructor:** The default constructor is a simple constructor which doesn’t accept any arguments. Its definition has only one argument which is a reference to the instance being constructed.
* **parameterized constructor:** constructor with parameters is known as parameterized constructor. The parameterized constructor takes its first argument as a reference to the instance being constructed known as self and the rest of the arguments are provided by the programmer.

**Example of default constructor :**

class GeekforGeeks:

# default constructor

def \_\_init\_\_(self):

self.geek = "GeekforGeeks"

# a method for printing data members

def print\_Geek(self):

print(self.geek)

# creating object of the class

obj = GeekforGeeks()

# calling the instance method using the object obj

obj.print\_Geek()

**Example of the** **parameterized constructor :**

class Addition:

first = 0

second = 0

answer = 0

# parameterized constructor

def \_\_init\_\_(self, f, s):

self.first = f

self.second = s

def display(self):

print("First number = " + str(self.first))

print("Second number = " + str(self.second))

print("Addition of two numbers = " + str(self.answer))

def calculate(self):

self.answer = self.first + self.second

# creating object of the class

# this will invoke parameterized constructor

obj1 = Addition(1000, 2000)

# creating second object of same class

obj2 = Addition(10, 20)

# perform Addition on obj1

obj1.calculate()

# perform Addition on obj2

obj2.calculate()

# display result of obj1

obj1.display()

# display result of obj2

obj2.display()

Write a python program to create an employee. txt file and store employee name and address

fileName = "emp.txt"

with open(fileName,'w') as file:

    file.write("Your name is Chhagan Kumawat")

    file.write("\nYour address is Pune")

print(f"'{fileName}' created successfully")

using valuer input from user

fileName = "emp.txt"

*# with open(fileName,'w') as file:*

*#     file.write("Your name is Chhagan Kumawat")*

*#     file.write("\nYour address is Pune")*

*# print(f"'{fileName}' created successfully")*

def write\_file(fileName,data):

    with open(fileName,'w') as file:

        file.write(data + '\n')

if \_\_name\_\_ == "\_\_main\_\_":

    num = int(input("Enter Number of employee: "))

    for i in range(1,num +1):

        name = input(f"Enter employee name: {i} ")

        address = input(f"Enter address: {i}")

        data = f"\n{name} \n{address}"

        write\_file(fileName,data)

print("Data written successfully")

Write a python program to create “employee” collection with fields” (ID, name, address, phone email and dept) in mongoDB. Prform the following operations. [5] i) Display all employees in “Accounts” department ii) Delete employee with ID - 210345 iii) Update phone with new phone for employee ID -123

import pymongo

*# Establish a connection to MongoDB*

client = pymongo.MongoClient("mongodb://localhost:27017/")

*# Create or access the "company" database*

db = client["company"]

*# Create or access the "employees" collection*

collection = db["employees"]

*# i) Display all employees in "Accounts" department*

def display\_accounts\_department():

    accounts\_employees = collection.find({"dept": "Accounts"})

    print("Employees in Accounts department:")

    for employee in accounts\_employees:

        print(employee)

*# ii) Delete employee with ID - 210345*

def delete\_employee\_by\_id(employee\_id):

    collection.delete\_one({"ID": employee\_id})

    print(f"Employee with ID {employee\_id} deleted successfully.")

*# iii) Update phone with new phone for employee ID - 123*

def update\_phone(employee\_id, new\_phone):

    collection.update\_one({"ID": employee\_id}, {"$set": {"phone": new\_phone}})

    print(f"Phone updated successfully for employee with ID {employee\_id}.")

if \_\_name\_\_ == "\_\_main\_\_":

*# Insert sample data*

    employees\_data = [

        {"ID": 123, "name": "Alice", "address": "123 Main St", "phone": "123-456-7890", "email": "alice@example.com", "dept": "Accounts"},

        {"ID": 210345, "name": "Bob", "address": "456 Elm St", "phone": "456-789-0123", "email": "bob@example.com", "dept": "HR"},

