תכנות מונחה עצמים בפייתון

תכנות מונחה עצמים הוא מתודולוגיה תכנותית שמפרידה בין חלקי קוד לאובייקטים שמתקשרים בניהם.
הרעיון של תכנות מונחה עצמים הוא פירוק התוכנית לחלקים עצמיים שמתקשרים בניהם כך שלא נצטרך למחזר קוד, כלומר כל שינוי שמשותף לכמה חלקים בקוד יתבצע במקום אחד במקום בכל חלק בנפרד.
הגישה מנסה לחקות את העולם האמתי בכך שהיא מציגה תכונות והתנהגויות משותפות במבנה אחד.
כאשר הקוד מאורגן לפי אובייקטים המייצגים עצמים, הדבר מאפשר שימוש חוזר בתוכנה, עם אפשרות טובה להרחיב את הקוד גם בתכוניות אחרות, כמו כן מעבר טבעי יותר בין התכנון של הקוד ומימוש שלו.
כבר בשפת C נראים ניצנים לשיטה עם פיתוח משתנה מטיפוס struct שניתן להגדיר לו "מהות" (מה הוא בא לייצג) ותכונות, למשל בן אדם- יש לו שם, גובה, מוצא, טמפרמנט וכו'
בשפות שממשיכות את C ניתן גם להגדיר למבנה שיטות, כלומר איך האובייקט מתנהג, למשל אדם יש לו את השיטה "להגיד שלום", "ללכת", "להשתעל", "להוציא את הילד מהגן" וכו'.
בג'אווה ובפייתון תכנות מונחה עצמים מיוצג במחלקות שהן התבנית של העצם, דהיינו התכונות והתנהגויות המשותפות לכל האובייקטים מאותו המבנה.
למחלקות יכולות להיות: שדות(fields) מאפייני המחלקה כלפי פנים, תכונות(properties) המגדירים מאפיינים של המחלקה כלפי חוץ, ושיטות (methods) שהן פונקציות הפועלות של משתני המחלקה או מגדירות את התנהגות העצם.
אובייקט הוא בעצם דוגמא(אינסטנס) לעצם מטיפוס המחלקה, ניקח למשל את הדוגמא שהצגנו מקודם- אדם, לאדם יש תכונות כמו שם, גובה, מוצא שלרוב אפשר לראות כלפי חוץ.
יש לו גם שדות כמו מצב נפשי, מצב בריאותי וכו' שעלולים לשנות את מצבו החיצוני, אבל אינם גלויים כלפי חוץ.
ויש לו גם שיטות שמתארות את ההתנהגות שלו או פועלות על המשתנים אחרים, למשל בנאדם שמצבו הבריאותי לא תקין יכול להפעיל איזו שיטה שתשנה את גוון עורו וכו'.
אובייקט הוא דוגמא לטיפוס מהמחלקה(instance), אם המחלקה מייצגת "בן אדם" , אז אובייקט הוא למשל "אורנית" או "בני" וכו'.

מאפייני השיטה-
המאפיינים המרכזיים של השיטה הם : כימוס(encapsulation), הפשטת נתונים(data abstraction), ירושה (inheritance) ורב צורתיות (polymorphism) .

כימוס- מהמילה כמוס כלומר מוסתר(כמו סוד כמוס לפרה ולסוס), הוא מצב בו כל אובייקט שומר מצב פרטי במחלקה, כך שלאובייקטים אחרים אין גישה ישירה למצב הזה, הם יכולים רק לקרוא לשיטות מרשימה של שיטות ציבוריות.
אז האובייקט מנהל את המצבים של עצמו דרך שיטות, ואף מחלקה אחרת לא יכולה לגעת באותן שיטות אלא אם ניתן לה רשות.
נניח יש לנו מחלקה של חתול ומחלקה של בן אדם, והמחלקות מתקשרות בניהן.
לחתול נניח יכולים להיות שדות כמו אנרגיה, מצב רוח, וכו', וכמו כן יש לו שיטות פרטיות כמו "לייל" או "להקיא כדור פרווה".
המחלקה של הבן אדם לא יכולה להפעיל את המתודה של החתול "לייל", אבל היא יכולה לבצע מתודה של בן אדם שיכולה לעורר את המתודה "לייל" של החתול, כמו "ללטף" או "לבעוט".

פישוט- אפשר לראות את הפישוט כהמשך טבעי לכימוס.
בעיצוב של תוכנית מונחת עצמים, תוכניות לעיתים קרובות גדולות מאוד, ואובייקטים מופרדים מתקשרים בניהם הרבה, אז לתחזק קוד בסיס כזה למשך שנים עם שינויים לאורך הדרך הוא מסובך.
פישוט הוא רעיון שנועד להקל על הבעיה הזו.
מימוש הפשטה אומר שכל אובייקט אמור לחשוף רק מנגנון ברמה גבוה לשימוש בו.
המנגנון הזה אמור להסתיר פרטי מימוש פנימיים, ואמור לחשוף רק פעולות שרלוונטיות למשתמש או לאובייקטים אחרים.
ניקח לדוגמא טלוויזיה , היא עושה הרבה מאחורי הקלעים, אבל כל מה שאנחנו עושים בשביל להשתמש בה הוא ללחוץ על כפתורים בשלט.
אנחנו מצפים שהמנגנון הזה יהיה פשוט לשימוש ושישמר לאורך זמן.

