Chapter 2: Class and Object in Python with GIS Application Emphasis

1 Why Using Object-Oriented Programming (OOP)?

Object-Oriented Programming (OOP) is a programming paradigm that uses "objects" to design software. Each object can represent a real-world entity, making the code more intuitive. Here's why you should consider using OOP:

- **Modularity**: OOP helps to organize complex code into simple, reusable modules.
- **Scalability**: Objects can be extended without modifying existing code.
- **Maintainability**: Easy to debug, update, and manage.
- **Abstraction**: Simplifies complex realities by modeling entities.
- **Reusability**: Promotes code reuse through inheritance.

2 Characteristics of OOP

- **Encapsulation**: Bundling data and methods that operate on the data within one unit (class).
- **Inheritance**: Creating new classes from existing classes, inheriting attributes and behaviors.
- **Polymorphism**: Allowing entities to be treated as instances of their parent class.
- **Abstraction**: Hiding complex implementation details and showing only the essentials.

3 Defining Class

A class is a blueprint for creating objects. For example, in a GIS application, a Point class might represent geographic coordinates.

```
# python
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f"Point({self.x}, {self.y})"
```

4 Defining Object and Attributes

Objects are instances of classes. Attributes are the data stored in objects.

```
# python
# Create an instance of Point
point1 = Point(10, 20)
print(point1)
```

5 Object Initialization and Self

The `__init__` method initializes an object's attributes. The self keyword represents the instance of the class.

```
# python
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

point1 = Point(10, 20)
print(f"X: {point1.x}, Y: {point1.y}")
```

Table 1: More on self: Understanding self in Python Classes

1. Defining a Class and Object Initialization
A class is a blueprint for creating objects. It can include attributes (data) and methods (functions). The **__init__** method is a special method called when an object is instantiated, and self is a reference to the current instance of the class.
Here's an example using a simple Point class:

```
# python
class Point:
    def __init__(self, x, y):
        self.x = x  # `self.x` refers to the instance's x attribute
        self.y = y  # `self.y` refers to the instance's y attribute

    def display(self):
        return f"Point({self.x}, {self.y})"

# Creating an object of class Point
point1 = Point(10, 20)
print(point1.display())  # Output: Point(10, 20)
```

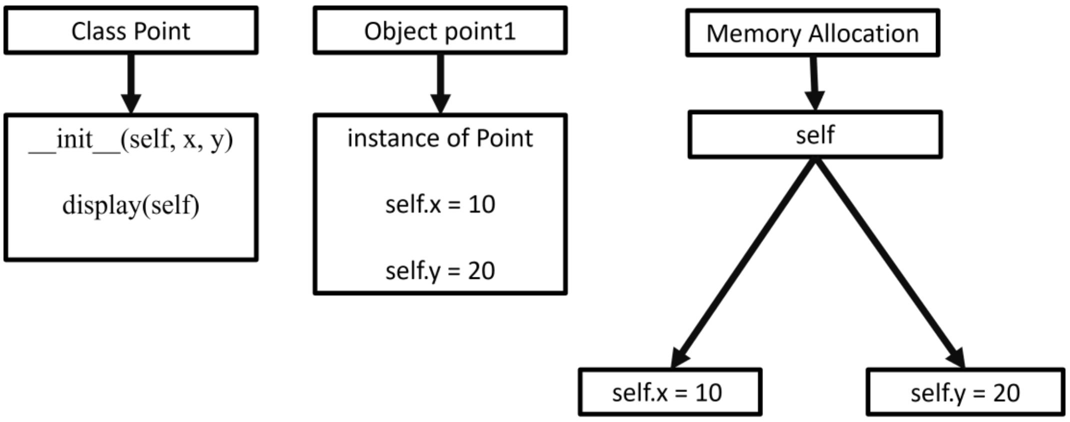
2. The Role of self

- **Instance Variables**: self allows each object to have its own set of attributes.
- **Distinguishing Variables**: It differentiates instance variables (unique to each instance) from class variables (shared among all instances).

