**שימו לב:** לפני שעברתם על מסמך זה יש להכין את סביבת העבודה שלכם. בצעו זאת על ידי לינק ההתקנה לסביבת עבודה. בנוסף, יש לקרוא את עמודים 4-8 בספר Python של זוהר זילברמן שנמצא באתר.

# חלק 1 – טיפוסים ותנאים

בחלק זה אתם הולכים ללמוד על הטיפוסים השונים שקיימים בפיתון ועל תנאים. טיפוס נתונים הוא אובייקט שמסוגל להחזיק ערכים שונים.  
דוגמאות לאותם ערכים: מספרים שלמים, מספרים עשרוניים, מחרוזות ועוד.

**טיפוסי נתונים**

**מספרים שלמים**

מספרים שלמים, integer אוint , הם הסוג הראשון שנלמד עליו.   
מספר שלם הוא כמובן מספר שאינו שבר, לדוגמה : 0 , 1 , 5 , 12-.

****

**type() –** פונקציה בפיתון המקבלת פרמטר אחד.   
**type** מחזירה את סוג טיפוס הנתונים שמכיל המשתנה. בדוגמא, number מכיל מספר שלם כאמור, int, ולכן **type** של numberמחזירה לנו int''.

**מספרים עשרוניים**

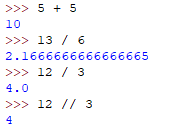
גם מספרים עשרוניים מיוצגים בפיתון, אך מהווים סוג שונה ממספרים שלמים.   
על מנת ליצור מספר עשרוני פיתון נעשה זאת באמצעות נקודה(.). מספר עשרוני בפיתון נקרא **float**.



המושג **float** מגיע מ-**floating-point number**. בעברית, "נקודה צפה", על שם האלגוריתם המיוחד והחסכוני בו משתמש המעבד כדי לייצג בזיכרון שברים עשרוניים בצורה היעילה ביותר.

**פעולות על מספרים**

כמובן מפני שאנחנו מתעסקים עם מספרים, אפשר לבצע עליהם פעולות חשבוניות (כפל, חילוק, חיבור, חיסור, ושארית החלוקה(מודולו – %))



בדוגמה הנ"ל ראינו ששימוש באופרטור / ייתן לנו את התוצאה כמספר עשרוני, ושימוש באופרטור // ייתן לנו את התוצאה כמספר שלם.

אופרטור נוסף שנו להשתמש בו הוא המודולו(שארית החלוקה) ואותו מייצג הסימן % . מודולו יחזיר לנו את שארית החלוקה באופרטור השני כלומר :

21 חלקי 5 , התשובה היא 4 שארית 1.

**בוליאניים**

הטיפוס הבוליאני הוא טיפוס המסוגל להכיל 2 סוגים של ערכים בלבד. **True** ו **False**. זוג הערכים האלו מבטאים אמת ושקר. ניקח דוגמא:

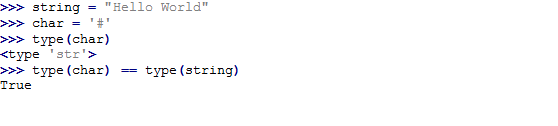
****ניסינו לבדוק האם 5 שווה ל6. כמובן שהתשובה היא לא. פיתון יודעת לחשב ביטויים לוגיים רק בצורה של אמת ושקר ולכן יוחזר לנו הערך **False**. ערכים בוליאניים אמנם מייצגים אמת ושקר אך הם יכולים להיות מיוצגים גם על ידי 0 ו 1.

כמו שאפשר לראות **True** ו **False** מיוצגים גם על ידי הספרות 0 ו 1 . **True** ועוד 1 ייתן לנו 2 וכמו בדוגמה **False** שווה ל 0. פיתון יודעת להתאים את ולתרגם את **True** ו**False** בהתאם לסיטואציה והשימוש בהם. כמובן שאי אפשר לחלק אמת בשקר ולכן במשוואה האחרונה פיתון מבינה שיש צורך להמיר את **True** ו **False** ל 0 ו 1 . ולכן גם קופצת לנו שגיאה. בהמשך הל"ע נראה איזה עוד ערכים מייצגים **True** ו-**False**.

**מחרוזות**

מחרוזת היא רצף תווים (לדוגמא: hello). מחרוזות יכולות להכיל צירופי תווים שונים.

המחרוזת הקצרה ביותר היא מחרוזת ללא תווים כלל (ריקה).    
המחרוזת הארוכה ביותר תמלא את כל הזיכרון הפנוי (למרות שלא באמת נצליח לעשות אחת כזאת).    
המחרוזות הנפוצות ביותר הן מחרוזות המכילות את אותיות האלפבית (a-z, A-Z) הספרות (0-9) והסמנים (+,-,!, \*, (, ), % וכו').

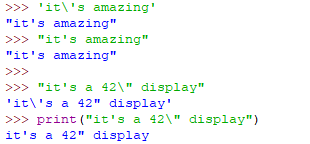
כפי שניתן לראות בדוגמא, על מנת להגיד מחרוזת אנחנו צריכים לפתוח אותה באחד משלושת הסימנים הבאים :

מרכאות – "

גרש – '

או מרכאות משולשות – """

בין גרש למרכאות אין כל כך הבדל אבל היתרון הוא שלפעמים נרצה להשתמש בגרש בתוך המחרוזת עצמה כלומר בתוך משפט וזאת אפשר לעשות בכמה דרכים :



נוכל להשתמש ב Backslash "\" על מנת לציין לפיתון שהתו הבא הולך להיות חלק מהמחרוזת. דרך נוספת היא פשוט להגדיר את המחרוזת בעזרת מרכאות ואז פיתון יודעת שעד שהיא לא תפגוש עוד מרכאות המחרוזת לא מסתיימת.

במקרה השני שהוא קצת יותר מסובך היינו צריכים את 2 סוגי המרכאות ולכן פה כבר פיתון מציינת לנו גם כן על ידי "\" מתי מגיע תו שיודפס כרגיל. אך על מנת לראות את המחרוזת בצורה התקינה נצטרך להשתמש בפעולת הדפסה.

לאחר ה - Backslash יכולים לבוא כל מני תווים שבצירוף ה - Backslash יש להם משמעות מיוחדת.

