

הורשה מרובה בפייתון

נתחיל מדוגמה שבה הגיוני לרשת משתי מחלקות שונות: נגדיר מחלקה של מכונית שמורשה למחלקה של מכוניות על בנזין ושל מחלקת מכוניות חשמליות.

```
class Car:
    def __init__(self,wheels=4):
        self.wheels = wheels
        # all cars, have four wheels by default.

class Gasoline(Car):
    def __init__(self,engine='Gasoline',tank_cap=20):
        Car.__init__(self)
        self.engine = engine
        self.tank_cap = tank_cap # fuel tank capacity
        self.tank = 0
```

```
    def refuel(self):
        self.tank = self.tank_cap
```

```
class Electric(Car):
    def __init__(self,engine='Electric',kWh_cap=60):
        Car.__init__(self)
        self.engine = engine
        self.kWh_cap = kWh_cap # battery capacity in
                                # kilowatt-hours
        self.kWh = 0
```

```
    def recharge(self):
        self.kWh = self.kWh_cap
```

כעת נרצה מחלקה של מכוניות היברידיות וככזו, היא יכולה לרשת ממחלקת מכונית הבנזין וממחלקת המכונית החשמלית.

```
class Hybrid(Gasoline, Electric):
    def __init__(self,engine='Hybrid',tank_cap=11,
                 kWh_cap=5):
        Gasoline.__init__(self, engine,tank_cap)
        Electric.__init__(self, engine,kWh_cap)
```

```
prius = Hybrid()  
print(prius.tank)  
print(prius.kWh)
```

```
0  
0
```

וכעת אם ניכתוב:

```
prius.recharge()
```

פייתון תבדוק אם המתודה recharge נימצאת במחלקה Hybrid שהיא המחלקה של העצם prius. היא לא תמצא אותה ולכן תמשיך לחפש קודם ב Gasline (כי זו הראשונה משמאל בהורשה), לא תמצא, תעבור ימינה ל Electric ושם תמצא אותה ותפעיל אותה.

סדר החיפוש של המתודות ניקרא: MRO (Method Resolution Order) והוא משמאל לימין ואחר כך למעלה רמה אחת. כלומר למשל אם היא לא היה מוצאת במקרה שלנו את המתודה ב Electric היא הייתה עולה רמה ומחפשת במחלקת הבסיס של Gasline, שהיא: Car (במקרה גם הבסיס של Electric הוא Car, אבל אם הוא היה אחר, היא הייתה ניגשת לשם בשלב החיפוש הבא).

דוגמה נוספת לסדר מציאת המתודות בפייתון:

```
class A:  
    num = 4  
  
class B(A):  
    pass  
  
class C(A):  
    num = 5  
  
class D(B,C):  
    pass
```

num הוא משתנה ברמת המחלקה. מה יקרה אם ניכתוב:

D.num

במחלקה D אין את num, ולכן פייתון תחפש במעלה ההירארכיה. ב B אין את num ולכן פייתון תנסה את C. שם כן יש את num

בפייתון קיימת פונקציה שנקראת super() פונקציה זו עוקבת אחרי ה MRO ויכולה ליתר את הפעלת הפעולה הבונה של המחלקה ממנה ירשנו בתוך המחלקה היורשת. לדוגמה:

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

class MyDerivedClass(MyBaseClass):
    def __init__(self,x,y,z):
        super().__init__(x,y)
        self.z = z
```

שימו לב שלא העברנו פרמטרים ל super אפילו לא self

דוגמה ל super עם ירושה מרובה:

```
class A:
    def truth(self):
        return 'All numbers are even'

class B(A):
    pass

class C(A):
    def truth(self):
        return 'Some numbers are even'

class D(B,C):
    def truth(self,num):
        if num%2 == 0:
            return A.truth(self)
        else:
            return super().truth()
```

```
d = D()
d.truth(6)
```

All numbers are even

```
d.truth(5)
```

```
'Some numbers are even'
```

פולימורפיזם

כאשר יש לנו עצמים שנוצרו בעזרת מחלקות שונות, אבל יש להם מתודות בעלות אותו שם, ניתן למשל להתייחס אליהם בצורה דומה, וכאשר מפעילים את המתודה בעלת אותו שם, מקבלים התנהגות שונה.

נראה דוגמה

```
class Animal:
    def __init__(self, name):      # Constructor
        self.name = name

    def speak(self):
        # Abstract method, defined by convention only
        raise NotImplementedError("Subclass must implement
        abstract method")
```

```
class Dog(Animal):  
    def speak(self):  
        return self.name+' says Woof!'  
  
class Cat(Animal):  
    def speak(self):  
        return self.name+' says Meow!'  
  
def main():  
    fido = Dog('Fido')  
    isis = Cat('Isis')  
  
    for obj in [fido, isis]:  
        print (obj.speak())  
main()
```

ראינו כאן גם מצב שמחלקה לא בהכרח צריכה פעולה בונה. המאפיין `self.name` הוא קיים ב `Dog` וב- `Cat` מתוקף הירושה.