

תכנות בשפה פיתון

ברק גonen

המרכז לחינוך סייבר
CYBER EDUCATION CENTER

תכנות בשפת פיתון

Python Programming / Barak Gonen

גרסה 3.0 מרץ 2021

כתיבה:

ברק גונן

עריכה:

עומר רוזנבוים

אין לשכפל, להעתיק, לצלם, להקליט, לתרגם, לאחסן במאגר מידע, לשדר או לקלוט בכל דרך או אמצעי אלקטרוני, אופטי או מכני או אחר – כל חלק שהוא מהחומר שבספר זה. שימוש מסחרי מכל סוג שהוא בחומר הכלול בספר זה אסור בהחלט, אלא ברשות מפורשת בכתב מהמרכז לחינוך סייבר, קרן רש"י.

מהדורה ראשונה תשע"ז 2017

מהדורה שנייה תש"פ 2020

מהדורה שלישית תשפ"א 2021

© כל הזכויות שמורות למרכז לחינוך סייבר של קרן רש"י.

<http://www.cyber.org.il>

תוכן עניינים

7	הקדמה והתקנות נדרשות
7	התקנות נדרשות
17	איקונים
17	תודות
18	תוכן עניינים מצגות פיתון
19	פרק 1 – מבוא ללימוד פיתון
19	מהו שפת סקריפטים?
22	עבודה באמצעות command line
23	מספרים בבסיסים שונים
25	Help
26	הסימן _ (קו תחתון)
27	הרצה תוכניות פיתון דרך ה-command line
28	סיכום
29	פרק 2 – סביבת עבודה PyCharm
29	פתיחת קובץ פיתון
31	קובץ הפיתון הראשון שלנו
32	סדר ההרצה של פקודות בסקריפט פיתון
34	התרעה על שגיאות
36	קביעת פרשן (Interpreter)
37	הרצה הסקריפט ומסך המשתמש
39	דיבוג עם PyCharm
41	העברת פרמטרים לסקריפט
41	סיכום
42	פרק 3 – משתנים, תנאים ולולאות
42	סוגי משתנים בפייתון
45	תנאים

46	תנאים מורכבים
47	שימוש ב-is
47	בלוק
49	תנאי elif
50	לולאה while
52	לולאות for
54pass
56	פרק 4 – מהרווזות
56	הגדרת מהרווזות
57	ביצוע ל-format
58	חיתוך מהרווזות – string slicing
60	פקודות על מהרווזות
61	dir, help
62	צירופי תווים מיוחדים ו-string
63	קבלת קלט מהמשתמש
67	פרק 5 – פונקציות
67	כתיבת פונקציה בפייטון
69	return
70	None
71	scope של משתנים
77	פייטון מתחת למכסה המנווע (הרחבה)
78id, is
80	העברה פרמטרים לפונקציה
82	סיכום
83	פרק 6 – List, Tuple
83	הגדרת List
85	Mutable, immutable

87	פועלות על רשימות
87	in
87	append
88	pop
88	sort
90	split
91	join
92	Tuple
93	סיכום
95	פרק 7 – כתיבת קוד נכונה
96	PEP8
99	חלוקת קוד לפונקציות
101	פתרון מודרך
106	assert
110	סיכום
111	פרק 8 – קבצים ופרמטרים לסקריפטים
111	פתיחת קובץ
112	קריאה מקובץ
113	כתיבה לקובץ
113	סגירת קובץ
115	קבלת פרמטרים לתוכנית
119	סיכום
120	פרק 9 – Exceptions
121	try, except
123	סוגים של Exceptions
126	finally
129	with

135	פרק 10 – תכנון מונחה עצמים – OOP
135	מבוא – למה OOP?
136	אובייקט – object
137	מחלקה – class
139	כתיבה class בסיסי
140	<code>__init__</code>
140	הוספת מתודות
141	Members
142	יצירת אובייקט
144	כתיבת class משופר
144	יצירת members "מוסתרים"
147	שימוש ב-accessor ו- <code>mutator</code>
149	יצירת מודולים ושימוש ב-import
152	אתחול של פרמטרים
152	קביעת ערך ברירת מחדל
153	<code>__str__</code> המתודה
154	יצירת אובייקטים מרובים
156	ירושה – inheritance
160	פולימורפיזם
162	<code>isinstance</code> הפונקציה
165	פרק 11 – OOP מתקדם (תכנות משחקים באמצעות PyGame)
166	כתיבת שלד של PyGame
167	שינוי רקע
172	הוספת צורות
174	תזוזה של גרפייקה
177	ציור Sprite
179	קבלת קלט מהעכבר

183	קבלת קלט מהמקלדת
183	השמעת צלילים
186	OOP מתקדם – שילוב PyGame
186	מבוא.
187	הגדרת class
188	הוספה מתודות mutators-1 accessors שימושיות
189	הגדרת אובייקטים בחוכנות הראשית
190	sprite.Group()
191	יצירת אובייקטים חדשים
192	הוזת האובייקטים
194	בדיקות התנגשויות
199	סיכום
200	פרק 12 – מיליוןים
201	Get, in, pop, keys, values
203	מילונים, מתחת למכסה המנווע (הרחבה)
205	סוגי מפתחות
206	סיכום
207	Magic Functions, List Comprehensions – 13
207	List Comprehensions
210	Lambda
211	Map
212	Filter
213	Reduce
213	סיכום

הקדמה והתקנות נדרשות

ברוכים הבאים לשפט פיתון! אם אתם קוראים ספר זה כנראה yourselves את צעדיכם הראשונים בмагמת הגנת סייבר. מדוע לומדים דוחק פיתון? שפט פיתון נבחרה ללימוד כיוון שהיא נמצאת בשימוש רחב בתעשייה, באקדמיה וגם בקרב הichidot הטכנולוגיות בצה"ל. חומר הלימוד הבאים של מגמת הסייבר מבוססים על שפט פיתון וכן נדרשת שליטה בשפה בסיסי ללימוד יתר התכנים.

הספר כולל את החומר הנדרש בסיסי ללימוד רשות מחשבים, מומלץ ללמידה מן הספר נושאים לפי הצורך – לדוגמה, סביבת העבודה, שימוש במחרוזות ורשימות נדרשים ללימוד רשות וכן יש ללמידה אוטם תחילתה. לעומת זאת, התקנות מונחה עצמים ניתן לדוחות את הלימוד של נושא זה להמשך. בנוסף כולל הספר נושאי הרחבה מעוניינים לשלוט בשפה יותר לעומק.

הספר לא מניח כמעט ידע מוקדם, אך הוא מניח היכרות עם רעיונות בסיסיים בתכנות כגון משתנים, תנאים ולולאות. הוא מועד לאפשר למידה עצמאו, והוא פרקטטי וככל תרגילים רבים. כדי לשלוט בשפט פיתון, כמו בכל שפת תכנות, אי אפשר להסתפק במידע תיאורטי. הספר אינו מתימר לכטוט את כל הנושאים בשפט פיתון, אלא להתמקד בunosaim שסביר שתזדקקו להם במהלך לימודיהם. אחד הנושאים עליהם הספר אינו מרחב רבות הוא אודות השימוש במודולים, ספריות שפותחו בפייתון ומאפשרות לבצע פעולות מורכבות בקלות יחסית. הוויתור על ההסברים המפורטים אודות מודולים הוא ראשית מכיוון שקשה מאוד להקיף את כל המודולים החשובים בספר למידה, ושנית מכיוון שמטרתנו היא להעניק לכם את הבסיס ללמידה עצמי של חומרים מתקדמים. כך, כאשר תסימנו את הלימוד מהספר, תוכל בקלות למצוא מידע באינטרנט אודות כל נושא שתרצו וילשלב אותו בהצלחה בתוכנה שאתם כותבים.

התקנות נדרשות

כאמור ספר הלימוד מבוסס על שפט פיתון. ישן התקנות רבות של פיתון, لكن נרצה להמליץ על סביבת עבודה ושימוש נכון בסביבת העבודה. להלן פירוט כלל התקנות הנדרשות להימוד פיתון והן ללמידה רשות מחשבים. גרסת הפיתון של התקינה היא 3.8. למשוניים ללמידה רק פיתון, ללא רשות, אפשר לוותר על התקנת Wireshark, Scapy ו-npcap.

שימוש לב:

התוכנות הן עבור מערכת ההפעלה Windows10, עבור מערכות הפעלה אחרות ישן גרסאות ספציפיות של התקנות ויש להוריד אותן באופן עצמאי. יתר התקינה צפוי להיות דומה.

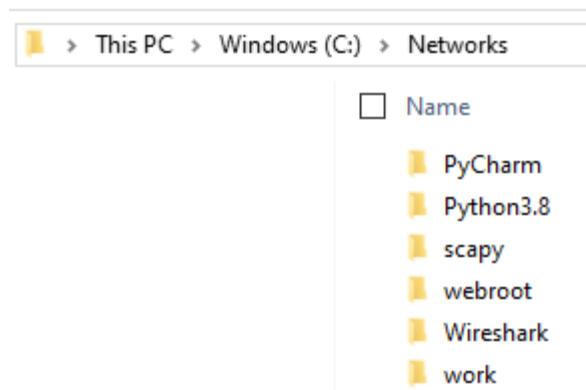
התוכנות הנח (לפי סדר ההתקנה):

1. **פייתון גרסה 3.8.0** – נדרש ללימוד פייתון
2. **סביבת פיתוח PyCharm, גרסה 2019.2.5** – נדרש ללימוד פייתון
3. **דריבר הסנפה מכרטיס רשת, Npcap גרסה 0.9984** – נדרש ללימוד רשתות, לומדי פייתון בלבד יכולים לוותר
4. **תוכנת הסנפה Wireshark גרסה 3.0.6** – נדרש ללימוד רשתות, לומדי פייתון בלבד יכולים לוותר
5. **מודול יצירת פקודות של פייתון, Scapy גרסה 2.4.3** – נדרש ללימוד רשתות, לומדי פייתון בלבד יכולים לוותר

קובץ התקינה מרוכז נמצא בכתובת:

<https://data.cyber.org.il/networks/install.zip>

אפשר להתקין את התוכנות בכל תקיה, אך מושווין רב שנים התקינה "מפוזרת" של התוכנות גורמת אינספור בעיות לסטודנטים. לדוגמה, חלק מהתוכנות לא מסוגלות לzechot תקיות שיש בהן עברית. חלק מהתוכנות דורשות התאמה של משתני סביבה. אם תפעלו לפני ההוראות הבאות, תחסכו לעצמכם בעיות. שימושם לב למבנה התקיות המומלץ בסיום התקנות:



התקיה הראשית היא Networks, שם נשים את כל תת התקיות. ישן חמיש תקיות המיעדות להתקנות ותיקיה נוספת בשם Work המיעדת לקבצי הפייתון שלהם.

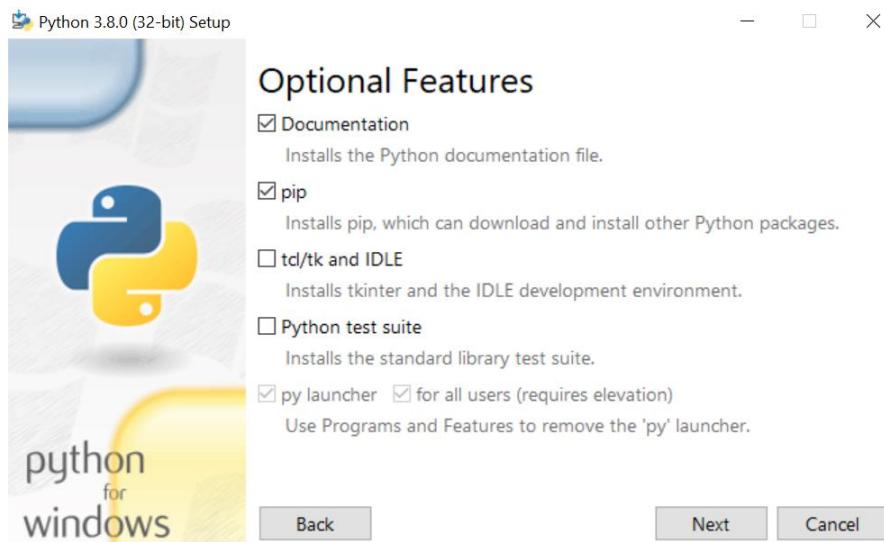
א. התקנת פיתון

הקליקו על הקובץ python-3.8.0.exe, יופיע מסך ההתקנה הבא:



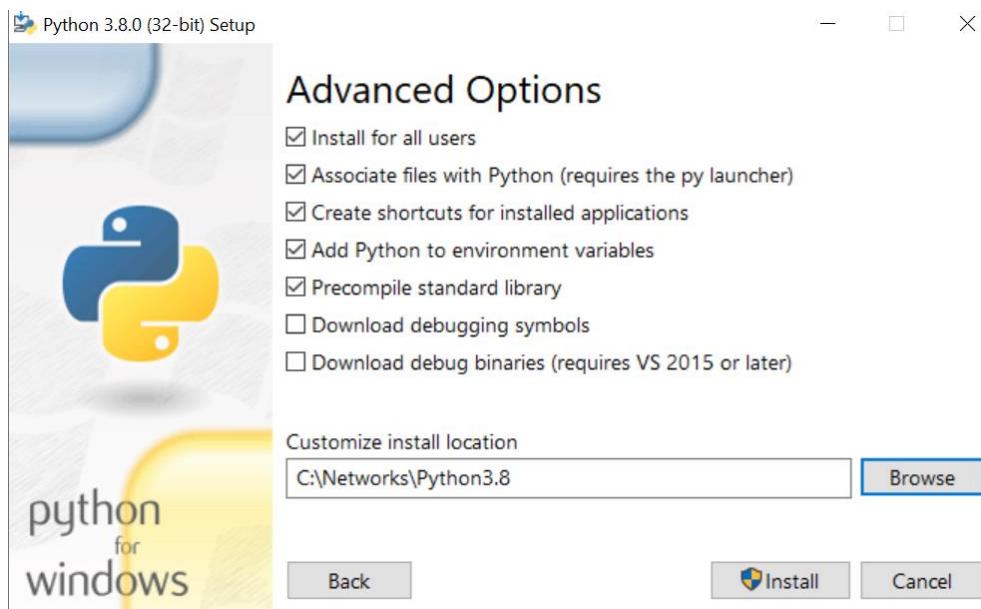
בחרו באפשרות "customize installation".

בחרו באפשרויות הבאות:

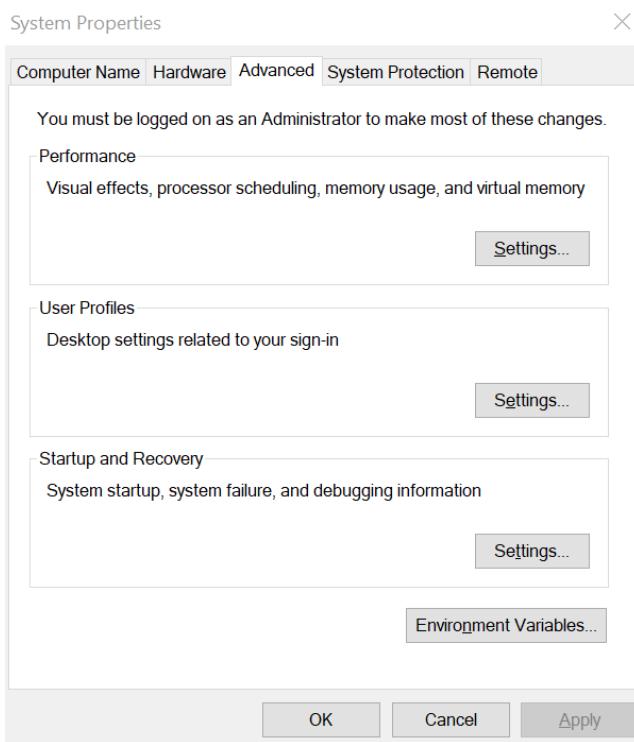


הקדמה והתקנות נדרשות

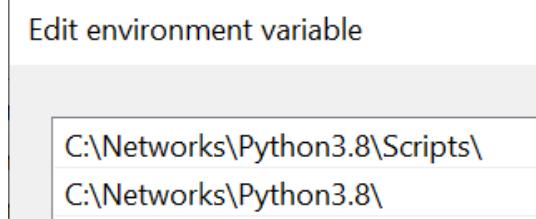
ובמוך הבא בחרו את האפשרויות המסומנות, והזינו בהתאם תקינות התקינה את 3.8 Python:\Networks\Python3.8



כעת בידקו שפייתו מוגדר בתוך ה-PATH של משתני הסביבה שלכם. במסך החיפוש של windows הקלידו env .Environment Variables על הלחצן .Edit The System Environment Variables ובחירה באפשרות



וודאו שבתוך המשתנה PATH יש את שתי הכוויניות הבאות. אם אין, הוסיפו אותן ידנית (בניהלה שהתקנותם את פיריטון בתוך (c:\networks\python3.8\



כדי לוודא שהכל עובד כמורה, פיתחו cmd והקלידו python. צפוי שתקבלו את ההדפסה הבאה:

A screenshot of a Windows command prompt window titled "cmd C:\WINDOWS\system32\cmd.exe - python". The window displays the following text:

```
C:\Windows [Version 10.0.18362.476]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\barak>python
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:21:23) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The window has a standard Windows title bar with minimize, maximize, and close buttons.

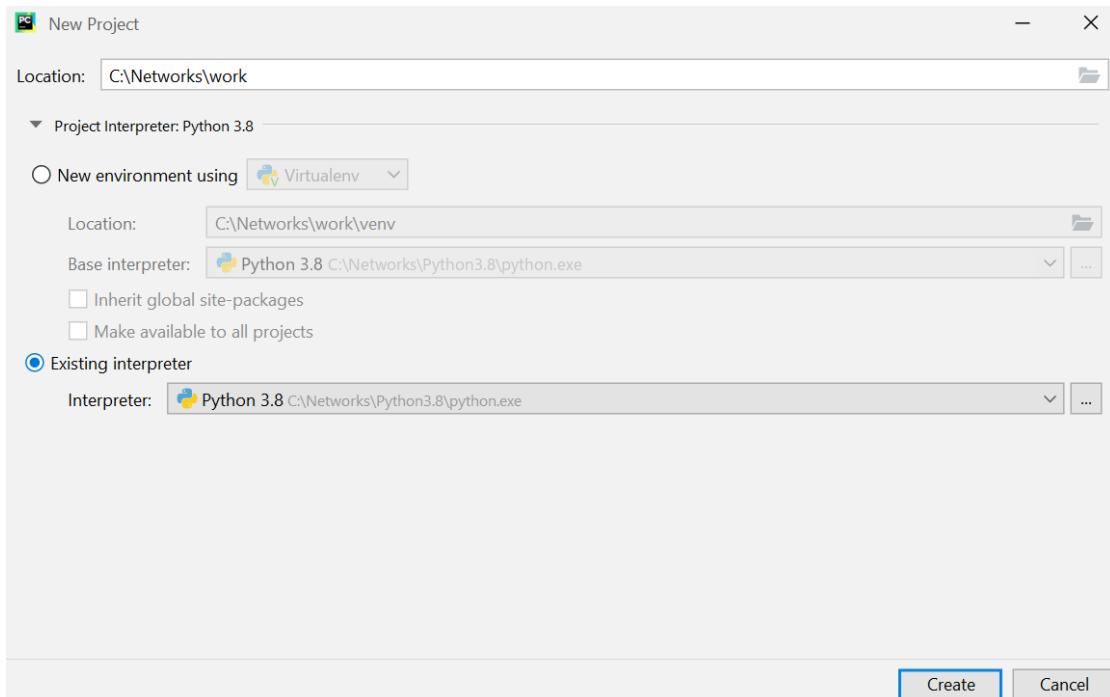
ב. PyCharm

הקליקו על קובץ ההתקינה, היא תבוצע עצמה.

עם הפעלה, בחרו באפשרות "Create new project" ופתחו פרויקט בספריה work\work\c. לצורך להגדר היכן נמצא python interpreter שלכם, בצעו זאת כך:

- בחרו באפשרות Existing Interpreter

- כאשר תקליקו על הלחץ שיש עליו ציר של שלוש נקודות, יפתח לפניכם מבנה התיקיות במחשב שלכם.
הגיעו אל התיקייה python.exe\Python3.8\python.exe ובחרו את הקובץ networks\Python3.8\python.exe

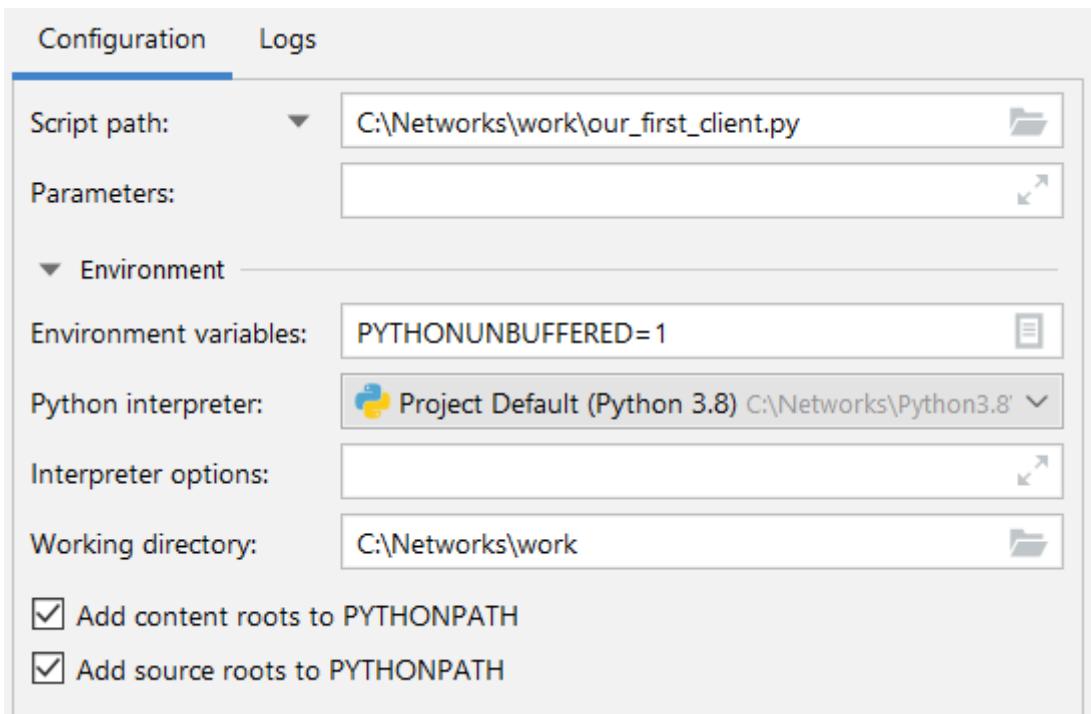


אפשרות נוספת היא להגדיר את ה- Interpreter בעצמכם, עברו כל קובץ אותו אתם מרצו. מהי נכוון לעשות זאת? אם קיבלתם הודעה שגיאה "Invalid Python Interpreter selected for the project" ("ה הודעה מופיעה שורות הקוד של הקובץ אותו אתם מנוטים להריץ:



הקליקו על "Configure Python Interpreter" בצד ימני של ההודעה ועברו למסך ההגדרות הבא:

הקדמה והתקנות נדרשות



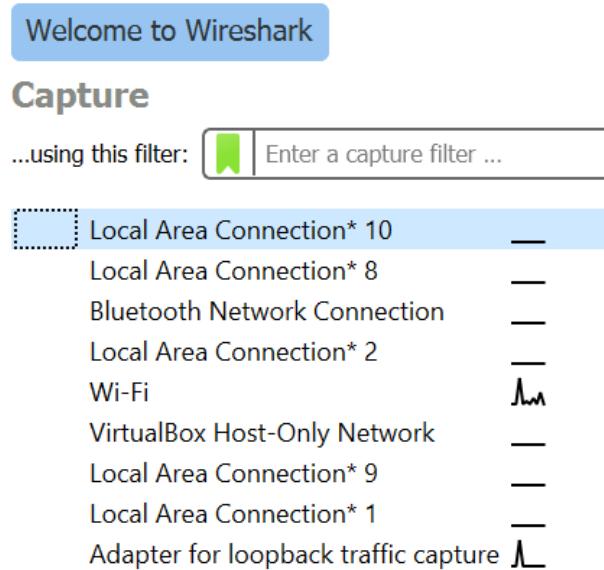
שנו את הערך של ה-*Python interpreter* כך שיצביע על הקובץ *exe*.exe שבתיקיה
כעת הקיליקו על כפתור ה-OK והבעיה נעלמה.

ג. npcap ו-Wireshark

לומדים פיתון בלבד אין צורך להתקין את התוכנות הללו.

הקליקו על תוכנת ההתקינה של Wireshark. התוכנה תשאל אם ברצונכם להתקין על הדרכם npcap, בחרו באפשרות הזו.

הקליקו על האיקון של Wireshark. התוכנה תגלה באופן אוטומטי את כל ממשק הרשות שלכם. כרטיס רשת פועל ראה גוף עולה ויורד, בהתאם לרמת הפעולות.



במקרה זה, המחשב מחובר דרך Wi-Fi. בחרו במשק הפעיל והתחלו הסנפה.

scapy.

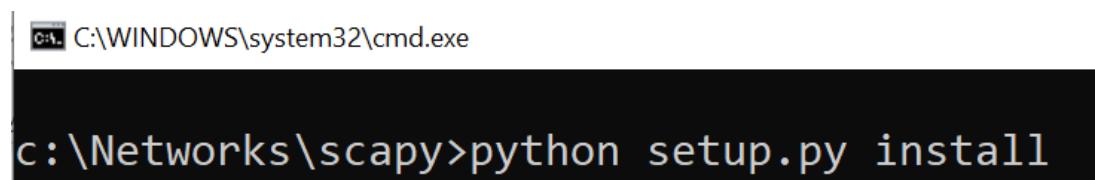
לומדים פיתון בלבד- אין צורך להתקין את התוכנה זהו.

העתיקו את תוכן של התקינה scapy-master מקובץ ה-zip אל תיקיית .c:\networks\scapy

פיתחו cmd והקלידו

```
cd c:\networks\scapy
```

כדי להתקין הקליידו python setup.py install



```
C:\WINDOWS\system32\cmd.exe
c:\Networks\scapy>python setup.py install
```

כדי לבדוק אם התקינה הצלחה, הקלידו scapy. קיבלנו את המסר הבא, אין מה להיות מוטרדים מהודעות השגיאה- הן שויות למודולים שאין לנו צורך בהם.

הקדמה והתקנות נדרשות

```
C:\WINDOWS\system32\cmd.exe - scapy
c:\Networks\scapy>scapy
INFO: Can't import matplotlib. Won't be able to plot.
INFO: Can't import PyX. Won't be able to use psdump() or pdfdump().
WARNING: No route found for IPv6 destination :: (no default route?)
INFO: Can't import python-cryptography v1.7+. Disabled WEP decryption/encryption. (Dot11)
INFO: Can't import python-cryptography v1.7+. Disabled IPsec encryption/authentication.
WARNING: IPython not available. Using standard Python shell instead.
AutoCompletion, History are disabled.
WARNING: On Windows, colors are also disabled

          aSPY//YASa
      apyyyyCY/////////YCa | Welcome to Scapy
      sY/////YSpcs  scpCY//Pp | Version git-archive.devd31378c886
  ayp ayyyyyyySCP//Pp      syY//C |
AYAsAYYYYYYYYY//Ps          cY//S |
      pCCCCY//p       cSSps y//Y | https://github.com/secdev/scapy
      SPPPP//a        pP///AC//Y |
          A//A        cyP///C | Have fun!
          p///Ac        sC///a |
          P///YCpc      A//A | Craft packets like I craft my beer.
  scccccp///pSP///p        p//Y | -- Jean De Clerck
  sY/////////y  caa        S//P |
  cayCyayP//Ya          pY/Ya |
  sY/PsY///YCc          aC//Yp |
      sc  sccaCY//PCypaappyCP//YSs
      spCPY//////YPSp |
          ccaacs
```

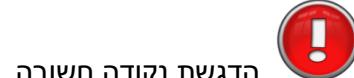
כדי לוודא שה-scapy שלכם עובד כמורה, הסנויפו באמצעותו 2 פקודות והציגו אחת מהן:

```
>>> p = sniff(count = 2)
>>> p[0].show()
```

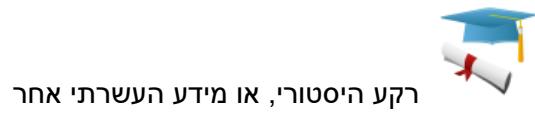
זהו, סימנו. מוכנים להתחל לעבוד!

אייקונים

בספר, אנו משתמשים באיקונים הבאים בצד להדגיש נושאים ובצד להקל על הקריאה:



תרגיל לביצוע. תרגילים אלו עלייכם לפתור בעצמכם, והפתרון בדרך כלל לא יוצג בספר



תודות

ספר זה לא יהיה נכתב אל מולא תרומתו של עומר רוזנבוים, ששיער הן בגיבוש תוכנית הלימודים בפייתון והן ביצע את העריכה של הספר. כמו כן מהתרגילים בספר נכתבו על ידי שי סדובסקי ועומר רוזנבוים, וניתן להם
เครดיט בצד התרגילים.

ברק גונן

תוכן עניינים מצגות פיתון

להלן קישורים למצגות הלימוד של פיתון, אשר עשויות לסייע לכם במהלך לימוד הפרקים השונים.

Before we start:	http://data.cyber.org.il/python/1450-3-00.pdf
Intro and CMD:	http://data.cyber.org.il/python/1450-3-01.pdf
PyCharm:	http://data.cyber.org.il/python/1450-3-02.pdf
Variables, conditions, and loops:	http://data.cyber.org.il/python/1450-3-03.pdf
Strings:	http://data.cyber.org.il/python/1450-3-04.pdf
Functions:	http://data.cyber.org.il/python/1450-3-05.pdf
Lists and tuples:	http://data.cyber.org.il/python/1450-3-06.pdf
Assert:	http://data.cyber.org.il/python/1450-3-07.pdf
Files and script parameters:	http://data.cyber.org.il/python/1450-3-08.pdf
Exceptions:	http://data.cyber.org.il/python/1450-3-09.pdf
Object Oriented Programming:	http://data.cyber.org.il/python/1450-3-10.pdf
PyGame:	ומליץ ללמידה מספר הלימוד
Dictionaries:	http://data.cyber.org.il/python/1450-3-12.pdf
Magic functions:	לא מצגת
Regular expressions:	http://data.cyber.org.il/python/1450-3-14.pdf

פרק 1 – מבוא ללימוד פיתון

מהי שפת סקריפטים?

ברוכים הבאים לשפת פיתון היא שפה שימושית וקלת לשימוש. לאחר שתשלטו בסיס השפה תוכלו לכתוב תוכניות מסוימות בקלות יחסית. לדוגמה, במספר שורות קוד תוכלו לגרום לשני מחשבים לשלוח הודעות אחד לשני. בפחות ממאות שורות קוד תוכלו לפתוח משחק מחשב, עם גרפיקה וצללים.

שפת פיתון פותחה ב-1990 כשפת סקריפטים. מהי שפת סקריפטים? כדי להבין מהי שפת סקריפטים נצטרך להבין קודם כל כיצד עבדת שפה שאינה שפת סקריפט, לדוגמה שפת C. כל שפת תוכנה צריכה להפקיד בדרך כלשהי לשפת מכונה, כדי שהמחשב יוכל להריץ אותה. ההבדל בין שפת סקריפטים לשפה שאינה שפת סקריפט הוא במסלול שעובר הקוד עד שהוא הופך לשפת מכונה. קוד שנכתב בשפת C צריך לעبور שני שלבים לפני שהמעבד יוכל להריץ אותו: השלב הראשון נקרא קומpileZA והוא מבוצע על ידי תוכנה שנקראת קומפיילר. הקומפיילר ממיר את הקוד משפת C לשפת אסמבלי. אסמבלי היא שפה שנמצאת רמה אחת מעל שפת מכונה וכך לתוכנתה יש צורך לעבד ישירות עם החומרה של המחשב. אם אתם רוצים לדעת יותר על שפת אסמבלי, תוכלו פשוט לפתח את ספר לימוד האסמבלי של גבאים. השלב השני מבוצע על ידי תוכנה שנקראת אסמלר. האסמלר ממיר את הקוד משפת אסמבלי לשפת מכונה, כלומר לשפה שהמעבד מבין. בעקבות ההמרה الأخيرة נוצר קובץ הריצה בעל הסיומת exe (קיצור של executable). הנקודה החשובה לזכירה היא שבסוף התהילה נוצר קובץ שמכיל את כל הפקודות שכתבנו, כאשר הן מתורגםות לשפת מכונה.

את השלבים הבאים תוכל לבדוק באמצעות קומפיילר אונליין כדוגמת:

https://www.tutorialspoint.com/compile_c_online.php

הבה נראה מה קורה כאשר יש שגיאה בתוכנית. ניקח תוכנית תקינה בשפת C, אשר מדפסה "Hello world", ונוסיף לה שורה חסרת משמעות – :blablabla

```

1 #include <stdio.h>
2
3 int main()
4 {
5     printf("Hello, World!\n");
6     blablabla
7     return 0;
8 }
9

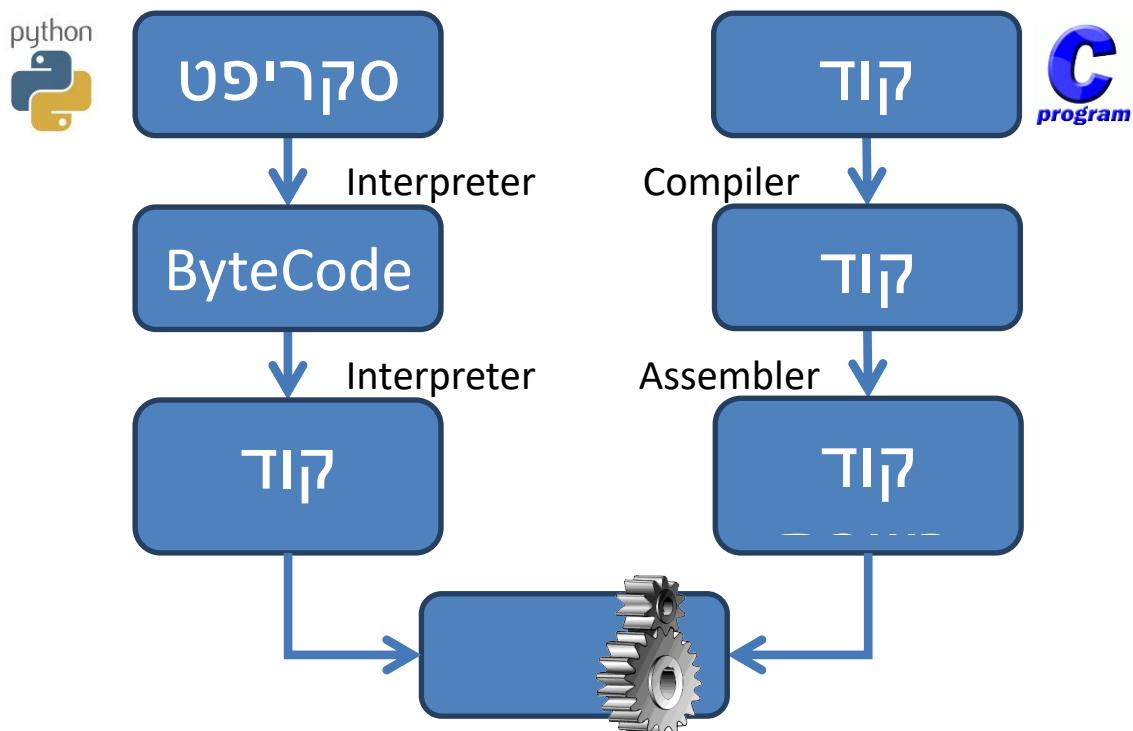
```

כאשר ננסה לבצע קומpileציה, נקבל הודעה שגיאה:

error: 'blablabla' undeclared (first use in this function)

במילים אחרות, לא נוכל להדפיס "Hello world", למרות שהשגיאה בתוכנית נמצאת בשורה שאחרי פקודה הדפסה. הסיבה היא שהתוכנית שלנו נכשלת כבר בשלב ההמרה לשפת אסמבלי, ואני עוברת כל המرة לשפת מוגנה.

עתנו נعود לדון במתוך השפה פיתון, שכזכור הינה שפת סקריפטים. על סקריפט פיתון פועלת תוכנה שנקראת interpreter ("פרשן"). ה-interpreter עובד בצורה אחרת לגמרי מאשר הקומpileר והאסמבילר בהם משתמשים כדי לתרגם את שפת C לשפת מוגנה. הוא אינו יוצר קובץ אסמבלי וגם אינו יוצר קובץ הריצה. במקומ זאת, כל פקודה שכתבנו בשפת פיתון מתורגמת לשפת מוגנה רק בזמן הריצה. תוך כדי תהליך הפירוש, נוצר קובץ עם סיומת .pyc, שמכיל bytecode – הוראות שונות של ה-interpreter – אך כאמור זה אינו קובץ בשפת מוגנה, ככלומר, מעבד לא מסוגל להרייך את הקובץ זהה.



שפת סקריפטים (פיתון) לעומת שפת C

מדוע חשוב לנו לדעת את שלבי ההמרה? כי כעת אנחנו יכולים להבין את הניסוי הבא. הפעם, ניקח תוכנית פשוטה:

```
print("Hello World")
blablabla
```

כאשר נריץ את התוכנית יתקבל הפלט הבא:

```
Hello world
Traceback (most recent call last):
  File "main.py", line 2, in
    blablabla
NameError: name 'blablabla' is not defined
```

מה קיבלנו? השורה הראשונה, שמדפסה Hello world למסך, בוצעה בהצלחה. לאחר מכן הטענו לנו שארת השורה blablabla, ניסיון שהסתיים בשגיאת הריצה. הנקודה המעניינת היא שהתוכנית רצתה באופן תקין עד שארעה השגיאה, וזאת בגיןות תוכנית זהה בשפת C, שכלל לא רצתה. הסיבה שהצלהנו להגעה לפיתון לנוקודה שחלק מהתוכנית רצתה, היא לבדוק בגליל תהליך ההמרה השונה. בפייתון לא נוצר קוד בשפה אסמליל וגם לא קובץ הריצה, ולכן השגיאה לא התגלתה עד לנוקודה שבה היה צריך לתרגם את blablabla חסר המשמעות לשפת מוכנה. רק אז הבין ה-interpreter שיש כאן בעיה ועצר את ריצת התוכנית תוך דיווח על שגיאה.

מה אפשר להסיק ממה שלמדנו עד כה? קודם כל, שפת פיתון היא הרבה יותר סלונית לשגיאות מאשר שפות אחרות. שימוש לב גם עד כמה הקוד בשפת פיתון קצר יותר מאשר בשפת C. לכן, הדעה הרווחת היא שקל יותר ללמוד לכתוב קוד בשפת פיתון. עם זאת, גם לשפת C יש יתרונות על פיתון: ראשית, אם נכתב קוד לא זהיר בשפת פיתון הוא יתרסק תוך כדי ריצה. אין מגנון כמו הקומpileר של C, שמנוע מאיינו לכתוב קוד שלא עומד בכלל התחביר של השפה ולכן הסיכוי לביעות בזמן ריצה הוא קטן יותר. התרסקות של קוד תוך כדי ריצה היא חמורה לאין שיעור מאשר שגיאת קומPILEציה, אותה ניתן לגלוות ולדבג לפני ההרצה. שנית, העובדה שקוד בשפת C מתורגם לשפת מוכנה לא שורה אחר שורה אלא קובץ אחד, מאפשרת לבצע תהליכי "יעול" (אופטימיזציה) של הקוד, כך שהוא עשוי לרוץ יותר מהר ולצרוך פחות זיכרון מאשר קוד מקביל בשפת פיתון. זה דבר חשוב למי שורצים להריץ אפליקציות "כבדות", כגון גרפיקה מורכבת או הצפנה.

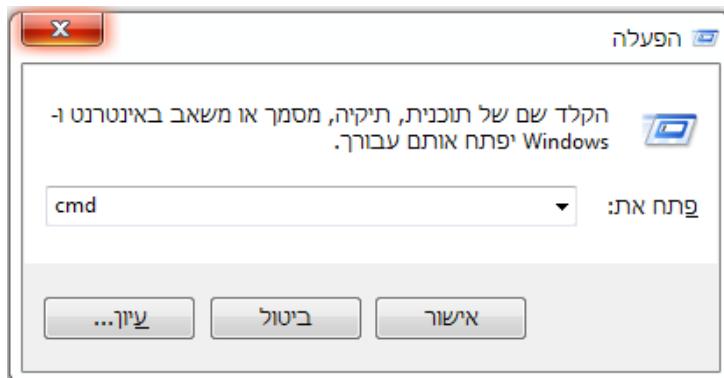
לסיכום, אפשר לומר שפיתון היא שפה נוחה וקלת כתיבתה, שמאפשרת להגעה לתוכניות עובדות, גם אם הדבר בא על חישוב מהירות או יעילות. בשל כך זכתה פיתון למפתחים רבים, שדAGO לכתוב מודולים – קטעי קוד SMBצעים משימות שונות – ולהפיץ אותם. כתוצאה לכך, אחד ה יתרונות העיקריים של פיתון הוא שניתן לקבל מן המוכן קוד בשפת פיתון SMBצע משימות רבות: חישובים מתמטיים, גרפיקה, ניהול קבצים במחשב וכל דבר שניתן להעלות על הדעת.

עבודה באמצעות command line

נעיל את פיתון בדרך הקצרה וה מהירה. ישנו חלון טקסטואלי פשוט, שנקרא command line, או בעברית "שורת הפקודה". כדי להגיע לחלון של command line, לוחצים על מקש ה-winkey במקלדת (בתמונה) ובו זמניית על מקש ה-"R":



"ופיע מסק "הפעלה" או באנגלית חנץ, ובתוכו כתבו את שם התוכנה שאתם רוצים להפעיל. במקרה זה – cmd.



```
C:\WINDOWS\system32\cmd.exe - python
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Users\barak>python
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:21:23) [MSC v.1916
32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.

>>>
```

עם לחיצה על מקש `enter` תגעו למסך `the-line command`. במסך יהיה כתוב שם התיקיה בה אתם נמצאים, לדוגמה `cyber:c`. כתע כתבו `python` והקישו `enter`. ברוכים הבאים לפיתון!

אנחנו יכולים לכתוב כל תרגיל חשבוני שאנו רוצים, לדוגמה $5+4$, ונקבל מידית את התשובה. אנחנו יכולים אפילו להגדיר משתנים, לדוגמה $a=4$ ו- $b=5$,oca ואחר נכתוב $a+b$ נקבל את הסכום שלהם. נסן זאת!

מספרים בבסיסים שונים

ניתן כאן הסבר מקוצר מאד לשיטות הבינאריות וההקסדצימליות. הנושא הזה מכוסה בהרחבה בספר האסמבלי של גבאים, ואם איןכם שולטים בבסיסי ספירה מומלץ להניח מעט לספר הפיתון וללמוד בסיסי ספירה בטרם תמשיכו.

אנחנו בני האדם סופרים בבסיס 10, זה דבר טבעי בעיניון כיון שיש לנו עשר אצבעות. מחשבים סופרים בבסיס 2, בינאריו. בסיס זה קיימות רק ספרות 0 ו-1. אין במחשבים שום ספרות חוץ מאשר 0 ו-1.



כיון שלבני אנוש מסובך מעט לקרוא רצפים ארוכים של אחדות ופסים, מקובל להציג מידע ששמור בזיכרון המחשב בספרות הקסדצימליות – בסיס 16. זאת ממש שכל ספרה הקסדצימלית מייצגת 4 ספרות ביןאריות (שימוש לב-ש-16 הינו 2 בחזקת 4, ולכן כל זה מתקיים). לדוגמה את הרצף:

1011 0001 1000 0010

ניתן לכתוב בספרות הקסדצימליות:

B182

...הרבה יותר קצר לקרואיה!

אם נרצה לכתוב בפייתון מספרים בינאריים, עליהם להתחיל ב-0b. לדוגמה 0b1110 הינו 3, ו-0b1110 הינו 7. כדי לכתוב מספרים הקסדצימליים בפייתון, נוסיף להם תחילית 0x. כך לדוגמה, 0x1A הינו 26, ו-0x2B הינו 43.

אפשר לראות זאת בקילות אם נכתוב אותם בסביבת הפיתון:

```
>>> a = 0x2b  
>>> a  
43  
>>> b = 0b111  
>>> b  
7
```

Help



כיתבו בפייתון 2 בחזקת 7.

לא יודעים איך לכתוב חזקה בפייתון? כיתבו ()`help()`. יופיע הכתוב הבא:

```
C:\WINDOWS\system32\cmd.exe - python
>>> help()

Welcome to Python 3.8's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at https://docs.python.org/3.8/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules. To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics". Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".

help>
```

ונכל לבחור בעזרה בנושאים הבאים: **Topics, Keywords, Modules**. נבקש את רשימת כל הנושאים ב-

ונקבל את הרשימה הבאה:

```
C:\WINDOWS\system32\cmd.exe - python
help> topics

Here is a list of available topics. Enter any topic name to get more help.

