ITERATORS AND GENERATORS

בסוף הפרק הקודם על מתודות קסם בפייתון התעסקנו בנושא של מתודות של אוספים והזכרנו את המתודת קסם __iter__ , אז מה זאת המתודה ואיך היא קשורה לאוספים?

בשפות תכנות בכלל בפייתון בפרט יש אובייקטים שנקראים אובייקטי iterable, והם כל אותם אובייקטים שניתן לבצע עליהם לולאת for, למשל אוספים כמו רשימות מילונים וכו'.
אך לא על כל האובייקטים ניתן לבצע לולאת for , אז איך נדע לאיזה אובייקט כן ניתן לבצע או יותר מדויק מה בדיוק הולך מאחורי הקלעים של לולאות for?
כשאנחנו מבצעים לולאת for each על אובייקט אנחנו בעצם ממירים את האובייקט לאובייקט אחר מטיפוס איטרטור, כן הפונקציה __iter__ היא בעצם סוג של מתודת קסם עבור המרה לאובייקט מטיפוס איטרטור.
אז מה זה בעצם האובייקט הזה איטרטור?
איטרטור הוא אובייקט ששומר מצב ראשוני (ולפעמים סופי), זוכר את המצב הנוכחי שלו בזמן איטרציה ויודע לחשב את המצב הבא אחריו.
מכאן שההבדל העיקרי בין אוסף לאיטרטור הוא שאוסף זוכר את כל המצבים שלו, ואיטרטור מכיר רק את המצב הנוכחי (והסופי אם יש) שלו, ויודע לחשב מה המצב הבא.
איך הוא עובד? לאיטרטור מאתחל לעצמו ערך התחלתי, ובאמצעות המתודה ()__next__ הוא יודע לחשב מה יהיה המצב הבא ומחזיר אותו, אם הוא שמר מצב סופי והמצב הבא עובר את המצב הסופי את תיזרק חריגה.
בפועל מה שקורא בלולאת for שהיא ממירה את האובייקט לאובייקט מטיפוס איטרטור ומפעילה עליו את הפונקציה __next__ עד שהיא מגיעה לחריגה:

```
my_list = ['start',' |',' |',' |',' ↓']
my_iterator = iter(my_list)
while True:
    try:
        item = next(my_iterator)
        print(item)
    except StopIteration:
        print('end')
        break
```

הדבר היעיל ביותר באיטרטורים הוא שבמקום לשמור את כל הערכים בזיכרון, אפשר פשוט לשמור ערך אחד ופונקציה לחישוב הערך הבא.
נראה דוגמא, נבנה אובייקט איטרטור שעובד בצורה דומה לפונקציה ()range הוא מקבל ערך התחלתי , ערך סופי, וכמה צעדים יש להתקדם:

```
class RangeIterator:
    def __init__(self,first,end,step=1):
        self.state = first
        self.end = end
        self.step = step

    def __iter__(self):
        return self


    def __next__(self):
```

```
        if self.state >= self.end : raise StopIteration
        res = self.state
        self.state+=self.step
        return res

for i in RangeIterator(0,10,2):
    print(i , end = ' ')
```

```
0 2 4 8
```

באוספים בדר"כ שומרים אובייקט מטיפוס איטרטור מתואם מראש, כך שמתי שממירים את המחלקה לאיטרטור מחזירים במתודות קסם __iter__ את אותו איטרטור מיוחד.

פונקציית GENERATOR –

חוץ מאיטרטורים יש דבר דומה שנקרא generators
generator היא פונקציה שמחזירה את המיקום בו היא נמצאת בבלוק שלה, כלומר זאת פונקציה שיכולה להחזיר כל פעם ערך אחר, תלוי מתי קוראים לפונקציה (או כמה פעמים קוראים לה):

```
def example_of_generator():
    yield "first"
    yield "second"
    yield "third"
example = example_of_generator()
print(next(example))
print(next(example))
print(next(example))
print(next(example))
```

```
first
second
third

---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-22-027d87ae8229> in <module>
      7 print(next(example))
      8 print(next(example))
----> 9 print(next(example))

StopIteration:
```

כפי שניתן לראות גם הפונקציה עוברת בין ערכים עם המתודה ()next כמו איטרטור.
אז איך בעצם עובדת הפונקציה?
כפי שאמרנו כבר בשיעורים הקודמים, בפייתון כל אובייקט הוא מחלקה אפילו פונקציות.
כשיוצרים אובייקט מטיפוס פונקציית ג'נראטור בעצם יוצרים אובייקט איטרטור רק בלי להגדיר לו מתודת __next__ או מתודת __iter__ , זה נעשה אוטומטית, כלומר ברגע שקוראים לפונקציה היא מחזירה לנו אובייקט מסוג generator ולא כמה ערכים כפי שניתן לחשוב:

```
my_example = example_of_generator()
print(f"example_of_generator is a {type(example_of_generator)}")
print(f"my_example is a {type(my_example)}")
```

```
example_of_generator is a <class 'function'>
my_example is a <class 'generator'>
```

```
help(my_example)
```

```
Help on generator object:
example_of_generator = class generator(object)
| Methods defined here:
| __del__(...)
|
| __getattribute__(self, name, /)
| Return getattr(self, name).
|
| __iter__(self, /)
| Implement iter(self).
|
| __next__(self, /)
| Implement next(self).
|
| __repr__(self, /)
| Return repr(self).
|
| close(...)
| close() -> raise GeneratorExit inside generator.

...
```

לרוב כשניתן לחשב ישירות את הערכים של המשתנים משתמשים בלולאת while בתוך הפונקציה כדי לחסוך בכתיבת קוד:

```
def fibonacci_generator(iterations):
    first = 0
    second = 1
    iter_num = 0
    while iter_num != iterations:
        yield first
        first , second = second , first+second
        iter_num+=1

num_of_iterations = 15
for fib in fibonacci_generator(num_of_iterations):
    print(fib,end = ' ')
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```