



ריכוז עזרי הלמידה

# צעדים ראשונים במדעי המחשב ותכנות בפייטון

פרופ' בני שור וד"ר אמיר רובינשטיין, אוניברסיטת תל אביב



# ריכוז עזרי הלמידה

## תוכן עניינים

### שיעור 1

1	הדפסות, פעולות חשבוניות והערות.....
1	טיפוסים.....
2	שניאות.....
2	משתנים.....
3	השוואות.....
4	מחרוזות בפיתון.....
4	פעולות על מחרוזות - חלק א'.....
5	פעולות על מחרוזות - חלק ב'.....

### שיעור 2

5	רשימות (list) - חלק א'.....
6	רשימות (list) - חלק ב'.....
7	רשימות מקוננות.....
7	משפטי תנאי - חלק א'.....
9	משפטי תנאי - חלק ב'.....
10	הגדרת פונקציות.....
11	פרמטרים אקטואליים ופורמליים.....
12	פונקציות שקוראות לפונקציות.....

### שיעור 3

12	לולאות for.....
13	לולאות for על range.....
15	לולאות while.....

16	לולאות מקוננות
17	סיבוכיות
שיעור 4	
18	חיפוש סדרתי
20	חיפוש בינארי
21	מיון בחירה
שיעור 6	
23	הצפנה באמצעות צופן קיסר
24	פיענוח באמצעות צופן קיסר
26	"שבירת" צופן קיסר
27	הצפנה באמצעות צופן הצבה
27	פיענוח באמצעות צופן הצבה
28	מציאת שכיחות אותיות בטקסט
שיעור 7	
30	חלק א' - P1L
33	חלק ב' - P1L
34	היפוך תמונה



# שיעור 1

עזר עבודה לסרטון סמן - הדפסות, פעולות חשבוניות והערות



## הדפסות, פעולות חשבוניות והערות

- הפקודה **print** מדפיסה מידע בחלונית הפלט. למשל - `print(123)` מדפיסה את המספר 123 בחלונית הפלט.

- ביצוע פעולות חשבוניות באמצעות **אופרטורים** בפייתון:

דוגמה	אופרטור	פעולה
$10 + 2$	+	חיבור
$8 - 4$	-	חיסור
$9 * 3$	*	כפל
$12 / 4$	/	חילוק

- הערות (**comments**) - ניתן לכתוב הערות בקוד שלנו על ידי שימוש בסימן `#`. בעת הרצת התוכנית, כל קוד שכתוב מימין לסימן `#` לא מורץ. את הסימן ניתן לשים בתחילת השורה (כך שלמעשה השורה כולה היא הערה), או מימין לשורת קוד קיימת (במצב זה שורת הקוד הקיימת רצה כרגיל).

עזר עבודה לסרטון סמן - טיפוסים



## טיפוסים

**טיפוס נתונים (data-type) או מחלקה (class)** הם מושגים המתארים מהו סוג הנתונים של ערך מסוים.

- הכרנו בשיעור זה 2 מחלקות:
  - int** - מספרים שלמים
  - float** - מספרים עם נקודה עשרונית (שלמים או לא שלמים)



- ניתן לבדוק באמצעות הפקודה **type** מהו טיפוס הנתונים של ערך מסוים. למשל:  
type(12.4)

## עזר עבודה לסרטון סמן - שגיאות



### שגיאות

- הכרנו את מושג השגיאה בפיתון - כאשר יש בעיה מסוימת שפיתון לא יודע להתמודד איתה, התוכנית נעצרת ומודפסת הודעת שגיאה ובה מידע על מה שגרם לשגיאה, לרבות השורה בקוד ותיאור תמציתי של השגיאה.
- שגיאות אותן הכרנו בסרטון הזה - **חלוקה ב-0, תחביר לא תקין, שם לא מוגדר**. ישנם סוגים רבים נוספים של שגיאות אותם לא פגשנו (עדיין).
- **שגיאות לוגיות** - שגיאות שגורמות להתנהגות שונה מזו שציפינו לה, אך אינן גורמות לשגיאה שפיתון מתריע עליה, בשונה מהשגיאות הקודמות אותן ציינו. בשגיאות אלו לקוד שלנו יש משמעות שפיתון יודע לפרש, אך היא שונה מזו שהתכוונו לה.
- בסוף הסרטון הכרנו את **פעולת החזקה בפיתון**, אותה מבצעים באמצעות האופרטור \*\*, למשל: 4\*\*7 מפורש על ידי פיתון כ-7 בחזקת 4.

## עזר עבודה לסרטון סמן - משתנים



### משתנים

- משתנים הם ערכים ששמורים בזיכרון עם שם מסוים. השם הוא צירוף של אותיות, מספרים וקווים תחתונים.
- נתינת ערך למשתנה נקראת **השמה** ומתבצעת באמצעות האופרטור =, למשל:  
`a = 7, this_is_a_variable = 123`
- ניתן להדפיס את ערכם של משתנים באמצעות הפקודה **print**, למשל: הפקודה `print(x)` תדפיס את ערכו של המשתנה x (בהנחה שקיים כזה).
- ניתן לבצע על משתנים פעולות שונות, בדומה לפעולות שמבצעים על ערכים, למשל:  
`print(x + 3)`
- משתנים כשם כן הם - ניתן לשנות את הערך השמור בהם.



- כאשר זה אפשרי, השתדלו לתת למשתנים שלכם שמות בעלי משמעות המרמזים על תפקידם.

## עזר עבודה לסרטון סמן - השוואות



### השוואות

למדנו כיצד מבצעים השוואות בין ערכים בפיתון - ניתן לבדוק האם ערכים שווים, שונים, האם אחד גדול מהשני וכו'.

- אופרטורים בהם משתמשים לביצוע השוואה:

דוגמה	אופרטור	פעולה
$a == b$	<code>==</code>	שווה
$a != b$	<code>!=</code>	שונה (אינו שווה)
$a < b$	<code>&lt;</code>	קטן מ-
$a > b$	<code>&gt;</code>	גדול מ-
$a <= b$	<code>&lt;=</code>	קטן מ- או שווה ל-
$a >= b$	<code>&gt;=</code>	גדול מ- או שווה ל-

**האופרטור == (שווה-שווה)** שבדק שוויון בין ערכים, שונה מהאופרטור = (שווה יחיד).  
 האחרון הוא אופרטור באמצעותו מבצעים **השמה** למשתנים.

כאשר אנו מבצעים פעולת השוואה, התוצאה היא **True** או **False** (אמת או שקר) בהתאם לנכונות הביטוי.  
 למשל -  $4 > 3$  מקבל ערך **True** כי אכן 4 גדול מ-3, אבל  $2 > 10$  מקבל ערך **False** כי 2 אינו גדול מ-10.



## עזר עבודה לסרטון סמן - מחרוזות בפייתון

### מחרוזות בפייתון

מחרוזת היא טיפוס נתונים המכיל רצף של תווים - בדרך כלל אלו יהיו אותיות באנגלית, מספרים, סימני פיסוק ורווחים.

- כאשר מאתחלים מחרוזת יש לכתוב את רצף התווים בתוך גרשיים, למשל: "this is a string".

- ניתן להגדיר משתנה מסוג מחרוזת, למשל: `s = "this is a string"`
- הדפסת מחרוזת מורכבת תיעשה באופן הבא:

```
print(v1,v2,v3,...)
```

כאשר במקום `v1, v2, v3` ניתן לרשום מחרוזות, מספרים, או שמות משתנים. אין הגבלה על הכמות - ניתן לשתמש בכמה פסיקים שרוצים.