When the Point object point1 is created, the __init__ method initializes point1's attributes (x and y). The self keyword in __init__ refers to the instance of the class (point1).

- **Diagram Explanation**

A conceptual diagram to illustrate how self works in the class:



6 Using Classes in Another Python Module

You can define classes in one module and use them in another. Here's an example:
In geometry.py:

```
# python
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance_to(self, other_point):
        return ((self.x - other_point.x)**2 + (self.y - other_point.y)**2)**0.5
```

In main.py:

```
# python
from geometry import Point

point1 = Point(10, 20)
point2 = Point(30, 40)

print(f"Distance: {point1.distance_to(point2)}")
```

Integrating GIS Applications

Let's integrate these concepts into a simple GIS application that manipulates geographic data using Python.

```
# python
from geometry import Point
class GISPoint(Point):
    def __init__(self, x, y, name):
        super().__init__(x, y)
        self.name = name

    def __repr__(self):
        return f"GISPoint({self.name}: {self.x}, {self.y})"

# Create GIS Points
point1 = GISPoint(10, 20, "LocationA")
point2 = GISPoint(30, 40, "LocationB")

# Calculate the distance between points
print(f"Distance between {point1.name} and {point2.name}:
{point1.distance_to(point2)}")
```

7 Summary

- **OOP**: Provides modularity, scalability, maintainability, abstraction, and reusability.
- **Classes**: Define a blueprint for objects.
- **Objects**: Instances of classes with attributes.
- **Initialization**: __init__ method for setting object attributes.
- **Using Classes**: Can be defined in one module and used in another.
- **GIS Integration**: Classes and OOP principles applied to GIS data in Python.

8 Practical demonstration

Practical exercise and demonstration for GIS students on Chapter 2: Class and Object in Python:

8.1 Discussions

:

1. **Why Use Object-Oriented Programming (OOP)?**
   - **Discussion**: Explain the benefits of OOP, such as modularity, reusability, and ease of maintenance. Discuss how OOP can help in managing complex GIS data by encapsulating data and functions into objects.
2. **Characteristics of OOP**
   - **Discussion**: Introduce the four main characteristics of OOP: Encapsulation, Inheritance, Polymorphism, and Abstraction. Provide examples of how these can be applied in GIS.
3. **Defining a Class**
   - **Exercise**: Define a class GISPoint that represents a point with latitude and longitude.

8.2 Exercise: Creating a GIS Point Class

8.2.1 Define a class GISPoint that represents a point with latitude and longitude

```
class GISPoint:
    def __init__(self, latitude, longitude):
        self.latitude = latitude
        self.longitude = longitude
```

- Defining Objects and Attributes

Exercise: Create objects of the GISPoint class and initialize them with specific latitude and longitude values.
point1 = GISPoint(9.145, 40.489)
point2 = GISPoint(7.946, 39.789)
Object Initialization

8.2.2 Exercise: Add a method to the GISPoint class to display the coordinates of the point.

```
class GISPoint:
    def __init__(self, latitude, longitude):
        self.latitude = latitude
        self.longitude = longitude

    def display_coordinates(self):
        print(f"Latitude: {self.latitude}, Longitude: 
{self.longitude}")

point1 = GISPoint(9.145, 40.489)
point1.display_coordinates()
```

8.3 Additional Exercises:

- **Exercise 1**: Extend the GISPoint class to include a method that calculates the distance to another point.
- **Exercise 2**: Create a GISLine class that represents a line made up of multiple GISPoint objects.

This exercise will help GIS students grasp the fundamental concepts of OOP in Python and see how these concepts can be applied to real-world GIS problems.

