תכנות מונחה עצמים (מבוא)

פייתׂון היא שפה קלאסית ללימוד הנושא. היא מנסה לפשט ככל האפשר את המכניזם והתחביר של oop, ובכך מנגישה אותו גם למתחילים בתכנות. כל הרעיון של תכנות מסוג זה בפייתׂון ניתן לסיכום בביטוי:

object.attribute

הביטוי הזה מתחיל חיפוש בתוך עץ של עצמים מקושרים, עד למציאת המופע הראשון של: attribute . עץ החיפוש מתחיל מהעצם עצמו, המחלקה של העצם וממשיך מיתר המחלקות מהן ירשה מחלקת העצם (משמאל לימין במחלקות מהן יורשים - יש הורשה מרובת מחלקות).

כשנדבר על ההורשה, זה יהיה ברור יותר, אבל בפייתׂון אומרים שהמופע (instance) שנוצר מהמחלקה ׳יורש׳ תכונות מהמחלקה, והמחלקה יורשת ממחלקות אחרות שמעליה בהיראַרכיה.

בגלל שלא צריכים בפיתׂון להגדיר משתנים לפני השימוש בהם, ההגדרה של מחלקה ניראית קצת שונה משפות אחרות בהן יש משתנים (instance variables) ופעולות (methods). עוד שוני בולט נעוץ בעובדה שבפייתׂון לא קיים מצביע אוטומטי לעצם שיוצרים ממחלקה, אלא צריכים תמיד לדאוג למשתנה שיהווה אינדיקציה שאנו מתיחסים לעצם הנוכחי (כמו this ב- C#)

יצירת העצם נעשית על ידי מתן שם המחלקה עם פרמטרים מתאימים (כמו קריאה לפונקציה)

כיוון שאתם כבר מנוסים בתכנות מונחה עצמים, נתחיל בדוגמה מיידית, שתבהיר כמה פשוט זה יכול להיות. לאחר הדוגמה, נלך אחורה ובצורה מסודרת נעבור על פרטים שמרכיבים את התכנות מונחה עצמים בשפה. זו תהיה מחלקה שעוסקת בקבצי טקסט.

```
class Textfile:
   ntfiles=0                          # counts how many text file objects created
   def __init__(self, fname):
      Textfile.ntfiles +=1            # like static variable
      self.name = fname
      self.fh=open(fname)
      self.lines=self.fh.readlines()
      self.nlines=len(self.lines)     # count lines in file
      self.nwords=0
      self.wordcount()
  def __del__(self):
     self.fh.close()
  def wordcount(self)
      " find the number of words in the file"
      for ln in self.lines:
         words=ln.split()             # internal variable to aid with clarity
         self.nwords +=len(words)
  def grep(self,target):              # printing lines with 'target' string in them
     for ln in self.lines:
        if ln.find(target) >=0:
           print(ln)
  def main():
     a=Textfile('test1.txt')
     b=Textfile('test2.txt')
     print("The number of text files open is: ", textfile.ntfiles)
     print("information about the files (name,  lines,  words) : ")
     for f in [a,b]:
        print (f.name, f.nlines, f.nwords)
     a.grep('example')
  main()
```