ASSERTION          DELETION          LOOPING           SHIFTING
ASSIGNMENT         DICTIONARIES      MAPPINGMETHODS  SLICINGS
ATTRIBUTEMETHODS  DICTIONARYLITERALS  MAPPINGS        SPECIALATTRIBUTES
ATTRIBUTES          DYNAMICFEATURES   METHODS          SPECIALIDENTIFIERS
AUGMENTEDASSIGNMENT ELLIPSIS         MODULES          SPECIALMETHODS
BASICMETHODS       EXCEPTIONS        NAMESPACES      STRINGMETHODS
BINARY             EXECUTION         NONE            STRINGS
BITWISE            EXPRESSIONS      NUMBERS          SUBSCRIPTS
BOOLEAN            FLOAT             OBJECTS          TRACEBACKS
CALLABLEMETHODS    FORMATTING       OBJECTS          TRUTHVALUE
CALLS              FRAMEOBJECTS     PACKAGES        TUPLELITERALS
CLASSES            FRAMES            OPERATORS       TUPLES
CODEOBJECTS        FUNCTIONS         POWER          TYPEOBJECTS
COMPARISON          IDENTIFIERS      PRECEDENCE     TYPES
COMPLEX             IMPORTING        PRIVATE NAMES  UNARY
CONDITIONAL        INTEGER           RETURNING      UNICODE
CONTEXTMANAGERS    LISTLITERALS     SCOPING         SEQUENCEMETHODS
CONVERSIONS        LISTS            SEQUENCES      SEQUENCES
DEBUGGING          LITERALS         POWER          TYPEOBJECTS

```

אם נكتب POWER (המוני האנגלי לחזקה) נקבל את התיאור המלא של POWER, שם נגלה שכך להעלות
בחזקה עליינו להשתמש בסימן **. לדוגמה, $7^{**}2$ הינו 2 בחזקת 7.

כדי לצאת מ-help נכתובquit ונגיע חזרה לחילון הפקודה שלנו. נסו כעת להשתמש במה שלמדנו!

הסימן _ (קו תחתון)

כפי שראינו, אפשר להגדיר בפייתון משתנים בקלות, סימן '=' ולאחר מכן
מכוון הערך שהמשתנה מקבל. קיימים בפייתון סוגים רבים של משתנים, עליהם נלמד בהמשך. כדי להגדיר משתנה
בשם a, אשר ערכו הוא 17, פשוט כתבים $a=17$.



תנו ערכים כלשהם למשתנים a ו-b. חשבו את הביטוי הבא:

$$4*(a+b)+3$$

כל? כתת חשבו את הביטוי הבא:

$$(4^*(a+b)+3)^{**2}$$

דרך אחת היא פשוט להעתיק את כל התרגיל מהתחלתו. אולם כפי שהבוחנתם, מדובר במעשה באותו תרגיל פרט לכך שהוא בחזקת 2. כדי למצאו את הפתרון באופן אלגנטי ובלי הקЛОDOT מיותרות פשוט רישמו את הביטוי הראשוני, לחזו `enter` ו בשורה הבאה כתבו:

`_^{**2}`

פירשו של הקוו התחתון – "קח את התוצאה האחרונה שיחסבת". באופן זה נוכל לחסוך זמן כתיבת ולבצע חישובים בקלות. שימושו לב שהטריך זהה עובד ב-`interpreter`, אך לא בסביבת העבודה אותה נלמד בהמשך.

הרצה תוכניות פיתון דרך ה-command line

כעת נכתוב את תוכנית הפיתון הראשונה שלנו! ראשית נצטרך תוכנת עריכה כלשהי. בטור התחלתה מומלץ להשתמש ב-`++notepad`, אשר ניתן להורדה בחינם מהאינטרנט. לא השתמש בו הרבה, אך שאמם אתם מעדיפים לא להתקין אותו תוכלו להשתמש `notepad` שmagiuha עם חלונות. איך מפעילים? בדיקן כמו שלמדנו להפעיל כל תוכנה. ליחזו על `R+winkey`, כתבו `notepad` בחלון הפעלה זהה.

את הקובץ שיצרתם שימרו עם סימנת `uk`, כלומר קובץ פיתון. לדוגמה `hello.py`. כתבו את הפקודה הבאה:

```
print("Hello cool cyber student!")
```

ושימרו את הקובץ.

כעת כל מה שנותר לנו לעשות הוא מתוך ה-command line להריץ את הקובץ הפיתון שלנו. ראשית יש להגיע אל התקינה שבתוכה שמרנו אותו. לדוגמה, אם שמרנו את הקובץ בתוך `cyber\c:` אז علينا לכתוב:

```
cd c:\
```

```
cd cyber
```

לאחר שהגענו לתיקיה הנכונה, נכתוב:

```
python hello.py
```

והקובץ שלנו יורץ מיד ☺

ΟΙΚΟМ

בפרק זה רכשנו את הבסיס לתוכנות פיתון. אנחנו מビינים שפיתון היא שפת סקריפטים, אנחנו יודעים להריץ פקודות פיתון פשוטות גם דרך `the line command` וגם דרך קבצי פיתון שיצרנו. אנחנו יודעים איך לחפש בפייתון עזרה על כל נושא שנרצה. למעשה, עם הידע שקיים אצלנו בשלב זה כבר אפשר להסתדר לא רע – הרי בסופו של דבר אנחנו יודעים לכתוב תוכניות ולהריץ אותן, וכמובן גם יודעים לחפש עזרה בנושאים לא מוכרים. מעכשיו, יוכל למצוא הדרכה על כמעט כל נושא שנלמד ולהסתדר בלבד, אם נעדיף שלא לקרוא את המשך הספר...

פרק 2 – סביבת עבודה PyCharm



עד כה למדנו איך כתבים פקודות בסיסיות בחלון command line או באמצעות קובץ שכתבנו ב-`notepad++`. מה שעשינו נחמד בתור התחלה, אבל קשה מאוד לכתוב תוכניות שימושיות בצורה זו. חסירה לנו סביבת עבודה שמאפשרת להריץ את הקוד מתוך הסביבה וכוללת דיבאגר. הכוונה בדיagger היא תוכנה שמסוגלת להריץ את הקוד שלנו פקודה אחרי פקודה, תוך כדי הציג ערכיו המשתנים השונים. זו יכולת קריטית לשוכרים להבין מדוע תוכנית שכתבנו לא עובדת באופן תקין.

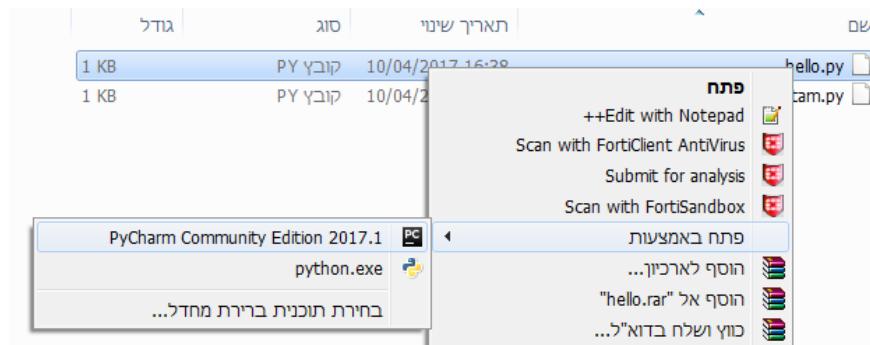
בחרנו להציג בפניכם את סביבת העבודה PyCharm. היכולות שיש给她ה זו מעניקה לנו הם:

- כתבן (תחליף `notepad++`).
- איתור שגיאות אוטומטי – כן, PyCharm מוצא בשביבתו את השגיאות בקוד שלנו. אין הכוונה לכך ש- PyCharm יודע לתקן עבורנוágים באלגוריתם, אך PyCharm בחלטת יגלה לנו אם שכחנו לשימוש נקודתיים, או שהשתמשנו במשתנה ששכחנו לתת לו ערך תחלה.
- דיבאגר – כאמור, يمكنك להריץ את הקוד שלנו פקודה אחר פקודה (step by step), תוך הציג ערכיו המשתנים.
- يمكنك לאתחל בקהלת תוכנית לאחר קירסה או באג – תוכל לחסוך זמן משמעותי כאשר התוכנית שלנו מתרסקת, על ידי כך שבמקרה לאתחל את התוכנה פשוט נורא `PyCharm` להריץ אותה שוב.

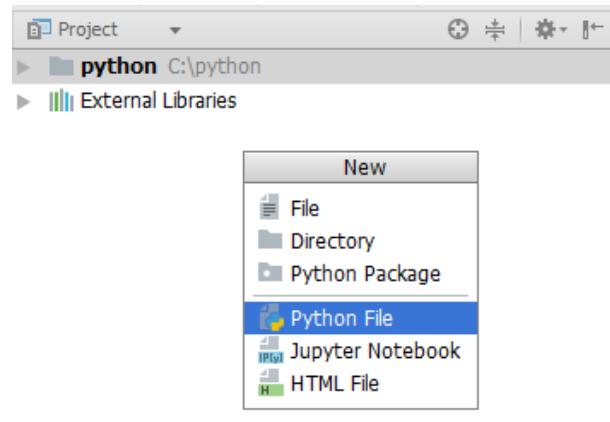
כעת נלמד להשתמש בסביבת העבודה PyCharm.

פתיחה קובץ פייתו

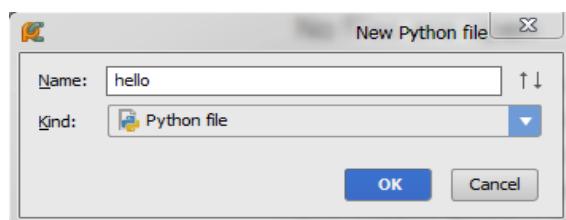
אם הקובץ כבר קיים, כל שנצרך לעשות הוא להקליק עליו קלייק ימני ולבחר אפשרות `open` ולאחר מכן `PyCharm Community Edition`.



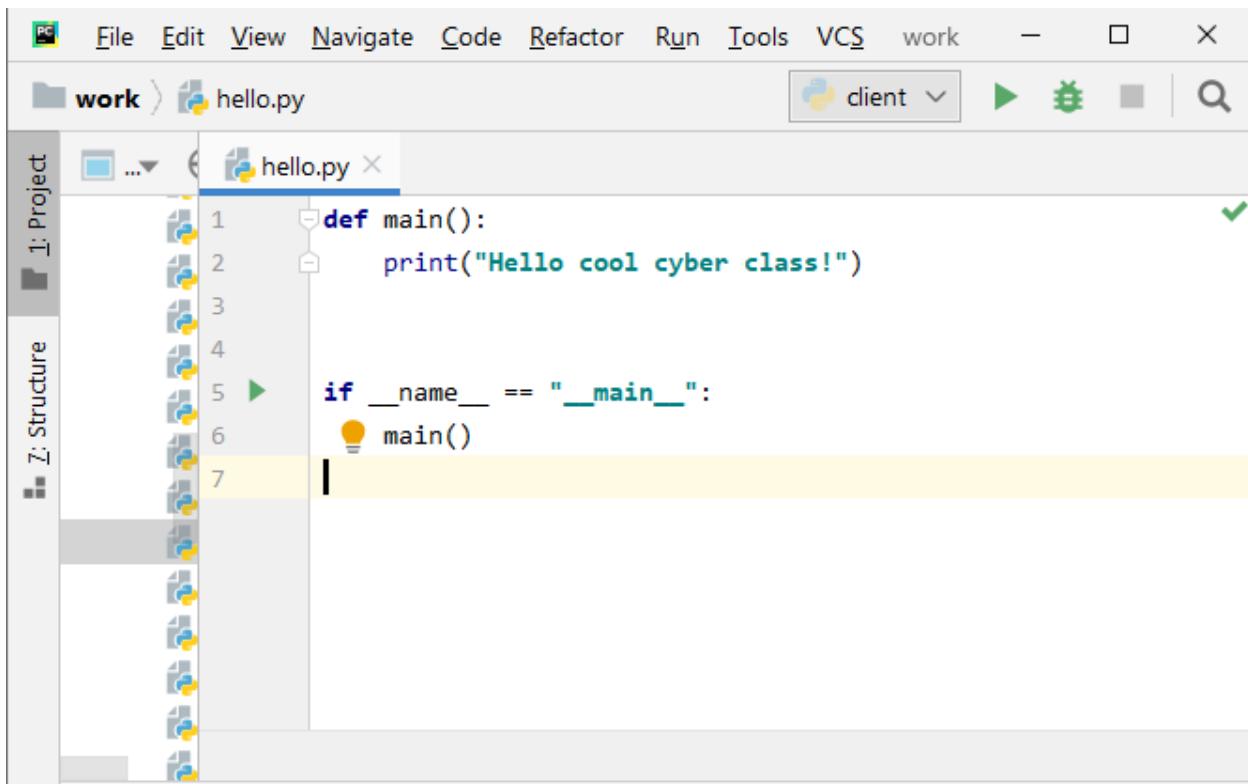
אם נרצה לפתוח קובץ חדש, אז נקליק על האיקון של PyCharm ומהתפריט נחבר ב-file, לאחר מכן new וatz מבין האפשרויות נבחר ב-new :python file



לאחר מכן ניתן לקובץ החדש שם:



וכעת ניתן לעורק את הקובץ החדש שיצרנו:



```

1 def main():
2     print("Hello cool cyber class!")
3
4
5 > if __name__ == "__main__":
6     main()
7

```

קובץ הפיתון הראשון שלנו

נסקרו את השורות בקובץ הפיתון הראשון שלנו:



```

1 def main():
2     print("Hello cool cyber class!")
3
4
5 > if __name__ == "__main__":
6     main()
7

```

בשורה 1 מוגדרת פונקציה בשם `main`. כדי שברור מהשם שלו, זו הפונקציה הראשית של התוכנית שלנו. המשמעות היא שגם הפונקציה הראשונה שנקראת (מיד נבין מי קורא לה) ולכן כל מה שאנו רוצים שיבוצע צריך להיות כתוב בתוך `main`, או להחליף ש-`main` תקרה לו.

בשורה 5 יש תנאי מושך, שבהמישך הלימוד נבין מה פירושו. בקצרה – אנחנו יכולים להריץ סקריפט פייתון בשתי צורות. האחת, היא פשוט להריץ אותו. השנייה, היא להריץ סקריפט אחר שקורא לסקריפט שלנו. הדרך השנייה מתבצעת בעזרת פקודה שנקראת `import` וnlmd עליה בהמשך. נניח שכתבנו סקריפט פייתון ש מכיל כמה פונקציות מעניות, וחבר לנו רוצה להשתמש בהן. אם החבר יקרא לסקריפט שלנו על ידי פקודה `import`, אז עלול להיווצר מצב שבו כל הקוד שכתבנו רץ. לעומת, במקרה שהתוכנית שלו פשוט תכיר את הפונקציות שלנו ותוכל לקרוא להן, התוכנית של החבר פשוט מפעילה את כל התוכנית שלנו. זה כמובן לא מצב רצוי. לכן, אנחנו מוסיפים את שורה 5 לקוד שלנו. שורות אלו אומרות לפייתון "שמע פייתון, יכול להיות שהסקריפט הזה יצורף לסקריפט אחר. לכן קודם כל תבדוק אם מי שMRIIZ את הסקריפט זה לא מישוה אחר שעשה לו `import`, ורק אם הסקריפט אינו מושך `import` אז תקרא לפונקציה `main` שתחל את ריצת הסקריפט".

סדר ההרצה של פקודות בסקריפט פייתון

התבוננו שוב בסקריפט `ukHello.py`. מה יהיה סדר ההרצה של הפקודות? כמובן, איזו פקודה תרוץ ראשונה?

ה-`interpreter` של פייתון, שתפקידו לתרגם את פקודות הפייתון לשפת מכונה, מחפש את הפקודה הראשונה `print` לצד שמאל ואז הוא בודק אם לא מדובר בהגדלה של פונקציה. לכן, בשורה מספר 1 פייתון לימד שישנה פונקציה בשם `main`. פייתון יוסיף את `main` לרשימה הפונקציות המוכרות לו. כאמור בשלב זה אפשר לקרוא ל- `main`, אך אין זה אומר שפייתון MRIIZ את `main` או אפילו בודק שהקוד של `main` הוא תקין. כל מה שידוע לפייתון – ישנה פונקציה `main`.

רק בשורה 5 נתקל פייתון בפקודה שעליינו להריץ ושמבצעת משזה. אם התנאי שבשורה מתקיים, פייתון ממשיר אל שורה 6 ושם נאמר לו להריץ את הפונקציה `main`. בשלב זה פייתון יקפוץ אל `main` ויחל לבצע את מה שנכתב בה. מאידך, אם התנאי שבשורה 5 אינו מתקיים, פייתון הגיע לסוף הסקריפט וריצת הסקריפט תסתיים.

כעת שימו לב לסקריפט הבא – מה יהיה סדר הפקודות שיבוצע?

שיםו לב לכך, שזו ה תוכנית פייתון הגיונית וצורת הכתיבה של הקוד לא משקפת בשום אופן צורת כתיבתה מומלצת של קוד. המטרה של הקוד הבא היא רק להמחיש את סדר שלבי ריצת סקריפט הפייתון. היה נכון הרבה יותר לכתוב את כל פקודות ההדפסה שלנו בתוך פונקציית `main`. אם כך, מה יהיה סדר הפקודות שיבוצע?

```
print("You say ", end="")
```

```
def main():
    print("Hello")
```

```
print("Goodbye")

if __name__ == "__main__":
    print("I say ", end="")
    main()
```

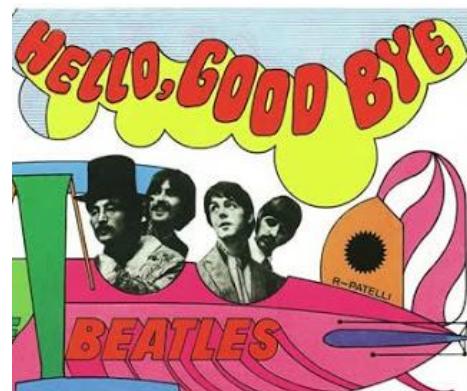
שורה 1 תדפיס למסך את הטקסט "You say ". מודיע כתוב "end="" בלי זה, לאחר print תבצע ירידת שורה.
מה שאנו רצים הוא שלאחר ההדפסה יהיהתו אחד של רווח.

לאחר מכן תבצע שורה 8, ידפסו "Goodbye", עם ירידת שורה בסופה.

בעקבות התקיימות התנאי בשורה 10, יבוצעו שורות 11 ו-12. שורה 11 תדפיס למסך say | עם רווח. שורה 12 תקרא לפונקציה `main`. בתוך הפונקציה `main` תבצע הדפסה של Hello.

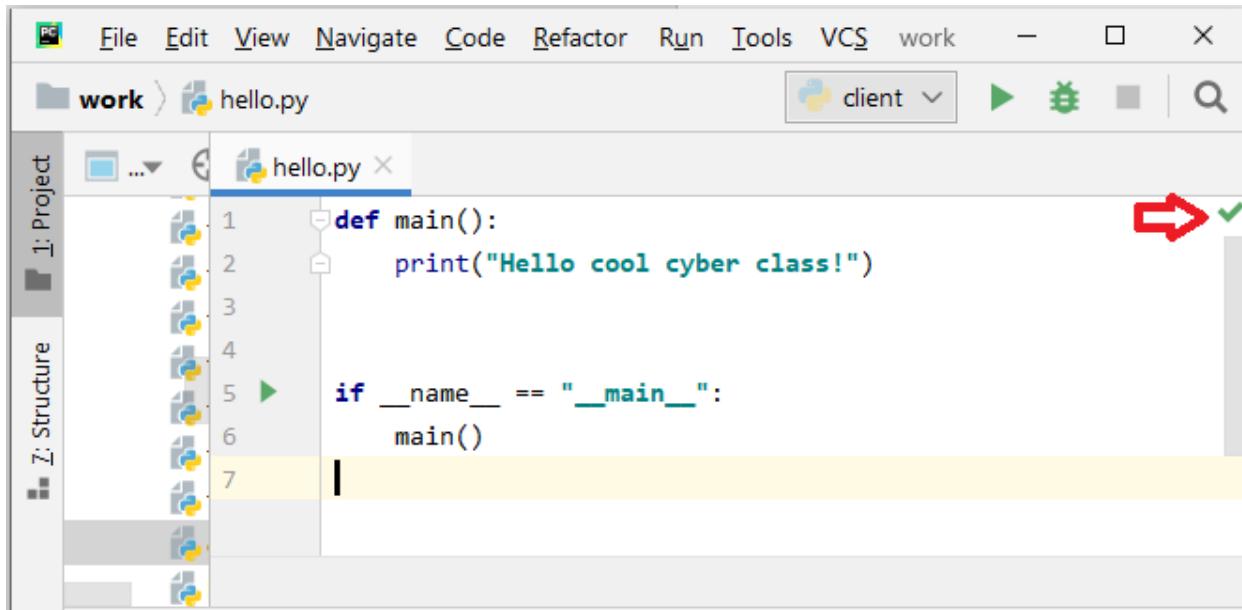
לאחר שהפונקציה `main` תסימן את ריצתה, אליה היא נקראת משורה 12, היא אמורה לחזור ולהריץ את המשך התוכנית – אילו היה צזה – אחרי שורה 12. כיוון שורה 12 היא סוף התוכנית, בכך יסימם הסקריפט את ריצתו.

You say Goodbye
I say Hello



התרעה על שגיאות

האם אתם מבחינים בסימן ה-V הירוק, שנמצא בחלק הימני העליון של PyCharm ?



The screenshot shows the PyCharm interface with a project named "work". In the code editor, there is a Python file named "hello.py" containing the following code:

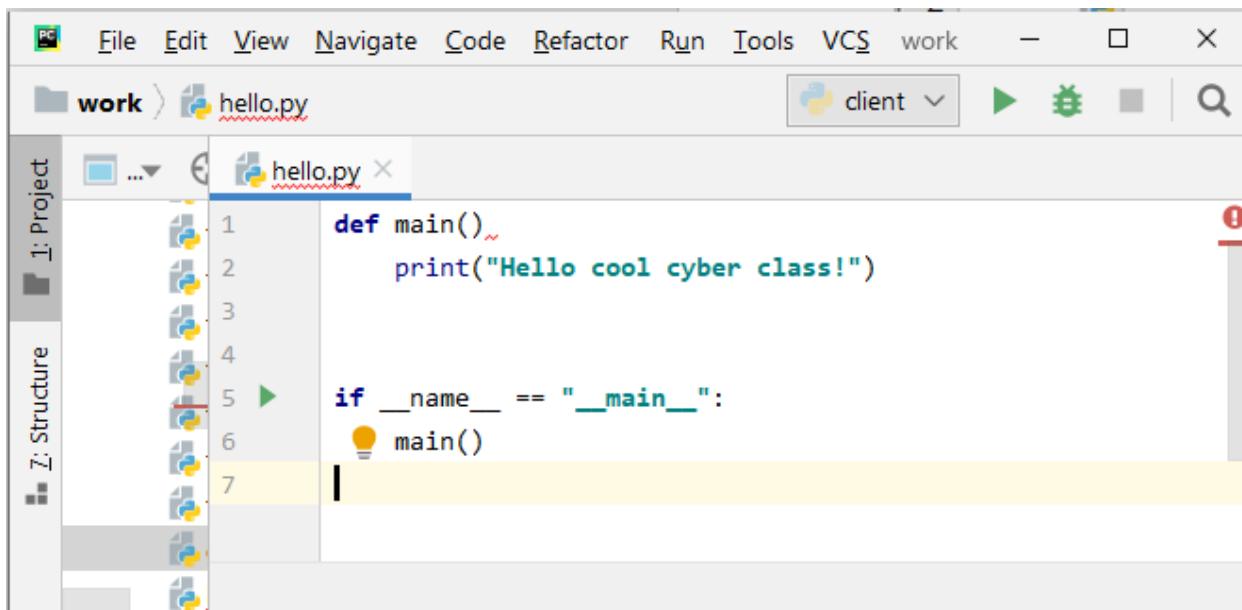
```

1 def main():
2     print("Hello cool cyber class!")
3
4
5 if __name__ == "__main__":
6     main()
7

```

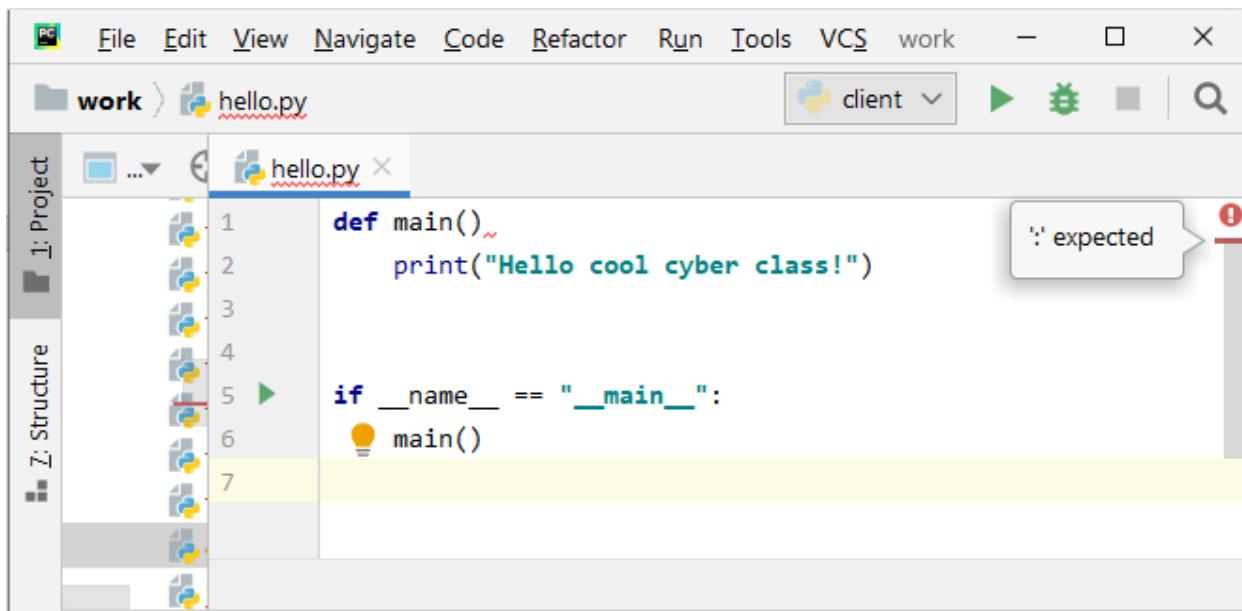
A red arrow points to a green checkmark icon in the status bar at the top right of the editor window.

זהו סימן שהתוכנית שכתבנו אינה מכילה שגיאות. שימוש לב מה קורה כאשר אנחנו מוחקים את סימן הנוקודתיים
שאחרי המילה `main`:



The screenshot shows the PyCharm interface with the same project and code as the previous one, but now the word "main" in the first line is underlined in red, indicating a syntax error. A red circle with a question mark is visible in the status bar.

קיבלנו סימנים אדומים במספר מקומות במשפט. ראשית, במקום הנקודות שמחקנו מופיע קו תחתי אדום. שניית, ה-V הירוק הפך לסיון קרייה לאדום. שלישיית, אם נעמוד עם העכבר על הקו האדום שבצד ימין, נקבל הסבר מה הבעיה בקוד שלנו.



The screenshot shows the PyCharm interface with a project named 'work' containing a file 'hello.py'. The code in 'hello.py' is:`def main():
 print("Hello cool cyber class!")

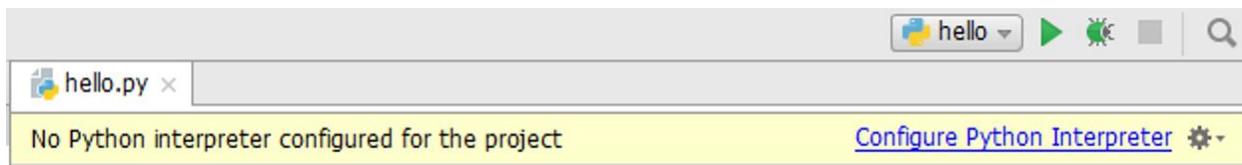
if __name__ == "__main__":
 main()`

A red underline is under the opening brace of the if statement, and a tooltip says ': expected'. The line 'if __name__ == "__main__":' is highlighted in yellow.

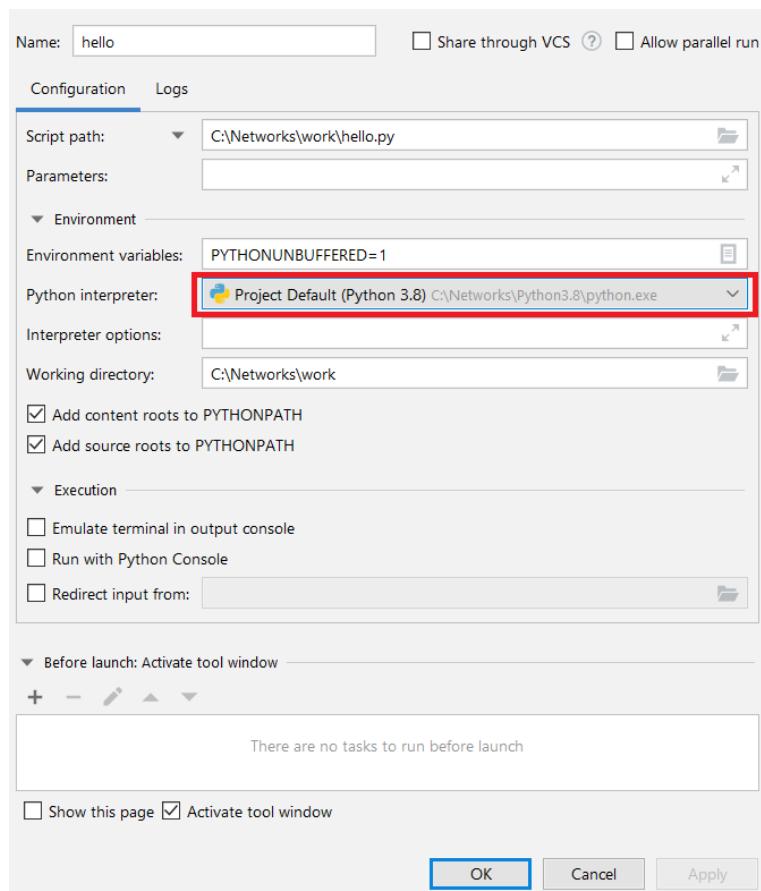
קליק שמאלית על הקו האדום מימין גם יוביל אותנו בדיק לשורה הבעייתית. בקיצור, כל מה שאנו צריכים בשבייל להימנע משלויות קוד ולפתרו אותן בקלות!

קביעת פרשן (Interpreter)

סבירות ההתקנות אמרה להתקין ולאפשר לך לעבוד מיידית בלי צורך בהגדרות נוספת, אולם אם מופיעה לך שגיאיה מסוג "No Python interpreter configured for the project", כדי שאפשר לראות מודגש בצד שמאל בתמונה, להן הדרך לסדר את ההגדרות. ראשית צריך להגדיר לו PyCharm איפה ניתן למצוא את ה-**interpreter** של שפת פיתון, זאת מכיוון שיש גרסאות שונות של שפת פיתון, ויתכן שעל אותו המחשב מותקנות גרסאות אחרות. לכן, נקליק הכיתוב **Configure Python Interpreter**:

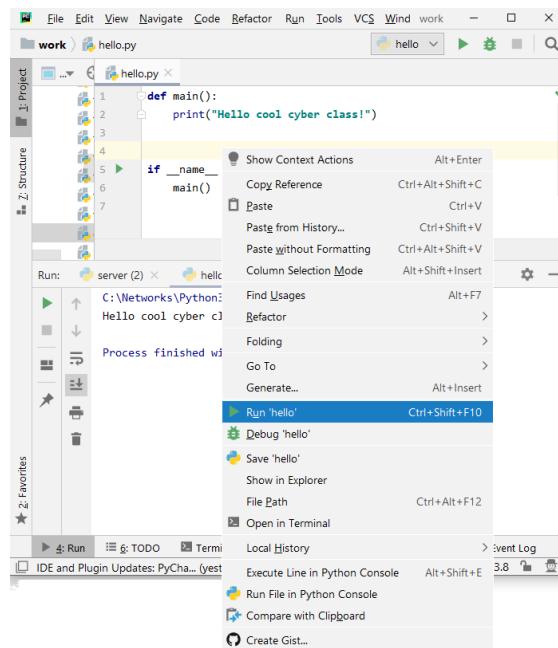


לאחר מכן נבחר את גרסת הפיתון שמותקנת במחשב:

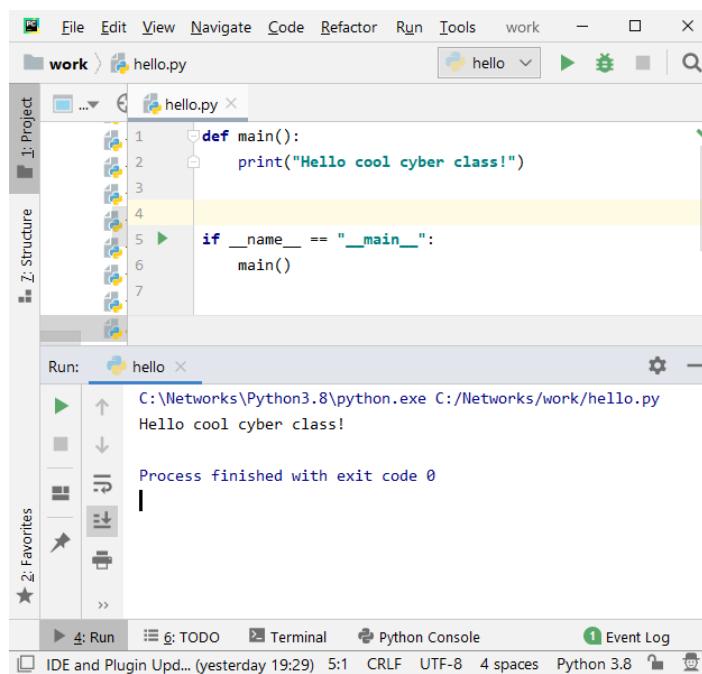


הרצה הסקריפט ומסך המשתמש

כדי להריץ את הסקריפט שכתבנו, נלחץ על עבר ימני ונבחר באפשרות חוץ:



בעקבות הלחיצה על run יופיע מסך משתמש, שם יודפס כל מה שהסקריפט שלנו מדפיס למסך. רואים את הטקסט שעשינו לו `print`



מסך המשתמש משמש גם לקבלת קלט מהמשתמש. שימוש לב לפיקודה `input`:

The screenshot shows the PyCharm interface. In the top navigation bar, the project name 'work' and file 'hello.py' are visible. Below the navigation bar is the code editor window. The code in 'hello.py' is:

```

1 def main():
2     name = input("Please enter your name: ")
3     print("Hello " + name)
4
5
6 if __name__ == "__main__":
7     main()

```

The line `name = input("Please enter your name: ")` is highlighted with a yellow background. In the bottom right corner of the code editor, there is a green checkmark icon.

Below the code editor is the 'Run' tool window. It shows the command `C:\Networks\Python3.8\python.exe C:/Networks/work/hello.py`. The output pane displays the following text:

```

Please enter your name: Shooki
Hello Shooki

Process finished with exit code 0

```

The 'Run' tab is selected at the bottom of the tool window.

פקודה זו מדפיסה למסך את מה שנמצא בסוגרים, וכל מה שהמשתמש מקלט נכנס לתוך המשתנה, במקרה זה קראנו לו `name`.

בשורה הבאה מתבצעת הדפסה של Hello ולאחר מכן הערך שנמצא בתוך המשתנה `name`. התבוננו בחלק התיכון של המסר ובידקו שאתם מבינים את תוצאת הריצה.

דיבוג עם PyCharm

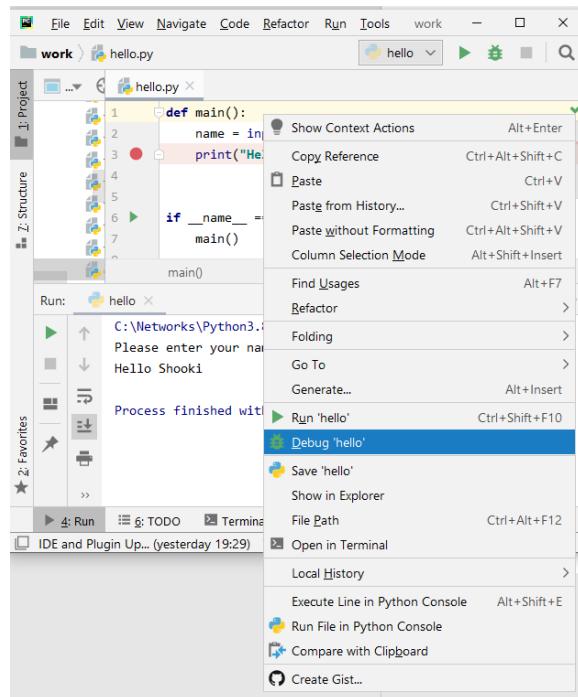
כאמור, היכולת המשמעותית של PyCharm היא האפשרות לדבג קוד. כיצד עושים זאת?

בשלב הראשון יש לשתול breakpoint בקוד. דבר זה מתרחש על ידי לחיצה שמאלית על העכבר, באזור האפור שליד מספר השורות. בעקבות כך, תופיע נקודה אדומה ליד מספר השורה, כפי שנitinן לראות ליד שורה 6 בקטע הקוד הבא:

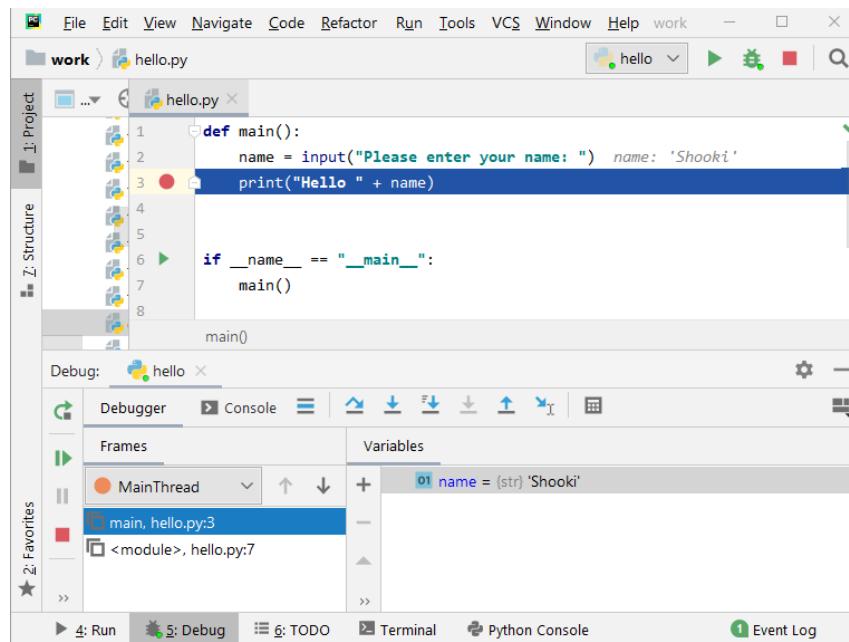


```
File Edit View Navigate Code Refactor Run Tools work
work > hello.py
Project 1: work
hello.py
1 def main():
2     name = input("Please enter your name: ")
3     print("Hello " + name)
4
5
6 if __name__ == "__main__":
7     main()
```

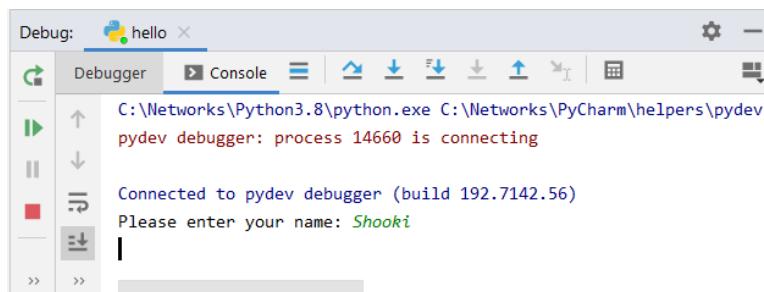
בשלב הבא צריך להריץ את התוכנית במוד debug, על ידי קליק ימני ובחירה האפשרות debug מבין האפשרויות:



לאחר מכן נראה כי התרחשו מספר דברים. ראשית, התוכנית רצתה עד לשורה עם הנקודה האדומה, ה-`breakpoint`, ושם עצמה. השורה סומנה בכחול. שנית, נפתח לנו חלון ה-`debugger`, אשר מכיל מידע על המשתנים שוגדרים בתוכנית שלנו. לדוגמה, המשתנה `name` הוא מסוג `str` (במה שיר, נבין את המשמעות של `str` – משתנים השונים) וערך הוא `Shooki`, הערך שהמשתמש הזין לתוכו. ניתן לראות את הערך של `name` בחלון `variables`:



לחיצה על לשונית `Console` תעביר אותנו אל מסך המשתמש (קלט / פלט):



נחזיר אל חלון הדיבאגר. כפי שניתן לראות ישנו שם חיצים שונים. לחיצה על החיצים תקדם את התוכנית שלנו. אם נעמוד עם העכבר על חץ כלשהו יופיע כיתוב שמספר מה החץ עשו. בשלב זה מומלץ להשתמש בחץ `Step Over`, השמאלי ביותר. נוח מאד להשתמש בקיצור המקלדת שלו – F8. חץ זה מרים שורת קוד אחת בכל פעם, שורה אחר שורה. כאשר נכתב פונקציות, נרצה להשתמש לעיתים קרובות גם בחץ השני משמאלי, `Step Into`. קיצור המקלדת שלו הוא F7.



אם נרצה להפסיק את הדיבוג, או להתחילה מחדש, יוכל להשתמש בלחיצנים מצד שמאל של חלון הדיבאגר:



מומלץ לשЛОוט בפעולות אלו, מכיוון שבבואה העת שימוש נכון בהן יאפשר לכם לגלוות בעיות בתוכנה שלכם!

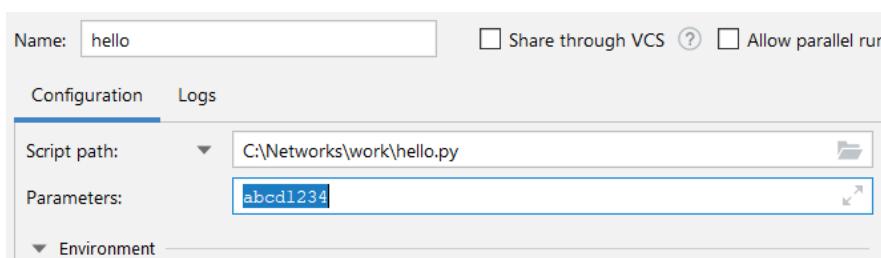
העברה פרמטרים לסקריפט

دلגו על החלק זהה, וחזרו אליו רק לאחר שתלמדו על `sys.argv`.

כדי להעביר פרמטרים לסקריפט, כולם מידע שהמשתמש מעביר לסקריפט לפני תחילת הריצה (לדוגמה, שם של קובץ שהסקריפט צריך לעשות בו שימוש), נקליק על שם הקובץ שנמצא מצד ימין למטה, ואז נבחר `:configurations`



כעת יפתח מסך שבו ניתן להקבע `Script Parameters`. ניתן להזין לתוכו כמה פרמטרים שנרצה, עם סימן רווח בין פרמטר אחד לשני:



סיכום

בפרק זה סקרנו את סביבת העבודה PyCharm, שתשמש אותנו בהמשך. למדנו כיצד לטעון קובץ פיתון, לעורר אותו, להריץ אותו ולדבג אותו. PyCharm יכולות רבות – נצלו אותן.

פרק 3 – משתנים, תנאים ולולאות

סוגי משתנים בפייתון

בפרק הראשון רأינו איך יוצרים משתנה בשם `a` שערך שווה למספרשלם. סוג המשתנה שמכיל מספרשלם נקרא `integer`, או בקיצור `int`.

איך אפשר לראות את סוג המשתנה? באמצעות הפקודה `type`, סוג.

ניתבו ב-interpreter:

```
>>> a=2
>>> type(a)
```

אפשר גם לכתוב בקיצור:

```
>>> type(2)
```

בפייתון יש סוגי משתנים רבים, לא רק `int` כMOVED, אנחנו נסקרו אותם אחד אחד, כאשר בפרק זה נתחיל מטיפוסי המשתנים פשוטים ביותר ובפרקם הבאים נcosa טיפוסי משתנים נוספים.

שים לב לשינויים השונים:



```
>>> type(True)
<type 'bool'>
>>> type(2)
<type 'int'>
>>> type(2.0)
<type 'float'>
>>> type('Hi')
<type 'str'>
>>> type([1,2,3])
<type 'list'>
>>> type((1,2,3))
<type 'tuple'>
>>> type({1:'a', 2:'b', 3:'c'})
<type 'dict'>
```

משתנה בוליאני (bool): משתנה בוליאני יכול לקבל שני ערכים בלבד – אמת או שקר, True או False. מגדירים משתנה בוליאני באופן הבא:

```
>>> a = True
>>> b = False
```

שימוש לב שחייב לשמר על האות הגדולה בתחילת המילה, אם נכתב true או false נקבל שגיאה.
מכאן אפשר להסיק שפייתו היא שפה שմבדילה בין אותיות גדולות לקטנות (מה שנקרא "case sensitive"). כמובן, המשתנה a שונה מהמשתנה A.



מהו ערך של משתנה בוליאני? מסתבר שיש לו ערך מסוים. True שווה ל-1, ואילו False שווה ל-0. לדוגמה אם נכתוב:

True + 1

פייתו ידפיס לנו את התשובה – 2.

```
>>> True+1
2
```

משתנה מסוג float/int: מספר הוא או מטיפוס int (אם הואשלם) או מטיפוס float (אם הוא עשרוני). כל תוצאה של פעולה חשבונית בין int ו-float תמיד תישמר בתור float.

```
>>> a = 2
>>> type(a)
<type 'int'>
>>> b = 2.0
>>> type(b)
<type 'float'>
>>> c = a + b
>>> c
4.0
>>> type(c)
<type 'float'>
```

שימוש לב לרך שיש במספרים מסוג float או דיווק עיר, שנובע מכך שבמסופו של דבר הם נשמרים בזיכרון המחשב באמצעות חזקות של 2, ויש כמות מוגבלת של ביטים שבה נשמר כל משתנה. לכן, רוב המספרים מסוג float נשמרים עם "עיגול" מסוימים. לדוגמה, אין דרך לייצג את התוצאה של 1 חלקו 7 באמצעות סופית של ביטים. אי הדיווק הזה גורם לכך שם נעשה פעולות חשבוניות שיגרמו להצטברות השגיאה, לבסוף נקבל תוצאה שהיא ברור שהיא אינה מדויקת. הנה, כאשר נחבר 0.1 עם עצמו מספר פעמים, ניווכח שההתוצאות אינן "עגולות" כפי שהיא צפוי:

```
>>> a = 0.1
>>> a + a
0.2
>>> _ + a
0.30000000000000004
>>> _ + a
0.4
>>> _ + a
0.5
>>> _ + a
0.6
>>> _ + a
0.7
>>> _ + a
0.7999999999999999
>>> _ + a
0.8999999999999999
>>> _ + a
0.9999999999999999
```

זהו, סימנו את הדין גם ב-int וב-float. ליתר טיפוס המשתנים – יוקדשו – String, List, Tuple, Dictionary – פרקים נפרדים.

תנאים

התנאי הבסיסי ביותר הוא `if`. לאחר מכן `if` יבוא ביטוי בוליאני כלשהו. אם ערך הביטוי הוא `True`, אז יבוצע הקוד של אחר ה-`if`, ואם ערך הביטוי הוא `False`, אז יבוצע דילוג. מיד נבין עד להיכן מבוצע הדילוג. בינהים, נסקור את הדריכים השונות לבדוק את היחס בין משתנה לביטוי כלשהו.

שווין – הסימן `==` (פעמיים `=`) פירושו "אם צד שמאל של הביטוי שווה לצד ימין". הקוד בדוגמה הבאה ידפיס `:Yay!`

```
x = 21
if x == 21:
    print("Yay!")
```

אי שווין – הסימן `!=` פירושו "אם צד שמאל של הביטוי אינו שווה לצד ימין". הקוד בדוגמה הבאה ידפיס `:Yay!`:

```
x = 20
if x != 21:
    print("Yay!")
```

גדול / קטן / גדול שווה / קטן שווה – כל אחד מהסימנים `>`, `<`, `>=`, `<=` בודק אם התנאים מקיימים את היחס שמדובר בסימן. לדוגמה:

```
x = 20
if x <= 21:
    print("Yay!")
```

שווין למשתנה בוליאני – כאשר משווים משהו למשתנה בוליאני לא נהוג לכתוב `==`, אלא משתמשים בביטוי `is`. בהמשך הפרק נסביר מה ההבדל בין `==` לבין `is`:

```
x = True
if x is True:
    print("The truth!")
```

תנאי בוליאני מוקוצר – כאשר יש לנו משתנה בוליאני, צורת שימוש מקובלת נוספת היא פשוט לרשום if ואז את שם המשתנה:

```
x = True
if x:
    print("Got it?")
```

תנאים מורכבים

לעתים קרובות לא נוכל להסתפק רק בבדיקה אחת, אלא נצטרך תנאים מורכבים. כלומר, יש לבצע משהו רק אם מתקיים תנאי א' וגם תנאי ב', או שמתקיים רק אחד מכמה תנאים אפשריים, או שמתקיים תנאי א' אך תנאי ב' אינו מתקיים. במקרים כאלה, משתמש בתנאים המורכבים `and`, `or`, `not`.

תנאי `and` משמעו "וגם". לדוגמה, התנאי הבא יתקיים רק אם מתקיים גם `x > 20` וגם `x < 22`:

```
x = 21
if (20 < x) and (x < 21):
    print("Yay!")
```

כמובן שאנו יכולים לכתוב גם בצורה מוקוצרת, באמצעות :

```
x = 21
if 20 < x < 21:
    print("Yay!")
```

תנאי `or` משמעו "או". לדוגמה, כדי שההתנאי הבא יתקיים, מספיק ש-`x` יהיה שווה ל-21 או כל מספר מעל 30:

```
x = 31
if (x == 21) or (x > 30):
    print("Yay!")
```

תנאי `not` מבצע היפוך. מקובל להשתמש בו יחד עם `or`. לדוגמה:

```
x = 5
if x is not 5:
    print("Not true!")
```

כאשר מדובר בעבר בוליאני, הדרך המקובלת היא לרשום `soch` לפני שם המשתנה:

```
x = False
if not x:
    print("Yes!")
```

שימוש ב-`so`

בדוגמאות הקודמות רأינו שאפשר לכתוב תנאי `um == ==` וגם את אותו תנאי `um hebeto so`. מה ההבדל ביניהם?

התנאי `==` בודק אם שני הצדדים של התנאי מכילים את אותם ערכים.

כדי להבין את התנאי `so`, נזכיר שככל המשתנה שיש לנו נשמר במקום כלשהו בזיכרון. כדי לגשת למשתנה כלשהו, פייתו משתמש בכתובת של המשתנה בזיכרון. התנאי `so` בודק אם שני הצדדים של התנאי מצביעים על אותה כתובת בזיכרון.

בסירטון הבא יש הדגמה של ההבדל בין `so` ל `==`:

https://www.youtube.com/watch?v=0_dQpUtcubM

בלוק

בדוגמאות שסקרנו תמיד הייתה שורת קוד אחת לאחר תנאי `if`. האם אפשר לשימוש יותר משורה קוד אחת? כמובן! את מושג הבלוק הכי פשוט להבין מצפיה בקוד:

```
x = 21
if x == 21:
    print("Hi")
    print("x is...")
    print("21")
```

כל שלוש הפקודות שלאחר ה-`if` נמצאות בהזחה – אינדנטציה – של טאב אחד, או ארבעה רווחים 'יחסית לתנאי `if`'. כיוון שכולן נמצאות באותה הזחה, הן יבוצעו ככל אם התנאי יתקיים. תוכנת Pycharm מסמנת לנו את כל הפקודות השייכות לאותו בלוק, באמצעות החיצים האפורים מצד שמאל של הפקודות.

שימוש לב להבדל בין הקוד לעיל לבין הקוד הבא:

```

x = 21
if x == 21:
    print("Hi")
    print("x is...")
print("21")

```

השורה الأخيرة, שמדפסה 21, תרוץ בכל מקרה – בין שהתנאי מתקיים ובין שהוא אינו מתקיים. זאת מכיוון שהיא נמצאת מחוץ לבlok של תנאי ה-if.

נושא ההזחה הוא קרייטי בפייתון, מכיוון שבניגוד לשפות אחרות בהן יש סימונים שונים של "סוף בלוק", לדוגמה סוגרים מסולסלים, בפייתון סוף הבלוק נקבע רק לפי ההזחה. על כן הכרחי גם שהזחה תהיה עקבית – אנחנו לא יכולים לעשות לפעמים הזחה של שני רוחחים ולפעמים של ארבעה רוחחים. חשוב לציין גם שתוו' רווח וטאב שונים זה מזה. למשל, גם אם לנו נראה ששורה שיש בה טאב נמצאת בהזחה זזה לשורה אחרת שיש בה הזחה של ארבעה תוו' רווח, מבחינת פיותן אלו תווים שונים לגמרי. לכן, אם התחלנו לעשות הזחה עם טאים, רצוי שנמשיך רק עם טאים.

לא רק מה שנמצא בהזחה של טאב אחד שייר לבלוק. בתוך בלוק יכולים להיות עוד טאים, לדוגמה עקב שימוש בתנאי if נוספים. כל הפקודות שנמצאות בהזחה של לפחות טאב אחד מתנאי ה-if שלנו שייכות לוותו בלוק. לדוגמה:

```

x = 21
if x < 23:
    print("Lower than 23")
    if x < 22:
        print("Lower than 22")
        if x < 21:
            print("Lower than 21")
            print("I")
            print("Love")
            print("Python")
    print("Yes :)")

```

מה יודפס כתוצאה מ裏צת הבלוק? יורצו כל השורות חוץ מאשר אלו שנמצאות בבלוק של `x < 21`, כיון שהתנאי זה אינו מתקיים. פلت התוכנית יהיה:

```
Lower than 23
Lower than 22
Love
Python
Yes :)
```

תנאי else, elif

נניח שאנו רוצים להריץ קוד כלשהו אם תנאי מתקיים, וקוד אחר אם התנאי אינו מתקיים. במקומות שניצטרך לבדוק פעמים – פעם אם התנאי מתקיים ופעם אם הוא אינו מתקיים – אפשר להשתמש בפקודה `else`. אם התנאי שנמצא ב-`if`-ו אינו מתקיים, יבוצע הקוד בבלוק של `else`.

