

סיכום מבוא למדעי המחשב

תוכן עניינים

2	אופרטורים מיוחדים
2	<i>print</i>
2	<i>input</i>
2	<i>max, min, abs</i>
2	<i>range</i>
3	<i>Casting</i>
3	<i>type</i>
3	<i>id</i>
3	אלגוריתם <i>GCD</i> (של אוקלידס)
4	<i>immutables vs mutables</i>
6	<i>Strings</i>
7	<i>Lists</i>
8	<i>dictionary</i>
9	<i>Set</i>
10	<i>Deep Copy vs Shallow Copy</i>
11	<i>Slicing</i>
12	<i>List Comprehension</i>
13	<i>Iterators</i>
14	<i>next</i>
15	<i>Generators</i>
16	<i>yield</i>
17	<i>lambda</i>
17	<i>map</i>
18	<i>Reduce</i>
18	<i>filter</i>
19	<i>Powerset</i>
19	<i>enumerate</i>
20	<i>nested functions</i>
21	<i>Exceptions</i>
23	<i>Binary Search</i>
24	<i>Selection Sort</i>
25	<i>Bubble Sort</i>
26	<i>Radix Sort</i>
27	<i>Merge Sort</i>
28	<i>Quick Sort</i>
29	זמני ריצה של שיטות נפוצות
30	<i>Linked Lists</i>
32	מימוש מחלקת <i>Linked List</i> עם אפשרות איטרציה
33	בנוס

אופרטורים מיוחדים:

חזקה - $a ** b = a^b$

חילוק ללא שארית - $17 // 5 = 3$

שארית חלוקה - $17 \% 5 = 2$

"חיבור" מחרוזות - `"hello" + "goodbye" = "hellogoodbye"`

"כפל" מחרוזות - `"hello " * 5 = "hello hello hello hello hello"`

print:

ל-`print` יש אפשרות לקבל 2 פרמטרים:

`end` – שינוי התו האחרון בהדפסה. כברירת מחדל הפונקציה מוסיפה `'\n'` (ירידת שורה).

`sep` – מפריד בין הערכים המודפסים.

דוגמה לשימוש בשניהם: `print(1,2,3,4,5, sep = ' 0000 ', end = ' &&&')`

1 0000 2 0000 3 0000 4 0000 5 &&& ידפיס:

input:

שיטה שמבקשת קלט מהמשתמש, מחזירה מחרוזת. חשוב לזכור שאם רוצים להשתמש בפונקציה

עבור מספרים נעשה `cast` ל-`int` או `float` לפי מה שמתאים.

למשל `int(input("Please enter a number"))`. (אם יכניסו ערך לא מספרי תתקבל שגיאה)

max, min, abs:

פונקציות מובנות שלא דורשות מחלקה חיצונית (כמו `math` למשל) למקרים שאסור להשתמש

במחלקות חיצוניות.

range:

`range(start: end: step)` – כולל `start`, לא כולל `end`, לפי קפיצות של `step`.

`range` הוא גנרטור (מוסבר בהמשך בהרחבה) ולכן בפועל לא מכין שום רשימה בזיכרון.

למשל עבור `range(n, n + 3, x)` נקבל `(n, n + x, n + 2x)` בתנאי שכל הערכים בין `n` ל-`n + 3`.

דוגמאות שחשוב להבין:

`list(range(0,5)) = [0,1,2,3,4]`

`list(range(5,0,-1)) = [5,4,3,2,1]`

`list(range(-10,-15,-1)) = [-10,-11,-12,-13,-14]`

`list(range(-15,-10,-1)) = []`

`list(range(-15,-10,1)) = [-15,-14,-13,-12,-11]`

:Casting

:bool()

- עבור מספרים, כל מספר שונה מ-0 יחזיר *True*, ו-0 יחזיר *False*
- עבור כל דבר אחר – אם ריק (מחרוזת ריקה, רשימה ריקה...) נקבל *False*, לא ריק *True*.

:set(), list()

- עבור מחרוזות, יהפוך כל אות לאיבר.
- אם רוצים סט או רשימה עם המחרוזת כמו שהיא, פשוט $a = \{\text{"hello"}\}$ או $a = [\text{"hello"}]$

:dict()

- נניח שאנחנו רוצים מילון מהצורה: $\{'two': 2, 'one': 1\}$
- ```
dict(one = 1, two = 2)
```
- ```
dict(['two', 2], ['one', 1])
```
- (אפשר גם ליצור אותו בדיוק כמו שהוא כתוב: $a = \{'two': 2, 'one': 1\}$)

:type

הפונקציה *type* תחזיר את סוג האובייקט, למשל *type("hi") is str* יחזיר *True*.
סוגי אובייקטים נפוצים: *str, dict, set, tuple, int, float, bool, None*. בנוסף:

```
type({}) >>> dict
type(()) >>> tuple
type([]) >>> list
type("") >>> str
```

:id

הפונקציה *id* תחזיר כתובת ייחודית לאובייקט הנתון. ה-*id* מובטח להיות קבוע וייחודי לכל אורך חייו של האובייקט. ברוב המחשבים הכתובת הייחודית תהיה פשוט הכתובת של האובייקט בזיכרון. למשל *id(lst1)* יחזיר את הכתובת בזיכרון ש-*lst1* יושב בו.

אלגוריתם GCD (של אוקלידס):

```
def GCD(x, y):
    while y > 0:
        x, y = y, x%y
    return x
```

immutable vs mutable

immutable: (נפוצים: *int, float, bool, string, tuple*)

אובייקטים שמרגע שיצרנו אותם הם לא ניתנים לשינוי. לדוגמה:

```
string1 = "Hi!"
```

```
print(id(string1), string1) >>> 2444772615312 Hi!
```

כעת "נוסיף" טקסט למחרוזת, ונדפיס שוב:

```
string1 += " how are you?"
```

```
print(id(string1), string1) >>> 2444772611248 Hi! how are you?
```

כלומר ה-*id* שונה כי לא מדובר באותם אובייקטים. כל "שינוי" שאנחנו יוצרים על מחרוזות (ובכלל על *immutable*) הוא בעצם יצירת אובייקט חדש לגמרי.

מעבר לזה, אם ניצור מחרוזת דומה למחרוזת קיימת, המשתנים שנבחר למחרוזות יצביעו לאותו מיקום בזיכרון. המחשב מזהה שמדובר במחרוזת קיימת (ולא מפחד שנוכל לשנות אותה לאנשים אחרים שמשתמשים בה כי היא *immutable*) ולכן מפנה אותם לאותה כתובת בדיוק:

```
string1 = "Hi!"
```

```
string2 = "Hi!"
```

```
print("String1:", id(string1), string1) >>> String1: 1640575837608 Hi!
```

```
print("String2:", id(string2), string2) >>> String2: 1640575837608 Hi!
```

- רלוונטי לכל *immutable* נפוץ שרשום בכותרת למעט *tuples*. (בגלל ש-*tuples* הם *immutable* אבל הערכים שהם מכילים הם *mutable*)
-

mutable: (נפוצים: *list, dict, set*)

אובייקטים שניתנים לשינוי לכל אורך הריצה. בהמשך לדוגמה הנ"ל, רשימה שנבצע עליה שינוי תשמור על אותו *id*. רשימות זהות לחלוטין שניצור יקבלו *id* שונה בגלל הסיבה שהן ניתנות לשינוי.

נקודה חשובה לגבי *mutables* – לא מומלץ לשים אותם כמשתנים קבועים בחתימת שיטה, למשל:

```
def my_function(param = []):  
    param.append("thing")  
    return param
```

אם נקרא לפונקציה פעמיים נקבל:

```
my_function() # returns ["thing"]  
my_function() # returns ["thing", "thing"]
```

ממבט ראשון היינו מצפים לקבל גם בקריאה השנייה רשימה חדשה שמכילה את *["thing"]* אך בפועל ביצענו שינוי על אותה רשימה. הסיבה היא שפייתון תייצר את הרשימה פעם אחת בלבד, וכל קריאה נוספת לשיטה רק תבצע בה שינויים.

פתרון אפשרי למה שרצינו לבצע:

```
def my_function2(param = None):  
    if param is None:  
        param = []  
    param.append("thing")  
    return param
```

בצורה זו אנחנו יוצרים רשימה חדשה בכל קריאה לפונקציה.

[שיקולים לבחירת סוג האובייקט המתאים](#): גישה לאובייקטים *immutable* היא מהירה יותר, אך כל "שינוי" בהם דורש יצירת עותק ולכן הם "יקרים" במקרה זה. שינוי באובייקטים *mutables* הוא "זול" הרבה יותר. לכן, אם נבנה שיטה שתצטרך לבצע שינויים תכופים באובייקט, כנראה שנבחר באובייקט *mutable*. אחרת נשקול שימוש ב-*immutable*.