ירושה- אובייקטים הרבה פעמים זהים מאוד אך אינם זהים לחלוטין.
כדי שנוכל לשמור על מבנה זהה בין האובייקטים אך עם שינויים בניהם נשתמש בירושה.
כשמחלקה יורשת ממחלקה אחרת היא בעצם לוקחת את כל המאפיינים של המחלקה המורישה, ומוסיפה לה מאפיינים משלה.
למשל ניקח את המושג עוף יש עופות שיכולות לעוף לשחות או לקרוא "קוקוריקו", ויש גם עופות שאין להם אף אחת מהשיטות האלה, אך המשותף למשפחת העופות זה שיש להם כנפיים מקור והם מתרבים בהטלת ביצים.
אם התוכנית שלנו תצטרך להשתמש באובייקט תרנגולת, ברוז וברבור אנחנו יכולים ליצור היררכיית ירושה כך שהברבור ירש מהברוז, והברוז והתרנגולת ירשו ממחלקת "עוף", ואז כל מחלקה תוסיף מה שרלוונטי לה לתכונות שהיא יורשת מהן.

פולימורפיזם- מקור המילה מיוונית ופרושה רב צורתיות.
אנחנו כבר הבנו את הרעיון של ירושה ויודעים את הכוח שלה, אבל לפעמים כל מחלקה שיורשת תכונה או שיטה מסוימת מפרשת אותה אחרת ממה שהמחלקה המורישה מפרשת אותה למשל אם למחלקת אדם יש תכונה "מקום מגורים", ומתודה "להגיד שלום" אם ניצור אינסטנס של בן אדם מישראל ואחד מספרד כל אחד יפרש את המאפיינים האלה אחרת, לאחד יקראו נגיד יוסי ולשני חוליו ,אחד יגיד "שלום" והשני "hola" וכו'.
פולימורפיזם מאפשר להשתמש במחלקה בדיוק כמו במחלקת האב שלה, כלומר יש לה בדיוק את אותם שדות, אבל כל מחלקת יורשת שומרת על המאפיינים האלה כפי ראות עיניה.
הפולימורפיזם נותן דרך שבה נוכל לשמור כמה אינסטנסים של מחלקות שונות אך יורשות מאותה מחלקת אב בתוך מקום אחד ולהשתמש בשיטה או תכונה ממחלקת האב כפי שהמחלקות היורשות ממשות אותן, מבלי לחשוש לשגיאות.

הגדרת מחלקה בפייתון-

כל מחלקה מתחילה במילה השמורה class אחרי יבוא שם המחלקה (שמקובל לכתוב אותו ב-CapitalizedWords), נקודתיים ובבלוק מתחת את מימוש המחלקה:

```
class MyClass:
    pass
```

אתחול מחלקה והגדרת המאפיינים שלה יעשה בפונקציית הבנאי, בשפות כמו ג'אווה או c++ שם המתודה הוא כשם המחלקה, בפייתון מציינים את הבנאי ב- __init__ (שני קווים תחתונים ברישא ובסיפא), הסימון של פונקציות עם שני קווים תחתונים בהתחלה ובסוף הוא חשוב, בהמשך נדבר יותר על סוג הפונקציות האלה.
הבנאי אמור לקבל משתנה מסוג self , ואת המשתנים שהוא אמור לאתחל.
הפרמטר self הוא בעצם האובייקט שפועלים עליו, כלומר קבלת האינסטנס של האובייקט כפרמטר.
בג'אווה ודומותיה למשתנה קוראים this, והוא אומנם לא מוגדר בחתימת הפונקציה כאחד הפרמטרים, אך מאחורי הקלעים הוא נשלח גם לפונקציה, גם בפייתון לא נשלח את האובייקט כארגומנט ביצירת אינסטנס חדש זה, יעשה אוטומטית כשיוצר האובייקט, אבל עדיין צריך להגדיר את הפרמטר בחתימה המתודה.
ההכרזה על המשתני העצם של האינסטנס בפייתון תעשה במתודת הבנאי:

```
class Dog:
    def __init__(self, name, age, gender='male'):
        self.name = name
        self.age = age
        self.gender = gender
rexi = Dog('Rexi' ,3)
```

משתני העצם של האינסטנס יכולים להשתנות בין אובייקטים שונים מאותו טיפוס מחלקה, יכול להיות שיש כלב שקוראים לו גופי בן 2 וכלבה אחרת שקוראים לה עזית בת 5.
לעומת זאת יש מה שנקרא משתני מחלקה או תכונות מחלקה, שהם תכונות שאמורות להיות משותפות לכל האובייקטים מטיפוס המחלקה.
נהוג להצהיר על משתני המחלקה מתחת לחתימת המחלקה, והם חייבים לקבל ערך ישר עם הצהרתם (אחרת נקבל

שגיאת NameError) .
כשנוצר אובייקט מטיפוס המחלקה, תכונות המחלקה נוצרות אוטומטית ומקבלות את ערכן המאותחל.
הערה: תכונות המחלקה הן אומנם תכונות שמשותפות לכל אובייקט מטיפוס המחלקה, אך הן לא סטטיות, כלומר אם נשנה את התכונה באחד האינסטנסים הוא ישתנה רק באותו אינסטנס ולא בכל השאר:

```
class Dog:
    species = "Canis familiaris"
    def __init__(self, name, age, gender = 'male'):
        self.name = name
        self.age = age
        self.gender = gender

rexi = Dog("Rexi", 3)
kookoo = Dog("Kookoo", 0.5)
kookoo.species = "Gallus gallus"
print(f"Kookoo species: {kookoo.species}")
print(f"Rexi species: {rexi.species}")
```

בפייתון אין מילה שמורה למשתנים קבועים כמו final או const
שאלה למחשבה: איך הכל זאת ניתן להגדיר משתנה קבוע בפייתון?