```
x = 20
if x == 21:
    print("21!")
else:
    print("Something else...")
```

מה אם יש לנו תנאי נוסף רוצים לבדוק? לדוגמה, אנחנו מתקשרים לחבר כדי לבוא איתנו לסרט. אם הוא לא יכול, אנחנו מתקשרים לאמא ושואלים מה יש לאכול בבית. אם אמא לא עונה – אנחנו נשארים בבית וצופים בשידור החזר של הסדרה `Friends`.

```
friend_is_free = False
mother_is_home = True
if friend_is_free:
    print("Going to the movies")
elif mother_is_home:
    print("Eating an apple pie")
else:
    print('Watching "Friends" on TV')
```

בדוגמה הנ"ל, הגדרנו משתנים בוליאניים וקבענו את ערכיהם כך שבסוף אכלנו עוגת תפוחים של אמא. ברגע שתנאי `elif` התקיים, התוכנית לא נכנסה אל תוך תנאי `else`. נῆפַה בטליזיה רק אם גם תנאי `if` וגם תנאי `elif` לא יתקיימו.



כיתבו סקריפט עם תנאים על המשתנה `age`. אם `age` שווה 18, הסקריפט ידפיס 'Congratulations'. אם `age` קטן מ-18, ידפס 'You are so young'. בידקו שהסקריפט שלכם עובד היטב באמצעות קביעת ערכים שונים ל-`age` ובדיקה.

לולאת while

עד כה רأינו דרכים לכתוב תנאי "חכם", כמו `age < 18`, או תנאי שפיטון מריצ' פעם אחת בלבד. לעיתים נרצה לבצע פעולה כלשהי כל עוד תנאי מסוים מתקיים. לדוגמה, נרצה לקרוא קלט מהמשתמש ולבצע את הוראות המשמש, כל עוד המשמש לא כתב 'Exit'. במקרים אלו שימוש ב-`if`-`else` הוא לא מספק טוב, כיון שאנו לא יכולים לדעת כמה פעמים התנאי שלנו צריך לזרע. אולי המשמש כתוב 'exit' כבר בהוראה הראשונה? אולי בהוראה העשירות? במקרים אלו נשתמש בלולאת `while`.

לולאת `while` מתחילה צפוי בהוראה `while`, ולאחריה יבוא תנאי אותו נגידר. לאחר מכן יש בлок של פקודות אשר יבוצעו בזוו אחר זו, ובסיוף הבלוק תהיה חזרה אל בדיקת התנאי.

```
while condition:
    # do something
    print("Hi")
    # return to the while condition
```

שים לב – בדיקת התנאי מתבצעת רק לפני הכניסה לבלוק. מה משמעות הדבר? נסו להבין מה ידפיס הקוד הבא (שורה 3 מעלה את ערכו של `age` ב-1):

```
age = 17
while age < 18:
    age += 1
    print("Not yet 18")
```

טעות נפוצה היא לומר שלא ידפס כלום, מכיוון שבתוך הלולאה, בשורה 3, הערך של `age` מועלה ל-18 ואז התנאי שבודק אם `age` קטן מ-18 כבר לא מתקיים ושורה 4 לא תבוצע. אך שורה 4 כן תבוצע, מכיוון שברגע שנכנסים לתוך הבלוק מרייצים את כל הפקודות שלו. הבדיקה של ה-`while` מתבצעת פעם אחת, בכניסה לבלוק, ולא כל פקודה מחדש. לכן המשפט 18 Not yet 18 ידפס פעם אחת. באיטרציה השנייה של הלולאה התנאי כבר לא מתקיים ולא תהיה כניסה לתוך הבלוק.

מה לדעתכם יקרה אם נריץ את הלולאה הבאה?

```
while True:
    print("Hi")
```

אכן, זהה לולאה אינסופית. הריצה שלה לעולם לא תסתיים. יש להיזהר מלולאות כאלה. נכיר כעת פקודה שימושית, במקרים שבהם אנחנו לא יודעים מראש כמה פעמים הלולאה שלנו צריכה לרוץ. ההוראה `break` אומרת לפיתון – צא מהבלוק שבו אתה נמצא כרגע. אם נשים `break` בתוך לולאה, ריצת הלולאה תיקטע ברגע שהמחשב יגיע ל-`break`. מה ידפיס הסקריפט הבא?

```
while True:
    print("Hi")
    break
    print("Bye")
```

שורה 2 תדפיס Hi. בשורה 3 אנחנו מורים למחשב לצאת מהלולאה, لكن שורה 4 לעולם לא תורץ (ואכן, שימו לב ש-`PyCharm` מסמן אותה בצלע בולט, ומדגיש לנו שכתבנו קוד שלulos לא יורץ).

מה ידפיס הסקריפט הבא?

```
while True:
    print("Hi")
    while True:
        print("Bye")
        break
```

אם אמרתם רצף אינסופי של "Hi" ו-"Bye" לסיוגין, צדקתם. ה-`break` שבשורה 5 יגרום לתוכנית לצאת מה-`while` שבשורה 3, אך לא מה-`while` שבשורה 1. لكن הלולאה שבשורה 1 היא עדין לולאה אינסופית, ואילו הלולאה שבשורה 3 נקטעת בכל פעם מחדש במהלך הריצה.

תרגיל – Take a Break



הדפיסו את כל המספרים בסדרת פיבונצ'י אשר ערכם קטן מ-10,000. חובה להשתמש ב-`while True!` תוכלן:
למצוא הסבר על סדרת פיבונצ'י בקישור הבא:

https://he.wikipedia.org/wiki/%D7%A1%D7%93%D7%A8%D7%AA_%D7%A4%D7%99%D7%91%D7%95%D7%A0%D7%90%D7%A6%27%D7%99

לולאות for

לולאות for הן שימושיות במיוחד כאשר יש לנו אוסף של איברים שאנחנו רוצים לבצע עליהם פעולה כלשהי, בוגדים לולאות while שהן שימושיות כאשר מרכיבים מספר לא ידוע של פעמים. בשפת פיתון, לולאות for נכתבות בצורה מעט שונה משפטות תכנות אחרות.

בפיתון, לולאת for מקבלת אוסף של איברים. לדוגמה, מספרים מ-1 עד 10, או ארבעה שמות של ילדים. בכלל איטרציה – מעבר על הלולאה – אחד האיברים מאוסף האיברים נתען לתוך משתנה שעליו רצה הלולאה. לאחר סיום האיטרציה, נתען האיבר הבא מאוסף האיברים וכך הלאה עד סיום כל האיברים באוסף.

לולאת for מתחילה במילה `for`, לאחר מכן יבוא שם של משתנה כלשהו שעליו רצה הלולאה (נקרא "איטרטור"), ולאחר מכן המילה `in` ולאחר מכן אוסף של איברים. בדוגמה הבאה אנחנו שמים כמה מספרים בתוך סוגרים מרובעים, מה שאומר שהם הופכים ל"רשימה", טיפולו של משתנה שנלמד עליו בהמשך:

```
for i in [0, 1, 2]:
    print(i*2)
```

פלט ההרצה יהיה:

```
0
2
4
```

כאמור, אפשר ליצור לולאות שרצות על כל מיני אוספים של איברים. לדוגמה:

```
for i in ['Ben Gurion', 'Sharet', "Begin", "Rabin"]:
    print(i + " was Israel's prime minister")
```

פלט הריצה יהיה:

```
Ben Gurion was Israel's prime minister
Sharet was Israel's prime minister
Begin was Israel's prime minister
Rabin was Israel's prime minister
```

מה אם נרצה לעשות לולאת `if` שעוברת על סדרה של מספרים? די לא נכון לכתוב את כל המספרים אחד אחריו... הפקנץיה `range` מייצרת סדרות של מספרים. היא מקבלת בתור פרמטרים התחלתה, סוף וקפיצה ויוצרת סדרת מספרים. לדוגמה:

```
range(3, 8, 1)
```

יצירת סדרה של מספרים אשר מתחילה ב-3, קטנה מ-8, ומתקדמים בקפיצות של 1.

דוגמה לסקריפט שמדפיס כמה סדרות:

```
for i in range(3, 8, 1):
    print(i, end=" ")
print()
for i in range(3, 8, 2):
    print(i, end=" ")
print()
for i in range(5):
    print(i, end=" ")
```

תוצאת הרצה:

```
3 4 5 6 7
3 5 7
0 1 2 3 4
```

הסבר: השורה הראשונה כאמור יוצרת רשימה של מספרים שמתחלים ב-3 ועצרים לפני 8, בקפיצות של 1. כל מספר מודפס ולאחריו יש טו רווח. בסוף הלולאה יש `print` רק כדי לרדת שורה. הלולאה השנייה היא זהה, פרט לכך שהיא קפיצות הן של 2. הלולאה השלישי מדגימה את ערכי ברירת המחדל של `range`: כאשר היא מקבלת רק פרמטר אחד, היא מניחה שיש צורך להתחיל מ-0 ולהתקדם בקפיצות של 1. במקרה אחריות, יש לפונקציה ערכי ברירת מחדל. אם לא מזנת קפיצה, אז נעשה שימוש בקפיצה של 1. אם לא מזנת התחלתה, מתחלים מאפס.

תרגיל (קדית: עומר רוזנבוים, שי סדובסקי)

כתבו לולאת `for` שמדפסה את כל המספרים מ-1 עד 40 (כולל).

תרגיל 7 בום (קדית: עומר רוזנבוים, שי סדובסקי)

הdfsso למסך את כל המספרים בין 0 ל-100 שמתחלקים ב-7 ללא שארית, או שמכילים את הספרה 7 לפי הסדר. השתמשו רק בפעולות חשבוניות! עזרה: פועלות מודולו – החזרת השארית מחלוקת – נכתבת בפייטון באמצעות סימן %. לדוגמה:

14 % 4

תחזיר 2 (14 % 4) שווה ל-3 עם שארית(2).

pass

מה אם נרצה שלולאה שכתבנו לא תבצע כלום? רגע אחד, למה נרצה לכתוב קוד שלא מבצע כלום? הדבר שימושי בתחום פיתוח קוד. אנחנו כותבים את שלד התוכנית, שמכיל לולאות ופונקציות, אבל משאים אותו רקות. כך אנחנו מסיימים במהירות את השلد ויכולים לבדוק שהתוכן של הקוד שלנו הוא נכון, לפני שאחננו מתחילה להתעסק עם המימוש עצמו. זה די שימושי כאשר כותבים תוכניות מורכבות. לכן קיימת הפקודה `pass`, פקודה שאומרת – אל תעשה כלום.

לדוגמה:

```
for i in range(5):
    pass
```

הlolאה זו תרוץ 5 פעמים ובכל פעם לא תעשה דבר. כרגע זה אולי לא נראה שימושי במיוחד, אבל כאשר נכתוב תוכניות עם פונקציות, זה יהיה די מועיל בשלב תכנון הקוד.

תרגיל מסכם (קדית: עומר רוזנבוים, שי סדובסקי)

הdfsiso למסך את כל המספרים מ-0.1 עד 5, בקפיצות של 0.1. אבל שימו לב – את המספרים השלמים צריכים להdfsio ללא נקודה עשרונית! בין כל שני מספרים יהיה רווח יחיד, לאחר כל מספר שלם תבוצע ירידת שורה.
בדוק כך:

```
0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1
1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2
2.1 2.2 2.3 2.4 2.5 2.6 2.7 2.8 2.9 3
3.1 3.2 3.3 3.4 3.5 3.6 3.7 3.8 3.9 4
4.1 4.2 4.3 4.4 4.5 4.6 4.7 4.8 4.9 5
```

טיפ:

חילוק עם שארית מתבצע על ידי האופרטור /, לדוגמה $6/4$ ניתן תוצאה 1.5 .

לעומת זאת האופרטור // מבצע חלוקה ללא שארית. לדוגמה $6/4$ ניתן תוצאה 1 .

פתרון יפה הוא באורך של 5 שורות קוד בלבד.

פרק 4 – מחרוזות



הגדרת מחרוזת

מחרוזת הינה אוסף של תווים. מגדירים מחרוזת באמצעות גרש יחיד או גרשאים. כך לדוגמה ניתן להגדיר משתנה בשם greeting אשר שווה למחרוזת Hello בשתי צורות, עם גרש יחיד או עם גרשאים:



```
>>> greeting = 'Hello'  
>>> greeting = "Hello"
```

והתוצאה היא זהה בשני המקרים.

למה טוב לזכור שאפשר להגדיר מחרוזות בשתי הדריכים? כי לפעמים נרצה להגדיר מחרוזת שיש בה את אחד הסימנים הללו. לדוגמה, אם נרצה להגדיר את המחרוזת up what לא יוכל לתחום אותה בגרש יחיד. עם זאת, נוכל לכתוב:

```
>>> greeting = "what's up"
```

נקודה נוספת שנוגעת למחרוזות היא שנותים לעיתים לבלב מחרוזות של ספורות עם המספר עצמו. לכן נבהיר זאת – המחרוזת '1234' אינה שווה למספר 1234. הבלבול נובע בכך שאם נעשה print בשני המקרים נקבל אותו הדבר – יודפס למסך 1234, אולם הסיבה לכך היא שכאשר מבקשים להדפיס מספר, פיתון מתרגם אותו לאחרורי הקלעים למחרוזת ואז מדפיס אותה. ההבדל בין המחרוזת למספר יתבהיר לנו ברגע שננסח לחבר להם מספר.

```

>>> 1234 + 5678
6912
>>> '1234' + 5678

Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    '1234' + 5678
TypeError: cannot concatenate 'str' and 'int' objects

```

מה קיבלנו? הפקודה הראשונה רצתה בהצלחה והדפיסה את תוצאה החישוב למסך. הפקודה השנייה כבר לא. קיבלנו הודעה שגיאה שאומרת "לא ניתן לחבר משתנה מטיפוס str עם משתנה מטיפוס int". זיכרו זאת בהמשך.



ביצוע `print`-`format`

עד כה בכל פעם שרצינו להדפיס משהו, היינו כתובים פקודות `print` ולאחריה את מה שרצינו להדפיס. זו שיטה מאוד נוחה כשרוצים להדפיס ערך ייחיד, אבל היא נהנית מעט מסורבלת כשרוצים להדפיס אוסף של ערכים. דבר זה נכון במיוחד אם חילק מההערכים הם מסוג `str` וחלק מהם מטיפוסים אחרים, כגון `int`. במקרה, אפשר להדפיס משהו שככל `str` ו-`int` בצורה הבאה, על ידי המרה של ה-`int` ל-`str`, אבל כאמור זה קצת מסורבל:

```

greet = "My age is"
age = 20
print(greet + " " + str(age))

```

פקודת `print` מסורבלת מכיוון שאנחנו צריכים קודם כל לחבר עם `+` את כל החלקים של המחרוזת החדשה שאנו מיצרים, להוציא סימן רווח בין החלקים השונים (אחרת הם יהיו צמודים והתוצאה לא תהיה אסתטית), ולהמיר את המספר למחרוזת. במקרים אחרות, זהו פתרון אפשרי אבל לא נוח.

ונכל להשתמש בפקודת `format`, ובמקום כל משתנה שאנו רוצים להדפיס, נשטים סוגרים מסולסלים. לאחר מכן נכניס הכל לתוך `format`. לדוגמה:

```
print("{} {}".format(greet, age))
```

או לדוגמה:

```
print("Hello! {} {}".format(greet, age))
```

חיתוך מחרוזת – string slicing

אפשר לחתוך חלקים ממחרוזות באמצעות הפקה:

```
my_str[start:stop:delta]
```

החותוך דומה לפונקציית `range` אותה הכרנו כבר.

אינדקס ההתחלה הוא `start`, בירית המוחלט שלו היא 0.

אינדקס הסיום הוא `stop`, בירית המוחלט היא סוף המחרוזת.

הפרמטר `delta` מצין בכמה אינדקסים קופצים, בירית המוחלט היא 1.

שימוש לב שגム ערכים שליליים מתקבלים באופן תקין. לדוגמה, אם נגידיר מחרוזת אז נראה ששלכל איבר יש גם אינדקס שלילי – קלומר מסוף המחרוזת:

```
>>> greeting = 'Hello!'
>>> greeting[-1]
'!'
>>> greeting[-2]
'o'
>>> greeting[-3]
'l'
>>> greeting[-4]
'l'
>>> greeting[-5]
'e'
>>> greeting[-6]
'H'
```

גם הקופציות יכולות להיות שליליות, קלומר אחרת. להלן כמה דוגמאות:

```
name = "Shrek"
print(name[1])
print(name[1:3])
print(name[1::2])
print(name[:])
print(name[:-1])
print(name[-1::-1])
```





השו את מה שחשבתם שיתקבל עם התוצאות הבאות:

h
hr
he
Shrek
Shre
kerhs

הסבר:

שורה 2 מדפיסה את האיבר באינדקס מס' 1 במחוזת. שימו לב שהו התו השני, כיוון שהאינדקסים מתחילה מ-0.

שורה 3 מדפיסה את האיברים שמתחלים באינדקס 1 עד (לא כולל) אינדקס 3.

שורה 4 מדפיסה את האיברים שמתחלים מאינדקס 1, מостиים בברירת המחדל (סוף המחרוזת) ומתקדים בקפיצות של 2.

שורה 5 מדפיסה את האיברים שמתחלים בברירת המחדל (תחילת המחרוזת), עד ברירת המחדל (סוף המחרוזת) בדילוגי ברירת מחדל, 1. למעשה זהה המחרוזת עצמה.

שורה 6 מדפיסה את האיברים שמתחלים בברירת המחדל, מостиים באינדקס 1- (לא כולל) – כלומר האיבר שלפני סוף המחרוזת.

שורה 7 מדפיסה את האיברים מהאיבר באינדקס 1- (האחרון) ועד לברירת המחדל (האיבר האחרון) בקפיצות של 1-, כלומר אחרת. במלים אחרות, מתחלים מהאיבר האחרון והולכים אחריה עד שגיעים לאיבר הראשון.

תרגילים (เครดיט: עומר רזנביים, שי סדובסקי)

כתבו סקריפט שמכיל משתנה מסוג `str`, בעל הערך 'Hello, my name is Inigo Montoya'. השתמשו רק בסיסים על המחרוזת והדפיסו את המחרוזות הבאות:

- 'Hello'
- 'my name'
- 'Hello my name is'
- 'Hello world'

פקודות על מחרוזות

פייתון מאפשרת לנו לעשות בקלות פעולות שונות.icut נראת דוגמה נחמדה לכך. נניח שיש לנו שתי מחרוזות ואנחנו רוצים ליצור מהן מחרוזת חדשה, צירוף של שתיהן. הדרך לעשות זאת היא פשוט להשתמש בסימן '+'. יש למחרוזות מתודות שונות, נדגיםicut כעת כמה מהן.

```
message = "Hello" + "world"
print(message)
print(len(message))
print(message.upper())
print(message)
print(message.find('o'))
```

נציג את פלט התוכנית לפני שנעבור לדון בכל שורה קוד:

```
Hello world
11
HELLO WORLD
Hello world
4
```

בשורה 1 מודגם החיבור של שתי מחרוזות באמצעות +.

בשורה 2 מודגם שימוש בפונקציה המובנית `len`, קיצור של `length`, אשרמחזירה את האורך של המחרוזת. בדוגמה זו, האורך של 'Hello' (שימו לב לרווח בסוף המילה) ביחד עם 'world' הוא 11 תווים.

בשורה 3 אנחנו פוגשים את המתודה `zpper`. מתודות הן כמו פונקציות, אבל של סוג משתנה ספציפי. בהמשך נלמד שמתודה היא בעצם פונקציה שמוגדרת בתוך מחלקה, אבל נשמר זאת לפרק אחר במחלקות. בכלל אופן, בשלב זה נשים לב בעיקר להבדל בצורת הכתיבה בין המתודה `zpper` לפונקציה `len`. המתודה `zpper` מגיעה אחרי סימן נקודה:

`message.upper()`

בניגוד ל-`len`, שמקבלת בתוך סוגרים את הפרמטר:

`len(message)`

המתודה `zpper` מחזירה את המחרוזת, כאשר כל התווים שלה הן אותיות גדולות. בשורה 5 אנחנו מדפיסים את `message` כדי להמחיש ש-`zpper` לא שינתה את ערכו של המשתנה `message` – כלומר, האותיות נותרו קטנות.

בשורה 6 אנחנו מכירים מתיודה בשם `find`, אשר מקבלת כפרמטר תו ומוחזירה את המיקום הראשון שלו בתחום המחרוזת. המיקום הראשון של האות 'o' בתחום 'Hello world' הוא אינדקס מס' 4.

dir, help

הדבר החשוב ביותר שיש לזכור לגבי מתודות של מחרוזות מגיע בעט: היכן אפשר למצוא את כל המתודות של מחרוזות? במיללים אחרות, אם אנחנו רוצים לעשות פעולה עם מחרוזת, האם יש משהו שאנו יכולים לעשות חוץ מאשר לנחש מה שם הפעולה?

ובכן, נכיר את הפונקציה המובנית `dir`. פונקציה זו מוחזירה לנו את כל המתודות של המשתנה. לדוגמה, אם נגיד `dir` מחרוזת בשם `message` וنعשה לה `dir`, נקבל את התוצאה הבאה:

```
>>> message = 'Hello World'
>>> dir(message)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__getslice__', '__gt__', '__hash__', '__init__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__formatter_field_name_split__', '__formatter_parser__', 'capitalize', 'center', 'count', 'decode', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

נحمد,icut אנחנו יודעים מה שמות כל המתודות שניתן להרץ על מחרוזת. אבל כיצד נוכל לדעת, לדוגמה, מה עושה המתודה 'count'?-can מוגעה לעזרתנו הפונקציה המובנית `help`. נבצע `help` על המתודה `count` של `message` ונקבל את התיעוד של המתודה:

```
>>> help(message.count)
Help on built-in function count:

count(...)
    S.count(sub[, start[, end]]) -> int

    Return the number of non-overlapping occurrences of substring sub in
    string S[start:end].  Optional arguments start and end are interpreted
    as in slice notation.

>>> message.count('el')
1
```

מוסבר לנו בדוגמה זו, ש-`count` מוצאת כמה פעמים מופיעה תת-מחרוזת בתוך מחרוזת. לדוגמה, כאשר נעשה `count` עם תת-המחרוזת 'el' קיבל ערך 1, וזה מכיוון ש-'el' מופיע פעם אחת בתוך 'Hello world'.

הדבר החשוב שלמדנו אינו המתודה `count`, אלא הידיעה שבכל פעם כשנרצה לבצע פעולה – על מחרוזות ועל טיפוסים אחרים – נדע היכן לחפש אותם ואיך לקבל עליהם מידע.

צירופי תווים מיוחדים ו-raw string

התו '\' הואתו מיוחד אשר קרי "escape character" והוא מאפשר ליצור מחרוזות עם צירופי תווים מיוחדים. ישנים תווים, שם נשים אותם אחרי '\\' הם לא יודפסו כמו שהם, אלא יקבלו משמעויות אחרות. אחד הצירופים הידועים ביותר הוא \\. אם נכתב חסן בחרוזת וננסה להדפיס אותה נקבל... נסו זאת בעצמכם.

להלן טבלה של הצירופים המיוחדים (מקור: <https://docs.python.org/2.0/ref/strings.html>)

Escape Sequence	Meaning
\newline	Ignored
\\\	Backslash (\)
\'	Single quote (')
\"	Double quote (")
\a	ASCII Bell (BEL)
\b	ASCII Backspace (BS)
\f	ASCII Formfeed (FF)
\n	ASCII Linefeed (LF)
\r	ASCII Carriage Return (CR)
\t	ASCII Horizontal Tab (TAB)
\v	ASCII Vertical Tab (VT)
\ooo	ASCII character with octal value ooo
\xhh...	ASCII character with hex value hh...

אבל מה אם יש לנו טקסט שכולל צירוף תווים מיוחד ואנחנו רוצים שהוא יודפס כמו שהוא? לדוגמה, יש לנו במחשב קובץ בשם txt.txt, שנמצא בתיקייה c:\cyber\number1\b52.txt. אנחנו רוצים להדפיס את שם הקובץ והתיקייה בה הוא נמצא. שימושו לבאר ההפסקה תבוצע באופן "רגיל":

```
print("c:\\cyber\\number1\\b52.txt")
```

```
c:\\cyber  
umber52.txt
```

לא בדוק מה שרצינו שיודפס...

יש לנו 2 אפשרויות לפתרון הבעיה. אפשרות אחת היא להכפיל כל סימן \'', כך:

```
print("c:\\\\cyber\\\\number1\\\\b52.txt")
```

ההכפלה אומרת לפיתון 'אנחנו באמת מתכוונים לסיון \'.'

האפשרות השנייה היא לכתוב לפני תחילת המחרוזת את התו z, שmagdir לפיתון שהכוננה היא ל-raw string. כלומר, עליו לנקח את התווים כמו שהם ולא לנסוט לחפש צירופי תווים בעלי משמעות.

```
print(r"c:\\cyber\\number1\\b52.txt")
```

קבלת קלט מהמשתמש input

את הפונקציה `input`(raw הכרנו בקורסיה בפרק הקודם). פונקציה זו משמשת לקבל קלט מהמשתמש. הפונקציה מדפיסה למסך מחרוזת לפי בחירתנו, ואת מה שהמשתמש מקליד היא מכניסה לתוך מחרוזת.

לדוגמה:

```
username = input("Please enter your name\\n")  
print("Hello {}!".format(username))
```

שימושו לבן לכך שהמחרוזת המודפסת שבחרנו להדפיס למשתמש כוללת את התו ח', וזאת כדי שקלט המשתמש יהיה בשורה חדשה, דבר זה נעשה מטעמי נוחות. התוצאה:

```
Please enter your name
Kalista
Hello Kalista!
```

תרגיל – אבניבי

זוכרים את שפת הב"ת? כתבו קוד שקולט משפט (באנגלית) מהמשתמש ומירגם אותו לשפת הב"ת. תזכורת: אחרי האותיות `s o e i` צריך להוסיף `d` ולהכפיל את האות. לדוגמה, עבור הקלט `ohev otach ani` יודפס: `Abanibi obohebev obotabach`

טיפ ליעול הקוד: כדי לדעת אםתו נמצא במחרוזת ניתן להשתמש בפקודה `in`. לדוגמה:
if 'a' in my_str:

**תרגילי strings (מתוך google class strings, בעיבוד גבהים)**

בצעו את תרגולי `strings`. הדרכה – לפניים קוד שיש בו פונקציות, בראש כל פונקציה ישנו תיעוד מה היא אמורה לעשות, אך הפונקציה ריקה. עליים לתוכנת את הפונקציה כמו שצורך כדי שהיא תבצע את מה שהיא אמורה לעשות. בפונקציית `main` יש קוד שבודק אם הפלט של הפונקציה זהה לפלט הצפי עבור מגוון קלטים. אם הקוד שלכם תקין, יודפסו הודעה `OK` למסך.

[lienק להורדת קוד הובץ הפיתון של התרגיל](http://data.cyber.org.il/python/ex_string.py):



לדוגמה, תרגיל `donuts`:

```
# A. donuts
# Given an int count of a number of donuts, return a string
# of the form 'Number of donuts: <count>', where <count> is the number
# passed in. However, if the count is 10 or more, then use the word 'many'
# instead of the actual count.
# So donuts(5) returns 'Number of donuts: 5'
# and donuts(23) returns 'Number of donuts: many'
def donuts(count):
    # ++++your code here++++
    return
```

בתרגיל זה אנו מתבקשים להציג מחרוזת שכוללת את מספר ה-`donuts` שקיבלו כפרמטר לפונקציה (בutor המשתנה `count`), אולם אם `count` הינו גדול מ-10, علينا להציג פשוט 'many'.

פתרון:

```
def donuts(count):
    if count < 10:
        return 'Number of donuts: ' + str(count)
    else:
        return 'Number of donuts: many'
```

תוצאת הרצה:

```
donuts
OK got: 'Number of donuts: 4' expected: 'Number of donuts: 4'
OK got: 'Number of donuts: 9' expected: 'Number of donuts: 9'
OK got: 'Number of donuts: many' expected: 'Number of donuts: many'
OK got: 'Number of donuts: many' expected: 'Number of donuts: many'
```

תרגיל מסכם – ד'אן ולז'אן (เครดיט: עומר רוזנבוים, שי סדובסקי)



כיתבו תוכנית שקולטת מהמשתמש מספר בעל 5 ספרות ומדפיסה:

- את המספר עצמו
- את ספרות המספר, כל ספרה בנפרד, מופרדת על ידי פסיק (אך לא לאחר הספר)
- את סכום הספרות של המספר

דוגמה לריצה תקינה:

```
Please enter a 5 digit number  
24601  
You entered the number: 24601  
The digits of this number are: 2,4,6,0,1  
The sum of the digits is: 13
```

הדרכה: בתרגיל זה אנחנו צריכים לבצע המרה בין סוגי טיפוסים שונים. זיכרו, שהפונקציה `raw_input` ממחישה מחרוזת של תווים. כזכור אם המשמש הzin 1234, תוחזר המחרוזת '1234'. אם אנחנו מעוניינים להשתמש במחרוזת כמספר, علينا לבצע קודם לכן המרה מטיפוס מחרוזת לטיפוס `int`. לשם כך נשתמש בפונקציה `int`.

לדוגמה:

```
>>> my_number = '1234'  
>>> int(my_number)  
1234
```

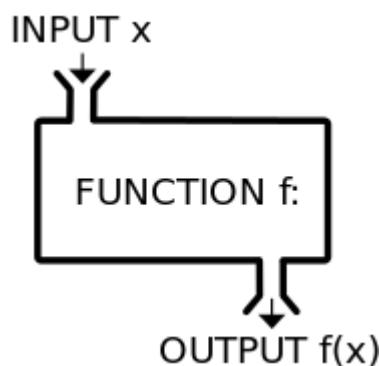
הערה: ניתן להניח שהמשמש הzin קלט תקין.

פרק 5 – פונקציות

כתיבת פונקציה בפייתו

פונקציה היא קטע קוד שיש לו שם ואפשר להפעיל אותו על ידי קריאה לשם. לקטע הקוד אפשר להעביר מידע, שעליו יבוצעו פעולות, וכמו כן שאפשר גם לקבל חזרה ערכיהם מהפונקציה.

במילים אחרות פונקציה היא קטע קוד שיש לו קלט `input`, ופלט `output`.



בואו נכתוב את הפונקציה הראשונה שלנו, שמדפסה>Hello:

```
def hello():
    """ Print Hello """
    print("Hello")
```

מתחלים עם המילה השמורה `def`. לאחר מכן שם הפונקציה, ולאחר מכן סוגרים שבתוכם אפשר לשים פרמטרים.

שימוש לב לדגשים הבאים:



- מומלץ מאוד לתת לפונקציה שם בעל משמעות, כך שגם מי שקורא את הקוד שלנו יוכל להבין מה רצינו.

- בתחילת הפונקציה יש לכתוב תיעוד, שמתאר מה הפונקציה עשויה. לティיעוד בתחילת פונקציות יש שם מיוחד – `docstring`, קיצור של `documentation string`. כתבים `docstring` בתוך

שלושה סימני גרשים כפולים רצופים, כמו בדוגמה. מדוע זה חשוב? מכיוון שאם נעשה `help` על הפונקציה, נקבל בחזרה את ה-`docstring` שלה. דבר מועיל מאוד כאשר אנחנו רוצים לשתף קוד!

```
help(hello)
```

`Help on function hello in module __main__:`

```
hello()
    Print Hello
```

פונקציה יכולה להיות מוגדרת עם פרמטרים של קלט או בלי פרמטרים. דוגמה לפונקציה בלי פרמטרים:

```
my_func()
```

דוגמה לפונקציה עם פרמטר אחד:

```
my_func(x)
```

הפונקציה `hello` היא כאמור דוגמה לפונקציה ללא פרמטרים. פונקציות עם פרמטרים נראות, לדוגמה, כך:

```
def print_message(message):
    """ Print a message
Args:
    message - a string
"""
print(message)
```

```
def print_messages(msg1, msg2):
    """ Print two messages
Args:
    msg1 - a string
    msg2 - a string
"""
print("{} {}".format(msg1, msg2))
```

שימוש לב לאופן התייעוד, ש כולל גם הסבר אודות הפרמטרים שמקבלת כל פונקציה. מדוע בתיעוד הם נקראים Args? זהו קיצור של "ארגומנטים" ארגומנט הוא המשתנה כפי שהוא נקרא על ידי מי שקורא לפונקציה. פרמטר הוא המשתנה כפי שהוא נקרא בתוך הפונקציה.

כעת נראה איך לקרואים לפונקציות עם ובלי פרמטרים.

```
def main():
    message = 'I am a function which receives one parameter'
    print_message(message)
    mes = 'I am a function'
    sage = 'which receives two parameters'
print_messages(mes, sage)
```

והתוצאה:

I am a function which receives one parameter
 I am a function which receives two parameters

נשים לב לכך שהפונקציה print_message נקראת עם ארגומנט בשם message, שזהה לשם הפרמטר כפי שמצוין בתוך הפונקציה. מאידך, הפונקציה print_messages נקראת עם ארגומנטים בשם mes ו-sage, שאינם זהים לשמות הפרמטרים שמוגדרים בתוך הפונקציה – msg1 ו-msg2. שתי האפשרויות חוקיות. בתוך הפונקציה messages מתבצעת העתקה של הארגומנט mes לתוך הפרמטר msg1 ושל הארגומנט sage לתוך הפרמטר msg2.

return

פונקציה יכולה להחזיר ערך, או מספר ערכים, על ידי return. לדוגמה:

```
def seven():
    x = 7
    return x
```

```
a = seven()
print(a)
```

השורה `a = print` תגרום להדפסת המספר 7.

אפשר להציג יותר מערך אחד:

```
def seven_eleven():
    x = 7
    y = 11
    return x, y
```



```
var1, var2 = seven_eleven()
print("{} {}".format(var1, var2))
```

התכנית תגרום להדפסת 11 7.

תרגיל



- כתבו פונקציה אשר מקבלת שני ערכים ומחזירה את המכפלה שלהם.
- כתבו פונקציה אשר מקבלת שני ערכים ומחזירה את המנה שלהם. זיכרו שחלוקת באפס אינו חוקי, במקרה זה החזרו הודעה "Illegal".

`None`

הבה נבחן את הפונקציה הבאה:

```
def stam():
    return None
```

מה היא מחזירה? מה יהיה ערכו של k אם נכתב כך?

```
k = stam()
```

ננו לעשוט `k` ונקבלו שהערך של `k` הוא `None`. הערך `None`, או "כלום", הוא ערך ריק. זהה מילה שומרה בפייתון. משתנה יכול להיות שווה לערך ריק, ואז הוא שווה `None`.

יש 3 מצבים בהם פונקציה מחזירה ערך `None`:

- לפונקציה אין `return` כלל.
- לפונקציה יש `return` אבל בלי שום ערך או משתנה. הכוונה לשורה שבה כתוב רק `return`.
- לפונקציה יש `return None`.

scope של משתנים

מה תבצע התוכנית הבאה?

```
def speak():
    word = "hi"
    print(word)
```

```
speak()
print(word)
```

כפי שאלוי שמתם לב, PyCharm מסמן את המילה `word` באדום. אם ננסה להריץ את הקוד, יודפס הפלט הבא:

```
hi
Traceback (most recent call last):
  print word
NameError: global name 'word' is not defined
```

אבל, איך זה יכול להיות שישנה שגיאת הרצה? הרי הפונקציה `main` קראה לפונקציה `speak`, אשר בה הוגדר המשתנה `word` ואף הודפס למסך. מדוע `main` לא מכיר את המשתנה `word`?

הסיבה היא שהמשתנה `word` הוגדר בפונקציה `speak` והוא קיימ רק בה. ברגע שייצאנו מהפונקציה `speak`, המשתנה `word` פשוט נמחק ואינו קיים יותר. מכאן שה-scope של המשתנה `word` הוא אך ורק בתוך הפונקציה `speak`.

נحدد את ההסבר ותור כד' נבין את ההבדל בין המשתנה גלובלי למשתנה מקומי, מקומי. נניח שכתבנו את הקוד הבא, שמו לב למשתנה החדש `name`:

```
def speak():
    word = "hi"
    print("{} {}".format(word, name))

name = "Shooki"
speak()
```

התוכנית תדפיס `hi`. מדוע? משום ש-`name` הוא משתנה גלובלי – משתנה שמוגדר מחוץ לכל הפונקציות, ולכן הוא מוכר בכלן. כלומר ה-scope של `name` הוא כל הסקריפט שלנו.

נתיקdem עוד שלב – כתעת אנחנו מגדרים את המשתנה `word` פעמיים. אחת כגלובלי ואחת כлокלי... מה ידפיס הקוד הבא?

```
def speak():
    word = "hi"
    print(word)

word = "bye"
speak()
```

ובכן, יודפס hi. מדוע? הרי אמרנו שמשתנה גלובלי מוכר גם בתחום פונקציה? נכון, אלא שבפונקציה speak אנחנו "דורסים" את המשתנה הגלובלי word עם המשתנה לוקלי בעל אותו שם. כתה כאשר נפנה בתחום speak למשתנה word, הוא כבר לא יכיר את המשתנה הגלובלי, אלא רק את מה שהוגדר לוקלית.

ניסוון נסוף... מה ידפיס הקוד הבא?

```
def speak():
    word = "hi"
    print(word)
```

```
word = "bye"
speak()
print(word)
```

!ידפו:

`hi``bye`

מדוע? את הדרישה של `hi` כבר הבנו. כאשר `speak` מוסימת את הרכיבה שלה, המשתנה המקומי `word` נמחק, וcut קורה משהו מעניין – מסתבר שהמשתנה הגלובלי `word` לא נמחק, אלא פשוט נשמר לצד. לפיתון יש מידרג של עדיפות: כאשר פונים למשתנה בתוך פונקציה, קודם כל מוחפש אם קיים משתנה מקומי כזה, ולאחר מכן אם קיים משתנה גלובלי. ההדרישה השנייה מתרחשת מתוך הפונקציה `main`, שמכירה רק את המשתנה `word`.

ניסו אחרון... מה ידפיס הקוד הבא?

```
def speak():
    word += " you"
    print(word)

word = "love"
speak()
print(word)
```

שגיאה!

```
Traceback (most recent call last):
  word += ' you'
UnboundLocalError: local variable 'word' referenced before assignment
```

המשתנה `word` אינו מוגדר. אבל מדוע? הרי הגדרנו משתנה גלובלי בשם זה? הסיבה היא, שכאשר אנחנו מבצעים פעולה שמשנה את ערכו של המשתנה בתחום פונקציה, כמו פעולה חיבור, פיתון מניח שהמשתנה שלנו יש עותק מקומי והוא מסזה לפעול עליו.

כעת נסקרו שתי שיטות לתקן את השגיאה בקוד. אפשרות א', והיא הפחות טובה, היא להשתמש במילה `global` בתחום הפונקציה, כר:

```
def speak():
    global word
    word += " you"
    print(word)

word = "love"
speak()
print(word)
```

בעקבות הריצה ידפו:

```
love you
love you
```

הפקודה `global word` אומרת לפיתון – 'ראה, אנו עומדים לעבוד בתחום הפונקציה עם המשתנה הגלובלי `word`'. אם ננסה לשנות את ערכו, עשה זאת בלי להזכיר שהוא אינו מוכר לך'.

האפשרות השנייה, היא להעביר לפונקציה `speak` את `word` בתור פרמטר, כר:

```
def speak(word):
    word += " you"
    print(word)
    return word

word = "love"
```

```
speak(word)
print(word)
```

בעקבות הריצה יודפו:

```
love you
love
```

כעת, מדוע האפשרות הראשונה אינה מומלצת? כי ששמתם לב, האפשרות הראשונה משנה את ערכו של המשתנה word גם מחוץ לפונקציה. הודף פעמיים `you`. לעומת זאת, הפונקציה שינתה את ערכו של המשתנה. דמיינו שאתם כתובים את הפונקציה `main` ומתקנת אחר כתוב את הפונקציה `speak`.Cutת תארו לעצמכם את הפעטהה, שהפונקציה שינתה ערך של משתנה בלי שהיא לכם מושג שהיא עשתה זאת! אם הייתם רוצים לאפשר לפונקציה לשנות את הערך של המשתנה, הייתם מעבירים לה אותו כפרמטר ודואגים להציב את ערך החזרה של הפונקציה במשתנה, כך:

```
word = speak(word)
```

זו הדרך הנכונה לשנות משתנה על ידי פונקציה – לקבל אותו כפרמטר ולהחזיר אותו עם `return` לקוד שקרה לו, כך:

```
def speak(word):
    word += " you"
    print(word)
    return word

word = "love"
word = speak(word)
print(word)
```

תרגילים

- כתבו פונקציה בשם `factorial` שמחזירה את התוצאה של $5!$ (5 עצרת). אין צורך להשתמש ברקורסיה.



- כתבו פונקציה בשם `beep` שמקבלת מחרוזת ומחזירה את המחרוזת ועוד `beep` בסופה.
- כתבו פונקציה בשם `sums_2nums` שמקבלת שני מספרים ומחזירה את המכפלה שלהם, או 0 אם התוצאה שלילית. שימו לב, אפשר לכתוב `return` יותר מפעם אחת בפונקציה.

פייטון מתחת למכסה המנוע (הרחבה)



בחלק זה נפרט שני נושאים הקשורים לפרק הלימוד עד כה:

- נמחיש את עקרון התרגום של פייטון לשפת מכונה בזמן ריצה
- נכיר את הפונקציה `ID` ואת האופרטור `is`
- נזכיר כיצד פרמטרים מועברים לפונקציות

זכור פייטון היא שפת סקריפטים, ולכן לא מתבצעת קומpileציה לקוד. במקרים אחרות, כל שורת קוד שאנו כתבים מומרת לשפת מכונה רק כאשר מגיע תורה של שורת הקוד להיות מorrect. נמחיש את העיקרונות הללו באמצעות תוכנית קטנה.

```
def check(num1):
    if True:
        print("OK")
    else:
        bla(blabla)
```

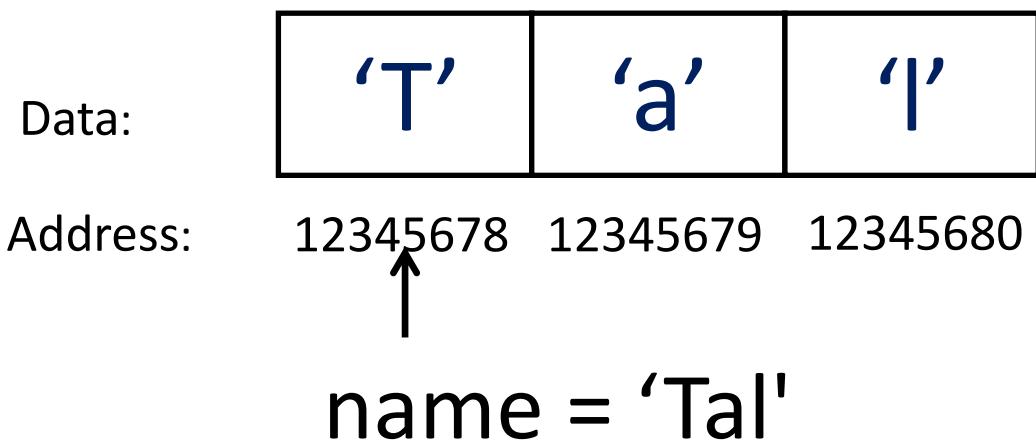
```
check(1)
```

כפי שאפשר לראות, הקוד מכיל קריאה לפונקציה בשם `a(a)` עם ארגומנט `blabla.blabla`. הן הפונקציה והן הארגומנט אינם מוגדרים. למרות זאת, אם נריץ את התוכנית נקבל תmid 'OK'. הסיבה לכך היא שתנאי ה-`if` מתקיים תמיד, וכן הקוד שבתוך תנאי `else` לעולם אינו מגיע להיות מושך. זהה המבשלה של העובדה שבייטון כל שורת קוד מפורשת ומתורגמת לשפט מכונה רק כאשר מגיע זמנה להיות מושכת.

