CS1010E NOTES - Brians Tjipto

OOP

What is OOP?

Object-Oriented Programming (OOP) is a **programming paradigm** that uses **objects and classes for designing and organizing code**. In OOP, the fundamental building blocks are objects, which are instances of classes. A class is a blueprint that defines the structure and behavior of objects. OOP promotes modularity, reusability, and a more intuitive way of thinking about programming.

Basic Syntax

Constructor, Attributes, Abstraction, and Methods

Constructor:

In Python, the constructor is a special method called `__init__` that is automatically called when an object is created. It initializes the object's attributes.

```python
class Student:
    def __init__(self, name, age, grade):
        self.name = name
        self.age = age
        self.grade = grade
```

Attributes:

Attributes are variables that store data within a class. They represent the characteristics or properties of an object.

```python
class Student:
    def __init__(self, name, age, grade):
        self.name = name
        self.age = age
        self.grade = grade

# Example of creating an instance of the Student class
student1 = Student("Fkface", 20, "A")

# Accessing attributes
print(student1.name)
print(student1.age)
print(student1.grade)
```

Abstraction:

Abstraction is the concept of hiding the complex implementation details and exposing only the necessary features of an object.

```python
class Student:
    def __init__(self, name, age, grade):
        self.name = name
        self.age = age
        self.grade = grade

    def display_info(self):
        print(f"Name: {self.name}, Age: {self.age}, Grade: {self.grade}")
```

Methods:

Methods are functions defined within a class. They represent the behavior or actions that objects of the class can perform.

```python
class Student:
    def __init__(self, name, age, grade):
        self.name = name
        self.age = age
        self.grade = grade

    def display_info(self):
        print(f"Name: {self.name}, Age: {self.age}, Grade: {self.grade}")

student1 = Student("Fkface", 20, "A")

# Using the display_info method
student1.display_info()
```

Inheritance

Inheritance allows a class (subclass or derived class) to inherit the properties and methods of another class (base class or parent class). It promotes code reuse and the creation of a hierarchy of classes.

```python
class HighSchoolStudent(Student):
    def __init__(self, name, age, grade, level):
        super().__init__(name, age, grade)
        self.level = level

    def display_info(self):
        print(f"Name: {self.name}, Age: {self.age}, Grade: {self.grade}, Level: {self.level}")

# Example of creating an instance of HighSchoolStudent
high_school_student = HighSchoolStudent("Shtface", 18, "B", 12)
high_school_student.display_info()
```

`super()` in Inheritance

In Python, `super()` is a built-in function that is used to refer to the superclass (parent class) of a derived class. It is commonly used in the `__init__` method of a subclass to invoke the constructor of the superclass.

```python
class Student:
    def __init__(self, name, age, grade):
        self.name = name
        self.age = age
        self.grade = grade

    def display_info(self):
        print(f"Name: {self.name}, Age: {self.age}, Grade: {self.grade}")

class HighSchoolStudent(Student):
    def __init__(self, name, age, grade, level):
        super().__init__(name, age, grade)
        self.level = level

    def display_info(self):
        print(f"Name: {self.name}, Age: {self.age}, Grade: {self.grade}, Level: {self.level}")

high_school_student = HighSchoolStudent("Shtface", 18, "B", 12)
high_school_student.display_info()
```

`isinstance()` in Inheritance

In Python, the `isinstance()` function is utilized to check if an object belongs to a specified class or a tuple of classes. This method is particularly useful in scenarios involving inheritance, where you want to determine if an instance is derived from a specific class or its subclasses.

Consider the following scenario with a base class `Shape` and two subclasses `Circle` and `Square`:

```python
class Shape:
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius ** 2

class Square(Shape):
    def __init__(self, side_length):
        self.side_length = side_length

    def area(self):
        return self.side_length ** 2

circle_instance = Circle(5)
square_instance = Square(4)

print(isinstance(circle_instance, Shape))  # Output: True
print(isinstance(square_instance, Circle))  # Output: False
print(isinstance(square_instance, (Shape, Square)))  # Output: True
```

In this example, `isinstance()` is employed to check the type of objects, providing flexibility in handling instances of the base class and its subclasses.