להלן טבלה המכילה חלק מצירופי התוויים עם Backslash ומשמעותם:

|  |  |
| --- | --- |
| **המשמעות** | **הרצף** |
| רד שורה | \n |
| Tab | \t |
| Backslash | \\ |
| גרש | \' |

**פונקציות על מחרוזות**

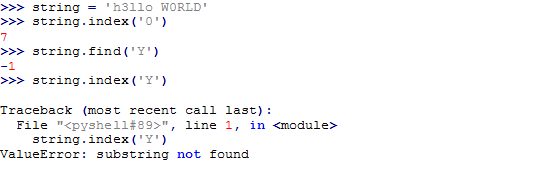
למחרוזות בפיתון יש עוד הרבה פונקציות מגניבות שיעשו לנו את החיים הרבה יותר קלים בניגוד לשפות אחרות שדי מקשות בנושא הזה.

* **upper & lower -**



**upper**ו**lower** הן פונקציות מאוד פשוטות שהופכות את כל התווים שלנו מאותיות גדולות לקטנות ומקטנות לגדולות. כמובן מחרוזות שבהן יש תווים שאינם אותיות אנגליות פשוט לא יושפעו מהפונקציה הזו.

* **index & find –**

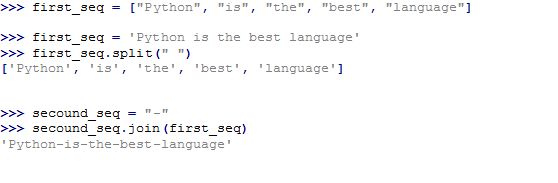


Index ו find הן פונקציות די זהות. הן מקבלות פרמטר אחד ומחזירות את המיקום הראשון שלו במחרוזת, כלומר באיזה אינדקס הערך הזה נמצא בפעם הראשונה. מה ששונה בין 2 הפונקציות הללו הוא שבמקרה ש-find לא תמצא איזשהו ערך היא פשוט תחזיר לנו מ 1- . לעומת זאת Index תזרוק לנו שגיאה.

* **len –**

הפונקציה len מקבלת פרמטר אחד שהוא מחרוזת(בהמשך נראה שlen- יודעת לעבוד עם כל טיפוס נתונים שהוא מסוג רצף) ומחזירה את אורכה.

* **split & join –**



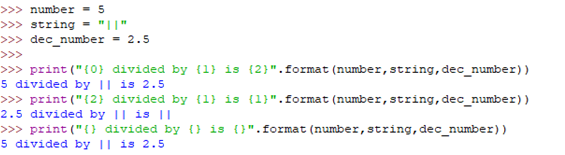
Split ו join הן מתודות מאוד מיוחדות שבעזרתן יהיה לנו הרבה יותר קל לפצל ולחבר איברי רצף שונים.

**split** –פועלת על מחרוזת ומקבלת כפרמטר תו. הפונקציה מפצלת את המחרוזת כך שהתו שהיא קבלה הוא זה שיפריד את המחרוזת. בדוגמא למעלה קיבלנו כפרמטר " "(כלומר רווח) ולכן הפונקציה החזירה לנו רשימה, שבה כל איבר הוא מה שהיה בין 2 רווחים. בשפה פשוטה יותר לפצל את המחרוזת על ידי הרווחים

**join** –היא בדיוק הפוכה מsplit.join  פועלת על מחרוזת וגם מקבלת כפרמטר טיפוס רצף.join  תיקח את טיפוס הרצף שאותו היא קיבלה כפרמטר ותוסיף לו בין כל איבר ואיבר את המחרוזת.

**פרמוט מחרוזות**

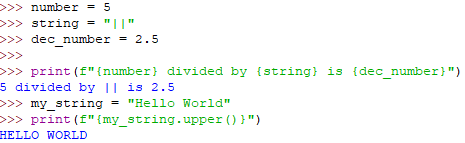
יהיו מקרים שבהם נרצה להדפיס כמה טיפוסי נתונים שונים בהדפסה אחת. בשפות אחרות סביר להניח שפשוט היינו משתמשים ב + בין האיברים השונים ומדפיסים. בפיתון אנחנו נשתמש בformat .



על מנת להשתמש בformat  אנחנו נצטרך לפתוח סוגריים מסולסלים "{}" במקומות שבהם נרצה להכניס את האובייקטים השונים. כל סוגריים כאלו יכילו אובייקט אחר שנרצה להכניס לתוך המחרוזת. לאחר המחרוזת שאותה אנו רוצים לפרמט נשים את format ואחריו את כל האובייקטים שאותם אנו רוצים להכניס אל המחרוזת. נהוג לסמן בתוך הסוגריים 0 ,1 ,2 וכך הלאה בהתאם למיקום הפרמטרים. בסוגריים המסולסלים שבתוכם 0 יכנס הפרמטר הראשון ובסוגריים המסולסלים שבתוכם 1 יכנס הפרמטר השני וכך הלאה. מפני שבדוגמא אנו מכניסים את הפרמטרים לפי סדר אין סיבה למספר.

דרך נוספת היא פרמוט על ידי %. בשיטה זו לא נשתמש במסגרת הקורס ולכן לא נתעמק בה.

C:\Users\u1777777\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\C550B283.tmpבפייתון 3 התווספה דרך חדשה לפרמוט מחרוזות.



בString Formatting אנחנו נצטרך לפתוח סוגריים מסולסלים "{}" במקומות שבהם נרצה להכניס את האובייקטים השונים. כל סוגריים כאלו יכילו שם משתנה שנרצה להכניס לתוך המחרוזת. לפני המחרוזת שנרצה לפרמט נשים את האות f. ניתן גם

לקרוא לפונקציות מתוך הפרמוט.   
**שימו לב** - לא נפעיל פונקציות מורכבות מדי או פונקציית קלט מתוך הפרמוט.

**Slicing**

פיתון מאפשר לנו עבודה מאוד מאוד נוחה עם אובייקטים מסוג רצף. ונותנת לנו לעבוד איתם באמצעות slicing. בעזרת slicing נוכל לחתוך את האובייקט שלנו בצורה קלה נוחה ומיוחדת. על מנת להבין אך slicing עובד בצורה טובה יותר נסו את התרגילים הבאים :

**תרגיל**

**כתוב עבור כל שורה מה יודפס על המסך, Interpreter.**

1   'Hello, World!'**[**3**]**

2   'Hello, World!'**[**4**:**9**]**

3   'Hello, World!'**[:**5**]**

4   'Hello, World!'**[**4**:]**

5   'Hello, World!'**[::**2**]**

6   'Hello, World!'**[**2**:**8**:**1**]**

7   'Hello, World!'**[-**4**]**

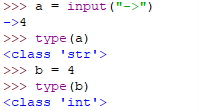
8   'Hello, World!'**[::-**1**]**

**תשובות**

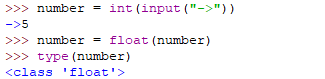
|  |  |
| --- | --- |
| 1. |  |
| 2. |  |
| 3. |  |
| 4. |  |
| 5. |  |
| 6. |  |
| 7. |  |
| 8. |  |