בכך טמון גם סיכון: שורת קוד שנמצאת בתוך בлок של תנאי שמתקיים לעיתים נדירות עלולה להכיל שגיאות, שיבולו גם לקריאה התוכנית, והדבר לא יתגלה עד שהתנאי יתקיים.

`id`, `is`

זכרון המחשב מכיל את כל המשתנים שמוגדרים בתוכנית שלנו. כל משתנה נמצא בכתובת מוגדרת, כמוyar כאשר אנחנו מגדירים משתנה הוא מקבל כתובת מתווך כתובות שמקצתה לרכיבת התוכנית שלנו. עבור כל משתנה שמוגדר בתוכנית פי'יטון, שם המשתנה משמש כדי להצביע על כתובת בזיכרון. לדוגמה, אנחנו יכולים להגיד משתנה בשם `surname` והוא יצביע על כתובת 12345678. בכתובת 12345678 יוחסן הבית הראשון של המשתנה, אם המשתנה שלנו הוא בגודל של יותר מבית אחד אז יתר הבטים שלו מאוחסנים בכתובות העוקבות, לדוגמה 12345679, 12345680 וכך הלאה.



באמצעות הפונקציה `id` אנחנו יכולים לחושף את הכתובת בזיכרון שהוקצתה למשתנה מסוים, או לפונקציה כלשהי. כן, גם לפונקציה יש כתובת בזיכרון – המעבד צריך לדעת לאיזה מקום בזיכרון לקפוץ כדי להריץ את הפונקציה.

דוגמה לשימוש ב-`id`:

```
>>> name = 'Tal'
>>> id(name)
50183144L
```

האות `L` בסוף הכתובת בזיכרון, מסמנת שהוא מספר מטיפוס `Long`, כלומר זהה המוצג על ידי 64 ביט.

חישבו: האם לשני משתנים יכול להיות אותו `id`, ואם כן – מה הדבר אומר?

לשני משתנים יכול להיות אותו `id`, אבל רק אם הם מצביעים על אותה כתובת בזיכרון. נראה דוגמה:

```
>>> name_copy = name
>>> name_copy
'Tal'
>>> id(name_copy)
50183144L
```

הגדרנו משתנה בשם `name_copy` וקבענו שהוא שווה למשתנה `name`. בכך, גרמו לו למעשה להצביע לאותה הכתובת בזיכרון שאליה מצביע המשתנה `name`. לאחר מכן, בדקנו שהערך של `name_copy` הוא גם כן 'Tal'. כפי שניתן לראות, הוא זהה ל-`id` של `name`.

את האופרטור `is` הכרנו כאשר למדנו לכתוב תנאי `if` שונים. ראיינו שאפשר לבדוק בעזרתו `is` אם משתנה הוא `True` או לא. דוגמה:

`if result is True:`

...

`if results is not True:`

...

עת אנחנו יכולים להבין טוב יותר מהו `is`. האופרטור `is` בודק האם `to` של שני משתנים הוא זהה. אם כן – נקבל .`True`, אחרת – `False`. שימו לב שגם `None` – `False` וגם `None` – `True`.

```
>>> id(True)
1516068552L
>>> id(False)
1516068136L
>>> id(None)
1516022872L
>>> a = True
>>> b = False
>>> c = None
>>> a is True
True
>>> b is False
True
>>> c is None
True
```

נסכם בכך שפעיל את `is` על שני המשתנים שהגדכנו – `copy_name` ו-`name` – התוצאה היא `True` מכיוון שהם מצביעים על אותו מקום בזיכרון:

```
>>> name_copy is name
True
```

כדי להבין את ההבדל בין `is` לבין פועלות השוואת `==`, אתם מוזמנים לצפות בסרטון שבリンク הבא:

https://www.youtube.com/watch?v=0_dQpUtcubM

העברה פרמטרים לפונקציה

האם שאלתם את עצמכם איך פרמטרים מועברים לפונקציה? אם חשוב לכם לדעת איך הדברים עובדים, ומה זה `stack`, מומלץ לקרוא את ספר האסטטלי של גבאים. בקצרה, ישנן שתי שיטות להעברת פרמטרים לפונקציה.

Pass by value -

Pass by reference -

בשיטת `pass by value` מועבר לפונקציה **העתק** של הפרמטר. העתק נמצא על איזור בזיכרון שנקרו מחסנית, או `stack`, ומשמש פונקציות. דמיינו שהמורה מחזיק דף נייר שכותב עליו המספר 10. המורה ניגש למוכנות הציום ומcin לכל אחד מתלמידי הכתיבה העתק של דף הנייר עם הספרה 10 עליו. לכל תלמיד יש העתק של המספר. אם אחד התלמידים יכתוב על הדף שלו 11 במקום המספר 10, הדף של המורה לא ישתנה. באופן זהה, אם הפונקציה משנה את ערכו של משתנה שהועבר אליה בשיטת `value by pass`, היא למעשה לא משנה את ערך המשתנה עצמו, אלא העתק שלו. אי כך, ביציאה מהפונקציה ערכו של המשתנה יהיה כפי שהוא לפני כן.

בשיטת `pass by reference` מועברת לפונקציה – **הכתובת בזיכרון** שבו נמצא המשתנה. ערך המשתנה לא מועבר לפונקציה, ואם ברצונה לקרוא את הערך עלייה לגשת לזכרון בכתובת שנמסרה. דמיינו שכעת המורה שלנו מניח את הדף עם המספר 10 בתא שלו בחדר המורים, ובמקום לגשת למוכנות הציום הוא ניגש למגנוול ומשכפל לכל אחד מתלמידי הכתיבה מפתח לתא שלו. המורה מחלק את המפתחות לתלמידים שלו. אף תלמיד אין את הדף עם המספר 10, אבל הפעם התלמידים יכולים לגשת אל תא המורה, לקרוא את המספר 10 ואם הם רוצים – גם לשנות אותו. שינוי שיבצעו התלמידים ישפיע על הדף המקורי שבידי המורה. שימוש לבשபעם לא נוצרו לדף עותקים, יש רק עותק מקורי.

אז מה קורה בשפת פיתון? האם פרמטרים מועברים בשיטת `pass by value` או בשיטת `pass by reference`? הנה ניצור פונקציה וונביר לה פרמטרים:

```
def func(x):
    print("id(x): {} x: {}".format(id(x), x))
    x = "Bye"
    print("id(x): {} x: {}".format(id(x), x))

x = "Hi"
print(id(x))
func(x)
print(id(x))
```

אנחנו קובעים מחוזת בשם `x`. מדפיסים את `(x)` רק כדי שנוכל להשוות אותו לפני ואחרי הכניסה לפונקציה. לאחר מכן אנחנו קוראים לפונקציה וונבירים לה את `x` כפרמטר. בתוך ההפונקציה אנחנו בודקים את `(x)`, ולאחר מכן משנים את ערכו של `x` ואז בודקים שוב את `(x)`. לאחר היציאה מהפונקציה, שוב בודקים את `(x)` כדי לראות אם ה-`-p` שהיא בתוך ההפונקציה נשמר.

והנה מה שקיבלנו:

```
37058640
id(x) : 37058640  x: Hi
id(x) : 37058000  x: Bye
37058640
```

בשורט הדרישה השנייה אנחנו רואים שלפונקציה הועבר ה-*id* של x, שהרי ה-*id* מוחז לפונקציה ובתוך הפונקציה הם זרים. אם זה אומר שהפרמטר הועבר *by reference*

נראה כך, אבל אז מגיעה שורת הדרישה השלישי, ואני רואים שהעובדת שהפונקציה שינתה את x שינתה גם את (x)*id*. במליל אחרות, נוצר בתוך הפונקציה משתנה חדש בשם x, שיש לו id שונה מאשר ה-*id* שנמסר לפונקציה. ואכן, בשורת הדרישה הרביעית, שמתארחשת מוחז לפונקציה, אנחנו רואים שערכו של (x)*id* חוזר להיות כפי שהוא לפני הקריאה לפונקציה.

לסיום: פיתון מעבירה לפונקציות את הכתובת של הפרמטרים, אך ברגע שפונקציה מנסה לשנות אותם נוצר העתק, כך שהערך של המשתנה המקורי לא ישתנה.

רגע לפני שנסכם, נציין שככל מה שתבנו כעת נכון למשתנים מסוימים, שנקראים *immutable*, שעד עכשווי עוסקו בהם בלי לקרוא להם כך. מהם משתנים מסוג *immutable*? ומהם משתנים מסוג *mutable*? על כך נרחב כשןלמד על משתנים מסוג רשימה, *list*.

סיכום

בפרק זה למדנו אודוט פונקציות בפייתון. למדנו להגדיר פונקציה, להעביר לה פרמטרים וגם לקבל מהם ערכים. לאחר מכן רأינו שלמשתנים יש שוקס שבו הם מוגדרים, כלומר משתנה מוכך רק בפונקציה שבה הוא מוגדר. סקרנו אפשרויות שונות של שימוש במשתנים גלובליים בתחום פונקציה והגענו למסקנה שתמיד כדאי להעביר משתנים כפרמטרים לפונקציה, ובכל מקרה עדיף להמנע משינוי של משתנים בתחום פונקציה בלי שהidendical שנקורא לפונקציה מודיע לך.

לאחר מכן, במסגרת ההרחבה, רأינו איך פיתון עובד "מתחת למכסה המנו", כיצד מועברים פרמטרים לפונקציה. כעת אנחנו יכולים לכתוב פונקציות ולהשתמש בהם בצורה חופשית.

פרק 6 List, Tuple –

הגדרת List

עד כה למדנו אודות מספר טיפוסי משתנים: int, float, boolean.string. בפרק זה נלמד אודות שני טיפוסי משתנים שימושיים במיוחד – list (רשימה) ו-tuple.tuple. אין שם עברי ולכן ניצמד למונח הלועזי.

מהו list וכיצד מגדירים אותו? זהו אוסף של איברים, אוסף שאפשר להווסף אליו איברים או להוציא ממנו איברים. כמו רשימת קניות – אפשר להווסף לה מוצרים שברצוננו לרכוש או למחוק ממנו מוצרים. מגדירים list באמצעות סוגרים מרובעים, כר:

```
stam = [11, 'aaaa', 36.5, True]
```

בין כל שני איברים ישנו פסיק מיוחד. שימו לב שבתוך ה-list יכול להיות אוסף של איברים מסוגים שונים. בדוגמה לעיל יש לנו int, float, string, boolean.

```
[11, 'aaaa', 36.5, True]
```

אפשר לגשת לכל איבר ברשימה בצורה מאד דומה לדרך בה ניתן אל תווים במחרוזת – על ידי סוגרים מרובעים:

```
print(stam[0])
print(stam[1])
print(stam[2])
print(stam[3])
```

וירטואלי:

```
11
aaaa
36.5
True
```

כמובן שצריכה להיות שיטה ייעילה יותר להדפיס את כל איברי הרשימה, נכון? לו לא תמצא `for` צוועדת יד ביד עם `list`. כל מה שצריך לעשות הוא להוסיף את המילה `for`, ולבחר שם של משתנה שיהיה "איטרטור", כמו `stam` יוכל כל פעם ערך של איבר אחר ברשימה. כך:

```
for element in stam:  
    print(element)
```

הלוואה תרוץ 4 פעמים, כמספר האיברים ב-`stam`. בכל ריצה – איטרציה – של הלוואה, האיטרטור `element` קיבל ערך מtower `stam`, לפי הסדר שבו האיברים נמצאים בתower `stam`. בסופו של דבר תוצאה ההדפסה תהיה זהה ל贤תוצאת ההדפסה איבר איבר.

דמיון נוסף בין `list` ובין `string`, היא יכולה לאתגר חלק מהאיברים באמצעות סוגרים מרובעים ונקודותיים – יכולן לשמש ב-slicing. לדוגמה, אם נרצה להשתמש באיברים של `stam` רק מהאיבר השני באינדקס 2 ואילך, נוכל לכתוב:

```
stam[2:]
```

יתר החוקים של slicing שראינו על מחזורות (כגון בירית מבדל לערכי התחלת וסיום, קפיצות וכו') תקפים גם רשימה.

תרגיל

צורך רשימה בת חמישה איברים ומתוכה הוציאו את כל האיברים מהראשון עד האחרון בקפיצות של 2.

תרגיל (เครดיט: עומר רוזנבוים, שי סדובסקי)

בפייטון ישנה פונקציה בשם `sum`, אשר מקבלת רשימה של איברים שכולם מספרים (`int` או `float`) ומחזירה את סכום האיברים. לדוגמה:

```
sum([10, 11, 12, 0.75])
```

תחזיר את הסכום – **37.5**.

עליכם למשרוף פונקציה שהיא כמו `sum` אבל לא בדיק. נקרא לה `summer`. המינוח ב-`summer`, הוא שהוא יכול לעבוד עם כל טיפוס של משתנה, לא רק `int` או `float`, אלא גם משתנים מטיפוס `string` ואף `list`. כן, אפשר לחבר רשיומות! נסו להגדיר שתי רשיומות ולחבר אותן באמצעות סימן `+`.

לדוגמא, אם נכתוב:

```
print summer([10, 11, 12, 0.75])
print summer([True, False, True, True])
print summer(['aa', 'bb', 'cc'])
print summer([[1, 2, 3, 'a'], [4, 'b', 'c', 'd']])
```

נקבל:

```
33.75
3
aabbc
[1, 2, 3, 'a', 4, 'b', 'c', 'd']
```

הערה: אתם יכולים להניח שלא תצטרכו לעשות פעולות לא חוקיות כמו חיבור `int` ו-`str`. כמובן, כל האיברים מהרשימה ניתנים לחיבור זה עם זה.

Mutable, immutable

כעת נבון מה פירוש שני המושגים הללו, שהוזכרו בפרק הקודם. נחשב על ההבדלים בין מחרוזת לרשימה. ראיינו ש כדי לגשת לאיבר ברשימה או במחרוזת, כל מה שצריך לעשות זה להכניס את האינדקס של האיבר לתוך סוגרים מרובעים. לדוגמא:

```
>>> my_list = ['a', 'b', 'c']
>>> my_string = 'abc'
>>> my_list[1]
'b'
>>> my_string[1]
'b'
```

מכאן שגישה לקריאה של איבר פועלת באופן זהה ברשימה ובמחרוזת. אבל מה עם גישה לכתיבת איבר? מה יקרה אם ננסה לשנות את האיבר באינדקס 1 של הרשימה ושל המחרוזת? ננסה:

```
>>> my_list[1] = 'e'
>>> my_list
['a', 'e', 'c']
>>> my_string[1] = 'e'

Traceback (most recent call last):
  File "<pyshell#128>", line 1, in <module>
    my_string[1] = 'e'
TypeError: 'str' object does not support item assignment
```

ראינו שאט הרשימה אפשר לשנות בלי בעיה, ושלאחר הכתיבה הרשימה מכילה את הערך החדש. לעומת זאת מחרוזת אי אפשר לשנות – קיבלנו שגיאה. עכשו אפשר להבין את המושגים שבכותרת. מהهو שאפשר לשנות אותו נקרא **mutable**, מהמילה "מוטזית". מהו שאי אפשר לשנותו הוא **immutable**.

עם זאת, ראיינו שאפשר לשנות מחרוזות. כמובן, אפשר להגיד מחרוזת אז להגיד אותה שוב עם ערך אחר. איך זה אפשרי?

```
>>> my_str = 'Hello'
>>> my_str = 'World'
```

במקרה הזה, חשוב להבין שהמחרוזת `my_str` כבר אינה מצביעה על אותו מקום בזיכרון. אפשר לוודא את זה באמצעות ה-`id`, לפני ואחרי השינוי:

```
>>> my_str = 'Hello'
>>> id(my_str)
50183344L
>>> my_str = 'World'
>>> id(my_str)
31698456L
```

מבצע שינוי בראשימה, וניוכח שה-ID נותר זהה:

```
>>> my_list = [1, 2, 3]
>>> id(my_list)
49207944L
>>> my_list[1] = 4
>>> id(my_list)
49207944L
```

פעולות על רשימות

in

נלמד כמה פעולות שימושיות על רשימות. ראשית נרצה לבדוק אם איבר כלשהו נמצא בתוך רשימה. נניח שהגדכנו רשימת קניות בשם shopping_list. איך נוכל לדעת האם היא כבר כוללת apples? באמצעות שימוש במילה חוו, אותה כבר הכרנו:

```
shopping_list = ['Cheese', 'Melons', 'Oranges', 'Apples', 'Sardines']
if 'Apples' in shopping_list:
    print('There it is!')
```

כאשר כותבים ביטוי מהצורה 'איבר חוו רשימה', התוצאה תהיה true או False, מה שהופך את השימוש לנוח במיוחד עבור משפט if כמו בדוגמה. בהזדמנות זאת נזכיר שפיטון מתייחסת לאותיות גדולות וקטנות, כלומר אם נחפש apples במילון נקבל False.

נרצה להוסיף איבר לרשימה או להוציא ממנו איבר. לשם כך יש את המתודות pop ו-append. אנחנו קוראים להן מתודות ולא פונקציות, כי יש נקודה בין המשתנה שהן פועלות עליו לבין שם המתודה, בኒיגוד לפונקציות שמקבלות את שם המשתנה בסוגרים. נבין זאת יותר טוב כשנלמד תכונות מונחה עצמים OOP. השימוש ב-pop ו-append מתבצע כך:

append

המתודה append תמיד מוסיפה איבר בסוף הרשימה. למשל, אי אפשר להוסיף באמצעות איבר לאמצע הרשימה. המתודה מקבלת את האיבר שתוכזם להוסיף. לדוגמה:

```
>>> stam = [1, 2, 'a']
>>> stam.append('b')
>>> stam
[1, 2, 'a', 'b']
```

pop

הmethodה `pop` מקבלת אינדקס של איבר, מוציאה אותו מהרשימה ומחזירה את ערכו. לדוגמה, כדי להוציא מהתוך `stam` את 'a' צריך לעשות `pop` לאינדקס השני:

```
>>> stam.pop(2)
'a'
>>> stam
[1, 2, 'b']
```

כמובן שלכל אורך תהליך ה הכנסת וההוצאה, ה-`id` של `stam` נותר ללא שינוי.

sort

מה לגבי מין רשימה? המethodה `sort` מטפלת בכך. כל רשימה ש-`sort` תפעל עליה תהופיע להיות ממויינת. אם נשתמש ב-`sort` בלי להזין לתוכה פרמטרים, מספרים ימיינו לפי הגודל ומחרוזות ימיינו לפי סדר הופעתם במילון:

```
>>> stam = [3, 1, 7, 2]
>>> stam.sort()
>>> stam
[1, 2, 3, 7]
>>> stam = ['cat', 'dog', 'apple', 'elephant']
>>> stam.sort()
>>> stam
['apple', 'cat', 'dog', 'elephant']
```

אך מין יכול להתבצע בהרבה אופנים. אפשר למין מהקטן לגודל, מהגדול לקטן, לפי סדר האלף בית... ובכן, הנה נחקרו את `sort`. לפני כן, נזכיר כמה שלמדנו כאשר ביצענו חיתוך מחרוזות. אם יש לנו מחרוזת, אנחנו יכולים ליצור ממנה מחרוזות אחרת באמצעות סוגרים מרובעים, אשר בתוכם נמצא אינדקס התחלת, אינדקס סיום ועל כמה אינדקסים מדויגים. לדוגמה, התחלת באינדקס 2, סיום לפני אינדקס 12 ודילוג של 3 אינדקסים בכל פעם:

```
>>> my_str = 'Cyber class is cool'
>>> my_str[2:12:3]
'b a '
```

כפי שראינו, למרות שישנם שלושה פרמטרים, לא חייבים לכתוב את שלושתם. אפשר לשימוש בסוגרים המורובעים פרמטר אחד, שניים, או להשאיר פרמטר ריק לאחר נקודותים. לדוגמה:

```
>>> my_str[2:12]
'ber class '
```

במקרה שאנו לא מזינים דילוג, מתבצע דילוג של 1. כלומר, יש ערך ברירת מהדל – אם אנחנו לא מזינים ערך אחר, כאילו הזנו 1.

כעת נחזור למethode sort. נעשה עליה `:help`:

```
>>> help(stam.sort)
Help on built-in function sort:

sort(*)
    L.sort(cmp=None, key=None, reverse=False)
```

נראה שהוא מקבלת שלושה פרמטרים. הפרמטר האחרון נקרא `reverse` וערך ברירת מהדל שלו הוא `False`. כלומר אין "היפוך". ננסה את ערכו `True`:

```
>>> stam = [3, 1, 7, 2]
>>> stam.sort(reverse=True)
>>> stam
[7, 3, 2, 1]
>>> stam = ['cat', 'dog', 'apple', 'elephant']
>>> stam.sort(reverse=True)
>>> stam
['elephant', 'dog', 'cat', 'apple']
```

קיבliśmy את המיוון בסדר הפוך. מספרים מהגדול לקטן ומחרוזות הפוך מסדר הופען במילון.

נحمد, אבל מה אם נרצה לעשות מיוון יותר יצרתי? לא רק מהקטן לגודל או מהגדול לקטן, אלא כל מיוון שנרצה? המethode `sort` מקבלת בתור פרמטר את המפתח למיוון, פרמטר אשר נקרא `key`. בתור מפתח אנחנו יכולים לקבוע איזו פונקציה שאנו רוצים. נראה דוגמה. יש לנו רשימה שכוללת כמה מחרוזות. אנחנו רוצים לבצע מיוון לפי אורך המחרוזות. כלומר, המחרוזת 'zz', שהאורך שלו הוא 2, צריכה להופיע לפני המחרוזת 'aaa' שהאורך שלו הוא 3 ואחרי המחרוזת 'c' שהאורך שלה הוא 1 בלבד. כדי לנו, הפונקציה `len` מחזירה אורך של מחרוזת. לכן פשוט נעביר בתור מפתח את הפונקציה `len`, כך:

```
>>> words = ['aaa', 'c', 'zz', 'bbbb']
>>> words.sort(key=len)
>>> words
['c', 'zz', 'aaa', 'bbbb']
```

אנו יכולים גם להגיד פונקציה משלנו ולהעביר אותה בתור מפתח. שימו לב, שהfonקציה `len` צריכה להחזיר ערך כלשהו, שלפיו יתבצע המילוי, כמו שהfonקציה `len` ממחזירה מספר שהוא אורך המחרוזת.

 בצעו מילון של מחרוזות לפי התו האחרון במחרוזות. לדוגמה, `love` צריכה להיות לפני `cat` משום שהיא תומפיה לפני האות `t`.

ראשית, נצטרך להגיד פונקציה שמקבלת מחרוזת וממחזירה את התו האחרון שלה:

```
def last(my_str):
    return my_str[-1]
```

לאחר מכן נגדיר רשיימה ונזכיר ל-`sort` עם `:key=last`

```
beatles = ['John', 'Paul', 'George', 'Ringo']
beatles.sort(key=last)
print(beatles)
```

ותוצאה הדפסה שנתקבל:

```
['George', 'Paul', 'John', 'Ringo']
```

split

לעתים נקבל מחרוזת ונרצה להפריד אותה למחרוזות קטנות יותר ולשמור אותן ברשיימה. הדוגמה הקלטאית היא מחרוזת שכוללת משפט, ואנו רוצים להפריד אותה למילים בלבד. לשם כך קיימת המетодה `split`. כאמור, פועלת על משתנים מטיבוס מחרוזת, אך כיוון שהיא מוחזירה רשיימה נוכל עצה להבין יותר טוב את אופן הפעולה שלה.

הMETHOD `split` מקבלת כפרמטרתו או מחרוזת שלפיהם תתבצע הפרדה. כדי להבין את רעיון הפרדה, ניקח דוגמה מחרוזת שכוללת שמות של מספר סרטיים:

```
brad_pitt_movies = 'Fight Club#Seven#Snatch#Moneyball#12 Monkeys'
```

נרצה ליצור רשימה בה כל איבר יהיה שם של סרט. למשל הרשימה מכילה את התו '#' בתורתו מפריד בין שמות הסרטים. לנכון נקרא ל-split עם פרמטר '#':

```
brad_pitt_movies = brad_pitt_movies.split('#')
```

נקיבלו רשימה:

```
['Fight Club', 'Seven', 'Snatch', 'Moneyball', '12 Monkeys']
```

למקרה split יש גם ערך ברירת מחדל, שהוא סימן רווח. במקרה אחרות, אם לא נעביר ל-split שם פרמטר, כל הרוחים במחוזות יוסרו והמחוזות שבין הרוחים יוכנסו לרשימה. לדוגמה:

```
rule = 'The 1st rule of fight club is you do not talk about fight club'
rule = rule.split()
```

הפכנו את rule לרשימה. אם נדפיס את ששת האיברים הראשונים ברשימה נקבל:

```
['The', '1st', 'rule', 'of', 'fight', 'club']
```

join

זהו הפעולה הפוכה ל-split. יש לנו רשימה של מחוזות ואנו רוצים לחבר אותם יחד למחרוזת אחת. צורת הכתיבה כאן היא גם הפוכה ל-split – ראשית יבוא התו המפריד בין המחרוזות השונות (הגינוי שזה יהיה סימן רווח), לאחר מכן נקודה ו-חוץ עם שם הרשימה בסוגרים. כר:

```
rule = ' '.join(rule)
```

כעת אם נדפיס את rule נקבל את המחרוזת המקורית:

```
The 1st rule of fight club is you do not talk about fight club
```



תרגילי סיכום list (מתוך google python class, בעריכת גבהים)

הורידו את סקRYPT הפיתון שבקישור http://data.cyber.org.il/python/ex_list.py והשלימו את הקוד החסר בפונקציות.

Tuple

נכיר טיפוס נוסף של משתנה בפייתון – tuple. מגדירים `tuple` בעזרת סוגרים עגולים, כך:

```
my_tuple = (1, 2, 'a')
```

הטיפוס `tuple` הוא כמו `list`, אבל `immutable`. כלומר, אי אפשר לשנות את הערכים שבו. לשם מה זה שימושי? ובכן, `tuple` הוא דרך נוחה להציג ערכים מרובים מפונקציה. לדוגמה הפונקציה הבאה מחזירה 2 ערכים:

```
def silly():
    return 'hi', 'there'
```

אם נציב את ערכי החזרה שלה בתוך משתנה כלשהו, נוכל לראות שהמשתנה הזה הוא מסוג `tuple`. כאשר נעשה `print greet`, התוצאה תודפס בתוך סוגרים עגולים:

```
greet = silly()
print(greet)
print(greet[0])
print(greet[1])
```

תוצאות הדפסה:

```
('hi', 'there')
hi
there
```

במקרה זה ה-tuple ששמו `greet` קיבל את שני הערכים שהפונקציה החזירה. ראיינו שאפשר לפנות לכל איבר ב-`tuple` באמצעות סוגרים מרובעים. ממש כמו ברשימה.

שים לב לכך אלגנטית לקלוט מספר ערכים מפונקציה שמחזירה מספר ערכים:

```
first, second = silly()
```

cutת כל אחד מה משתנים `first`, `second` הוא מחוזת שמכילה את אחד הערכים שהוחזר מפונקציה.

נשאל – מדוע בכלל יש צורך ב-tuple? הרי היינו יכולים לבצע את אותן הפעולות באמצעות רשימה. לדוגמה, יכולנו ליצור פונקציה שמחזירה רשימה של ערכים ולא tuple. כאשר היינו קוראים לה ערכי החזירה היי נטענים למשתנים.

```
def stam():
    return [1, 2]

a, b = stam()
```

אם כך, מדועtuple?

הסיבה המעניינת קשורה לכך שבה פיתון מקצה זיכרון לרישומות. פיתון מניח שם הגדרנו רשימה, יתכן שנרצה להוסיף אליה ערכים באמצעות `append` או לשנות את הערכים השמורים בה. لكن פיתון צריך לדאוג שהיו לרשימה מוצבים שמאפשרים להוסיף איברים ולעורך איברים קיימים – יש לך עלות מסוימת בזיכרון. לעומת זאת, tuple הוא טיפוס שלא ניתן להוסיף לו ערכים וגם לא לשנות ערכים קיימים, שכן פיתון מקצה בדיק את כמות המקבימות שהגדכנו. ככלומר, אם אנחנו יודעים שאנו לא מתכוונים להוסיף איברים, הטיפוס tuple הוא חסוני יותר במקומו בזיכרון. כאשר אנחנו מחזירים ערכים מפונקציה, אנחנו יודעים כמה ערכים החזרנו, לכן השימוש בתuple הוא מתבקש.

סיכום

בפרק זה למדנו אודות שני טיפוסי משתנים שימושיים – `list` ו-`tuple`. סקרנו מוגדות ופונקציות שימושיות של `list`: בדיקה אם איבר נמצא ברשימה, הוספת והוצאת איבר, מיון רגיל, מיון לפי מפתח מיוחד. רأינו כיצד בעזרת `split` ו-`join` אפשר להמיר מחרוזת לרשימה ולהיפך, דבר שימושי כשרצחה לנתח טקסט ולעבד בו מילים בודדות.

במהלך הפרק התוודענו למושגים `mutable` ו-`immutable`. רأינו שרשימה היא משתנה `mutable` – ניתן לשינוי – בעוד מחרוזת היא `immutable`. לסיום רأינו כיצד `tuple` משמש להחזרת ערכים רבים מפונקציה.

פרק 7 – כתיבת קוד נכונה

עד כה למדנו לכתוב קוד פיתון בסיסי, אך עשינו זאת kali הkpfaה הרבה על איות הקוד שאנו כתבimos. מדוע חשוב לכתוב קוד פיתון "נכון"? הרי אנחנו יודעים למה אנחנו כתבemos כאשר כתבנו את התוכנית...? הבעייה היא שבעולם ה"אמיתי" סביר מאד שהקוד שכתבנו יהיה רק חלק מתוך מערכת גודלה יותר, שנכתבת על ידי צוות מתכנתים, מחלקת מתכנתים או אפילו קהילה של מתכנתים. כאשר אנשים אחרים יקראו את הקוד שכתבנו, הוא צריך להיות קרייא מספיק על מנת שהוא יוכל להשתמש בו. בנוסף, שמירה על כללי כתיבת קוד איות תסייע לנו למנוע כתיבה של שגיאות, וביחוד שגיאות כאלה לנו קשה במיוחד לאתר ולתקן. כמו כן, כתיבת קוד איות תסייע לנו במהלך כתיבת הקוד ותמנוע שכפול של פעולות קיימות, ובכך תחסוך לנו זמן יקר.

מהו בכלל קוד איות? בפרק זה נטמקד בכמה כלליים:

א. הקוד צריך לעבוד. כמובן, אם קוד נכתב במטרה לבצע פעולה מסוימת, עליו לבצע אותה בצורה נכונה.

לדוגמה, אם נכתב משחק מחשב קטן, המשחק צריך לזרז באופן חלק kali להיתקע.

ב. הקוד צריך להתחשב במרקם קצה, כולל במצב שבו הקלט אינו תואם הציפיות של המתכנת, ועל הקוד לא לקלוט כתוצאה מכך. לדוגמה, כתבemo פונקציה שמחשבת את השורש של מספר כלשהו שהמשתמש מקליד. המשמש הקליד מספר שלילי. הפונקציה צריכה לא לקלוט. גם אם המשמש הכנס ערף שאינו מספר, הפונקציה צריכה לא לקלוט.

ג. הקוד צריך להכתב עם חלוקה נכון לפונקציות. כל פונקציה צריכה לטפל במשימה אחת ולעשות את מה שהיא אמורה לעשות ורק את מה שהיא אמורה לעשות. למה זה בכלל משנה? הרי אם הקוד עובד, אז הוא עובד? ראשית, בתוכניות גדולות חלוקה נכון לפונקציות יכולה לחסוך זמן פיתוח רב. נאמר שפיתחנו תוכנה מורכבת ולא עשינו חלוקה לפונקציות. כאמור, אנחנו עובדים בצוות של מתכנתים. סביר שמתכנת שעבוד איתנו יצרך לקרוא ולהכיר את הקוד שלנו, וכטיבת כל הקוד בבלוק אחד תקשה עליו למד'. בנוסף, דרישות התוכנה משתנות לעיתים קרובות. כאשר נרצה להוסיף לתוכנה קטיע קוד, או לשנות מעט את האופן הפעולה שלה, לרוב גללה שהשינוי הקטן משפיע על קטיע קוד נוספים. לו היינו מתכנתים מראש עם חלוקה לפונקציות, סביר שהיינו צריכים לבצע שינוי רק בפונקציות בודדות.

ד. הקוד צריך לכלול שמות משמעותיים לפונקציות ומשתנים. לדוגמה, `12 L` אינם שם טוב למשתנה – מי שקורא אותו אין מושג מה הוא שומר בתוכו. לעומת זאת, `salary` (משכורת) הוא שם ממשתנה יותר ברור. הדבר תקין גם לגבי פונקציות. לפונקציה שמבצעת בדיקה אם קלט תקין אפשר לקרוא `stam`, או `g1`, שמות שמי שקורא אותם אינו מבין מהם מה הפונקציה מבצעת, או פשוט `input_inquiry_check`. מה יותר קרייא?

ה. הקוד צריך להיות מתוועד. כלומר, יש לכתוב docstring בתחילת כל פונקציה ויש להוסיף תייעוד שמסביר מה עשוינו, אך יש להמנע מתייעוד יתר, שכן מצין את המובן מאליו ואין מօסיף מידע חשוב לקורא. דוגמה לתייעוד יתר זה:

```
print(my_str)      # print the contents of my_str
```

באופן כללי, תייעוד צריך להסביר למה הקוד נראה כפי שהוא נראה, ולא איך או מה הקוד מבצע.

ו. הקוד צריך להתאים לkonvensiyot, או בעברית "מוסכמות". כאן הסיבה היא פשוט נוחות של מי שמנסה לקרוא את הקוד שלכם. למשל, שמות של קבועים ייכתבו באותיות גדולות.vr, אם נקרא קוד של מישוא וונמצא שם אותיות גדולות, קיבלנו מידע חשוב בלי מאמץ.

את הפרק הזה נקדים לנושא כתיבת קוד נכון. נחלק את הלימוד לתחי הנושאים הבאים:

- כתיבת קוד פיתון לפי konvensiyot PEP8

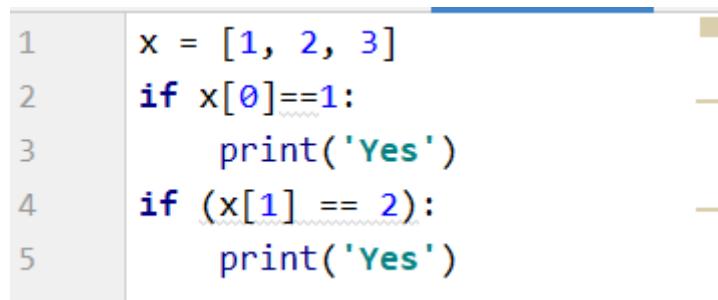
- חלוקה נכון של קוד לפונקציות

- בדיקת תקינות קוד ומקרים קצה באמצעות assert

PEP8

אוסף כללי כתיבה נפוץ של קוד פיתון נכון נקרא PEP8. מן הסתם לא נבעור כאן על כל הכללים – מדריך קצר מאת עומר רוזנבוים ושוי סדובסקי נותן מושג לגבי עיקרי הכללים: <http://data.cyber.org.il/networks/PEP8.pdf>

נתעכבר על השימוש ב-*PyCharm* על מנת לאתר טעויות PEP8 (כלומר, קוד שלא עומד בkonvensiyot שהוגדרו לפי PEP8) ולתקן אותן. שימוש לבקיטת הקוד הבא:



```

1  x = [1, 2, 3]
2  if x[0]==1:
3      print('Yes')
4  if (x[1] == 2):
5      print('Yes')

```

ניתן לראות שהריבוע בצד ימין למעלה הימן בצבע צהוב, לא יירוק אבל גם לא אדום. הדבר מראה לנו שהקוד יירץ בצורה תקינה, אבל יש בו טעויות PEP8. קל למצוא את הטעויות – האם אתם מבחינים בשני הקווים הצהובים שמתוחת לריבוע? נעמוד על אחד מהם:

```

1 x = [1, 2, 3]
2 if PEP 8: missing whitespace around operator
3     print('Yes')
4 if (x[1] == 2):
5     print('Yes')
6

```

The screenshot shows a code editor with Python code. Line 2 has a tooltip: "PEP 8: missing whitespace around operator". A yellow box highlights the line "if (x[1] == 2):". The code is as follows:

כתבנו לנו שיש בעיית PEP8, וקליק שמאלית מעביר אותנו לשורה הנכונה. במקרה זה, פירוט הבעיה הוא שחסרים רוחים לפני ואחרי סימן -=='.

יתכן שתמצם לב גם לסייע הנוראה הצהובה שדוולקת לצד שורת הקוד הבעיתית. לחיצה שמאלית עלייה תפתח לנו תפריט, ואם נבחר באפשרות Reformat file הבעיה תתוקן לבד!

אפשרות נוספת לגלוות בעיות PEP8 בקוד היא באמצעות סימן אפור דק מתחת לתווים הבעיתיים. דבר זה מסיע לנו להמנע מבעיות כבר בשלב הכתיבה.

תרגיל



העתיקו את התוכנית הבאה אשר יש בה שתי בעיות PEP8, והשתמשו ב-PyCharm כדי למצוא ולתקן את השגיאות.

```

x = [1, 2, 3]
if x[0]==1:
    print('Yes')
if (x[1] == 2):
    print('Yes')

```

לסייעו לנו נושא ה-PEP8, נתמך במספר דברים שחשוב שתשים לב אליהם.



א. תיעוד: כל פונקציה שאתם כתבים צריכה להכיל docstring, כפי שראינו בתחילת הפרק אוזות פונקציות. מומלץ להזכיר על הדוגמאות שմסבירות כיצד לכתוב docstring.

ב. שימוש בקבועים: אחת הטעויות הנפוצות של מתכנתים היא שימוש ב"מספרי קסם". לדוגמה, קוד שמדפיס 10 פעמים Hello, יכתב בדרך (הלא מומלצת) הבאה:

```
for i in range(10):
    print('Hello')
```

יש בצורת הכתיבה זו שתי בעיות. ראשית, לא ברור מה מציין המספר 10 (אפשר להבין זאת מקריאה הקוד, אבל בקוד מרכיב יותר זה ייקח זמן). שניית, אם נרצה שהתוכנית שלנו תדפיס 11 פעמים ולא 10, علينا לחפש בקוד את המספר 10 ולשנות אותו ל-11. נאמר שהמספר 10 חוזר על עצמו מספר פעמים בקוד – מכיוון שאנו רוצים להציג דברים שונים, וכל אחד מהם – עשר פעמים. במקרה זה נדרש לשנות את המספר 10 שוב ושוב. עם זאת, יכול להיות שבחילק מהמרקם המספר 10 יתיחס למקרה אחר – למשל במקרים מסוימים שאנו רוצים לקרוא מידע מהמשתמש, יוכל להיות שדווקא שם נרצה לשמור על המספר 10 כשלעצמו. ככל שהתוכניות שלנו יהיו מורכבות יותר, כך הזמן שנצרך להקדים זה יגדל.

האפשרות הטובה יותר היא שימוש בקבועים. לדוגמה:

```
TIMES_TO_PRINT = 10

for i in range(TIMES_TO_PRINT):
    print('Hello')
```

הגדרנו בתחילת התוכנית קבוע (שימו לב לאותיות的大寫), שרק מカリית השם שלו אנחנו כבר יודעים מה הוא עושה.aset אם נרצה להציג מספר שונה של פעמים, כל שנצחרך לעשות הוא לשנות את הקבוע, סימנו.

שימוש לב שפיטון לא מוגדר עבורנו שהקבוע שלנו לא ישנה. לעומת, עדין ניתן לבצע שינוי לערך הקבוע שלנו:

```
TIMES_TO_PRINT = 5
```

במקרה זה, הקונבנצייה אמורה לשמר علينا מפני עצמנו – המתכוונים. אנו מצינים את שם הקבוע באותיות גדולות כדי לזכור שהוא קבוע – ולא לשנות אותו בקוד שלנו.

ג. לא לדרס שמות מובנים בפייתון: לפיתון יש שמות מובנים – built-in names – שהפרשן שלו מכיר גם בלי שהגדכנו אותם. נניח שאנו ממעוניינים לקלוט מה משתמש מספר שמייצג אורך של משה. לדוגמה, אורך של ספר בעמודים, או אורך של משחק כדורי בדקות. מאוד מפתחה להציג את המשתנה בשם `len` – קיצור של `length`. הבעה היא שהשם `'len'` הוא שם מובנה בפייתון, כלומר יש לפיתון כבר פונקציה שנקראת `len` והיא יודעת להחזיר אורך של פרמטרים שהיא מקבלת. מה יקרה אם נדרס אותה? ככלום, פרט לכך שייתר לא יוכל להשתמש בה... לא תמיד, אבל כל עוד הסקריפט שלנו רץ.



חלוקת קוד לפונקציות

כאשר נכתב קוד, נחשוב איך אפשר לחלק את הקוד לפונקציות שעשוות דברים מוגדרים. בקוד איקוטי, כל פונקציה תעשה בדיקת מה שהיא צריכה לעשות – לא פחות ולא יותר. זיכרו – מטרתנו אינה לכתוב את הקוד הכי קצר או הכי יעיל, אלא קוד שעבוד ללא תקלות, ברור לקריאה ונitin בקלות להתקאה למשימות אחרות. מדוע זו מטרתנו? משום שלמעט מקרים נדירים, בהן המערכת שלנו רצתה ב מגבלות מסוימים, הרבה יותר סביר שיש לנו משאבי מחשב רבים אך מאידך אנחנו מוגבלים בכמות הזמן שיש לנו לטובת כתיבת קוד, או שהמחיר של באג במערכת הוא גבוה מאוד, או שאנו נדרשים לכתוב קוד לצורה שצוות או קהילת מתכוונים יכולים להבין ולהשתמש בו. מכל הסיבות האלה, קוד איקוטי הוא קוד שאין בו באגים, שהוא קל להבנה וşaפער בקלות להשתמש בו למשימות נוספות.

הבה ניקח משימה פשוטה יחסית ונדגים עליה צורות שונות של כתיבת קוד. בתור משימה ניקח את הבעה הבאה: ברצינו לקבל מה משתמש רשימה של מספרים, ולבזוק אם היא מקיימת חוקיות מסוימת. לדוגמה, שכל מספר הוא ממוצע שני המספרים שצמודים לו. הרשימה –

1, 3, 5, 7

היא רשימה בה כל מספר הוא ממוצע שני המספרים שצמודים לו (נכון, רשימה זו היא בהכרח סדרה חשבונית, אך יכולנו לבחור כל חוקיות אחרת, וכך נציג את הפתרון הכללי לבעיה התיכנوتית). נניח שהמשתמש מכניס רשימת מספרים, ובאשר הוא מעוניין לסיים את הכנסת הרשימה הוא מכניס 'STOP' בתור קלט. לדוגמה:

1

3

5

7

STOP

התוכנית שלנו צריכה לטפל בקלט המספרים ולבדוק אם החוקיות מתקינה. כזכור אם 3 הוא ממוצע של 1 ו-5, ואם 5 הוא ממוצע של 3 ו-7. את האיבר הראשון והאחרון אין צורך לבדוק, מן הסתם.



פתרון מודרך

הדרך הראשונה לפתור את התרגיל היא פשוט לקלוט את המספרים אחד אחריוiani השני, ובכל מספר שנקלט לבדוק אם מתקיים התנאי שהמספר האמצעי מקיים את התנאי שהוגדר (במקרה זה, ממוצע המספר שלפניו והמספר לאחריו). כזכור שאת שני המספרים הראשונים נקלוט ללא בדיקה – אי אפשר לבדוק אם התנאי מתקיים עליהם לפני שקלטנו את המספר השלישי. הקוד הבא מבצע את המשימה, אך המשיכו להסביר ורק אחר כך קיראו את הקוד:

```

index = 0
before = 0
middle = 0
ok = True
while True:
    user_input = input('Enter num, STOP to quit ')
    if user_input == 'STOP':
        break
    else:
        user_input = int(user_input)
        if index == 0:
            before = user_input
        elif index == 1:
            middle = user_input
        else:
            avg = float(before + user_input)/2
            if middle != avg:
                ok = False
                break
            before = middle
            middle = user_input
        index += 1
if ok:
    print('List is good')
else:
    print('List is not good')

```

הפתרון הנ"ל מבצע את המשימה, והוא גם יעיל למדי – כל מספר נבדק פעמי אחת בלבד, ואם מתרברר שהתנאי לא מתקיים אז יתר המספרים כלל לא נקלטים. אם כך, האם זהו קוד טוב? כלל וכלל לא. נפרט מדוע זה אינו קוד טוב ולא מומלץ כלל ללקחת ממנו דוגמה.

ראשית, ספר הלימוד בכוננה אינו מסביר כיצד עובד הקוד. נסו להבין בעצמכם איך הקוד מבצע את המשימה שהוא-Amor לבעצם. דמיינו שאתם מתכוונים במצוות וחברכם לצוות השair לכט את הקוד הזה ויצא לחופשה. לבטח תצליחו להבין את הקוד, אבל המשימה צפואה לגזול ממכם זמן רב יחסית לכך לא ארוך. שנית, את הסיבות המרכזיות שהקוד הזה קשה לקרואיה (וגם לכתיבתה ולדיבוגו!) הוא כמות התנאים שנמצאים בתוך תנאים. שימו לב לכך שיש שתי שורות שנמצאות בתוך לא פחות מ-5 רמות אינדנטציה. רמה אחת של פונקציה, רמה אחת של לולאה ו-3 רמות של תנאים. לא קל בכלל לעקוב אחרי תוכנית שיש בה תנאים בתוך תנאים בתוך תנאים, והדבר מתחבطة גם בזמן שלוקח לדבג את הקוד. במיללים אחרות, אם יש לכם באג בקוד שנראה כך, צפו לערב ארוך מול המחשב...

נסזה לשפר את הקוד שלנו. בניסיון השני, נפשת את הקוד שלנו באמצעות הפרדה בין המשימות בקוד. בעוד שבקוד של הניסיון הראשון פועלות קלילות המספרים הייתה משולבת בפעולות הבדיקה, בניסיון השני נפריד את קטעי הקוד – יהיה לנו קטע קוד שאחראי לקליטת כל המספרים וקטע קוד אחר שאחראי לבדוק אם התנאי מתקיים על סדרת המספרים. בכך, אפשר לטען שהקוד המקורי ביצע את המשימה בצורה יותר מהירה, אבלזיכרון המטרה שלנו אינה להציג את מהירות הריצה של הקוד אלא לצמצם את כמות הזמן שלוקח לכתוב, לדבג ולהזדקן את הקוד. שימו לב כיצד עצם החלוקה למשימות שונות מורידה את כמות התנאים בתוך תנאים. יש לנו כרגע לכל היוטר 3 רמות אינדנטציה, במקום 5:

```
nums_list = []
while True:
    user_input = input('Enter num, STOP to quit ')
    if user_input == 'STOP':
        break
    else:
        nums_list.append(int(user_input))
ok = True

for index in range(1, len(nums_list)-1):
    avg = float(nums_list[index-1] + nums_list[index+1])/2
```

```

if nums_list[index] != avg:
    ok = False
    break

if ok:
    print('List is good')
else:
    print('List is not good')

```

נحمد. ועדיין זה אינו קוד מוצלח. מדוע? ראשית, הקוד אינו מתווד. בתוך הקוד אין הסברים לדרך שבה הקוד פותר את הבעיה. אמנם, בדיקה אם מספר מקיים תנאי של להיות ממוצע של שני מספרים היא בדיקה מאוד פשוטה ואפשר להבין מ自然而 הקוד מה מתבצע, אולם ככל שנרצה לבצע שימוש מסווגות יותר כרך יותר יחסר לנו תיעוד. שנית, קשה להשתמש בקוד זהה למשימות אחרות. נסביר: נניח שחבר שעבוד איתנו נתקל גם הוא בעיה בה הוא צריך לעבור על רשימת מספרים ולבדק אם מתקיימת החוקיות הנ"ל. לחבומו לעובדה כבר יש תוכנית אחרת שדואגת לקליטת המספרים, כך שלא מתאים לו להעתיק את כל התוכנית שלנו. מה שהחבר יצרך לבצע הוא להעתיק חלק מקטע הקוד שלנו ולהתאים את שמות המשתנים לשמות בתוכנית שלו. מסובך! בשביל זה יש פונקציות. שלישיית, האם אתם מזהים אפשרות כלשהי שהקוד שلن יקרוס תוך כדי ריצה? מה לדעתכם יקרה אם המשתמש לא נעה להוראות שלנו והזין ערכים שאינם ספורות? בשורה 7, מתבצעת המרה של קלט המשתמש מחרוזת `-int`. אם המחרוזת אינה ברת המרה `-int` (לדוגמא, נסו להמיר את `'table'` ל-`int`...) אז הקוד יקרוס תוך כדי ריצה. אנחנו צריכים להבטיח שהקוד שלנו לא יקרוס גם אם המשתמש מאתגר אותו.