## עזר עבודה לסרטון סמן - פעולות על מחרוזות - חלק א'

### פעולות על מחרוזות - חלק א'

- **שרשור מחרוזות** יבוצע באמצעות האופרטור `+`. למשל, אם ערכי המשתנים `s1, s2` הם מחרוזות כלשהן, אז הערך של `s1+s2` הוא שרשור המחרוזות האלו.
- **שכפול מחרוזות** יבוצע באמצעות האופרטור `*`. למשל, אם `s` הוא משתנה מטיפוס מחרוזת, הערך של `s*4` הוא שכפול של `s` 4 פעמים (ברצף).
- הפקודה **len** משמשת לקבלת אורך המחרוזת - עבור המחרוזת `s`, אורך המחרוזת הוא `len(s)`. אורך המחרוזת הריקה ("") הוא 0.
- הפקודה **in** משמשת לבדיקה האם מחרוזת אחת היא תת מחרוזת של מחרוזת שניה (כלומר האחת מופיעה בתוך השניה כפי שהיא). למשל, `s1 in s2` ישוערך ל-True אם `s1` הוא תת מחרוזת של `s2`, ול-False אם לא.



## עזר עבודה לסרטון סמן - פעולות על מחרוזות - חלק ב'



### פעולות על מחרוזות - חלק ב'

- **השוואה** בין מחרוזות - כדי לבדוק אם מחרוזות **זהות** נשתמש באופרטור  $==$ . למשל, עבור מחרוזות  $s_1, s_2$ , הביטוי  $s_1 == s_2$  ישוערך ל-True אם  $s_1$  זהה ל- $s_2$ , ול-False אם לא.
- באופן דומה ניתן לבדוק אם מחרוזות **שונות** על ידי שימוש באופרטור  $!=$ , למשל:  $s_1 != s_2$ .
- **השוואה לקסיקוגרפית** -מילונית - בין מחרוזות תיעשה באמצעות האופרטורים  $>$ ,  $<$ ,  $=$ , למשל, עבור מחרוזות  $s_1, s_2$ , הביטוי  $s_2 > s_1$  ישוערך ל-True אם  $s_1$  קודמת ל- $s_2$  בסדר המילוני ביניהן.
- גישה לתווים במחרוזת לפי **אינדקס** - עבור מספר  $i$  ומחרוזת  $s$ , הביטוי  $s[i]$  הוא התו שנמצא באינדקס  $i$  במחרוזת  $s$ . שימו לב כי ניתן לגשת רק לאינדקסים בין  $0$  ל- $\text{len}(s)-1$ .
- אינדקסים בשפת פייתון מתחילים ב- $0$ , ולא ב- $1$ .

## שיעור 2

## עזר עבודה לסרטון סמן - רשימות (list) - חלק א'



### רשימות (list) - חלק א'

- **אתחול רשימה נעשה ע"י סוגריים מרובעים [ ]**: את איברי הרשימה נכתוב בתוך הסוגריים המרובעים כאשר הם מופרדים ע"י פסיקים, והם יכולים להיות מכל טיפוס שהוא - מספר, מחרוזת, וכו'. ניתן לכתוב ישירות ערכים, שמות של משתנים, או שילוב שלהם. ניתן לאתחל גם רשימה ריקה ע"י סוגריים מרובעים ריקים.
- סדר האיברים ברשימה חשוב - 2 רשימות בהן אותם איברים אך בסדר שונה אינן זהות!
- **בדיקת אורך הרשימה L**: תיעשה באמצעות הפקודה  $\text{len}$ , כך:  $\text{len}(L)$ .
- **שרשור רשימות**: רשימות  $L_1, L_2$  ניתן לשרשר כך:  $L_1+L_2$ .
- השרשור מתבצע לפי סדר הכתיבה, משמאל לימין, כלומר, קודם  $L_1$  ואחר כך  $L_2$ .
- **שכפול רשימה**: בהינתן רשימה  $L$  ומספר שלם גדול או שווה  $0$  שנסמנו  $n$ , הפעולה  $L*n$  יוצרת רשימה חדשה, שהיא שכפול של הרשימה  $L$  בדיוק  $n$  פעמים. כלומר, מתבצעים  $n-1$  שרשורים של  $L$  לעצמה.





- **השוואת רשימות:** ניתן להשוות את הרשימות  $L_1, L_2$  באמצעות האופרטור  $=, \neq$ , כך:  
 $L_1 = L_2$ . ביטוי זה משוערך ל-True אם ורק אם הרשימות זהות (אותם איברים באותו סדר).
- **בדיקת שייכות של איבר לרשימה:** בהינתן רשימה  $L$  ואיבר  $x$ , נבדוק אם  $x$  נמצא ב- $L$  כך:  $x \in L$ . ביטוי זה ישוערך ל-True אם ורק אם  $x$  נמצא ברשימה  $L$  (פעם אחת או יותר).
- **גישה לאיבר באינדקס מסוים ברשימה:** אם רוצים לגשת לאיבר שנמצא באינדקס ה- $i$  ברשימה  $L$ , עושים זאת באמצעות אופרטור הסוגריים המרובעים, כך:  $L[i]$ .  
שימו לב! ניתן לגשת אך ורק לאינדקסים בין  $0$  ל- $len(L)-1$ . ניסיון לגשת לאינדקסים גדולים או שווים  $len(L)$  יוביל לשגיאה. ע"י גישה בדרך זו ניתן להקרוא את הערך השמור באינדקס  $i$  והן לכתוב באינדקס זה ערך חדש.

## עזר עבודה לסרטון סמן - רשימות (list) - חלק ב'



### רשימות (list) - חלק ב'

- **מציאת מקסימום ומינימום ברשימה:** ניתן למצוא איבר מקסימלי ברשימה  $L$  ע"י שימוש ב- $max$ , כך:  $max(L)$ . באופן דומה, נמצא איבר מינימלי ע"י שימוש ב- $min$ , כך:  $min(L)$ .
- **מיון רשימה:** הפעולה  $sorted$  מייצרת העתק ממויין של הרשימה, כלומר רשימה חדשה בה אותם האיברים מסודרים מקטן לגדול, כך:  $sorted(L)$ .
- **הערה לגבי הפעולות  $sorted$ ,  $min$ ,  $max$ :**  
פעולות אלו יעבדו רק ברשימות שניתן לבצע השוואה בין הטיפוסים שלהן. לדוגמה, לא ניתן להשוות בין מחרוזות למספר, ולכן עבור רשימה המכילה מחרוזות ומספרים הפעולות יובילו לשגיאה.  
בנוסף, השוואה בין מחרוזות נעשית לפי הסדר המילוני של המחרוזות. לדוגמה, המחרוזת "abc" קודמת למחרוזת "cde".
- **הבדלים בין מחרוזות ורשימות:** מחרוזות מכילות אך ורק תווים, בעוד רשימות יכולות להכיל כל מיני איברים.  
ניתן לשנות איברי רשימה אחרי שזו אותחלה, אך לא ניתן לשנות תווים של מחרוזת באותו האופן.



## עזר עבודה לסרטון סמן - רשימות מקוננות



### רשימות מקוננות

- רשימות מקוננות הן רשימות שמכילות בתוכן רשימות נוספות. בדרך כלל נשתמש במונח "רשימה חיצונית" כדי לתאר את הרשימה שמכילה את הרשימות, ואילו "רשימה פנימית" תהיה אחת מהרשימות שנמצאות בתוך הרשימה החיצונית.
- **רשימה פנימית** יכולה להיות רשימה מקוננת בעצמה (כלומר, להכיל עוד רשימות), אך לא ניתקל בכאלו רשימות בקורס שלנו.
  - **רשימה חיצונית** היא רשימה לכל דבר ועניין - גם לה יש אורך (אותו ניתן לדעת על ידי שימוש ב-len) וניתן לגשת לאיברים שלה, שהם רשימות, ע"י גישה לאינדקס המתאים באמצעות האופרטור [].
  - הדברים אמורים גם לגבי כל אחת **מהרשימות הפנימיות** - כל אחת מהן היא רשימה.
  - **גישה לאיבר באינדקס מסוים ברשימה:** אם הרשימה החיצונית שמורה במשתנה בשם L, ניתן לגשת לאיבר באינדקס j של הרשימה (הפנימית) שבאינדקס i (ביחס לרשימה החיצונית) באופן הבא: L[i][j].
- לדוגמה, עבור הרשימה  $L = [[1, 2], [3]]$ , הפקודה  $L[1][0]$  ניגשת תחילה לאיבר באינדקס 1 של L, כלומר, הרשימה [3], ולאחר מכן לאיבר באינדקס 0 של רשימה זו, כלומר, המספר 3.

