מתודות קסם

זוכרים את הדוגמא מהפרק על הכימוס בפייתון עם המחלקות A ו-B?
מתי שהפעלנו את הפונקציה dir() על המחלקה B היינו אמורים לראות רק מתודות ושדות של B שהגדרנו, אבל למרבה ההפתעה קיבלנו שם רשימה של תכונות שיש למחלקה שבכלל לא ידענו על קיומן:

```
print(dir(B))
```

```
['_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']
```

המכנה המשותף לכל התכונות האלה הוא שהן מתחילות בשני קווים תחתונים ומסתימות בשני קווים תחתונים.
אז מה אלו התכונות האלה?
המתודות האלו נקראות "מתודות קסם" או dunder methods (double underscores) והן אינן נועדו לשימוש ישיר אלא הן מתעוררות בצורה עקיפה.
למעשה יצא לנו כבר לראות כמה שדות ומתודות כאלו, למשל המתודה __init__() היא סוג של מתודת קסם שמאתחלת אובייקט מטיפוס מחלקה כלשהי, והיא מופעלת אוטומטית כאשר יוצרים אינסטנס חדש למחלקה.
לרוב משתמשים במתודות קסם כדי לדרוס אופרטורים, למשל כדי לדרוס את האופרטור '+' נדרוס את המתודה __add__(self,other), או עבור אופרטור '==' נדרוס את __eq__(self,other) .
אך למעשה יש עוד הרבה סוגים של מתודות קסם מלבד אופרטורים.
במסמך הבא נראה כמה מהשימושים הבולטים של מתודות הקסם :

1. אופרטורים

כפי שכבר ציינו קודם למתודות קסם יש את האפשרות לדרוס אופרטורים.
שלא כמו ב-++c או #c המתודות שמגדירות את סוג אופרטור אינן כייצוג שלו בפועל:

אופרטור	מתודה
+	__add__(self,other)
-	__sub__(self,other)
*	__mul__(self, other)
//	__floordiv__(self, other)
/	__truediv__(self, other)
%	__mod__(self, other)
**	__pow__(self, other[, modulo])
<	__lt__(self, other)
=>	__le__(self, other)
==	__eq__(self, other)
=!	__ne__(self, other)
=<	__ge__(self, other)
<	__gt__(self, other)
>>	__lshift__(self, other)
<<	__rshift__(self, other)
&	__and__(self, other)
|	__or__(self, other)
^	__xor__(self, other)

אם מוסיפים לחתימת המתודות את האות 'i' לפני השם האופרטור ניתן לדרוס את ה'syntactic sugar' אופרטור עם השמה, למשל עבור '+=' נדרוס את המתודה __iadd__(self,other), או בשביל '-=' את __

. isub__(self,other)
ובשביל הפעלת אופרטור מימין, למשל: obj+5 נוסיף את האות r לתחילת שם המתודה: __radd__(self,other)

נדגים עם המחלקה הבאה:

בנינו מחלקה שמקבלת ביטויים בעברית ומחזירה את הערך הגימטרי שלהם.
(מסתבר שכבר קיימת ספרייה שנקראת gematria , לכן שם המחלקה הוא Gymatria למקרה שהתקנתם אותה כבר).
במחלקה דרסנו שלושה אופרטורים- חיבור ,חיסור וכפל :

```
class Gymatria:

    _aleph_beth = None

    def __init__(self,expression:str) -> None:
        self._expr = expression
        self._expr_value = Gymatria.get_value(expression)

    @property
    def expr_value(self)->int:
        return self._expr_value
    @property
    def expr(self) -> str:
        return self._expr

    def __add__(self , other) -> int:
        return self.expr_value + other.expr_value

    def __sub__(self , other) -> int:
        return abs(self.expr_value - other.expr_value)

    def __mul__(self , other) -> int:
        return self.expr_value * other.expr_value

    def __repr__(self) -> str:
        return f"{self.expr} בגימטריה זה: {self.expr_value}"

    @classmethod
    def get_aleph_beth(cls):
        if cls._aleph_beth == None:
            cls._set_aleph_beth()
        return cls.aleph_beth

    @classmethod
    def get_value(cls,expression:str) -> int:
        aleph_beth = cls.get_aleph_beth()
        expr_value = 0
        for ot in expression:
            if ord('א') <= ord(ot) <= ord('ת'):
```

```
            expr_value += aleph_beth[ot]
        return expr_value

    @classmethod
    def ot_sofit(cls, ot: str)-> bool:
        cls.otiot_sofiot = ['ם','ן','ף','ך','ץ']
        if ot in cls.otiot_sofiot:
            return True
        return False

    @classmethod
    def _set_aleph_beth(cls) -> None:
        ot_num = ord('א')
        cls.aleph_beth={}
        val = 1
        for i in range(27):
            cls.aleph_beth[chr(ot_num+i)] = val
            if not cls.ot_sofit(chr(ot_num+i)):
                if 90 >= val >= 10:
                    val+=10
                elif val >= 100:
                    val+= 100
                else: val+=1
```

