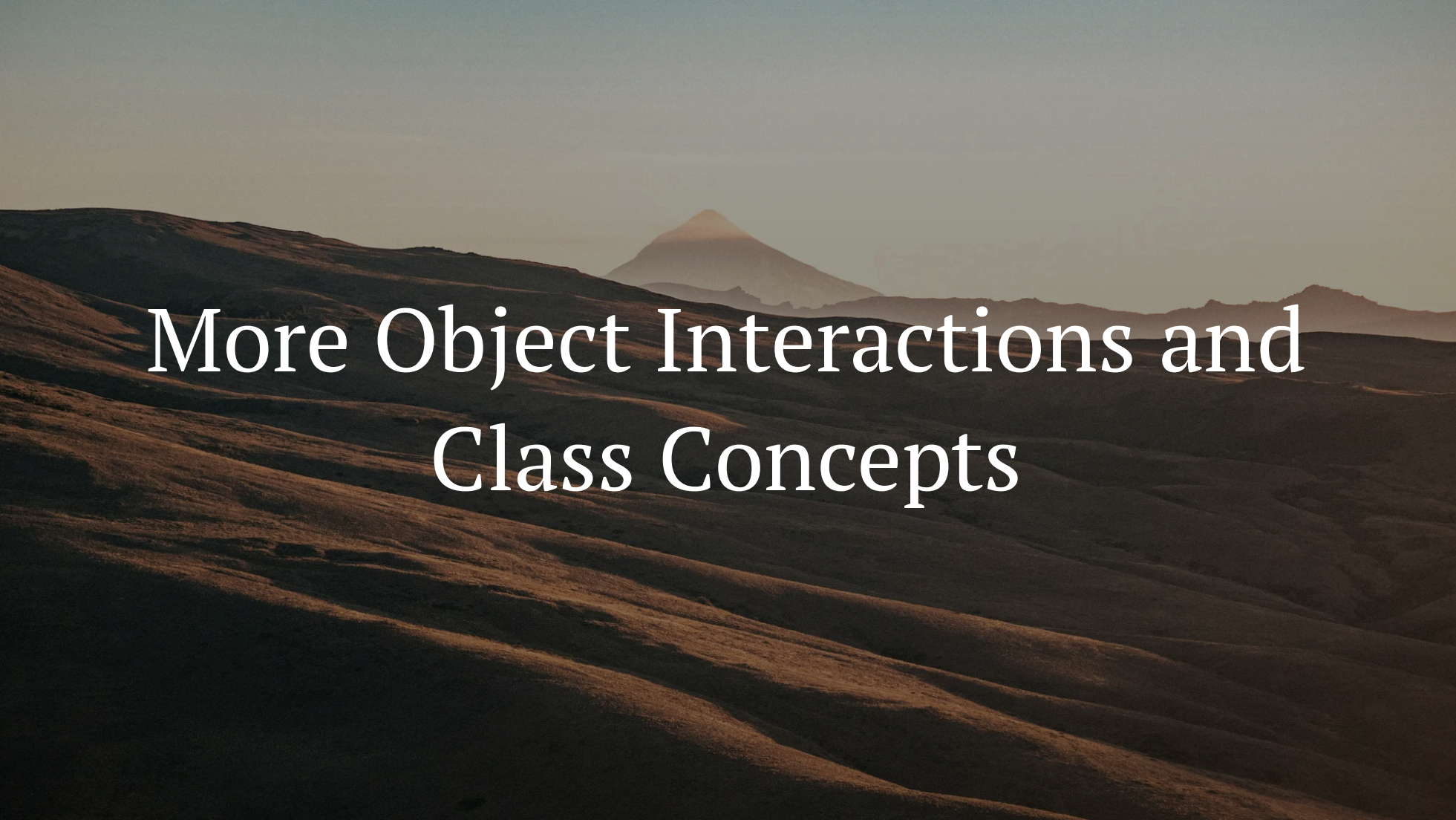

More Object Interactions and Class Concepts

Quick Recap: Functions as Abstraction

- Functions package a set of instructions that perform a specific task
- They help us organize code and avoid repetition
- Example:

```
def calculate_area(length, width):
    return length * width

# Using the function
room_area = calculate_area(10, 15)
print(f"The room area is {room_area} square units")
```

Classes: The Next Level of Abstraction

- Classes bundle related data (attributes) and functions (methods) together
- They represent real-world concepts or entities in code
- Classes are like creating your own custom data type

```
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def calculate_area(self):
        return self.length * self.width

# Using the class
room = Rectangle(10, 15)
print(f"The room area is {room.calculate_area()} square units")
```