## עזר עבודה לסרטון סמן - משפטי תנאי - חלק א'



### משפטי תנאי - חלק א'

- משפטי תנאי** (או פקודות תנאי) הם סוג פקודות בסיסי בתכנות המאפשר לנו לשלוט בסדר הפעולות של התוכנית אותה אנחנו כותבים.
- משפטי תנאי מאפשרים להתנות ריצה של חלק מסוים בקוד בקיומו (או אי-קיומו) של תנאי מסוים.
- לפניכם מספר דוגמאות לקטעי קוד ובהם תנאי כללי בשם condition.
- תנאי זה הוא בעל ערך בוליאני בפייתון, כלומר ערכו הוא True או False.



כאשר אנחנו כותבים תוכנית בפייתון, תנאי זה יכול ללבוש צורה של כל ביטוי בעל ערך בוליאני, כמו למשל  $5 < 3$  (False),  $3 == 3$  (True) וכו', כפי שראינו בשיעור.

- **ביצוע פקודות מסוימות אם תנאי מסוים מתקיים.** הצורה הבסיסית ביותר למשפט תנאי. במקרה זה הקוד בנוי כך:

```
if condition:  
    # The code we want to run if condition is True
```

שימו לב! לאחר התנאי (בשורת ה-if) יש צורך לשים נקודותיים.

אם ערכו של condition הוא True, אז הקוד שנמצא בבלוק שמתחת לשורה שמתחילה ב-  
if (ומוזח ברווחים או Tab ימינה) יתבצע.  
אם ערכו של condition הוא False, נדלג על הקוד בבלוק הנ"ל ונמשיך בריצת התוכנית  
שאחרי משפט התנאי.

- **ביצוע פקודות מסוימות כאשר תנאי מסוים מתקיים, ופקודות אחרות כאשר התנאי אינו מתקיים.** במקרה זה הקוד בנוי כך:

```
if condition:  
    # The code we want to run if condition is True  
else:  
    # The code we want to run if condition is False
```

גם כאן אם condition הוא True ירוץ הקוד שמתחת ל-if, אך אם condition הוא False, ירוץ דווקא הקוד שנמצא מתחת ל-else (שימו לב – גם לאחר ה-else מופיעות נקודותיים).  
במבנה כזה של משפט תנאי בכל מקרה אחד מקטעי הקוד הנ"ל ירוצו, בהתאם לערכו של condition. לאחר ריצת אחד מקטעי הקוד התוכנית ממשיכה לרוץ לפי הקוד שאחרי משפטי התנאי האלו.



## הזחות

כאשר כותבים קוד בפייתון, ובפרט כאשר משתמשים במשפטי תנאי, יש חשיבות גדולה למיקום של קטעי הקוד השונים.

כאשר אנו מעוניינים שקטע קוד מסוים ירוץ לפי קיום או אי קיום של תנאי מסוים, נזיח אותו ברווחים (מקובל להשתמש ב- 4 רווחים, ניתן גם ללחוץ על Tab) מתחת לשורה בה כתבנו if או else. למשל, נתבונן בקטע הקוד הבא:

```
if condition:  
    print("Hello")  
print("World")
```

שימו לב כי ההדפסה של **המחרוזת Hello מוזחת ימינה** מתחת למשפט התנאי. המשמעות היא כי שורה זו תודפס רק אם condition ישוערך ל-True. לעומת זאת, ההדפסה של **המחרוזת World אינה מוזחת ימינה**, ולכן היא אינה חלק מה"בלוק" של משפט התנאי - ההדפסה הזו תבוצע בכל מקרה.

כאשר אתם כותבים קוד בסביבת הפיתוח Codeboard, ולמעשה בכל סביבת פיתוח נפוצה אחרת של פייתון, מרבית ההזחות מבוצעות באופן אוטומטי כאשר יורדים שורה לאחר כתיבת משפט תנאי (שמתחיל ב-if או else ומסתיים בנקודותיים).

עזר עבודה לסרטון סמן - משפטי תנאי - חלק ב'



### משפטי תנאי - חלק ב'

#### משפטי תנאי מקוננים

בסרטון ראינו צורה נוספת, מורכבת יותר לכתיבת משפטי תנאי, הכוללת משפטי תנאי מקוננים. ניתן לכתוב משפט תנאי אחד בתוך קוד שרץ כחלק ממשפט תנאי אחר. הנה דוגמה אפשרית לקטע קוד כזה:  
(condition מייצג תנאי כללי שערכו אמת או שקר)




```

if condition:
    # The code we want to run if condition is True
    if condition2:
        # The code we want to run if condition2 is True
    else:
        # The code we want to run if condition2 is False
else:
    # The code we want to run if condition is False
    
```

כאן ישנו תנאי נוסף שמצויין על ידי condition2. תחילה בודקים האם condition הוא True - אם כן, ממשיכים לקוד שמתחת ל-if הראשון ובודקים האם גם condition2 הוא True, וממשיכים כמו קודם.

שימו לב כי אם condition הוא False, ממשיכים במקרה הזה מיד לקוד שמתחת ל-else (החיצוני), זה שנמצא בתחתית קטע הקוד, ולמעשה כלל לא בודקים מהו ערכו של condition2.

ניתן לבצע עוד שלל שילובים של משפטי תנאי, אך נסתפק בדוגמה זו.

עזר עבודה לסרטון סמן - הגדרת פונקציות 

**הגדרת פונקציות**

**הגדרת פונקציות חדשות בפייתון:**

תחילה כותבים def, לאחר מכן מגדירים את שם הפונקציה, לאחר מכן בסוגריים את שמות הקלטים (או הפרמטרים) של הפונקציה, ולבסוף נקודותיים. למשל:

```
def foo(p1,p2):
```

כאן הגדרנו פונקציה בשם foo המקבלת שני קלטים - p1 ו-p2.

שורה זו נקראת "חתימת הפונקציה".

את **גוף הפונקציה**, כלומר, הקוד שהפונקציה מבצעת כאשר קוראים לה, כותבים מתחת לחתימה של הפונקציה בהזחה ימינה (כרגיל, רווחים או tab).



בגוף הפונקציה ניתן להשתמש בשמות המשתנים שניתנו כקלט לפונקציה, וכן במשתנים נוספים המוגדרים בתוך הפונקציה עצמה.

## עזר עבודה לסרטון סמן - פרמטרים אקטואליים ופורמליים



### פרמטרים אקטואליים ופורמליים

כאשר אנו משתמשים בפונקציות בשפת פייתון (ולמעשה גם בכל שפת תכנות אחרת) אנו מבחינים בין **הפרמטרים הפורמליים** של הפונקציה לבין **הפרמטרים האקטואליים** שלה:

- **פרמטרים פורמליים** – אלו הפרמטרים המשמשים אותנו להגדרת הפונקציה, ומופיעים בחתימתה. בעת הגדרת הפונקציה אין לפרמטרים אלו ערכים ספציפיים. ערכים כאלו ינתנו רק בעת הקריאה לפונקציה.
- **פרמטרים אקטואליים** – אלו הפרמטרים הניתנים כקלט לפונקציה בעת הקריאה לה, ומוצבים לתוך הפרמטרים הפורמליים. לפניכם דוגמה לתוכנית פייתון קצרה:

```
def foo(x, y):  
    # some code
```

```
foo(7, 42)
```

בתוכנית הזו הגדרנו את הפונקציה `foo` (לא באמת מעניין אותנו מה היא עושה ולכן המימוש לא כתוב), ולאחר מכן קראנו לה.

**הפרמטרים הפורמליים** של `foo` הם אלו שמופיעים בחתימת הפונקציה – `x` ו-`y`. בעת הקריאה ל-`foo` מועברים לה הערכים 7 ו-42 – אלו **הפרמטרים האקטואליים** שלה. ערכים אלו מוצבים בפרמטרים הפורמליים, לפי אותו הסדר (`7 x`, `42 y`).