**המרה בין טיפוסי נתונים וקליטת משתנים**

ברגע שאנחנו קולטים או מגדירים משתנים סוג טיפוס הנתונים יישאר קבוע. כך למשל, אם נקלוט בעזרת input הקלט תמיד ישמר כ-string או אם פשוט נגדיר משתנה כמספר שלם הוא תמיד יוגדר כמספר שלם. לפעמים נרצה לעבור ולשנות את סוג טיפוס הנתונים שאנו עובדים לדוגמא :



בשביל לעבור בין טיפוסי נתונים פיתון מביאה לנו את הפונקציות :   str(), int(), long(), float() וגם עבור כל טיפוס נתונים שנכיר בהמשך יש פונקציה דומה. הפונקציות הללו, מקבלות פרמטר אחד והוא האובייקט שאותו נרצה להמיר לסוג טיפוס הנתונים שאותו מייצגת הפונקציה.



**תנאים**

**ביטוי לוגי**

ביטוי לוגי הוא ביטוי המורכב משני אופרנדים או יותר שמחוברים ביניהם באמצעות אופרטורים לוגיים. חישוב מתמטי מחזיר תוצאה מספרית ואילו ביטוי לוגי מחזיר תוצאת אמת או שקר.

אופרטורים לוגיים בסיסיים: <,>,==,!=,>=,<= (יש לשים לב לסדר הכתיבה, למשל: <= ולא =<)

דוגמאות לביטויים לוגיים:



דגשים:

* הערך המספרי של אותיות גדולות קטן מערך אותיות קטנות (קראו על ASCII באינטרנט)
* ערכה של מילה קצרה קטן מערכה של מילה ארוכה אשר מורכבת מאותן אותיות כמו בדוגמא החמישית

**תנאי בסיסי**

תנאים משמשים לשליטה על זרימת התוכנית. במקרה שלתוכנה שלנו יש מספר פעולות אפשריות שהיא יכולה לבצע (לאשר כניסה לאדם לרכבת הרים או לאסור כניסה) אנו נקבע את המשך פעולת הקוד ע"פ קלט מסוים (גובה האדם שרוצה להיכנס לרכבת ההרים) וכך נשלוט על זרימת התוכנית.

דוגמא:

**אם** הגובה יותר מ-1.2 מטר **אז**

תאשר להיכנס

**אחרת**

תאסור להיכנס

את התנאי בדוגמא ניתן לכתוב בפייתון בצורה הבאה:

הערה:

* height הוא משתנה שמכיל מספר שנקלט למערכת.

המרת טיפוסים שונים לטיפוס בוליאני

בפייתון כאשר אנחנו רושמים בתנאי משתנה בפני עצמו הinterpreter ממיר את המשתנה לטיפוס בוליאני.

אובייקטים שמומרים לFalse הם 0 ואובייקטים ריקים כמו רשימה ריקה tuple ריק ועוד. כל שאר האובייקטים מומרים לTrue.

דוגמא:

יש לנו חיישן מרחק שמותקן בדלת אוטומטית. החיישן מחזיר את המרחק של האובייקט שהוא מזהה על מנת שנפתח את הדלת אם הוא מזהה מישהו מול הדלת.

את הפעולה של החיישן בדוגמא ניתן לכתוב בפייתון בצורה הבאה:

הערות:

* distance הוא משתנה אשר מכיל מספר מסוג float שמייצג את המרחק במטרים. מכיוון שאנחנו רוצים לפתוח את הדלת אם החיישן מזהה מישהו (ללא תלות במרחק) אנו נבצע את התנאי בצורה הזאת. כאשר החיישן לא מזהה אף אחד הדלת לא תיפתח כי המרחק שיוחזר מהחיישן הוא אפס.
* openDoor() היא פונקציה שמכילה מספר שורות קוד שגורמות לדלת להיפתח למספר שניות.

אופרטור not

אופרטור זה יוכל לבוא לפני תנאי כדי להפוך את תוצאתו. כלומר, אם התוצאה היא אמת אז not יהפוך את התוצאה לשקר ולהיפך.



הערה:

* כאשר משתמשים ב-not מומלץ לשים סוגריים מסביב לביטוי אשר אנחנו רוצים להפוך את תוצאתו.

בלוקים

בלוק הינו קטע קוד אשר מתחיל לאחר נקודתיים ונמצא בהזחה שונה מקטע הקוד שלפניו. מטרת הבלוקים היא לשייך קוד להוראה מסוימת כמו: תנאי, לולאה הכרזה על פונקציה ועוד.

הזחה היא ריווח של הקוד מתחילת השורה. בפייתון נהוג להשתמש בהזחה של 4 רווחים. יש לשים לב שבעורכי טקסט מסוימים tab ייחשב ל-4 רווחים ובעורכים אחרים tab ייחשב ל-8 רווחים. הזחה לא אחידה בקוד כמו למשל שורה עם 4 רווחים ושורה עם 3 רווחים תגרום לשגיאת אינדנטציה (Indentation) והקוד לא ירוץ. שגיאה זאת קורית מכיוון שהInterpreter לא יודע איזה שורה שייכת לאיזה בלוק.

שיטת הבלוקים הופכת את הקוד לקריא יותר ואת תהליך כתיבת הקוד למהיר יותר בכך שהיא חוסכת את השימוש בסוגריים מסולסלים.

מכיוון שפייתון היא השפה הכי טובה בעולם בפייתון אין צורך בסוגריים מסולסלים כמו בשפות אחרות בכדי להגדיר תחילת וסיום בלוק.

**תנאי מקונן**

תנאי מקונן הוא תנאי בתוך תנאי. אנו נצטרך תנאי בתוך תנאי בכדי לבדוק מקרים מורכים בהם נרצה לבדוק שרצף של תנאים קורה.

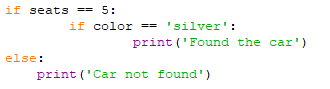
**אם** יש 5 מקומות במכונית

**אם** המכונית כסופה

הדפס "נמצאה המכונית"

**אחרת**

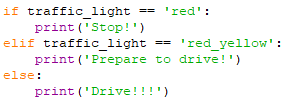
הדפס "לא נמצאה המכונית"



**elif**

הביטוי elif משמש לבדיקת תנאים ע"פ סדר וכאשר יש בין התנאים קשר לוגי. elif בודק תנאי רק אם התנאי הקודם לו לא התקיים.