לסיכום, אנחנו צריכים להכין 3 שיפורים בקוד:

- תיעוד
- הוספת פונקציות שיבצעו קטעי קוד חשובים
- בדיקה שקלט המשתמש תקין ולא תיגרם בשום אופן קריסה תוך כדי ריצה

להלן הגרסה השלישית של הקוד, שטיפלה בשיפורים הנדרשים. שימו לב, ברור שהקוד יהיה ארוך יותר, אך בוואנו נראה אםicut קל יותר להבין מה הקוד עושה. כיצד לקרוא את הקוד ולהבין אותו במינימום זמן? מומלץ להתחילה מפונקציית `main` ולנסות להבין מה היא עושה. כדי להבין זאת, תוכלן להעזר בשמות הפונקציות – לעיתים אפשר להבין מה פונקציה עשויה בלי לקרוא אותה כלל. לאחר מכן התקדם אל הפונקציות, קיראו קודם כל את ה-`docstring` שלהן ורק לאחר מכן את הפונקציות עצמן. השוו בין כמות הזמן שלוקחת לכם הקריאה כתעת לעמודת הגרסאות הקודומות. מה יותר מהיר?

```

# The program receives numbers from the user and checks if
# each number is the average of the prior and next numbers

END_INPUT = 'STOP'
MIN_LIST_LENGTH = 3

def list_is_nums(input_list):
    """ Check if all the elements of a list are nums
    (int or float)
    Args:
        input_list - the list to be checked
    Return value:
        True / False
    """
    for element in input_list:
        if not element.isdigit():
            return False
    return True

def list_is_average(input_list):
    """ Check if a list meets the condition that each element
    is the average of the adjacent elements, like: 1, 3, 5, 7
    Args:
        input_list - the list to be tested
    Return value:
        True / False
    """
    nums = input_list
    for index in range(1, len(nums) - 1):
        avg = (nums[index-1] + nums[index+1])/2
        if nums[index] != avg:
            return False
    return True

def main():
    input_list = []
    while True:

```

```

user_input = input('Enter num, {} to quit '.
                   format(END_INPUT))
if user_input == END_INPUT:
    break
else:
    input_list.append(user_input)
if len(input_list) >= MIN_LIST_LENGTH and \
    list_is_nums(input_list):
    if list_is_average(input_list):
        print('List is good')
    else:
        print('List is not good')

if __name__ == '__main__':
    main()

```

דבר נוסף שהרווינו מכתיבה הקוד באופן זהה, הוא שנווכל להשתמש בפונקציות גם בתוכניות אחרות. זאת משומש השפונקציות לא עשות שימוש במשתנים גלובליים – כל מה שהן צリכות מועבר להן בתור פרמטרים. התיעוד המפורט עוזר לנו להבין כיצד לקרוא לכל פונקציה כדי להשתמש בה. דבר זה יחסור לנו גם זמן כתיבה בעtid.

כיצד נכוון לבצע חלוקה לפונקציות? להלן כמה דגשים:

א. הקפידו על כך שפונקציית `main` היא "מנהל העבודה" שלכם. כלומר, כל ניהול הזרימה של התוכנה מתבצע על ידי `main` ולא על ידי פונקציה אחרת. אל תיצרו פונקציית `choose` שרוב מה שהיא עשוה לקרוא לפונקציה אחרת שמבצעת את מרבית העבודה במקומה.

ב. מהם הדברים הייחודיים לפונקציית `main`? רק הפונקציה `main` עוסקת בקלט-פלט ורק הפונקציה `main` מדפיסה דברים למשתמש.

ג. הקפידו על כך שכל הפונקציות האחרות מבצעות משימה מוגדרת ספציפית. כל פונקציה מבצעת משימה אחת.

ד. קיראו לפונקציות שלכם בשמות משמעותיים, שימושים על מה הן עושים.

ה. אם לפי התיעוד שנთתם לפונקציה היא אמורה לעשות משימה מסוימת, אל תנתנו לה משימות נוספות. כתבו פונקציה אחרת אם יש לכם עבודה נוספת.

- . צרו פונקציות שיש להן ערך מוסף כלשהו. מתכוונים מתחילה מוטים לעיתים לכתוב פונקציה שכל מה שהיא מבצעת הינו קריאה לפונקציה אחרת. אין בכך טעם.

assert

מה דעתכם, האם יש עוד מה לשפר בגרסה האחרונה של הקוד? נשאל את עצמנו – איך אנחנו יודעים שהפונקציות שתכתבו אכן עובדות? כמובן, יכול להיות שהן עובדות ברוב במקרים, אך במקרה קצהה הן קורסות. דמייננו פונקציה שמבצעת חילוק – יכול להיות שהיא עובדת כמעט תמיד, אך קורס את המכנה הוא אף...

בנוסף, יכול להיות מצב בו הפונקציה שלנו עובדת, גם במקרה קצהה, אך ביצענו בה שיפור. השיפור גורם לכך שבמקרה קצהה הפונקציה שלנו כבר לא עבדה היטב, או בכלל.

נרצה למצוא דרך להריץ בקלות בדיקות על פונקציה, בין שאנו כתבנו אותה ובין שאנו משתמשים בפונקציה שימושה אחר כתוב. הבדיקות יאפשרו לנו לגלות גם אם הפונקציה רצה באופן תקין וגם אם גרמנו לבעה בעקבות שינוי שעשינו בקוד. ישן מספר דרכים לבדוק את הקוד שלנו, ואנו נזכיר את הדרך פשוטה ביותר – assert.

כדי להשתמש ב-`assert`, כתבים `assert`, לאחר מכן ביטוי כלשהו, שיכל להיות `True` או `False`. במקרה שלנו נכתב שם של פונקציה, לאחר מכן סוגרים עם קלט לפונקציה, ולאחר מכן את הפלט הצפוי מהפונקציה עבור הקלט הנ"ל. ניקח לדוגמה את הפונקציה `list_is_average`, אשר מקבלת רשימה ובודקת אם כל איבר בה הוא ממוצע של המספר שלפניו ואחריו. אם כן, הפונקציה מחזירה `True`, אחרת `False`. נוסיף לתוכנית שלנו `assert`'ים מיד בתחילת ה-`main`, שיבדקו גם מצבים בהם הפונקציה צריכה להחזיר `True` וגם מצבים בהם הפונקציה צריכה להחזיר `:False`:

```
def main():
    assert (list_is_average([1, 2, 3, 4])) is True
    assert (list_is_average([1, 1.5, 2, 2.5, 3])) is True
    assert (list_is_average([1, 2, 4])) is False
```

החלק המעניין ב-`assert` הוא להכניס קלטים מיוחדים, על מנת לבדוק מקרה קצה. נתמקד-cut בפונקציה `sums`.



- א. כתבו `Assert`'ים עבור הפונקציה הזו. אחד שייחסיר ערך `True` ואחד שייחסיר ערך `False`.
- ב. האם אתם יכולים לחשב על ערך כלשהו שעלול להחזיר תוצאה בלתי צפואה? התשובה בסעיף הבא

ג. מה יקרה אם הפונקציה `sums_is_nums` מקבל רשימה ריקה? בידקו זאת.

במקרה של רשימה ריקה נרצה לוודא שיוחזר לנו `False`, לא `cr`? נבדוק:

```
assert list_is_nums([]) is False
```

בשלב הרצת התוכנית נקלט את הפלט הבא:

Traceback (most recent call last):

```
    assert list_is_nums([]) is False
```

AssertionError

מה קרה כאן? קיבלנו `AssertionError`. שגיאה שאומרת לנו – "ביקשتم שנודיעו אם הפונקציה הנבדקת מחזירה ערך שונה מהערך שציפיתם לו, ואכן זה מה שקרה". זאת מכיוון שהפונקציה `list_is_nums` ממחישה `True` אם היא מקבלת רשימה ריקה. אכן, היה עוד מה לשפר בתוכנית שלנו, וגילמו זאת באמצעות ה-`assert`. כדי לתקן זאת, נדרש להוסיף לפונקציה `sums_is_nums` בדיקה האם הפונקציה קיבלה רשימה ריקה.

מספר דגשים:

- הפונקציה `list_is_average` לא כוללת `assert` עם רשימה ריקה. זאת מכיוון שגם הגענו לפונקציה הזאת, זה אומר שעברנו כבר את הבדיקה שאורך הרשימה גדול-מינימום (קבוע שערכו 3). צריך להוסיף לтиיעוד של `list_is_average` שהיא מקבלת רשימה שאורכה 3 לפחות, כדי שמי שישתמש בה בעתיד – אולי בתוכנית אחרת – יידע זאת.

- הפונקציות שלנו לא כוללות הדפסות. זאת מכיוון שאין אפשרות לבדוק הדפסות בעזרת `assert`, שבודק רק ערכים שהפונקציה מחזירה. לכן, הדרך המקובלת היא שהפונקציה מחזירה ערך, והקוד שקרה לה מופיע מה שצריך לפי הערך שהוחזר. בדיקן כמו בדוגמה, בה הפונקציה `list_is_average` החזירה רק `True` וההדפסה בוצעה בשורות הקוד שאחרי הקראיה לפונקציה.

זהו סימנו את הדיון במשימה שהוזגה בתחילת הפרק. עברנו ארבע גרסאות קוד שונות והגענו לקוד אליו שאנו – עובד, נוכן לקרוא ובודק.

עד כה ראיינו שימוש ב-`assert` לבדיקת פונקציות שמחזירות רק `True` או `False`. כמובן שאפשר להשתמש בו `assert` לבדיקת כל פונקציה. לשם המראה, נגדיר פונקציה פשוטה שמבצעת חלוקה בין שני מספרים. חשוב לציין שזו פרטיקה תכנותית לא טוביה לכתוב פונקציה שמבצעת דבר מה פשוט עוטף פונקציה `key` מת של פיתון, אולם במקרה זה אנחנו רוצים להמחיש נקודה מסוימת:

```
def my_div(num1, num2):
    """ Return the division of num1 by num2 """
    return num1/num2
```

בתוֹר הַתְּחִלָּה נְבֹדֹק שֶׁפּוֹנְקִצְיהָ שֶׁלּנוּ עֲוֹבְדָת הַיְּטֵב בְּמַקְרִים הַ"רְגִּילִים":

```
def main():
    assert my_div(6, 4) == 1.5
    assert my_div(6, 1) == 6, 'Not the expected result'
```

פעולות ה-`assert` הראשונה בודקת אם התוצאה היא צפוי. פעולה ה-`assert` השנייה כוללת דבר נוסף – הודעת שגיאיה שתודפס במקרה שהערך שיתקבל לא יהיה שווה לערך הצפוי. במקרה זה יודפס `'Not the expected result'`, אך כמובן שאפשר להdfsיס כל הودעה. מומלץ כמובן שההודעה תכלול מידע אודוט השגיאה, כך שמי שMRIIZ יוכל בקלות לדבג ולמצוא את מקור הבעיה.

כעת תורכם – אילו עוד פעולות `assert` כדאי לעשות על `my_div()`? נסו לחשוב על מקרים קצה.

ובכן, הדבר הראשון שמומלץ לבדוק בחלוקת היא התמודדות עם חלוקה באפס. אם הפונקציה קורסת זו בעיה, והיינו רוצים שבמקרה של חלוקה באפס תוחזר לנו הודעת שגיאיה כגון `'Can not divide by zero'`.

בנוסף, מה יקרה אם נעביר ל-`my_div()` ערכים שאינם מספרים? גם כאן, היינו רוצים לקבל בחזרה מהפונקציה הודעה כגון `'Parameters are not numbers'`. באופן זה מי שקורא לפונקציה עם ערכים לא נכונים לא יגרום ליריסוק התוכנית. במקרים כאלה אנחנו נבצע `Assert` בלי ערך חרזה צפוי, רק לוודא שהפונקציה לא קורסת תור כדין ריצה:

```
assert my_div(6, 0)
assert my_div('hi', 2)
```

תרגיל מסכם – **Deja Vu** (เครดיט: עומר רוזנבוים, שי סדובסקי)



כיתבו תוכנית שקולטת מהמשתמש מספר בעל 5 ספרות ומדפיסה:

- את המספר עצמו
- את ספרות המספר, כל ספרה בנפרד, מופרדת על ידי פסיק (אך לא לאחר הספרה الأخيرة)
- את סכום הספרות של המספר

רגע, מה זה? ראיתי כבר את התרגיל הזה! יש לי דז'ה!!...

נכון מאד ☺ רק שהפעם, לא ניתן להניח שהמשתמש העביר קלט תקין של 5 ספרות. במקרה שבו המשתמש הכניס קלט לא תקין, נבקש מהמשתמש להכנס שוב קלט – עד שנתקבל קלט חוקי. לדוגמה:

Please insert a 5 digit number:

Hello!

Please insert a 5 digit number:

24601

You entered the number: 24601



The digits of the number are: 2, 4, 6, 0, 1

Déjà vu happens when the code of the matrix is altered.

לתוכנית שלכם אסור לקרואו בשם אופן. הקפידו על כל הדברים שלמדו בפרק זה – חלוקה נכונה של הקוד לפונקציות, שימוש ב-PEP8, תיעוד ובדיקה פונקציות על ידי assert.

סיכום

פרק זה הتمקד בשדרוג יכולות התכונות שלנו. התחלנו מנושא שנראה טכני למדי במבט ראשון – שמיירה על קונבנציות לפי PEP8 – אבל רأינו את החשיבות של נושא זה לכתיבה קוד קרייא, שמתכונויות אחרים יכולים להבין בקלות ולעשות בו שימוש.

לאחר מכן התרמודנו עם אחד המחשומים המרכזיים שעומדים בפני מתכונויות מתחילה: כתיבת תוכנית שלא רק מבצעת את מה שהיא צריכה לבצע, אלא גם מחולקת למשימות שבוצעות כל אחת על ידי פונקציה אחרת. כתיבת קוד בדרך זו היא הדרך הארכאה אבל המהירה. הקוד שלנו גם יותר קל לדיבוג וגם יותר קל לשימוש חוזר.

לבסוף רأינו איך `assert` עוזר לנו לבדוק שהקוד שולמו תקין ועובד היטב. למדנו שכשר כתובים פונקציה, צריך לחשב על כל מקרי הקצה ולבדוק אותם באמצעות קריאה מהתוכנית הראשית.

פרק 8 – קבצים ופרמטרים לסקרייפטים

בפרק זה נלמד שני נושאים שימושיים ביותר כTİת תוכניות פִי'יתון. הראשון, שימוש בקבצים – נראה איך פותחים קובץ לקריאה ולכתיבה. השני, הערות פרמטרים לסקרייפטים – כלומר היכולת להריץ סкриיפט עם מידע שיגרום לסקרייפט לזרע בצויה מוגדרת, למשל לבקש מהמשתמש להזין ערך כלשהו תוך כדי ריצת הסкриיפט. בתור תרגיל מסכם נשלב את שני הדברים יחד – נריץ סкриיפט שמקבל כפרמטר שמות של קבצים ופועל עליהם.

פתחת קובץ

בפי'יתון ישנה פונקציה מובנית בשם `open`, אשר מקבלת בתור פרמטר שם של קובץ. שם הקובץ צריך להיות מחוזת, כאשר מומלץ לשם לפניו את הסימן `,`, שlify שלמדו מסמן מחוזת `raw`, כך שם הקובץ יוזן כמו שהוא ולא תבצע המירה לתווים מיוחדים. חיברים להעביר לפונקציה גם את אופן פתיחת הקובץ, לדוגמה אנחנו יכולים לפתוח קובץ לקריאה או לפתוח קובץ לכתיבה. אופן הפתיחה נקרא `mode`. נזכיר חלק מה-`mode`'ים השונים (טיאור מלא נמצא ב-<https://docs.python.org/2/tutorial/inputoutput.html> בסעיף 7.2):

- לכתיבה של טקסט נשימוש ב-`w`, קיצור של `write`
- לקריאה של טקסט נשימוש ב-`r`, קיצור של `read`
- אם נרצה לכתוב מידע לקובץ בלי לדרכו את המידע המקורי, נשימוש ב-`a`, קיצור של `append`. אם לא נפתחvr את הקובץ, אלא נשימוש ב-`w`, כל כתיבה שנכתבו לקובץ תתחיל מתחילה הקובץ – ולמעשה התוכן של הקובץ ימחק בכל פעם שנרצה לכתוב אליו.

לא כל הקבצים הם קבצי טקסט. לדוגמה, תמונות נשמרות בפורמט בינארי. אם נפתח תמונה באמצעות כתוב לא מוכן לקרוא את התוכן שלה. כדי לטפל בקבצים בינאריים יש צורות מיוחדות של קריאה וכתיבה:

- לכתיבה של מידע בינארי נשימוש ב-`wb`, קיצור של `write binary`
- לקריאה של מידע בינארי נשימוש ב-`rb`, קיצור של `read binary`

דוגמה לשימוש ב-`open` לפתיחת של קובץ טקסט:

הורידו את הקובץ https://data.cyber.org.il/python/dear_prudence.txt ושמרו אותו בתיקייה `networks\work\c`. כתעת הדינו את הפוקודה הבאה:

```
input_file = open(r'c:\networks\work\dear_prudence.txt', 'r')
```

מהו סוג האובייקט שמחזירה הפונקציה `open?` על מנת לגנות, נוכל להשתמש בפונקציה `type` המוכרת לנו. נסו לכתוב (`file`)`type` – מה קיבלתם?

קריאה מקובץ

הmethod `read` פועלת על אובייקטים מסוג `file`. בתור ברירת מחדל, המתוודה קוראת את כל הקובץ ושומרת אותו בתור מחוזת בתוך משתנה שהוגדר על ידי המשתנה. לדוגמה:

```
lyrics = input_file.read()
print(lyrics)
```

תוצאת פקודת הדפסה:

`"Dear Prudence" / The Beatles`

`Dear Prudence, won't you come out to play?
Dear Prudence, greet the brand new day
The sun is up, the sky is blue
It's beautiful and so are you
Dear Prudence, won't you come out to play?`

קל ופשוט. החסרון של שיטה זו היא שככל הקובץ נקרא בבת אחת לתוכה המשתנה שהגדרנו. אם הקובץ גדול מאוד – זה בעייתי, כיון שייגרם עומס גדול על הזיכרון וכתוכזהה מכך הקריאה מהקובץ תהיה איטית. לכן מומלץ להשתמש בשיטה קצת אחרת כדי לקרוא קובץ, שורה אחר שורה.

אפשרות אחרת היא להשתמש בmethod `readline`, שlify שמרמז השם שלו קוראת שורה אחר שורה. לדוגמה:

```
lyrics = input_file.readline()
```

מה יקרה אם הגענו לסוף הקובץ? במקרה זה הערך שנקלט מ-`readline` יהיה "", כולם מחוזת ריקה. דוגמה לקטע קצר שמדפיס את הקובץ שורה אחר שורה:

```
lyrics = None
while lyrics != '':
    lyrics = input_file.readline()
    print(lyrics, end="")
```

שימוש לב לכר שבסוף השורה האחרונה יש לנו `"=end"`, שאומר שלא להוסיף ירידת שורה בסוף הדפסה. הסיבה היא שבקובץ הטקסט ממילא יש ירידת שורה בסוף כל שורה, ואילו לא היינו מוסיפים את ההנחיה הזאת היה רוחן של שורה נוספת בין כל שתי שורות מודפסות.

עלקב המשימוש של הדפסת שורה אחר שורה, פיתון מאפשר לנו להשתמש בולולאת `for` רגילה עם איטרטור. בכל איטרציה מתבצעת למעשה קריאה של שורה אחת. כך, הקוד שלנו ניתן לכתיבה מקוצרת ונחמדה:

```
for line in input_file:
    print(line, end="")
```

שימוש לב לכר שלא היינו צריכים אפילו להשתמש ב-`read` או ב-`readline`. השיטה הזאת קצרה לכתיבה ומתאימה יותר לטיפול בקבצים גדולים.

בנוסף, שימוש לב לכר שבחרנו בשם `line` כדי לייצג כל שורה – כך ברור מה המשתנה הזה כולל בכל קריאה. קל יותר לקרוא קוד, במיוחד אם הוא ארוך, כשהכל שמות המשתנים הם בעלי משמעות. זהו אחד מעקרונות כתיבת הקוד הנכון אותן הזכירנו בפרק הקודם.

כתבה לקובץ

ראשית אנחנו צריכים לפתח את הקובץ לכתיבה (בנחיה שהוא סגור – מיד נראה איך סוגרים קובץ). כיוון שהקובץ קוד, במיוחד אם הוא ארוך, נרצה לפתוח אותו ב-`mode` של `append`. לאחר מכן משתמש בMETHOD write על מנת לכתוב מידע לתוך הקובץ:

```
input_file = open(r'c:\networks\work\dear_prudence.txt', 'a')
input_file.write('Dear Prudence open up your eyes\n')
```

סגירת קובץ

לאחר שימושים את הטיפול בקובץ מומלץ לסגור אותו. אמנם קובץ שפתחנו "סגר אוטומטית ברגע שתסתה" מרצית התוכנית שלנו, אבל אי סגירה של קובץ יכולה לגרום לתוכנית שלנו להתנתק בצורה לא צפופה וקשה מאוד לדיבוג. נמחיש על ידי דוגמה. התוכנית הבאה קוראת לפונקציה שפותחת קובץ לכתיבה בלבד, לסגור אותו, וכן להמחיש שהה איננו תכונות נכון, הפונקציה קרוייה `open_without_closing`. הפונקציה משנה את תוכן הקובץ. לאחר מכן נפתח אותו קובץ שוב, הפעם לקרואו. המצביעים לקובץ נקראים כאן `hfile` כיוצר של file.

```
FILENAME = r'c:\networks\work\dear_prudence.txt'
```

```
hfile1 = open(FILENAME, 'a')
hfile1.write('Dear Prudence open up your eyes\n')
hfile2 = open(FILENAME, 'r')
for line in hfile2:
    print(line, end="")
```

מה לדעתכם תהיה תוצאה ההדפסה? ובכן, באופן מפתיע ההדפסה לא כוללת את השורה שהוספנו לשיר! הסיבה היא שהשינויים נשמרים בקובץ רק לאחר סגירת הקובץ.

"Dear Prudence" / The Beatles

```
Dear Prudence, won't you come out to play?
Dear Prudence, greet the brand new day
The sun is up, the sky is blue
It's beautiful and so are you
Dear Prudence, won't you come out to play?
```

המסקנה היא שבדאי תמיד לסגור קבצים אחרים שסימנו להשתמש בהם. כדי לעשות זאת משתמשים במתודה `close`. פשוט כך:

```
hfile1 = open(FILENAME, 'a')
hfile1.write('Dear Prudence open up your eyes\n')
hfile1.close()
```

כעת, ההדפסה שנבצע מתוך פונקציית `main` תופיע כמו שצライר גם את השורה الأخيرة שהוספנו.

יש אפשרות נוספת יותר, שמאפשרת לנו לפתח קבצים בלי לדאוג לעשות להם close. פתיחת הקובץ עם הפקודה with דואגת לאsegirtat הקובץ אוטומטית. כיצד מבצעים זאת?

```
with open(FILENAME, 'r') as input_file:  
    for line in input_file:  
        print(line, end= "")
```

לאחר הפקודה with, נכתב open עם הפרמטרים הרגילים. לאחר מכן, נוסף as ואת שם המשתנה שיכיל את המצביע לקובץ. שורות הקוד הבאות מדפיסות את הקובץ, בדיק באותו אופן שבו הדפסנו אותו קודם. מתי יסגר הקובץ? הקובץ ישר פותח רק כל עוד אנחנו נמצאים בבלוק של with. ברגע שהבלוק יגמר, יסגר הקובץ אוטומטית.

לשימוש ב-with נוסף: נניח שהשתמשנו ב-close, אבל לפני שפיטון הגיע ל-close הוא נתקל בשגיאה והתוכנית הפסיקה לרווח עט שגיאה. כתוצאה לכך הקובץ שלנו נותר פתוח, למרות שבתוכנית הורינו לסגור אותו. ההורה with גורמת לכך שתבוצע סגירה של הקובץ לפני שהתוכנית מפסיקה לרווח ומחזירה שגיאה. כך אנחנו יכולים להיות בטוחים שהקובץ שלנו נסגר בכל מקרה. אין לבדוק whether אוirk היא מצליחה לסגור את הקובץ למרות שארעה שגיאה בדרך? על כן – כשנלמד exceptions.

עד כאן Learned כיצד להשתמש בקבצים – כיצד לקרוא מהם, לכתוב אליהם ולהוסיף להם מידע. כמו כן Learned את החשיבות שבסגירת הקובץ בתום השימוש בו. עצה, נעבור לחלק השני של פרק זה.

תרגיל – מכונת שכפול

צרו באמצעות סייר חולנות שני קבצי טקסט, האחד ריק והשני כולל טקסט כלשהו. כתבו סקריפט אשר מוגדרים בו שמות שני קבצים. הסקריפט יעתיק את הטקסט אל הקובץ הריק, כך שלאחר סיום הרכיצה הקבצים ייכלו את אותו טקסט.

קבלת פרמטרים לתוכנית

דמיינו שאתם משתמשים בסקריפט פיתון שבודק משזה על המחשב שלכם. לדוגמה, אם תיינית קבצים כלשהו מכילה קבצי פיתון, בעלי הסיומת py או csc. כאשר אתם מרכיבים את הסקריפט, אתם מתבקשים להזין את שם התקינה אותה אתם מעוניינים לבדוק. זה בסדר, אבל משזה פה מיותר: מילא כאמור את מרכיבים את הסקריפט אתם יודעים על איזו תיינית תרצו לפעול. למה צריך שהסקריפט ידפיס הודעה ויבקש ממכם להזין קלט? למה שלא תעבירו את שם התקינה לסקריפט כבר ברגע ההרצה? אם לסקריפט הדמיוני שלנו קוראים `udfindpy.py`, אז נרצה לכתוב ב-pcmd פקודה כגון:

```
|c:\>python findpy.py c:\python\homework
```

נקרא משמאל לימין: הסימן \c הוא שם התקינה בה אנחנו נמצאים כתע. הפקודה python אומרת להריץ את פיתון. השם sys.findpy.py הוא שם הקובץ שאנו רוצים להריץ – שמו לב שהוא למעשה פרמטר שנמסר לתוכנית homework...python לבסוף \homework\python\homework\findpy.py: c היא שם התקינה אותה אנחנו רוצים לבדוק.

בצורה זו, ניתן גם לבדוק בקלות הרבה יותר את הסкриיפט. אפשר להריץ אותו עם ערכים שונים ולבודק שקיבלו תוצאות נכונות.

בחינה טכנית אנחנו רוצים שהקובץ sys.findpy.py יהיה כתוב כך שהוא יוכל לקבל כפרמטר את שם התקינה שבה הוא צריך לבדוק האם קיימים קבצי פיתון. הנה נראה איך עושים זאת.

בutor התחלת נכיר בקצתה את המודול sys. מודול (או ספריה) הוא קובץ שמכיל פונקציות, ובפרקים הבאים נכיר מודולים נוספים. המודול sys מכיל פונקציות שמאפשרות פעולות מערכת שונות, כגון הגדרות של הדפסה למסך http://www.python-course.eu/sys_module.php

כדי לשלב את sys בקוד שלנו, צריך לגרום לסקרייפט שלנו להכיר אותו. לשם כך נשתמש בפקודת import:

```
import sys
print(sys.argv)
```

בעקבות השימוש ב-import, כל מבנה (פונקציה, משתנה או קבוע) שנכתב במודול sys נגישים לנו, כלומר אנחנו יכולים להשתמש בו. כדי להשתמש במבנה שמוגדר במודול, אנחנו צריכים קודם כל לכתוב את שם המודול, לאחר מכן נקודה ואז את שם המבנה. לדוגמה, כדי להשתמש בראשימה arg שמוגדרת ב-sys, צריך לרשום print sys.argv, בדיק כמה בדוגמה שבקוד. בהמשך הספר, בפרק על OOP, נלמד על שיטות נוספות להשתמש בפונקציות ובמבנהים נוספים שנמצאים בתחום מודולים.

arg מאפשרת לנו לקבל את הארגומנטים שהועברו לסקרייפט שלנו. באינדקס ה-0 של הרשימה נמצא שם הסкриיפט שאנו מרים, וביתר האיברים נמצאים הפרמטרים שהעבכנו לסקרייפט (אם העברנו כללה, אחרת הרשימה מכילה רק איבר אחד – שם הקובץ).

נשדרג מעט את הסкриיפט שלנו ונהפוך אותו לסקרייפט שמקבל כפרמטר שם, ומדפיס 'Hello' ואת השם.

```
import sys
NAME = 1
```

```
print("Hello {}".format(sys.argv[NAME]))
```

כאשר עובדים עם העבר PyCharm נוח להעביר פרמטר לסקריפט שלנו באמצעות cmd. PyCharm, ולא באמצעות ה-cmd. כיצד מעבירים את הפרמטר ל-PyCharm? זה זמן טוב לחזור לפרק אשר דן ב-PyCharm ולקרוא את החלק אודוט [העברית פרמטרים לסקריפט](#). קבענו כפרמטר את המחרוזת 'Shooki' וכותזאה מההרצה הודפס למסך:

Hello Shooki

תרגיל – Printer



כתבו סקריפט שמקבל כפרמטר שם שלקובץ ומדפיס את התוכן שלו למסך.



רגע אחד – מה יקרה אם ננסה להעביר לסקריפט שם שלקובץ שאין קיימ? נסו זאת. סביר שהסקריפט שלנו יקרוס עם הודעת שגיאה. זו בעיה, מכיוון שגם אם המשתמש שגה והזין שם קובץ שאין קיימ, לא נרצה שהקובד שלנו יקרוס. במקום זה, עדיף להחזיר למשתמש שגיאה שמסבירה לו שהקובץ אין קיימ. כדי לתקן את הבעיה, נכיר את המודול os. מודול שימושי נוסף זה, קישור של Operating System, מספק יכולות שונות של מערכת הפעלה. בתור דוגמה, נראה איך מציגים שמות של קבצים בתיקיה.

נתחילה כמובן מ-os import. ביצוע dir על os יראה לנו את כל הֆונקציות שששייכות ל-os, ביניהן נמצאת הfonקציה החביבה listdir. כתעת נשנה את הסקריפט שלנו בהתאם:

```
import sys
import os
PATH = 1

directory = sys.argv[PATH]
```

```
print(os.listdir(directory))
```

אנחנו טוענים לתוכה `directory` את הפרמטר שקיבל הסקריפט שלנו, ואז אנחנו מעבירים את `directory` ל-`os.listdir()` על מנת לקבל את רשימת הקבצים.

תרגיל – os.path



הבעיה בקוד שלנו, כמו בקוד שכתבתם כפתרון לתרגיל `printer`, היא שעדין נהיה בבעיה אם לסקריפט יועבר שם של תיקיה שאינה קיימת. עליו לפרט את הבעיה באמצעות בדיקה האם התיקיה קיימת ואם היא אינה קיימת – להדפיס שגיאה כגון "Directory not found", לפני שתתthem מבקשים את הקבצים שנמצאים בה. טיפ: `os.path` מודול שמקבלת `path` לתיקיה ובודקת אם היא קיימת. תוכלן לקבל את רשימת כל המתודות באמצעות `dir(os.path)` ותוכלו להשתמש ב-`help` על מנת לקרוא מה עשוה כל מתודה, עד שתמצאו את המתודה המתאימה.

תרגיל מסכם – Lazy Student



קיבלתם כשיורי בית קובץ עם תרגילים חשבונ. כל תרגיל הוא בפורמט הבא: מספר-רווח-פעולה-רווח-מספר. לדוגמה:

$46 + 19$

$15 * 3$

פעולה יכולה להיות אחת מארבע הפעולות: חיבור (+), חיסור (-), כפל (*) או חילוק (/) בלבד. כתבו סקריפט שמקבל קובץ תרגילים, כאשר כל תרגיל בשורה נפרדת, ושומר לקובץ נפרד את כל התרגילים כשמות פתורים. לדוגמה:

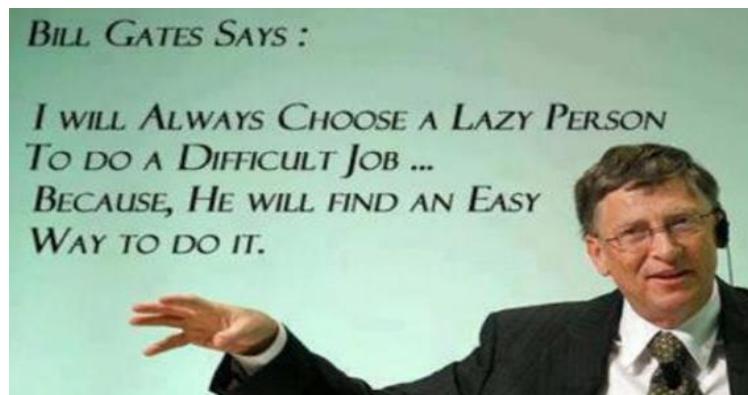
$46 + 19 = 65$

$15 * 3 = 45$

הסקריפט יקבל כפרמטרים שמות של שני קבצים – מקור ופתרון. לדוגמה:

```
python lazy_student.py homework.txt solutions.txt
```

הסקריפט יקרא את התרגילים מתוך file homework.txt וישמר את הפתרונות אל solutions.txt. הניחו מבון שיש שגיאות בפורמט של חלק מהתרגילים, או בשמות הקבצים. אסור לסקריפט שלכם לקרוא בשום אופן אם יש בעיה בתרגיל, כתבו במקום המתאים בקובץ הפתרונות הודעה שגיאה והמשיכו לתרגיל הבא.



סיכום

בפרק זה למדנו מספר דברים שימושיים למדעי. ראשית למדנו איך משתמשים בקבצים, לקריאת ולכנתיבת. הכרנו את הפונקציות המובנות לעבודה עם קבצים: open, read ו-write. ראיינו מה החשיבות של סגירת קובץ ולמדנו שיש דרך אוטומטית לסגור קבצים, באמצעות with.

שנית, עברנו אל העברת פרמטרים לסקרייפטים. במהלך חלק זה, הכרנו שני מודולים שימושיים – os ו-sys. אנחנו יכולים להשתמש במודולים אלו על מנת להכניס לסקרייפטים שלנו יכולות חדשות ומעניינות, כמו לדוגמה להכניס בצורה אוטומטית את שיעורי הבית שלנו בחשבון ☺

פרק 9 – Exceptions

בפרק זה נלמד להגן על הקוד שלנו מהתראקות בזמן ריצה באמצעות שימוש ב-exceptions. כמו שבמוכנות יש גם חגורת בטיחות וגם כריות אויר, כך גם במקרה שיש מספר אמצעים שנועד להגן עליו מקריסה, ו-exceptions הם אמצעי בטיחות נוספים.



פגשנו כבר ב-exceptions בפרקם הקודמים – בכל פעם שהקוד שלנו "עף", הודפס למסך טקסט שכלל exception כזה או אחר. הנה מספר דוגמאות מהפרקים שלמדנו:

```
6912
```

```
Traceback (most recent call last):
```

```
    print '1234' + 5678
```

```
TypeError: cannot concatenate 'str' and 'int' objects
```

```
['a', 'e', 'c']
```

```
Traceback (most recent call last):
```

```
    my_string[1] = 'e'
```

```
TypeError: 'str' object does not support item assignment
```

```
hi
```

```
Traceback (most recent call last):
```

```
    print word
```

```
NameError: global name 'word' is not defined
```

```

Traceback (most recent call last):
  word += ' you'
UnboundLocalError: local variable 'word' referenced before assignment

```

```

Traceback (most recent call last):
  assert list_is_nums([]) is False
AssertionError

```

הטעסט שמקורו באדום הוא שם ה-`exception`, והטעסט כתוב לפניו הוא תיאור מפורט של ה-`exception`. אפשר לראות שבכל פעם שתכתבו משהו שה-`interpreter` של פיתון לא הצליח להתמודד איתו, קיבלנו exception. אפשר גם לראות, שקיימים מסוגים שונים, כך שאם ניסינו לחבר מחרוזת עם מספר קיבלנו שגיאה שונה מכך שבמקרה שבו ניסינו לגשת אל משתנה שאינו קיים.

try, except

הפקודה `try` והפקודה התאומה שלה `except` הן פקודות מיוחדות במיןן, שמאפשרות לנו להריץ כל קוד שאחננו רצחים ולדואג למקרי הקצה בחלק הקוד אחר. כך הקוד שלנו הופך לנקי הרבה יותר. הרעיון של פקודת `try` הוא זהה: "נסה להריץ את קטע הקוד הבא. אם הכל טוב – יופי. אם יש בעיה, אל תתרסק, אלא פשוט תעבור לבצע את הקוד שנמצא אחרי הפקודה התאומה של `except`".



נראתה דוגמה שימושית. זוכרים שכתבנו סקריפט שמקבל מהמשתמש שם של תיקיה ומדפיס את כל הקבצים שנמצאים בה? חששנו מ מצב בו המשתמש מכניס שם של תיקיה שאינו קיימת ואז הסקריפט מתרסק. להלן הקוד הלא מוגן שכתבנו:

```
import sys
import os
PATH = 1

directory = sys.argv[PATH]
print(os.listdir(directory))
```

כעת נגן על הקוד מהתרסקות באמצעות try-except. יש בקוד זהה שתי פקודות "מסוכנות". הראשונה היא הפוקודה שקוראת את מה שנמצא ב-[PATH]sys.argv. אם המשתמש לא הכניס כלל פרמטרים, הקוד יקרוס עקב פניה לאינדקס שאינו קיים. הפוקודה השנייה היא כמובן הקריאה ל-os.listdir(). עם התיקיה שהמשתמש הכניס, שכאמר תקרוס אם התיקיה לא קיימת. נזכיר את שתיהן לתוך try:

```
try:
    directory = sys.argv[PATH]
    print(os.listdir(directory))
except:
    print("Error")
```

אם תהיה בעיה כלשהי, התוכנית תקוף ל-except ושם יודפס 'Error'.

הינו יכולים לבצע את אותו תהליך בלי try-except, רק בעזרת else-if'ים. הבדיקה הראשונה תהיה על אורך הרשימה argv והבדיקה השנייה על ערך החזרה של פונקציה שבודקת אם התיקיה קיימת. היתרון של try, הוא ש衲סכת מאייתנו כתיבת קוד – אפשר לבדוק את כל המקרים שעולים להוביל לקריסת קוד באמצעות פקודה יחידה.

מה הבעיה במצב כתיבה זו? אם ארצה שגיאה לא נדע לבדוק מה השגיאה – ישנן שתי אפשרויות. لكن בקרוב נראה איך אנחנו כתבים את ה-except באופן שגם יתן לנו מושג אודות השגיאה. אך קודם כל נחקרו עוד קצת את try-and-except.

לפניכם פונקציה שעולה להגעה ל-except. בידקו את עצמכם: באיזה מקרה הפונקציה תגיע לכך שמצא בתוך ?except

```
def do_something(thing):
    try:
```

```

    print("Hello " + thing)
except:
    print("Error")

```

אם נסמן, אם `thing` אינו מחרוזת, הפונקציה תדפיס 'Error'.

נסביר מעט את הפונקציה. מה לדעתכם יהיה הערך של `x` אם הקוד יכנס ל-`except`? במלחמות אחרות, אם לפונקציה יועבר פרמטר שאינו מחרוזת, מה יהיה הערך שיודפס?

```

def do_something(thing):
    x = 0
    try:
        x = 1
        print("Hello " + thing)
        x = 2
        print(x)
    except:
        print(x)

```

שנראה נפוצה היא לומר שיודפס 0. הטענה שגורסת שיודפס 0 היא "בתחילת התוכנית `x` מקבל את הערך 0. הריצה של `try` לא מתבצעת עקבות exception, לכן ערכו של `x` נותר 0 כאשר הוא מגיע ל-`except`". הטעות היא בכך שהיא מחלק מה-`try` דואקן מבוצע. כל מה שקדם לקיומה של exception יבוצע. במקרה זה, מתבצעת השורה בה מושמים ב-`-x` את הערך 1, ולאחר מכן ערכו כאשר הקוד קופץ ל-`except`. כמובן שהתוכנית אינה מגיעה לשורת הקוד בה `x` הינו 2.

השורה התחתונה היא, שאם אם התרחש exception, כל הפקודות שכבר התרחשו עדין תקפות. המעבד אינו " חוזר אחורה" ...

סוגים של Exceptions

נחזיר אל הקוד שהציגנו לפני זמן קצר:

```

try:
    directory = sys.argv[PATH]
    print(os.listdir(directory))

```

```
except:
    print("Error")
```

כפי שראינו בקטע הקוד האחרון, במקרה שמודפס 'Error', ישנו שתי אפשרויות לבעה:

- המשמש לא חזין כל פרמטר, ולכן הגישה ל-[PATH] sys.argv תהייה לאינדקס לא חוקי בראשימה.
- המשמש חזין תקין שגוייה, لكن os.listdir() יחזיר שגיאה.

הגינוי שנרצה לבדוק למשתמש מה הייתה הבעיה. הדרך לעשות זאת היא להחזיר את קוד השגיאה של ה-`exception`, דבר שמתבצע באופן הבא:

```
try:
    directory = sys.argv[PATH]
    print(os.listdir(directory))
except Exception as e:
    print("Error: {}".format(e))
```

צורת הכתיבה של `except` משמעותה "הכנס את הודעת השגיאה של ה-`exception` לתוך המשתנה `e`". כפי שראינו לכל `exception` יש הודעת שגיאיה ייחודית, המשתנה `e` יוכל אותה.

נסו להריץ את הסקריפט בלי שנותם לו פרמטרים כלל.

cut תנו לסקריפט פרמטר, אך של תקין שאינו קיימת באמת. לדוגמה `bla:c`. מה קיבלתם?

תשובה: כל אחד מהמקרים יופיע exception אחר. במקרה הראשון קיבלנו

`Error: list index out of range`

ואילו במקרה השני קיבלנו

`Error: [WinError 3] The system cannot find the path specified: 'c:\bla'`

לכל אחת מהשגיאות ישנו סוג שונה של exception. כדי לגלוות מהו, שימוש breakpoint בתוך ה-`except`.

במקרה הראשון, מתרברר ש-`e` הוא מסוג IndexError, כפי שאפשר לראות בחולון הדיבוג.

במקרה השני, מתברר ש-e הוא מסוג WindowsError.

בדוגמה הבאה אפשר לראות איך אנחנו תופסים exceptions שונים ומטופלים בכל אחד מהם בנפרד. אם היה שגיאה שלא מסוג IndexError או WindowsError, היא תיתפס על ידי ה-exception האחר. דבר זה מאפשר לנו גם לפעול בצורה שונה עבור כל מקרה של שגיאה (במקרה זה, לחת מידע רלוונטי למשתמש) וגם לטפל במקרה שגיאה שלא חשבנו עליהם מראש או שאין לנו קוד מיוחד שנרצה להריץ עבורם:

```
try:  
    directory = sys.argv[PATH]  
    print(os.listdir(directory))  
except IndexError:  
    print("Missing script parameter")  
except WindowsError:  
    print("No such directory")  
except Exception as e:  
    print("Error: {}".format(e))
```

מהו e? היא class של פיתון (마וחר יותר נלמד תכונות מונחה עצמים), אך הוא אובייקט שמכיל את כל השדות של Exception, כולל הודעה השגיאה.

finally

הפקודה `finally` נמצאת לעיתים קרובות בשימוש יחד עם ה指挥符 `try-except`. פקודה זו שימושית כאשר נרצה שקוד כלשהו ירוץ בכל מקרה, בין שאירוע exception ובין שלא. מן הסתם, לא ניתן להעתיק את אותן שורות קוד הן ל-`try` והן ל-`except`. כאן מוגיעה `finally` לעזרתנו. כל מה שנמצא בבלוק של `finally` ירוץ בכל מקרה.

נסקרו דוגמה. חישבו מה יופיע הקוד הבא?

```
def do_something(thing):
    try:
        print("Hello " + thing)
    except Exception as e:
        print(e)
    finally:
        print("Final")
```

```
do_something("Shooki")
do_something(123)
```

התשובה בעמוד הבא.

הקריאה הראשונה ל-do_something תרוץ ללא בעיות מיוחדות. אך יורץ הקוד שנמצא ב-try וב-finally. הקריאה השנייה תגרום מיד לזריקת exception, ולאחר מעבר ל-except exception, ולבסוף מכך לביצוע finally. תוצאה הריצה תהיה ההדפסה הבאה:

```
Hello Shooki
Final
can only concatenate str (not "int") to str
Final
```

נסקרו דוגמה נוספת נוספת, הפעם עם פונקציה שמחזירה ערך. חישבו, מה ידפיס הקוד הבא?

```
def do_something(thing):
    try:
        print("Hello " + thing)
        return "OK"
    except Exception as e:
        print(e)
        return "Error"
    finally:
        print("Final")

print("'Shooki' returns: {}".format(do_something("Shooki")))
print("123 returns: {}".format(do_something(123)))
```

תוצאת הרצה היא:

```
Hello Shooki
Final
'Shooki' returns: OK
can only concatenate str (not "int") to str
Final
123 returns: Error
```

בקרייה הראשונה לפונקציה, ה-try מסתיים בהחזרת ערך. CAN מתרחש שהוא מעניין – ה-*interpreter* של פיתון מזהה שאנו עומדים לצאת מהפונקציה ולכן הוא בודק אם ישנו קוד ב-*finally* שעליו לבצע לפני כן. לאחר ביצוע פקודת ההדפסה של 'Final' מתבצעת חזרה אל בлок ה-try ומשם מתקבל ערך החזרה 'OK'.

בקרייה השנייה לפונקציה מתבצע תהליך דומה, קופיצה ל-*finally* וחזרה, אלא שהוא מתבצע מה-.*except*.

cut נבחן דוגמה נוספת, שמחישה את פעולה *finally* – שוב, חישבו מה מבצע הקוד הבא:

```
def do_something(thing):
    try:
        print("Hello " + thing)
        return "OK"
    except Exception as e:
        return "Error"
    finally:
        return("Final")

print("123 returns: {}".format(do_something(123)))
```

הfonקציה תגיע אל ה-except, שם כפי שראינו בדוגמה הקודמת interpreter-finlaly של פיתון יקפוץ אל עלי. ההבדל מהדוגמה הקודמת, הוא שבמקרה זה מכילה הוראת חזרה וכן בכר תסתיים ריצת הפונקציה. תוצאת הדפסה:

```
123 returns: Final
```

with

כזכור, כאשר למדנו על פתיחת קובץ באמצעות open-with, אמרנו שגם אם מתרחשת שגיאה כלשה' במהלך הריצה עדין הקובץ יסגר. כיצד זה מתרחש?

