

תכנות מונחה עצמים (מבוא)

אם יש צורך לריענון או למי שלא התנסה בתכנות מונחה עצמים, אוכל לתאר זאת בקצרה.

(oop_intro)

פייתון 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 בפייתון 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 בפייתון אומרים שהמופע (`instance`) שנוצר מהמחלקה "יורש" תכונות מהמחלקה, והמחלקה יורשת ממחלקות אחרות שמעליה בהירארכיה.

בגלל שלא צריכים בפייתון 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 שבפייתון 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) - אין צורך לציין `new`.

גם אם לא התנסתם 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.

נגדיר מחלקה בשם `Textfile` שעובדת עם קבצי טקסט.

יצירת אובייקט מהמחלקה תתבצע על ידי מתן שם קובץ על הדיסק.

תכנות המחלקה: שם קובץ פיזי, שם קובץ לוגי, מחרוזת שתכיל את כל שורות הקובץ, מספר שורות הקובץ, מספר המילים בקובץ.

בנוסף נישמור את מספר הקבצים הפתוחים בכל רגע נתון.

מלבד הבניה והההרסה של האובייקט תהיינה 2 מתודות, אחת שתספור מילים והשניה שתחפש מילה בקובץ.

```
class Textfile:
    nfiles=0                # counts how many text file objects created
    def __init__(self, fname):
        Textfile.nfiles += 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.nfiles)
    print("information about the files (name, lines, words) : ")
    for f in [a,b]:
        print (f.name, f.nlines, f.nwords)
    a.grep('example')
    b.grep('example')
```

המילה self חוזרת על עצמה רבות. זה מקביל ל- this של C++ או C# (מצביע למופע הנוכחי, או האובייקט שנוצר מהמחלקה), אבל בניגוד להן, בפייתון חייבים להגדירו כפרמטר הראשון (או יחיד) של הבונה ושל כל פעולה (method) אחרת, וגם להשתמש בו בתוך הבונה ובתוך פעולות של המחלקה (אחרת יוצר 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 קווים תחתונים לפני ואחרי המילה `init` שני הקווים התחתונים נקראים לפעמים בחיבה `dunder`). אגב, **לא חייב להיות בונה במחלקה של פייתון**, 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 דן 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: `self.nwords += len()` `ln.split()`). 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, תכונה (כמו למשל: `a.name`) כמו היו מוגדרים עם גישת: `public`. אין כרגע תמיכה בפייתון למשתנים או מאפיינים פרטיים של מחלקה, אולם ישנן דרכים יצירתיות ליישום חלקי של תבנית זו. נדבר על זה בהמשך.

בנוסף לבונה, בפייתון ישנו גם הורס (`destructor`). והוא נראה כך: `__del__` (`dunder 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 קווים תחתונים וננסה להתייחס אליהם מחוץ להגדרת המחלקה, פייתון לא תאפשר זאת. למשתנה שמוגדר כך, פייתון 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.

כפי שכבר נוכחתם ישנן מתודות מיוחדות ששמן מוקף בקוים תחתונים, וכעת נראה עוד מאלה. יש כאלה שקוראים להן: magic methods

```
__repr__  
__str__
```

שתי מתודות אלה מיועדות להחזיר ייצוג של אובייקט כמחרוזת (למי שמכיר כמו ToString() ב-C# למשל). בדוגמה הנ"ל יכולנו להגדיר:

```
__repr__(self):  
    return 'file: ' + self.name + ' has ' + self.nlines + ' rows'
```

ואז אם היינו מנסים להדפיס אובייקט f1 שיצרנו בעזרת Textfile (נניח קובץ עם 100 שורות בשם temp.txt היינו מקבלים:

```
print (f1)
```

yields:

```
file: temp.txt has 100 rows
```

בדרך כלל אם מי שמשמש בהדפסת אובייקטים הוא מפתח תוכנה, יכתוב את __repr__ המתודה: __str__ בד"כ מיועדת להצגת האובייקט למשתמשים. אם שתיהן קיימות, פייתון תבצע את: __str__ כאשר נכתוב: print object