המחלקה שומרת כמשתנה סטטי מילון עם הערכים של כל אות באל"ף בי"ת העברית (באותיות הסופיות שוות למקבילות הלא סופיות שלהן במקרה זה), מקבלת ביטוי שומרות אותו ואת ערכו הגימטרי.
הרצנו את התוכנית על הביטויים הבאים וזה מה שקיבלנו:

```
aba = Gymatria('אבא')
aima = Gymatria('אמא')

print(aba)
print(aima)
print(f'אבא+אמא בגימטריה = {aima+aba}')
print(f'אבא-אמא בגימטריה = {aima-aba}')
print(f'אבא*אמא בגימטריה = {aima*aba}')
```

```
4 :אבא בגימטריה זה
42 :אמא בגימטריה זה
46 = אבא+אמא בגימטריה
38 = אבא-אמא בגימטריה
168 = אבא*אמא בגימטריה
```

2.אתחול והריסה של אובייקט-

בכל פעם שנוצר אינסטנס חדש של אובייקט שתי מתודות נקראות , האחת כבר יצא לנו להכיר והיא המתודה __init__, והשנייה היא מתודה שנקראת עוד לפני והיא __new__().
בשפות כמו ג'אווה c++ וכדו' אנחנו מכירים את המילה השמורה new כמגדירה לתוכנית להקצות זיכרון עבור אובייקט חדש מטיפוס המחלקה, בפייתון השימוש ב-new הוא קצת שונה, המתודה אמורה להיות מתודת מחלקה (class method), כלומר אמורה לקבל כפרמטר משתנה מסוג cls, ואמורה אוטומטית להחזיר את האובייקט בזמן האתחול.
במילים אחרות המתודה __new__ נקראת כדי ליצור את האינסטנס עצמו והמתודה __init__ כדי לאתחל אותו בערכים.
ברמת העיקרון אין בכלל צורך במתודה __init__, ניתן להגדיר ולהוסיף לאובייקט תכונות גם בלי שיהיו חלק מהגדרת המחלקה:

```
class Empty:
    pass
empty = Empty()
empty.something = "something"
print(empty.something)
```

```
something
```

אפילו יותר מזה ניתן להוסיף למחלקות קיימות מתודות חדשות בן אם לכל המחלקה בין אם רק לאובייקט ספציפי:

```
Gymatria.__iadd__= lambda self , other: Gymatria(self.expr +" "+ other.expr)
aba += aima
print(aba)
```

```
אבא אמא בגימטריה זה: 46
```

כשמשתמשים במתודה __init__ בעצם מאתחלים את השדות של האובייקט מרגע יצירתו, כך שמראש נקבל אותו עם תכונות מוכנות, אבל בשביל שהמתודה תיקרא אוטומטית היא צריכה לקבל אובייקט מטיפוס המחלקה (היא אמורה לקבל self) ומי שמספק לה את ה-self אמורה להיות המתודה __new__ שאמורה להחזיר אינסטנס מטיפוס המחלקה, אם __new__ לא תספק אינסטנס כזה לא תופעל המתודה __init__:

```
class AmIEmpty(Empty):
    def __new__(cls):
        print("__new__ method was executed")
        return None

    def __init__(self):
        print("__init__ method was executed")

    def a_method(self):
        print("a_method was executed")


am_i_empty = AmIEmpty()
```

```
__new__ method was executed
```