        {"ID": 567, "name": "Charlie", "address": "789 Oak St", "phone": "789-012-3456", "email": "charlie@example.com", "dept": "Accounts"}

    ]

    collection.insert\_many(employees\_data)

*# i) Display all employees in "Accounts" department*

    display\_accounts\_department()

*# ii) Delete employee with ID - 210345*

    delete\_employee\_by\_id(210345)

*# iii) Update phone with new phone for employee ID - 123*

    update\_phone(123, "987-654-3210")

*# Display updated Accounts department*

    print("\nUpdated Accounts department:")

    display\_accounts\_department()

What is tuple

A tuple is an ordered, immutable collection of elements in Python. This means that once a tuple is created, its elements cannot be changed, added, or removed. Tuples are similar to lists, but they have the key distinction of immutability.

You can create a tuple by enclosing elements within parentheses **()** and separating them by commas. Here's an example:

my\_tuple = (1, 2, 3, "hello", True)

Tuples can contain elements of different types, including numbers, strings, booleans, or even other tuples. They can also be empty. For example:

empty\_tuple = ()

single\_element\_tuple = (42,)

mixed\_tuple = (1, "hello", True, (4, 5, 6))

One of the main use cases for tuples is when you want to store a fixed sequence of values that you don't intend to modify. For example, you might use tuples to represent coordinates, database records, or configurations.

Here are some key characteristics of tuples:

1. **Ordered**: Tuples maintain the order of elements as they are inserted.
2. **Immutable**: Once a tuple is created, its elements cannot be changed, added, or removed.
3. **Heterogeneous**: Tuples can contain elements of different data types.
4. **Iterable**: You can iterate over the elements of a tuple using a loop or other iterable methods like **for** loops or list comprehensions.

Here's an example demonstrating these characteristics:

**my\_tuple = (1, "apple", True, 3.14)**

**# Accessing elements**

**print(my\_tuple[0]) # Output: 1**

**# Iterating over elements**

**for item in my\_tuple:**

**print(item)**

**# Attempting to modify a tuple (results in an error)**

**# my\_tuple[0] = 2 # TypeError: 'tuple' object does not support item assignment**

What is the difference between list and tuple?

| **Feature** | **List** | **Tuple** |
| --- | --- | --- |
| Mutability | Mutable | Immutable |
| Syntax | Defined using square brackets **[ ]** | Defined using parentheses **( )** |
| Usage | Used for mutable collections | Used for immutable collections |
| Performance | Slightly slower performance due to mutability | Generally faster to access |

What is dictionary

A dictionary in Python is an unordered, mutable collection of key-value pairs. Each key-value pair in a dictionary maps the key to its associated value. Dictionaries are also known as "associative arrays" or "hash maps" in other programming languages.

Here's an example of a dictionary in Python:

# Creating a dictionary

my\_dict = {

"name": "Alice",

"age": 30,

"city": "New York"

}

# Accessing values

print(my\_dict["name"]) # Output: Alice

print(my\_dict["age"]) # Output: 30

# Modifying values

my\_dict["age"] = 35

print(my\_dict["age"]) # Output: 35

# Adding new key-value pairs

my\_dict["email"] = "alice@example.com"

print(my\_dict) # Output: {'name': 'Alice', 'age': 35, 'city': 'New York', 'email': 'alice@example.com'}

# Deleting a key-value pair

del my\_dict["city"]

print(my\_dict) # Output: {'name': 'Alice', 'age': 35, 'email': 'alice@example.com'}

In this example:

* We create a dictionary **my\_dict** containing key-value pairs.
* We access values in the dictionary using their keys.
* We modify the value associated with a key.
* We add new key-value pairs to the dictionary.
* We delete a key-value pair from the dictionary using the **del** statement.