Understanding `self` in Detail

- `self` refers to the instance of the class being worked with
- It's how the method knows which object's data to use
- `self` is passed automatically when you call a method on an object

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

    def bark(self):
        print(f"{self.name} says Woof!")

buddy = Dog("Buddy")
buddy.bark()  # Python automatically passes 'buddy' as 'self'
```

The `__init__` Method Explained

- `__init__` is called when creating a new object
- It initializes the object's attributes
- You can think of it as the "setup" method for each new object

```
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
        self.mileage = 0  # Default value

    def drive(self, distance):
        self.mileage += distance

my_car = Car("Toyota", "Corolla", 2022)
print(f"Initial mileage: {my_car.mileage}")
my_car.drive(100)
print(f"After driving: {my_car.mileage}")
```

Scope of Parameters in Classes and Methods

- Parameters in `__init__` are used to initialize object attributes
- Method parameters are used within the method's scope
- Class attributes are shared by all instances of the class

```
class BankAccount:
    interest_rate = 0.02  # Class attribute

    def __init__(self, account_number, balance):
        self.account_number = account_number  # Instance attribute
        self.balance = balance                # Instance attribute

    def apply_interest(self):
        self.balance += self.balance * BankAccount.interest_rate

account = BankAccount("12345", 1000)
account.apply_interest()
print(f"New balance: ${account.balance:.2f}")
```

Objects Interacting with Each Other

- Objects can be attributes of other objects
- Methods can take objects as parameters
- This allows for complex interactions between objects

```
class Person:
    def __init__(self, name):
        self.name = name
        self.pet = None

    def adopt_pet(self, pet):
        self.pet = pet
        print(f"{self.name} adopted {pet.name}")

class Pet:
    def __init__(self, name, species):
        self.name = name
        self.species = species

john = Person("John")
fluffy = Pet("Fluffy", "cat")
john.adopt_pet(fluffy)
print(f"{john.name}'s pet is a {john.pet.species} named {john.pet.name}")
```

Group Assignment Option 1: Create a Simple Game System (🌶️🌶️🌶️)

Create classes for a simple text-based adventure game:

1. Create a `Player` class with attributes like name, health, and inventory
2. Create an `Item` class for objects that players can pick up
3. Create a `Room` class to represent locations in the game
4. Implement methods for the player to move between rooms and pick up items

Work in groups to design and implement these classes, then demonstrate how they interact.

Group Assignment Option 2: Design a School Management System (🌶️)

Create classes to manage a school:

1. Create a `Student` class with attributes like name, grade, and a list of courses
2. Create a `Teacher` class with attributes like name, subject, and a list of classes they teach
3. Create a `Course` class with attributes like name, subject, and a list of enrolled students
4. Implement methods for enrolling students in courses and assigning teachers to courses

Work in groups to design these classes and demonstrate how they can be used to manage a school system.

Key Takeaways

1. Classes are a higher level of abstraction, bundling data and functions
2. `self` refers to the instance and is crucial for accessing object attributes
3. `__init__` sets up each new object with its initial state
4. Parameters in methods have a specific scope within that method
5. Objects can interact by being attributes of other objects or method parameters
6. Designing systems with multiple interacting classes helps model complex real-world scenarios

Practice and Experimentation

- Try expanding on the group assignments
- Create your own systems using multiple interacting classes
- Don’t hesitate to ask questions and discuss your designs with others

Introduction on Dictionaries

- A dictionary is a collection of key-value pairs
- It’s like a real-world dictionary: word (key) -> definition (value)
- In Python, we use curly braces `{}` to define dictionaries
- Keys must be unique and immutable (like strings or numbers)

Creating a Dictionary

```
# Empty dictionary
empty_dict = {}

# Dictionary with initial key-value pairs
person = {
    "name": "Alice",
    "age": 30,
    "city": "New York"
}
```

Accessing Values in a Dictionary

- Use square brackets `[]` with the key to access values
- Use the `get()` method for safer access (returns None if key doesn't exist)

```
person = {"name": "Bob", "age": 25}

print(person["name"])  # Output: Bob
print(person.get("age"))  # Output: 25
print(person.get("city", "Unknown"))  # Output: Unknown
```

Adding or Modifying Key-Value Pairs

```
person = {"name": "Charlie"}

# Adding a new key-value pair
person["age"] = 35

# Modifying an existing value
person["name"] = "Charles"

print(person)  # Output: {'name': 'Charles', 'age': 35}
```

Removing Key-Value Pairs

```
person = {"name": "David", "age": 40, "city": "London"}

# Remove a specific item
del person["age"]

# Remove and return an item
city = person.pop("city")

print(person)  # Output: {'name': 'David'}
print(city)    # Output: London
```