## עזר עבודה לסרטון סמן - פונקציות שקוראות לפונקציות



### פונקציות שקוראות לפונקציות

- ניתן להשתמש בפונקציות - כאלה שמובנות בשפת פייתון או שכתבנו בעצמנו - בתוך פונקציות אחרות שאנחנו כותבים.
- **השימוש בפונקציות הפנימיות** נעשה באופן זהה לאופן השימוש בהן עד עכשיו - מעבירים להן פרמטרים (ייתכן שחלק או כל הפרמטרים מגיעים מהפונקציה החיצונית, זו שקוראת להן), ואם יש ערך חזרה - ניתן לקבל אותו ולהשתמש בו.

## שיעור 3

### עזר עבודה לסרטון סמן - לולאות for



### לולאות for

לולאת for היא תבנית תכנותית המאפשרת לנו לבצע קוד מסוים, מספר ידוע מראש של פעמים. לכן, לולאות for נוחות לצורך מעבר על רשימות או מחרוזות איבר אחר איבר (במקרה של מחרוזות "איבר" הוא תו במחרוזת), וביצוע פעולות מסוימות על כל איבר. על פי רוב נרצה להשתמש בלולאות כאשר נרצה לבצע פעולה דומה לכל איבר ברשימה או במחרוזת, לדוגמה: הדפסת כל איברי רשימה, סיכום איברי הרשימה, וכו'. כל מחזור של ביצוע הלולאה נקרא איטרציה (iteration).

כך נראה התחביר של לולאות for בשפת פייתון:

```
for x in Y:  
    Do something with x
```

בדוגמה זו אנו עוברים על כל איברי האוסף (רשימה או מחרוזת) שנקרא Y. המשתנה x הוא שם משתנה (זמני) שאנו נותנים לאיבר הנוכחי בכל איטרציה.



בגוף הלולאה, שנמצא מתחת לשורה שמתחילה ב-for ומוזח ימינה (רווחים או טאב), נוכל לבצע פעולות מסוימות על האיבר הנ"ל, כאשר בכל הפעולות נשתמש בשם המשתנה שנתנו לו. למשל, אם נרצה להדפיס את כל איברי האוסף, הקוד הבא יבצע זאת:

```
for x in Y:
    print(x)
```

**עזר עבודה לסרטון סמן - לולאות for על range**

**לולאות for על range**

בפייתון, הביטוי range(a,b) מייצג את סדרת המספרים השלמים שמתחילה ב-a ומסתיימת ב-b-1. למשל range(0,100) מייצג את סדרת המספרים בין 0 עד 99. כאשר כותבים בתוך הסוגריים רק מספר אחד, ברירת המחדל היא במתחילים מ-0. למשל הביטוי range(100) גם מייצג את סדרת המספרים בין 0 ל-99. אם כותבים בסוגריים שלושה מספרים, המספר השלישי מציין את ה"קפיצה" בין מספר למספר בסדרה. למשל range(0,100,5) מייצג את הסדרה 0,5,10,15,20,...,85,90,95.

לולאת for בשילוב עם range מאפשרת לנו לבצע מעבר סדרתי על טווח המספרים שהוגדר באמצעות range.

לפניך שתי צורות נפוצות לשימוש בתבנית הזו:

1. **שימוש במספרי הטווח עצמם** – דוגמה לקוד שמדפיס את כל המספרים בין 1 ל-100:

```
for i in range(1,101):
    print(i)
```

דוגמה לקוד שמדפיס את כל המספרים האי-זוגיים בין 1 ל-100:

```
for i in range(1,101,2):
    print(i)
```





## 2. שימוש במספרי הטווח כאינדקסים באוסף כלשהו (רשימה או מחרוזת) - דוגמה

לקוד שמדפיס את כל האיברים ברשימה L ע"י גישה באמצעות אינדקס:

```
n = len(L)
for i in range(n):
    print(L[i])
```

שימו לב כי לולאות for שאינן משתמשות ב-range גם מבצעות מעבר על איברי האוסף בזה אחר זה, אך אז אין לנו גישה לאינדקס של האיבר הנוכחי:

```
for x in L:
    print(x)
```

לכן, אם המידע הזה - כלומר, האינדקס של כל איבר ברשימה - הוא מידע שימושי עבורנו, נעדיף להשתמש בלולאת for בשילוב עם range.

לסיכום, ניתן לבצע מעבר סדרתי על איברי אוסף C בשתי הדרכים הבאות: שתי הדרכים שקולות לחלוטין.

### 1. לולאת for ללא שימוש ב-range:

```
for x in C:
    Do something with x
```

### 2. לולאת for עם range:

```
for i in range(len(C)):
    Do something with C[i]
```

שימו לב לשימוש ב-len(C) כפרמטר ל-range, וכי במקום x משתמשים ב-C[i]



## עזר עבודה לסרטון סמן - לולאות while

### לולאות while

לולאת while היא סוג נוסף של לולאה.

בדומה ללולאת for (עם או בלי range) גם לולאת while מאפשרת לבצע שוב ושוב סדרת פעולות מסוימת. לולאת while עובדת לפי העיקרון הבא – כל עוד מתקיים תנאי מסוים, הקוד שבגוף הלולאה ימשיך לרוץ.

כך נראה מבנה כללי של לולאת while, כאשר condition הוא תנאי כללי המשוערך לערך בוליאני (True או False):

```
while condition:
    Do something
```

כל עוד התנאי condition משוערך ל-True, הקוד שבגוף הלולאה (כל מה שנמצא מתחת לשורה שמתחילה ב-while ומוזח ימינה) ירוץ פעם אחר פעם.

בפעם הראשונה שבה condition משוערך ל-False, הלולאה נפסקת, ועוברים לביצוע הקוד שמופיע לאחר ה"בלוק" של הלולאה (אם יש).

על פי רוב, התנאי יורכב ממשנתה שעלינו לשנות את ערכו בתוך הלולאה. נתבונן למשל בקטע הקוד הבא:

```
i = 0
while i < 10:
    Do something
    i = i + 1
```

כאן התנאי תלוי במשתנה i. שימו לב כי i מאותחל ל-0, והלולאה ממשיכה לרוץ על עוד i קטן מ-10. בגוף הלולאה, i גדל ב-1 בכל איטרציה, כך שלאחר 10 איטרציות ערכו יהיה 10. בשלב זה התנאי  $i < 10$  כבר לא יתקיים (כלומר, ישוערך ל-False), ואנו נצא מהלולאה.



## לולאות אינסופיות

כאשר משתמשים בלולאות while, אחת הטעויות הנפוצות היא כתיבה של לולאה אינסופית, כלומר, לולאה שהקוד שבגוף שלה רץ שוב ושוב לנצח, או עד שהתוכנית תיעצר על ידי גורם חיצוני. לולאה אינסופית תיווצר כאשר נשכח לעדכן את המשתנה הקשור לתנאי הלולאה, או שנגדיר תנאי שלעולם לא ישוערך ל- False.

למשל, בדוגמה למעלה, אם היינו מורידים את השורה בה הגדלנו את ערכו של  $i$  ב-1, הלולאה היתה הופכת ללולאה אינסופית – מכיוון שערכו של  $i$  לא מעודכן, הוא נשאר 0, והתנאי  $i < 10$  תמיד משוערך ל- True. בדומה, גם אילו רשמנו את התנאי  $i >= 0$  while הלולאה היתה רצה לנצח, כי  $i$  מאותחל ב- 0 ורק גדל בכל פעם.

בפיתוח לולאות אינסופיות קורות בעיקר בשימוש לולאת while. לולאות for חסינות יחסית מכניסה ללולאה אינסופית, מכיוון שהן רצות לפי אוסף או טווח מספרים מוגדר מראש.

## עזר עבודה לסרטון סמן - לולאות מקוננות



### לולאות מקוננות

לולאות מקוננות הן שתיים או יותר לולאות הנמצאות אחת בגוף של השניה. בקורס אנו נשתמש רק בלולאות מקוננות בעלות שתי לולאות – פנימית וחיצונית.

לפניך הדוגמה אותה ראינו בסרטון:

```
for i in range(10):
    for j in range(10):
        print("i =", i, ", j =", j)
```

כאן הלולאה החיצונית משתמשת במשתנה  $i$  ואילו הלולאה הפנימית משתמשת במשתנה  $j$ .