ניתן להקביל הסבר זה לרמזור. לרמזור כמה מצבים שהקשר הלוגי ביניהם הוא שבנקודת זמן מסוימת רק אחד מהם יכול להתקיים.



בקוד הנ"ל התנאי הראשון ייבדק, אם הוא נכון אז תודפס המחרוזת Stop ואז לא יבדקו שאר התנאים. הקוד ימשיך לפקודה הבאה אחרי קבוצת התנאי הנ"ל.

אם התנאי הראשון לא נכון ייבדק התנאי השני ואם הוא לא נכון יתבצעו הפקודות שנמצאות בelse. בשימוש בelif לא חייבים להשתמש בelse.

יש לציין ששיטה זו שנה מ-switch case אשר קיים בשפות אחרות. בפייתון כמובן יש שיטה יותר פשוטה לעשות switch case.

דוגמאות של שימוש ב-elif:

1. במצב שבו יש בדיקה האם המספר הנקלט הוא 1, 2, 3, או 4.
2. במצב שבו כל מה שה-ELSE מכיל זהIF נוסף.
3. במקרים בהם אנו רוצים ליצור רצף של התניות שקיומן תלוי באי קיומן של ההתניות הקודמות.

**אופרטורים לוגיים מורכבים**

אופרטורים לוגיים מורכבים מאפשרים לנו לעשות תנאים אשר מורכבים ממספר ביטויים לוגיים. מקור האופרטורים הלוגיים הללו הוא מאלקטרוניקה.

|  |  |  |
| --- | --- | --- |
| A | B | output |
| True | True | True |
| True | False | False |
| False | True | False |
| False | False | False |

אופרטורים:

**and** – כאשר שני התנאים מצידי האופרטור מחזירים אמת כל הביטוי יחזיר אמת.

בדוגמא הבאה ישנם שתי משתנים A ו-B אשר נמצאים בתנאי הבא:

הסבר:לפי השורה הראשונה אם ערכם של A ו-B הוא אמת אז תוצאת התנאי תהיה אמת ויודפס OK וכן האלה.



|  |  |  |
| --- | --- | --- |
| A | B | output |
| True | True | True |
| True | False | True |
| False | True | True |
| False | False | False |

**or** – כאשר אחד מצידי האופרטור מחזיר אמת כל הביטוי יחזיר אמת.

בדוגמא המשתנים A ו-B נמצאים בתנאי הבא:



סדר קדימות אופרטורים:

כמו במתמטיקה גם בתנאים לוגיים יש סדר שלפיו מחושב התנאי.

סוגריים ß not ß פעולות חשבוניות ß לוגי פשוט ß and ß or

הערה:

כאשר משתמשים ביותר מאופרטור מורכב אחד מומלץ לשים כל חלק בתנאי בסוגריים כדי שלא יהיו טעויות לוגיות.

**תרגיל**

**צרו מחשבון שמקבל 2 מספרים, מדפיס תפריט שמבקש לבחור איזו פעולה חשבונית לבצע ומקבל את הפעולה לביצוע (חיבור חיסור כפל או חילוק). המחשבון צריך להדפיס את הפלט בצורה מסודרת.**

**הוסיפו למחשבון קליטה של בסיס המספרים (hexadecimal, decimal, octal, binary). פלט החישוב יודפס בצורה שפייתון מייצג את אותו בסיס. לדוגמא, המספר 1101 בבסיס בינארי מיוצג בפייתון בצורה: 0b1101.**

**שימו לב, אין צורך לקלוט את המספרים בבסיס שאינו דצימלי. הבסיס שיתקבל כקלט ישמש רק להגדרת הפורמט שבו יוצג הפלט.**

**Conditional Expression**

שיטה זו מאפשרת לכתוב תנאי בשורה אחרת. Conditional Expression הופך קוד במצבים מסוימים לקריא יותר ונוח לכתיבה למתכנת. אנו שנשתמש בשיטה זו רק כאשר היא מוסיפה לקריאות של הקוד ולא גורעת ממנו.



אם x מחזיר אמת אז ביטוי יחזיר את הערך a ואם לא אז יוחזר הערך b. נוכל לכתוב בביטוי קריאה לפונקציות אך הפונקציה תיקרא רק אם יוחזר הצד שלה בתנאי. לא נוכל להשתמש בפקודות כמו print ב"ביטוי מותנה" אך נוכל להשתמש בprint בצורה הבאה:



# חלק 2 - קבצים

במהלך הל"ע נלמד איך עובדים עם קבצים – איך פותחים וסוגרים קובץ? איך קוראים וכותבים ועוד. ב-Python כל התכנות שלנו יותר נוח ופשוט ולכן גם העבודה עם קבצים ב-Python היא פשוטה.

**פתיחת קובץ**

על מנת לפתוח קובץ, נקרא לפונקציה open:



הפונקציה open מחזירה **אובייקט מסוג קובץ**, כאשר הפרמטרים לפונקציה הם:

* שם הקובץ – יכול להיות שם יחסי, או נתיב מלא.
* סוג הגישה – 'w' – Write, 'r' – Read, 'a' – Append.

**כתיבת לקובץ**

נכתוב לקובץ באמצעות הפונקציה write:

שים לב – כאשר נשתמש ב-write המידע הקיים יידרס ויוחלף במידע החדש. אם נרצה להוסיף למידע הקיים, נשתמש ב-append (**בסוג הגישה)**. לדוגמה:  
  
(שימו לב, שהכתיבה היא אותו הדבר, אך הדבר שהשתנה זה סוג הגישה, ל-'a')

**סגירת קובץ**

בסיום העבודה עם הקובץ, יש לסגור אותו באמצעות הפונקציה close.   
**שים לב** – המידע החדש ירשם רק לאחר שנסגור את הקובץ:



**קריאת מידע מקובץ**

על מנת לקרוא את **כלל** המידע מהקובץ, נשתמש בפונקציה read:



על מנת לקרוא שורה אחת בכל פעם, יש להשתמש בפונקציה readline:



**שימו לב** – readline עוברת על השורות בקובץ באופן אוטומטי. אין צורך **להצביע** לה בכל פעם על השורה הבאה.

**שימוש בwith statement**

נוכל להשתמש בwith כדי לפתוח קבצים בצורה נוח יותר כבלוק בקוד שלנו:



כשנשתמש בwith, כאשר פייתון תצא מהבלוק הנוכחי הקובץ ייסגר אוטומטית - בלי שהיינו צריכים להשתמש בפונקציה close.