מתודות-
בדוגמאות שהצגנו למעלה הצגנו שלאובייקט יכולים להיות שדות שמיצגים תכונות ויכולות להיות מתודות שמייצגות את התנהגות של האובייקט.
מתודות האינסטנס הן מתודות שמגדירות את האינסטנס הספציפי מטיפוס המחלקה, מה הכוונה? לכל אינסטנס של מחלקה יכול להיות את הפירוש שלו לתכונות המחלקה כפי שראינו לעיל בדוגמא של לאסי וגופי.
למתודות שבהם אנו צריכים את פירוש האינסטנס לתכונות נקרא מתודת האינטנס, במילים אחרות אם אנחנו משתמשים בפרמטר self של המחלקה אנחנו משתמשים במתודת אינסטנס.
כאמור מתודות אינסטנס מוגדרות כמו פונקציות רק שהפרמטר הראשון שלהן הוא self :

```
class Dog:
    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Instance method
    def description(self):
        return f"{self.name} is {self.age} years old"

    # Another instance method
    def speak(self, sound):
        return f"{self.name} says: {sound}"

rexi = Dog("Rexi", 3)

print(rexi.speak("I'm a dog, I can't speak!"))
print(rexi.description())
```

```
Rexi says I'm a dog, I can't speak!
Rexi is 3 years old
```

מתודות המחלקה הן מתודות שמוגדרות לכלל טיפוסי המחלקה ללא תלות בהגדרת משתני עצם של המחלקה, בדוגמא שלנו- לא אכפת לנו אם לכלב קוראים רקסי לאסי גופי וכו' ,המתודה באה להגדיר תכונה שמשותפת לכלל הכלבים.
לחתימה של מתודת מלקה נוסיף ופרמטר cls שהוא מצביע על המחלקה ולא על האינסטנס בקריאת המחלקה, ומשום כך המתודות לא יכולות לשנות את משתני האינסטנס, אך הן יכולות לשנות את משתני המחלקה וגם אז השינוי יעשה רק באותו האינסטנס שקרא למתודה:

```
class Dog:
    species = "Canis familiaris"

    def __init__(self, name, age , gender='male'):
        self.name = name
        self.age = age
        self.gender= gender
    def mutation(cls, new_species):
        cls.species = new_species


catdog_mutation= Dog("Experiment X",0.2,gender='unknown')
catdog_mutation.mutation('Felis silvestris')
guffy= Dog('Guffy',2)
print(f"Experiment X species: {catdog_mutation.species}")
print(f"Guffy species: {guffy.species}")
```

```
Experiment X species: Felis silvestris
Guffy species: Canis familiaris
```

אבל אם נשתמש בקשטן @classmethod המתודה כבר לא תתבצע על האינסטנס שקרא למתודה אלא על כלל ההאינסטנסים של המחלקה:

```
class Dog:
    species = "Canis familiaris"

    def __init__(self, name, age , gender='male'):
        self.name = name
        self.age = age
        self.gender= gender
    @classmethod
    def contagious_mutation(cls, new_species):
        cls.species = new_species
        print('>:-) whahaha')
        print('You have infected all the dogs in the world!')
```

```
guffy= Dog('Guffy',2)
catdog_mutation= Dog("Experiment X",0.2,gender='unknown')
catdog_mutation.contagious_mutation('Felis silvestris')
print(f"Experiment X species: {catdog_mutation.species}")
print(f"Guffy species: {guffy.species}")
```

```
>:-) whahaha
You have infected all the dogs in the world!
Experiment X species: Felis silvestris
Guffy species: Felis silvestris
```

ויש עוד סוג של מתודות והן מתודות סטטיות.
מתודות סטטיות לא מקבלות פרמטר מיוחד (self או cls) והן לא נועדו לשינוי האינסטנס או תכונה של המחלקה.
למחלקות סטטיות אנחנו מוסיפים את הקשטן @staticmethods כדי לסמן למפרש שהמתודה לא משמשת לשינוי המחלקה, אלא לשימוש כללי תחת שם המחלקה.
לרוב זה שימושי כדי לתת namespace לקבוצת מתודות תחת שם של אובייקט אחד מבלי ליצור אינסטנס שלו:

```
class Maze:
    @staticmethod
    def get_predetermined_maze():
        return  f"""
            # # # # # # # # # # # # # # # #
            # S                     #   #
            #         #             #   #
            # # # #             #   #   #
            #                   #   #   #
            #               # # # # #   #
            #                           #
            #                           #
            # # # # # #       # # # #   #
            #                         E #
            # # # # # # # # # # # # # # # #
        """
    @staticmethod
    def maze_solver(maze:str)-> str:
        #TODO: implement maze solver using BFS
        pass

    @staticmethod
    def maze_generator()->str:
        #TODO: implement the maze generator using prim
        pass
```

ירושה ורב צורתיות בפייתון-

כפי שאמרנו קודם ירושה היא אחת המאפיינים של תכנות מונחה עצמים.
בפייתון ירושה מתבצעת ע"י הוספה של סוגריים ובתוכם שם המחלקה שמהן המחלקה יורשת, נקודותיים ומימוש רגיל של מחלקה.
מחלקה יכולה לרשת מכמה מחלקות, ולא כמו בג'אווה אין הגבלה למספר המחלקות שניתן לרשת מהן.
כל מחלקה נוספת שיורשים ממנה מוספת כפי שמוסיפים פרמטרים לפונקציה, ע"י ',':

```
class Kid1(Parent1):
class Kid1(Parent1, Parent2):
class Kid1(Parent1, Parent2, Parent3):
class Kid1(Parent1, Parent2, Parent3, Parent4):
class Kid1(Parent1, Parent2, Parent3, Parent4, Parent5):
```