היינו מצפים שגם המתודה __init__ תופעל עם יצירת אובייקט חדש מטיפוס AmIEmpty אבל משום ש__new__ לא החזירה אינסטנס שעליו ניתן להוסיף משתנים לא יתקיים, ואפילו יותר מזה אם ננסה להפעיל את המתודה a_method של המחלקה תיזרק שגיאה:

```
am_i_empty.a_method()  # => AttributeError
```

```
_____________________
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-11-0bb7bf48eb32> in <module>
----> 1 am_i_empty.a_method() # => AttributeError
AttributeError: 'NoneType' object has no attribute 'a_method'
```

לעומת זאת אם __new__ כן תחזיר אובייקט מטיפוס המחלקה:

```
class AmIEmpty(Empty):
    def __new__(cls):
        print("__new__ method was executed")
        return super(Empty, cls).__new__(cls)

    def __init__(self):
        print("__init__ method was executed")

    def a_method(self):
        print("a_method was executed")


am_i_empty = AmIEmpty()
am_i_empty.a_method()
```

```
_________________________
__new__ method was executed
__init__ method was executed
a_method was executed
```

בדומה לc++, גם בפייתון יש מה שנקרא destructor שמוחק את התוכן של המשתנה מהמערכת עם הפונקציה del:

```
x = [1,2]
print(x)
del x
print(x) #=> NameError
```

```
___________________
[1, 2]

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-3-867c6f943e95> in <module>
         2 print(x)
         3 del x
   ----> 4 print(x) #=> NameError
NameError: name 'x' is not defined
```

זה עוזר בעיקר מתי שרוצים להגדיר שאין שימוש לאובייקט מסויים יותר בתוכנית, או כאשר רוצים למחוק מתודה או שדה של אובייקט\מחלקה:

```
del Gymatria.__repr__
print(aba)
print(aba.expr)
print(aba.expr_value)
```

```
<__main__.Gymatria object at 0x0000000005638668>
אבא אמא
 46
```

המתודה __repr__ מגדירה מחרוזת מייצגת לאובייקט, בהמשך נדבר עליה יותר ומה ההבדל בינה לבין המרה ל-str.

בניגוד למחשבה המקובלת המתודת קסם __del__() לא נקראת ע"י הפונקציה del, אלא ע"י הgarbage collector ברגע שהוא אוסף את כל המשתנים מסוג המחלקה, ועדיף להימנע ממימושה אלא אם אתם חייבים כדי להגדיר משהו לסוף התוכנית, או כדי לבחון מתי ה-grabage collector מוחק את האובייקט.

3. פונקציות אונריות על משתנים-

לפייתון יש כמה פונקציות אונריות בנויות מראש, למשל הפונקציה floor() מביאה את הערך התחתון של המשתנה, או הפונקציה abs() שמביאה את הערך המוחלט של האובייקט.
כשפונקציות אלה נקראות הן משתמשות במתודת קסם של המחלקה עם שם דומה.

מה היא עושה	מתודה
נקראת עם הפונק' abs()	__abs__(self)
אמורה להחזיר את הערך החיובי של המשתנה	__pos__(self)
אמורה להחזיר את הערך השלילי של המשתנה	__neg__(self)
ערך תחתון נקראת עם math.floor	__floor__(self)
ערך עליון נקראת עם math.ceil	__ceil__(self)
נקראת עם math.trunc	__trunc__(self)
נקראת עם הפונק' round()	__round__(self,n)
נקראת עם האופרטור ~	__invert__(self)

4. המרות (casting)

המרות לאובייקטים מטיפוס אחר עושים בדר"כ עם פונקציית המרה , למשל כדי להפוך מספר אי-רציונלי לאינטג'ר נשתמש בפונקציה int() על המספר. הפונקציה קוראת למתודת קסם __int__() של המחלקה float וכך יודעת איך להמיר את המספר.
לכל סוג של המרה יש פונקציית קסם עם שם דומה.
נחזור לדוגמא של הגימטריה ונוסיף לה את ההמרות הבאות-המרה למספר(int ו-float) והמרה למחרוזת:

```
Gymatria.__int__ = lambda self: self.expr_value
Gymatria.__str__ = lambda self: f"{self.expr} בגימטריה זה: {self.expr_value}"
Gymatria.__float__ = lambda self: float(self.expr_value)

print(str(aba))
print(int(aba))
print(float(aba))
```

```
אבא אמא בגימטריה זה: 46
46
46.0
```