Strings:

מקרי קיצון אפשריים למחרוזות:

- ירידת שורה בסוף - נוכל להשתמש ב-`string_name.rstrip()` כדי למחוק רווחים, טאבים ושורה חדשה ("`\n`").

השיטה `ord()`:

מחזירה את הייצוג המספרי של אותיות מסוימות (לפי *Ascii*).
הערך המספרי עובר בצורה הבאה: $ord(Z) < ord(A) < ord(z) < ord(a)$
כלומר הערך המספרי של אותיות קטנות גדול מהערך המספרי של אותיות גדולות, והערך המספרי בין האותיות עצמן הוא לפי סדר ה-*ABC*.

שיטות נוספות:

`string_name.find("substring")` – אם "`substring`" היא תת-מחרוזת של `string_name`
השיטה תחזיר את האינדקס של האות הראשונה ב-`substring` בתוך `string_name` (1 – אם לא)
`string_name.count("substring")` – מחזירה את מספר המופעים של `substring` בתוך `string_name` (נוח לספירת מספר המופעים של תו ספציפי)
`string_name.capitalize()` – מחזירה מחרוזת עם אותיות ראשיות גדולות בכל משפט.
`string_name.upper()` – מחזירה מחרוזת בה כל האותיות גדולות.
`string_name.lower()` – מחזירה מחרוזת בה כל האותיות קטנות.

Lists:

- רשימות יכולות להכיל סוגים שונים של איברים (באותה רשימה).
- כדי לייצר בקלות רשימה של אותיות, נשתמש ב- `list("word") = ['w','o','r','d']`
- כדי לייצר בקלות רשימה של מספרים נשתמש ב- `list(range(5)) = [0,1,2,3,4]`

פעולות אריתמטיות על רשימות:

"חיבור" רשימות: $[1,2,3] + [4,5,6] = [1,2,3,4,5,6]$
"כפל" רשימות - $[1,2,3] * 3 = [1, 2, 3, 1, 2, 3, 1, 2, 3]$

חשוב לדעת:

2 הפעולות מחזירות *Shallow Copy* של הרשימות המקוריות. אובייקטים מסוג *int* הם *immutable* ולכן בדוגמה הנ"ל לא נוכל לפגוע ברשימות המקוריות ע"י שינוי כלשהו של הרשימות החדשות. אך אם היינו "מכפילים" רשימה של רשימות, כמו למשל:

```
old_list = [[1,2,3]]
new_list = old_list * 3 >>> [[1, 2, 3], [1, 2, 3], [1, 2, 3]]
```

שינוי מהצורה `new_list[2][0] = 3` יגרום לשינוי בכל תת-הרשימות כולל ברשימה המקורית! הדפסת הרשימות תראה כך:

```
new_list >>> [[3, 2, 3], [3, 2, 3], [3, 2, 3]]
old_list >>> [[3, 2, 3]]
```

חשוב מאוד להבין למה זה קרה! מי שלא הבין יכול לעבור להסבר בהמשך על *Shallow Copy*.

הוספת ערך לרשימה: `list.append(value)`

מחיקת ערך מהרשימה: `list.remove(value)`

חשוב לזכור שהפונקציה הנ"ל לא מקבלת אינדקס אלא ערך ספציפי שאותו אנחנו רוצים למחוק. למשל עבור הרשימה `list1 = [1,2,3]` הפקודה `list1.remove(2)` תמחק את האיבר השני ברשימה למרות שהאינדקס שלו הוא 1. (כלומר הרשימה תהיה `[1,3]`)

בנוסף, אם קיימים יותר מ-2 איברים דומים ברשימה, למשל `list1 = [1,2,1,2]` הפקודה `list1.remove(2)` תמחק את המופע הראשון של 2. (כלומר הרשימה תהיה `[1,1,2]`)

אפשרות נוספת למחיקה:

```
names = ["Alice", "Bob", "Charlie", "Dave"]
del names[1:3]
print(names) >>> ['Alice', 'Dave']
```

:dictionary

מילון הוא *hashable* לכן גישה, מחיקה או הוספה יהיו ב- $O(1)$ (למעט מקרים קיצוניים של $O(n)$)
החיסרון בו (עד פייתון 3.6) הוא חוסר סדר.

יצירת מילון באמצעות fromkeys:

```
seq = ('name','age','sex')
new_dict = dict.fromkeys(seq) >>> {'age': None, 'name': None, 'sex': None}
new_dict = dict.fromkeys(seq, 10) >>> {'age': 10, 'name': 10, 'sex': 10}
```

הוספת ערך למילון: (dict[key] = value)

```
new_dict["hello"] = 3
```

מחיקת ערך מהמילון: (deleted_value = new_dict.pop(key, None))

באמצעות *pop* אנחנו מוחקים את הערך מהמילון "ועל הדרך" מקבלים בחזרה את הערך (*value*)
של מה שהוצאנו. במידה וה-*key* לא נמצא במילון, יוחזר *None* (כי זה מה שהחלטנו שזה יחזיר)

```
new_dict = {'hi': 1, 'bye': 2}
print(new_dict) >>> {'hi': 1, 'bye': 2}
deleted_value = new_dict.pop('hi', None)
print(new_dict) >>> {'bye': 2}
print(deleted_value) >>> 1
```

לולאה (נוחה) למעבר על ערכים במילון: (for key, value in dict.items())

```
grades = {'foo': 80, 'bar': 90}
for name, grade in grades.items():
    print("The grade of " + name + " is " + str(grade))
```

ידפיס:

```
The grade of bar is 90
The grade of foo is 80
```

נקודה חשובה:

keys במילון יכולים להיות אך ורק אובייקטים *immutable*.

Set:

סט הוא מילון שיש בו רק ערכים (ללא כפילויות!).

גם הוא *hashable* (חסר סדר, סיבוכיות $O(1)$)

הוספת ערך: `set_name.add(value)`

מחיקת ערך: `set_name.remove(value)`

אופרטורים (מחזירים *Shallow Copy*):

`set1 < set2` - יחזיר ערך בוליאני האם `set1` הוא תת-סט של `set2` (וגם הקבוצות לא שוות)

`set1 <= set2` - יחזיר ערך בוליאני האם `set1` הוא תת-סט של `set2` (הקבוצות יכולות להיות שוות)

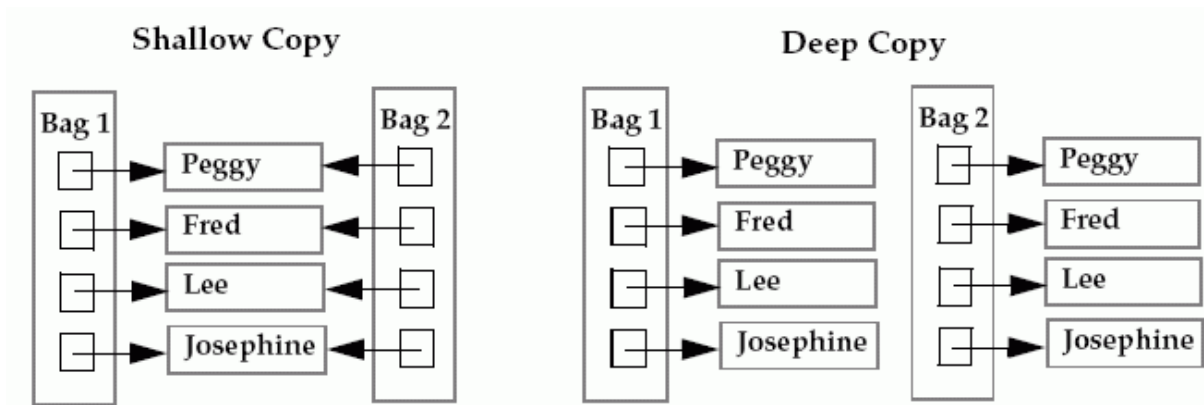
• עובד באותה צורה גם עבור `" > "` ו- `" >= "`.

`set1 | set2` – (איחוד) יחזיר סט חדש שמכיל את הערכים של 2 הסטים (כמובן בלי כפילויות)

`set1 & set2` – (חיתוך) יחזיר סט חדש עם כל האיברים שנמצאים ב-2 הסטים.

`set1 - set2` – (חיסור) יחזיר סט חדש שמכיל את כל האיברים מ-`set1` שלא נמצאים ב-`set2`.

