STRATEGY DESIGN PATTERN

נניח יש לנו מחלקה של חיות ,והחיות כלב וציפור יורשות ממנה. כל חיה מתנהגת אחרת בהתאם למושג תנועה-התנועה העיקרית של ציפור, על אף שיש לה רגליים, היא בתעופה, והכלב לעומת זאת הולך על ארבע.
נניח אנחנו רוצים לממש את היכולת לנוע עבור כל אחת מהמחלקות שיורשות ממחלקת "חיות", אז אנחנו נכנסים לבעיה כי תנועה זה מושג אבסטרקטי מידי ולכל חיה יש את המימוש שלה לתנועה.
האפשרות הנאיבית היא לכתוב פונקציה כללית או אבסטרקטית למושג תנועה, ולתת לכל חיה לממש כראות ענייה את "תנועה".
עכשיו אם נסתכל על המאזן של מספר האפשרויות של חיות לנוע (בלי להיכנס לפינות של קומבינציות) נראה שלא קיימות יותר מידי אופציות: הליכה, שחיה, תעופה, אולי זחילה, אבל מספר החיות לעומת זאת יכול להגיע לכמה אלפים.
הכמות קוד שנצרך להוסיף עבור כל חיה שיורשת מהמחלקה "חיות" לא פרופורציונלי למספר האופציות לתנועה, וכמתכנתים המוטו שלנו הוא כמה שיותר בכמה שפחות מאמץ.
כאן בדיוק נכנסת תבנית עיצוב strategy:

תבנית עיצוב strategy היא תבנית התנהגותית שמאפשרת לאלגוריתם להתנהג בצורה שונה בזמן ריצה.
כל התנהגויות הממומשות הן מחלקות ,מתודות או פונקציות, והן נקראות אסטרטגיה (strategy).
החלק של הקוד שבוחר איזה אסטרטגיה לבחור נקרא מקשר (context).
האסטרטגיה בנויה על שני עקרונות חשובים:
- עקרון ה- פתוח\סגור: ישויות תוכנה(מחלקות, מתודות פונקציות וכו') אמורות להיות פתוחות להתרחבות, אך סגורות לשינויים.
- היפוך שליטה (inversion of control)- חלקים של תוכנית שנכתבו בהתאמה אישית(מתודות של מחלקת הבן) יקבלו את צורת שליטה שלהם ממסגרת כללית (מחלקת אב).

שני העקרונות האלה שימושיים במיוחד כשרוצים לעצב ממשק משותף (החלק הסגור בעיקרון הפתוח\סגור), אבל מאפשרים שינויים במימוש הפרטים (החלק הפתוח של העיקרון הראשון).
כל פעם שרוצים לתכנת מימוש חדש, מעבירים אותו לממשק משותף מבלי לשנות בקוד הממשק, ומחברים את הקוד של הלקוח לממשק ,כך הקוד של הלקוח מחובר בצורה רופפת, כלומר הוא מחובר עם אבסטרקציה(הממשק) ולא עם הקוד הממומש.

מימוש של תבנית עיצוב strategy-
אם היינו צריכים לממש תבנית strategy בג'אווה למשל ,היינו בונים מחלקה אבסטרקטית 'חיה' שיש לה תכונה שקוראים לה 'תנועה' שהיא אובייקט מטיפוס ממשק תנועה, ועבור כל סוג של תנועה מסוימת היינו בונים מחלקה חדשה שיורשת מהממשק תנועה וממשת אותו בהתאם לצרכים שלה.
ואז במחלקות שיורשות מהמחלקה 'חיה', היינו מעבירים בבנאי גם את סוג התנועה שמתאימה לחיה. אם נרצה נוסיף גם יכולת לשנות את התנועה של החיה אם יש חיה מסוימת שנעה שלא כנורמה, למשל הכלב קריפטו.
לכאורה גם בפייתון אפשר לממש את המחלקה בצורה דומה:

```
import abc
class Animal:
    _name = None
    _movement = None

    def move(self):
        self._movement.move()

    def change_movement(self,other_movement):
        self._movement = other_movement
```

```
class Movment(abc.ABC):
    @abc.abstractmethod
    def move(self):
        pass
class Flying(Movment):
    def move(self):
        print("I'm flying")

class Walking(Movment):
    def move(self):
        print("I'm walking")

class Dog(Animal):
    def __init__(self,name):
        self._name = name
        self._movement = Walking()

    def bark(self):
        print("waff!")

class Bird(Animal):
    def __init__(self,name):
        self._name = name
        self._movement = Flying()

    def chirp(self):
        print("chiff chiff")


rexi = Dog("Rexi")
tweety = Bird("Rexi")
kripto = Dog("Kripto")
kripto.change_movement(Flying())
rexi.move()
tweety.move()
kripto.move()
```

```
I'm walking
I'm flying
I'm flying
```

הפתרון לעיל הוא לגיטימי לחלוטין, אבל לא ממש פייתניסטי.
כי בפייתון אין הגדרה ממש לטיפוס המחלקה של המשתנה אז לא ניתן להסיק מכך שהמשתנה Movement צריך להיות דווקא מטיפוס המחלקה Movement , ויתרה מזאת movement עצמה היא מחלקה שאפשר לשנות אותה בזמן אמת ולא ממש צריך להרבות בירושה בשביל זה.
לכן נשקול אסטרטגיה אחרת לתבנית אסטרטגיה: ניצור מחלקת אסטרטגיה אחת, ונחליף את המתודה הרלוונטית שלה כל פעם לפי המקשר:

```
class Animal:
    def __init__(self, name ,movment = None):
        self._name = name
        if movement is not None:
            self.movement = movment
    def movemen(self):
        print("I'm moving")
```

כל זה יפה והכל, אבל הפונקציה שאנחנו מספקים לא בטוח מכילה את self, ויכול להיות שהיא תחשב כפונקציית מחלקה (classmethod), אבל גם לזה יש פתרון עם המודול הזכור לטוב types:

```
import types
class Animal:
    def __init__(self, name ,movment = None):
        self._name = name
        if movement is not None:
            self.movement = types.MethodType(movment,self)

    def movemen(self):
        print("I'm moving")
```

עכשיו נגדיר את הפונקציות שנשלח כארגומנט:

```
def flying():
    print("I'm flying")

def walking():
    print("I'm walking")


class Dog(Animal):
    def __init__(self,name):
        self.name = name
        self.movement = walking
class Bird(Animal):
    def __init__(self,name):
        self.name = name
        self.movement = flying


rexi = Dog("Rexi")
tweety = Bird("Rexi")
kripto = Dog("Kripto")
kripto.movement = flying
rexi.movement()
tweety.movement()
kripto.movement()
```

עוד גישה מעניינת בנוגע לתבנית strategy, ודומה מאוד לצורת החשיבה הקודמת, היא מימוש תבנית strategy מתוך פונקציה עם מילון, כלומר ניצור פונקציה שיש בה מילון עם שמות הפונקציה כמפתח ותוכן הפונקציה כערך ונבחר מתוך הפונקציה את האופרציה המתאימה לנו.
למשל:

```
def calculate(name_of_operartion, *nums):
    def multiplication(*nums):
        mult= 1
        for num in nums:
            mult*=num
        return mult

    def addition(*nums):
        addi = 0
        for num in nums:
            addi+=num
        return addi
    operations = {
        'mul':multiplication,
        'sum':addition }
    return operations[name_of_operartion](*nums)



print(calculate('mul', 1,2,3,4))
print(calculate('sum', 1,2,3,4))
```

```
24
10
```