למעשה, ניתן לתרגם את with לפקודות הקשורות try-and-finally. לדוגמה, הקוד הבא:

```
with open('dear_prudence.txt', 'r') as input_file:
    do_something_that_will_raise_exception()
```

שקלול לקוד הבא:

```
input_file = open('dear_prudence.txt', 'r')
try:
    do_something_that_will_raise_exception()
finally:
    input_file.close()
```

כפי שאנו רואים, with כולל בתוכו finally, אשר דואג לסגירת הקובץ בכל מקרה – גם אם ארצה שגיאת מקום כלשהו בבלוק שיירן.

תרגיל מסכם – lazy student 2



כיתבו מחדש את הפתרון לתרגיל lazy student, אך תוך שימוש ב-try-except במקום הנדרשים. הקפידו על כך שבמידה ואירועה שגיאת כלשהו (קובץ לא נפתח, חלוקה באפס) התוכנית מדפיס הודעה שגיאת מדוייקת ולא רק שארעה שגיאת כלשה'.

קיבלתם כשיורי בית קובץ עם תרגילי חשבון. כל תרגיל הוא בפורמט הבא: מספר-רווח-פעולה-רווח-מספר. לדוגמה:

$46 + 19$

$15 * 3$

פעולה יכולה להיות אחת מארבע הפעולות: חיבור (+), חיסור (-), כפל (*) או חילוק (/) בלבד. כתבו סקሪיפט שמקבל קובץ תרגילים, כאשר כל תרגיל בשורה נפרדת, ושומר לקובץ נפרד את כל התרגילים כשהם פטורים. לדוגמה:

$46 + 19 = 65$

$15 * 3 = 45$

הסקרייפט יקבל כפרמטרים שמות של שני קבצים – מקור ופתרון. לדוגמה:

```
python lazy_student.py homework.txt solutions.txt
```

הסקרייפט יקרא את התרגילים מתוך `homework.txt` וישמר את הפתרונות אל `solutions.txt`. הניחו כМОון שיש שגיאות בפורמט של חלק מהתרגילים, או בשמות הקבצים. אסור לסקרייפט שלכם לקרוא בשום אופן! אם יש בעיה בתרגיל, כתבו במקום המתאים בקובץ הפתרונות הודעה שגיאה והמשיכו לתרגיל הבא.

תרגיל מסכם פיתון בסיסי – LogPuzzle



(קודית: `google class`, התאמה לגבאים: דנהabenheim, אורילוי)

על מנת לסכם את הידע שצברנו עד כה, נבצע תרגיל שיכלול רבים מהדברים שלמדנו ותוך כדי נלמד גם מספר דברים חדשים.

בתרגיל זה תצטרכו להרכיב תמונה מחלקים אשר פוזרו באינטרנט. בטור התחלה, נראה כיצד אפשר להוריד כל קובץ שנמצא באינטרנט באמצעות פיתון, תוך שימוש במודול `urllib`.

ראשית – מהו URL? אלו ראשי תיבות של Universal Resource Locator. לכל משאב באינטרנט יש URL מסויל. לשם המחשבה, יכול להיות שרת האינטרנט שכותבו `www.blabla.com`. בתוך השרת זהה שמור קובץ `pdf` בשם `bla1.pdf` ועמוד `html` בשם `blapage.html`. ניתן יהיה להגיע לכל משאב לפי ה-URL'ים הבאים:

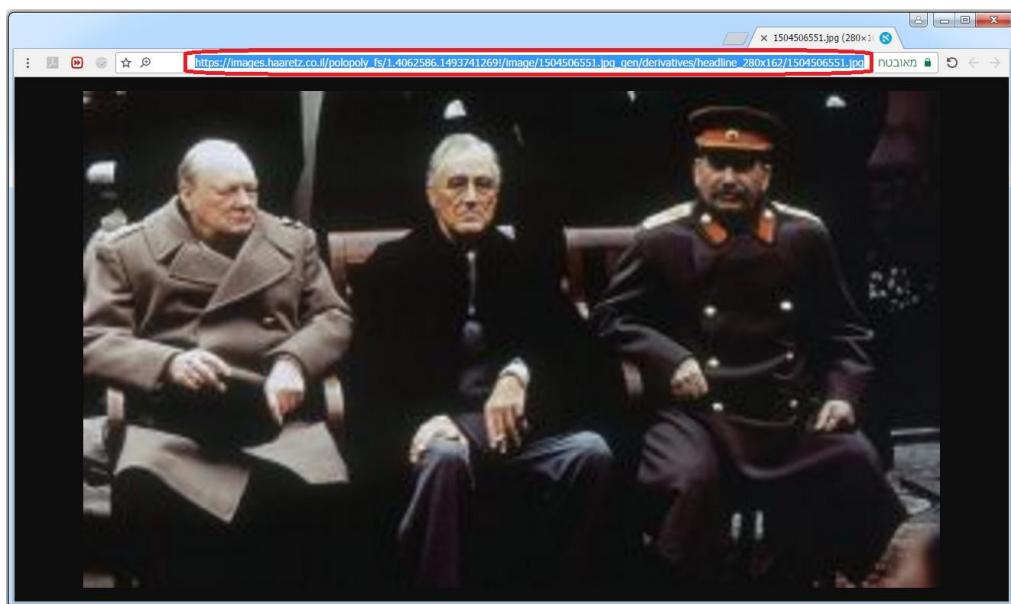
<http://www.blabla.com/blapage.html>

<http://www.blabla.com/bla1.pdf>

כמובן שאלו לינקים שאינם קיימים, הם מובאים רק לצורך ההמחשה של הרעיון.

עת אנו בשלים להבין את השימוש ב-`urllib.request.urlopen`. מודול זה כולל פונקציה שנוצרה – `urlopen`. הפונקציה מקבלת כפרמטר את ה-URL של קובץ שרוצים להוריד ולשמור במחשב שלהם. נתרgal את השימוש בו.

גילשו לאתר אינטרנט כלשהו, בחרו תמונה ולחצו על הלחצן הימני, אז על "פתח תמונה בכרטיסיה חדשה". לאחר שהתמונה תיפתח, עיברו אל הכרטיסיה החדשה שנפתחה והעתיקו את ה-URL של התמונה משדה הכתובת של הדף.



פיתחו סקריפט פ'יתון וכיתו:

```
import urllib.request
```

```
URL =
```

לאחר סימן ה- "=" בצעו "הדבק" לערך של ה-URL של התמונה שבחרתם. הוסיףו שם של קובץ אליו יש לשמר את הקובץ שאתם מורים. כמובן שצריך לדאוג לכך שם הקובץ יהיה בעל סימנת זהה לקובץ שאתם מורים. לדוגמה, אם אנחנו מורים קובץ עם סימנת jpg אז שם הקובץ אליו אנחנו שומרים צריך גם הוא להסתיים ב-.jpg. סימנת הקובץ שאנו מרים נמצאת תמיד בתווים האחרונים (הימניים) של ה-URL.

לדוגמה:

```
import urllib.request

URL = 'https://img.haarets.co.il/img/1.4062586/1504506551.jpg'
FILENAME = r'c:\networks\work\image1.jpg'
```

כעת תוכלן להוריד את התמונה מהשרת באמצעות הפקודה הבאה:

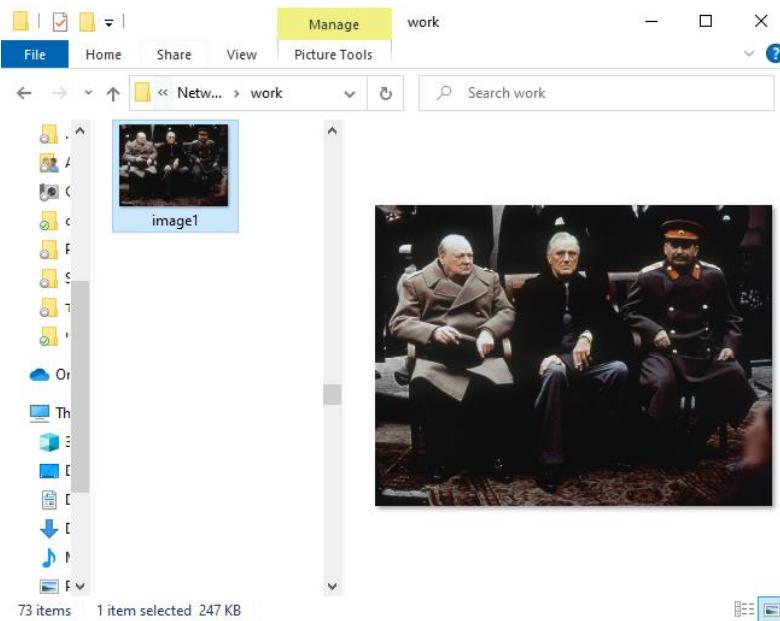
```
with urllib.request.urlopen(URL) as response:
    image = response.read()
```

כל שנוטר לכם לעשות הוא לשמר את התמונה לתיק קובץ, דבר שלמדנו לעשות בפרק הקודם. להלן הקוד המלא:

```
import urllib.request

URL = 'https://img.haarets.co.il/img/1.4062586/1504506551.jpg'
FILENAME = r'c:\networks\work\image1.jpg'
with urllib.request.urlopen(URL) as response:
    image = response.read()
    with open(FILENAME, 'wb') as output_file:
        output_file.write(image)
```

והתוצאה:



כעת משאנו יודעים איך להוריד תמונות מהאינטרנט, יש פרט נוסף שנוצר ללמידה כדי לפתור את הפАЗל – חיבור תמונות לתמונה אחת. לשם כך נשתמש בקובץ html. כאשר רושמים בתוך הקובץ את שמות התמונות, הפעלת הקובץ מציגה אותן זו לצד זו.

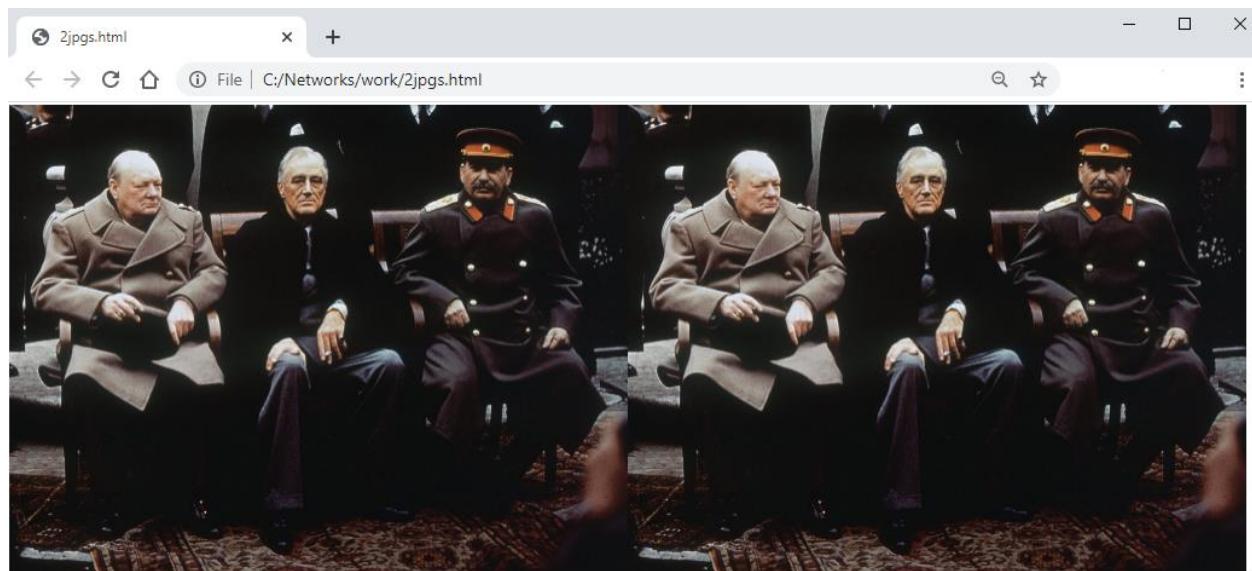
כל מה שאנו צריכים לדעת כדי לכתוב את קובץ html שנדרש לצורך פazaל, הוא כיצד להשתמש בתבנית הבאה:

```

C:\Networks\work\2jpgs.html - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
2jgs.html X
<verbatim>
1 <html>
2   <body>
3     
4   </body>
5 </html>
6

```

הקלקה על הקובץ שייצרנו תפתח דףן ובו תציג התמונה שהורדנו – פעמיים.



כמובן, כל מה שאנו צריכים לעשות זה להחליף את שמות הקבצים שבתבנית בשמות של התמונות, חלק הפАЗל, וכך נוכל להציג את התמונות זו לצד זו.

לאחר הקדמה קצרה זו, ניגש לתרגיל עצמו. בליין הבא נמצאים שני קבצי לוג. בכל קובץ לוג ישנו טקסט, שמתחביבים בו אweis של תמונות זו-זו שצרי להוריד. לאחר ההורדה של התמונות ושמירתן לדיסק במקום שתבחרו, עלייכם לMININ אותו ואז להציג אותו זו לצד זו בהתאם לסדר המין שנקבע בתרגיל.

<http://data.cyber.org.il/python/logpuzzle.zip>

כיצד לזרות את התמונות שצריך להוריד מקובצי הלוג? ה-URL של כל תמונה נמצא בטקסט שמופיע החל מס'ום פקודת ה-'GET' ועד סוף ה-'jpg'. כמו כן, ה-URLים כוללים את השם שרת גבהים.

מה ה-URL המלא של הקבצים שאתם מורידים?

עליכם לחת את שם קובץ ה-'jpg' ולהוסיף לו את שם קובץ הלוג ואת הכתובת של שרת גבהים. לדוגמה, הקובץ python/a-baaa.jpg שנמצא בקובץ הלוג logo_cyber/logo, ה-URL המלא הוא:

<http://data.cyber.org.il/python/logpuzzle/a-baaa.jpg>

air מtabצע המיון?

- עברו הקובץ logo, המיון הוא לפני סדר האלף-בית של הקבצים
- עברו הקובץ message, המיון הוא לפני סדר האלף-בית של המילה **השניה** בשמות הקבצים. לדוגמה הקובץ dpg.jpg-a-aaaaa-zzzz-cccc-cccc-cccc-z, כיוון שבמילון cccc קודם ל-zzzz.

בלינק ישו קובץ בשם uy.logpuzzle, שמהווה את קובץ השلد לפתרון התרגילים. הוא כולל את הפונקציות שנדרשות לביצוע המשימה, ועליכם להשלים את הקוד החסר בפונקציות.

cut תורכם – פיתרו את הפלז. בהצלחה!

פרק 10 – תכונות מונחה עצמים – OOP

בפרק זה נלמד כיצד מבצעים תכונות מונחה עצמים בפייתון. יתכן ששמעתם על המושג "תכונות מונחה עצמים", אוanganilit Object Oriented Programming, אך הפרק אינו מניח ידע קודם בתכונות מונחה עצמים, רק שליטה בנושאים שנלמדו בפרקם הקודמים. הנושא הוא רחב, ולכן הפרק יתחלק ל-5 חלקים:

- א. מבוא – מדוע בכלל צריך תכונות מונחה עצמים, `class`, `object`,
- ב. כתיבת `class` בסיסי,
- ג. כתיבת `class` משופר
- ד. ירושה – inheritance
- ה. פולימורפיזם



מבוא – למה OOP?

לפני שאנחנו לומדים נושא חדש, צפוי שנשאל את עצמו מה נרוויח מזה. הלא עד כה הצלחנו לפתור את כל המשימות שקיבלנו. התשובה היא – ידע בתכונות מונחה עצמים יאפשר לנו לכתוב קוד הרבה יותר מהר מאשר אלמלא היה לנו את הידע הזה. נמחיש על ידי דוגמא.

אנחנו רוצים לכתב תוכנית שתשתמש בית ספר. יש צורך לשמר את שמות כל התלמידים ואת הציוןיהם שלהם במקצועות השונים. אנחנו יכולים לעשות זאת כך – לכל תלמיד וכל מקצוע של תלמיד נקזה משתנה:

```
talmid1_name = 'Shimshon Gani'
```

```
talmid1_english = 90
```

```
talmid1_math = 95
```

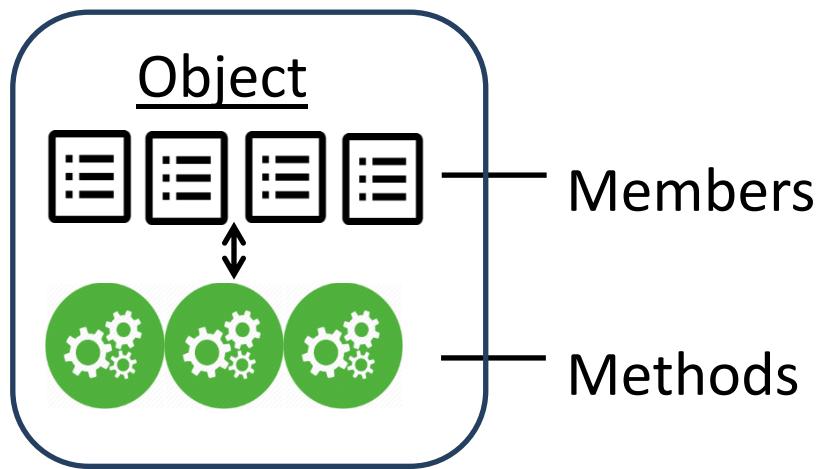
```
talmid1_geography = 85
```

אם יש לנו 200 תלמידים ו-10 מקצועות לכל תלמיד, בלי OOP נצטרך להזכיר שם של משתנה לכל תלמיד וכל מקצוע של כל תלמיד. אנחנו בדרכם הבוטוכה להגדיר אלפי משתנים... גם רשימה של ציונים עבור כל תלמיד אינה רעיון מוצלח, כיוון שנצטרך לעבוד עם אינדקסים כדי להגיע אל המקצועות השונים. באמצעות OOP נוכל לצמצם את כמות המשתנים שלנו. נלמד לבנות `class` של תלמיד, ובאמצעות 200 אובייקטים מסווג תלמיד – מיד נלמד גם מהו אובייקט – נחזיק את כל המידע על התלמידים. צמצמנו את כמות המשתנים שלנו מאלפים למאות. נחמד? ניתן לטעון בצדק שאנו הצמצום המהיר למאות הוא נפלא, אבל גם לבצע פעולות על מאות אובייקטים יגרום לנו לתוכנת קוד די ארוך. ובכן, נראה שאפשר לעבור על כל האובייקטים עם לולאת `for` וכן לkürר עוד יותר את הקוד שלנו.

בהמשך נראה רעיון נוסף שמאפשר לנו להתבסס על קוד של אחרים על מנת לkürר את הקוד שלנו – ירושה. נשאיר זאת לחלק המתקדם יותר של לימוד ה-OOP.

אובייקט – object

از מהו ה-'O' הראשון שב-OOP? אובייקט הוא ישות תוכנה שמכילה מידע ופונקציות. המידע נקרא `members` והfonktsiyot נקראות `methods`, או מתודות.



אובייקט מורכב ממידע ומMETHODOT שפועלות על המידע

נתקלנו כבר בMETHODOT והסבירנו שהזאת סוג מיוחד של פונקציות. אם כך, עכשו אנחנו יכולים לחזור את ההסבר – מתודה היא פונקציה, אבל לא סתם פונקציה אלא פונקציה שמודגרת כחלק מאובייקט ופועלת על האובייקט. כמובן, רק האובייקט מכיר אותה ורק הוא יודע איך לعباد אותה. לדוגמה, ראיינו שקובץ הוא אובייקט. לאובייקט מסווג file יש METHODOT שקוראים לה `read()`. אנחנו לא קוראים לה בצורה זו:

```
read(filename)
```

הרי `read` לא יודעת לקבל שם דבר שהוא לא `file`. מעבר לכך, בקריאה כזו – פיתון חושב שהוא מתיחסים לפונקציה כללית בשם `read`, ולא METHODOT אובייקט ספציפי. במקומות השימוש לעיל, אנחנו משתמשים בה כך, עם נקודה:

```
filename.read()
```

בצורה זו אנחנו מוחים את ה-`interpreter` של פיתון – "גש אל האובייקט שנקרא `filename`. תמצא שיש לו METHODOT בשם `read`. הפעל אותה על `filename`".

לטיכום, אובייקט מכיל גם מידע וגם METHODOT. ראיינו איך ניתן לMETHODOT של אובייקט, מיד נראה איך "מייצרים" אובייקט שככל גם METHODOT וגם מידע.

מחלקה – class

מחלקה, או `class`, הוא קטע קוד שגדיר את כל ה-members והMETHODOT של אובייקט ואשר משתפים לו וליתר האובייקטים של אותה מחלקה. נמחייב זאת. ניקח את כוכב הלכת שבתאי. זהו כוכב לכט אחד מתוך מספר כוכבי לכט במערכת השמש שלנו. למרות שכל אחד מהם הוא שונה, כוללם יש מסה, רדיוס, מרחק מהשמש, זמן הקפה של השמש וכו'. אפשר להגיד מחלקה של כוכבי לכט, לדוגמה בשם `planet`, אשר תוכל את כל המאפיינים המשותפים לכל כוכבי הלכת.



בכל פעם שנרצה להגיד כוכב לכט, פשוט נשתמש במחלקה `planet` – היא מכילה כבר את התבנית לשםית כל המידע. אפשר לדמיין ש-`planet` היא דף מידע שמכיל את כל השודות שצריך בשביל להגיד כוכב לכט.

Planet

שם הכוכב:

מסה:

מרחק מהשמש:

זמן הקפה:

טמפרטורה:

از מה ההבדל בין אובייקט למחלקה? מחלקה מתארת את המאפיינים של כל האובייקטים ששווים למחלקה. כשהתוכנית רצה, בכל פעם שנגידר אובייקט מסוים, יוקצת זיכרון במחשב בהתאם לכמות המידע שצריך כדי לשמר את כל המאפיינים של המחלקה. אפשר לומר שמחלקה היא כמו תבנית של עוגיות: התבנית אינה עוגיה ואי אפשר לאכול התבנית, אבל באמצעות התבנית אפשר ליצור עוגיות. בכל פעם שנשתמש בתבנית, חתיכה של בץ תקבל את הצורה של התבנית זו. כמוות הבץ שנקצת לעוגיה נקבעת על ידי התבנית, דומה לכך שכמות הזיכרון שמקצת לאובייקט נקבעת על ידי המחלקה.



כעת, כאשר הבנו את הקשר בין מחלקה לאובייקט, נוכל להגדיר מושג חדש שיתאר במליה אחת את הקשר ביןיהם – `instance`. כל אובייקט הוא `instance` של המחלקה ממנו נוצר. לדוגמה, עוגיה היא `instance` של תבנית העוגיות ואילו שבתאי הוא `instance` של המחלקה `planet`.

בקרוב, נראה כיצד כתבים `class`, וכייז יוצרים אובייקט שלו – ככלומר `instance` של המחלקה הזו.

כתיבת `class` בסיסי

נדמיין שאנו מפעילים מגדל פיקוח, ששולט בתנועת המטוסים סביב שדה תעופה. תפקידנו למנוע התנגשויות מטוסים באוויר. בכל רגע נתון, יש באוויר מספר מטוסים, והבעיה היא שהטייסים השתגעו וטסם בכיוונים אקראיים. תפקידנו הוא לעקוב אחרי תנועת המטוסים באוויר.



כדי לתוכנת את הפתרון, ראשית אנחנו צריכים ליצור את המטוסים שלנו. נתחיל מיצירת `class` שיהיה תבנית ליצירת מטוסים:

```
class Plane:
```

```
  def __init__(self):
    self.x = 0
    self.y = 0
```

נפרק את מה שכתוב לחלקים.

שם ה-class הוא `Plane`. שמו לב לכך שהשם מתייחס לאות גודלה – זהה הקונבנצייה לקביעת שמות של מחלקות. מיד נסביר מהם ה-`__init__` וה-`self`. לאחר מכן מוגדרים שני members (זכירים שאמרנו שאובייקט מורכב ממתודות ומ-members?)`members`. הראשון הוא `x` והשני הוא `y`. כל אחד מהם מקבל ערך התחלתי – 0.

__init__

זהו המתודה הראשונה שנכיר. השם `init` נגזר מ-`initializer`, או אתחול. אפשר לקרוא לה גם `constructor` או בנאוי. בתוך `__init__` נהוג לשים את האתחול של כל ה-members של המחלקה, כפי שראינו בדוגמה. זהה מתודה מיוחדת בכך שהיא רצתה באופן אוטומטי בכל פעם שנוצר `instance` של `Plane` בזיכרון המחשב.

כפי שאנו רואים בדוגמה, המתודה מקבלת פרמטר שנקרא `self`. כדי להבין מהו, נשאל את עצמנו שאלה: איך פיתון ידוע על איזה אובייקט להפעיל את המתודה? במילאים אחרות, אנחנו יכולים ליצור כמה אובייקטים ששיכים לאוטו `class`, לדוגמה כמו אותה תבנית עוגיות שימושת ליצירת עוגיות רבות. אם אנחנו רוצים לפעיל מתודה על עוגיה מסוימת, איך פיתון ידוע על איזו עוגיה הוא צריך לפעול?

התשובה היא `self`. אנחנו מעבירים למתודה מצביע לאובייקט שעליו היא אמורה לפעול. אנחנו אומרים למתודה "תפעיל על העוגיה זו" בזמן שהאצבע שלנו מצביעה על עוגיה מסוימת. אי לכך, נעביר את `self` כפרמטר לכל מתודה של המחלקה שצפוייה לרווח על האובייקט הזה.



הוסף מתודות

לאחר שכתבנו את המתודה הראשונה, `__init__`, הגיע הזמן להוסיף מתודות שעשוות פעולה שאינן אתחול. כיוון שמדובר במטוסים, נרצה לאפשר לעדכן מיקום ולקבל את המיקום:

```
import random

class Plane:

    def __init__(self):
        self.x = 0
        self.y = 0

    def update_position(self):
        self.x += random.randint(-1, 1)
        self.y += random.randint(-1, 1)

    def get_position(self):
        return self.x, self.y
```

המתודה `update_position` פועלת על ערכי ה-`x` וה-`y` המציגים את מיקום המטוס ומעדכנת אותם רנדומלית (כמו שאמורנו, הטיסים השתגעו). המתודה `get_position` פשוט מחזירה את מיקום המטוס.שוב, נשים לב לכך שככל מתודה צריכה לקבל את `self` בתור פרמטר. כמו כן, נעשה שימוש במודול `random` וכן נדרש לבצע `import random` בתחילת הקובץ.

Members

כפי שלמדנו זה עתה, ברגע שנוצר אובייקט מסווג `Plane` יוצרו לנו `x` ו-`y` שישיכים לאובייקט שלנו. על כן הם נקראים `members` של האובייקט. כדי להבין מדוע יש ל-`x` ו-`y` שם מיוחד, ולא סתם "משתנים", נציג משתנה ונראה את ההבדל בין `yx`:

```
class Plane:

    def __init__(self):
        self.x = 0
        self.y = 0

    def count_down(self):
```

```
for i in range(10, 0, -1):
    print(i)
```

הmethodה `count_down` סופרת מ-10 ומטה, ומוגדר בה זו המשתנה `i` "ח'י". רק בתוך לולאת ה-`for` של המתודה. ברגע שהלולאה מסתיימת, זו אימת קיימ יותר ולא ניתן לגשת אליו. זאת לעומת `x`, שקיימים כל עוד האובייקט קיימ. אך `x` הם `members`.

יצירת אובייקט

לאחר שהגדכנו את המחלקה, הגיע הזמן להשתמש בתבנית שהגדכנו על מנת ליצור אובייקטים. הקוד הבא יוצר אובייקט בשם `plane1` ומשתמש באובייקט על מנת לקבל את מיקום המטוס:

```
def main():
    plane1 = Plane()
    xpos, ypos = plane1.get_position()
    print(xpos, ypos)

if __name__ == "__main__":
    main()
```

שים לב לכך שאנו מופיעים לשים את הקוד שלנו בתוך פונקציית `main`, ושורת הקוד למטה דואגת לקרוא לה. הסיבה לכך שאנו מופיעים בכך כעת, ולפניהם לא, תתרբור בפרק זה כאשר נלמד אוזות הפקודה `import`.

חישבו – מה יהיה ערכיהם של `xpos`, `ypos` ?

יצרנו אובייקט בשם `plane1`, שהוא `instance` של `Plane`. ברגע שיצרנו אותו, אוטומטית מוצרכת פונקציית `__init__`, אשר יוצרת את `x` ו-`plane1.y` ומתחילה את ערכיהם ל-0. המethodה `get_position` פועלת על האובייקט `plane1` ומהזירה את ערכי ה-`x` וה-`y` שלו, כמובן 0, 0. נסו זאת בעצמכם – צרו אובייקט ובידקו שקיבלתם את ערכי ההתחלה שקבעתם.

נעשה בתוכנית שלנו שינוי קטן. במקום `sokx` נכתב `x` ובמקום `soky` נכתב `y`. האם התוכנית תעבור כעת, או שתתקבל שגיאה?

```
def main():
    plane1 = Plane()
    x, y = plane1.get_position()
    print(x, y)
```

התשובה היא שהתוכנית תעבור ללא כל בעיה. חשוב להבין ש-*y*, *x* שהגדכנו בפונקציה *main* אינם *y*, *x* שישיכים ל-*plane1*. בambilם אחרות, *x* אינו *x1.y* *plane1* ו-*y* אינו *y1.x*. אין להם את אותו (*id*). גם אם נשנה את *x*, ערכו של *x1.plane* לא ישתנה.

תרגיל



כתבו *class* של חיה האהובה עליכם (לדוגמה Cat, Dog וכו').

- הוסיפו מתודת *__init__* שתכלול את שם החיה (לדוגמה Kermit) ואת גיל החיה

- הוסיפו מתודת *birthday*, שתעלה את גיל החיה ב-1

- הוסיפו מתודת *get_age* שתחזיר את גיל החיה

שיםו לב שבשלב זהה כל החיות שתיצרו באמצעות התבנית של המחלקה יקבלו את אותו שם אשר מופיע ב-*__init__*. בקרוב נלמד כיצד נתונים לכל חיה שם שונה בשלב האתחול.



כתיבת class משופר

בחלק זה נלמד לשפר את ה-class שכתבנו בחלק הקודם. לשם כך נלמד לבצע כמה דברים:

- ליצור members "מוסתרים"

- להפוך את ה-class לקובץ שנייתן לייבא על ידי import

- לקבוע ערכים התחלתיים לאובייקט חדש

- ליצור פקודת הדפסה מיוחדת ל-class שלנו

- ליצור מתודות accessor ו-mutator

יצירת members "מוסתרים"

קטע הקוד הבא דורס את נתוני המיקום של המטוס:

```
plane1 = Plane()
plane1.x = 10
plane1.y = 10
x, y = plane1.get_position()
print(x, y)
```

הmethodה get_position תחזיר את הערכים 10, 10. בעולם האמיתי, תוצאה של קוד כזה עלולה להיות התנגשות בין מטוסים. علينا למצוא דרך "להסתיר" את נתונים המיקום של המטוס, כך שהתוכנה שעושה בהם שימוש לא תשנה אותם בטעות.



בפייתון, הסטרה של members מותבצעת באמצעות תחילה `__` (קו תחתית כפוף). נסתר את המשתנים של `:Plane`

```
class Plane:

    def __init__(self):
        self.__x = 0
        self.__y = 0

    def update_position(self):
        self.__x += random.randint(-1, 1)
        self.__y += random.randint(-1, 1)

    def get_position(self):
        return self.__x, self.__y
```

הקו התחתית ההפוך גורם לכך שרק מתודות של `Plane` מכירות את ה-members הללו. מי שמשתמש ב-class `Plane` יוכל לראות שהם קיימים. שימו לב להבדל: לפני ההסתירה, כתיבת `"plane1."` העלה אפשרות שונות, בין היתר `x`, `y`.

```
plane1 = Plane()
plane1.
```

m	<code>get_position(self)</code>	<code>Plane</code>
f	<code>x</code>	<code>Plane</code>
m	<code>update_position(self)</code>	<code>Plane</code>
f	<code>y</code>	<code>Plane</code>

לאחר ההסתירה, אפשר באמצעות המושלים האוטומטי לראות רק את המתודות הבאות:

```
plane1 = Plane()
plane1.|
```

m	<code>get_position(self)</code>	<code>Plane</code>
m	<code>update_position(self)</code>	<code>Plane</code>

האם זה אומר שכבר אין אפשרות לגשת אל members מסוימים? לא. פיתון מאפשרת לנו גם אפשרות זו. בשפות עליות ישנו מושג שנקרא "private", שימושו משתנים או מתודות שאפשר לגשת אליהן מחוץ ל-class. בפייתון אין private, יש רק הסתרה. אם מתעקשים, אפשר לגשת אל members אפילו אם הם מסוימים.

זכור פונקציית dir מוחזירה את כל המתודות וה-members לשبيיכים לאובייקט מסוים. אם כך, נעשה (plane1)dir ונקבל רשימה ארוכה שמהחילה כה:

```
_Plane_x
_Plane_y
_class_
_delattr_
_dict_
_dir_
```

שים לב לשני האיברים הראשונים ברשימה, אשר מסתיימים ב-"x" ו- "y". כן, נראה דומה למה שאנו מתחפשים. ננסה לפנות אליהם:

```
def main():
    plane1 = Plane()
    plane1._Plane_x = 5
    plane1._Plane_y = 6
    print(plane1.get_position())
```

הקוד רץ ללא שגיאות וכאשר מדפיס את הערך שמחזירה get_position מקבל (5,6), מה שמעיד על כך שאכן שינו את הערכים.

האם מדובר בBUG של פיתון? לא, כך פיתון תוכננה. הרעיון הוא לאפשר גמישות מקסימלית לתוכנתים. פיתון אומরת "ראו, יש סיבה שהמשתנים הללו מסוימים. מי שכתב את הקוד לא רצה שתוכנית חיצונית תיגש ותשנה אותם. אבל אם אתם יודעים מה אתם עושים, והקדשתם את הזמן להתגבר על ההסתירה, אז בבקשה – שנו כל מה שתרצו". יכולת זו שימושית רק במקרים נדירים, אבל במקרה הצורף היא אפשררת לנו, לדוגמה, לבצע שינויים בספריות פיתון שיש בהםאגים, בלי להזדקק לקוד המקורי.

אם כך, משתנים מסוימים בפייתון הם למעשה דרך שלנו לסמן כי אין לשנות את תוכן המשתנה שלא על ידי קוד של המחלקה עצמה. עם זאת, זהו רק סימן – ומשתמש חיוני יכול לשנות את ערכי המשתנה המוסתר, אם יבחר בכך.

שימוש ב-accessor ו-**mutator**

סקרנו שתי שיטות גישה ל-members של מחלקה.

השיטה הראשונה, היא לגשת אל ה-members ישרות. לדוגמה `x1.plane` היא גישה ישירה ל-member של `plane1`. אם מי שתכנת את המחלקה `Plane` דאג להס提ר את `x`, עדין יוכל לגשת אליו בעזרת `plane1._Plane_x`.

השיטה השנייה, היא באמצעות שימוש בMETHODS שנמצאות במחלקה ומשמשות במיוחד לטובת קרייה ושינוי של members. לדוגמה, `get_position` היא דוגמה לMETHODS כזו. ראיינו שם נקרא `get()` `plane1.get_position` נקבל את ערכי המיקום, למרות שהם מוסתרים. לMETHODS שמאפשרת קרייה של members של מחלקה קוראים accessor ונהוג שהוא שהוא מתחילה ב-`get`.

METHODS שמאפשרת שינוי ערכים של members נקראות mutator ונהוג שהוא מתחילה ב-`set`. לדוגמה, אם נרצה לאפשר שינוי המיקום של המטוס, נגדיר פונקציה בשם `set_position`, שתתקבל מיקום כפרמטר ותשנה בהתאם את מיקום המטוס.

יש דיון נרחב האם שימוש ב-accessors ו-**mutators** (getters, setters) הוא נכון, לא רק בשפת פיתון אלא באופן כללי בתכונות מונחה עצמים:

JAVA TOOLBOX

By Allen Holub, JavaWorld | SEP 5, 2003 1:00 AM PT

HOW-TO

Why getter and setter methods are evil

Make your code more maintainable by avoiding accessors

<http://www.javaworld.com/article/2073723/core-java/why-getter-and-setter-methods-are-evil.html>

ולעומת זאת:

Getters and Setters Are Not Evil



by Bozhidar Bozhanov MVB · Oct. 14, 11 · Java Zone

<https://dzone.com/articles/getters-and-setters-are-not>

היתרון של accessor, mutators המתכונת לשימוש בערכים המוכנסים ולבודד תקינות. לדוגמה, אם המתכונת יודע שיש מקום בעייתי שאסור שימושו יימצא בו הוא יכול למנוע זאת באמצעות קוד מתאים. הקוד הבא לא מאפשר להזין למטעום מקום שלילי או את מקום 4,4 כיוון שנמצא שם מגדל שאינו מאפשר מעבר מטעומים:

```
def set_position(self, x, y):
    if (x, y) == CONTROL_TOWER_LOCATION:
        print("Location of the tower")
    elif x < 0 or y < 0:
        print("Illegal location")
    else:
        self.__x = x
        self.__y = y
        print("Position set")
```

מайдך, טענים נגד accessors, mutators טענות רבות, בין היתר שהם מסובכים את כתיבת הקוד. לדוגמה, אם יש אובייקט בשם a ו-member בשם b, אז במקומות לכתוב כך:

a.b += 1

אנחנו נאלצים לכתוב, פחות או יותר, כך:

a.set_b(a.get(b) + 1)

ספר זה אינו שואף לפסוק בשאלת מה השיטה העדיפה. נציין שחשוב להכיר את שתי השיטות מכיוון שתיהן נפוצות בקוד בו אתם עשויים ליהי קול בעtid, וחשוב לדעת להשתמש ב-accessors ו-mutators, וכן בשביל לקבל בהמשך החלטה לא להשתמש בהם.

יצירת מודולים ושימוש ב-import



הקוד שלנו כרגע כולל גם את הגדרת המחלקה, גם את ה-`main` ואולי גם עוד פונקציות וקבועים שהגדרכנו. מדוע כדאי לשנות את זה? משומ שם נרצה להשתמש במחלקה שלנו בתוכנית אחרת, נצטרך להעתיק את כל הקוד שלנו ואז לשנות מה שצריך. לא פתרון נוח. היינו רוצים שתוכנית אחרת תוכל "לייבא" רק את המחלקה שהגדרכנו, ואולי כמה קבועים שקשורים אליה – וזהו.

לשם כך, נבצע שתי פעולות: ראשית נפריד את המחלקה לקובץ עצמאי. שניית, נבצע בתוכנית הראשית `import` למחלקה. נראה איך זה מתבצע.

נשמר קובץ פיתון בשם `uk.transport`. לטובת ההמשך הגדרנו מחלקה נוספת, `Boat`. להלן הקוד שבקובץ:

```
import random
CONTROL_TOWER_LOCATION = (4, 4)

class Plane:

    def __init__(self):
        self.__x = 0
        self.__y = 0

    def update_position(self):
        self.__x += random.randint(-1, 1)
        self.__y += random.randint(-1, 1)

    def get_position(self):
        return self.__x, self.__y

    def set_position(self, x, y):
        if (x, y) == CONTROL_TOWER_LOCATION:
            print("Location of the tower")
```

```

        elif x < 0 or y < 0:
            print("Illegal location")
        else:
            self.__x = x
            self.__y = y
            print("Position set")

class Boat:

    def __init__(self):
        self.__x = 0
        self.__y = 0

def main():
    print("This main function is not reached if the file is
imported")

if __name__ == "__main__":
    main()

```

cut גדי רtocnit שמשתמשת במחלקה שנמצאת בקובץ :transport.py

```

import transport

def main():
    plane1 = transport.Plane()
    plane1.set_position(3, 4)
    print(plane1.get_position())

if __name__ == "__main__":
    main()

```

בשורה הראשונה אנחנו מבצעים import לקובץ transport. cut זו הזרמתה הנדרת להסביר את התפקיד של

```
if __name__ == '__main__':
```

בקובץ transport.py קיימת פונקציה בשם `main`. כאשר אנחנו מיבאים את הקובץ, אנחנו לא מעוניינים להפעיל את `main` של הקובץ המקורי – יש לנו `main` משלהנו שאנו שודדים אותו. אנחנו רוצים רק את ההגדרות של המחלקות, וקבועים כלשם אם הם קיימים, לדוגמה מיקום המגדל. תנאי ה-`if` הנ"ל מודיע לפיתון "אם הגעת לקובץ זהה תור כדין ריצת התוכנית ולא בעקבות `import`, תריץ את `main`" – בדוק מה שרצינו שקרה.

לאחר שביצענו `import`, אנחנו יכולים להשתמש במחלקות שייבאנו אל התוכנית שלנו. השימוש במחלקות מצירפינו קטן בקוד – לפני שאנו מגדירים אובייקט מסווג `Plane` צריך לסמן שהאובייקט שייר לקובץ `transport` שיבאנו, כפי שניתן בשורה השלישית. רגע – איך פיתון לא ידע בעצמו ש-`Plane` הוא מחלוקת שמודגרת ב-`transport`? ובכן, פיתון יודע, אבל פיתון מניח שיכל להיות שינה עוד מחלוקת בשם זה בקובץ אחר. מקרה כזה עלול בהחלה להתרחש, במקרה שיש מתכוונים רבים שכותבים מודולים נפרדים. לכן, פיתון>Dורש שנציג בפירוש לאיזה קובץ אנחנו מתכוונים. ואם בכלל זאת אנחנו רוצים לוותר על ציון שם הקובץ? אפשר לוותר על כך בשינוי קטן. שימו לב לצורה הבאה של `import`:

```
from transport import Plane
```

```
def main():
    plane1 = Plane()
```

cut פיתון יודע שגם אנחנו כתבים `Plane` אנחנו מתוכנים רק למחלוקת הנ"ל מתוך `planes`, אך אפשר לוותר על ציון שם הקובץ. שימו לב לכך שגם אין היה קיימם קובץ אחר שיש בו מחלוקת בעלייה אותו שם, כרגע "דרסן" את שם המחלוקת בקובץ الآخر ולא נוכל להשתמש בו עוד.

ביצענו `import` רק למחלוקת אחת מתוך הקובץ וכן כתה רק למחלוקת `Planes` יובהה. המחלוקת השנייה, `Boat`, נותרה לא מוכרת לתוכנית הראשית. יצרנו את המחלוקת `Boat` כדי שנוכל להתרשם מכך.

לעתים תיתקלו בייבוא של כל המחלקות והקבועים של הקובץ באופן הבא:

```
from transport import *
```

```
def main():
    plane1 = Plane()
    boat1 = Boat()
```

צורה זאת של יבוא עלולה לגרום לבאגים. מדוע? מכיוון שניתן להיות שבשני קבצים שונים ישן שתי מחלוקות בעלות שם זהה. לדוגמה, נניח שיש מחלוקת בשם `Plane` גם בקובץ `transport.py` וגם בקובץ אחר, נניח

?Plane? airports.py האם מה שמודגר ב-transport או ב-airports? התשובה היא שהקובץ שייבאנו אחרון "דורס" את ההגדרות הקודמות. לכן, שיטה זו אינה מומלצת.

אתחול של פרמטרים

אפשרויות הטיסה של המטוסים השתנו, כאשר נפתח לרווחת הציבור שדה תעופה נוסף.Cut אנחנו רוצים לאפשר לחלק מהמטוסים להMRIA משדה התעופה החדש. נניח שהמיקום של שדה התעופה הוא (5, 5). כלומר, אנחנו צריכים להעביר את קואורדינטות ההרמראה של המטוס כאשר אנחנו יוצרים מטוס חדש, מה שכמובן נכון לבצע ב-`__init__`:

```
def __init__(self, airport):
    self.__x = airport[0]
    self.__y = airport[1]
```

אם אנחנו רוצים להגדיר מטוס חדש, אנחנו צריכים להגדיר אותו מראש עם הפרמטרים המבוקשים. לדוגמה:

```
from transport import Plane
NEW_YORK = (5, 5)
```

```
def main():
    american_airlines = Plane(NEW_YORK)
    print(american_airlines.get_position())
```

קביעת ערך ברירת מחדל

כיוון שרוב המטוסים ממראים משדה התעופה היישן, היינו רוצים שברירת המחדל למטוס חדש שמראה תהיה שדה התעופה היישן, שמייקומו (0,0). נעשה שינוי קל במתודת האתחול:

```
def __init__(self, airport=(0, 0)):
    self.__x = airport[0]
    self.__y = airport[1]
```

כעת, פיתון בודק אם כאשר אנחנו יוצרים מטוס חדש אנחנו מעבירים את המיקום כפרמטר. אם כן – המיקום שהעבכנו קבוע, אחרת – מיקום ברירת המחדל קבוע. נוכל להגדיר מטוסים חדשים באופנים הבאים:

```
american_airlines = Plane(NEW_YORK)
elal = Plane()
```

האובייקט `elal` קיבל את ערכו המיקום של ברירת המחדל – קלומר 0, ואילו האובייקט `american_airlines` קיבל את ערכו ההתחלת שהעבכנו לו – מיקום שדה התעופה של ניו יורק.

המתודה `__str__`

נסה לעשות `print` למטוס שהגדכנו, באמצעות הפקודה `print american_airlines`:

```
<transport.Plane object at 0x01031FB8>
```

air פיתון יודע מה להדפיס כאשר אנחנו עושים `print` לאובייקט מסוים? לכל אובייקט ישנה מתודה `__str__`. אנחנו יכולים לדרכו את `__str__` ולהחליפּ אותה בכל צורת הדפסה שנרצה. הנה נבצע זאת:

```
def __str__(self):
    return "Plane position: {}".format(self.get_position())
```

הסביר: ראשית, המתודה צריכה להחזיר ערך, שהוא הערך שיודפס. לכן ישנו במתודה `return`. הערך שמחזר הוא מחרוזת, שחלקה קבוע וחלקה מקבל את הערך שמחזירה המתודה `get_position` ומעביר אותו לפורמט המחרוזת.

אם עכשוו נבצע `print`, נקבל:

```
Plane position: (5, 5)
```

זהו. יותר יפה ממה שהחזרה פקודת הדפסה לפני השינוי? ☺

יצירת אובייקטים מרובים

עד כה יצרנו אובייקט אחד מהמחלקה שלנו. אפשר ליצור מה"תבנית" כמה אובייקטים שאנחנו רוצים, רק צריך לדאוג שלכל אובייקט יהיה שם ייחודי, אחרת נדרס אובייקטים שכבר הגדרנו. נגיד ארבעה מטוסים:

```
elal = Plane()
american_airlines = Plane(NEW_YORK)
british_airways = Plane(LONDON)
lufthansa = Plane(BERLIN)
```

מטוס אלעל ממיריא מנוקודת בירית המחדל שלנו ואילו היתר ממיראים ממיקומים שונים כפרמטרים.