:Deep Copy vs Shallow Copy



:Shallow Copy

העתקה "רדודה" - למשל עבור רשימה, יחזיר רשימה חדשה עם פוינטרים לאותם ערכים של הרשימה הראשונה. בדיוק כמו בתמונה השמאלית. כל שינוי של מבנה הרשימה שאינו כולל שינוי של הערכים עצמם (כמו למשל הוספה או הסרה של איברים לרשימה) יבצע שינוי רק בעותק. אך כל שינוי של הערכים עצמם יגרום לשינוי ב-2 הרשימות (פשוט בגלל ששניהם מצביעים לאותו ערך)

יש המון דרכים ליצירת *Shallow Copy*:

2 דרכים ליצירת עותק "רדוד" לרשימות:

```
new_list = old_list[:]
new_list = list(old_list)
```

כדי ליצור עותק "רדוד" למילון או ל-*set* נשתמש ב:

```
new_dict = old_dict.copy()
new_set = old_set.copy()
```

:Deep Copy

העתקה "עמוקה" – מעתיק גם את הערכים הפנימיים של הרשימה לערכים חדשים בדיוק כמו בתמונה הימנית. במקרה כזה, כל שינוי ברשימה החדשה יבצע שינוי רק ברשימה החדשה.

כדי לבצע *Deep Copy* נשתמש במחלקה *copy* ע"י `from copy import deepcopy`

השימוש יראה כך: `deepcopy(object)`

Slicing

`numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`

עבור `list, string, tuples`. החיתוך מתבצע לפי הצורה הבאה: `[start:end:step]`
מחזיר עותק "רדוד" (*Shallow Copy*) לרשימות לפי הפרמטרים.

`start` ו-`end` הם אינדקסים. החיתוך כולל את `start` אך לא כולל את `end`.

`step` מסמל את הקפיצות בחיתוך.

למשל עבור `numbers[0:6:2]` נקבל `[0, 2, 4]` - חיתוך מ-0 (כולל) עד 6 (לא כולל) בקפיצות של 2.

נזכור שהאינדקסים מתחילים מ-0.

למשל עבור `numbers[5,6]` נקבל `[5]` למרות ש-5 הוא הערך השישי ברשימה.

אם לא רשמנו `start` הכוונה היא מהאינדקס הראשון כולל.

אם לא רשמנו `end`, הכוונה היא עד האינדקס האחרון כולל.

לכן `numbers[:]` יצור העתק של כל הרשימה.

האינדקסים יכולים להיות שליליים, דוגמה לאינדקסים החיוביים והשליליים של המחרוזת `hello`.

<i>h</i>	<i>e</i>	<i>l</i>	<i>l</i>	<i>o</i>
0	1	2	3	4
-5	-4	-3	-2	-1

- למשל עבור `"hello"[-4:-2]` נקבל `"el"`.

נתחיל באינדקס -4 (כולל) עד האינדקס -2 (לא כולל).

- למשל עבור `"hello"[1:-2]` גם נקבל `"el"`.

נתחיל באינדקס 1 (כולל) עד האינדקס -2 (לא כולל)

דוגמאות נוספות שחשוב להבין: (בשחור הפקודה, באדום ההדפסה)

<code>numbers[:2]</code>	<code>[0, 1]</code>
<code>numbers[2:]</code>	<code>[2, 3, 4, 5, 6, 7, 8, 9]</code>
<code>numbers[:2]</code>	<code>[0, 2, 4, 6, 8]</code>
<code>numbers[:]</code>	<code>[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]</code>
<code>numbers[:-1]</code>	<code>[0, 1, 2, 3, 4, 5, 6, 7, 8]</code>
<code>numbers[0]</code>	<code>[]</code>
<code>numbers[::-1]</code>	<code>[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]</code>
<code>numbers[1:5:2]</code>	<code>[1, 3]</code>
<code>numbers[8:-1]</code>	<code>[9]</code>

List Comprehension

(בדוגמאות הבאות נייער רשימה, אך ניתן לייצר סט באותה צורה אם נשתמש בסוגריים מסולסלים)

```
my_list = [x ** 2 for x in range(10)]
print(my_list) >>> [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

מה עשינו?

$x ** 2$ - זה הערך שאנחנו רוצים בכל איבר ברשימה.

$for x in range(10)$ - בדיוק כמו בלולאת for רגילה, לכל איבר מ-0 עד 10 (לא כולל 10) ולכן קיבלנו רשימה שהיא הריבוע של כל מספר בין 0 ל-10 לא כולל.

```
my_2D_list = [[0] * 4 for _ in range(6)]
print(my_2D_list) >>> [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

מה עשינו?

בדיוק כמו בדוגמה הקודמת, $[0] * 4$ (רשימה של אפסים באורך 4) הוא הערך שאנחנו רוצים בכל איבר ברשימה, כאשר יש 0 עד 6 (לא כולל 6) איברים ברשימה. הבחירה של "_" היא שרירותית, מכיוון שלא היה לנו צורך בערך עצמו שאנחנו מקבלים מ- $range$.

```
word = "antidisestablishmentarism"
vowels = list("aeiou")
example = [letter for letter in word if letter not in vowels]
print(example) >>> ['n','t','d','s','s','t','b','l','s','h','m','n','t','r','l','s','m']
```

מה עשינו?

הפעם הוספנו תנאי מסוים, ולכן הרשימה תכיל רק איברים שעומדים בתנאי. נקרא את זה כך: "עבור כל אות מהמילה $word$, תוסיף כל אות לרשימה אם היא לא נמצאת ב- $vowels$ ". $letter$ זה שם שאנחנו בחרנו לטובת קריאות והיינו יכולים באותה מידה לקרוא לזה x .

```
all_coordinates = [(i,j) for i in range(10) for j in range(10)]
print(all_coordinates) >>>
[(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (0, 6), (0, 7), (0, 8), (0, 9), (1, 0), (1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (1, 6), (1, 7),
(1, 8), (1, 9), (2, 0), (2, 1), (2, 2), (2, 3), (2, 4), (2, 5), (2, 6), (2, 7), (2, 8), (2, 9), (3, 0), (3, 1), (3, 2), (3, 3), (3, 4), (3, 5),
(3, 6), (3, 7), (3, 8), (3, 9), (4, 0), (4, 1), (4, 2), (4, 3), (4, 4), (4, 5), (4, 6), (4, 7), (4, 8), (4, 9), (5, 0), (5, 1), (5, 2), (5, 3),
(5, 4), (5, 5), (5, 6), (5, 7), (5, 8), (5, 9), (6, 0), (6, 1), (6, 2), (6, 3), (6, 4), (6, 5), (6, 6), (6, 7), (6, 8), (6, 9), (7, 0), (7, 1),
(7, 2), (7, 3), (7, 4), (7, 5), (7, 6), (7, 7), (7, 8), (7, 9), (8, 0), (8, 1), (8, 2), (8, 3), (8, 4), (8, 5), (8, 6), (8, 7), (8, 8), (8, 9),
(9, 0), (9, 1), (9, 2), (9, 3), (9, 4), (9, 5), (9, 6), (9, 7), (9, 8), (9, 9)]
```

מה עשינו?

"כל איבר ברשימה יהיה טאפל מהצורה (i, j) כאשר i ו- j הם מספרים מ-0 עד 10 (לא כולל 10)"
נשים לב שיש פה לולאות מקוננות. עבור כל j , הלולאה של i תרוץ 10 פעמים.

Iterators:

אינטואיטיבית, להגיד על אובייקט שהוא *iterable* זה להגיד שאפשר לעבור עליו בלולאה. אובייקט שהוא *iterable* יממש את השיטה `__iter__` שתחזיר *iterator*. נתחיל בלהבין איך לולאת `for` עובדת. כשאנחנו רושמים `for i in range(10)`, אנחנו בעצם מגדירים ללולאה על איזה *iterator* לרוץ. במקרה הזה הלולאה תבקש את `iter(range(10))` כלומר *iterator* של `range(10)` ובכל סיבוב לולאה תשתמש ב-`next` כדי לקבל את הערך הבא.

iterator הוא אובייקט "שיעבור" על ערכים של אובייקט אחר.

כדי להיות איטרטור על המחלקה לממש:

- משתנה שיכיל בכל רגע נתון את האיבר הבא בתור.
- המחלקה צריכה להיות *iterable* בעצמה ולכן תממש גם היא את השיטה `__iter__` (ופשוט תחזיר את עצמה - `return self`)
- המחלקה תממש את השיטה `__next__` שתפקידה הוא:
 - להחזיר את הערך הנוכחי (המשתנה שהגדרנו במחלקה)
 - לעדכן את המשתנה לערך הבא שנצטרך להחזיר בפעם הבאה.
 - אם אין עוד איברים להחזיר, להעלות שגיאת *StopIteration*.