בכל איטרציה של הלולאה החיצונית, מתבצעת הלולאה הפנימית במלואה. מועיל לחשוב על הלולאה הפנימית כעל פקודה שנמצאת בתוך הלולאה החיצונית, ומתבצעת בכל איטרציה מחדש.



למשל, באיטרציה החיצונית הראשונה, בה  $i=0$ , תתבצע הלולאה הפנימית ו-  $j$  ירוץ מ- 0 ועד 10. כך יקרה בכל איטרציה חיצונית בהמשך.

בדוגמה השתמשנו רק בלולאות for, אך ניתן כמובן להשתמש בלולאות מקוננות גם עם לולאות while, וכן לשלב בין לולאת for ולולאת while.

השימוש בלולאות מקוננות לא שונה מהותית משימוש בלולאות רגילות, אך יש לשים דגש מיוחד על **ההזחות**. כדי שהלולאה הפנימית תהיה חלק מגוף הלולאה החיצונית, עלינו להזיח אותה ימינה.

## עזר עבודה לסרטון סמן - סיבוכיות



### סיבוכיות

סיבוכיות זמן ריצה היא מדד ליעילותו של אלגוריתם או של תוכנית מחשב. הצגנו 2 מחלקות סיבוכיות זמן:

- 1. סיבוכיות זמן לינארית** - בתוכניות ממחלקה זו הכפלת גודל הקלט בגורם מסוים תגרום להכפלה באותו הגורם של זמן הריצה, למשל - קלט גדול פי 3 יגרום לזמן ריצה גדול פי 3 בערך. פונקציה לדוגמה ממחלקה זו - הפונקציה  $f1$  בה השתמשנו בלולאת for יחידה בעלת  $n$  איטרציות:

```
def f1(n):
    s = 0
    for i in range(n):
        s = s + i
    return s
```

- 2. סיבוכיות זמן ריבועית** - בתוכניות ממחלקה זו הכפלת גודל הקלט בגורם מסוים תגרום להכפלה של זמן הריצה באותו גורם **בריבוע**, למשל - קלט גדול פי 3 יגרום לזמן ריצה גדול פי 9 בערך. פונקציה לדוגמה ממחלקה זו - הפונקציה  $f2$  שבה השתמשנו בלולאות for מקוננות - לולאה פנימית ולולאה חיצונית, כל אחת מהן בעלת  $n$  איטרציות:



```
def f2(n):
    s = 0
    for i in range(n):
        for j in range(n):
            s = s + (i + j)
    return s
```

הכרנו כלי מעשי - קוד המאפשר לנו למדוד זמני ריצה באמצעות הפונקציה clock מהספריה time. בקטע זה שומרים את זמן ההתחלה וזמן הסיום של קטע הקוד שאת זמן הריצה שלו מודדים, ומדפיסים את ההפרש ביניהם:

```
import time
start = time.clock()
(Code to measure)
end = time.clock()
print("measured time:", end-start)
```

## שיעור 4

עזר עבודה לסרטון סמן - חיפוש סדרתי



### חיפוש סדרתי

חיפוש סדרתי הוא חיפוש של איבר ברשימה, על ידי מעבר סדרתי על כל איברי הרשימה והשוואה לאיבר אותו מחפשים - עד שמוצאים אותו (ומחזירים תשובה חיובית) או שלא (ואז מחזירים תשובה שלילית).



הנה הפסאודו קוד שראינו לביצוע חיפוש סדרתי:

קלט: רשימה L, איבר x  
פלט: True אם x מופיע ב-L, False אם לא.  
1. עבור על איברי הרשימה L מההתחלה לסוף, ולכל איבר e כזה:  
1.1 אם האיבר e שווה ל-x:  
1.1.1 החזר True (וסיים)  
2. (אם הגעת עד לכאן) החזר False

בפיתון מימשנו את האלגוריתם כך:

```
def search(L, x):  
    for e in L:  
        if e == x:  
            return True  
    # if we got here the search failed  
    return False
```

**יתרונות החיפוש הסדרתי:**

- מימוש פשוט
- עובד על כל רשימה ללא תנאים מיוחדים

**חסרונות החיפוש הסדרתי:**

- זמן ריצה – במקרה בו האיבר אותו מחפשים נמצא בסוף הרשימה, או לא נמצא כלל, יש להשוות את האיבר אותו מחפשים לכל איברי הרשימה.



## עזר עבודה לסרטון סמן - חיפוש בינארי



### חיפוש בינארי

חיפוש בינארי הוא חיפוש של איבר ברשימה לפי שיטת "הפרד ומשול" – מצמצמים את טווח החיפוש שלנו בכל פעם עד שמוצאים את האיבר ומחזירים תשובה חיובית. במידה שהאיבר לא נמצא, מחזירים תשובה שלילית.

חשוב: כאשר משתמשים בחיפוש בינארי יש לוודא כי הרשימה בה מבצעים את החיפוש ממוינת. כאשר הרשימה לא ממוינת, החיפוש לא תמיד מצליח.

לפניך הפסאודו קוד בו השתמשנו לביצוע חיפוש סדרתי:

<p><u>קלט</u>: רשימה (ממוינת) L, איבר x</p> <p><u>פלט</u>: True אם x מופיע ב-L, False אם לא.</p> <p>1. הגדר את כל הרשימה כטווח החיפוש הראשוני</p> <p>2. כל עוד טווח החיפוש אינו ריק:</p> <p>2.1 אם x שווה לאיבר האמצעי בטווח:</p> <p>2.1.1 החזר True (וסיים)</p> <p>2.2 אחרת, אם x קטן מהאיבר באמצע טווח החיפוש:</p> <p>2.2.1 צמצם את טווח החיפוש לחצי השמאלי, וחזור לשלב 2</p> <p>2.3 אחרת:</p> <p>2.3.1 צמצם את טווח החיפוש לחצי הימני, וחזור לשלב 2</p> <p>3. (אם הגעת לכאן) החזר False</p>
--

מימוש האלגוריתם בפיתון:



```
def binary_search(L, x):
    left = 0
    right = len(L) - 1
    while left <= right:
        mid = (left + right) // 2
        if x == L[mid]:
            return True
        else:
            if x < L[mid]: # go to left half
                right = mid - 1
            else: # go to right half
                left = mid + 1
    return False # if we got here the search failed
```

**האופרטור //** מבצע חלוקה בין שני מספרים ולאחר מכן מעגל את התוצאה למטה. לדוגמה: התוצאה של  $21 // 2$  היא 10 – תוצאת החלוקה הרגילה היא 10.5, ולאחר עיגול למטה מתקבל 10.

### יתרונות החיפוש הבינארי:

- סיבוכיות זמן ריצה טובה יותר מזו של חיפוש סדרתי. חיפוש בינארי רץ בסיבוכיות לוגריתמית, בעוד שחיפוש סדרתי רץ בסיבוכיות ליניארית.

### חסרונות החיפוש הבינארי:

- מימוש מורכב יותר מהחיפוש הסדרתי.
- האלגוריתם עובד במיטבו על רשימות ממוינות בלבד.

עזר עבודה לסרטון סמן - מיון בחירה



### מיון בחירה

מיון בחירה הוא אלגוריתם למיון של רשימת איברים. האלגוריתם עובד באופן הבא: תחילה מאתחלים רשימה ריקה (להלן: הרשימה הממוינת).





עוברים על הרשימה (המקורית)  $n$  פעמים (כאשר  $n$  הוא מספר האיברים ברשימה), ובכל איטרציה מוציאים את האיבר המינימלי ברשימה ומוסיפים אותו לסוף הרשימה הממוינת.

לפניך הפסאודו-קוד למיון בחירה:

קלט: רשימה  $L$

פלט: רשימה  $sorted\_L$  שהינה עותק ממוין של  $L$ .

