05 Python - Classes and Objects

Introduction

```
class IPLPlayer:

def __init__(self, name, team):
        self.name = name
        self.team = team
        self.runs = 0
        self.wickets = 0

    def enterRuns(self):
        self.runs = int(input('Enter runs scored by %s: ' % self.name))

    def enterWickets(self):
        self.wickets = int(input('Enter wickets taken by %s: ' % self.name))

    def displayStats(self):
        print('%s from %s scored %d runs and took %d wickets.' % (self.name, self.team, self.runs, self.wickets))

name = input('Enter name of IPL player: ')
team = input('Enter team name: ')

p1 = IPLPlayer(name, team)
p1.enterRuns()
p1.enterWickets()
p1.displayStats()
```

1. The code defines a Python class named `IPLPlayer`.
2. The class has four methods: `__init__(self, name, team)`, `enterRuns(self)`, `enterWickets(self)` and `displayStats(self)`.
3. The constructor method initializes the instance variables `name`, `team`, `runs` and `wickets`.
4. The `enterRuns(self)` method takes input from the user for the number of runs scored by the player.
5. The `enterWickets(self)` method takes input from the user for the number of wickets taken by the player.
6. The `displayStats(self)` method displays the name of the player, their team name, runs scored and wickets taken.

```
class Point:

def __init__(self):

self.x = 0.0

self.y = 0.0



class Rectangle:

def __init__(self):

self.width = 0.0

self.height = 0.0

self.corner = Point()



box = Rectangle()

box.width = 100.0

box.height = 200.0

box.corner = Point()

box.corner.x = 0.0
box.corner.y = 0.0



print("Box width:", box.width)

print("Box height:", box.height)

print("Box corner coordinates:", box.corner.x, box.corner.y)
```

1. The given code defines two classes: "Point" and "Rectangle". The Point class represents a point in 2D space with an x and y coordinate. The Rectangle class represents a rectangle on a plane, defined by its width, height, and the coordinates of its bottom-left corner.

2. The Point class has an **init** method that initializes the x and y attributes to `0.0`.

3. The Rectangle class has an **init** method that initializes the width and height attributes to `0.0`, and the corner attribute to an instance of the Point class with x and y coordinates set to `0.0`.

4. The code instantiates a Rectangle object named "box" and assigns the values of `100.0` and `200.0` to its width and height attributes, respectively. It also creates a Point object to represent the corner of the rectangle, with its x and y coordinates set to `0.0`. These values are then assigned to the corresponding attributes of the "box" object. Finally, the code prints out the width, height, and corner coordinates of the "box" object.

In the rectangle part, the corner attribute is a Point object that represents the bottom-left corner of the rectangle. By instantiating a new Point object and assigning it to the corner attribute of the Rectangle object, we can specify the coordinates of the corner. This allows us to define the position of the rectangle on a plane.

Time

```
from datetime import datetime

birthday = input('Enter your birthday (YYYY-MM-DD): ')
birthday = datetime.strptime(birthday, '%Y-%m-%d')
today = datetime.now()
age = today.year - birthday.year

if today.month < birthday.month or (today.month == birthday.month and today.day < birthday.day):
    age -= 1

print(f'You are {age} years old.')
```

1. The `datetime` module provides classes like `datetime`, `date`, `time`, `timedelta`, etc. for working with dates and times.
2. The `datetime` class provides several methods to get the current date and time, and to create datetime objects with specific dates and times.
3. The `strptime()` method in the `datetime` class is used to parse a string representing a date and time into a `datetime` object.
4. In the given code, the `datetime` module is used to calculate the age of a person based on their birthday and the current date. The input provided by the user is converted into a `datetime` object using the `strptime()` method, and then the difference between the current year and the birth year is calculated. Finally, the calculated age is printed to the console.
5.

Pure function

```
def add(a, b):
        return a + b

result = add (2, 3)
print (result) # Output: 5
```

- A pure function is a type of function in computer programming that, given the same input, will always return the same output and has no side effects.
- This function is pure because it does not have any side effects and its output only depends on its input. It always returns the same output for the same input, and it does not change any values outside of its own scope.
- In this example, we define a function named add that takes two arguments: a and b . The function returns the sum of a and b .
- The add function is a pure function because it always returns the same result for the same input (e.g., calling add(2, 3) always returns 5) and it does not have any observable side effects (e.g., it does not modify any global variables or print anything to the screen).

- We call the add function with the arguments 2 and 3, and store the result in a variable named result . Finally, we print the value of the result variable, which is 5 . Is there anything specific you would like to know or discuss?

Modifier

```
def add_to_list(my_list, value):
    my_list.append(value)

my_list = [1, 2, 3]
add_to_list(my_list, 4)
print(my_list) # Output: [1, 2, 3, 4]
```

1. Modifiers are functions that modify the objects they receive as parameters.
2. The modifications made by a modifier are visible to the caller of the function.
3. Modifiers can be useful when you want to update the state of an object without creating a new object.
4. `add_to_list()` function modifies the original `lst` object by appending `value` to it. Thus, it alters the state of the original object.
5. The function doesn't create a new object and return it. It directly modifies the original object passed to it as a parameter.
6. The changes made to the original object are visible to the caller even after the function returns.
7. Any code that has a reference to the `lst` object can see the changes made by the `add_to_list()` function.