**שימוש בנתיבים בקבצים**

כאשר נעבוד עם קבצים, נצטרך לתת כפרמטר את נתיב הקובץ. כידוע, יש לבצע Escaping לתווים מיוחדים בפייתון, שביניהם '\'. לכן – כאשר נרצה לכתוב נתיב של קובץ נצטרך להשתמש ב-'\' **כפול**.  
לדוגמה: C:\\Python27\\Lib\\site-packages.  
אפשרות נוספת ונוחה יותר היא להשתמש ב-'r' לפני מחרוזת (לרוב נתיבים). 'r' הוא סימן ל-Raw string, ונשים אותו **לפני** מחרוזת כדי שפיתון תתייחס למחרוזת מבלי להתחשב בתווים מיוחדים.  
לדוגמה:  
פתחנו את הקובץ מבלי להתייחס לתווים המיוחדים שבו.

* **יש לשים לב שמצב פתיחת הקובץ 'r' והשימוש ב-Raw string 'r' הם דברים שונים.**

**קראו על הפונקציה readlines.**

**תרגול**

1. כתוב תוכנית הקולטת מהמשתמש נתיבים של שני קבצים.  
   התכנית תבדוק האם תוכן הקבצים זהה ותדפיס למשתמש הודעה בהתאם.
2. צרו ידנית שני קבצי טקסט, האחד ריק והשני כולל טקסט כלשהו. כיתבו סקריפט אשר מוגדרים בו שמות שני קבצים. הסקריפט יעתיק את הטקסט אל הקובץ הריק, כך שלאחר סיום הריצה הקבצים יכילו את אותו טקסט.

# חלק 3 – לולאות וטיפוסים מתקדמים

**טיפוסים מתקדמים**

**List**

רשימה היא טיפוס נתונים מיוחד. רשימה מסוגלת להכיל כמה איברים יחד והיא מאפשרת לנו לגשת לכל איבר בתוכה. רשימה בפיתון , בניגוד לרשימות אחרות ברוב השפות, מאפשרת לרשימה לשמור את כל סוגי הנתונים בכל איבר של הרשימה :



כך נגדיר רשימה פשוטה. בדוגמא הראשונה הכנסנו לרשימה רק מספרים אבל אפשר לראות שהרשימה יודעת להתמודד עם כל איבר שנכניס לו גם אם אין איברים זהים כלל. כמובן שהאיברי הרשימה מסודרים אחד אחרי שאני בסדר קבוע. האיבר הראשון נמצא במקום 0 וכך הלאה.

על מנת לגשת לאיבר ספציפי ברשימה נעשה כך :



נוכל לגשת לכל איבר לפי האינדקס(index) שלו ברשימה. אם יש לנו איבר שמכיל רצף(sequence) כמו רשימה או מחרוזת נוכל לגשת גם לכל איבר שלהם בנפרד :



האיבר באינדקס 4 הוא רשימה. כאשר נרצה לגשת לאיבר מתוך הרשימה נפתח עוד סוגריים מרובעים ופיתון תדע שאנחנו רוצים לגשת לאיבר בתוך האיבר. גם מחרוזות אמרנו שהם איבר רצף , ולכן גם בהן אנו יכולים לגשת לאיבר ספציפי.

כאשר ננסה לגשת לאיבר שלא קיים ברשימה כמובן שתקפוץ לנו שגיאה:



**הכנסת\שינו איברים ברשימה**

לאחר שיצרנו את הרשימה אנו יכולים להכניס אליה איברים בשתי דרכים שונות :



בעזרת הפונקציה **append** נכניס איבר לסוף הרשימה כלומר אחרי האיבר האחרון. אך, אם נרצה להכניס את האיבר למקום ספציפי ולא בהכרח לסוף, נוכל לעשות זאת בעזרת **insert**, שמקבלת גם אינדקס שאליו יכנס האיבר ואת האיבר עצמו. Insert תדחוף את כל שאר האיברים ברשימה צעד אחד קדימה.

אז אם אני רוצה לשנות איבר ברשימה ולא להוסיף ? נעשה זאת כך :



**Len**

את len הכרנו כבר במחרוזות, כמובן שהיא עובדת גם פה. נוכל לדעת כמה איברים יש לנו ברשימה:



**מציאת איבר ברשימה**

פיתון מאפשר לנו למצוא האם איבר נמצא ברשימה שלנו בצורה מאוד פשוטה ונוחה :



בעזרת **in** אנחנו יכולים לבדוק אילו איברים קיימים לנו ברשימה. במקום לרוץ על הרשימה ולבדוק in עושה לנו את החיים קלים. ביטוי שנכתוב בעזרת in יחזיר לנו true אם האיבר קיים ו-false אם לא. כמובן שנוכל להכניס את הביטוי לדוגמא לתנאי ובכך להשתמש בו בתוכנית. גם in עובדת על כל אובייקט רצף ולכן תעבוד גם על מחרוזות.

**הסרת איבר מהרשימה**

יש כמה דרכים להסיר איברים מהרשימה :

1)**pop**



בעזרת **pop** אנחנו נשלוף את האיבר האחרון ברשימה וpop גם תחזיר לנו אותו. אופציה נוספת היא לספר לpop אינדקס של איבר וגם כך pop תמחר את האיבר באינדקס שניתן.

2)**remove**

**remove** מקבלת ערך מסוים ומחפשת אותו ברשימה. הערך הראשון שתואם לפרמטר שניתן היא תמחק אותו.remove לא מחזירה כלום.

3)**del**

****

בעזרת **del** אנחנו יכולים למחוק אובייקטים מהזיכרון.del מאפשרת לנו למחוק גם איבר ספציפי של איבריי רצף מהזיכרון.

**Slicing**

פיתון מאפשר לנו עבודה מאוד מאוד נוחה עם אובייקטים מסוג רצף. ונותנת לנו לעבוד איתם באמצעות slicing. בעזרת slicing אני אוכל לחתוך את האובייקט שלי בצורה קלה נוחה ומיוחדת. לדוגמא ,ניצור רשימה של מספרים בעזרת range (עליה נדבר בהמשך) ונעשה עלייה חיתוכים:

את החיתוך נעשה בעזרת סוגרים וביניהם B:A. A יהיה המספר שממנו נרצה להתחיל את החיתוך ו B יהיה המספר שבו יסתיים החיתוך כמובן שנוכל לוותר על אחד מהם ואז החיתוך יהיה מהתחלה או עד הסוף. אופציה נוספת שיש לנו עם slicing היא לחתוך את האובייקט עם קפיצות על אינדקסים כך :

הוספנו עוד נקודתיים ואחריהם יגיע מספר הקפיצות שנעשה בין איבר לאיבר. בדוגמא , התחלנו באיזה הראשון כי לא ציינו כלום , עד האיבר האחרון כי גם בו לא ציינו כלום ובקפיצות של 3.