1. הגדר רשימה ריקה בשם  $sorted\_L$  אליה ייתווספו האיברים שישלפו מ- $L$ .
2. בצע  $n$  פעמים (כאשר  $n$  הוא אורך הרשימה  $L$ ):
  - 2.1 מצא מינימום ב- $L$ .
  - 2.2 מחק את המינימום הנ"ל מ- $L$ .
  - 2.3 שרשר את המינימום הנ"ל לסוף הרשימה  $sorted\_L$ .
3. החזר את  $sorted\_L$ .

בפיתון מימשנו את האלגוריתם כך:

```
def selection_sort(L):
    n = len(L)
    sorted_L = []
    for i in range(n):
        minimum = min(L)
        L.remove(minimum)
        sorted_L.append(minimum)
    return sorted_L
```

### תזכורת:

- הפעולה  $L.remove(minimum)$  מסירה מן הרשימה את האיבר שערכו  $minimum$ .
- אם יש שני איברים בעלי ערך זהה, מוסר האיבר הראשון מביניהם (ביחס לסדר הרשימה).
- הפעולה  $sorted\_L.append(minimum)$  מוסיפה איבר חדש שערכו  $minimum$  בסוף הרשימה  $sorted\_L$ .



סיבוכיות זמן הריצה של מיון בחירה היא **ריבועית**, כלומר, בהרצה של האלגוריתם מבוצעות סדר גודל של  $n^2$  פעולות.

שימו לב כי זו סיבוכיות זמן הריצה בכל מקרה – אפילו אם מריצים את האלגוריתם על רשימה ממוינת!

ישנם אלגוריתמים המבצעים מיון בצורה יותר יעילה (כלומר, בעלי סיבוכיות זמן ריצה נמוכה יותר), אך לא נלמד עליהם בקורס זה.

## שיעור 6

עזר עבודה לסרטון סמן - הצפנה באמצעות צופן קיסר



### הצפנה באמצעות צופן קיסר

בסרטון זה ראינו את הפונקציה להצפנה באמצעות צופן קיסר:

```
def caesar_encrypt(plaintext, offset):  
    alphabet = "abcdefghijklmnopqrstuvwxyz"  
    ciphertext = ""  
    for char in plaintext:  
        if char in alphabet:  
            position = alphabet.find(char)  
            new_position = (position + offset) % 26  
            new_char = alphabet[new_position]  
            ciphertext = ciphertext + new_char  
        else:  
            ciphertext = ciphertext + char  
  
    return ciphertext
```

- **offset: המפתח** בו אנו משתמשים בהצפנה זו הוא מספר המבטא את ההיסט שעל פיו מוחלפות האותיות. בפונקציה שכתבנו זהו הקלט **offset**.



ההיסט יכול להיות מספר חיובי או שלילי, דבר המשפיע על כיוון ההיסט – החלפת האותיות שנמצאות מימין או משמאל.

- **אופטור % (מודולו):** מה עושים אם ההיסט "גדול מדי" עבור אות מסוימת? בהצפנה המבוססת על האלף-בית האנגלי, כיצד נצפין את  $z$  אם ההיסט הוא  $2$ ? התשובה היא שההיסט מתבצע באופן מעגלי – במקרה שלנו האות שבאה לאחר  $z$  היא  $a$ , ואז  $b$  וכך הלאה. לכן בדוגמה הזו,  $z$  תוחלף ב- $b$ . את הפעולה המעגלית משיגים על ידי השימוש ב**אופטור % (מודולו)**, כפי שהודגם בסרטון.
- בדוגמה זו, במימוש של הפונקציה השתמשנו באלף-בית האנגלי ובאותיות קטנות בלבד. התעלמנו מאותיות שאינן מופיעות באלף-בית שהגדרנו (כמו סימני פיסוק, ואפילו אותיות אנגליות גדולות). ניתן כמובן לכתוב מימוש שכן ידע להצפין (ולפענח) גם תווים אחרים. **ניתן להגדיר איזה אלף בית שנרצה** – אותיות בעברית, בספרדית, או אפילו סתם רצף של תווים אקראיים.

עזר עבודה לסרטון סמן - פיענוח באמצעות צופן קיסר



### פיענוח באמצעות צופן קיסר

בסרטון זה ראינו את פונקציה לפיענוח של טקסט שהוצפן באמצעות צופן קיסר. הפונקציה משתמשת בקוד להצפנה של צופן קיסר, אותו כבר פגשנו:



```
def caesar_encrypt(plaintext, offset):
    alphabet = "abcdefghijklmnopqrstuvwxyz"
    ciphertext = ""
    for char in plaintext:
        if char in alphabet:
            position = alphabet.find(char)
            new_position = (position + offset) % 26
            new_char = alphabet[new_position]
            ciphertext = ciphertext + new_char
        else:
            ciphertext = ciphertext + char

    return ciphertext
```

#### לפניכם המימוש של פונקציית הפיענוח:

```
def caesar_decrypt(ciphertext, offset):
    plaintext = caesar_encrypt(ciphertext, -offset)
    return plaintext
```

- **מבנה פונקציית הפיענוח:** הרצת פונקציית ההצפנה, אך עם **היסט בכיוון ההפוך**. תהליך הפיענוח ותהליך ההצפנה זהים, למעט שינוי בהיסט – ההיסט שישמש אותנו בפיענוח הוא ההיסט הנגדי לזה ששימש אותנו בהצפנה. בעולם ההצפנה המבנה הזה נפוץ במיוחד – משתמשים באותה "מכונה" (פונקציה) הן להצפנה והם לפיענוח, אך עם שינוי קל בקלטים.
- **היפוך כיוון ההיסט,** כלומר הוספת סימן השלילה להיסט שהתקבל כקלט, נעשית **בתוך** פונקציית הפיענוח. ההיגיון העומד מאחורי מימוש זה הוא, שהן המצפין והן המפענח ישתמשו **בדיוק** באותו המפתח, במקום שאחד יעבוד עם היסט חיובי והאחר עם היסט שלילי.



## עזר עבודה לסרטון סמן - "שבירת" צופן קיסר

### "שבירת" צופן קיסר

בסרטון זה ראינו כיצד ניתן "לשבור" טקסט שפוענח באמצעות צופן קיסר, ולפענח את הטקסט מבלי שאנו יודעים מראש מהו המפתח הסודי ששימש להצפנה. כתבנו את הפונקציה הבאה, המקבלת כקלט טקסט **מוצפן** ומדפיסה את כל הפיענוחים האפשריים שלו:

```
def caesar_break(ciphertext):
    for offset in range(1, 26):
        maybe = caesar_decrypt(ciphertext, offset)
        print(offset, "-->", maybe)
```

- הפונקציה מניחה כי האלפבית כולל אותיות אנגליות קטנות בלבד. לכן ההיסטים האפשריים הם בין 1 ל-25 בלבד.
- **מתחילים לבדוק את ההיסטים מ-1 ולא מ-0**, כיוון שאין היגיון להשתמש בהיסט 0 - שימוש בהיסט כזה משאיר את הטקסט כפי שהוא. בנוסף, בגלל האופן המעגלי שבו מתבצעת ההצפנה, **מספיק לבדוק עד היסט של 25 בלבד**, שכן היסט של 26 שקול להיסט של 0, היסט של 27 שקול להיסט של 1, וכך הלאה.
- מימוש זה מדפיס את כל הפיענוחים האפשריים. מכיוון שיש מעט כאלה (25), ניתן לעבור במהירות יחסית ולזהות מהו הפיענוח ההגיוני ביותר. זו הסיבה שהופכת את צופן קיסר לצופן "שביר" - ניתן לפענח אותו בקלות.
- ראינו בסרטון מקרה (מנוון) בו יש יותר מאפשרות פיענוח הגיונית אחת. עם זאת, בהנחה שבדרך כלל מצפינים טקסט ארוך המכיל מספר רב של מילים, הסיכוי לקבל שני פיענוחים הגיוניים הוא נמוך במיוחד.