9 Explanation for exercise

9.1 The code for both exercises (Exercise 1 and Exercise 2 – see Chapter 2 section 8.3)

9.1.1 Exercise 1: Extend the GISPoint Class to Include a Method that Calculates the Distance to Another Point

We'll use the Haversine formula to calculate the distance between two points on the Earth's surface.

```
import math

class GISPoint:
    def __init__(self, latitude, longitude):
        self.latitude = latitude
        self.longitude = longitude

    def display_coordinates(self):
        print(f"Latitude: {self.latitude}, Longitude: {self.longitude}")

    def distance_to(self, other_point):
        # Radius of the Earth in kilometers
        R = 6371.0

        lat1 = math.radians(self.latitude)
        lon1 = math.radians(self.longitude)
        lat2 = math.radians(other_point.latitude)
        lon2 = math.radians(other_point.longitude)

        dlat = lat2 - lat1
        dlon = lon2 - lon1

        a = math.sin(dlat / 2)**2 + math.cos(lat1) * math.cos(lat2) * math.sin(dlon / 2)**2
        c = 2 * math.atan2(math.sqrt(a), math.sqrt(1 - a))

        distance = R * c
        return distance

# Example usage
point1 = GISPoint(9.145, 40.489)
point2 = GISPoint(7.946, 39.789)
print(f"Distance: {point1.distance_to(point2)} km")
```

9.1.2 Exercise 2: Create a GISLine Class that Represents a Line Made Up of Multiple GISPoint Objects

```
import math

class GISPoint:
    def __init__(self, lat, lon, elevation=0):
        self.lat = lat
        self.lon = lon
        self.elevation = elevation

    def distance_to(self, other_point):
        R = 6371.0  # Radius of the Earth in kilometers

        lat1 = math.radians(self.lat)
        lon1 = math.radians(self.lon)
        lat2 = math.radians(other_point.lat)
        lon2 = math.radians(other_point.lon)

        dlat = lat2 - lat1
        dlon = lon2 - lon1

        a = math.sin(dlat / 2)**2 + math.cos(lat1) * math.cos(lat2) * math.sin(dlon / 2)**2
        c = 2 * math.atan2(math.sqrt(a), math.sqrt(1 - a))

        distance = R * c

        return distance

    def display_coordinates(self):
        print(f"GISPoint({self.lat}, {self.lon}, {self.elevation})")

    def __repr__(self):
        return f"GISPoint({self.lat}, {self.lon}, {self.elevation})"

class GISLine:
    def __init__(self):
        self.points = []

    def add_point(self, point):
        self.points.append(point)

    def total_length(self):
        if len(self.points) < 2:
            return 0
        total_distance = 0
        for i in range(len(self.points) - 1):
            total_distance += self.points[i].distance_to(self.points[i + 1])
        return total_distance

    def display_line(self):
        for point in self.points:
            point.display_coordinates()

# Example usage
line = GISLine()
line.add_point(GISPoint(9.145, 40.489))
line.add_point(GISPoint(7.946, 39.789))
line.add_point(GISPoint(8.980, 38.757))
line.display_line()
print(f"Total Length: {line.total_length()} km")
```

9.2 Explanation

1. **Exercise 1**:
    - The distance_to method in the GISPoint class calculates the distance to another GISPoint using the Haversine formula.
2. **Exercise 2**:
    - The GISLine class manages a list of GISPoint objects.
    - The add_point method adds a new point to the line.
    - The total_length method calculates the total length of the line by summing the distances between consecutive points.
    - The display_line method prints the coordinates of all points in the line.

10 Demonstration: Expanding the class

Let's see into how we can expand our simple class to include more attributes and methods, and integrate it into a real-world GIS application. We'll demonstrate this with a CSV file that contains point pairs with attributes like place names, coordinates (x, y), and elevation.