__str__ ו- __repr__ -

בפייתון יש שתי פונקציות שלכאורה עושות אותו דבר- repr() ו- str().
המתודה __repr__ אמורה להחזיר למשתמש צורה ייצוגית של אובייקט כאשר הוא מנסה להכניס את האובייקט לאיזשהו קובץ output כגון הדפסה או לקובץ log.
המתודה __str__ אמורה להמיר את האובייקט לייצוג של מחרוזת, ולמעשה כשאין מימוש ל-__str__ אבל יש ל-__repr__ המתודה שממירה תתנהג כמו המתודה המייצגת, במילים אחרות __repr__=__str__ (אך לא להפך), אבל אם מימשנו את __str__ האובייקט שיוצג בהדפסה יהיה של המתודה הנ"ל , אך לא בכל מקרה:
נחזור לדוגמא עם הגימטרייה, מחקנו מקודם את המתודה __repr__ שלה, אבל הוספנו מתודה __str__ , בואו נחזיר את המתודה אך עם שינוי קטן מהמתודה __str__ ונבדוק האם בכל מצב הפונקציה הממירה תיקרא:

```
Gymatria.__repr__ = lambda self: f"{self.expr} == {self.expr_value}"
shalom = Gymatria("שלום")
print(shalom)
gymatria_list = [shalom,Gymatria("עולם")]
print(gymatria_list)
```

```
שלום בגימטריה זה: 376
[שלום == 376, עולם == 146]
```

מתי שהדפסנו רק מילה אחת היא הודפסה לפי המתודה __str__, אבל כשהדפסנו את האובייקט כחלק מאוסף הוא הודפס לפי הייצוג שלו במתודה __repr__.
המטרה של המתודה __repr__ היא להיות חד משמעית (unambiguous), ונשתמש בה כדי לזהות באיזה אובייקט משתמשים.
איך יודעים שאנחנו מספיק 'חד משמעיים'? אם אפשר להתמש בפונקציה eval() על המודה __repr__ ולקבל אובייקט מטיפוס המחלקה.
נחזור לדוגמא לעיל ונשנה את המתודה __repr__ בהתאם:

```
Gymatria.__repr__ = lambda self: f"Gymatria({self.expr!r})"
print(gymatria_list)
olam = eval(repr(gymatria_list[1]))
print(olam)
```

```
[Gymatria('שלום'), Gymatria('עולם')]
עולם בגימטריה זה: 146
```

שימו לב ששנינו את המתודה כך שתוכל לספק יצירת אובייקט גימטרייה חדש במידה שנעשה על המחרוזת איבלואציה.
תזכורת: הפונקציה eval() מקבלת מחרוזת וממירה אותו לקוד פייתון ממשי (עושה על המחרוזת 'איבלואציה').
הערה: הסימון !r הוא סימון פורמט מיוחד שמחזיר את המשתנה אחרי שהפעילו עליו repr(), היינו צריכים את זה כדי להתמודד עם מקרים שונים של מחרוזות, למשל אם היינו עושים אובייקט גימטרייה שקוראים לו 'צה"ל' אז אנחנו משתמשים ב- " ובמחרוזת של ' ' ,ואם היינו יוצרים אובייקט עם המילה " וכו' " היינו צריכים להשתמש במחרוזת של " " .

```
gymatria_list.append(Gymatria('צה"ל'))
gymatria_list.append(Gymatria("וכו'"))
```

```
print(gymatria_list)
```

```
[Gymatria('שלום'), Gymatria('עולם'), Gymatria('צה"ל'), Gymatria("וכו'")]
```

השימוש ב-__str__ הוא בעיקר לתת תצוגה יפה יותר של האובייקט עבור משתמשי הקצה, או עבור המרות של תוכן האובייקט למחרוזת.