אם מחלקה מסוימת היא *iterable* והיא מימשה את השיטה `__iter__` במחלקה, נוכל לרשום:
`for item in iterable:`

כפי שאמרנו מקודם, לולאת `for` תבקש את `iter(iterable)` ותקבל את האיטרטור שהמחלקה מימשה בשיטה הזו. ולאחר מכן הלולאה תרוץ עד שלא יישארו עוד ערכים (*StopIteration*)

[:next](#)

שימוש בשיטה `next(iterator)` על איטרטור, יקרא לפונקציה שמומשה במחלקה `__next__` ולכן שימוש ב-`next` (במימוש נכון) יחזיר את הערך הבא באיטרציה בכל קריאה.

במידה והגענו לסוף האיטרטור (ביקשנו את כל הערכים באיטרטור) הקריאה הבאה תעלה שגיאת ***StopIteration***. כלומר השיטה `next` מתריעה בפני מי שהשתמש בה שנגמרו האיברים באיטרטור. למשל לולאת `for` תדע להפסיק את הלולאה ברגע שמתקבלת שגיאת ***StopIteration***.

נוכל להוסיף ערך דיפולטיבי `next(iterator, None)` שיוחזר במקרה שלא נשאר ערכים. במקרה זה השיטה `next` תחזיר `None` במידה ויגמרו הערכים באיטרטור. (מה שיקרה בפועל זה שהשיטה `next` תחזיר `None` כשהיא תקבל את האקספסן ***StopIteration***)

בעזרת הערך הדיפולטיבי נוכל לבדוק מתי לא נשאר ערכים באיטרטור (למשל בלולאת `while`):

```
lst = [1,2,3]
default = object()
iterator = iter(lst)
cur = next(iterator, default) # first value of the iterator
while cur is not default:
    print(cur)
    cur = next(iterator, default)
```

נתקלנו בהתנהגות דומה לזו של `next` בתרגילים כשעשינו `import` לקובץ חיצוני והשתמשנו בשיטה `readline()` כדי לקרוא שורות בקובץ. בכל פעם שקראנו לשיטה, קיבלנו את השורה הבאה בקובץ ולא יכלנו לחזור שורה אחורה ברגע שהתקדמנו. אותו דבר עם `next`. בכל קריאה ל-`next` נקבל את הערך הבא, ולא נוכל לבקש ערכים קודמים. (בפועל מדובר בדיוק באותו מימוש)

Generators:

כשפיתון מזהה ששיטה מסוימת מכילה את הפקודה *yield* הוא מבין שמדובר בגנרטור ולכן קריאה לשיטה הזו תחזיר גנרטור. *yield* פועל כמו *return* למעט שינוי אחד – בפעם הבאה שניכנס לשיטה, נמשיך מהמקום האחרון שעצרנו בו. מסיבה זו, שימוש ב-*next()* ניתן גם בגנרטורים. בכל קריאה ל-*next()* על גנרטור, הקוד ירוץ מהמקום האחרון בו הגנרטור עצר (שורה אחרי ה-*yield* האחרון שנתקלנו בו) עד ה-*yield* הבא. כל גנרטור הוא איטרטור אבל לא להפך. בדוגמה לאיטרטורים, אם ייגמרו הערכים תעלה שגיאת *StopIteration*.

נוכל לייצר גנרטורים בעצמנו בעזרת *List Comprehension* (עם סוגריים עגולים):

```
my_iter = (x ** 2 for x in range(10))
```

my_iter הוא איטרטור שמכיל 10 איברים שהם ריבועי המספרים 0 – 9

דוגמה למימוש גנרטור. במקרה זה נממש גנרטור שיעבור על עץ בינארי. הגנרטור יחזיר את קודקודי העץ אחד אחרי השני בסדר *in – order*. כלומר קודקוד שמאלי < שורש < קודקוד ימני.

class Node:

```
def __init__(self, data = "", left = None, right = None):
```

```
    self.data = data
```

```
    self.left = left
```

```
    self.right = right
```

```
def __iter__(self):
```

```
    return generate_tree(self)
```

```
def generate_tree(root):
```

```
    if root.left:
```

```
        yield from generate_tree(root.left)
```

```
    yield root.data
```

```
    if root.right:
```

```
        yield from generate_tree(root.right)
```

מה ההבדל בין איטרטור לגנרטור?

אם נשתמש ב-*yield* בשביל להחזיר אובייקט *iterable*, אז יצרנו גנרטור.

אחרת, מדובר באיטרטור. למשל מימוש של מחלקה שמחזיקה במשתנה את הערך הבא של

האיטרציה, ובכל קריאה לשיטה *__next__* היא עושה *return* לערך הזה.

איטרטור בגדול זה סוג של מוסכמה, המימוש של האיטרציה באחריותו.

לעומת זאת, גנרטור פועל באמצעות סינטקס מובנה בפיתון *yield*.

yield:

סינטקס של פיתון להנפקת ערכים. *yield* פועל כמו *return* למעט שינוי אחד – בפעם הבאה שניכנס לשיטה, נמשיך מהמקום האחרון שעצרנו בו. למשל:

```
def simple_generator():
```

```
    yield "my"
```

```
    print("hi")
```

```
    yield "simple"
```

```
    yield "generator"
```

```
my_iter = simple_generator()
```

```
print(my_iter) >>> <generator object simple_generator at 0x000002342F126B48 >
```

```
print(next(my_iter)) >>> my
```

```
print(next(my_iter)) >>> hi
                           simple
```

נשים לב:

- הדפסה של הגנרטור עצמו תדפיס את הייצוג שלו ולא את הערכים שלו.
- בשביל להדפיס את הערכים, נדפיס את ה-*next* שלו.
- ההדפסה של *hi* בוצעה בקריאה השנייה ולא הראשונה.
- השיטה המשיכה מהמקום האחרון שבו עצרה בפעם האחרונה.

yield from: (מפיתון 3.3 ומעלה)

הסינטקס הנ"ל מאפשר לנו לבצע הנפקה לכל הערכים של גנרטור אחר בלי לעשות לולאת *for*. בעצם מדובר בסוג של קיצור דרך ל-*for v in g: yield v*. לדוגמה:

```
def iterable1():
```

```
    yield 1
```

```
    yield 2
```

```
def iterable2():
```

```
    yield from iterable1()
```

```
    yield 3
```

```
print(list(iterable2())) >>> [1, 2, 3]
```

yield from הנפיק את כל הערכים מ-*iterable1()* ואז *yield* הנפיק את 3.

לבסוף קיבלנו רשימה עם הערכים 1,2,3 כמצופה.

lambda:

סינטקס של פיתון ליצירת פונקציה קטנה. עד שלמדנו *lambda* השתמשנו ב-*def* כדי ליצור פונקציות. כעת *lambda* מאפשרת לנו להגדיר פונקציה באותה צורה אך עם סינטקס שונה ונוח יותר לפונקציות קטנות בשורה אחת. הסינטקס הוא כזה: *lambda arguments: expression*. למשל *lambda x: x + 1* היא פונקציה שמקבלת *x* ומחזירה *x + 1*.

יכלנו באותה מידה ליצור אותה באמצעות *def*:

```
def plus_one(x):  
    return x + 1
```

ההבדל הוא שלפונקציה באמצעות *def* יש שם (בדוגמה זו *plus_one*), ולפונקציה שנגדיר באמצעות *lambda* אין שם. לכן כדי שנוכל להשתמש בה, נצטרך לבצע השמה שלה למשתנה, למשל:
plus_one = lambda x: x + 1
ומאותו רגע המשתנה *plus_one* הוא הפונקציה שהגדרנו. כלומר *plus_one(x)* יחזיר את העוקב של *x*.

map:

map מקבלת לפחות 2 ארגומנטים, פונקציה ו-*iterable* (אובייקט שניתן לעבור עליו בלולאה) נניח שהבאנו ל-*map* רשימה ואת השיטה *lambda x: x * x*.
נקבל בחזרה איטרטור של ערכי ההחזרה של הפונקציה על הערכים של ה-*iterable*.
כלומר, בכל פעם שנבקש ערך מהאיטרטור שנקבל בחזרה מ-*map*, תופעל הפונקציה על הערך הבא ברשימה ונקבל בחזרה את ערך ההחזרה. חשוב להבין ש-*map* חוסכת מקום (וזמן) מכיוון שהיא לא מפעילה את הפונקציה על הערכים לפני שאנחנו מבקשים ממנה לעשות זאת ("ע"י *next*) וגם אז היא מבצעת את הפעולה כל פעם על ערך בודד. דוגמה:

```
iterable_object = [1,2,3,4,5]  
func = lambda x: x * x  
new_iter = map(func, iterable_object)  
print(next(new_iter)) >>> 1  
print(next(new_iter)) >>> 4  
print(next(new_iter)) >>> 9
```

נקודה חשובה:

שימוש ב-*List Comprehension* בצורה הבאה: *[f(x) for x in iterable]* יחזיר רשימה שתכיל באופן דומה את ערכי ההחזרה של הפונקציה אך תפעיל את הפונקציה על כל האיברים מראש! להבדיל מ-*map* שיבצע את הפעולה איבר-איבר רק אם נבקש ממנו.