**תרגול :**

**תגדירו רשימה עם range(20) מצאו בא את הפרטים הבאים בעזרת חיתוכים:**

* **כל האיברים הזוגיים.**
* **כל האיברים האי-זוגיים.**
* **הצגה של האיברים בקפיצות של 3.**
* **האיבר האחרון ברשימה (2 דרכים).**
* **הרשימה בסדר הפוך.**
* **כל האיברים עד האיבר ה-6.**

**Mutable & Immutable**

בפיתון , המשתנים שלנו לא באמת מכילים את האובייקט שאותו אנו מיישמים להם, אלא הוא מכיל מצביע לזיכרון שבו האובייקט שמור.אז מה שיש לנו כרגע זה את a שהוא מצביע בזיכרון למקום שבו המספר 10 שמור וכך גם b מכיל מצביע לזיכרון למקום שבו המספר 15 שמור.

|  |
| --- |
| a |
| b |

|  |
| --- |
| 400 |
| 415 |

בפיתון, יש לנו 2 סוגים שונים של משתנים והם ה-mutables וה-immutables. בואו נבין קודם אילו משתנים שייכים לנו לכל קבוצה ולאחר מכן מה ההבדל בין הקבוצות הללו.

המשתנים : int, float, long, tuple, string הם משתנים הנחשבים "מורכבים" ומכונים **immutable**.

המשתנים: list, set, dict מכונים **mutable**.

**ההבדל העיקרי בין 2 הסוגים הנ"ל הוא שמרגע ההשמה לזיכרון immutable לא מסוגל להשתנות לעולם ואילו mutable הוא דינמי ומסוגל להשתנות בזיכרון בכל רגע נתון.**

על מנת להבין את הנושא הזה יותר טוב, נשתמש במתודות id ו is .

**id –** פונקציה המציגה לנו את המיקום בזיכרון אליו מצביע המצביע. המיקום יהיה מוצג כמספר. אותו מספר הוא לא באמת המיקום של האובייקט בram אלא הוא פשוט דרך ייצוג של פיתון.

**is –** אופרטור שמשווה בין 2 id-ים של משתנים ומחזירה True במידה והם שווים ו False במידה ולא.



במקרה הזה יש לנו את a ו b ששניהם immutable כי הם מכילים מספרים. כמובן שהשניים לא מצביעים לאותו מיקום בזיכרון מסיבה שהם מכילים ערכים שונים ולכן כל אחד מצביע למקום אחר.



אז מה קרה פה? אמרנו ש immutable הם לעולם לא משתנים. תחילה, ישמנו ל a את המספר 25 ובדקנו מה המקום שלו בזיכרון. לאחר מכן הוספנו 4 ל 25. במקום שהמיקום בזיכרון שאליו מצביע a פשוט ישתנה מ25 ל29. פיתון יוצרת מיקום חדש בזיכרון אליו היא משימה 29 ומשנה גם את המצביע של a אליו. זהו העיקרון המרכזי ביותר ב immutables בואו נראה מה קורה אם הקבוצה השנייה:

במקרה הזה יצרנו רשימה, שרשימה בניגוד למספר היא כבר שייכת ל mutables ולכן היא מסוגלת להשתנות במקום שלה בזיכרון. כמו שניתן לראות גם לאחר ששינינו אותה היא עדיין נמצאת באותו מקום בזיכרון

|  |
| --- |
| a |
| b |

|  |
| --- |
| 3 |
|  |

על מנת להבין את המקרה הזה, נעזר בסרטוט. תחילה השמנו 3 ל a ואמרנו ל b = a. בפועל זה מה שקרה :

כרגע גם a וגם b מצביעים על 3 ולא b מצביע על a שמבציע על 3. ברגע שאמרנו b=a בעצם אמרנו לפיתון שb יצביע על המיקום בזיכרון ש a מבציע איליו. עכשיו מפני שאמרנו שמספרים הם immutable לאחר שהוספנו 2 ל a , אז כמו בדוגמא למעלה פיתון תפתח מיקום חדש בזיכרון ותכניס אליו 5(כלומר 2 ועוד 3),בנוסף לכך ישנה המצביע של a אל מיקום הזיכרון החדש. מכיוון שb- מצביע על המקום בזיכרון הישן של a ולא על a עצמו, b לא ישתנה וזה יראה כך :

|  |
| --- |
| 3 |
| 5 |

|  |
| --- |
| a |
| b |

כמובן אם נעשה את אותו הדבר לרשימה גם a וגם b ישתנו :

זאת קורה בדיוק בגלל מה שאמרנו שרשימה היא Mutable והיא מסוגלת להשתנות כל הזמן אז במקרה הזה a מצביעה על רשימה מסיומת ועלייה מצביע גם b וכשאנו משנים אותה גם a וגם b משתנים מפני שהם מצביעים על אותה רשימה שמשתנה.

יש כאן אבל בעיה קטנה, מה יקרה שאני ירצה להעתיק רשימה? או כל איבר אחר שהוא mutable? בשביל זה נשתמש ב slicing שלמדנו קודם לכן. אמרנו ש slicingיודעת להחזיר לנו **עותק** של רשימה עם חיתוכים שונים :



אם נשתמש בslicing מהמקום הראשון עד לאחרון הכוונה כך : [:] למעשה נעתיק את הרשימה במלואה ונוכל להכניס אותו למשתנה אחר.

**Tuple**

Tuple הוא טיפוס נתונים שמאוד מזכיר רשימה אך בניגוד לרשימה tuple לא יכול להשתנות לעולם. זאת אומרת מרגע שהגדרנו אותו לא נוכל לעשות בו שינויים. על מנת להגדיר tuple נשתמש בסוגריים עגולים ().



העבודה עם tuples היא זהה כמו עם רשימות. כמובן שאין לנו את כלל הפונקציות של רשימה אך עדיין נוכל להשתמש בslicing ובפונקציות המובנות של tuple שהם index ו count. Index מקבל פרמטר וכפלט מחזיר כמה פעמים הפרמטר שהכנסנו נמצא בtuple וindex יחזיר לנו את הindex הראשון של הערך שהוא מקבל כפרמטר.