10.1 Expanding the Class

First, let's expand the Point class to include additional attributes and methods for calculating distances:

```
# python
import math

class GISPoint:
    def __init__(self, name, x, y, elevation):
        self.name = name
        self.x = x
        self.y = y
        self.elevation = elevation

    def horizontal_distance(self, other_point):
        return math.sqrt((self.x - other_point.x) ** 2 + (self.y - other_point.y) ** 2)

    def vertical_distance(self, other_point):
        return abs(self.elevation - other_point.elevation)

    def surface_distance(self, other_point):
        horizontal_dist = self.horizontal_distance(other_point)
        vertical_dist = self.vertical_distance(other_point)
        return math.sqrt(horizontal_dist ** 2 + vertical_dist ** 2)

    def __repr__(self):
        return f"GISPoint({self.name}: {self.x}, {self.y}, {self.elevation})"
```

10.2 Parsing CSV File and Calculating Distances

Let's assume our CSV file (points.csv) is structured as follows:

```
place1name,x1,y1,elevation1,place2name,x2,y2,elevation2
LocationA,10,20,300,LocationB,30,40,350
```

We can read this CSV file, create GISPoint objects, and calculate the distances:

```
# python
import csv

# Function to read CSV file and create point pairs
def read_points_from_csv(file_path):
    point_pairs = []
    with open(file_path, newline='') as csvfile:
        reader = csv.reader(csvfile)
        next(reader)  # Skip header
        for row in reader:
            point1 = GISPoint(row[0], float(row[1]), float(row[2]), float(row[3]))
            point2 = GISPoint(row[4], float(row[5]), float(row[6]), float(row[7]))
            point_pairs.append((point1, point2))
    return point_pairs

# Function to calculate distances
def calculate_distances(point_pairs):
    for point1, point2 in point_pairs:
        horizontal_dist = point1.horizontal_distance(point2)
        vertical_dist = point1.vertical_distance(point2)
        surface_dist = point1.surface_distance(point2)
        print(f"Distances between {point1.name} and {point2.name}:")
        print(f"Horizontal Distance: {horizontal_dist}")
        print(f"Vertical Distance: {vertical_dist}")
        print(f"Surface Distance: {surface_dist}")
        print("-" * 40)
```

```
# Example usage
point_pairs = read_points_from_csv('points.csv')
calculate_distances(point_pairs)
```

10.3 Real-World Application

GIS professionals can use such classes to manage and analyze spatial data in various ways:

- **Data Management**: Create and manipulate spatial data points with attributes like location, elevation, and names.
- **Distance Calculations**: Calculate distances for infrastructure planning, environmental studies, and navigation.
- **Spatial Analysis**: Perform complex spatial analyses like proximity analysis, network analysis, and topographic analysis.
- **Visualization**: Integrate with visualization libraries (e.g., Matplotlib, Folium) to display data on maps and charts.

For example, a city planner could use this code to calculate distances between different infrastructure points, such as schools, hospitals, and parks, to optimize resource allocation and urban planning.

10.4 Summary

- We expanded our GISPoint class to include more attributes and methods for distance calculations.
- We demonstrated how to read point data from a CSV file and calculate horizontal, vertical, and surface distances.
- We discussed how GIS professionals can use such classes to manage and analyze spatial data.

11 Appendix

11.1 The Haversine formula

The Haversine formula is a mathematical equation used to calculate the great-
circle distance between two points on the surface of a sphere, given their latitudes and longitudes. Thi
s formula is especially useful in navigation and geography for finding the shortest distance over the ea
rth's surface.
The formula is defined as:

$$d = 2r \cdot \arcsin\left(\sqrt{\sin^2\left(\frac{\Delta\phi}{2}\right) + \cos(\phi_1)\cdot\cos(\phi_2)\cdot\sin^2\left(\frac{\Delta\lambda}{2}\right)}\right)$$

where:

- d is the distance between the two points.
- r is the radius of the Earth (mean radius = 6,371 km).
- ϕ_1 and ϕ_2 are the latitudes of the two points in radians.
- λ_1 and λ_2 are the longitudes of the two points in radians.
- $\Delta\phi = \phi_2 - \phi_1$ is the difference in latitudes.
- $\Delta\lambda = \lambda_2 - \lambda_1$ is the difference in longitudes.

Steps to Calculate the Distance

- Convert degrees to radians: Use the formula radians=degrees×π180.
- Calculate the differences: Find the differences in latitudes and longitudes in radians.
- Apply the Haversine formula: Compute the values using the formula and obtain the distance.