Reduce

reduce היא שיטה של *functools* (נצטרך לעשות `from functools import reduce`) היא מקבלת 2 ארגומנטים:

- שיטה שהיא עצמה גם מקבלת 2 ארגומנטים (אסור שיהיה מספר ארגומנטים שונה מ-2)
- *iterable*.
- (יכולה לקבל ארגומנט שלישי, אם כן הארגומנט הזה יהיה ה- x בריצה הראשונה של f)

הקריאה תתבצע כך: *reduce(func, iterable)* לדוגמה:

```
f = lambda x, y: x * y
product = reduce(f, [1, 2, 3, 4])
```

שלחנו שיטה שמקבלת 2 ארגומנטים x, y ומחזירה $x * y$ ושלחנו רשימה עם 4 מספרים. מה ש-*reduce* עושה זה לוקחת את 2 האיברים הראשונים ברשימה 1,2 ושולחת אותם לפונקציה. במקרה זה $x = 1, y = 2$. הפונקציה תחזיר 2. עכשיו *reduce* תשלח לפונקציה את מה שהיא החזירה בתור x ואת האיבר השלישי ברשימה בתור y . כלומר $x = 2, y = 3$. הפונקציה תחזיר $2 * 3 = 6$ ותבצע שוב את הפונקציה הפעם עם הערכים $x = 6, y = 4$. מכיוון שלא נשארו ערכים, הפונקציה תחזיר את תוצאת הכפל ביניהם (24) ותסיים. נוכל לתאר את מה ש-*reduce* עשתה כהרכבה של הפונקציה f בצורה הבאה: $f(f(f(1,2), 3), 4)$

filter

הפונקציה *filter* מקבלת 2 ערכים:

- פונקציה שמחזירה ערך בוליאני *True/False*
- *iterable*.

ומחזירה איטרטור שיכיל את כל הערכים מתוך ה-*iterable* שהפונקציה מחזירה *True* עבורם.

למשל:

```
f = lambda x: x is not None
iterable_object = [1, None, ["hello"], None]
filtered_iter = filter(f, iterable_object)
```

במקרה הנ"ל, *filtered_iter* יהיה איטרטור שיחזיר רק את הערכים ברשימה שהם לא *None*:

```
print(next(filtered_iter)) >>> 1
print(next(filtered_iter)) >>> ["hello"]
print(next(filtered_iter)) >>> StopIteration
```

Powerset:

שיטה שמחזירה את כל תת-המחרוזות של רשימה מסוימת. בדוגמה זו נממש שיטה שמחזירה רשימה עם כל הפרמוטציות של רשימה, אך יוכלו לבקש מאיתנו לממש עם סט וכו'.

```
def power_set_list(input_list):
    output = [[]]
    for input_item in input_list:
        output += [output_item + [input_item] for output_item in output]
    return output
```

מה השיטה עושה:

1. יוצרת רשימה עם רשימה ריקה. (כי רשימה ריקה היא תת-רשימה של כל רשימה)
2. לכל *input_item* מהרשימה שקיבלנו כפרמטר:
א. נוסיף לרשימה *output* את כל הפרמוטציות האפשריות של האיבר הנוכחי בלולאה *input_item* עם כל הרשימות שכבר קיימות בתוך *output*. (בצורה זו נסיים עם רשימה שתכיל את כל הפרמוטציות האפשריות של איברים ברשימה הנתונה)
3. לבסוף נחזיר את *output*

enumerate:

נשתמש ב-*enumerate* כשנרצה לרוץ על רצף מסוים של איברים (למשל רשימה) ולקבל גם את הערך וגם את האינדקס בכל רגע נתון.

enumerate מחזיר *tuple* מהצורה (*index, value*) והוא מכיל את היתרונות של 2 הלולאות:

```
for i in range(len(lst))
for item in lst
```

מכיוון שבריצה אחת נקבל גם את האיבר עצמו וגם את האינדקס שלו.

דוגמה:

```
names = ["Alice", "Bob"]
for index, value in enumerate(names):
    print("index: ", index, "value: ", value)
```

ההדפסה תיראה כך:

```
index: 0 value: Alice
index: 1 value: Bob
```

nested functions

פונקציה שמוגדרת בתוך פונקציה אחרת היא פונקציה מקוננת (*nested function*). יכולות להיות מספר סיבות לממש פונקציה מקוננת למשל כדי למנוע מאחרים גישה לשיטה הפנימית (למרות שזו לא הדרך המקובלת). או למשל לייצר פרמטרים ב-*Scope* של השיטה החיצונית כך שהשיטה המקוננת (הפנימית) יכולה לגשת לכל הפרמטרים האלה (מכיוון שהיא באותו *Scope*) בשאלות ממבחינים קודמים בדרך כלל השתמשו בפונקציות מקוננות כדי לייצר רשימה ב-*Scope* של השיטה החיצונית ולאחר מכן לעדכן ערכים ברשימה בתוך השיטה הפנימית. חשוב לזכור שבמקרה זה הרשימה היא רשימה אחת והיא לא נוצרת כל פעם מחדש, לכן כל קריאה לשיטה תבצע שינוי על אותה רשימה. נראה דוגמה:

```
def f():
    L = []
    def g(x, h):
        L.append(x)
        return list(filter(h, L))
    return g
```

```
foo = f()
>>> foo(3, lambda x: x%2)
>>> foo(2, lambda x: x%3)
>>> foo(1, lambda x: x%2)
```

מה עשינו?

המשתנה *foo* מחזיק בתוכו את מה שהשיטה *f()* מחזירה. השיטה *f()* מחזירה את השיטה *g*. ולכן המשתנה *foo* הוא הפונקציה *g*! ולא הפונקציה *f*. (אם היה רשום *foo = f* אז *foo* היה הפונקציה *f*, הסוגריים מעידים על כך שהשמו למשתנה את ערך ההחזרה של *f* ולא את *f* עצמה) בקריאה לפונקציה *f* יצרנו את הרשימה *L*, ומכיוון ש-*g* נמצאת ב-*Scope* של *f*, היא יכולה לגשת לרשימה ולשנות אותה.

נעת נסתכל על הקריאה *foo(3, lambda x: x%2)* : בקריאה זו שלחנו לפונקציה *g* ערך ופונקציה שהוגדרה על ידי *lambda*. (הפונקציה מחזירה את שארית החלוקה של ערך מסוים ב-2. למשל עבור 3 היא תחזיר $3\%2 = 1$) הפונקציה *g* מוסיפה לרשימה *L* את 3 ולאחר מכן מחזירה רשימה חדשה עם הערכים שעברו את הפילטר של הפונקציה ששלחנו ל-*g*. כלומר כל הערכים ברשימה שלא מתחלקים ב-2. ולכן מהקריאה הזו תודפס לקונסול רשימה חדשה עם הערך 3: [3] נעת הטריק בשאלה, עבור הקריאה השנייה, *foo(2, lambda x: x%3)*, השיטה *g* תוסיף את 2 לרשימה *L* שכבר מכילה את הערך 3! והפעם הפילטר יחזיר ערכים שלא מתחלקים ב-3. 3 מתחלק ב-3 ולכן לא יוחזר בפילטר אך 2 לא מתחלק ולכן יוחזר: [2] והקריאה האחרונה היא בשבילכם להבין:

Exceptions:

אקספצשנים הם שגיאות ריצה. נסתכל לדוגמה על הפונקציה `input()`.
נניח שיש לנו תוכנית שמעוניינת לקבל `int` מהמשתמש בעזרת `input`:
`num = int(input("Please enter a number"))`

עד לא מזמן, "סמכנו" על המשתמש שאכן יכניס `int` אחרת ה-`cast` יזרוק שגיאה והתוכנית תעצור.
מה קורה בפועל במקרה כזה שהמשתמש הכניס תו לא מספרי?
המתודה `int()` שאחראית על ביצוע `cast` לקלט מבינה שהקלט לא מכיל רק מספרים. אנחנו כמתכנתים סומכים על המתודה שתבצע את עבודתה נאמנה ולכן היא לא יכולה פשוט להתעלם מאותיות ולקחת רק מספרים כי זה יכול לגרום לאי הבנות. למשל משתמש שירשום "8 plus 4" יתכוון ל-12, אבל השיטה תבין שמדובר ב-84? או אולי $8 * 4$? לא מובן.
לכן הדבר ההגיוני ש-`int` יכולה לעשות הוא להעלות (`raise`) שגיאה למתכנת שיחליט מה לעשות במקרה הזה. אנחנו בתור מתכנתים יכולים לתפוס את השגיאה (כדי למנוע מהתוכנית לעצור) ולעשות משהו במקרה שמתקבלת שגיאה (כמו למשל להדפיס הודעה למשתמש שהקלט לא תקין).

