@November 21, 2024

Object:

- A thing in the world
- Everything in the world is an object
- Physical things, ideas, emotions etc

Object oriented programming

- Python is an object oriented programming language
- Everything in Python is an object

```
def func():
        print("hello")


print(func)

<function func at 0x101c81120>

type("ahmed")
<class 'str'>
type(1)
<class 'int'>
type(1.0)
<class 'float'>
type([])
<class 'list'>
```

- Objects in the real worlds:
  - Books
  - Table

    - Chair
    - White board
    - Computer
    - Mug/cup
    - Water bottle
    - Mirror
    - tablet
    - folder
    - Sofa
    - Software
    - Code
    - Terminal
    - Window subsystem for linux
- So you can model real world objects
- Book
    - Properties
        - Author
        - Name
        - Serial number
        - If its special edition
    - Object of a book in python to store this information
- Table
    - Properties
        - Material (wood, plastic, metal)
        - Number of legs
        - Glass top

    - Decorations
- Properties of objects define an object
- Python all objects also have properties
- You create objects using classes
- Classes are blueprints of an object
- Classes are used to create objects
- Properties are stored as variables inside your objects

Methods:

- Simply functions within classes
- They are used to make the object do things
- __init__ is used to initialise the variables of the class when object is created
- They are used to interact with and make your objects do things

```
class Chocolate:
    def __init__(self, shape, flavour, brand): # constructor fu
        self.shape = shape
        self.flavour = flavour
        self.brand = brand
        def make_milkshake(self): #This is a method
        if self.brand == "Mars":
            print("yes you can make a shake")
        elif self.brand == "Kit Kat":
            print("cant make a shake")
        elif self.brand == "Random choco brand":
            print("yes you can make a shake")

class Books:
    def __init__(self, name, author, serial_number): #This is a
        self.name = name #This is a property
```

```
        self.author = author
        self.serial_number = serial_number


#lets make a mars bar

mars = Chocolate("Rectangle", "Caramel", "Mars")
kit_kat = Chocolate("Rectangle", "Milk", "Kit Kat")
random_chocolate = Chocolate("Triangle", "White", "Radom choco b
```

- To access a property you need to add a “.”after the object name

```
mars.flavour #flavour of mars
kit_kat.shape #shape of kit kat
random_chocolate.brand #brand of random chocolate
```

- Methods do things on objects

Project 6

- Create a class for character and store
  - Their power
  - Their name
  - Their costume

```
#basic video game interactions
class Character:
  def __init__(self, name, power, costume, damage):
    self.name = name
    self.power = power
    self.costume = costume
    self.health = 100
    self.damage = damage
  def attack(self):
```

```
        print(f"{self.name} is attacking by {self.power}")
        return self.damage
  def get_damage(self, damage_amount):
      print(f"{self.name} attacked by amount of {damage_amount}"
      self.health = self.health - damage_amount
      print(f"{self.name}'s new health: {self.health}")

hero = Character("Hero", "agility", "blue", 10)
villan = Character("Dangerous person", "Fireball", "Red", 20)

#villan does damage to hero
damage_done = villan.attack()
hero.get_damage(damage_done)

damage_done = hero.attack()
villan.get_damage(damage_done)
```

Examples for 5th Dec:

EXAMPLE 1:

Problem:

1. Create a class `Point` to represent a point in a 2D plane with `x` and `y` coordinates.
2. Overload the `+` operator so that adding two points produces a new point with the summed coordinates.

Task:

1. Create two `Point` objects.
2. Add them using the `+` operator and print the resulting point.

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

    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)

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

# Example
p1 = Point(1, 2)
p2 = Point(3, 4)
p3 = p1 + p2
print(p3)  # Output: Point(4, 6)
```

EXAMPLE 2:

Problem:

1. Create a class `Engine` with an attribute `horsepower`.
2. Create a class `Car` that uses an `Engine` object as one of its attributes. Add a method `car_details()` to display the car's brand and engine horsepower.

Task:

1. Create an `Engine` object with a specific horsepower.
2. Use it to create a `Car` object and call the `car_details()` method.

```
class Engine:
    def __init__(self, horsepower):
```

```
        self.horsepower = horsepower

class Car:
    def __init__(self, brand, engine):
        self.brand = brand
        self.engine = engine  # Using Engine object as a compone

    def car_details(self):
        return f"The {self.brand} car has an engine with {self.e

# Example
engine = Engine(300)
car = Car("Ford", engine)
print(car.car_details())  # Output: The Ford car has an engine w
```

EXAMPLE 3:

Problem:

1. Create a parent class `Vehicle` with attributes `brand` and `color`. Add a method `description` to return the details of the vehicle.
2. Create a child class `Car` that inherits from `Vehicle` and has an additional attribute `model`. Override the `description` method to include the `model`.

Task:

1. Instantiate an object of the `Car` class.
2. Print the overridden `description`.

```
class Vehicle:
    def __init__(self, brand, color):
        self.brand = brand
        self.color = color

    def description(self):
        return f"This is a {self.color} {self.brand} vehicle."
```

```
# Child class
class Car(Vehicle):
    def __init__(self, brand, color, model):
        super().__init__(brand, color)  # Calling the parent cla
        self.model = model

    def description(self):  # Overriding parent method
        return f"This is a {self.color} {self.brand} car, model

# Using inheritance
my_car = Car("Toyota", "Red", "Corolla")
print(my_car.description())  # Output: This is a Red Toyota car,
```

EXAMPLE 4:

Problem:

1. Define an abstract base class `Shape` with an abstract method `area()`.
2. Create two subclasses: `Rectangle` and `Circle`, implementing the `area` method for each.

Task:

1. Create objects of both `Rectangle` and `Circle`.
2. Calculate and print the area of each shape.

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

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

```
    def area(self):
        return self.width * self.height

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

    def area(self):
        return 3.14159 * self.radius ** 2

# Example
shapes = [Rectangle(4, 5), Circle(3)]
for shape in shapes:
    print(shape.area())
```

EXAMPLE 5:

Problem:
Write a class
`BankAccount` to model a simple bank account:

1. The class should have private attributes: `owner` and `__balance`.
2. Include methods to deposit, withdraw, and get the balance.
3. Ensure that invalid withdrawal attempts are handled gracefully.

Task:

1. Create a `BankAccount` object for a customer.
2. Perform a deposit, a valid withdrawal, and an invalid withdrawal.
3. Display the final balance.

```
class BankAccount:
    def __init__(self, owner, balance):
        self.owner = owner
        self.__balance = balance  # Private attribute
```

```
    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
        else:
            print("Insufficient balance or invalid amount!")

    def get_balance(self):
        return self.__balance

# Example
account = BankAccount("Alice", 1000)
account.deposit(500)
account.withdraw(300)
print(account.get_balance())  # Output: 1200
```