נהוג לכנות את המחלקה המורישה מחלקת אב, והמחלקה היורשת מחלקת בן.
בעת ההורשה כל שדה או מתודה שיש למחלקת האב מועתקת למחלקת הבן בשלמותה (כולל המימוש שלה).
ניתן לזהות האם מחלקה מסוימת יורשת ממחלקה אחרת ע"י הפונקציה issubclass(), ואם האובייקט הוא אינסטנס של מחלקה מוסיימת עם הפונקציה isinstance() :

```
class Vehicle:
    pass

class Car(Vehicle):
    pass


volvo = Car()
#isinstance
print(f"is volvo an instance of Car? {isinstance(volvo, Car)}")
print(f"is volvo an instance of Vehicle? {isinstance(volvo, Vehicle)}")

#issubclass
print(f"is Vehicle a subclass of Car? {issubclass(Vehicle, Car)}")
print(f"is Car a subclass of Vehicle? {issubclass(Car, Vehicle)}")
```

```
is volvo an instance of Car? True
is volvo an instance of Vehicle? True
is Vehicle a subclass of Car? False
is Car a subclass of Vehicle? True
```

נשים לב שאינסטנס של מחלקת הבן הוא גם אינסטנס של מחלקת האב.
למעשה כל האובייקטים בפייתון, כמו ב- C#, הם אינסטנסים של המחלקה object ויורשים ממנה:

```
issubclass(int, object) #--> True
isinstance(int, object) #--> True
```

לפעמים נרצה לשנות תכונה של מחלקת האב במחלקת הבן, למשל נניח יש לנו מחלקה שקוראים לה 'מדינה' ויש לה תכונות כמו שפה, עיר בירה, מספר תושבים וכו'. אלה תכונות שמשותפות לכלל המדינות, אם ניקח את ארה"ב למשל השפה המדוברת במדינה היא אנגלית (לרוב), מספר התושבים כ-שלוש מאות מליון נפש, עיר הבירה: "ושינגטון די סי" וכו', אבל התכונות הן של ארה"ב בלבד, אם ניקח את ישראל התכונות יהיו שונות לחלוטין.

במקרים כאלה נרצה לעשות דריסה של תכונות של מחלקת האב, ולהתאימן למחלקת הבן.
בפייתון הדריסה היא פשוט כתיבה של אותה מתודה או תכונה שוב, ולא להתבלבל עם הסימון @override של ג'אווה,
בג'אווה הסימון הוא annotation למתודה, בפייתון ה-'@' מסמן קשטן, **ואין** קשטן override:

```
class Country():
    def capital(self):
        return "{} is the the capital of {}"

    def language(self):
        return "{} is the the language of {}"

    def population(self):
        pass

class Usa(Country):
    def capital(self):
       return super().capital().format("Washington, D.C.","USA")

    def language(self):
        return super().language().format("English","USA")

    def population(self):
        return '~328,239,523'

usa= Usa()
print("USA:")
print(f"Capital: {usa.capital()}")
print(f"language: {usa.language()}")
print(f"population: {usa.population()}")
```

```
USA:
Capital: Washington, D.C. is the the capital of USA
language: English is the the language of USA
population: ~328,239,523
```

הפונקציה super() מאפשרת להשתמש בתכונות של מחלקת האב. כפי שנראה בדוגמא השתמשנו בפונקציה כדי לקבל את המימוש של מחלקת האב למתודות שמייצגות את השפה של המדינה והבירה שלה, ואז עם הפונקציה format שיינינו את המבנה הקיים של המחרוזת שיתאים לארה"ב.

סדר ביצוע מתודות-
נניח יש לנו מחלקה A מחלקה B עם שם של מתודה משותפת, ומחלקה C שיורשת משתי המחלקות הראשונות,
מבין שתי המחלקות המתודה של מי מבניהן תופעל?

```
class A:
    def my_name(self):
        print("My name is class A")

class B:
    def my_name(self):
```

```
        print("My name is class B")
class C(A,B):
    pass
C().my_name()
```

```
My name is class A
```

המתודה ()my_name של המחלקה C הפעילה דווקא את המתודה של המחלקה A ולא את שם המחלקה B.
הסיבה היא די פשוטה לפייתון יש מערכת MRO (Method Resolution Order) שמבצעת את המתודות לפי סדר הירושה, ובהיקרא למתודה של המחלקה היורשת המערכת עובר על כל האובייקטים מהם ירשה המחלקה לפי סדר הירושה ובודקת מי המחלקה הראשונה שיש לה את המתודה שנקראה.
כך במקרה שלנו המחלקה A היא הראשונה בסדר הירושה לכן המתודה שלה בוצעה בסוף.
האם ניתן לעקוף את ה-MRO של פייתון? ואם נרצה להשתמש במתודות של כלל המחלקות?
ניתן לעקוף ואף להשתמש בפונקציה של מחלקות אחרות עם ע"י קריאה לאובייקט מטיפוס המחלקה והפעלת המתודה דרכו עם הוספת ארגומנט self:

```
class C(A,B):
    def my_name(self):
        B.my_name(self)
        A.my_name(self)

C().my_name()
```

```
My name is class B
My name is class A
```

מתי העסק מתחיל להסתבך? כאשר אנחנו משתמשים בירושה משתי מחלקות שיורשות מאותה מחלקת אב.
נתסכל על הדוגמא הבאה:

```
class DomesticatedAnimel(object):
    def use(self):
        return "the {} is used for {}"

class Horse(DomesticatedAnimel):
    def use(self):
        return super().use().format(self.__class__.__name__, "riding and freight and racing")

class Donkey(DomesticatedAnimel):
    def use(self):
        use_str= super().use().format(self.__class__.__name__, "riding and freight")
        use_str+=" ,did I tell that I'm a donkey?"
        return use_str

class Mule(Donkey,Horse):
    pass

print("Horse:")
print(Horse().use())
print("Donkey:")
print(Donkey().use())
```

```
print("Mule:")
muly= Mule()
print(muly.use())
```