:raise

נניח שאנחנו כמתכנתים בנינו מחלקה שמבצעת מספר פעולות שימושיות והיא בשימוש נפוץ על ידי מתכנתים שונים בעולם (: יהיו מקרים מסוימים בהם השיטות שלנו יתקלו בשגיאות. למשל כמו השגיאה המתוארת בפסקה הקודמת, שהמשתמש מחליט להכניס ערך לא תקין. במקרה כזה אנחנו נעלה אקספצשן באמצעות `raise`. זה יראה כך:

```
if something_bad_happened:  
    raise ValueError
```

בדוגמה הזו העלנו את השגיאה `ValueError` למתכנת שהשתמש במחלקה שלנו. סוג השגיאה נבחר רנדומלית אך כמובן שלשגיאה צריכה להיות משמעות בהתאם לשגיאה שקרתה.

:try, except

הדרך שלנו לתפוס שגיאות היא בעזרת `try, except`.

```
try:  
    num = int(input("Please enter a number"))  
    return num  
except ValueError:  
    print("oops! you did not mean that.")
```

כל מה שנמצא ב-`try` יבוצע. אם במהלך הריצה נתקבל שגיאת `ValueError` (שזו השגיאה שנקבל את נכניס תו לא מספרי), אז נדפיס למסך שהקלט היה לא תקין. "תפסנו" את השגיאה ולכן הריצה לא תעצור.

:else, finally

```
def get_int():
    try:
        # some thing to do
        result = int(input("Enter a number: "))
    except ValueError:
        # if there is an error:
        print("oops!")
    else:
        # if there was no exception
        result = result * 2
        return result
    finally:
        # do this ALWAYS
        print("hello")
```

הסבר לכל אחד:

try – ניסיון ביצוע הפעולה.

except – אם הייתה שגיאה תבצע את הפקודות הבאות

else – אם לא הייתה שגיאה תבצע את הפעולות הבאות

finally – לא משנה מה קרה במהלך הריצה, מה שנמצא ב-*finally* תמיד יבוצע.

- לפי מה שאמרו בתרגול, אין צורך לזכור סוגי אקספּרשנים למבחן.

Binary Search

```
def binary_search(val, lst):  
    binary_search_helper(val, lst, 0, len(lst))  
  
def binary_search_helper(val, lst, low, high):  
    if high <= low:  
        return -1 # not found  
    mid = (high + low)//2  
    if lst[mid] == val:  
        return mid  
    if lst[mid] > val:  
        return binary_search_helper(val, lst, low, mid)  
    return binary_search_helper(val, lst, mid + 1, high)
```

חיפוש בינארי:

השיטה הנ"ל היא שיטה רקורסיבית שמקבלת רשימה **ממוינת בסדר עולה** (lst) וערך מסוים (val), ומחזירה את האינדקס של האיבר ברשימה. אם האיבר לא נמצא ברשימה נחזיר -1 . השיטה יוצאת מנקודת הנחה שהרשימה הנתונה ממוינת ולכן יכולה לחסוך המון בדיקות.

מה השיטה עושה?

בתחילת ההרצה נייצר 2 משתנים שיגדירו את טווח החיפוש ברשימה בכל כניסה לשיטה. הריצה הראשונה לא יודעת כלום על הרשימה ולכן הטווח יהיה כל הרשימה (מ-0 עד אורך הרשימה) בכל קומה של הרקורסיה נבדוק את הערך במקום האמצעי ברשימה $lst[mid]$:

1. אם טווח החיפוש קטן או שווה ל-0, אין לנו מה לחפש ולכן נחזיר -1 .
2. אם $lst[mid]$ שווה ל- val , סיימנו - תחזיר את האינדקס mid .
3. אם $lst[mid]$ גדול מ- val , אז בהכרח כל הערכים שאחריו גם גדולים מ- val ולכן אין לנו מה להמשיך לחפש בחלק הזה של הרשימה.
נכנס לשיטה שוב, הפעם עם טווח מ- low עד mid .
4. אם $lst[mid]$ קטן מ- val , אז בהכרח כל הערכים שלפניו גם קטנים מ- val ולכן אין לנו מה להמשיך לחפש בחלק הזה של הרשימה. נכנס לשיטה שוב, הפעם עם טווח מ- $mid + 1$ (כי את mid עצמו בדקנו כבר בקומה הנוכחית) עד $high$.

נקודה חשובה:

במהלך הריצה low ו- $high$ משתנים בהתאם לתנאים. נזכור לא לרשום בטעות 0 במקום low ולא $len(lst)$ במקום $high$ כי ככה לא נצמצם את הטווח כמו שצריך.

Selection Sort

```
def selection_sort(lst):  
    for i in range(len(lst)):  
        min_index = i  
        for j in range(i, len(lst)):  
            if lst[j] < lst[min_index]:  
                min_index = j  
        lst[i], lst[min_index] = lst[min_index], lst[i]
```

מיון בחירה:

השיטה הנ"ל היא שיטה איטרטיבית (ללא רקורסיה) שמקבלת רשימה לא ממוינת וממיינת אותה בצורת מימוש *inPlace*. כלומר, ממיינת את הרשימה בתוך עצמה מבלי לייצר רשימת עזר.

מה השיטה עושה?

עבור כל אינדקס ברשימה הנתונה:

1. נחפש את הערך הקטן ביותר:

א. נעבור בלולאה על כל האיברים מהאינדקס של הלולאה החיצונית ועד סוף הרשימה

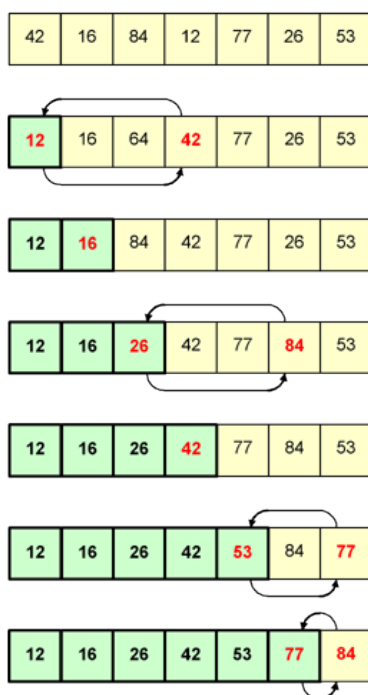
ב. אם הערך הנוכחי בחיפוש קטן יותר מכל קודמיו, נעדכן את *min_index* בהתאם

2. כשנצא מהלולאה הפנימית, המשתנה *min_index* יהיה האינדקס של הערך הקטן ביותר

בתת הרשימה. כעת נחליף את הערך במקום ה-*i* עם הערך במקום ה-*min_index*

בתמונה הבאה יש המחשה טובה לאופן הפעולה. בכל שלב נעבור על תת-הרשימה הצהובה כדי

למצוא את הערך הקטן ביותר ואז נחליף בין המקומות.



Bubble Sort

```
def bubble_sort(lst):  
    for i in range(len(lst)):  
        swap = False  
        for j in range(len(lst) - i - 1):  
            if lst[j] > lst[j + 1]:  
                lst[j], lst[j + 1] = lst[j + 1], lst[j]  
                swap = True  
        if not swap:  
            break
```

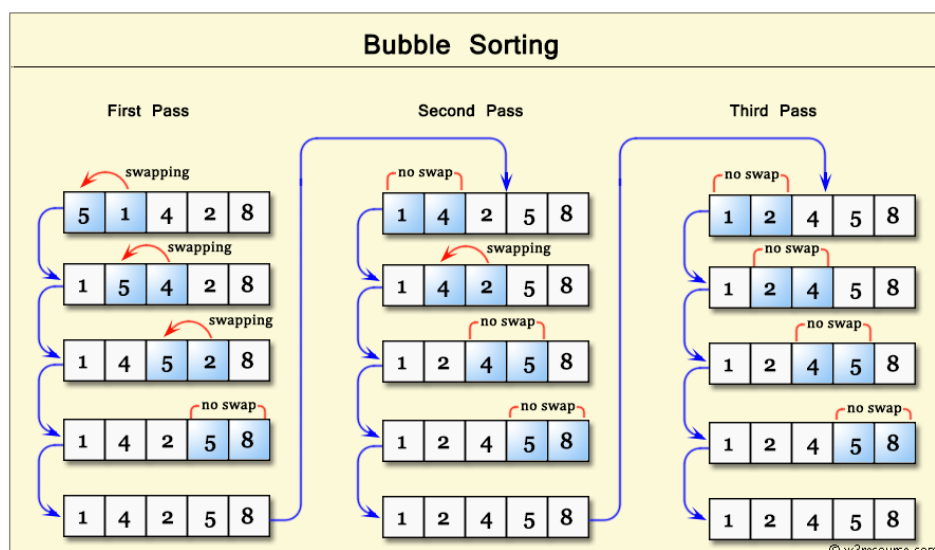
מיון בועות:

השיטה הנ"ל היא שיטה איטרטיבית (ללא רקורסיה) שמקבלת רשימה לא ממוינת וממיינת אותה בצורת מימוש *inPlace*. כלומר, ממיינת את הרשימה בתוך עצמה מבלי לייצר רשימת עזר.

