Klasa - abstrakcyjny opis obiektu/rzeczywistości, np. samochód, który można opisać za pomocą marki, roku produkcji, parametrów silnika, aktualnej prędkości, etc.
Obiekt - konkretny egzemplarz samochodu (oraz jego stan), np. Kia Sportage 2022 1.6 T-DGI, pruszająca się z prędkością 75km/h na 6 biegu.
Metoda - funkcja powiązana z daną klasą, która może być wywołana na rzecz obiektu danej klasy.
Pole / *Parametr* - dane dotyczące obiektu.

Zad 1. Budowanie klasy.

```
class Car:
    def __init__(self, make):
        self.make = make
        self.motor_run = False
        self.gear = 0
        self.speed = 0

    def motor_start(self):
        self.motor_run = True

    def motor_stop(self):
        self.motor_run = False

    def gear_change_up(self):
        if self.gear <= 7:
            self.gear+=1
        print(self.gear)

    def gear_change_down(self):
        if self.gear >= 0:
            self.gear-=1
        print(self.gear)

my_car = Car("Kia Sportage")
my_car.motor_start()
my_car.gear_change_down()
my_car.gear
```

Dodaj do powyższego kodu metody `speed_up` i `brake`.

self - w Pythone metody przyjmują jako pierwszy parametr obiekt, na rzecz którego są wywoływane. Przy wywoływaniu metody argument *self* należy pominąć.

Zad 2. Hermetyzacja.

Hermetyzacja / Enkapsulacja - ograniczenie dostępu do wybranych składowych klasy. Zwyczajowo:

- *publiczne* / *public* - dostępne dla każdego;
- *chronione* / *protected* - dostęp dla klas dziedziczących (patrz zad.5);
- *prywatne* / *private* - tylko wewnętrz klasy.

```
class Encapsulation:
    def __init__(self):
        self.public, self._protected, self.__private = 1, 2, 3


def main():
    encapsulation = Encapsulation()
    print(encapsulation.public)
    print(encapsulation._protected)
    print(encapsulation._Encapsulation__private)
# a teraz będzie błąd
    print(encapsulation.__private)


main()
```

W przypadku klasy *car*, z zad.1:

```
class Car:
    def __init__(self, make):
        self.make = make
        self.__motor_run = False
        self.__gear = 0
        self.__speed = 0

    def motor_start(self):
        self.__motor_run = True
```

```
    def motor_stop(self):
        self.__motor_run = False

    def gear_change_up(self):
        if self.__gear <= 7:
            self.__gear+=1
        print(self.gear)

    def gear_change_down(self):
        if self.__gear >= 0:
            self.__gear-=1
        print(self.gear)

my_car = Car("Kia")
my_car.motor_start()
my_car.gear_change_down()
my_car.gear
```

Zaimplementuj automatyczną zmianę biegów.

```
def __gear_change_up(self):
     if self.__gear <= 7:
     self.__gear+=1
     print(self.gear)
```

Zad 3. Pola statyczne. Metody statyczne.

Pola statyczne(*static*) są współdzielone przez wszystkie obiekty danej klasy. Przynależą one do klasy, a nie do obiektu. Metody statyczne nie mogą korzystać z pól i metod instancyjnych (niestatycznych) - nie są wywoływane na rzecz danego obiektu.

```
class Car:
    how_many = 0

    def __init__(self):
        Car.how_many += 1
        self.car_number = Car.how_many
        print(f"Numer of cars is equal to {Car.how_many}")

    def __del__(self):
```

```
        Car.how_many -= 1
        print(f"Number of cars is equal to {Car.how_many}")

    @staticmethod
    def count_cars():
        return Car.how_many

car_1 = Car()
car_2 = Car()
car_3 = Car()
print(f"Total number of cars {Car.count_cars()}")
car_2 = None
print(f"Total number of cars {Car.count_cars()}")
```

Zad 4. Stwórz klasę *Q_function*. Klasa powinna pozwalać na zdefiniowane funkcji kwadratowej (a*x^2 + b*x + c) i zawierać medodę *Solve()* zwracająca miejsca zerowe.

Zad 5. Dziedziczenie.

Mechanizm pozwalający na przejęcie pewnych cech (pola i metody) z klasy bazowej przez klasę pochodną/potomną. Klasa pochodna może dodać własne składowe, a także może zmieniać działanie metod klasy bazowej.

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

    def hasName(self):
        print("Has name")

class Student(Person):
    def __init__(self,name,surname,age,field_of_study):
        super().__init__(name,surname,age)
        self.field_of_study = field_of_study

    def isStudent(self):
        print("Is student")

person_1 = Person("Tom","Nowak",25)
```

```
student_1 = Student("Tom","Nowak",25,"Informatics")

student_1.hasName()
```

Zad 6. Przygotowanie środowiska pracy.
Sugerowanym środowiskiem pracy jest PyCharm 2024.2.3 oraz Python 3.12. Na zajęciach można jednak używać dowolnego środowiska pozwalającego na uruchamianie skryptów Python - Jupyter Notebook, Thonny IDE, Microsoft Visual Studio Code, etc. Potrzebne moduły: *opencv-python*, *mediapipe*, *numpy*. W środowisku PyCharm można je dodać korzystając z:

File -> Settings -> Project: python -> Python interpreter

Po dodaniu modułów należy podłączyć kamerę do portu USB komputera oraz uruchomić skrypt - `zad6_camera_test.py`

Zad 7. Uruchomić w wybranym IDE skrypt `zad7_hand_detector.py`.

Zad 8. Zbudować klasę *HandTracking* - `zad8_hand_detector_class.py`.

Zad 9. Wykorzystać klasę z zadania 8 do zliczania ilości palców - `zad9_finger_count.py`.

Zad 10. (1 punkt) Rozbudować klasę z zadania 8 o metodę `finUp` zwracającą ilość wyprostowanych palców.