המילה self חוזרת על עצמה רבות. זה מקביל ל- this של C++ או C# (מצביע למופע הנוכחי, או האובייקט שנוצר מהמחלקה), אבל בניגוד להן, בפייתׂון חייבים להגדירו כפרמטר הראשון (או יחיד) של הבונה ושל כל פעולה (method) אחרת, וגם להשתמש בו בתוך הבונה ובתוך פעולות של המחלקה (אחרת יווצר משתנה מקומי). אין שום קסם במילה:self, זהו רק הרגל שהשתרש ואפשר להמציא כל שם חוקי אחר.
אם עדיין לא פיענחתם, הבונה תמיד ניקרא: __init__ (שני קוים תחתונים לפני ואחרי המילה init). אגב, **לא חייב להיות בונה במחלקה של פייתׂון** , אבל זה רעיון טוב לכתוב אותו כדי לוודא שאם פעולה אחרת משתמשת במשתנה שתמיד אמור להיות בעל ערך (בכל מופע), לא תיווצר שגיאה. המשתנים בעלי הקידומת .self הם, מה שניקרא: משתני מופע (instance variables), מהו, אם כן , המשתנה ברמת המחלקה ntfiles?
ניחשתם נכונה (מקווה), זה מקביל למשתנה סטטי בשפות אחרות, כלומר משתנה של המחלקה עצמה ולא של כל אובייקט שנוצר מהמחלקה. במיקרה דנן הוא סופר את מספר הקבצים שפתחנו בעזרת המחלקה, אז מן הראוי שיהיה כזה. זהו משתנה שקיים גם לפני שיצרנו אובייקט כלשהו מהמחלקה.
המשתנה words בתוך הבונה, הוא משתנה פנימי ולוקאלי של הבונה שמופיע רק שם (משתנה עזר עבור חישוב פנימי, אפשר גם היה לוותר עליו ולכתוב: (()ln.split)len=+ self.nwords). הוספתי אותו להדגים שהוא שונה, כי לעיתים יותר ברור להשתמש במשתני עזר.
שימו לב לפשטות, כל הפעולות (methods) מוגדרת ללא סוג החזר, על ידי def ועם אותו עימוד בתוך בלוק של המחלקה (שמסתיימת כאשר מגדירים את ()main. כמו כן, בתוך ה- ()main יכולנו להתייחס למשתני מופע על ידי קידומת של המופע, נקודה, משתנה (כמו למשל: a.name) כמו היו מוגדרים עם גישת: public.
אין כרגע תמיכה בפייתׂון למשתנים או מאפיינים פרטיים של מחלקה, אולם ישנן דרכים יצירתיות ליישום חלקי של תבנית זו. זה חומר שהוא כרגע מעבר למה שתכננתי ללמד, אז אם יתאפשר בהמשך ותהיו מעוניינים, אוכל להוסיף עוד בנושא.
בנוסף לבונה, בפייתׂון ישנו גם הורס (destructor). והוא ניראה כך: __del__ וגם מקבל את self כפרמטר. הוא מופעל אוטומטית בזמן איסוף זבל (כאשר האובייקט יוצא מטווח ההכרה שלו והשטח עבורו מנוקה). בדוגמה הנ״ל ההורס מכיל פקודת סגירה של הקובץ, דבר שהוא תמיד חיוני למערכת ההפעלה.

הסבר על משתנים שאפשר במידה מסוימת להגן עליהם כפרטיים כדי למנוע שגיאות, אך לא למנוע שימוש זדוני. אם נגדיר בתוך מחלקה משתנים בעלי קידומת של 2 קוים תחתונים וננסה להתייחס אליהם מחוץ להגדרת המחלקה, פייתׂון לא תאפשר זאת. למשתנה שמוגדר כך, פייתׂון מצמידה קו תחתון ואת שם המחלקה בה הוגדר. לדוגמה אם בדוגמה של המחלקה Textfile היינו קוראים לשם הקובץ: __name, הפקודה: print (f.__name, f.nlines, f.nwords) הייתה ניכשלת. במקום זה היינו צריכים לכתוב:
print (f._Textfile__name, f.nlines, f.nwords)

זה די מקובל להשתמש בקו תחתון יחיד לפני שמות משתנים ומתודות שרוצים שיהיו פרטיים בתוך המחלקה ואינם חלק מה- API של המחלקה.

הורשה
קיימת מחלקה בשם a, וכעת נירצה ליצור מחלקה בשם b, שתירש את a.
class b(a)
פייתׂון גם תומך בהורשה רב-מחלקתית (כמו C++), אם c הייתה יורשת מ-a ו- b, היינו מציינים:
class c(a,b) - והסדר בתוך הסוגריים חשוב לעץ החיפוש. כאשר הבונה של המחלקה היורשת (derived class) מופעל, הבונה של המחלקה המורישה (base-class) אינו ניקרא בצורה אוטומטית, ואם רוצים להפעילו, צריכים לקרוא לו מפורשות. נמשיך עם דוגמת ההורשה:

```
class b(a):
   def __init__(self, xinit)  # constructor for class b
      self.x=xinit
      a.__init__(self)        # call base class's constructor
```

כיוון שבפייתׂון אין הגדרות של סוג משתנים, תהליך ההורשה יכול להיות מתואר כדלקמן: כאשר יוצרים עצמים בעזרת מחלקה, קיימים מרחבי שמות (namespaces) שקשורים ביניהם גם של העצמים וגם של המחלקה. לדוגמה:

```
class FirstClass:                # Define a class object
   def setdata(self, value):     # Define class methods    זוכרים: לא חייב להיות בונה
      self.data = value          # self is the instance
   def display(self):
      print(self.data)           # self.data: per instance
```