Here are some key characteristics of dictionaries:

1. **Unordered**: Unlike lists and tuples, dictionaries do not maintain the order of elements. Items are stored and retrieved by their keys, not by their positions.
2. **Mutable**: Dictionaries can be modified after creation. You can add, remove, or modify key-value pairs.
3. **Keys**: Keys in a dictionary must be unique and immutable. Common types used as keys include strings, numbers, and tuples.
4. **Values**: Values in a dictionary can be of any data type and can be duplicated.

diferante between tuple and dictionary

| **Feature** | **Tuple** | **Dictionary** |
| --- | --- | --- |
| Syntax | Defined using parentheses **( )** | Defined using curly braces **{ }** |
| Access | Accessed by index | Accessed by key |
| Mutability | Immutable | Mutable |
| Ordering | Ordered | Unordered |
| Use Cases | Fixed collection of elements | Collection of key-value pairs |
| Elements | Homogeneous (elements of the same type) | Heterogeneous (keys and values can be any type) |
| Storage | Less memory overhead (no need to store keys) | More memory overhead (need to store keys) |
| Example | **my\_tuple = (1, 2, 3)** | **my\_dict = {"name": "Alice", "age": 30}** |

What is set? Explain with example.

A set in Python is an unordered collection of unique elements. Sets are similar to lists and tuples, but unlike lists and tuples, sets do not allow duplicate elements. Sets are highly optimized for membership testing, which makes them ideal for tasks like removing duplicates from a sequence or checking for the presence of specific elements.

Here's an example of a set in Python:

# Creating a set

my\_set = {1, 2, 3, 4, 5}

# Displaying the set

print(my\_set) # Output: {1, 2, 3, 4, 5}

# Adding elements to the set

my\_set.add(6)

print(my\_set) # Output: {1, 2, 3, 4, 5, 6}

# Adding duplicate elements (no effect)

my\_set.add(2)

print(my\_set) # Output: {1, 2, 3, 4, 5, 6}

# Removing elements from the set

my\_set.remove(3)

print(my\_set) # Output: {1, 2, 4, 5, 6}

# Membership testing

print(2 in my\_set) # Output: True

print(3 in my\_set) # Output: False

Write a program to retrieve and display employee details from “Employee” collection stored in mangoDB database

import pymongo

def display\_employee\_details():

# Establish connection to MongoDB

client = pymongo.MongoClient("mongodb://localhost:27017/")

# Access the "company" database and the "Employee" collection

db = client["company"]

collection = db["Employee"]

# Retrieve all documents from the collection

employee\_details = collection.find()

# Display employee details

print("Employee Details:")

for employee in employee\_details:

print(employee)

if \_\_name\_\_ == "\_\_main\_\_":

display\_employee\_details()

Write a program to update the employee details stored in “Employee” collection stored in Mangodb database

import pymongo

def update\_employee\_details(employee\_id, new\_details):

# Establish connection to MongoDB

client = pymongo.MongoClient("mongodb://localhost:27017/")

# Access the "company" database and the "Employee" collection

db = client["company"]

collection = db["Employee"]

# Update employee details

collection.update\_one({"ID": employee\_id}, {"$set": new\_details})

print(f"Employee with ID {employee\_id} details updated successfully.")

if \_\_name\_\_ == "\_\_main\_\_":

# Example of updating employee details

employee\_id = 123 # ID of the employee to update

new\_details = {

"name": "Updated Name",

"address": "Updated Address",

"phone": "Updated Phone",

"email": "updated\_email@example.com",

"dept": "Updated Department"

}

update\_employee\_details(employee\_id, new\_details)

Write python program to read “employee” . txt” file and display alternate employee record.

def display\_alternate\_records(file\_name):

    try:

        with open(file\_name, 'r') as file:

            lines = file.readlines()

            for i in range(0, len(lines), 2):

                print(lines[i].strip())

    except FileNotFoundError:

        print(f"File '{file\_name}' not found.")

    except Exception as e:

        print(f"An error occurred: {e}")

if \_\_name\_\_ == "\_\_main\_\_":

    file\_name = "emp.txt"

    display\_alternate\_records(file\_name)