## עזר עבודה לסרטון סמן - הצפנה באמצעות צופן הצבה



### הצפנה באמצעות צופן הצבה

בסרטון זה הכרנו פונקציה להצפנה באמצעות צופן הצבה:

```
def sub_encrypt(plaintext, alphabet, shuffled):  
    ciphertext = ""  
    for char in plaintext:  
        if char in alphabet:  
            position = alphabet.find(char)  
            new_char = shuffled[position]  
            ciphertext = ciphertext + new_char  
        else:  
            ciphertext = ciphertext + char  
  
    return ciphertext
```

- **המפתח** בו אנו משתמשים בהצפנה זו מחולק לשניים: האלף-בית בו אנו משתמשים בסדר המקורי, ועותק נוסף של תווי האלף-בית, אך בסדר חדש. בפונקציה שלנו אלו הקלטים **alphabet** ו-**shuffled** בהתאמה. אות שנמצאת באינדקס ה-i ב-alphabet, תוצפן על ידי האות שנמצאת באותו האינדקס ב-shuffled.
- תהליך ההצפנה דומה למה שראינו בצופן קיסר – עוברים על הטקסט אות אחר אות ומצפינים אותה, כאשר אותיות שאינן מופיעות באלף-בית נותרות ללא שינוי.

## עזר עבודה לסרטון סמן - פיענוח באמצעות צופן הצבה



### פיענוח באמצעות צופן הצבה

בסרטון זה ראינו פונקציה לפיענוח של טקסט שהוצפן באמצעות צופן הצבה. הפונקציה עושה שימוש **בקוד להצפנה** של צופן הצבה, אותו כבר פגשנו:



```
def sub_encrypt(plaintext, alphabet, shuffled):  
    ciphertext = ""  
    for char in plaintext:  
        if char in alphabet:  
            position = alphabet.find(char)  
            new_char = shuffled[position]  
            ciphertext = ciphertext + new_char  
        else:  
            ciphertext = ciphertext + char  
  
    return ciphertext
```

לפניכם המימוש של פונקציית הפיענוח:

```
def sub_decrypt(ciphertext, alphabet, shuffled):  
    plaintext = sub_encrypt(ciphertext, shuffled, alphabet)  
    return plaintext
```

- גם כאן אנו עדים למבנה של פונקציית פיענוח שעושה שימוש בפונקציית ההצפנה. כאן היפוך המפתח בתוך הפונקציה נעשה על ידי החלפת סדר משתני הקלט של הפונקציה sub\_encrypt, כפי שהוסבר בסרטון.

עזר עבודה לסרטון סמן - מציאת שכיחות אותיות בטקסט



### מציאת שכיחות אותיות בטקסט

צופן הצבה נחשב גם הוא לצופן שביר, אם כי מעט יותר קשה לשבור אותו מאשר את צופן קיסר. אם בצופן קיסר לכל טקסט יש 25 הצפנות אפשריות (אם מתעלמים מ"הצפנה" עם היסט 0), כאן יש כבר  $26! = 26 \cdot 25 \cdot \dots \cdot 2 \cdot 1 = 403291461126605635584000000$  (זה די הרבה...). לכן, בניגוד לצופן קיסר, לא ניתן לעבור על כל הפיענוחים האפשריים - זה ייקח יותר מדי זמן, אפילו בשביל מחשב עם יכולות חישוב גבוהות.



מה כן ניתן לעשות?

**ניתן לבצע ניתוח סטטיסטי של הטקסט המוצפן** – נבדוק את שכיחויות האותיות, ונשווה

לשכיחות האותיות בטקסט סטנדרטי באותה השפה.

ההנחה היא שברוב המקרים האות הנפוצה ביותר בטקסט המוצפן החליפה את האות השכיחה ביותר בטקסט המקורי, וכך גם האות השניה הכי שכיחה, וכו'.

כדי לחשב את שכיחות האותיות של טקסט כלשהו, כתבנו את הפונקציה הבאה:

```
def char_count(text):
    alphabet= "abcdefghijklmnopqrstuvwxyz"
    for char in alphabet:
        how_many = text.count(char)
        if how_many > 0:
            percent = how_many * 100 / len(text)
            print(char, "frequency:", percent, "%")
```

- נזכיר כי לשם פשטות השתמשנו כאן באלף-בית האנגלי, עם אותיות קטנות בלבד. ניתן כמובן להפעיל את הפונקציה באופן דומה על כל אלף-בית שהוא.
  - הגרסה הזו של הפונקציה כוללת את השינוי שמדפיס רק תווים ששכיחותם גדולה מ-0 (כלומר, רק תווים שמופיעים בטקסט), כפי שראינו בסרטון.
  - כפי שציינו בסרטון, זהו אינו המימוש היעיל ביותר לפונקציה זו, מכיוון שבכל איטרציה של הלולאה יש לעבור על כל הטקסט ולספור מחדש את האות הנוכחית שאת שכיחותה רוצים לחשב.
  - בפועל ניתוח כזה בדרך כלל אינו מספיק כדי לפענח טקסט מוצפן – בדרך כלל יש צורך בניית מעט מתקדם הכולל גם בדיקת שכיחויות של זוגות ושלושות של תווים המופיעים בסמיכות. למשל – ודאי לא תופתעו לגלות ששלישיית התווים the היא השכיחה ביותר בשפה האנגלית.
- בדיקת שכיחויות כזו היא יותר מורכבת ולא נלמד אותה בקורס, אך היא בהחלט משימה בת ביצוע למחשב מודרני.





# שיעור 7

עזר עבודה לסרטון סמן - PIL - חלק א'



## PIL - חלק א'

PIL (Python Imaging Library) היא ספרייה בפיתוח לעיבוד תמונה. בשיעור זה, כדי להשתמש בספרייה, נרשום בתחילת התוכנית שלנו את פקודת הייבוא:

```
from PIL import Image
```

(הערה: למעשה כך מייבאים תת ספרייה של PIL שנקראת Image)

בסרטון ראינו מספר פעולות בסיסיות שמאפשרת הספרייה הזו:

### פתיחת תמונה קיימת באמצעות הפונקציה `open`:

```
img = Image.open(image_name)
```

הקלט של הפונקציה הוא מחרוזת המכילה את שם התמונה. למשל, התמונה עליה הדגמנו זאת נקראה `example.jpg`:

```
img = Image.open("example.jpg")
```

בדוגמה זו התמונה נשמרת במשתנה `img`.

### הצגת תמונה באמצעות הפונקציה `show`:

```
img.show()
```

התמונה תופיע בחלונת הפלט

### המרה לתמונת 256 גווני אפור:

```
img2 = img.convert('L')
```



הפקודה מחזירה תמונה חדשה שבה כל פיקסל מכיל אחד מ-256 גוני אפור (כל פיקסל מקודד בזיכרון של המחשב באמצעות 8 ביטים, מה שמאפשר, כזכור, לייצג מספרים בין 0 ל-255 - 256 ערכים בסך הכל). בדוגמה זו התמונה לאחר ההמרה נשמרת במשתנה `img2`.

### גישה למטריצה שמייצגת את התמונה:

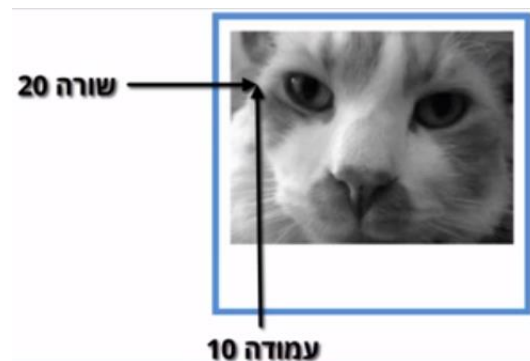
```
mat = img.load()
```

לאחר מכן ניתן לגשת לכל פיקסל בתמונה באמצעות סוגריים מרובעים כך:

```
mat[x, y]
```

כאשר `x` מייצג את אינדקס העמודה ו-`y` את אינדקס השורה. למשל:

```
mat[10, 20]
```



יש לדאוג כמובן שאלו יהיו מספרים שלמים בגבולות מימדי התמונה. אם התמונה ברוחב `w` וגובה `h`, אז חייב להתקיים ש- $0 \leq x \leq w - 1$  ו- $0 \leq y \leq h - 1$ . אם מציבים בפיקסל מסויים ערכים שליליים, הם "יעוגלו" ל-0 (שחור), ואילו ערכים גדולים מ-255 "יעוגלו" ל-255 (לבן).