בדוגמא לעיל יש מחלקה שמייצגת חיות מבויתות עם מתודה אחת "שימוש" .
ממנה יורשים שתי מחלקות: "סוס" ו-"חמור" ולשתיהן יש מימוש שונה למתודה –הסוס משמש למשא, רכיבה ומרוצים, והחמור משמש רק למשא ורכיבה, אך הוא גם חמור והוא לא זוכר אם הוא הציג את עצמו(מסתבר), אז הוא מזכיר את זה.
משתי המחלקות יורשת מחלקת "פרד" (כלאיים של סוס וחמור).
לכאורה ההדפסה שנצפה לקבל מהפונקציה `Mule.use()` היא כמו ההדפסה של החמור כי הוא הראשון בסדר הירושה, אבל קורא כאן דבר מעניין:

```
Horse:
the Horse is used for riding and freight and racing
Donkey:
the Donkey is used for riding and freight ,did I tell that I'm a donkey?
Mule:
the Mule is used for riding and freight and racing ,did I tell that I'm a donkey?
```

אפשר לומר שגם ההדפסה שקיבלנו היא סוג של כלאיים בין המתודה של הסוס והמתודה של החמור- מסתבר שהפרד משמש לרכיבה למשא ולמרוצים והוא גם שכחן כמו החמור ומזכיר לנו שהוא חמור.
למה זה?
הסיבה היא כי אנחנו משתמשים במחלקה "חיות מבויתות" עבור שתי מחלקות הבסיס של הפרד- כשמחלקה יורשת ממחלקה אחרת היא יורשת ממנה לא רק את המתודות אלא גם את כל המחלקות ממנה מחלקת האב יורשת, לכן כשהפרד יורש מהחמור ומהסוס הוא אוטומטית יורשת גם ממחלקת חיות הבית.
כשמשתמשים במתודה "שימוש" של מחלקת האב, סדר הפעלת המתודות של פייתון מחפש את המחלקה הראשונה ממנה יורש ה"פרד" ,שגם לה יש את השימוש במתודה הזו והיא אינסטנס של "חיות מבויתות" , במקרה הנ"ל זאת המחלקה "חמור", אבל המתודה בחמור משתמשת במתודה של מחלקת האב שלה שהיא "חיות מבויתות".
אז מה שעושה ה-MRO זה לחפש את המחלקה הבאה מבין במחלקות שיורש מהן ה"פרד" ,שמשתמשת באותה המתודה וצאצא של המחלקה "חיות מבויתות" מנקודת הנחה שהיא מחלקת האב, היות והמחלקה הבאה היא המחלקה- "סוס" תופעל המתודה "שימוש" של הסוס.
היא גם מפעילה את המתודה של האב "חיות מבויתות" כשהיא קוראת ל-super(), לכן ה-MRO יחפש את המחלקה הבאה ברשימה שיש לה את המתודה ויורשת מ-"חיות מבויתות" (אינסטנס של "חיות מבויתות"), ובמקרה זה המחלקה "חיות מבויתות" ,ואז סוף סוף תופעל המתודה של המחלקה תחזור אחורה למחלקה "סוס" שיחזיר את ערכו למחלקה "חמור" שיחזיר את הערך למחלקה "פרד" וחד גדיא חד גדיא.
בשביל לראות את זה טוב יותר נוסיף לכל אחת מהמחלקות במימוש של המתודה "שימוש" הדפסה בתחילת המתודה, שמירה של המחרוזת במשתנה, הדפסה של יצאה מהמחלקה והחזרה של המחרוזת:

```
class DomesticatedAnimel(object):
    def use(self):
        print("in DomesticatedAnimel")
        print("bye bye domesticated animal ")
        return "the {} is used for {}"

class Horse(DomesticatedAnimel):
    def use(self):
        print("in Horse")
        use_str= super().use().format(self.__class__.__name__,"riding and freight and racing")
        print("bye bye horse")
        return use_str

class Donkey(DomesticatedAnimel):
```

```
    def use(self):
        print("in Donkey")
        use_str= super().use().format(self.__class__.__name__, "riding and freight")
        use_str+=" ,did I tell that I'm a donkey?"
        print("bye bye donkey")
        return use_str

class Mule(Donkey,Horse):
    pass


print("Mule:")
muly= Mule()
print('\n',muly.use())
```

```
Mule:
in Donkey
in Horse
in DomesticatedAnimel
bye bye domesticated animal
bye bye horse
bye bye donkey

the Mule is used for riding and freight and racing ,did I tell that I'm a donkey?
```

אם רוצים לראות את סדר הפעלת המתודות של מחלקה מסוימת אפשר להשתמש במשתנה __mro__ שמציג איזו מחלקה מגיעה קודם בסדר הירושה:

```
Mule.__mro__
```

```
(__main__.Mule,
 __main__.Donkey,
 __main__.Horse,
 __main__.DomesticatedAnimel,
 object)
```

כימוס ואבסטרקציה בפייתון?