This method is valuable for conditional statements and ensuring compatibility when dealing with objects derived from a common base class.

Polymorphism

Polymorphism allows objects to be treated as instances of their parent class, enabling flexibility and code simplification.

```python
class Student:
    def __init__(self, name, age, grade):
        self.name = name
        self.age = age
        self.grade = grade

    def display_info(self):
        print(f"Name: {self.name}, Age: {self.age}, Grade: {self.grade}")

class HighSchoolStudent(Student):
    def __init__(self, name, age, grade, level):
        super().__init__(name, age, grade)
        self.level = level

    def display_info(self):
        print(f"Name: {self.name}, Age: {self.age}, Grade: {self.grade}, Level: {self.level}")

student1 = Student("Fkface", 20, "A")
high_school_student = HighSchoolStudent("Shtface", 18, "B", 12)

# Using polymorphism with the display_info method
def print_student_info(student):
    student.display_info()

# Example of polymorphic behavior
print_student_info(student1)
print_student_info(high_school_student)
```

In the example above, the `print_student_info` function can take objects of different classes (Student and HighSchoolStudent) and call their `display_info` methods, demonstrating polymorphism.

Complex Example

```python
import random
from tabulate import tabulate


class Champion:
    def __init__(self, name, abilities):
        self.name = name
        self.level = 1
        self.cs = 0
        self.kills = 0
        self.deaths = 0
        self.assists = 0
        self.abilities = abilities

    def level_up(self):
        self.level += 1
        print(f"{self.name} leveled up to {self.level}!")

    def gain_cs(self):
        self.cs += 1
        print(f"{self.name} gained 1 CS. Total CS: {self.cs}")

    def kill(self):
        self.kills += 1
        print(f"{self.name} got a kill! Kills: {self.kills}")

    def die(self):
        self.deaths += 1
        print(f"{self.name} died. Deaths: {self.deaths}")

    def assist(self):
        self.assists += 1
        print(f"{self.name} assisted. Assists: {self.assists}")

    def take_damage(self, damage, dodge=False):
        if dodge:
            damage = self.dodge(damage)
        print(f"{self.name} took {damage} damage.")
        return damage

    def use_ability(self, ability_name, target=None):
        if ability_name in self.abilities:
            ability = self.abilities[ability_name]
            damage = ability.use()
            if target:
                print(
                    f"{self.name} used {ability_name} on {target.name} for {damage} damage.")
                target.take_damage(damage)
            else:
                print(f"{self.name} used {ability_name}.")
        else:
            print(f"{self.name} does not have {ability_name}.")

    def use_ultimate(self, target=None):
        self.use_ability('R', target)

    def heal(self, heal_pts):
        print(f"{self.name} healed for {heal_pts} points.")

    def dodge(self, damage):
        return damage // 2


class Ability:
    def __init__(self, name, damage, heal=0, dodge=False):
        self.name = name
        self.damage = damage
        self.heal = heal
        self.dodge = dodge

    def use(self):
        return self.damage, self.heal, self.dodge


class UltimateAbility(Ability):
    def use(self):
        if random.random() < 0.7:
            return super().use()
        else:
            print(f"{self.name} failed!")
            return 0


class Team:
    def __init__(self, name, members):