5. ניהול גישה לתכונות-

בדוגמאות לעיל נתנו את האפשרות להוסיף תכונות חדשות למחלקה כחומר ביד היוצר, אבל מה אם לא נרצה שהמשתמש יוסיף תכונות, או שנרצה לקבוע כיצד יראו התכונות שאנחנו מחזירים אותם\ מוחקים אותם.
ניתן לעשות זאת ע"י המתודות __getattr__, __setattr__ , שמשמשות כעוד תחליף לgetter ו -setter:

```
class MutableDefault:
    ''' Saves any value is provited
        or initiates to "default" '''
    def __init__(self , *args , **kwargs):
        if args : self.attr1 = args
        else: self.attr1 = 'default'
        if kwargs: self.attr2 = kwargs
        else: self.attr2 = 'default'
    def __getattr__(self,name:str):
        try: self.__dict__[name]
        except: return None
    def __setattr__(self,name,value):
        if name == 'attr1' and self.attr1 == None:
            self.__dict__[name] = value
        elif name == 'attr2' and self.attr2 == None:
            self.__dict__[name] = value
        else: raise Exception(f"self.{name} already exists")

stam = MutableDefault(d = 1)
print(stam.attr1)
stam.all = lambda self:print(f"{self.attr1} , {self.attr2}") # => Exception
stam.attr1 = (1,2,3) #=> Exception
```

הפונקציה לא מאפשרת להוסיף אובייקטים או לשנות ערכים מעבר למה שנתנו לה בהתחלה.

6.ניהול משאבים (CONTEXT MANAGER)-

בהרבה שפות תכנות, השימוש במשאבים כמו קבצים או מסדי נתונים הוא מאוד נפוץ.
אך משאבים אלה מוגבלים לשימוש, לכן הבעיה המרכזית שלהם היא לוודא ששחררנו את המשאבים לאחר שימושם, אחרת עלולה להיווצר דליפה שיכולה להאט את המערכת או לגרום לקריסתה.
זה יכול להיות מאוד שימושי אילו היה לנו איזשהו מנגנון שידע לפתוח את המשאבים ולסגור אותם אוטומטית בסוף השימוש.
בפייתון כבר יצא לנו לראות מנגנון כזה והוא שימוש במילה השמורה with , אך עדיין לא ראינו איך הוא עובד בפועל.
ישנן שתי מתודות קסם מיוחדות בפייתון שמגדירות את המצב של האובייקט בכניסה אליו, והמצב ביציאה ממנו, והן __enter__ ו- __exit__ .
המתודה __enter__ מחזירה את המשאב שאמור להיות מנוהל, והמתודה __exit__ אמורה לנהל את סגירת המשאב ולהחזיר None.
המתודות האלו מופעלות בתחילת שימוש ב-(__enter__) with ובסיום הבלוק של ה-(__exit__) with:

```
class ContextManager():
    def __init__(self):
        print('init method called')
    def __enter__(self):
        print('enter method called')
        return self
    def __exit__(self, exc_type, exc_value, exc_traceback):
        print('exit method called')


with ContextManager() as manager:
    print('with statement block')
```

```
init method called
enter method called
with statement block
exit method called
```

שימו לב שקודם נוצר האובייקט ואח"כ המתודה __enter__ מחזירה אותו ל-with.
(הדוגמא לעיל מ- geeksforgeeks)

7. פעולות על קבוצות -

לכל פונקציה בנויה מראש בפייתון קיימת מתודת קסם שניתן לדרוס כדי להגדיר התנהגות בקריאה לאותה פונקציה.
גם לפונקציות ואופרטורים שפועלים על קבוצות ניתן להגדיר התנהגות, למשל עבור הפונקציה len() אפשר לממש את המתודה __len__() , ועבור האופרטור [] אפשר לממש את __getitem__ ו-את __setitem__ .