כפי שאמרנו בהקדמה, כימוס הוא אחד המאפיין הראשיים של תכנות מונחה עצמים, והוא מאפשר לנו הסתרה של שדות ומתודות שלא קשורות ישירות לייעוד המחלקה מהמשתמש.
למה בכלל צריך להסתיר מידע מהמשתמש?
ניקח את הדוגמא הבאה, יש לנו מחלקה שסופרת מספר אנשים שבחרו המועמד מסוים במחלקה "קלפי".
אם תהיה למשתמש גישה למשתנה הוא יכול לשנות את מספר האצבעות למועמד מסוים:

```
class BallotBox:
    def __init__(self, parties, name):
        self.candidates = parties
        self.voters = {}
        self.name = name

    def add_a_vote(self, voter_id,candidate):
        """Adds new vote to the ballot box """
        if(not voter_id in self.voters) and (candidate in self.candidates):
            self.candidates[candidate] +=1
            self.voters[voter_id] = True
            print("Thank you for voting :-)")
        else : print("Voter has already voted >:-(")

    def mina_zemach(self):
        """Shows the Election poll"""
        print('\nMina Zemach election poll:')
        for key,value in self.candidates.items():
            print(f"The party '{key}' has {value:,} votes")

parties = {
    'Bibi something':0,
    'To right':0,
    'The work':0,
    'Lapid something':0,
    'vigor' : 0
}
ballot_box = BallotBox(parties, 'TLV')
ballot_box.candidates['Bibi something'] =1000000
ballot_box.mina_zemach()
```

```
Mina Zemach election poll:
The party 'Bibi something' has 1,000,000 votes
The party 'To right' has 0 votes
The party 'The work' has 0 votes
The party 'Lapid something' has 0 votes
The party 'vigor' has 0 votes
```

משום שנתנו גישה למילון של המחלקה "קלפי" המשתמש יכל לשנות את הערכים של המועמדים.
בפייתון אין מילה שמורה private למשתנים, לכאורה יש דרך להגדיר משתנה שאין אפשרות להשתמש בו מחוץ למחלקה

בצורה ישירה אם מגדירים את שם המשתנה עם שני קווים תחתונים בתחילת שמו, זה יחשיב את המשתנה כלא שייך למחלקה כאשר קוראים לו בחוץ אבל מתוך המחלקה האובייקט נחשב בדיוק אותו דבר.
נוכל עכשיו לשנות את שם המשתנה candidates ל- __candidates, ואת ה-voters ל- __voters.
וכמו שיש משתנים פרטיים ניתן גם להגדיר מתודות פרטיות בצורה זהה- שם המתודה מתחיל בשני קווים תחתונים.
ועכשיו שאנחנו יודעים את כל זה בואו "נתקן" את המחלקה ונוסיף כמה מתודות חדשות:

```
class BallotBox:
    def __init__(self, parties , name):
        self.__candidates = parties
        self.__voters = {}
        self.name = name

    def add_a_vote(self, voter_id,candidate):
        """Adds new vote to the ballot box """
        if(not voter_id in self.__voters) and candidate in self.__candidates:
            self.__candidates[candidate] +=1
            self.__voters[voter_id] = True
            print("Thank you for voting :-)")
        else : print("Voter has already voted >:-(")

    def __the_greatest_paty(self):
        """returns the greatest party in the ballot box """
        import operator
        greatest_party= max(self.__candidates.items(), key=operator.itemgetter(1))[0]
        return greatest_party


    def mina_zemach(self):
        """Shows the Election poll"""
        greatest_party = self.__the_greatest_paty()
        print('\nMina Zemach election poll:')
        for key,value in self.__candidates.items():
            # print the greatest party with yellow
            # the colors where taken form:
            # https://svn.blender.org/svnroot/bf-blender/trunk/blender/build_files/scons/tools/bcolors.py
            if key == greatest_party:
                print(f"The party '\033[93m{key}\033[0m' has {value:,} votes")
            else: print(f"The party '{key}' has {value:,} votes")

    def add_a_candidate(self,name):
        """Adds new party to the ballot box """
        if not name in self.__candidates:
            self.__candidates[name] = 0
        else: print("The party is already running")
parties= {
    'Bibi something':0,
    'To right':0,
    'The work':0,
    'Lapid something':0,
```

```
    'vigor' : 0
}
ballot_box = BallotBox(parties , 'TLV' )
ballot_box.add_a_candidate("Halad balad")
ballot_box.add_a_candidate("Haredim la athid")
ballot_box.add_a_vote(voter_id="1234",candidate='The work')
ballot_box.add_a_vote(voter_id='22222',candidate='Bibi something')
ballot_box.add_a_vote(voter_id="1234",candidate='vigor')
#ballot_box.__candidates['Bibi something'] =1000000 => Error
ballot_box.mina_zemach()
```

```
Thank you for voting :-)
Thank you for voting :-)
Voter has already voted >:-(
Mina Zemach election poll:
The party 'Bibi something' has 1 votes
The party 'To right' has 0 votes
The party 'The work' has 1 votes
The party 'Lapid something' has 0 votes
The party 'vigor' has 0 votes
The party 'Halad balad' has 0 votes
The party 'Haredim la athid' has 0 votes
```

שימו לב שהמשתנה name עדיין ציבורי ,וזה בכוונה כדי להוביל לנושא הבא:

האם באמת המחלקות והמתודות הן פרטיות כמו שאנחנו חושבים?
אמם.. לא ממש, עדיין אפשר לגשת אליהן למרות ש-"כמסנו" אותן.
בפייתון יש אפשרות לגשת לתכונות ומתודות נסתרות ע"י שימוש ב-name mangling, כדי להבין את זה יותר טוב נשתמש במשתנה __dict__ של האובייקט ונראה משהו מעניין:

```
ballot_box.__dict__
```

```
{'_BallotBox__candidates': {'Bibi something': 1,
  'To right': 0,
  'The work': 1,
  'Lapid something': 0,
  'vigor': 0,
  'Halad balad': 0,
  'Haredim la athid': 0},
 '_BallotBox__voters': {'1234': True, '22222': True},
 'name': 'TLV'}
```