Prototype and patch

```
def increment(time, seconds):
    time.second += seconds
    if time.second >= 60:
        time.second -= 60
        time.minute += 1
    if time.minute >= 60:
        time.minute -= 60
        time.hour += 1

time = Time()
time.hour = 1
time.minute = 59
time.second = 59
increment(time, 1)
print_time(time)
```

The "prototype and patch" development approach is a strategy where you create a rough prototype of a function to perform a basic calculation and then test it, fixing any errors you find along the way. The basic idea is to develop the program incrementally by testing and patching it, starting with a simple version and gradually improving it

In this example, the `increment` function is first prototyped with basic functionality that only handles the addition of seconds. The function is then tested with a `Time` object initialized to 1 hour, 59 minutes, and 59 seconds, and a value of 1 passed to the `increment` function. The output should be 2:00:00.

If errors are encountered during testing, they are patched incrementally until the function works as intended. In this case, the special cases where the minute and hour attributes need to be incremented

init method and str method

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

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

person = Person("John", 30)
print(person)
```

- The `__str__` method is a special method in Python classes that provides a string representation of an object. The `__init__` method is a special method in Python classes that is called when an object is created.
- The `__init__` method is defined with two arguments: `name` and `age`.

2. In the `__init__` method, the `name` and `age` arguments are assigned to instance variables with the same names using `self.name = name` and `self.age = age`.

3. The `__str__` method is defined to return a string representation of the `Person` object using f-strings. The string includes the person's name and age enclosed in parentheses.
4. An instance of the `Person` class is created and assigned to the `person` variable with name "John" and age 30.
5. The `print` statement calls the `__str__` method implicitly and outputs the string representation of the `person` object, which is "John (30)".

Type dispatch

```
def add(a, b):
    if isinstance(a, int) and isinstance(b, int):
        return a * b
    elif isinstance(a, str) and isinstance(b, str):
        return a + ' ' + b
    else:
        return None
```

1. This program defines a function `add` that takes two parameters, `a` and `b`.
2. The program uses the `isinstance` function to check the types of the parameters.
3. If both `a` and `b` are integers, the function returns the product of the two integers.
4. If both `a` and `b` are strings, the function returns the concatenation of the two strings with a space in between.
5. If `a` and `b` are not of the same type, the function returns `None`.

The code uses type dispatch to handle different types of inputs differently. It uses the `isinstance` function to check the types of the parameters and then executes different code blocks based on the type of the input.
This approach is useful when we want to define a function that works differently based on the type of its input.

Polymorphism

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

class Rectangle(Shape):
    def __init__(self, length, width):
        self.length = length
        self.width = width
    def area(self):
        return self.length * self.width

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        return 3.14 * self.radius ** 2

def get_area(shape):
    return shape.area()

# creating objects of Rectangle and Circle
rectangle = Rectangle(5, 10)
circle = Circle(7)

# calling the get_area function with different objects
print(get_area(rectangle)) # Output: 50
print(get_area(circle)) # Output: 153.86
```

1. Polymorphism allows for code reuse and flexibility by allowing functions to work with multiple types of data.

2. It simplifies the code by allowing the same method to be used for different types of objects.

3. Polymorphism is a core concept of object-oriented programming.

4. The code defines a parent class `Shape` with a method `area()` that doesn't do anything, but it provides a template for other classes that will inherit from it.

5. The code defines two child classes `Rectangle` and `Circle`, which inherit from `Shape` and provide their own implementations of the `area()` method.

6. The `get_area()` function takes an argument `shape`, which is an instance of either the `Rectangle` or `Circle` class. It calls the `area()` method on this instance, but the specific implementation of `area()` that gets called depends on the type of the instance passed in.

7. Because both `Rectangle` and `Circle` have an `area()` method, and they inherit from the same parent class `Shape`, the `get_area()` function can be applied to objects of either class. This is an example of polymorphism, where different objects can be treated as though they are the same type (in this case, `Shape` objects) and can respond to the same method call (`area()`).

Operator Overloading

```python
class Adder:
    def __init__(self, num):
        self.num = num

    def __add__(self, other):
        return self.num + other

a1 = Adder(5)
print(a1 + 10) # Output: 15
```

1. In Python, operator overloading is achieved by defining special methods in a class that are associated with the operator.
2. The special method for operator overloading is indicated by using double underscores before and after the method name. For example, the special method for overloading the "+" operator is "**add**()".
3. Operator overloading allows you to use operators on user-defined types in a way that is similar to how they are used with built-in types.
4. This can make code more readable and expressive, as you can use operators in a way that is natural and intuitive.
5. In the given code, the Adder class overloads the "+" operator using the "**add**()" method to allow adding an instance of the Adder class with an integer. This allows for more natural and intuitive code, and makes the Adder class behave more like a built-in type.