# 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: 5+obj נוסיף את האות r לתחילת שם המתודה: \_\_radd\_\_(self,other)

נדגים עם המחלקה הבאה:

בנינו מחלקה שמקבלת ביטויים בעברית ומחזירה את הערך הגימטרי שלהם.  
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 = ['ץ','ך','ף','ן','ם']

        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}')

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

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 \_\_str\_\_=\_\_repr\_\_ (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 \_\_setattr\_\_, \_\_getattr\_\_ , 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-with (\_\_enter\_\_ ) ובסיום הבלוק של ה-with (\_\_exit\_\_):

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](https://www.geeksforgeeks.org/context-manager-in-python/))

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](https://rszalski.github.io/magicmethods/) .