מה השיטה עושה:

עבור כל אינדקס ברשימה הנתונה:

1. נייצר משתנה בוליאני כ-*False* שישתנה ל-*True* רק אם במהלך הריצה הנוכחית ביצענו החלפה כלשהי. כך נדע אם הרשימה ממוינת או לא. (אם במהלך ריצה על הרשימה לא ביצענו שום החלפה סימן שהיא ממוינת כנדרש)
2. "נבעבע" את הערך הגדול ביותר ברשימה לסוף הרשימה באמצעות לולאה פנימית כך:
א. בכל סיבוב לולאה נשווה את הערך הנוכחי עם הערך הבא אחריו.
אם הערך הנוכחי גדול מהבא אחריו, נחליף ביניהם.
(בכל סיבוב של הלולאה הפנימית "נבעבע" לפחות ערך אחד לסוף הרשימה)
ב. אם אכן ביצענו החלפה כלשהי, אפילו רק אחת, נשנה את המשתנה הבוליאני ל-*True* כדי לציין שיש צורך בסיבוב נוסף על הרשימה.
3. אם סיימנו את הלולאה הפנימית והערך הבוליאני נשאר *False*, הרשימה ממוינת ולכן נסיים.



Radix Sort:

```
def radix_sort(lst, radix = 10):
    max_val = max(lst)
    power = 0
    while radix ** power < max_val:
        power += 1
    for p in range(power):
        factor = radix ** p
        buckets = [list() for _ in range(radix)]
        for val in lst:
            tmp = val / factor
            buckets[int(tmp % radix)].append(val)
        lst = []
        for b in range(radix):
            for val in buckets[b]:
                lst.append(val)
    return lst
```

מיון בסיס:

השיטה הנ"ל היא שיטה איטרטיבית (ללא רקורסיה) שמקבלת רשימה לא ממוינת וממיינת אותה בצורת מימוש `outPlace`. כלומר, ממיינת את הרשימה לתוך רשימה חדשה. הרשימה המקורית לא תשתנה.

הסבר טוב מויקיפדיה לאיר המיון עובד:

נביט ברשימה הלא ממוינת הבאה:

170, 45, 75, 90, 802, 2, 24, 66

נמין את הרשימה לפי הסיבית הכי פחות משמעותית, האחדות:

170, 90, 802, 2, 24, 45, 75, 66

כדאי לשים לב ש-802 בא לפני 2, מכיוון ש-802 הופיע לפני 2 ברשימה המקורית. אותו דבר קורה גם עבור הזוגות 170, 90 ו-45, 75.

נמין את הרשימה לפי הסיבית הבאה, העשרות:

802, 2, 24, 45, 66, 170, 75, 90

שימו לב ששוב 802 בא לפני 2 מכיוון ש-802 בא לפני 2 ברשימה הקודמת. לבסוף, נמין את הרשימה לפי הסיבית המשמעותית ביותר:

2, 24, 45, 66, 75, 90, 170, 802

חשוב להבין שכל אחד מהצעדים המופיעים מעלה מצריכים רק מעבר אחד על הנתונים, מכיוון שכל פריט יכול להיות ממוקם בסל הנכון מבלי להיות מושווה עם פריטים אחרים.

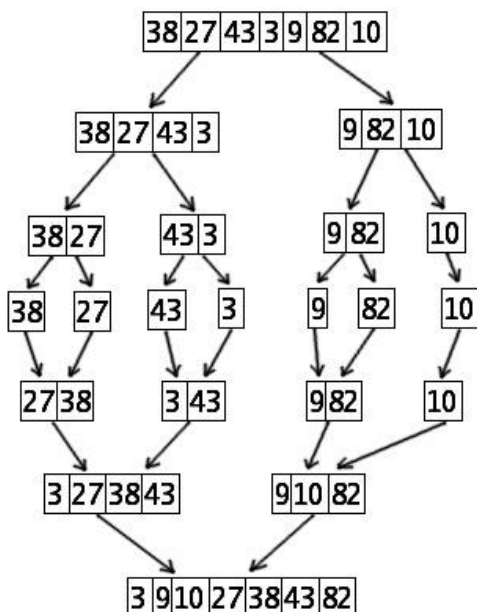
Merge Sort

```
def merge_sort(input):  
    if (len(input) == 0 or len(input) == 1):  
        return input  
    mid = int(len(input) / 2)  
    order1 = merge_sort(input[0:mid])  
    order2 = merge_sort(input[mid:])  
    return merge_two_sorted_lists(order1, order2)
```

```
def merge_two_sorted_lists(a, b):  
    output = []  
    aI = 0  
    bI = 0  
    while aI < len(a) and bI < len(b):  
        if (a[aI] < b[bI]):  
            output.append(a[aI])  
            aI += 1  
        else:  
            output.append(b[bI])  
            bI += 1  
    if (aI < len(a)):  
        output = output + a[aI:]  
    if (bI < len(b)):  
        output = output + b[bI:]  
    return output
```

מיון מיזוג:

השיטה הנ"ל היא שיטה רקורסיבית שמקבלת רשימה לא ממוינת וממיינת אותה בצורת מימוש `outPlace`. כלומר, ממיינת את הרשימה לתוך רשימה חדשה. הרשימה המקורית לא תשתנה. מיון-מיזוג מסתמך על שיטת `merge` שמקבלת 2 רשימות ממוינות ומחזירה רשימה ממוינת אחת (במקרה שלנו השיטה `merge_two_sorted_lists`)



איך המיון עובד?:

מומלץ לקרוא ולהבין דרך התמונה משמאל. בהינתן רשימה לא ממוינת, נפצל את הרשימה ל-2. אם תת הרשימות לא באורך 1 נמשיך לפצל כל אחת מהן ל-2. נמשיך כך עד שכל תת רשימה מכילה איבר בודד. כעת נאחד כל 2 רשימות בצורה ממוינת בעזרת שיטת העזר `merge_two_sorted_lists`. לבסוף תוחזר רשימה חדשה ממוינת.

:Quick Sort

```
import random

def quick_sort(input):
    less = []
    equal = []
    greater = []
    if len(input) > 1:
        pivot = input[random.randint(0, len(input) - 1)]
        for x in input:
            if x < pivot:
                less.append(x)
            if x == pivot:
                equal.append(x)
            if x > pivot:
                greater.append(x)
        return quick_sort(less) + equal + quick_sort(greater)
    else:
        return input
```

מיון מהיר:


השיטה הנ"ל היא שיטה רקורסיבית שמקבלת רשימה לא ממוינת וממיינת אותה בצורת מימוש *outPlace*. כלומר, ממיינת את הרשימה בתוך רשימה חדשה. הרשימה המקורית לא תשתנה. (המימוש הנ"ל מייצר רשימות חדשות בכל קומה של הרקורסיה אך הוא יותר אינטואיטיבי. יש מימוש נוסף (במצגות) שלא מייצר רשימות חדשות למי שמעוניין)

מה השיטה עושה:

1. בהינתן רשימת איברים, בחר איבר מהרשימת באקראי (נקרא: *pivot*, או "איבר ציר").
2. סדר את כל האיברים כך שהאיברים הגדולים מאיבר הציר יופיעו ברשימה חדשה, האיברים ששווים לו ברשימה נוספת והאיברים שקטנים ממנו ברשימה נוספת.
3. באופן רקורסיבי, הפעל את האלגוריתם על רשימת האיברים הגדולים ממנו בנפרד ועל רשימת האיברים הקטנים ממנו בנפרד.
4. תנאי העצירה של האלגוריתם הוא כאשר ישנו איבר אחד. במקרה זה יוחזר האיבר הבודד.
5. לבסוף תוחזר רשימה חדשה ממוינת.