להמחשה בנינו מחלקה (חלקית) של רשימה מקושרת שיכולה לקבל אובייקטים מטיפוס Node :

```
from typing import Type
node = Type[Node]
class Node:
    def __init__(self,data = None, next = None ,prev = None):
        self.data = data
        self.next = next
        self.prev = prev
```

```
    def __repr__(self):
        return f"Node({self.data})"
```

למחלקה בנינו מתודה __getitem__ שמגדירה מה הערך המוחזר בהינתן אינדקס מסויים, מתודה __setitem__ שמגדירה עריכה של משתנה מסוים, ומתודה Insert שמכניסה לסוף הרשימה את ה-Node שהיא מקבלת כארגומנט, ומחזירה אובייקט מטיפוס LinkedList :

```
linkedList =Type[LinkedList]
class LinkedList:
    def __init__(self) ->None:
        self._head = None
        self._tail = Node(None)
        self._size = 0

    def __len__(self) ->int:
        return self._size

    def __getitem__(self, index:int) ->node:
        if index >= self._size: raise IndexError("invalid input, index out of range")
        temp = self._head
        for i in range(index):
            temp = temp.next
        return temp

    def __setitem__(self , index:int, value:object) ->None:
        #if not isintance(value , Node): raise ValueError("Not a Node type vlaue")
        self.__getitem__(index).data = value

    def insert(self, new_node :node , index :int = None) ->linkedList:
        if not isinstance(new_node , Node): raise ValueError("Not a Node type")
        if index == None:
            if self._head == None:
                self._head = new_node
                self._head.next = self._tail
                self._tail.prev = self._head
                self._head.prev = None
            else:
                (self._tail.prev).next, new_node.prev = new_node , self._tail.prev
                self._tail.prev , new_node.next =  new_node , self._tail
            self._size += 1
            return self

    def __str__(self) -> str:
        s = " "
        temp = self._head
        while temp != self._tail:
            if temp != self._tail.prev:
               s += f"{temp} -> "
            else: s += f"{temp}"
```

```
            temp = temp.next
        return s

    def __repr__(self) ->str:
        temp = self._head
        s = "LinkedList()"
        while temp != self._tail:
            if temp!= self._tail.prev:
                s="("+s
            s += f".insert({temp})"
            if temp!= self._tail.prev:
                s += ')'

            temp = temp.next
        return s
```

ובאמת אם ננסה להגיע לאינדקס מסוים או לשנות ערך תחת ההגבלות שהצבנו במתודה נראה שאכן הצלחנו:

```
linked_list = LinkedList()
# linked_list[0] = 3  => IndexError
# linked_list[len(linked_list)] => IndexError
for i in range(1,15,2):
    linked_list.insert(Node(i))
print(f"linked_list = {linked_list}")
print(f"linked_list[3] = {linked_list[3]}")
print(f"len(linked_list) = {len(linked_list)}")
linl = eval(repr(linked_list))
linl[0] = "Tom Pythonovitz"
print(linl)
```

```
linked_list = Node(1) -> Node(3) -> Node(5) -> Node(7) -> Node(9) -> Node(11) -> Node(13) linked_list[3]
= Node(7)
len(linked_list) = 7
Node(Tom Pythonovitz) -> Node(3) -> Node(5) -> Node(7) -> Node(9) -> Node(11) -> Node(13)
```

חוץ מהמתודות האלה ניתן להגדיר גם מחיקה של אובייקט מהמחרוזת עם __delitem__:

```
    def __delitem__(self , index):
        if index==0:
            self._head = self.__getitem__(index+1)
            self._head.prev = None
        else:
            prev_node = self.__getitem__(index-1)
            next_node = self.__getitem__(index+1)
            prev_node.next , next_node.prev = next_node ,prev_node
        self._size-=1
```

נוסיף המתודה למחלקה ונקבל:

```
print(linl)
del(linl[0])
print(linl)
```

```
_________________
Node(Tom Pythonovitz) -> Node(3) -> Node(5) -> Node(7) -> Node(9) -> Node(11) -> Node(13) Node(3) ->
Node(5) -> Node(7) -> Node(9) -> Node(11) -> Node(13)
```

חוץ מהמתודות הנ"ל יש גם מתודות קסם __iter__ שממירה את האובייקט לאובייקט מטיפוס איטרטור, אבל על כך בנושא אחר.

אם אתם רוצים לקרוא עוד בנושא מתודות קסם, אתם יכולים למצוא כאן .