כעת ניצור עצמים:

```
x = FirstClass() # Make two instances, each is a new namespace
y = FirstClass()
```

ל- x ול-y ישנה גישה למאפיינים של המחלקה: FirstClass, וניתן לאמר ש-x הוא: FirstClass ו-y הוא: FirstClass. שני אובייקטים אלה מתחילים ריקים, אולם הם קשורים למחלקה ממנה נוצרו. אם ניתן ערך מסוים למאפיינים שנימצאים במחלקה, פייתׂון יביא את שם המאפיין מהמחלקה, ע״י חיפוש בעץ ההורשה, למשל:

```
x.setdata("King Arthur") # Call methods: self is x
y.setdata(3.14159)       # Runs: FirstClass.setdata(y, 3.14159)
```

לא ל-x ולא ל-y ישנו מאפיין שניקרא: setdata משלו, אז כדי למצוא אותו, פייתׂון צריך לחפש את הקשר למחלקה ממנה נוצרו. זה קורה בזמן הביצוע של הפקודות. וכעת אם נירצה להפעיל את הפעולה display כך:

```
x.display()  —-> King Arthur       # self.data differs in each instance
y.display()  —-> 3.14159
```

שימו לב שבכל אובייקט שמרנו נתון מסוג שונה וזה אומר שמשתנה המופע: data, אינו קיים מבלי שהפעלנו את הפעולה: setdata. למרות שהבונה ניקרא בזמן יצירת האובייקטים. אם היינו מנסים לקרוא ל- display לפני שקראנו ל- setdata היינו מקבלים שגיאה. ולכן יש קשר חזק בין הגדרת המחלקה והגדרת העצמים ופייתׂון תמיד ניגש למחלקה כדי לקבוע כיצד ליישם את הפעולה על העצם עצמו. זהו מודל מאד דינמי. אפשר גם לשנות את משתני המופע מחוץ למחלקה , למשל: x.data='new value' - כמו שכבר ציינו המשתנים נגישים בכל מקום (public). המודל אפילו יותר דינמי מזה, אפשר גם לתת בתוכנית פקודה כמו:
x.anothername =“spam”

כלומר ליצור מאפיין חדש בתוך העצם עצמו. בד״כ בתוך המחלקה על ידי השמה למשתנים עם קידומת self. יוצרים מאפיינים, אבל התוכנית כאמור יכולה לשנות מאפיינים עבור כל עצם בניפרד. כעת נמשיך עם הדוגמה לייצר מחלקה שהיא תת-מחלקה של: FirstClass.

```
class SecondClass(FirstClass):
   def display(self):
      print("current value = ", self.data)
```

כאן הגדרנו מחדש את הפעולה: display , שכבר הוגדרה במחלקת הבסיס (FirstClass) להדפיס בפורמט שונה. כלומר ביצענו override (או overload). לעומת זאת לא נגענו בפעולה: setdata ולכן היא זמינה לשימוש בעצמים שניצור מ- SecondClass. לדוגמה:

```
z = SecondClass()
z.setdata(42) # Finds setdata in FirstClass
z.display()   # Finds overridden method in SecondClass and prints: Current value = “42"
```

השימוש במחלקות עם הורשה הינו גמיש הרבה יותר מפונקציות, כי אפשר תמיד ליצור תתי-מחלקות עם ייחוד שיבדילן ממחלקת הבסיס ויתן לנו גם גמישות מקסימלית למטרה מסוימת וגם שימוש חוזר בקוד שכבר ניבדק ודובג.

העמסת פעולות (operator overloading)