ישנן עוד הרבה מתודות מיוחדות שמשמשות ב-dunder. נראה עוד שתיים שתייצגנה את סוג הדברים שניתן לעשות איתן. זה למעשה שקול למה שנקרא 'העמסת אופרטורים' בשפות אחרות. נגדיר מהי הוספה של שני אובייקטים מסוג Textfile כהצמדה של שמותיהם הפיזיים ולזה להצמיד את סכום השורות בשניהם.

```
def __add__(self, other):  
    return self.name + other.name + str(self.nlines + other.nlines)
```

המתודה הבאה תעמיס את הפעולה: len כך שאם נבצע len על אובייקט מסוג Textfile נקבל את אורך שם הקובץ.

```
def __len__(self):  
    return len(self.name)
```

דרך אגב, גם כשמבצעים פעולות על מספרים פשוטים או מחרוזות בעצם המתודות עם ה-dunder ניכנסת לפעולה). לדוגמה:

```
'abc'.__len__() ==> 3
```

```
int.__add__(4,5) ==> 9
str.__add__('a','b') ==> 'ab'
```

אם ברצוננו להגדיר במחלקה מסוימת חיבור של עצמים מיוחד, למשל אם נחבר קבצים ונרצה לקבל את אורכם המשותף, נוכל להגדיר במפורש את: `__add__` בצורה הבאה:

```
def __add__(self, other):
    f1 = open(self.name, 'r')
    f1_cont = f1.read()
    f2 = open(other.name, 'r')
    f2_cont = f2.read()
    tot = len(f1_cont)+len(f2_cont)
    return tot
```

מקשתים (מעטרים) עבור תכונות (property decorators)

אפשר בפייתון להגדיר מתודות ולהשתמש בהן כמו בגישה לתכונות (בשפות אחרות **getters / setters**)

נראה דוגמה עם מחלקה פשוטה ביותר:

```
class Employee:
```

```
    def __init__(self, first, last):
        self.first = first
        self.last = last
        self.email = first + '.' + last + '@email.com'
```

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

```
# Use it
```

```
emp1 = Employee('John', 'Smith')
print (emp1.first)
print (emp1.email)
print (emp1.fullname())
```

```
# הדפסה
```

```
John
john.Smith@email.com
John Smith
```

מכיוון שבפייתון ניתן לגשת לתכונות גם מחוץ להגדרת המחלקה (אין את ההגדרה private), אפשר למשל אחרי יצירת העצם, לכתוב:
emp1.first = 'Jim'

ואז ההדפסה תיראה כך:

```
Jim  
john.Smith@email.com  
Jim Smith
```

למרות ששינינו את התכונה first, כיוון שיצרנו את העצם על ידי ה- __init__ וכתובת המייל נוצרה שם, היא לא השתנתה. אפשר היה כמובן ליצור מתודה עבור יצירת המייל, אבל אם כבר הפצנו את המחלקה לשימוש, יהיה קשה לשכתב את הקוד כי אז בכל מקום היינו צריכים לכתוב email() במקום רק email. במצב כזה היינו רוצים משהו כמו getter/setter

אפשר בפייתון לעשות גם וגם. להגדיר את email כמתודה ולהשתמש בו כתכונה כך:

```
@property  
def email(self):  
    return self.first+'.'+self.last + '@email.com'
```

כעת ההדפסה הנ"ל תכלול את השינוי מ John ל Jim גם במייל. שימו לב שבהדפסה כתוב: emp1.email ולא emp1.email()

ומה עם setter?

נוסיף @property גם לפני המתודה: fullname. נניח שנרצה לשנות את fullname במהלך התוכנית ועל ידי כך לשנות גם את first ואת last. לשם כך נגדיר את התכונה עם setter בצורה הבאה:

```
@fullname.setter  
def fullname(self, name):  
    self.first, self.last = name.split()
```

וכעת אם נכתוב בתוכנית:

```
emp1.fullname = 'Michael Jackson'
```

ההדפסה תיתן:

```
Michael  
Michael.Jackson@email.com  
Michael Jackson
```