בנוסף ניתן לחבר tuple-ים בדיוק כמו עם רשימות :



**Tuple Assignment**

כאשר מבצעים השמה בין 2 tuple-ים פיתון מבצעת השמה אוטומטית בין כל 2 איברים תואמים כך :

בגלל שפיתון מכירה את השיטה הזו לא נהיה חייבים להשתמש בסוגריים.



**Set**

**Set** (קבוצה) דומה ל**רשימה** (list), מלבד העובדה ש**לא** ניתן לאחסן **פעמיים** את אותו האיבר. Set הוא טיפוס **שאינו מסודר** – כלומר, איבר שאנחנו מכניסים אליו לאו דווקא יכנס ל-Set בתור האיבר האחרון.

1. קראו על טיפוס זה ב-Docs בנתיב הבא:

**The Python Tutorial\Data Structures\Sets**

**שימו לב** – התנסו בכל הדברים שאתם קוראים עליהם. חקרו מקרים שונים על מנת להבין בדיוק כיצד set עובד.

1. קראו ב-**help** על הפונקציות הבאות של הטיפוס Set ו**התנסו** בהן:
   * Add.
   * Discard.
   * Remove. (מה ההבדל בין Discard ל-Remove?)
   * Clear.
   * Pop.
   * Difference. (שימו לב – הפונקציה מקבלת Set אחד או יותר, בתור קלט)
   * Intersection. (אותו כנ"ל)
   * Union. (נכון, גם פה(
   * קיימות כמובן עוד פונקציות, תוכלו לקרוא גם עליהן.
2. אם נרצה, נוכל גם ליצור קבוצה מרשימה או מ-Tuple **קיים**, וכך לקבל את אותה רשימה, **ללא** כפילויות:



**לולאת While**

**מבנה לולאה**

לולאה – רצף פקודות המתבצע מספר פעמים תוך התחשבות בביטוי לוגי.

בלולאת while, כל עוד התנאי הלוגי מחזיר אמת, הקוד בתוך הלולאה מתבצע. נשתמש בלולאת while לקוד שצריך לחזור על עצמו מספר לא ידוע של פעמים, הפסקת הלולאה תיקבע ע"י תנאי.

למשל בדוגמא הבאה ישנו קובץ שאנו רוצים לקרוא. הבעיה היא שאנחנו לא יודעים כמה שורות יש בקובץ אך אנחנו יודעים שכשהפונקציה readline() מגיעה לשורה האחרונה היא מחזירה מחרוזת ריקה. בכדי לבצע זאת נכתוב את הקוד הבא:

כמובן שיש שיטות יותר טובות לקרוא קובץ ובשיטה זו לא נשתמש.

**דגשים ושגיאות בשימוש בלולאות**

* מינימום פעמים שלולאה תתבצע – 0.
* מקסימום פעמים – אינסוף, מצב זה נקרא לולאה אינסופית. ברוב המקרים לא נרצה להגיע למצב הזה – זוהי שגיאה. במקומות בהם נרצה לולאה אין סופית, כמו שרת שמקבל מידע, נכתוב בתוך הלולאה מנגנון שמאפשר לעצור אותה מקטע קוד חיצוני.
* יש לדאוג לאתחל את המשתנים לפני הלולאה, ולדאוג לשינוי של ערך המשתנים בתוכה.

**משתנה flag**

משתנה דגל הינו משתנה אשר מתאר מצב מסוים ונשתמש בו בכדי לשלוט בלולאה. כאשר מצב מסוים יתקיים "נרים" את הדגל והלולאה תיפסק. שם המשתנה יציין את משמעותו וינוסח באופן המתאר קיום או אי קיום מצב מסוים.

הדוגמא הבאה מנסה למצוא סיסמא מסוימת ע"י מעבר על כל האפשרויות ומציגה את התחביר של פייתון ברוב תפארתו.

כל עוד הסיסמה היא לא “123456” תמשיך להוסיף 1 לסיסמה שאנחנו מנסים.

**שימו לב** – לא נשתמש במשתנה Flag כמו שנשתמש בשפות עיליות אחרות (#C, ++C וכן הלאה). בפייתון נמנע להשתמש במשתנה Flag של True או False כאשר תנאי כלשהו מצליח. דוגמה שלילית:



נשתדל שלרוב תנאי היציאה יהיה בהגדרת הלולאה.

**לולאה מקוננת**

תרגיל

כתבו תוכנית המציגה את לוח הכפל, השתמשו בפונקציה ljust כדי להציג את לוח הכפל בצורה מסודרת. רשמו הפקודה הבאה:



**לולאת זקיף**

לפעמים לא נוכל לדעת מראש באיזה שלב צריך לעצור את הלולאה. לכן, נוכל להשתמש בזקיף – ערך המהווה ללולאה סימן לעצור.

דוגמא:



**תרגיל**

**עליכם לכתוב תוכנית אשר קולטת עבור כל לקוח של הפיתייה את המידה בה הוא מרוצה מהאוכל המעולה שלו. (מספר בין 1 ל – 10, כאשר 1 זה עבור לקוח שבכלל לא מרוצה ו – 10 עבור לקוח שממש מרוצה).  
הקלט יתבצע עד שייקלט הזקיף exit המציין את סוף הלקוחות.  
התוכנית תדפיס את התפלגות מידת הרצון מהשירות לפי הקטגוריות הבאות:  
 א. כמות הלקוחות המרוצים (8 ומעלה).  
 ב. כמות הלקוחות שבעי הרצון (5-7).  
 ג. כמות הלקוחות שלא מרוצים (4 ומטה).**

**לולאת For**

**מבנה לולאת for**

אנו נשתמש בלולאה for לריצה על מספר ידוע של איברים. בפייתון לולאת for בנויה בצורה שהטבע התכוון אליה, אנו רצים על משתנים מסוג רצף (sequence) ולא לפי תנאי לוגי או אינדקסים.

הלולאה מקבלת רצף ורצה עליו, בכל איטרציה הלולאה מתייחסת רק לאיבר אחד ברצף.

איטרציה (iteration) – חזרה, ריצה אחת על הלולאה.

בדוגמא הבאה יודפסו לנו ימות השבוע כאשר כל יום יהיה בשורה חדשה:



בעצם מה שלולאת for עושה זה לקחת בכל ריצה איבר לפי הסדר (מהראשון לאחרון) של הרצף (days) ולשים את האיבר במשתנה שאנו נותנים ללולאה (day).

**break, continue**

קיימים לנו 2 statements אשר עוזרים לנו לשלוט בפעולות הלולאה והם continue ו-break.

continue – פקודה הגורמת ללולאה להמשיך לאיטרציה הבאה מבלי להריץ את הקוד שבא אחרי הפקודה.

break – מפסיק את ההרצה של הלולאה ומיד יוצא החוצה מבלי לסיים את האיטרציה.