אם אנחנו רוצים לבצע פעולה כלשהי על כל המטוסים, נכניס אותם לרשימה (אם אנחנו רוצים לאפשר הוספה מטוסים – אחרת tuple) וזה יוכל לעבור על כלם בלאלה:

```
fleet = [elal, american_airlines, british_airways, lufthansa]
for plane in fleet:
    print(plane)
```



תרגיל סיכון בניין – כתיבת class משופר

שפרו את ה-class של החיה שיצרתם:

העבironו את ה-class לקובץ חיצוני ועשו לו `import` -

הסתירו את שם החיה ואת הגיל שלה (זיכרון: `__`) -

אפשרו לקבוע את שם החיה בזמן יצירת האובייקט -

אפשרו לשנות את שם החיה (method `set`) -

אפשרו לקרוא את שם החיה ואת גיל החיה (method `get`) -

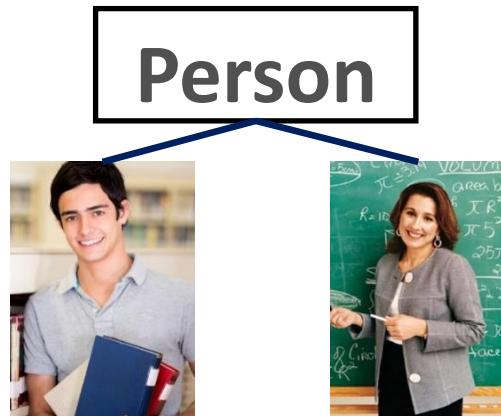
אפשרו הדפסת פרטי החיה על ידי קראיה ל-`print` (method `__str__`) -



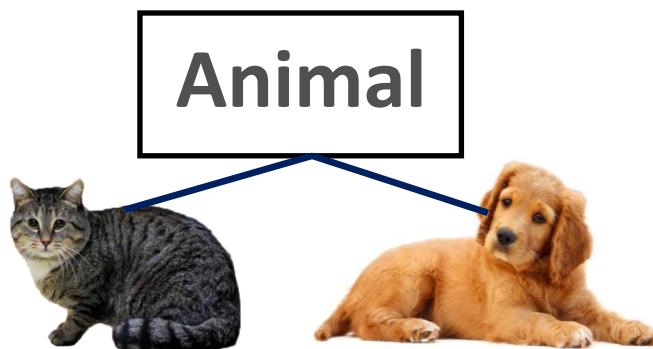
ירושא – inheritance

לפעמים מחלוקת היא סוג ספציפי של מחלוקת אחרת. לדוגמה – חתול הוא דוגמה ספציפית של חייה, ומטווו הוא דוגמה ספציפית של כלי תחבורה. הרעיון של יושא מאפשר לנו לחתות מחלוקת קיימת וליצור ממנה מחלוקת חדשה, ש כוללת את כל התכונות שקיימות במחלוקת שירשנו ממנה ועוד תכונות נוספות, שהן מיוחדות רק למחלוקת החדשה שיצרנו. המחלוקת החדשה, זו שיורשת מהמחלוקת המקורי, נקראת subclass. המחלוקת שירשנו ירשנו נקראת superclass. לדוגמה:

"Person" superclass של המ sublasses "Student" ו- "Teacher" -



"Animal" superclass של המ sublasses "Dog" ו- "Cat" -



כדי להמחייב איר מייצרים subclass, בטור התחלה ניצור מחלקה בשם Person, עם כמה מתודות בסיסיות:

```
class Person:

    def __init__(self, name='Shooki', age=20):
        self.__name = name
        self.__age = age

    def say(self):
        return "Hi :)"

    def __str__(self):
        return "Person {} is {} years old".\
            format(self.__name, self.__age)

    def get_name(self):
        return self.__name

    def get_age(self):
        return self.__age

    def set_name(self, name):
        self.__name = name

    def set_age(self, age):
        self.__age = age
```

כעת ניצור בית ספר. בבית ספר יש מורים ותלמידים.

למורים יש:

שם -

גיל -

שכר -

לתלמידים יש:

- שם

- גיל

- ממוצע ציונים

יש לנו כבר את המחלקה Person. בואו נשתמש בה. נגיד שמהמחלקות החדשות יורשות מ-Person – שימוש לבשמצינים זאת בסוגרים ליד הגדרת המחלקה:

class Teacher(Person):

class Student(Person):

שאלה למחשבה: נניח שהגדכנו את Teacher באופן שאין בו כלום, כך:

class Teacher(Person):
 pass

מה יבצע קטע הקוד הבא?

```
teacher1 = Teacher()
print(teacher1)
```

תשובה: כאשר יוצרים אובייקט מסוג Teacher, פיתון מחפש את ה-__init__ של Teacher. כיוון שלא הגדכנו לו __init__, פיתון הולך אל המחלקה ממנה Teacher יירש, כמובן Person, ומפעיל את ה-__init__ של Person. באתחול של Person נקבעים מחדש משתני ברירת מחדל ו-1 teacher1 יירש אותם. נציג שאלתו ל-Teacher כך היה __init__, לא היה מופעל אוטומטית ה-__init__ של המחלקה ממנה הוא ירש. לגבי פקודות הדפסה, למרות שלא הגדכנו שום מתודה ל-Teacher, הוא יירש את __str__ מ-Person ולכן ידפו:

Person Shooki is 20 years old

כפי שהחלהנו, צריך לכל מורה יהיה שכר. لكن נוסף לו מתודת אתחול. המתודה צריכה לגרום לכך שמה שאפשר נירש מ-Person ומה שספציפי למורה, נגיד. נבצע זאת כך:

class Teacher(Person):

```
def __init__(self, name, age, salary):
    Person.__init__(self, name, age)
    self.__salary = salary
```

סביר מה ביצענו. הגדרנו מחלקה בשם Teacher שירשת מ-Person. במתודת `__init__` הפעלנו את מתודת `__init__` של Person עם פרמטרים שלחננו לה (`name, age`). כיוון ש-Person לא יודעת מה לעשות עם פרמטר `salary`, הגדרנו לו-Teacher משתנה נוספת. למעשה, משתנה זה הוא כל מה שבידיל כרגע בין Teacher ל-Person. הגדרנו פשוט ש-Person הוא Teacher עם שכר.

אם אנחנו רוצים לקרוא למתחות של המחלקה ממנה ירשנו, ניתן להשתמש ב-`super`. פונקציה זו מוצאת את המחלקה ממנה ירשנו. הקוד הבא קורא לפונקציית `__init__` של המחלקה ש-Teacher ירשה ממנה, כלומר ל-`:Person`:

```
class Teacher(Person):

    def __init__(self, name, age, salary):
        super(Teacher, self).__init__(name, age)
        self.__salary = salary
```

הסיבות המרכזיות לשימוש ב-`super` הן:

- א. אנחנו רוצים לקרוא למתחה של מחלקה, אשר המחלקה שלנו דרצה עם מתחה בעלת שם זהה.
- ב. במקרים של מחלקות שירשות מספר מחלקות, שימוש ב-`super` הוא הכרחי כדי לוודא שמבצעים `__init__` של כל המחלקות מהן ירשנו.

מה יודפס אם נרים כתעת את הקוד הבא?

```
teacher1 = Teacher("Barak", 40, 100)
print(teacher1)
```

תשובה:

Person Barak is 40 years old

וזאת מכיוון ש-`Person` יירש את ה-`__str__` של `Teacher`.

תרגיל מסכם – ירושה



צרו את המחלקה `Student` שיורשת מ-`Person`. עלייכם:

- לקבוע ערכי בריית מחדל כרצונכם
- להוסיף לאתחול משתנה נסתר שישמר את ממוצע הציונים
- הוסיף לממוצע הציונים מתודות mutator-accessor-i

פולימורפיזם

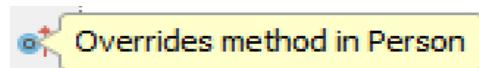
הסיבה שהגדכנו ל-`Person` מתודה בשם `say` היא כדי לבדוק לטובות חלק זה. נפעיל את `say` על אובייקט המורה שלנו, ונקבל את המחרוזת "(;) Hi". אבל נרצה שהמורה יאמר משחו "חוד'", לדוגמה `"Good morning cyber students!"`, וכמו כן שיירש את כל התכונות של `Person`.

כדי לגרום למethode `say` לעבוד כמו שאנו רוצים, נגדיר אותה מחדש בתחום `Teacher`:



```
def say(self):
    return("Good morning cyber students!")
```

שימוש לב לסייעו היגייל הכהול לצד המתודה. אם נעמוד עליו עם העכבר, נקבל הסבר:



במילים אחרות – דרשו את המתודה `say` של `Person`.



מה יקרה אםicut אם נפעיל את say על אובייקט מסווגTeacher? נקבל את המשפט שרצינו.

אנחנו יכולים להגדיר מחלקה נוספת שתיריש מ-`Person`, לדוגמה `teacher`. כמו של-`teacher` יש מתודת `say` "יחודית", גם ל-`Student` אפשר להגדיר `say`.icut אם מוצע להבין מדווקוראים לפעולה זו "פולימורפיזם". משמעו "רבה" ואילו "morph" משמעו "צורה". הרבה צורות. ואכן, לעתים אובייקט מגיע בהרבה צורות, שיש להן בסיס משותף. מורה, תלמיד וסוגים שונים של בעלי מקצוע הם כולם צורות שונות של `Person`.

לעתים אנחנו רוצים subclass ipna למשתנים מוסתרים של ה-class superclass שלו. חשוב לדעת שימושים מוסתרים הינם מהיורשיים. לכן, אם אנחנו רוצים להשתמש ב-members superclass של `members`, ניתן לבחור לעשות זאת באמצעות מתודות accessor (זכור יש דיון נרחב בשאלת האם אכן רצוי לעשות זאת). לדוגמה, נעדכן את מתודת ה-`str` של `Teacher`:

```
def __str__(self):
    return "Teacher {} is {} years old, salary {} per hour".\
        format(self.get_name(), self.get_age(), self.__salary)
```

כל מה שצריך לעשות כדי להשתמש בפונקציה זו הוא פשוטAAD לבע `print` לאובייקט שלהם. לדוגמה:

```
teacher1 = Teacher("Barak", 40, 100)
print(teacher1)
```

הפונקציה `isinstance`

בבית הספר המעלוה של נועה חמיצים נפתחה מגמת סייבר. נגידר תלמיד שלומד בмагמה בתור `CyberStudent`, קלומר אובייקט שיורש מ-`Student` אך צזה שכלל גם ציון סייבר:

```
class CyberStudent(Student):
```

```
    def __init__(self, name, age, grade, cyber_grade):
        super(CyberStudent, self).__init__(name, age, grade)
        self.__cyber_grade = cyber_grade

    def get_cyber_grade(self):
        return self.__cyber_grade
```

עד כה אין שם דבר חדש במאה שעשינו. הגדרנו subclass של `Student`. אך כתה, מנהל בית הספר מבקש שכל תלמיד שהצין שלו בסיביר טוב יקבל הודעה "Wow!".

מתקנת הכנס לרשימה בשם `students` את האובייקטים של כל התלמידים, גם אלו שנמצאים במגמת סייבר וגם אלו שאינם. לאחר מכן המתקנת כתוב את הקוד הבא. האם הקוד יעבד באופן תקין?

```
rami = CyberStudent("Rami", 16, 90, 95)
yael = CyberStudent("Yael", 17, 85, 100)
guy = Student("Guy", 15, 95)

students = [rami, yael, guy]

for student in students:
    if student.get_cyber_grade() >= GOOD_GRADE:
        print("Wow!")
```

תשובה:

כאשר נרים את הקוד, נקבל שגיאת הרצה -

Wow!

Wow!

...AttributeError: 'Student' object has no attribute 'get_cyber_grade'

הסיבה היא שאנו לא-`CyberStudent` יש מתודה בשם `get_cyber_grade` אין מתודה כזו. לכן, האובייקט הראשון מסווג `Student` שנמצא ברשימה יגרום לשגיאת הרצאה הנ"ל. על מנת לפתור את הבעיה, הפונקציה `isinstance` מקבלת שם של אובייקט ושם של מחלקה, ובודקת אם האובייקט שיש למחלקה או יורש מהמחלקה זו. אם כן – מוחזר `True`, אחרת – `False`. חשוב לציין, שאובייקט תמיד שייך למחלקה של המחלקה שלו.

כעת נוכל לכתוב מחדש את הלולאה שלו, כך שרק אם `student` הוא מסווג `CyberStudent` תבוצע קריאה ל-`:get_cyber_grade`

```
for student in students:
    if isinstance(student, CyberStudent):
        if student.get_cyber_grade() >= GOOD_GRADE:
            print("Wow!")
```

נסים בשאלת מחשבה. מוגדרים התלמידים הבאים:

```
yael = CyberStudent("Yael", 17, 85, 100)
guy = Student("Guy", 15, 95)
```

מה תהיה התוצאה של כל אחת מהבדיקות הבאות?

```
print(isinstance(guy, Student))
print(isinstance(yael, CyberStudent))
print(isinstance(guy, Person))
print(isinstance(guy, CyberStudent))
print(isinstance(yael, Student))
```

תשובה:

שתי הבדיקות הראשונות יחזירו `True`.

הבדיקה השלישית תחזיר `True` מכיוון ש-`noam` הוא `Student`, כלומר `sub class` של `Person`.

הבדיקה הרביעית תחזיר `False`, כיון ש-`noam` הוא `Student`, והוא יורש מ-`CyberStudent`.
הבדיקה החמישית תחזיר `True`, מכיוון ש-`daniel` הוא `CyberStudent`, אשר יורש מ-`Student`.

תרגיל מסכם פולימורפיים – BigCat (קדagit: שי סדובסקי)



הגדרו מחלקת בשם `BigThing`, אשר מקבלת כפרמטר בזמן היצירה משתנה כלשהו (המשתנה יכול להיות כל דבר – מחרוזת, רשימה, מספר וכו'). למחלקה יש מתודה בשם `size`, אשר עובדת כך:

- אם המשתנה הוא מספר – המתודה מחזירה את המספר

- אם המשתנה הוא רשימה / מילון / מחרוזת – המתודה מחזירה את `len` של המשתנה

לדוגמה, עבור ההגדרה:

```
my_thing = BigThing('table')
```

התוצאה של `size(my_thing)` יהיה 5, מאורע המחרוזת 'table'.

עתה הגדרו מחלקת בשם `BigCat`, אשר יורשת מ-`BigThing` ומתקבלת כפרמטר בזמן היצירה גם משקל.

- אם המשקל גדול מ-15, המתודה `size` תחזיר "Fat"

- אם המשקל גדול מ-20, המתודה `size` תחזיר "Very fat"

- אחרת יחזיר "OK"

לדוגמה, עבור ההגדרה:

```
latif = BigCat('latif', 22)
```

התוצאה של `size(latif)` תהיה "Very fat"

פרק 11 – OOP מתקדם (תכנות משחקים באמצעות PyGame)



בחלק זה נלמד לכתב משחקים באמצעות מודול PyGame של פיתון.

בשלב הראשון נעשה שימוש בפונקציות בסיסיות של PyGame – ניצור מסך עם גרפייה נעה, אשר מגיבה למקלדת ולעכבר ומשמיעת צלילים. בשלב השני, נשלב OOP (תכנות מונחה עצמים), על מנת ליצור משחקים המבוססים על שיטות של עצמים ובדיקה אם שני עצמים נוגעים זה זהה (לדוגמה משחקי יריות, משחקי כדור ומשחקי מבוכים).

תוכן העניינים של פרק זה הינו כדלקמן:

- א. כתיבת שلد של PyGame
- ב. הוספת תמונה רקע
- ג. הוספת צורות גאומטריות
- ד. הuzzת צורות על המסך
- ה. הוספת דמויות Sprites
- ו. קבלת קלט מהעכבר
- ז. קבלת קלט ממקלדת
- ח. השמעת צלילים
- ט. שילוב OOP
- י. בדיקת התנגשויות בין עצמים

כתיבת שלד של PyGame

לפני הכל, בתוך cmd כתבו

```
pip install pygame
```

כדי להיות מסוגלים לייבא אותו.

```
import pygame

WINDOW_WIDTH = 700
WINDOW_HEIGHT = 500

pygame.init()
size = (WINDOW_WIDTH, WINDOW_HEIGHT)
screen = pygame.display.set_mode(size)
pygame.display.set_caption("Game")

pygame.quit()
```

ראשית כל הסקריפט שלנו צריך לבצע import לモודול PyGame.

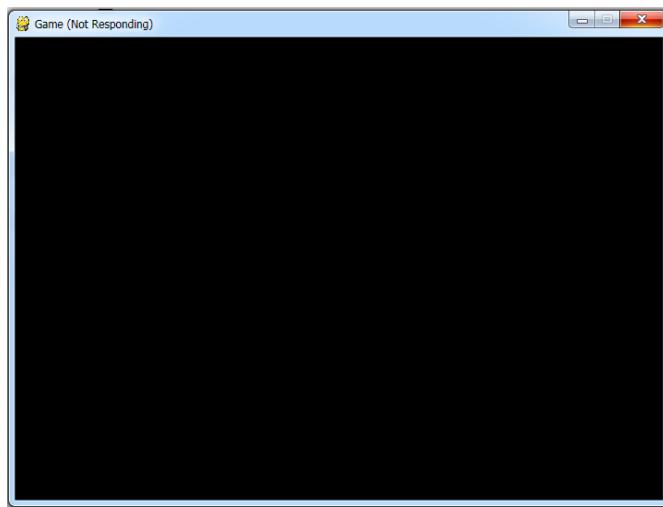
נגיד בטור קבועים את גודל החלון – כמות הפיקסלים לרוחב ולגובה החלון.

לאחר מכן נאתחל את PyGame על ידי קרייה לפונקציה `init()`, כך שנוכל להתחיל להשתמש בפונקציות שונות שלו.

יצור מסך בגודל שקבענו ונקבע לו את השם "Game".

מיד לאחר מכן תבצע הפקודה `quit()`.

כאשר נריץ את התוכנית, יופיע לרגע קצר מסך של PyGame ואז המסך יסגר.



אם נרצה שהמסך יישאר, נכנס לפני שורה 13 לולאה אינסופית, אך שימו לב שכרגע – באופן זמני – הדריך היחידה לסגור את התוכנית היא באמצעות לחץ העצור של PyCharm או באמצעות ניהול המשימות. לחוץ סגירת החלון שבקצתה העליון של המסך עדין אינם עובד. מיד נסדר את זה ☺

נרצה לגרום למצב בו לחיצה על כפתור הסגירה של מסך המשחק שלנו מבצעת את מה שהיא אמורה לעשות, וסגרת את המסך. לשם כך, علينا לחתת לתוכנית שלנו הוראה מה לעשות במקרה שיש לחיצה על כפתור הסגירה:

```
finish = False
while not finish:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            finish = True
```

כל פעולה שהמשתמש מבצע במסך המשחק מגיעה אל רשיימה, וכל פעולה צו מוצמד סוג של אירוע – `event.type`. לדוגמה, לחיצה על כפתור סגירת המסך היא סוג אירוע בשם `PyGame.QUIT`. המתודה `pygame.event.get()` מספקת לנו את הרשימה של כל הפעולות שהמשתמש ביצע מאז הפעם האחרונה שקרה לנו. הקוד לעיל עובר על כל הפעולות שהמשתמש ביצע ובודק אם אחת מהן היא `PyGame.QUIT`. אם כן – הקוד י יצא מהלולאה האינסופית.

שינוי רקע

הרקע השחור אמן נחמד, אך לעיתים נרצה לצבוע את המסך שלנוצבע אחר, לדוגמה בצביע לבן.

נצרך להגיד את הצבע הלבן בתור `tuple` של 3 מספרים -

WHITE = (255, 255, 255)

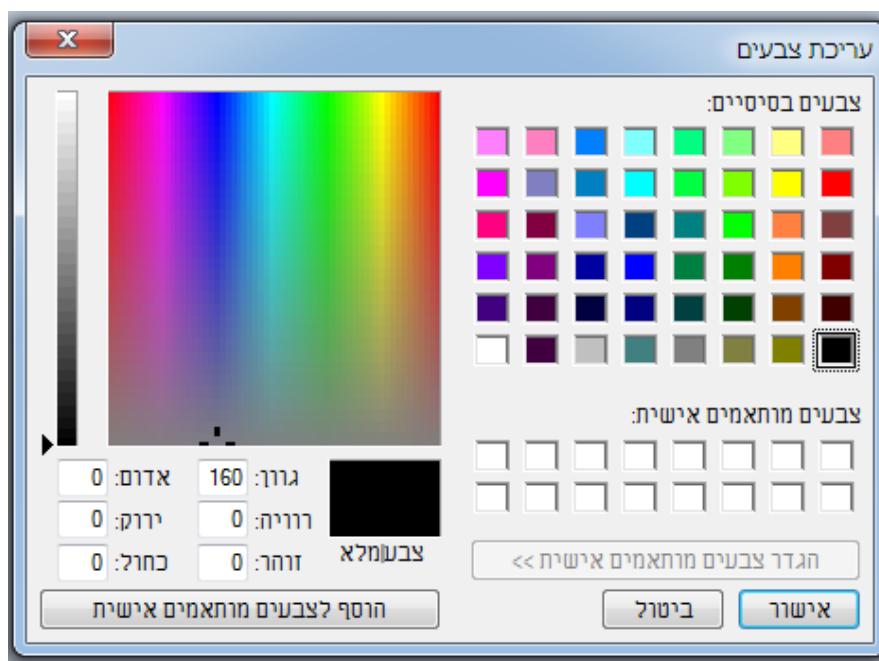
אלו ערכי RGB (Red, Green, Blue) של הצבע הלבן. כל צבע מוגדר על ידי שלישית מספרים שונה. לדוגמה:

RED = (255, 0, 0)

וכמוכן:

BLACK = (0, 0, 0)

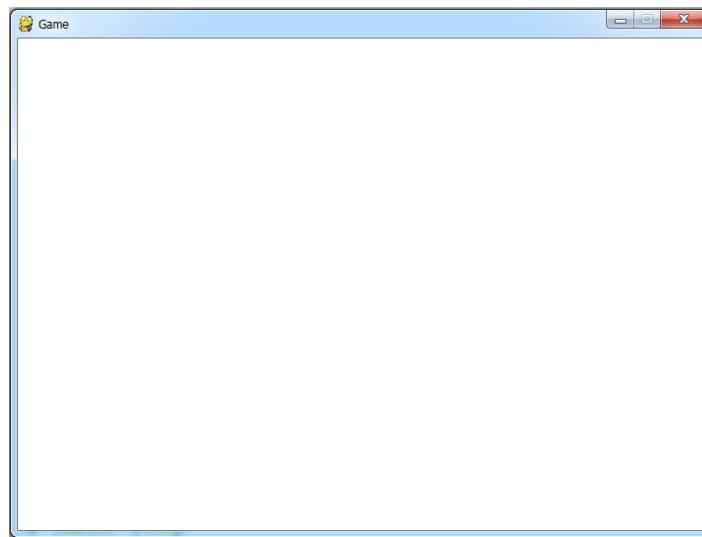
אפשר למצוא את שלישית ה-RGB של כל צבע שתרצה באמצעות תוכנת `paint` (צייר) הבסיסית שmagia עם Windows, תחת תפריט "עריכת צבעים":



כעת נגדיר בתוכנית שהצבע של המסך שלנו הוא לבן. לפני הכניסה לולאה, נוסיף את השורות הבאות:

```
screen.fill(WHITE)
pygame.display.flip()
```

והתוצאה:

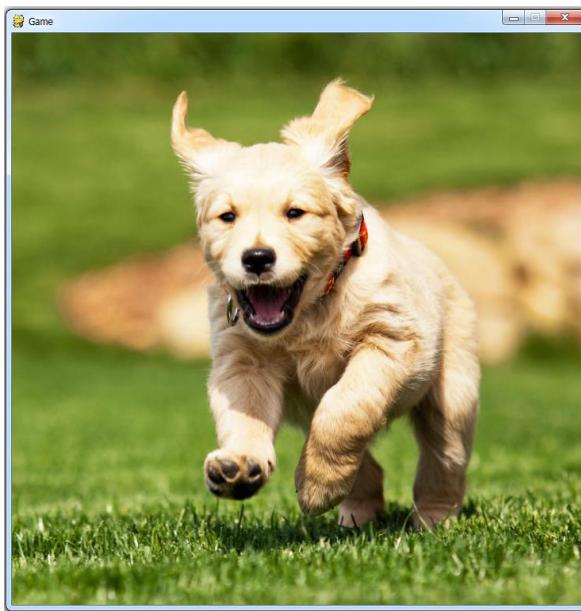


מה מבצעת הפקודה `?flip`?

מסך המחשב קורא את המצב של כל פיקסל ממוקם מוגדר בזיכרון המחשב (שנקרא "video memory"). הפקודה `fill` עדין לא משנה את המצב של הזיכרון ממנו המסר מציג, אלא רק את האובייקט `screen`. בשורה הבאה, אנו מורים לתוכנית שלנו לבצע `"flip"`, כלומר להחליף את המידע ב-video memory ששמרנו בתוך האובייקט `screen`. רק ביצוע ה-`flip` הוא משנה בפועל את מצב המסך שלנו.

במילים אחרות, אנחנו יכולים לעורק את האובייקט `screen` כפי רצוננו, אך ללא `flip` כל השינויים שנבצע יהיו נסתרים למשתמש ולא יוצגו על המסך.

רקע לבן הוא משעם קצר, בואו נעלם תמונה רקע!



ראשית נבחר תמונה כרצונו. שימו לב לגודל התמונה בפיקסלים. אפשר למצוא את הגודל באמצעות ה-properties של הקובץ. במקרה זה, הגודל הוא 720x720 פיקסלים.



שנו את הקבועים שנמצאים בראש התוכנית – גובה ורוחב החלון – על מנת להתאים אותו לגודל התמונה שבחורתם. לאחר ששינו את גודל המסר כדי שיתאים לתמונה, נטען אותה כתמונה רקע.

ראשית נגדיר קבוע בשם IMAGE שיכיל את שם הקובץ שלנו, לדוגמה:

```
IMAGE = 'example.jpg'
```

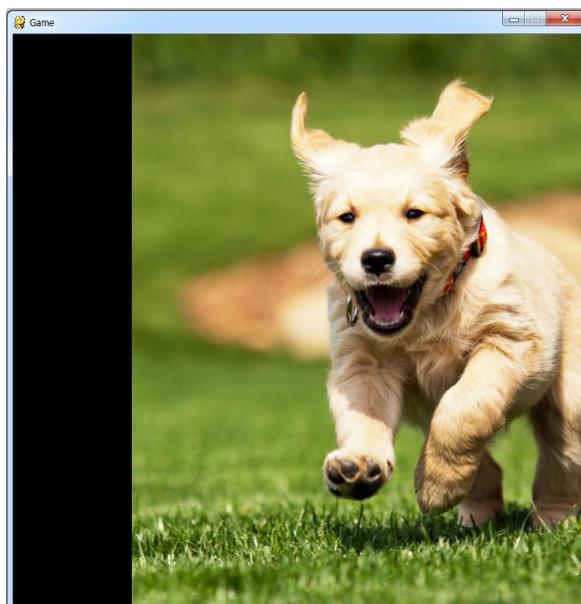
וכעת נטען את התמונה כתמונה רקע:

```
img = pygame.image.load(IMAGE)
screen.blit(img, (0, 0))
```

pygame.display.flip()

הפקודה השנייה טעונה את התמונה לתוך האובייקט screen, החל ממיקום (0, 0). זהה הפינה השמאלית העליונה של המסך. כיוון שדאגנו מראש שגודל המסר יהיה שווה לגודל התמונה, תמונה הרקע תופסת כעת את כל גודל המסך.

שים לב מה היה קורה אילו היינו מנסים לטעון את התמונה ממקום התחלתי (0, 150):



כלומר ככל שאנו עולים בפרמטר הראשון אנחנו זזים ימינה על המסך. ככל שאנו עולים בפרמטר השני אנחנו זזים למטה. אם הגדרנו מסך בגודל 720 על 720, הפיקסל הימני התחתון הוא במספר (719, 719).

הווסף צורות

כעת נוסיף כמה צורות על הרקע שלנו. ספריית PyGame מאפשרת לנו לצייר קוים, עיגולים, מלבנים, אליפסות וקשתות.

נתחיל בכר שנצייר קו. לשם כך, נשתמש במתודה `PyGame.draw.line`. מתודה זו מקבלת בתור פרמטרים:

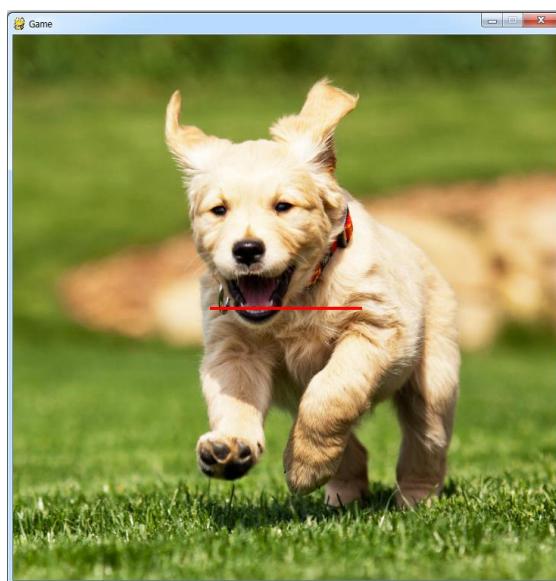
- משטח – אובייקט של `PyGame` עליו יצייר הקווים
- צבע הקווים
- נקודת התחלת קו
- נקודת סיום קו
- עובי הקווים (ברירת מחדל – 1)

כך:

```
pyGame.draw.line(surface, color, start_pos, end_pos, width=1)
```

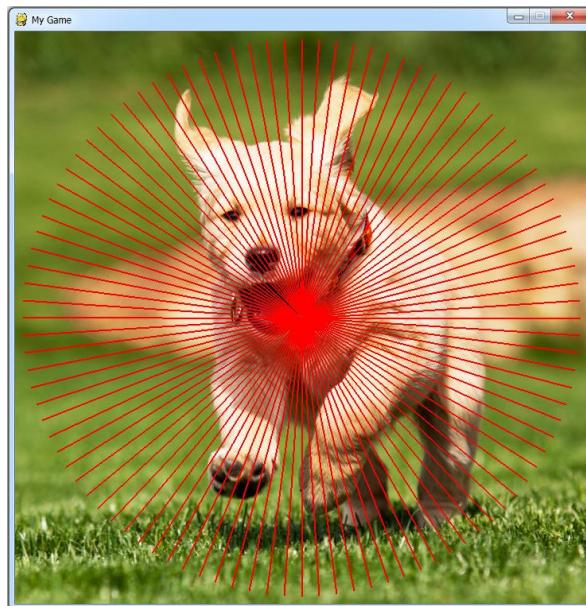
כלומר, כדי לצייר קו אדום מנקודה [260, 360], אל נקודה [460, 360] ובעובי 4:

```
pygame.draw.line(screen, RED, [260, 360], [460, 360], 4)
pygame.display.flip()
```



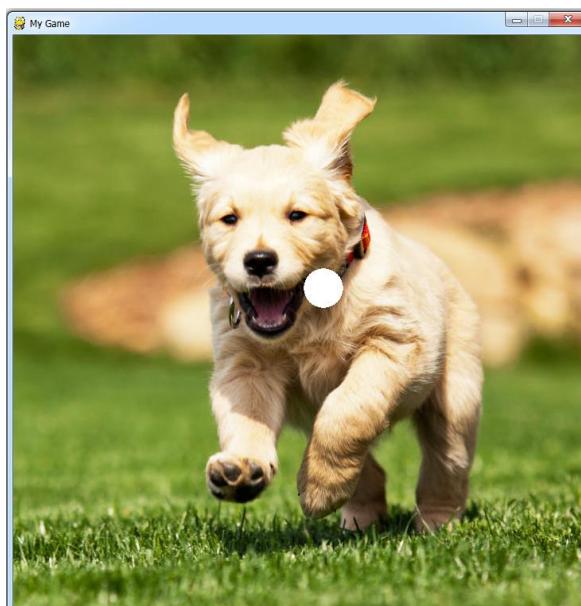
תרגיל

צרו בעצמכם את הדוגמה הבאה, המורכבת מ-100 קווים באורך 350 היוצאים ממרכז התמונה. טיפ: השתמשו ב-`import math` כדי לחשב את נקודות הסיום של כל קו, בעזרת הפונקציות `sin` ו-`cos`.

**תרגיל**

באמצעות התיעוד שנמצא בקישור הבא, צירו עיגול במקום כלשהו על המסך.

<https://www.PyGame.org/docs/ref/draw.html>



תזוזה של גרפייקה

עד עכשיו צירנו על המסר צורות שונות, מיד נלמד איך להציג אותן. איך אפשר לארום לצופה לחוש שצורה זהה על המסר?

מציריים צורה במקום מסוים. לאחר זמן מה מוחקים אותה, ומציירים אותה שוב במקום אחר, ליד המקום הקודם. כך נדמה לצופה שהצורה "זהה".

נתיחוס לשלבים השונים:

מחיקת צורה וציור שלה מחדש: כל מה שעשינו לעשות כדי למחוק צורה, הוא פשוט לצייר את תמונה הרקע מעלה. להזכירם, לצייר צורה מעל תמונה הרקע כבר למדנו.

המתנה של פרק זמן מוגדר בין המחיקה לציור: על מנת לעשות זאת נדרש לשעון (טימר), שיתזמן את פרק הזמן בו יש לחזור על פעולות המחיקה והציור מחדש.

כדי להוסיף שעון, נגידו:

```
clock = pygame.time.Clock()
```

שימוש לב לcker שה-C Clock עם גודלה. נוסף לקבועים שלנו את:

```
REFRESH_RATE = 60
```

כלומר, המסר מתעדכן 60 פעמים בשניה.

בנייה מחדש של הלולאה המרכזית של התוכנית:

```

clock = pygame.time.Clock()
ball_x_pos = 0
ball_y_pos = 0

finish = False
while not finish:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            finish = True

    screen.blit(img, (0, 0))
    ball_x_pos += 1
    ball_y_pos += 1

    pygame.draw.circle(screen, WHITE, \
                       [ball_x_pos, ball_y_pos], RADIUS)
    pygame.display.flip()
    clock.tick(REFRESH_RATE)

```

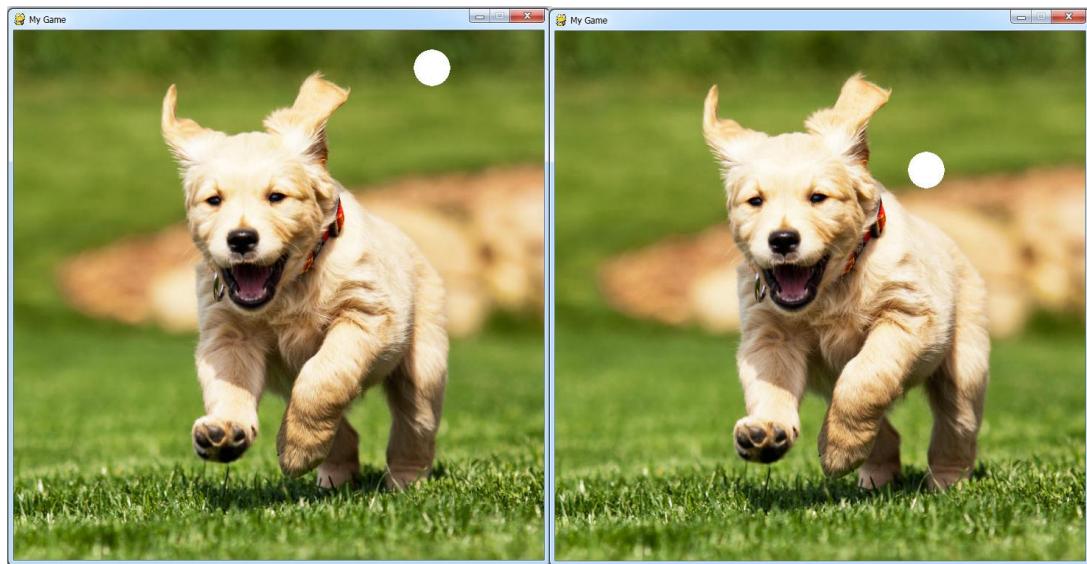
הכנסנו את הפקודה `flip` לתוך הלולאה מכיוון שכעת אנחנו צריכים לצייר את הרקע בכל פעם מחדש (וזאת על מנת למחוק את הצורה שאנוינו מעוניינים להציג).

במקרה הפקודות שמקדמות את ערכי ה-x וה-y ביחידות אחת, צריך לבוא קטע קוד כלשהו שմזיז את הצורה שלו. לדוגמה, אפשר להציג מעט את העיגול שציירנו בתרגיל הקודם. בדוגמה הקוד לעיל, אנו מזיזים את העיגול בפיקסל אחד למטה ובפיקסל אחד ימינה בכל איטרציה של הלולאה, רק כדי להמחיש את העקרון. שימושם לבשאננו לא בודקים ששיעור העיגול הוא הגיוני (לדוגמה, שהעיגול נמצא בתוך המסר...).

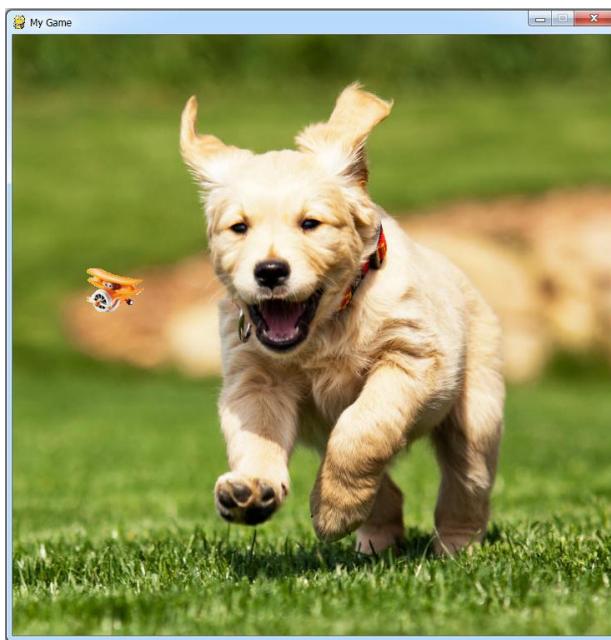
הפקודה האחרונה מגדרה לשעון שלנו להמתין פרק זמן לפני האיתרציה הבאה של לולאת ה-`while`. כאמור, בlij המתנה זו, לא נוכל לשנות במהירות ההתקדמות של הגרפיקה שלנו על המסך.

תרגיל – פינג פונג

שכללו את התרגיל בו ציירתם עיגול על המסר, כך שהעיגול יתקדם על המסר מצד לצד. במידה והעיגול נוגע בשולי המסר, שנו את וקטורי המהירות שלו כך שהוא שnitץ ממשטח.



ציור Sprite



סימנו להשתעשע עם צורות גאומטריות. כעת נניח על תמונה הרקע שלנו תמונה שחקן קטן, שנראית **Sprite**, ונציג אותה מקום למקום.

בתוך תמונה שחקן נבחר קובץ תמונה קטן:



שיםו לב שהמתווך נמצא על רקע אחד. מיד נבין את החשיבות שכך.

כדי ליצור תמונה עם רקע אחד נפתח את תוכנת הצייר, **paint**, וונטעיק לתוכה כל תמונה שנרצה. כעת נגדיר צבע רקע באמצעות **תפריט "עריכת צבעים"** אותו הכרנו כבר. חשוב להגדיר צבע שאינו נמצא בתמונה שבחרנו. במקרה זה בחרנו את הצבע הורוד שה-RGB שלו הוא **(255, 20, 147)**.

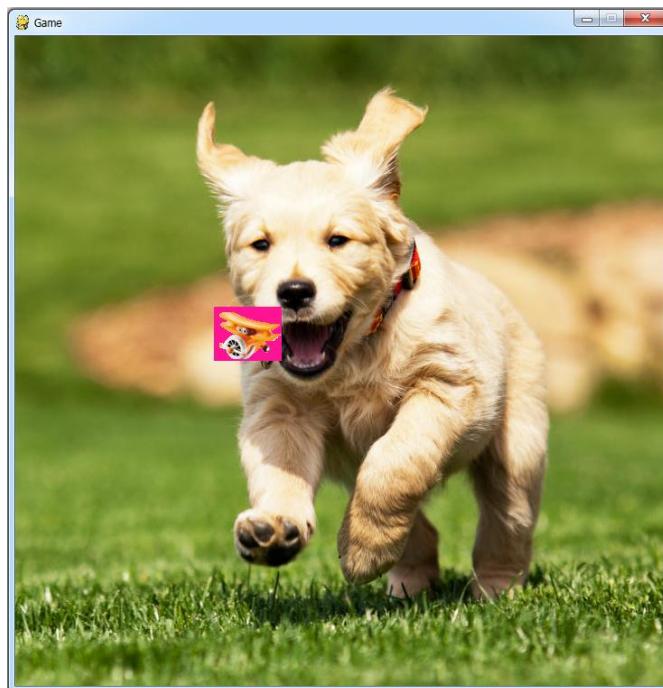
לאחר שיש לנו, מומלץ לשמר את התמונה בפורמט jpg. זהו פורמט שאינו מעוות את הצלבים בתמונה וכך הצלב שהכרנו בתור רקע ישמר כמו שהוא.

בשלב הבא נטען את ה-Sprite שלנו לתוכנית:

```
img = pygame.image.load(IMAGE)
screen.blit(img, (0, 0))
player_image = pygame.image.load('plane.png').convert()
screen.blit(player_image, [220, 300])
pygame.display.flip()
```

בשורות שנוספו אנחנו טוענים את קובץ התמונה שלנו וקובעים את המיקום שלו על תמונה הרקע.

אם נריץ את התוכנית כעת, נקבל את התוצאה הבאה:



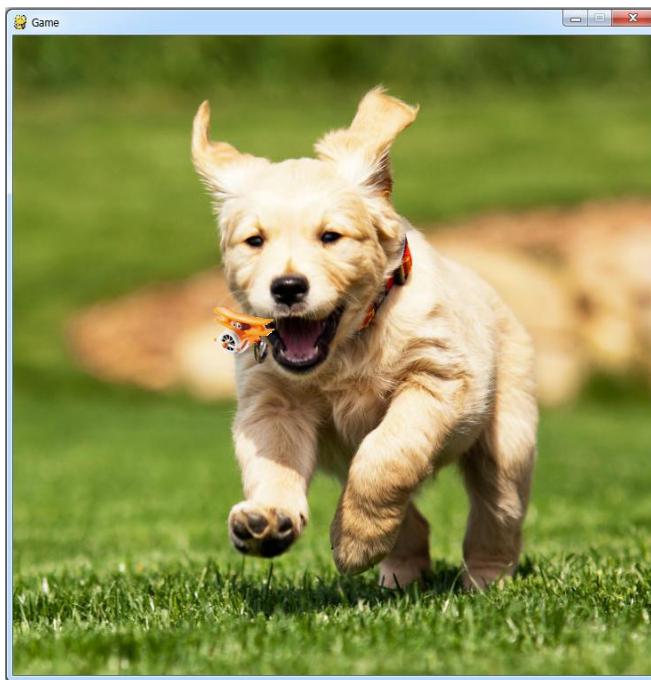
כאן נכנסו לשימוש הרקע שהגדכנו ל-Sprite שלנו.

בתחילת התוכנית נגדיר קבוע את הצלב שאיתו צבענו את רקע ה-Sprite שלנו:

PINK = (255, 20, 147)

נגידר לתוכנית שלנו להציגו מהצבע הזהה. כלומר, אם היא מצאה ב-`Sprite` שלנו פיקסל בצבע שהגדרנו, היא תתייחס אליו כשקופ ולא תצביע איתה מעל תמונה הרקע (שים לב לשורה המודגשת):

```
img = pygame.image.load(IMAGE)
screen.blit(img, (0, 0))
player_image = pygame.image.load('plane.png').convert()
player_image.set_colorkey(PINK)
screen.blit(player_image, [220, 300])
pygame.display.flip()
```



נפלא! כעת אנחנו יכולים להעלות כל תמונה מעל תמונה הרקע שלנו ולהזיז אותה.

קיבלה קלט מהעכבר

כאשר אנו מקבלים קלט מהעכבר, אנחנו רוצים למעשה לדעת שני דברים:

- מה מיקום העכבר על המסך

- אילו כפתורים לחוצים

לש machtanu, PyGame מספק לנו את המידע זהה די בקלות.

באמצעות המתודה `(PyGame.mouse.get_pos()`) אנחנו מקבלים את המיקום הנוכחי של העכבר. במידה שנרצה Sprite ינוע בהתאם למיקום העכבר, כל מה שנותר לנו לעשות הוא להודיע למסך שלנו לצייר את ה-Sprite במקומות של העכבר ואז לבצע `flip` למסך.

```
finish = False
while not finish:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            finish = True

    screen.blit(img, (0, 0))
    mouse_point = pygame.mouse.get_pos()
    screen.blit(player_image, mouse_point)

    pygame.display.flip()
    clock.tick(REFRESH_RATE)
```

כפי ששמתתם לב, האיקון של העכבר מופיע על ה-Sprite שלנו וזה לא נראה יפה במיוחד. לכן, בתחילת התוכנית, מיד לאחר פקודת `pygame.init()`, נרים את שורת הקוד הבאה, שתעלים את האיקון של העכבר:

```
pygame.mouse.set_visible(False)
```

כעת נרצה לבצע פעולות שונות על פי לחיצות המשתמש על העכבר.

ראשית נגדיר את לחצני העכבר. המספרים המשווים יכולים להשתנות בהתאם לערבים של PyGame, אנחנו נותנים להם שמות כדי שהקוד יהיה יותר קרייא:

```
LEFT = 1
SCROLL = 2
RIGHT = 3
```

בכל פעם שהמשתמש יקליק על הכפתור השמאלי של העכבר, נשמר את מיקום העכבר בראשימה שאנו חנו ניצור – נוכל להשתמש בראשימת המיקומים של העכבר בשבייל לעשות כל דבר שנרצה.

נבחן את הקוד הבא:

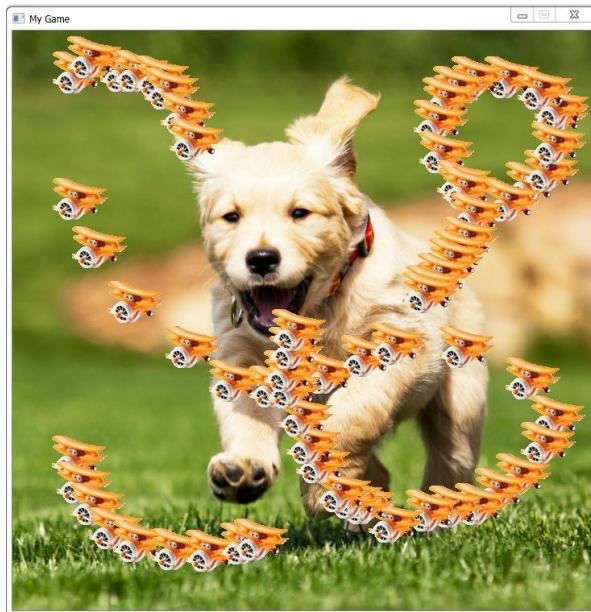
```
mouse_pos_list = []
finish = False
while not finish:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            finish = True
        elif event.type == pygame.MOUSEBUTTONDOWN \
            and event.button == LEFT:
            mouse_pos_list.append(pygame.mouse.get_pos())
```

התנאי שהגדכנו מתחילה בבדיקה `event.type` – אנו בודקים האם מדובר בלחיצה על כפתור כלשהו בעכבר. שימו לב שאין צורך להגיד את הקבוע `MOUSEBUTTONDOWN`, משום שהקבוע זהה מופיע כבר בתוך ה-`event`-`event` המוגדרים של PyGame. לאחר שווידאו שהתרחשה לחיצה על כפתור כלשהו בעכבר, אנו בודקים איזה כפתור בדיק נלחץ.

תרגיל – עכבר

כתבו תוכנית אשר בכל פעם שהעכבר נלחץ בקлик שמאלי, תשאר על המסך עותק של ה-Sprite שלו.

וודאו שניתן להקליק על העכבר ללא הגבלה.



קבלת קלט מהמקלדת

קבלת קלט מהמקלדת דומה למדи לקבלת קלט מהעכבר. גם כאן, אנו מחפשים event'ים בתוך הרשימה שחוזרת מקריאה ל-`PyGame.event.get()`, רק שהפעם נחפש אירועים שקשורים למקלדת.