זהו מתן אפשרות לעצמים (שיצרנו בעזרת מחלקה) להגיב לפעולות שעובדות על טיפוסי משתנים מובנים בשפה. התחביר של העמסת פעולות בפייתון מכיל הגדרה של שם הפעולה מוקף בשני קוים תחתונים מימין ומשמאל. למשל כדי להעמיס את פעולת ה + (חיבור), נגדיר: __add__
אחת מהפעולות האלה כבר הכנסנו לפעולה בבונה של מחלקה שקראנו לו: __init__.
נמשיך כעת את הדוגמה הנ"ל, וניצור מחלקה חדשה, ThirdClass שתירש מ- SecondClass, כדי להדגים העמסת פעולות של חיבור (__add__) ושל הפיכת אובייקט למחרוזת בזמן ההדפסה (__str__), כלומר זה כמו ההפעלה של הפונקציה: ()str שהיא חלק מהפונקציות שהן builtin בשפה. לנוחיות קיבצתי כאן גם את המחלקות הקודמות שהגדרנו:

```
class FirstClass:
    def setdata(self, value):
        self.data = value
    def display(self):
        print(self.data)
class SecondClass(FirstClass):
    def display(self):
        print("current value = ", self.data)

class ThirdClass(SecondClass):
    def __init__(self, value):
        self.data = value
    def __add__(self, other):
        return ThirdClass(self.data + other)
    def __str__(self):
        return '[ThirdClass: %s]' % self.data
    def mul(self, other):
        self.data *= other
        print(self.data)
#-----------------------------------------------------
def main():
    a=ThirdClass('abc')
    a.display()
    print(a)
    b=a+'xyz'
    b.display()
    print(b)
    a.mul(3)
    print(a)
main()
```

תוצאה:

```
current value = abc
[Thirdclass: abc]
current value = abcxyz
[Thirdclass: abcxyz]
abcabcabc
[Thirdclass: abcabcabc]
```

<u>הסבר:</u> הפקודה a.display() מפעילה את הפעולה : display שהוגדרה מחדש ב- SecondClass (שימו לב שב- FirstClass זוהי הדפסה רק של הערך), ולכן מדפיס: current value = abc .
הפקודה print(a) היא פקודת ההדפסה הרגילה בשפה, אבל העצם a בהיותו עצם מסוג ThirdClass עובר טרנספורמציה ברגע שהוא מודפס, על פי הפעולה __str__ ולכן הוא מודפס כ: [Thirdclass: abc] - זה מראה לנו שהעמסה בעזרת __str__ היא בעצם יצירת הדפסה (print) של האובייקט, כלומר הדפסה של

אובייקט פירושו הפעלת __str__ עליו. אנו כמובן יכולים להחזיר ערך כרצוננו מ __str__ בעזרת return וזה מ שיודפס.

הפקודה: b=a+'xyz' מפעילה את __add__ שמחזיר שירשור של: a עם הפרמטר other ('xyz') ויוצרת עצם חדש מסוג: ThirdClass שניקרא: b. (כעת ברורות הפקודות: b.dislay() ו- print(b) - לפחות כך אני מקווה) ופקודה: a.mul(3) היא כבר פשוטה להבנה. mul לא יוצרת עצם חדש,אלא משנה את העצם שאליו היא מוצמדת (a)

ניראה דוגמה שתמחיש את החיפוש בהיראכיה בין העצם והמחלקה:

```
class rec: pass
```

הגדרנו מחלקה ריקה, ללא משתנים וללא פעולות בשם: rec. בתוכנית נצמיד מאפיינים וערכים:

```
rec.name='Bob'
rec.age=20
```

כעת אפשר להתיחס למאפיינים, למשל: print(rec.name) ידפיס: Bob
שימו לב שעדיין לא יצרנו שום עצם מסוג: rec. כלומר, גם מחלקה בפיית'ון היא עצם. כעת ניצור עצמים מסוג rec

```
x=rec()
y=rec()
print(x.name, y.name) ——> Bob Bob
```

העצמים שיצרנו הם למעשה ריקים, אבל ירשו למעשה את המאפיינים של המחלקה rec כאשר התיחסנו אליהם (עשו חיפוש במעלה עץ). אם כעת ניתן ערכים לעצמים x ו-y (למופעים של rec), ניראה מה קרה:

```
x.name='Sue Ellen'
print(rec.name, x.name, y.name) —-> Bob Sue Ellen Bob
```

הערך עבור name של y עדיין מצוי רק ב-rec, אבל עבור y ישנו כבר ערך חדש ברמה נמוכה יותר (רמת המופע)

<u>משתנים ופעולות (מתודות) סטטיים</u>

משתנה ברמת המחלקה מוגדר מחוץ למתודות, וכדי להשתמש בו צריכים את שם המחלקה. מתודה סטאטית מוגדרת ללא self.
דוגמא:

```
class Person:
   population=0                    #static variable
   def __init__(self,name, age):
      self.name=name
      self.age=age
      Person.population +=1
  def totaPopulation():            #static method
      print('There are {0} people in this world'.format(Person.population))

p1=Person('Jenny', 30)
print(Person.totalPopulation())
```