

הורשה

בפייתון, כל מחלקה יורשת באופן אוטומטי ממחלקה שניקראת: object

כדי לקבל אינפורמציה על ההורשה, ניתן להדפיס את ה- help עבור המחלקה שיצרנו. למשל, ניצור מחלקה ריקה, ללא הורשה ונדפיס את המידע.

```
class Employee:
    pass

def main():
    e1 = Employee()
    print(help(Employee))

if __name__ == '__main__':
    main()
```

----- זה מה שיודפס:

```
class Employee(builtins.object)
| Data descriptors defined here:
|
| __dict__
|     dictionary for instance variables (if defined)
|
| __weakref__
|     list of weak references to the object (if defined)
```

None

כלומר, יכולנו באותה מידה גם לכתוב למשל:

```
class Employee(object):
    pass
```

הצורה הקלאסית של הורשה נובעת מרצוננו לא לחזור על קוד שכבר כתבנו.

נראה דוגמה:

```

class Employee:
    raise_mult = 1.04 # Used for raising salary

    def __init__(self, first, last, pay):
        self.first = first
        self.last = last
        self.email = first[:2]+'.'+last[:7] + '@email.com'
        self.pay = pay

    def fullname(self):
        return '{} {}'.format(self.first, self.last)

    def apply_raise(self):
        self.pay = int(self.pay*self.raise_mult)

```

```
##### MAIN #####
```

```

def main():
    e1 = Employee('Steph', 'Curry', 10000000)
    e2 = Employee('Phelippe', 'Coutinho', 12000000)

    print (e1.fullname())
    print (e2.fullname())

```

כעת ברצוננו לבדל בין עובדים רגילים ומפתחי תוכנה. כל הפרטים של עובד רגיל עדיין תקפים עבור מפתח תוכנה ולכן נשתמש בזה לעזרה.

```

class Developer(Employee):
    pass

```

כעת , למרות שהמחלקה Developer נראית 'ריקה', אפשר כבר להשתמש בה כדי ליצור עצמים, למשל:

```

d1 = Developer('Bill', 'Joy', 10000000)
d2 = Developer('Ken', 'Thompson', 200000, 'python')
print (d1.fullname())

```

כלומר , המחלקה Developer ירשה את כל התכונות והמתודות של Employee.

כעת נרחיב את Developer, למשל נשנה את המשתנה של המחלקה:

```
class Developer(Employee):
```

```
    raise_mult = 1.1
```

שימו לב שבתוך המתודה: `apply_raise` השתמשתי במשתנה המחלקה: `raise_mult` אבל לא נתתי לו את קידומת המחלקה כמו: `Employee.raise_mult` שזו הדרך הרגילה לציין משתנה ברמת המחלקה.

זה נעשה בכוונה, כדי להיות מסוגלים להשתמש בו עם ערך אחר במחלקה היורשת. כאשר פייטון רואה משתנה עם הקידומת `self` (או איך שלא תקראו לו לפרמטר הראשון של המתודה), הוא מחפש לראות אם קיים כזה באובייקט.

אם הוא לא קיים, הוא מחפש ברמת המחלקה (גם מחלקות הן אובייקטים בפייטון)

ולכן כאשר אתן העלאה לעובד, הוא יקבל רק 4%, אבל מפתח תוכנה יקבל 10%

```
print (e1.fullname(), e1.pay)
print (d1.fullname(), d1.pay)
```

לרוב במחלקה היורשת יהיו דברים שאינם קיימים במחלקת הבסיס. למשל בדוגמה הנ"ל אפשר להוסיף למתכנת עוד מאפיין שהוא שפת תכנות.

לשם כך נגדיר במחלקה `Developer` את המתודה `__init__` בתוספת המאפיין `.prog_lang`.

איננו צריכים להעתיק את האיתחול מהמחלקה `Employee` ומה שנעשה במקום זאת יהיה:

```
class Developer(Employee):
```

```
    raise_mult = 1.1
```

```
    def __init__(self, first, last, pay, prog_lang):
```

```
        super().__init__(first, last, pay) # Use the super class constructor
```

```
        self.prog_lang = prog_lang
```

מילת המפתח `super` בעצם אומרת להשתמש בהורה ממנו ירשנו, וכך מפעילים את ה- `__init__` שלו. היינו יכולים גם לכתוב את ההורה במפורש כך:

```
Employee.__init__(self, first, last, pay)
```

אולם כאשר נדבר על הורשה מרובה ניראה מדוע עדיף לדבוק בצורה הראשונה.

כעת המתודה `__init__` של `Developer` מוסיפה איתחול עבור שפת התכנות.

בואו וניצור מחלקה נוספת שיורשת מ-Employee שתקרא: Manager ועבור מנהל נוסף מאפיין שהוא רשימת עובדים שהוא מפקח עליהם, כאשר ברירת המחדל היא רשימה ריקה.

```
class Manager(Employee):
    def __init__(self, first, last, pay, employees=None):
        super().__init__(first, last, pay)
        if employees is None:
            self.employees = []
        else:
            self.employees = employees
```

נוסיף גם מתודות לצרף ולהסיר עובדים וגם להדפיס את הרשימה

```
def add_emp(self, emp):
    if emp not in self.employees:
        self.employees.append(emp)
```

```
def remove_emp(self, emp):
    if emp in self.employees:
        self.employees.remove(emp)
```

```
def print_emps(self):
    for emp in self.employees:
        print('--->', emp.fullname())
```

בהמשך להגדרות d1 ו-d2, נעשה שימוש לדוגמה ב-main:

```
mgr1 = Manager('Ada', 'Lovelace', 300000, [d1, d2])
mgr1.print_emps()
mgr1.remove_emp(d1)
mgr1.print_emps()
```

עד כאן ראינו כמה שימושי זה לרשת קוד ולהרחיבו על פי הצורך בשתי המחלקות היורשות שכתבנו. לסיכום החלק הזה נראה 2 פונקציות שבודקות את ההיררכיה של המחלקות: `issubclass`, `isinstance` ונראה את פעולתן על ידי דוגמאות:

```
print (isinstance(mgr1,Manager))      # True
print(isinstance(mgr1, Employee))    # True
```

```
print(isinstance(mgr1, Developer))    # False
print(issubclass(Developer, Employee)) # True
print(issubclass(Manager, Developer)) # False
print(issubclass(Manager, Employee))  # True
print(issubclass(Employee, Manager))  # False
```

בשורה השניה קיבלנו True למרות ש- mgr1 אינו אובייקט ישיר של Employee,
אולם זה עדיין באותה הירארכיה, כי Manager ירש מ- Employee