SDEV 1001

Programming Fundamentals

Classes and Objects - 3

A LEADING POLYTECHNIC COMMITTED TO YOUR SUCCESS

Expectations - What I expect from you

- No Late Assignments

- No Cheating

- Be a good classmate

- Don’t waste your time

- Show up to class

Agenda

On the right is what we will cover today.

- Building Relationships with Classes
- Example: Team and Player Classes
- Creating and Using Relationships
- Helper Methods for Searching and Access
- Aggregating Data Across Objects
- Customizing Object Comparison
- Summary

Building Relationships with Classes

Classes can represent not just individual objects, but also relationships between objects.

- Example: A `Team` class can contain multiple `Player` objects.
- This lets you model real-world groupings and interactions.

Example: Team and Player Classes

On the right, we will create a `Team` class that contains multiple `Player` objects.

A `Player` class will represent individual players with attributes like name and position.

A `Team` class will manage a collection of `Player` objects, allowing you to add players and list them.

```python
class Player:
    def __init__(self, name, position):
        self.name = name
        self.position = position

    def __str__(self):
        return f"{self.name} ({self.position})"

class Team:
    def __init__(self, name):
        self.name = name
        self.players = []

    def add_player(self, player):
        self.players.append(player)

    def list_players(self):
        for player in self.players:
            print(player)
```

Creating and Using Relationships

Here's how you can create a `Team` and add `Player` objects to it:

```
team = Team("Oilers")
team.add_player(Player("Stuart", "Goalie"))
team.add_player(Player("Connor", "Forward"))

print(f"Team: {team.name}")
team.list_players()
```

Here's the Output:

```
Team: Oilers
Stuart (Goalie)
Connor (Forward)
```

Helper Methods for Searching and Access

You can add methods to find or retrieve related objects.

```
class Team:
    # ...existing code...
    def get_player(self, name):
        for player in self.players:
            if player.name == name:
                return player
        return None

goalie = team.get_player("Stuart")
if goalie:
    print("Found:", goalie)
```

Here's the Output:

```
Found: Stuart (Goalie)
```

Aggregating Data Across Objects

Classes can help you summarize or analyze data from related objects.

```
class Player:
    def __init__(self, name, points):
        self.name = name
        self.points = points

class Team:
    # ...existing code...
    def total_points(self):
        return sum(player.points for player in self.players)

team = Team("Wolves")
team.add_player(Player("Maya", 12))
team.add_player(Player("Leo", 8))
print("Total team points:", team.total_points())
```

Here's the Output:

```
Total team points: 20
```

Customizing Object Comparison

In Python, you can customize how objects are compared using special methods like `__eq__`. This allows you to define what it means for two objects to be considered equal.

```
class Player:
    def __init__(self, name, points):
        self.name = name
        self.points = points

    def __eq__(self, other):
        return self.name == other.name

p1 = Player("Stuart", 10)
p2 = Player("Stuart", 15)
print(p1 == p2)  # True, because names match
```

Note there are other special methods like `__lt__`, `__gt__`, etc., that you can use to define less than, greater than, etc. Here's the link to docs for other special methods: Python Data Model. You can override these methods to customize how your objects behave in comparisons.

Summary

- Classes can model relationships and groupings, not just single objects.
- Use methods to add, find, and summarize related objects.
- Special methods like `__eq__` let you control how objects are compared.
- This approach helps you build more realistic and powerful programs.

Example

Let’s go run a few examples together