זמני ריצה של שיטות נפוצות:

רשימה

Operation	Average Case
Copy	$O(n)$
Append[1]	$O(1)$
Pop last	$O(1)$
Pop intermediate	$O(k)$
Insert	$O(n)$
Get Item	$O(1)$
Set Item	$O(1)$
Delete Item	$O(n)$
Iteration	$O(n)$
Get Slice	$O(k)$
Del Slice	$O(n)$
Set Slice	$O(k+n)$
Extend[1]	$O(k)$
 Sort	$O(n \log n)$
Multiply	$O(nk)$
x in s	$O(n)$
min(s), max(s)	$O(n)$
Get Length	$O(1)$

מילון

Operation	Average Case
Copy[2]	$O(n)$
Get Item	$O(1)$
Set Item[1]	$O(1)$
Delete Item	$O(1)$
Iteration[2]	$O(n)$

ט

Operation	Average case
x in s	$O(1)$
Union s t	$O(\text{len}(s) + \text{len}(t))$
Intersection s&t	$O(\min(\text{len}(s), \text{len}(t)))$
Multiple intersection s1&s2&...&sn	
Difference s-t	$O(\text{len}(s))$

אלגוריתמים נוספים

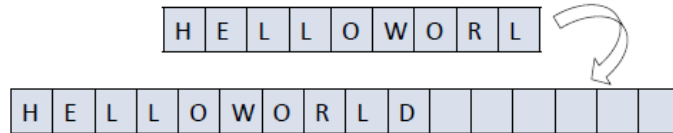
סיבוכיות זמן	אלגוריתם
$O(n)$	חיפוש ליניארי Linear Search
$O(\lg_2(n))$	חיפוש בינארי Binary Search
$O(n^2)$	מיון בחירה Selection Sort
$O(n \cdot \lg_2(n))$	מיון-מיזוג Merge Sort
$O(n^2)$	מיון בועות Bubble Sort
$O(n^2)$	מיון מהיר Quick Sort
$O(\lg_2(x) + \lg_2(y))$	GCD (לפי אלגוריתם אוקלידס)

למידע נוסף:

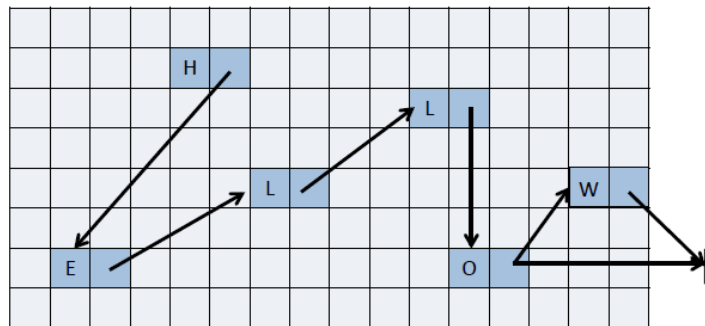
<https://wiki.python.org/moin/TimeComplexity>

Linked Lists

רשימות רגילות מיוצגות בזיכרון כבלוק אחד ארוך של איברים. אם נרצה להוסיף איברים ולהגדיל את הרשימה, נצטרך להעביר את כל האיברים למיקום אחר בזיכרון עם מספיק מקומות פנויים לכל האיברים הנוספים.



לאופן הפעולה הנ"ל יש יתרונות ויש חסרונות. רשימות מקושרות נועדו כדי לפתור חלק מהחסרונות של רשימות רגילות (וכמובן שגם להן יש חסרונות משלהן). בגדול, רשימה מקושרת היא מוסכמה ואופן המימוש שלה יכול להשתנות בהתאם לבעיה הנתונה. נראה דוגמה למימוש מחלקה של רשימה מקושרת בה כל אובייקט מכיל 2 ארגומנטים: ערך ומצביע לאיבר הבא ברשימה.



בצורה זו, אנחנו יכולים להוסיף איברים לרשימה מבלי להעתיק אותם למקומות חדשים בזיכרון. נייצר איבר חדש, נדאג שאובייקט מסוים יצביע אליו ושהוא יצביע לאובייקט אחר וסיימנו. נוכל להוסיף ולמחוק איברים בסיבוכיות של $O(1)$. מימוש המחלקה בדרך כלל יראה כך:

1. המחלקה תשמור במשתנה (נהוג לכנות אותו *head*) את המצביע לאיבר הראשון ברשימה
2. כל איבר יכיל משתנה *next* שיצביע לאיבר הבא בתור ברשימה
3. ה-*next* של האיבר האחרון ברשימה יהיה *None* וככה נדע שלא קיימים איברים נוספים

נזכור שרשימות מקושרות הן מוסכמה והמימוש תלוי במתכנת, לכן נצטרך לבנות למחלקה שיטות נפוצות כמו החזרת איבר ספציפי, הוספת ערכים (במיקום מסוים או בסוף הרשימה) וכו' מעבר על ערכי הרשימה בדרך כלל נעשה באמצעות משתנה *traveler*.

דוגמה ללולאה שמדפיסה את כל ערכי הרשימה:

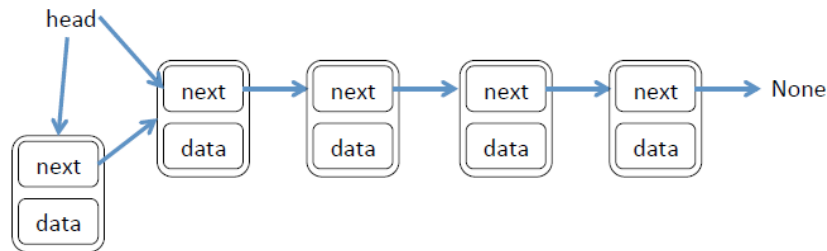
```
traveler = head
while traveler != None:
    print(traveler.data)
    traveler = traveler.next
```

הוספת איבר חדש בתחילת הרשימה המקושרת:

מה נעשה?

1. נייצר אובייקט חדש
2. נעדכן את הפרמטר *head* של המחלקה להצביע אליו
3. נעדכן את ה-*next* שלו להצביע ל-*head* הישן

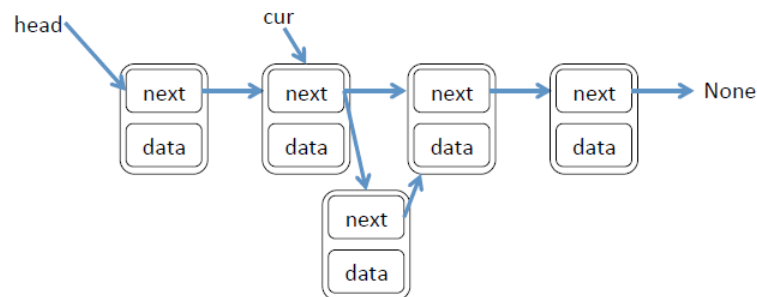
```
head = Node("new val", head)
```



הוספת איבר חדש אחרי איבר ספציפי (נקרא לו *current*):

מה נעשה?

1. נגיע בלולאה או ברקורסיה ל-*current*
 2. נייצר אובייקט חדש
 3. נעדכן את ה-*next* של *current* להיות האובייקט החדש
 4. נעדכן את ה-*next* של האובייקט החדש להיות ה-*next* הקודם של *current*
- ```
current.next = Node("new val", current.next)
```



### מחיקת ה-*head* מהרשימה:

מה נעשה?

1. נשנה את המשתנה במחלקה שמצביע ל-*head* להצביע ל-*next* של *head*
- ```
head = head.next
```

מחיקת איבר ספציפי (נקרא לו *current*):

מה נעשה?

1. נגיע בלולאה או ברקורסיה ל-*current*
 2. נגרום לו להצביע ל-*next* של האיבר שבא אחריו.
- ```
cur.next = cur.next.next
```

## מימוש מחלקת Linked List עם אפשרות איטרציה:

```
class Linked_list:
 def __init__(self, data):
 """Class constructor which receives a data input for the object's first Node.
 Creates a Node object and sets the object's parameters"""
 self.first = Node(data)
 self.iteration_pointer = self.first
 self.last = self.first

 def add_node(self, node):
 """Method which adds a given Node object to the list"""
 self.last.next = node
 self.last = node

 def __next__(self):
 """__next__ method which returns the current iteration_pointer or
 raises a StopIteration exception if there are no more members in the list."""
 if self.iteration_pointer is None:
 raise StopIteration
 output = self.iteration_pointer
 self.iteration_pointer = self.iteration_pointer.next
 return output

 def __iter__(self):
 """__iter__ method which allows creation of an iteration object over the list."""
 return self

class Node:
 """
 Parameters:
 data: a parameter which holds data for the current Node
 next: a pointer to the next Node object which the current one is
 connected to.
 """
 def __init__(self, data):
 """Constructor for the Node class, initiates parameter data with given data parameter."""
 self.data = data
 self.next = None
```