המשתנה __dict__ הוא מילון שהמפתחות שלו הם שם השדה והערכים הם מה שהוא מכיל,
כך ניתן לראות מה הם השדות של המחלקה.
אנחנו יודעים שלמחלקה יש שני שדות שמורים: המילון שמתאר את המצביעים (voters) והמילון של המעומדים (candidates).
משום שהגדרנו את השדות פרטיים הוספנו שני קווים תחתונים לתחילת שמם, ולכאורה היינו אמורים לראות את השם שהגדרנו, וההוכחה היא שהמשתנה name עדיין נקרא name.
אותו דבר אם נריץ את הפקודה dir() על המחלקה, שאמורה לתת לנו את כל התכנות של המחלקה כולל השיטות שלה ,
נראה שהפונקציה "משנה" את השם של המתודות והשדות הפרטיים ל- _BallotBox__<name of the method> .
לדבר הזה קוראים name mangling, והוא קורה משום שהגדרנו למפרש שהשיטות\התכונות הן פרטיות, אז כדי שלא

נוכל לגעת בהן אבל המפרש כן הוא יוצר העתק של המתודות עם הname mangling , כך שמחוץ למחלקה המתודות באמת לא קיימות, אבל העתק שלהן קיים ואפשר להשתמש בו גם מחוץ למחלקה:

```
print(">:-) whahaha: ")
ballot_box._BallotBox__candidates["Bibi something"] += 1000000000
ballot_box.mina_zemach()
```

```
>:-) whahaha:
Mina Zemach election poll:
The party 'Bibi something' has 1,000,000,001 votes
The party 'To right' has 0 votes
The party 'The work' has 1 votes
The party 'Lapid something' has 0
votes The party 'vigor' has 0
votes The party 'Halad balad' has 0 votes
The party 'Haredim la athid' has 0 votes
```

השימוש בשני קווים תחתונים בתחילת השם אומנם מספק לנו סוג של "שמירה על פרטיות המשתנה" אבל זה לא היעוד המקורי שלהם, נשתמש בהם כאשר נרצה למנוע דריסה של אובייקט או תכונה של מחלקה כתוצאה מירושה, או כדברי המדריך הרשמי לכתיבת קוד נכון בפייתון (PEP8):

If your class is intended to be subclassed, and you have attributes that you do not want subclasses to use, consider naming them with double leading underscores and no trailing underscores. This invokes Python's name mangling algorithm, where the name of the class is mangled into the attribute name. This helps avoid attribute name collisions should subclasses inadvertently contain attributes with the same name.

נמחיש את זה עם הדוגמא הבאה:

```
class A:
    def __private_function(self):
        print("what are you doing here?!")
    def public_function(self):
        print("wellcome")
class B(A):
    def __private_function(self):
        print("*What are you doing here?!")
    def public_function(self):
        print("*Wellcome")
b= B()
b._A__private_function()
b._B__private_function()
print(dir(B))
```

```
what are you doing here?!
*What are you doing here?!
['_A__private_function', '_B__private_function', '__class__', '__delattr__', '__dict__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
```

```
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__',
'public_function']
```

המחלקה B שירשה מהמחלקה A ניסתה לדרוס את שתי המתודות של מחלקת האב (מתודה אחת פרטית והשנייה ציבורית), ואם מסתכלים על תכונות המחלקה רואים שיש לה גם את המתודה הפרטית של מחלקת האב למרות שדרסנו אותה('_A__private_function'), לעומת זאת המתודה הציבורית אכן נדרסה.

ברמת העיקרון כימוס הוא לא מאפיין מצוי בפייתון כמו בשפות אחרות, או כפי שאמר גוידו מפתח השפה:
"we're all consenting adults here"
ועדיין מוגדר במדריך הרשמי לכתיבת קוד נכון בפייתון כי:

Always decide whether a class's methods and instance variables (collectively: "attributes") should be public or non-public. If in doubt, choose non-public; it's easier to make it public later than to make a public attribute non-public.

Public attributes are those that you expect unrelated clients of your class to use, with your commitment to avoid backwards incompatible changes. Non-public attributes are those that are not intended to be used by third parties; you make no guarantees that non-public attributes won't change or even be removed.

We don't use the term "private" here, since no attribute is really private in Python (without a generally unnecessary amount of work).

אז אם ככה איך נוכל לייצג משתנה לא ציבורי בפייתון ועדיין לדרוס אותו בירושה?

Use one leading underscore only for non-public methods and instance variables.

השימוש בקו תחתון אחד בתחילת השם הוא יותר נכון מבחינת פייתון, אמנם הוא לא "מפריט" את השדה או המתודה, אבל הוא מורה למתכנת אחר כי השדה או הפונקציה נועדו לשימוש פנימי של המחלקה ולא לשימוש הכלל.

שאלה למחשבה: האם לפי דעתכם זה רעיון טוב שאין כימוס מלא בפייתון?

-@properties

בשפות כמו ג'אווה או #c שבהן שכיח למצוא משתנים פרטיים, יש את האפשרות "לערוך" מה יכול להיכנס למשנה, כלומר לתת לוגיקה למשתנה עם setter ו-getter .
בפייתון המשתנים לא פרטיים, ולכאורה תמיד יהיה ניתן לשנות משתנה מסוים בצורה לא מבוקרת.
בשביל זה הוסיפו שני קשטנים מיוחדים אחד משמש כ-getter, והשני כ- setter,
נדגים: ניקח למשל מלבן, למלבן יש משתנה שמגדיר את הגובה ומשתנה שמגדיר את האורך שלו.
נניח שאנחנו מנסים לצייר את המלבן אבל החלטנו לנסות לאתגר את חוקי המתמטיקה הכנסנו בטעות ערך שלילי למשתנה 'גובה', ניתן לצייר מלבן עם גובה שלילי?
בשביל זה אנחנו צריכים setter ו –getter.
כדי להגדיר getter למשתנה ניצור מתודה עם שם המשתנה (בלי קו תחתון) ונוסיף לה את הקשטן `@property` .
כדי להגדיר אותו כ- setter נדאג שהיא תקבל ערך מלבד הפרמר self ונוסיף לה קשטן עם שם המשתנה, נקודה, ו- setter
למשל בגובה: `@height.setter`
יש לשים לב ששם המשתנה ושם המתודה המקושטת אמורים להיות שונים, המתודה אמורה להיקרא כשם המשתנה (לו לא היה 'לא ציבורי'), והמשתנה עצמו אמור להיחשב כ-'לא ציבורי' כלומר נקרא לו עם קו תחתון אחד בתחילת שמו.
אם לא נגדיר שמות שונים תיזרק שגיאה.
ושוב אין כאן כימוס מלא באמת- עדיין ניתן לגשת למשתנה הפרטי, אבל לפחות דאגנו שתהיה אלטרנטיבה לשינוי האובייקט עם setter ו-getter וכך המתכנת האחר ידע איך אמורים לשנות את האובייקט למרות שהוא 'לא ציבורי':