### יצירת תמונה חדשה:

```
img3 = Image.new('L', (width, height), initial_value)
```

הפרמטר הראשון כאמור מסמל תמונה בפורמט 256 גוני אפור, הפרמטר השני הוא זוג מספרים שמייצגים את מימדי התמונה, והפרמטר השלישי הוא הערך שיינתן לכל הפיקסלים בתמונה החדשה. לאחר מכן ניתן לשנות ערכים אלו כמובן. בדוגמה זו התמונה החדשה שנוצרה נשמרת במשתנה `img3`.



## הדפסת מימדי התמונה:

```
print(img.size)
```

הפקודה מדפיסה זוג מספרים, הרחב והגובה של התמונה. שימו לב ש-size אינה פונקציה, ולכן לא שמנו אחריה סוגריים כמו למשל אחרי show (למתעניינים, size זהו שדה של אובייקט תמונה, נושא שלא נלמד בקורס).

## שינוי מימדי התמונה באמצעות הפונקציה resize:

```
img4 = img.resize((width, height))
```

הפקודה מחזירה תמונה חדשה שהיא למעשה התמונה img עם הרחב (width) והגובה (height) שניתנו כפרמטרים (כלומר, מימדי התמונה img משתנים, ושימו לב כי היחס בין רחב וגובה התמונה לא חייב להישמר). בדוגמה זו התמונה לאחר השינוי נשמרת במשתנה img4.

## שימו לב: להלן נספח למתקדמים - הוא לגמרי בגדר רשות, אבל כדאי לפחות להציץ...

### נספח - שימוש בתמונות אישיות בתרגילים

בתרגילים בשיעור זה, בו אנו עוסקים בעיבוד תמונה, שמרנו עבורכם תמונה אחת ויחידה בה משתמשים בכל התרגילים. מתוך הבנה שחלקכם בוודאי ירצה להשתמש בחומר הנלמד כאן גם על תמונות שלכם, כתבנו פונקציה ייעודית המאפשרת להשתמש בכל תמונה שתרצו, ובתנאי שהיא הועלתה לאינטרנט (לא ניתן להעלות ישירות מהמחשב עליו אתם עובדים):

```
save_image_from_url(url, image_name)
```

הפונקציה נמצאת בתוך הספרייה IP שמיובאת בכל התרגילים (ולכן גם תוכלו להשתמש בה בכל התרגילים בשיעור זה). הפונקציה מקבלת כקלט שתי מחרוזות:

- url - זוהי מחרוזת ובה כתובת ה-url של התמונה. ניתן לאחסן את התמונה בכל מקום שתרצו - אתר פשוט המאפשר לשמור תמונות בחינם הוא <https://imgbb.com>. לפני שאתם מעלים את התמונות לאינטרנט אנחנו ממליצים להקטין את מימדי התמונה כדי שזו תוצג בגודל סביר בחלונית הפלט של קודבורד. ניתן להיעזר באתר הבא: <https://imageresize.org>. אגב, אפשר להעלות בגודל רגיל ולבצע resize באמצעות

PIL, כפי שהוסבר בסרטון 😊



• image\_name – זוהי מחרוזת ובה שם התמונה. יש להוסיף לשם התמונה סיומת

המתאימה לפורמט התמונה (כגון jpg, png, bmp וכו') – למשל: "dog.jpg".

שימו לב שכדי לקרוא לפונקציה יש לרשום תחילה את שם הספרייה בה היא מופיעה (IP), כפי שתראה כעת. בואו נראה דוגמת הרצה פשוטה – העליתי לאתר שהוזכר קודם תמונה שצילמתי (לאחר שהקטנתי את מימדיה), וכתובתה: <https://image.ibb.co/dq07dL/duck.jpg> (שימו לב – כדי לקבל את ה-URL של התמונה עצמה ניתן ללחוץ על התמונה עם המקש הימני של העכבר, ובתפריט שנפתח ללחוץ על 'copy image address'). כדי לשמור את התמונה בקובץ בשם "donald.jpg" יש לבצע את הפעולה הבאה:

```
IP.save_image_from_url("https://image.ibb.co/dq07dL/duck.jpg",  
"donald.jpg")
```

כעת, אם נרצה להשתמש בתמונה ששמרנו, נוכל לפתוח אותה בדיוק כמו שעשינו עד כה (עם התמונה example.jpg):

```
img = Image.open("donald.jpg")
```

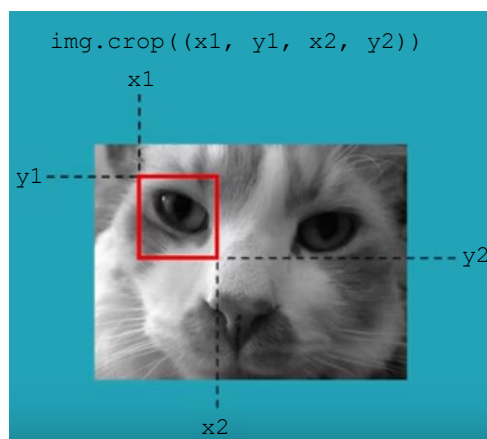
## עזר עבודה לסרטון סמן - PIL - חלק ב'



### PIL - חלק ב'

בסרטון הכרנו שלוש פונקציות שימושיות מהחבילה PIL:

#### crop - חיתוך חלק מלבני מתמונה





הפונקציה crop מקבלת פרמטר אחד ובו רביעיית מספרים בתוך סוגריים. שני המספרים הראשונים הם אינדקס העמודה והשורה של הפינה השמאלית העליונה של המלבן הנחתך  $(x1, y1)$ , ושני המספרים הבאים הם אינדקס העמודה והשורה של הפינה הימנית התחתונה  $(x2, y2)$ .

המלבן הנחתך הוא בעצמו תמונה, וניתן לשמור אותו במשתנה ולבצע עליו כל מה שניתן לבצע על תמונה, למשל:

```
slice = img.crop((30, 20, 80, 80))
slice.show()
```

### rotate - סיבוב תמונה

הפונקציה rotate מקבלת פרמטר אחד - מידת הסיבוב במעלות נגד כיוון השעון. למשל:

```
img2 = img.rotate(45)
img2.show()
```



שימו לב כי הפרמטר יכול להיות גם מספר שלילי.

### histogram - חישוב מספר הפיקסלים בתמונה מכל גוון בין 0 ל-255

עבור תמונת 256 גוני אפור, הפונקציה histogram מחזירה רשימה באורך 256, ובה באינדקס  $i$  מופיע מספר הפיקסלים בתמונה בעלי גוון  $i$  (בין 0 ל-255).

```
hist = img.histogram() # hist is a list
print(hist)
```





בסרטון מימשנו את הפונקציה `upside_down` שמקבלת תמונה, ומחזירה תמונה חדשה שהיא ההיפוך מלמעלה למטה של התמונה המקורית:

```
def upside_down(im):  
    w,h = im.size  
    im_new = im.new('L', (w,h), 255 )  
    mat = im.load()  
    mat_new = im_new.load()  
  
    for x in range(w):  
        for y in range(h):  
            mat_new[x,y] = mat[x,h-y-1]  
  
    return im_new
```

#### דגשים:

- 4 השורות הראשונות בפונקציה "מכינות את הקרקע" לפעולת ההיפוך: יוצרים תמונה חדשה באותו גודל כמו התמונה המקורית, ומשתמשים בפונקציה `load` כדי לקבל גישה לשתי המטריצות של שתי התמונות.
- הלולאות המקוננות עוברות על התמונה פיקסל פיקסל (ליתר דיוק, עמודה עמודה משמאל לימין, ובכל עמודה מלמעלה למטה). ערך הפיקסל במיקום  $(x,y)$  בתמונה החדשה הוא ערך הפיקסל במיקום  $(x, h-y-1)$  בתמונה המקורית (כלומר אינדקס העמודה לא משתנה ואינדקס השורה משתנה לאינדקס שנמצא באותו מרחק מהקצה השני של התמונה).
- הפונקציה מחזירה את התמונה החדשה שנוצרה. שימו לב כי התמונה המקורית לא משתנה.