**range**

הפונקציה הבאה תשמש אותנו הרבה בלולאות for בכדי ליצור רשימה או אובייקט אחר שנרוץ עליו.

range – הפונקציה מוגדרת בצורה range([start], stop[, step]) זאת אומרת שיש לה 2 פרמטרים אופציונאלים (start, step) ומשתנה אחד חובה (stop). פונקציה זאת מחזירה אובייקט אשר שומר בזיכרון איבר התחלתי סוף והפרש. האובייקט משתמש בנתונים אלו בכדי לחשב את האיבר הבא של הלולאה והוא אינו שומר רשימה בזיכרון. פונקציה זו מחזירה איבר מסוג “range” שניתן להפוך אותו לרשימה, או לרוץ עליו כמו שהוא.



**שגיאות נפוצות**

כל השגיאות הנפוצות בלולאת תנאי תקיפות גם כאן. שגיאות נוספות:

* רישום ערך התחלתי גדול מערך סופי בלולאת מונה בקרה עולה, הלולאה לא תתבצע אפילו פעם אחת.
* רישום ערך התחלתי קטן מערך סופי בלולאת מונה בקרה, הלולאה לא תתבצע אפילו פעם אחת.
* שינוי ערך משתנה הבקרה בתוך הלולאה. (כמו day בדוגמא הראשונה)

**תרגיל**

**קלוט מספר והדפס פלט בצורה הבאה (הדוגמא מתייחסת לקלט 5):**

**1 \* 1 2 1 \* 1 2 3 2 1 \* 1 2 3 4 3 2 1 \* 1 2 3 4 5 4 3 2 1 \* 1 2 3 4 3 2 1 \* 1 2 3 2 1 \* 1 2 1 \* 1 \***

# חלק 4 – לולאות וטיפוסים מתקדמים

מילון (Dictionary) ב-Python הוא טיפוס שמטרתו היא המרה בין מפתחות לבין ערכים.

מילון מחזיק מפתחות, ערכים ואת המיפוי ביניהם. כאשר המשתמש פונה למילון, הוא מציין מפתח – ומקבל עבורו ערך.

**הצורך במילון:**

מילון מאפשר לנו לגשת לערכים לא בעזרת מיקומים(index) ברשימה, אלא באמצעות כל דבר שנבחר, נוכל לפנות אליהם בעזרת אותיות בודדות, מחרוזות וגם מספרים לבחירתנו.

מה הכוונה?

אם עד היום התרגלנו לפנות למשתנים השמורים ברצף (למשל רשימה) לפי האינדקס המספרים שלהם

למשל:

List = ['a','b','c']

על מנת לפנות לאותיות נצטרך לפנות למיקום שלהם.

למשל כדי לפנות לאות 'a' נצטרך לפנות אליה כך: List[0]

כעת בעזרת מילונים נוכל להגדיר איך נרצה לפנות אליהם

**הגדרת מילון:**

מילון נגדיר בעזרת סוגריים מסולסלים:

example **=** **{**'bob' **:** 10**,** 'bill' **:** 20**,** 'alice' **:** 3**0}**

המילון שלנו הוא example

על מנת להגדיר צמד של מפתח וערך נשתמש בנקודותיים.

מצד שמאל נמצא המפתח ומצד ימין נמצא הערך.

לאחר הגדרת המילון אם נרצה לפנות לערך מסוים נשתמש במפתח שהוגדר לו.

כלומר, אם נרצה לפנות למספר 10 נשתמש ב'bob' בתור אינדקס.

example['bob']

כך לא נצטרך להתעסק עם מיקומים אלא נוכל לפנות בעזרת כל דבר שנגדיר כמפתח.

דגש חשוב: מפתח חייב להיות immutable

**הוספת מפתחות וערכים:**

על מנת להוסיף ערכים נשתמש בסוגריים מרובעים:

Example['morris'] = 40

כעת נוכל לפנות אל הערך 40 עם המפתח morris

זוהי הדרך שגם נשנה ערכים, כלומר אם המפתח morris היה קיים, אז הערך אליו ניתן לפנות עם המפתח morris היה משתנה ל40.

**מחיקת ערכים:**

על מנת למחוק ערכים נשתמש בdel ונכתוב את המילון המפתח המתאים.

del example['bob']

כעת הערך 1 נמחק ולא נוכל לפנות אליו יותר עם 'bob'

**פונקציות שימושיות:**

ניתן לבדוק האם מפתח קיים במילון.in בעזרת

דגש: זה עובד רק עם מפתח ולא עם ערכים.

'Japan' **in** example

False

'bob' **in** example

True

**items**

מחזיר אובייקט שניתן להפוך לרשימה של tuples שמכילה את המפתחות והערכים בזוגות.

x = example**.**items**()**list(x)

**[(**'bob'**,** 1**),** **(**'bill'**,** 2**),** **(**'alice'**,** 3**)]**

**keys**

מחזירה אובייקט שניתן להפוך לרשימה של כל המפתחות.

x = example**.**keys**()**

list(x)

**[**'bob'**,** 'bill'**,** 'alice'**]**

**values**

מחזירה אובייקט שניתן להפוך רשימה של כל הערכים.

x = example**.**values**()**

list(x)

**[**1**,** 2**,** 3**]**

**update**

מקבלת מילון ומשרשרת אותו אל המילון הקיים(מוסיפה את כל המפתחות והערכים של המילון הנוסף בהתאמה).

example**.**update**({**'hello' : 1**,** 'world' : 2**})**

{'bob' : 10, 'bill' : 20, 'alice' : 30,'hello' : 1 , 'world': 2}

ואם פשוט נדפיס את המילון הוא יראה כך:

example

{'world': 2, 'bob': 10, 'bill': 20, 'alice': 30, 'hello': 1}

**סידור מילון:**

מצאו איך לסדר מילון לפי הערכים שלו, וצרו מילון שהמפתחות שלו הם שמות החניכים בכיתה והערכים הם מספרי החניך, ומיינו אותו לפי מספר חניך.

**אקסטרה:**

* קראו את הPDF הבא כדי להבין איך מילון נשמר מאחורי הקלעים - <https://www.laurentluce.com/posts/python-dictionary-implementation/>
* לאחר שקראתם, ישנם כמה שינויים מאז כתיבת המאמר הזה שנכנסו בפייתון 3.6  
  קראו עליהם כאן:  
  <https://mail.python.org/pipermail/python-dev/2012-December/123028.html>