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

    @property
    def height(self):
        return self._height
    @height.setter
    def height(self, other):
        if other > 0:
            self._height = other
        else: raise Exception("Invalid input- negative value for height")
    @property
    def width(self):
        return self._width
    @width.setter
    def width(self, other):
        if other > 0 :
            self._width = other
        else: raise Exception("Invalid input- negative value for width")
```

חוץ מזה קשטני setter ו- getter יכולים לעזור גם כדי להוסיף אובייקטים חדשים הבנויים מאובייקטים אחרים מבלי להגדיר אותם.
ניקח את הדוגמא הבאה, נניח יש לנו את המחלקה 'אישיות' ויש לה את שדות: שם פרטי, שם משפחה, ושם מלא:

```
class Person:
    def __init__(self, first_name , last_name):
        self.first_name= first_name
        self.last_name = last_name
        self.full_name = f"{self.first_name} {self.last_name}"
```

תמי ג'אוונסקי היא בחורה נחמדה שלא מזמן התחתנה עם תום פייתונוביץ (האיש והאגדה), היא אפילו שינתה את שם משפחתה במחלקה 'אישיות' , אבל לצער כולם המחלקה לא זיהתה את השינוי בכל שאר השדות:

```
tammi = Person("Tammi", "Javansky")
print(tammi.full_name)
tammi.last_name = "Pythonovitch"
print(tammi.full_name) # => should be Tammi Pythonovitch
```

```
Tammi Javansky
Tammi Javansky
```

תמי ידועה בהיותה ישראלית מין המניין וכזאת היא לא תפסיד הזדמנות להתלונן.
אז איך נוכל לשנות את המחלקה כך שיהיה נח למשתמשים כמו תמי להשתמש בה?
כמובן שניחשתם נכון, עם properties:

```
class Person:
    def __init__(self, first_name , last_name):
```

```
        self.first_name= first_name
        self.last_name = last_name
    @property
    def full_name(self):
        return f"{self.first_name} {self.last_name}"
```

וכעת אנשים כמו תמי יוכלו להשתמש במחלקה בצורה נוחה :

```
tammi = Person("Tammi", "Javansky")
print(tammi.full_name)
tammi.last_name = "Pythonovitch"
print(tammi.full_name) # => should be Tammi Pythonovitch
```

```
Tammi Javansky
Tammi Pythonovitch
```

ממשקים ומחלקה אבסטרקטית-

ממשק ומחלקות אבסטרקטיות משמשים כדי לתת פיגום עבור מחלקות עתידיות- מספקים מבנה שעל פיו המחלקות אמורות לעמוד.
ממשקים לרוב הם יותר מבנים ריקים שניתן לרשת מהם, וברוב השפות אין הגבלה על מספר הממשקים שמהם ניתן לרשת.
מחלקות אבסטרקטיות הן מחלקות שלפחות אחת המתודות שלה לא ממומשת בכלל.
מהמחלקה האבסטרקטית ניתן לרשת אך לא להשתמש במתודה הלא ממומשת, כלומר אם ירשת אתה חייב לדרוס את המתודה.
מחלקה אבסטרקטית יכולה להכיל מתודות ממומשות לפעמים, ובהרבה שפות תכנות ניתן לרשת רק ממחלקה אחת.
ממשקים בפייתון מתנהגים אחרת מממשקים בשפות תכנות אחרות.
בפייתון ההתייחסות לממשקים היא כמו למחלקות אבסטרקטיות, כלומר חלק אם לא כל המתודות של המחלקה יהיו ללא מימוש.
ברמת העיקרון ניתן לממש מחלקות אבסטרקטיות ע"י שימוש במילה pass בכל אחת מהמתודות, אך עדיין נוכל להשתמש במתודות על אף שאינן אמורות להיות בנות קיימה:

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

shape = Shape()
shape.area()
```

אז למחלקה באמת אין מימוש למתודות, אבל היא עדיין לא מחלקה אבסטרקטית, אם היא הייתה אבסטרקטית היא הייתה אמורה לזרוק שגיאה כשמנסים לגשת למתודות הלא ממומשות שלה.
אז איך נפשט את המחלקה? ע"י מודול חיצוני וקשטן.
בפייתון יש ספריה שנקראת abc (abstract base classes) שמספקת תשתית לבניית מחלקת בסיס אבסטרקטית,
לספריה יש מחלקה או משתנה שקוראים לו ABC ובשביל להגדיר מחלקה כאבסטרקטית ראשית נצטרך "לרשת" מ-ABC,
ועבור כל מתודה לא ממומשת של המחלקה נשתמש בקשטן @abstractmethod:

```
import abc
class Shape(abc.ABC):
```

```
    @abc.abstractmethod
    def area(self):
        pass
shape = Shape()
        shape.area()
```

```
_____________________
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-22-3def8600697f> in <module>
      5 pass
      6
----> 7 shape = Shape()
      8 shape.area()
TypeError: Can't instantiate abstract class Shape with abstract methods area
```