        self.name = name
        self.members = members


# T1
top_1_abilities = {'Q': Ability('Aatrox Q', 50), 'W': Ability(
    'Aatrox W', 40), 'E': Ability('Aatrox E', 30), 'R': Ability('Aatrox R', 70, heal=20)}
```

```python
top_1 = Champion("Zeus", top_1_abilities)

jg_1_abilities = {'Q': Ability('Leesin Q', 40), 'W': Ability(
    'Leesin W', 30, heal=20), 'E': Ability('Leesin E', 20), 'R': Ability('Leesin R', 60)}
jg_1 = Champion("Oner", jg_1_abilities)

mid_1_abilities = {'Q': Ability('Akali Q', 40), 'W': Ability(
    'Akali W', 0, dodge=True), 'E': Ability('Akali E', 30), 'R': Ability('Akali R', 70)}
mid_1 = Champion("Faker", mid_1_abilities)

bot_1_abilities = {'Q': Ability('Xayah Q', 40), 'W': Ability(
    'Xayah W', 30), 'E': Ability('Xayah E', 30), 'R': Ability('Xayah R', 0, dodge=True)}
bot_1 = Champion("Gumayusi", bot_1_abilities)

sup_1_abilities = {'Q': Ability('Rakan Q', 30), 'W': Ability(
    'Rakan W', 30), 'E': Ability('Rakan E', 0, heal=20), 'R': Ability('Rakan R', 60)}
sup_1 = Champion("Keria", sup_1_abilities)

t1 = Team("T1", [top_1, jg_1, mid_1, bot_1, sup_1])

# WBG
top_2_abilities = {'Q': Ability('Kennen Q', 40), 'W': Ability(
    'Kennen W', 30), 'E': Ability('Kennen E', 30), 'R': Ability('Kennen R', 70)}
top_2 = Champion("TheShy", top_2_abilities)

jg_2_abilities = {'Q': Ability('Jarvan IV Q', 40), 'W': Ability(
    'Jarvan IV W', 0, heal=20), 'E': Ability('Jarvan IV E', 30), 'R': Ability('Jarvan IV R', 60)}
jg_2 = Champion("Weiwei", jg_2_abilities)

mid_2_abilities = {'Q': Ability('Azir Q', 40), 'W': Ability(
    'Azir W', 30), 'E': Ability('Azir E', 0, heal=20), 'R': Ability('Azir R', 70)}
mid_2 = Champion("XiaoHu", mid_2_abilities)

bot_2_abilities = {'Q': Ability('Varus Q', 50), 'W': Ability(
    'Varus W', 40), 'E': Ability('Varus E', 30), 'R': Ability('Varus R', 70)}
bot_2 = Champion("Light", bot_2_abilities)

sup_2_abilities = {'Q': Ability('Bard Q', 30), 'W': Ability('Bard W', 0, heal=20), 'E': Ability(
    'Bard E', 0, dodge=True), 'R': Ability('Bard R', 0, dodge=True)}
sup_2 = Champion("Crisp", sup_2_abilities)

weibo_gaming = Team("Weibo Gaming", [top_2, jg_2, mid_2, bot_2, sup_2])


def take_turn(attacker, defender, kill_chance=0.05):
    ability_to_use = random.choice(list(attacker.abilities.keys()))
    ability = attacker.abilities[ability_to_use]
    damage, heal, dodge = ability.use()

    if heal > 0:
        attacker.heal(heal)
    if 'dodge' in ability.name:
        defender.take_damage(damage, dodge=True)
    else:
        defender.take_damage(damage)

    if damage > 0:
        attacker.gain_cs()
        if attacker.cs % 5 == 0:
            attacker.level_up()

    if heal > 0:
        attacker.heal(heal)
    if damage > 0 and random.random() < kill_chance and defender.take_damage(damage, dodge=True) > 0:
        attacker.kill()
        defender.die()
        attacker.assist()


def simulate_game(team1, team2):
    for _ in range(1000):
        attacker_team = random.choice([team1, team2])
        defender_team = team2 if attacker_team == team1 else team1
        attacker = random.choice(attacker_team.members)
        defender = random.choice(defender_team.members)
        take_turn(attacker, defender)
    stats = []
    for team in [team1, team2]:
        for member in team.members:
            stats.append([team.name, member.name, member.level,
                          member.cs, member.kills, member.deaths, member.assists])
    print(tabulate(stats, headers=["Team", "Name",
                                   "Level", "CS", "Kills", "Deaths", "Assists"]))


simulate_game(t1, weibo_gaming)
```