ראשית, נבדוק שהיתה לחיצה על המקלדת, וזאת באמצעות ה-`event` שנקרא KEYDOWN:

```
if event.type == pygame.KEYDOWN:
```

בנהנזה שאכן הייתה לחיצה על המקלדת, נוכל לבדוק האם מקש ספציפי הוקש. כל המקשים במקלדת מתחילה ב-`K` ואחריו יש את המKeySpec. לדוגמה, מקש ה-`a` הוא `a_K`. נוכל לבדוק אם מקש ה-`a` הוקש על ידי:

```
if event.key == pygame.K_a:
```

```
finish = False
while not finish:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            finish = True
        elif event.type == pygame.KEYDOWN:
            if event.key == pygame.K_a:
```

השמעת צלילים

ראשית ניצור קובץ שמכיל את הצליל שנחנו רוצים להשמי. רוב הצלילים ניתנים למציאה ב-[youtube](#) (אפשר לחפש לדוגמה laser beam sound effect). לאחר מכן נוריד את קובץ ה-`mp3` באמצעות אתר כגון:

<http://www.youtube-mp3.org/>

שימוש לב שהורדת מוזיקה שיש עליה זכויות יוצרים באמצעות אתר זה הינה בניגוד לתנאי השימוש של יוטיוב.

YouTube mp3

<http://www.youtube.com/watch?v=KMU0tzLwhbE>

Convert Video

כדי לחזור רק קטע מסוים ממقطع קובץ mp3 שהורדנו, נשתמש באתר הבא:

<http://mp3cut.net/>



קטע יש בראשותנו קובץ mp3 עם הצליל המבוקש.

העלאה והשמעה שלו מתבצעת כך:

```
SOUND_FILE = "kaboom.mp3"
pygame.mixer.init()
pygame.mixer.music.load(SOUND_FILE)
pygame.mixer.music.play()
```

תרגיל סיכון ביןיהם



צרו סקריפט PyGame אשר מבצע את הדברים הבאים:

- העלאה של תמונה רקע
- הזזה של Sprite כלשהו על תמונה רקע

- אפשר יהיה להזיז את sprite על ידי העכבר
- אפשר יהיה להזיז את sprite על ידי המקלדת
- לחיצה על קליין שמאלי בעכבר תותיר את הסימן של sprite במקום שבו על המסך
- לחיצה על מקש space תמחק את כל sprites שצוירו על הרקע
- לחיצה על המקש הימני של העכבר תשמע אפקט קולי כלשהו

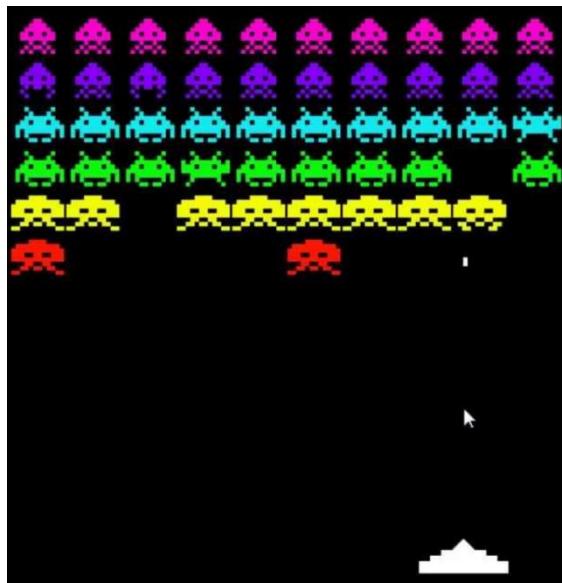
תהנו!

פרק 11 – OOP מתקדם – שילוב PyGame

מבוא

לאחר שלמדנו לכתב OOP בפייטון, נשלב את הידע שלמדנו ב-Game עם הידע שלנו ב-OOP ליצירת משחקים מורכבים. מדוע אנחנו זוקים ל-OOP בשביל לכתב משחקים בפייטון? הלא אנחנו יודעים כבר להעלות למסך כל צורה או Sprite שאחננו רוצים ולהזיז אותן?

נכון, אבל הבעיה היא שהשיטה שהשתמשנו בה עד כה טוביה בשביל להזיז צורה או שתיים. דמיינו שהמסך שלנו מלא ב-Sprites של Space Invaders שצריך להזיז אותן... נצטרך להגדיר כמה שירות Sprites, לכל אחד מהם לטען תמונה, לקבוע מיקום ולחנות את התוחזה שלהם על המ מסך. אחת הביעות הגדלות שלנו תהיה לתת שמות למשתנים. נניח שנרצה להגדיר מיקום על המ מסך עבור שני invaders. נזדקק לשתנה שישמר את המיקום x של invader1, משתנה למיקום y של invader1, משתנה למיקום x של invader2 ועוד אחד למיקום y של invader2 ... וזה עוד לפני שהתחלנו לדבר על משתנים ומהירותיהם שליהם (לא תמיד הם נעים באותה מהירות). לא נעים במיוחד לתכנת משחק כזה ☺



בນקודה זו מגע לעזרתנו OOP. הרעיון הכללי הוא שנגידר מחלוקת שכוללת את כל התוכנות של האובייקט שאחננו צריים, לדוגמה איך נראה invader space, מה המיקום שלו על המ מסך ומה מהירותו שלו, ובכל פעם שנרצה להוסיף invader חדש – פשוט ניצור instance נוסף של המחלוקת שלנו. קדימה, מתחילהם.

הגדרת class

בואו ניתן לכלב לב שלנו כמה כדורים לשחק בהם!

נפתח קובץ חדש, נקרא לו לדוגמה `py.shapes`. מאוחר יותר נעשה לו `import` לתוכנית הראשית שלנו.

נגיד בתוכו `class Ball` בשם `Ball`. חשוב מאד ש-`Ball` יירש מ-`Sprite`, כך שנוכל לרשט את כל המתודות המועלות של `Sprite`, מתודות אשר ישמשו אותנו בהמשך. כמובן, לא נשכח לעשות `import PyGame` בתחילת הקובץ.
כעת נכתוב את ה-`constructor` שלנו.

לשימושכם, כדור בייסבול עם רקע ורוד ☺



```
import pygame

PINK = (255, 20, 147)
MOVING_IMAGE = 'baseball.png'
HORIZONTAL_VELOCITY = 3
VERTICAL_VELOCITY = 5

class Ball(pygame.sprite.Sprite):

    def __init__(self, x, y):
        super(Ball, self).__init__()
        self.image = pygame.image.load(MOVING_IMAGE).convert()
        self.image.set_colorkey(PINK)
        self.rect = self.image.get_rect()
        self.rect.x = x
        self.rect.y = y
        self.__vx = HORIZONTAL_VELOCITY
        self.__vy = VERTICAL_VELOCITY
```

הדבר הראשון שאנו צריכים לבצע הוא להריץ את פונקציית האתחול של המחלקה ממנה `Ball` יירש.

השורות הבאות מוכרכות לנו כבר, טעינת הציור.

לאחר מכן מוגדר משתנה בשם `rect` ששייך לאובייקט שלנו. משתנה זה שומר את המיקום של הcador שלנו על המסך, והוא מאד חשוב בשבייל לדעת מה הcador שלנו עושה. לדוגמה, אילו אובייקטים אחרים הוא מתנגש. שימוש לב שלמרות שאנו יודעים מציירים על המסך עיגול, הרי ש-`rect` הוא מרובע. זיכרו שב уни התוכנית התמונה שלנו אינה עיגול, אלא מרובע שביקשנו להציג צבע אחד ממנו בתור "ש��וף". לתוכנית אין דרך לדעת שאחרי שהציגו את הצבע הורוד בתור ש��וף, נותרה צורה של עיגול. הקואורדינטות של `rect` הן הפינה השמאלית העליונה של הריבוע שלנו (כלומר הריבוע השלם, כולל החלקים השקופים), והגודל של `rect` הוא הגודל של הריבוע.

בשורות האחרונות נגדיר שה-`Sprite` שלנו מקבל מיקום תחומי בזמן היצירה שלו. השתמש בו מאוחר יותר לטובת הציור על המסך.

נותר לנו רק לקבוע את מהירות הcador. שימוש לב לכך שהמשתנים של מהירותם הם מוסתרים – מתחילהם ב-`__`. הינו יכולים לקבוע מהירות רנדומלית או לקבל את מהירות כפרמטר, נשאיר זאת להחלטתכם.

הוסף מתודות mutators-accessors שימושיות

בואו נזיז את הcador שלנו על המסך.

```
def update_v(self, vx, vy):
    self.__vx = vx
    self.__vy = vy

def update_loc(self):
    self.rect.x += self.__vx
    self.rect.y += self.__vy

def get_pos(self):
    return self.rect.x, self.rect.y

def get_v(self):
    return self.__vx, self.__vy
```

יצרנו מספר מתודות שתפקידן למצוא את מיקום הcador בכל פעם שנרצה. מיקום הcador החדש יוחשב בתור המיקום הקודם של הcador, בתוספת מהירות הcador בכל אחד מציירים.

זהו, התבנית של הכדור שלנו מוכנה וכעת נוכל למסור לכלבב שלנו כמה כדורים שנרצה.

הגדרת אובייקטים בתוכנית הראשית

בתוכנית הראשית נצטרך לעשות `import shapes` כדי שיצרנו. כדי שלא נצטרך לציין שהצורה נלקחה מהמודול `shapes` בכל פעם, נעשה `import shapes` באופן הבא:

```
from shapes import Ball
```

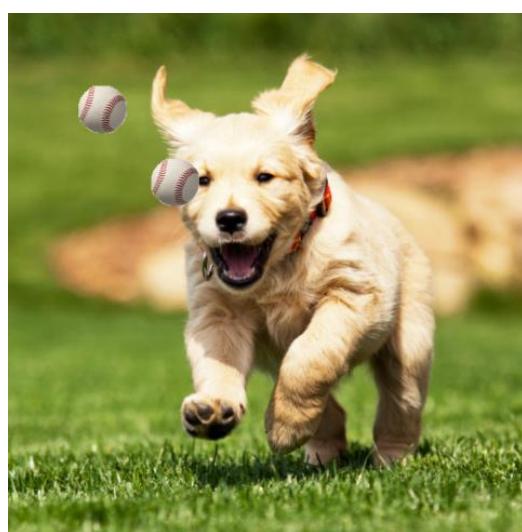
נגדיר שני כדורים ונציג אותם על המסך:

```
ball1 = Ball(100, 100)
ball2 = Ball(200, 200)
screen.blit((ball1.image, ball1.get_pos()))
screen.blit((ball2.image, ball2.get_pos()))
```

בשתי השורות הראשונות אנחנו יוצרים שני כדורים, לכל אחד מהם יש מיקום ההתחלתי שאנו מعتبرים לו. מיקום ההתחלתי זה יועתק שם ל-member `rect` של המחלקה `Ball`.

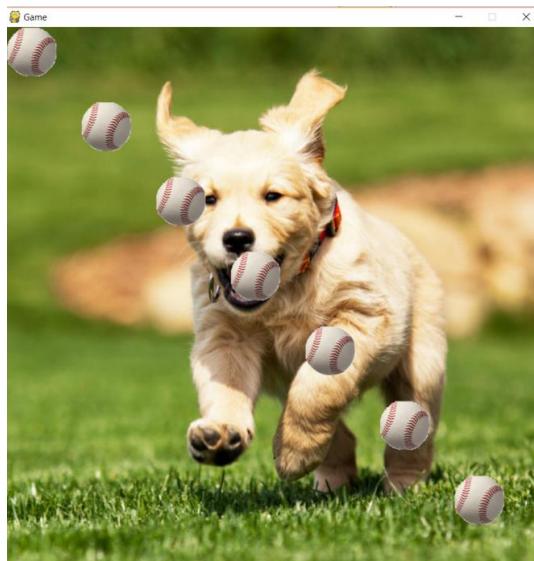
בשתי השורות האחרונות אנחנו מعتبرים למסך את הכדורים. המתודה `blit` מקבלת כזוכה את התמונה שאנו רוצים להעלות ואת המיקום. התמונה היא `ball.image` (עבור כל כדור), ועל המיקום אנחנו ניגשים בעזרת המתודה `get_pos` שיצרנו לשם כך.

נrix ונקבל כדורים במקומות המתאימים על המסך:



sprite.Group()

עד כה הצלחנו להציג כדורים על המסך כרצונו,מצוין. עם זאת, הדרך בה עשינו זאת מעט מסובבלת. דמייננו שאנחנו רוצים לצייר שבעה כדורים ולא שניים, כמו בדוגמה הבאה:



האם הגיוני שבשביל להציג כמהות גדולה של כדורים נצטרך לכתוב שוב ושוב שורות כמו 23 ו-24? לא. האם אין דרך יותר פשוטה להציג את כל ה כדורים שלנו, ולא לכתוב שורת קוד נפרדת לכל כדור? ...בודאי שיש.

```
balls_list = pygame.sprite.Group()
for i in range(NUMBER_OF_BALLS):
    ball = Ball(i*DISTANCE, i*DISTANCE)
    balls_list.add(ball)
balls_list.draw(screen)
```

כדי לעבור על כל ה כדורים בביטחון אחת אנחנו צריכים לולאת `for`, ולולאת ה-`for` צריכה לעבור על משהו שהוא `iterable`. (`sprite.Group` הוא `iterable`. אבל למה אנחנו מגדירים (`sprite.Group`) ולא סתם רשיימה ריקה, []?)
בגלל שלרשימה מסווג (`sprite.Group`) יש כל מיני מתודות שימושיות, שחווכות לנו עבודה. בקרוב נכיר שתוים מהן:

- מתודה שמדפסה למסך בביטחון את כל האובייקטים שברשימה

- מתודה שבזקמת אם יש התנגשויות בין אובייקטים (נמצאים באותו מקום)

בכל איטרציה של הלולאה אנחנו מגדרים כדור חדש ולאחר מכן מוסיפים אותו לרשימה שלנו באמצעות методת `add`.果然, יש לנו כתה רשימה הכוללת מספר כדורים. שימו לב לכך שהעובדת שאנו עושים משתמשים בכל איטרציה באותו משתנה בשם `ball` אינה גורמת למחיקת הקיימים הקודמים: כל משתנה בשם `ball` הוא מצביע על אובייקט מסווג `Ball`. בכל פעם שהתוכנית מגיעה לתחילת הלולאה, נוצר אובייקט חדש מסווג `Ball` והמשתנה `ball` מצביע עליו. לעומת זאת, רק המצביע המשתנה – ולא האובייקט עצמו. בנוסף לכך, בשורה הבאה המצביע על האובייקט שנוצר נשמר ברשימה, וכך אנחנו מסוגלים לגשת אליו גם אחרי ש-`ball` כבר מצביע על האובייקט הבא.

בפקודה התחתונה אנחנו קוראים למתחודה שחוותה לנו המון עבודה: `draw`. המתחודה הזה פועלת על (`Group`) sprite ומאפשרת להעביר למסך בבת אחת את כל האובייקטים שיש ברשימה. פעולה זו שකולה לרוץ בlolooat `for` ולהדפיס כל אובייקט למסך באופן נפרד.

יצירת אובייקטים חדשים

כעת נרצה שסכמות האובייקטים שלנו תהיה ניתנת לקביעה על ידי המשתמש. לדוגמה – שכל הקלקה על לחץ שמאלית בעבר תיצור כדור חדש על המסך.



נתבoso על הקוד שבודק אם היתה הקלקה על הלוחן השמאלי של העכבר ומה היה מיקום העכבר:

```
while not finish:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            finish = True
        # add a ball each time user clicks mouse
        elif event.type == pygame.MOUSEBUTTONDOWN \
                and event.button == LEFT:
            x, y = pygame.mouse.get_pos()
            ball = Ball(x, y)
            balls_list.add(ball)
        # update screen with balls
        screen.blit(img, (0, 0))
        balls_list.draw(screen)

    pygame.display.flip()
    clock.tick(REFRESH_RATE)
```

לאחר שמצאנו את מיקום העכבר אנחנו מגדירים ball חדש במקומ זה ומוסיפים אותו לרשימה. את יתר הפקודות כבר הכרנו לפני: הדפסת הרקע, הדפסת כל הכדורים, עדכון המספר וקייבת זמן עדכון התוכנית.

הצתת האובייקטים

אובייקטיםZZים יוצרים יותר מעניינים מאובייקטים קבועים במקום. בואו ניתן לכל אובייקט מהירות ההתחלתית אקראית (חשוב לעשות (import random :)

```
while not finish:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            finish = True
        # add a ball each time user clicks mouse
        elif event.type == pygame.MOUSEBUTTONDOWN \
                and event.button == LEFT:
            x, y = pygame.mouse.get_pos()
            ball = Ball(x, y)
            vx = random.randint(-MAX_VELOCITY, MAX_VELOCITY)
            vy = random.randint(-MAX_VELOCITY, MAX_VELOCITY)
            ball.update_v(vx, vy)
```

```
balls_list.add(ball)
```

בשורות המודגשות אנחנו מוגרילים מהירות התחלהית ומעדכנים את מהירות הcador בהתאם. בכל פעם שנרים את המתודה `() update_loc` ball.updates rect.x ו rect.y של הcador, כפי שקבענו בהתחלה:

```
for ball in balls_list:  
    ball.update_loc()
```

זהו, הcadors שלנו נעים על המסך ☺

תרגיל bounce

הבעיה במצב הנוכחי היא, שגם הcadors מגיעים לקצה המסך הם ממשיכים לנوع ויוצאים ממנו. אך הוסיףן קוד, שבודק אם הcador שלנו נוגע בשולי המסך ואם כן – מעדכן את מהירותו שלו כך שהcador "קופץ" חזרה. טיפ: קפיצת הcador חזרה מתבצעת על ידי היפוך מהירות האופקית שלו (אם פגע בקצה ימני או שמאל של המסך או היפוך מהירות האנכית שלו (אם פגע בקצה העליון או התיכון של המסך).

בדיקות התנגשויות

ברוב משחקי המחשב נרצה לדעת אם שני אובייקטים עולמים זה על זה. לדוגמה, אם ירייה נוגעת בדמות, או אם הפקמן שלנו נתפס על ידי שודן.

נרצה לשדרג את התוכנית שלנו כך שכל שני כדורים שמתנגשים זה בזה – "ימחקו".Cut נראה איך אפשר לזהות בקלות התנגשויות בין אובייקטים.

הmethodה spritecollide משמשת למטרה זו. המתוודה מקבלת אובייקט ורשימה של אובייקטים ומחזירה את כל האובייקטים מהרשימה שמתנגשים באובייקט הנבדק. בשורת הקוד הבאה אנו בודקים אם אובייקט מסוים בשם ball מתנגש ב כדורים שמוראים ב-`balls_list`:

```
balls_hit_list = pygame.sprite.spritecollide \
    (ball, balls_list, False)
```

כיצד המתוודה בודקת את ההתנגשויות? באמצעות ה-`rect` של כל אובייקט. כלומר, שני אובייקטים מתנגשים אם ה-`rect`'ים שלהם חופפים זה עם זה.

בנוסף, המתוודה מקבלת פרמטר בוליани בשם `kill` ("האם להרוג"). אם ערך הפרמטר הוא `True`, אז כל אובייקט ברשימה האובייקטים שמתנגש באובייקט הנבדק – "ימחק".

במקרה שלנו אנחנו לא רוצים למחוק את האובייקט המתנגש. מדוע? מיד נראה.

```
new_balls_list.empty()
for ball in balls_list:
    balls_hit_list = pygame.sprite.spritecollide \
        (ball, balls_list, False)
    if len(balls_hit_list) == 1:          # ball collides
                                         # only with itself
        new_balls_list.add(ball)

balls_list.empty()
for ball in new_balls_list:
    balls_list.add(ball)
```

אנו מגדירים לולה שעוברת על כל הcodors שבתוך רשימת codors שלנו. כמובן אנחנו בודקים בכמה codors בתוך הרשימה מתנגש כל codor בראשימה. ברגע שההתשובה תהיה תמיד לפחות 1, מכיוון שהcodor שבחרנו מהרשימה מתנגש עם עצמו (שכן ה-rect שלו חופף את זה של עצמו). זו גם הסיבה לכך שאנו לא מוחקים את codor המתנגש מהרשימה, כי אחרת נישאר עם רשימת codors ריקה.

לכן אנחנו בודקים האם codor שלנו התנגש רק בcodor אחד בראשימה (כלומר, עצמו בלבד). אם כן, זה אומר שהcodor צריך להשאר על המסך. לכן נשמר אותו בראשימת codors ה"שורדים" `list_balls_list`. שמו לב לכך שזו sprit.Group(). שזיהי מסוג () מסמך(list_balls_list) שבודקודה הראשונה, לפני הלולאה, אנחנו מרוקנים אותה מcodors, באמצעות המתוודה `empty`.

כל שנותר לנו לעשות הוא להחזיר את codors השורדים לרשימת `list_balls` לפני שנמשיך ונציג אותם על המסך. לשם כך אנחנו מרוקנים את `list_balls` ולאחר מכן אנחנו מעתיקים כל codor מתוך רשימת השורדים אל תוך `list_balls_list`.

שים לב לכךuai אפשר לכתוב פשוט `list_balls_list = new_balls_list`. מה יקרה לדעכם במקרה זה? תוכלו להעזר בדוגמה הבאה של שתי רשימות שמצוינות על אותו מקום בזיכרון.

```
In[12]: list1 = [1, 2, 3]
In[13]: list2 = list1
In[14]: list1.pop(0)
Out[14]: 1
In[15]: list1.pop(0)
Out[15]: 2
In[16]: list1.pop(0)
Out[16]: 3
In[17]: list2
Out[17]: []
```

לסיום, הנה הקוד המלא של התוכנית שבודקת התנגשויות של codors:

```
""" Pygame example program
Move sprites randomly and detect collisions
Author: Barak Gonen
"""

import pygame
import random
from shapes import Ball

WINDOW_WIDTH = 720
WINDOW_HEIGHT = 720
LEFT = 1
BACKGROUND_IMAGE = 'example.jpg'
REFRESH_RATE = 32
BALL_SIZE = 68
MAX_VELOCITY = 3

img = pygame.image.load(BACKGROUND_IMAGE)
pygame.init()
size = (WINDOW_WIDTH, WINDOW_HEIGHT)
screen = pygame.display.set_mode(size)
pygame.display.set_caption("Game")
screen.blit(img, (0, 0))
clock = pygame.time.Clock()

balls_list = pygame.sprite.Group()
new_balls_list = pygame.sprite.Group()
finish = False

while not finish:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            finish = True
        # add a ball each time user clicks mouse
        elif event.type == pygame.MOUSEBUTTONDOWN \
                and event.button == LEFT:
            x, y = pygame.mouse.get_pos()
            ball = Ball(x, y)
            vx = random.randint(-MAX_VELOCITY, MAX_VELOCITY)
            vy = random.randint(-MAX_VELOCITY, MAX_VELOCITY)
```

```

ball.update_v(vx, vy)
balls_list.add(ball)

# update balls locations, bounce from edges
for ball in balls_list:
    ball.update_loc()
    x, y = ball.get_pos()
    vx, vy = ball.get_v()
    if x + BALL_SIZE > WINDOW_WIDTH or x < 0:
        vx *= -1
    if y + BALL_SIZE > WINDOW_HEIGHT or y < 0:
        vy *= -1
    ball.update_v(vx, vy)

# find which balls collide and should be removed
new_balls_list.empty()
for ball in balls_list:
    balls_hit_list = pygame.sprite.spritecollide \
        (ball, balls_list, False)
    if len(balls_hit_list) == 1:           # ball collides
                                            # only with itself
        new_balls_list.add(ball)

balls_list.empty()
for ball in new_balls_list:
    balls_list.add(ball)

# update screen with surviving balls
screen.blit(img, (0, 0))
balls_list.draw(screen)

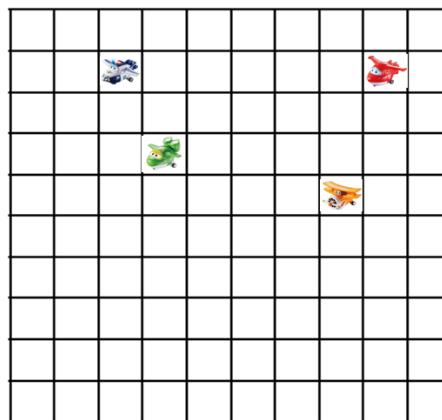
pygame.display.flip()
clock.tick(REFRESH_RATE)

pygame.quit()

```

תרגיל מסכם – פיקוח האוויר

אתם יושבים במגדל הפיקוח האווירי ואחראים למנוע התנגשות בין מטוסים במרחב האווירי שלכם. המרחב האווירי הוא מטריצה בגודל 10 × 10 משבצות (כל משבצת היא בגודל של כמה פיקסלים שתחליטה, לדוגמה 50 × 50 פיקסלים).



למרחב האווירי שלכם נכנסו 4 מטוסים מסווגים Plane שנעים בו אקראית. בכל תור, כל אחד מהמטוסים יכול לזרז לכל משבצת שצמודה למשבצת בה הוא נמצא (כולל אלכסון).

עליכם לכתוב אלגוריתם שמנהל את תנועת המטוסים: בכל תור על האלגוריתם לבחור אם לתרת למוטס המשיך לנעו אקראית או להזרות לו לאיזו משבצת צמודה הוא צריך לפנות.

ニיקוד: המטרה היא שהאלגוריתם שלכם ישלח כמה שפחות הוראות למטוסים. לכן, כל מטוס שנע למשבצת אקראית מעניק לכם נקודה. כל מטוס שנע למשבצת עקב פרודיה שקיבל מכם, מוריד לכם נקודה. המשחק נגמר כאשר שני מטוסים מתנגשים או לאחר 1000 תורות.

חישוב ניקוד לדוגמה:

ארבעת המטוסים שרדו 1000 תורות, כלומר נעו ביחד 4000 משבצות. בסך הכל המטוסים קיבלו 200 פקודות שינוי מיקום, כלומר הם נעו 3800 צעדים אקראיים (3800 נקודות). סך הכל הרווחתם $3800 - 200 = 3600$ נקודות.

הצלחתם לגרום למטוסים שלכם לשרוד?יפה מאוד! עכשו שחקו את המשחק עם 10 מטוסים ☺ האם האלגוריתם שלכם עדין עובד היטב? אם תוכלו לשפר אותו?

סיכום

בפרק זה למדנו לתכנן משחקי מחשב באמצעות PyGame. ראיינו איך יוצרים מסך משתמש, מגדרים תמונה רקע, מעלים על המסך תמונה קטנה, מקבלים קלט מהעכבר ומהמקלדת ומשמעיים צלילים.

לאחר מכן ראיינו כיצד השימוש ב-OOP מאפשר לנו להגדיר בקלות כל תמונה שנרצה בתור אובייקט ולהשתמש במתודות מתאימות כדי להזיז תמונות ולבזוק אם הן מתר觸שות.

בצד ההנאה מתכנות משחק מחשב, המטרה היא להתנסות בכתיבת קוד OOP פיתוני ולראות את ה יתרונות שלו על פני קוד שאינו OOP.

פרק 12 – מיליוןים

בפרק זה נכיר טיפוס חדש של פיתון – מילון (dictionary). מהו מילון? זה אוסף של זוגות, כאשר כל זוג מכיל מפתח וערך. מפתח נקרא key וערך נקרא value. לדוגמה, אפשר להגיד מילון שמכיל ציונים, כאשר המפתח הוא תעודת זהות. מגדרים מילון באמצעות סוגרים מסולסלים, כך:

```
students_grade = {}
```

כעת ניתן להזין לתוכו זוגות של מפתחות וערכים. לדוגמה:

```
students_grade['00001234'] = 85
students_grade['00003579'] = 95
students_grade['00002468'] = 65
```

בצד שמאל, בירוק, אלו המפתחות. במקרה זה בחרנו שהמפתחות יהיו מחרוזות, אך אפשר לבחור בטיפוס משתנים נוספים, לדוגמה `student`. הצד ימין, בכחול, אלו הערכים. במקרה זה בחרנו שהערכים יהיו מטיפוס `int`, אך אפשר להזין ערכים מכל טיפוס שנרצה.

אם נרצה לשולף את אחד הערכים, נקרא למילון עם המפתח המתאים. לדוגמה:

```
print(students_grade['00003579'])
```

דפס את הערך 95.

אם נרצה להגיד מילון שמראש יש לו ערכים התחלתיים, נוכל לעשות זאת באמצעות הסימן נקודות'ים:

```
students_grade = {'00001234': 85, '00003579': 95}
```

כיוון שמיון הוא אוסף של איברים, ניתן לעבור על האיברים שבו באמצעות לולאת `for`. לדוגמה:

```
for key in students_grade:
    print('Student ID: {}, grade: {}'.format(key, students_grade[key]))
```

הסבר: כאשר אנחנו מבצעים לולאת `for` על מילון אנחנו רצים על אוסף כל המפתחות שקיים במילון. עברו כל מפתח, אנחנו מדפיסים את המפתח ואת הערך הצמוד אליו.

Get, in, pop, keys, values

מה יקרה אם נחפש במילון מפתח שאינו קיים? לדוגמה, נבקש את הציון שיש לו לתעודת זהות שאינה קיימת במילון:

```
print(students_grade['00009999'])
```

נקבל exception:

```
Traceback (most recent call last):
  print students_grade['00009999']
KeyError: '00009999'
```

פקודת `get` מאפשרת לנו לבצע חיפוש "בטוח". כמובן, אם המפתח קיים יוחזר הערך הנוכחי, אך אם המפתח אינו קיים יוחזר הערך `None`. נזכיר, כי הערך `None` משמעו "כלום". זו מילה ש谋ורה. להלן דוגמאות לשימוש ב-`get` עם מפתח קיים ומפתח שאינו קיים. הריצו אותן וצפו בתוצאות שמתקבלות:

```
print(students_grade.get('00001234'))
print(students_grade.get('00009999'))
```

שים לב, ש-`get` דוחשת סוגרים עגולים, כיוון שהיא מותודה (של אובייקט מתייחס מילון).

אם אנחנו רק רוצים לדעת אם מפתח נמצא במילון, נוכל להשתמש באופרטור `in`. לדוגמה:

```
print('00001234' in students_grade)
```

נקבל `True` או `False`.

אם נרצה לקבל את רשימת כל המפתחות שהקיימים במילון, או של כל הערכים, נוכל להשתמש בMETHODS `keys` ו-`values`. לדוגמה:

```
print(students_grade.keys())
print(students_grade.values())
```

ונקבל:

```
[ '00003579', '00002468', '00001234' ]
[95, 65, 85]
```

תרגיל – קניות

- א. צרו מילון בשם `prices`. המפתח הוא שם המוצר והערך הוא מחיר המוצר. הוסיפו כ-5 מוצרים למלון. לדוגמה – בנות, 10 ₪. תפוחים, 8 ₪. לחם, 7 ₪. גבינה, 20 ₪. מיץ, 15 ₪.
- ב. צרו מילון בשם `shopping_cart`. המפתח הוא שם המוצר והערך הוא כמות המוצרים מסווג זה בעגלת הקניות. הוסיפו מספר מוצרים למלון. לדוגמה – בנות, 2 יחידות. לחם, 3 יחידות. גבינה, 1 יחידה.
- ג. הכניסו למשתנה בשם `total` את סכום הקניות בעגלה. בדקו שהסכום שמתקובל הוא נכון.
- ד. כתע, הזינו לתוך `shopping_cart` מוצר שאיןו קיימים בראשימת המחיר. עליכם לוודא שהתוכנית אינה עפה על שגיאה אלא מדפסה הודעה מתאימה.

תרגיל מסכם מיליוןיות – wordcount (credit: google classes)

הורידו את קובץ התרגיל מהلينק הבא:

<http://data.cyber.org.il/python/wordcount.zip>

בתרגיל אתם נדרשים לקרוא את הקובץ `alice.txt` ולהציג למסך את כמות המופעים השונים של כל מילה.

טיפ: השתמשו ב-`split` על מנת להפריד בין מילים.

למתקדים: טלו במקרים של מילים זרות, שנבדלות רק בסימני פיסוק שצמודים אליהן. לדוגמה – `'her.'` ו-`'her'` (עם נקודה בסוף). בדוגמה זו עליכם לזהות שמדובר באותה מילה ולספור יחד את כל המופעים.

מיליונים, מתחת למכסה המנווע (הרחבה)



בחלק זה נבין יותר לעומק איך עובד מילון. זה אינו חלק מעשי, אבל תלמידים סקרנים ימצאו בו עניין.

דמיינו שאתם עובדים בבית מלון בשם 'Hash Gardens'. זהו מלון מאד מיוחד מכיוון שבשל המלון הוא מתמטיקיי מטורף שלא מוכן לשומר שום מידע על האורחים שלהם, לא על מחשב ולא על פיסת ניר. لكن במלון אין רשימה של מספרי החדרים של כל האורחים. לעומת זאת, אין לכם דרך לדעת באיזה מספר חדר מתגורר כל אורח. בוקר אחד הטלפון בדלתך מצלצל. מישהו מבקש לשוחח עם מר smith john. איך אתם מעבירים את השיחה בחדר שלו?

כל הנראה, תצטרכו לחפש את מר סמית' בכל החדרים. לא קל, במיוחד אם המלון שלכם גדול מאוד. תבזבזו הרבה זמן באיתור מספר החדר של כל אורח שמתקשרים אליו.

אתם פונים לבשל המלון ומתחננים שייתן לכם לשומר במקום כלשהו את מספרי החדרים של האורחים, אבל הוא פשוט מחייב ונותן לכם עצה מתמטית. מעכשיו, כל אורח שמגיע למלון מקבל מהם מספר חדר שיש קשר מתמטי בין לבין שם האורח. כך, למרות שלא רשום לכם בשום מקום איזה חדר קיבל כל אורח, כאשר מתקשרים וambilשים לשוחח עם אורח כלשהו, אתם יכולים לחשב במהירות את מספר החדר המבוקש.

להלן דוגמה לפונקציה שמחשבת מספר חדר על פי שם האורח: נניח שבמלון יש 113 חדרים, אשר ממושפרים מ-0 ועד 112 (כפי שניתן לצפות מתמטיקיי, כמות החדרים במלון היא מספר ראשוני). כל אורח שמגיע למלון מקבל חדר שמספרו מחושב לפי הנוסחה הבאה: נכפול את ערכי ה-ascii של כל התווים בשם האורח ולתוצאה נבעץ מודולו 113. באופן זה נהפוך כל מחרוזת למספר בין 0 ל-112. התוצאה תהיה מספר החדר.

להלן קוד של פונקציה שמבצעת את החישוב הנ"ל:

ROOMS = 113

```

def find_room_number(guest_name):
    tmp = ord(guest_name[0])
    for letter in guest_name[1:]:
        tmp *= ord(letter)
    return tmp % ROOMS

room = find_room_number("John Smith")
print(room)

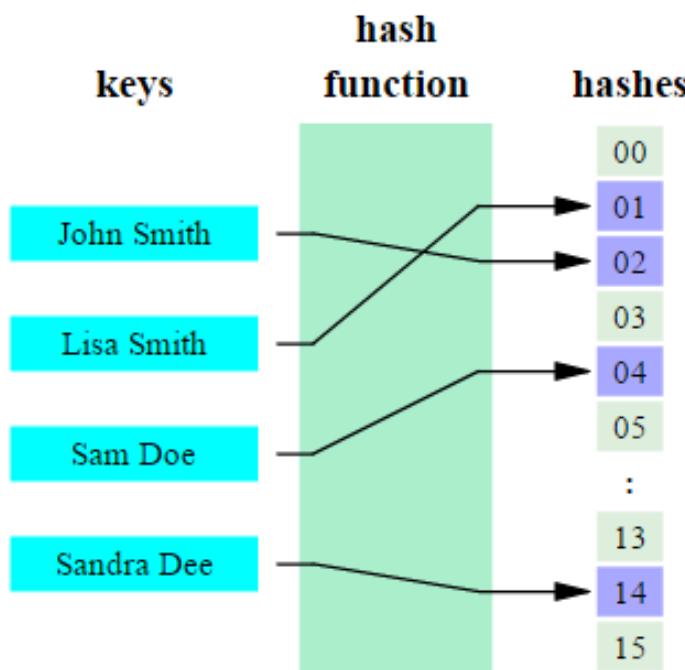
```

כאשר מגיע אלינו האורח John Smith, הפונקציה תחשב את מספר החדר שלו – 84. אם מישהו יתקשר למר סמית', יוכל לחשב מחדש את מספר החדר שלו.

כפי שאלוי ניחשתם, הפונקציה `find_room_number` היא בעצם פונקציית `hash`. מה מיוחד בה?

- א. היא מתאימה לכל מחרוזת מספר בתחום קבוע מראש
- ב. למחרוזות זהות מתקבל `hash` זהה. לא יכול להיות שאורחיהם עם שמות זהים יקבלו חדרים שונים
- ג. אי אפשר (או לפחות, קשה מאוד) לשחזר את שם האורח מתוך `hash` של השם שלו. אם אנחנו יודעים שאורח נמצא בחדר 84, אי אפשר לגלוות שמו הוא `smiths john`. במלחמות אחרות, זהה פונקציה חד-значונית.

אך יש בעיה אחת עם פונקציית `hash` שלנו: יכול להיות מצב שבו שני אורחים בעלי שמות שונים יקבלו את אותו חדר, מה שייצור התנגשות (collision). לכן, פונקציית `hash` טובה היא צזו שהטبيعي להתנגשות בה הוא מאוד קטן.



המחשה של פונקציית `hash`. מקור: ויקיפדיה

נחזיר כעט נתונים המלונאות והתיירות אל עולם הפייטון. בפני מי שבנה את שפת פייטון עדמה אותה בעיה של העובד במלון: איך מוצאים מהר את העורק שצמוד למפתח מסוים? כפי שראינו, חיפוש על פיינטן כל המפתחות הוא יקר בזמן, במיוחד אם יש במלון שלטן הרבה מפתחות. הפתרון שמלוניים עושים בו שימוש הוא כMOVEDן פונקציית `.hash`.

מילון בפיטון מתחילה בתור טבלה ריקה. כאשר אנחנו מכנים למילון מפתח וערך, פיטון מחשב את ה-hash של המפתח ומתקבל מספר. המספר זה יהיה האינדקס בטבלה שבו פיטון ישמר את המפתח ואת הערך הצמוד אליו. לדוגמה, אם ה-hash היה 5, האינדקס בטבלה יהיה 5.

מה קורה כאשר אנחנו רוצים לשולף ערך מהמילון? אנחנו מעבירים לפיטון את המפתח. על בסיס המפתח פיטון מחשב את ה-hash וניגש אל המקום בטבלה שזה המספר שלו. באופן זה, חיפוש של ערך בתור מילון לוקח זמן קצר מאד ומעט אינו תלוי בכמות המפתחות שיש במילון.

עלול לקרות מצב שבו לשני מפתחות יש את אותו hash. ראוי לציין שפיטון משתמש בפונקציית hash "טובה", כזו שודאגת שהסיכוי שנייה מפתחות יהיו בעלי אותו hash הוא קטן. במצב שנייה מפתחות יש אותו hash קוראים collision ויש דרך לטפל בו. הערך הראשון מוכנס לטבלה כרגע. כאשר פיטון רוצה להכניס את הערך השני לאותו מקום בטבלה, הוא בודק ומגלה שבטבלה כבר יש מפתח במקום המבוקש. אם המפתח החדש שונה מהמפתח שקיים בטבלה, סימן שקרה collision. המפתח החדש ישמר במקום הפניו הבא. לדוגמה, אם המקום באינדקס 5 בטבלה כבר תפוס, פיטון יכול לנסוטו לשומר באינדקס 6. אם נרצה לשולף מהמילון את הערך של המפתח השני, פיטון יגש קודם כל לאינדקס 5. שם הוא יגלה שהמפתח אינו מתאים ולכן הוא י Mish'יר לאינדקס הבא עד שיימצא את המפתח המבוקש.

סוגי מפתחות

ראינו שכל צמד שמוכנס למילון מורכב ממספר מפתח וערך. בתור מפתח השתמשנו במשתנים מטיפוס מחוזת. האם כל טיפוס משתנה יכול להיות מפתח?

כדי להסביר על זה, נזכיר בדיון שלנו בנושא mutable ו-immutable. אובייקט מסווג mutable הוא אובייקט שנitin' לשנות, כדוגמת רשימה. כשלמדנו על רשימות ראיינו שאנו יכולים להגיד רשימה, לשנות את אחד האיברים בה ולאחר השינוי ה-*id* של הרשימה ישאר ללא שינוי. הערך של אחד האיברים השתנה אך זו עדין אותה רשימה. לעומת זאת, אובייקטים מסווג immutable לא ניתנים לשינוי. כלומר, הדרך היחידה לשנות אותם היא להגיד אותם מחדש ואז כMOVן משתנה ה-*id* שלהם.

הבה נניח שהיינו יכולים להשתמש במשתנה מסווג mutable בתור מפתח. לצורך הדיון, ניקח שתי רשימות, שנזכיר שהין Seable:

```
a = ['apple']
```

```
b = ['banana']
```

שימוש לב שאלן רשימות, אשר מכילות כל אחת איבר אחד מטיפוס מחוזת.

כעת, נגידר מילון:

```
fruits = {a:1, b:2}
```

פייתון חישב את ה-hash של a ושל b והכניס כל אחד מהם למיקום המתאים לו בטבלה.

כעת אנחנו משים את ערכו של d:

```
b[0] = 'apple'
```

אנחנו יכולים לעשות זאת כיוון שמדובר ברשימה, שהיא `mutable`.

מה יקרה אם כעת ננסה לשלווף את d מהמילון?

פייתון יחשב את ה-hash של b ויקבל ערך מסוים, אך הערך לא יתאים למיקום ש-b שומר בו. אם לא די בכך, ה-hash שיתקבל דואקן יתאים למיקום אחר בטבלה – המיקום אשר שומר את a ושיש לו מפתח זהה – ולכן החיפוש יחזיר את הערך השגוי "1".

סיכום

בפרק זה למדנו אודות מבנה נתונים מיוחד. מילון מאפשר לנו לשמר נתונים ולמצוא אותם במקרים רבים, תוך שימוש במפתחות ובפונקציה מתימנית – hash. למדנו ליצור מילון, להזין לתוכו ערכים ומפתחות ולהפץ ערכים במילון באמצעות מפתח. בהמשך סקרנו מספר תכונות של פונקציית hash וראינו כיצד תכונות אלו עוזרות לה להיות שימושית עבור מילוניים.

Magic Functions, List Comprehensions – 13

בפרק זה נלמד לנצל יכולות של פיתון על מנת לכתוב קוד בצורה יותר קצרה ו"פיינונית".

List Comprehensions

נניח שאנו רוצים ליצור רשימה של איברים שיש להם חוקיות מסוימת. לדוגמה, חזקיות של מספרים. רשימה כזו אי אפשר ליצור באמצעות range, מכיוון שהקפיצה בין מספרים אינה קבועה. מצד שני, אנחנו לא רוצים לכתוב באופן ידני את המספרים ואנו יכולים לנצל את העבודה שישנה חוקיות למספרים.

דרך אחת ליצור את הרשימה שלנו היא באמצעות לולאת for, לדוגמה, כך:

```
squares = []
for i in range(100):
    squares.append(i**2)
```

כך יוצרים רשימה שמכילה את האיברים $[0, 2, 4, \dots, 99^2]$

כעת נלמד syntax אלטרנטיבי, שיוצר את אותה הרשימה לעיל בשורה אחת בלבד. צורת כתיבת זו נוהגת מאוד בשפת פיתון, שכן לאחר שמתרגלים אליה היא נוחה מאוד לקרוא ולכתוב. להsyntax זה קוראים list comprehensions

באמצעות list comprehensions אפשר ליצור את הרשימה לעיל בצורה הבאה:

```
squares = [i**2 for i in range(100)]
```

שימוש לב לשינוי בסדר הפקודות. בעוד את לולאת ה-for קוראים "עבור כל אחד מהאיברים ב-(100) תבצע העלאה בריבוע", את הביטוי שבסוגרים צריך לקרוא "תבצע את 2^{*2} עבור כל אחד מהאיברים ב-(100)".

אנחנו יכולים להוציא תנאים. נגד, שמו רק את הריבועים של איברים זוגיים.שוב, נציג קודם את צורת הכתיבה עם לולאת for:

```
squares = []
for i in range(100):
    if i % 2 == 0:
        squares.append(i**2)
```

והכתב המקוצר, באמצעות list comprehensions:

```
squares = [i**2 for i in range(100) if i % 2 == 0]
```

לא תמיד קל לקרוא קוד שכתוב בשורה אחת ארוכה. למען קלות הקריאה, אפשר לחלק את השורה הנ"ל למספר שורות:

```
squares = [i**2
           for i in range(100)
           if i % 2 == 0]
```

בнтאים כתבו לולאות פשוטות, שניתן היה לכתוב ב-4-3 שורות ללא list comprehensions. לעיתים, נרצה לבצע דברים מורכבים יותר באמצעות list comprehensions.

לצורך התרגיל, נרצה ליצור רשימה של כל המספרים הראשוניים. כדי לעשות זאת, ניצור קודם כל רשימה של המספרים הלא ראשוניים. לאחר מכן, כל מספר שלא נמצא בראשימת הלא ראשוניים – ייחסב מספר ראשוני. להלן קוד שמבצע זאת באמצעות לולאות for:

```
import math
LIMIT = 100

root = int(math.sqrt(LIMIT))
non_primes = []
for i in range(2, root):
    for j in range(2*i, LIMIT, i):
        non_primes.append(j)
non_primes.sort()

primes = []
for i in range(LIMIT):
    if not i in non_primes:
        primes.append(i)
```

הסבר: הקבוע LIMIT מגדיר מה הגבול העליון של המספרים הראשוניים שנחנו מחופשים. בדוגמה זו, נרצה לקבל את הראשוניים עד 100. כדי למצוא את כל המספרים הלא ראשוניים עד 100 אין צורך לבדוק את המכפלות של כל המספרים, אלא רק עד שורש 100. לכן, מוגדר המשתנה root אשר ימשח את הלולאה החיצונית.

בלולאה הפנימית, מוצאים את כל המכפלות של 2 * LIMIT. לדוגמה, נניח ש-i הוא 5, אינדקס ההתחלה הוא 2^*5 ומתקדמים בקפיצות של 5. כל מכפלה צזו מתווספת לרשימה בשורה הבאה. מיון הרשימה מחוץ ללולאה מתרחש רק לשם היופי.

בollowה הבה אנחנו יוצרים רשימה חדשה, לתוכה מוכנסים כל המספרים עד LIMIT אשר אינם מופיעים ברשימה של הלא ראשוניים.

כעת, אותו קוד כפ' שהוא כתוב עם list comprehensions:

```
import math
LIMIT = 100

root = int(math.sqrt(LIMIT))
non_primes = [j
    for i in range(2, root)
    for j in range(2*i, LIMIT, i)]

primes = [i
    for i in range(LIMIT)
    if not i in non_primes]
```



תרגיל – קיצוצים (קדיט: עומר רוזנבוים, שי סדובסקי)

כתבו את הפונקציה avg, אשר מקבלת שתי רשימות של מספרים ומחזירה את ההפרש הממוצע ביניהן. לדוגמה, עבור הרשימות [1,2,3,4],[1,1,1,1] יוחזר 1.5. נשמע פשוט? אז זהו, שוקב קיצוצים בתקציב אנחנו נאלצים להתייעל ולכתוב את הפונקציה בשורה אחת בלבד. זה אומר ש-:

- אסור לרדת שורה

- אסור להשתמש בתוו ":"

אפשר להניח שתwo הרשימות מכילות מספרים, שהן בעלות אותו אורך ושחן אין ריקות.



תרגיל – אנטיבי

כתבו את הפונקציה anti_b1 אשר מקבלת מחזורת ומחזירה אותה ללא מופיעים של התו 'b'. זיכרו – הקיצוצים עדין נמשכים! ☺

לטיכום, מהוות דרך חזקה לכתוב בצורה קצרה מאוד קוד מורכב. הן יכולות להכיל גם תנאים מורכבים, כאשר מתרגלים אליהן – עוזרות לנו לכתוב קוד בצורה קריאה,יפה ומהירה.

Lambda

פונקציות lambda הן פונקציות לשימוש חד פעמי – אנחנו רוצים לעשות פעולה לא מאוד מורכבת ונראה לנו מיותר להגיד פונקציה במיוחד בשביל

זה.

שימוש לב לאופן הכתיבה הבא:

```
f = lambda x: 2*x+1
```

איך קוראים את זה? "הפונקציה f מקבלת כפרמטר x ומחזיר אותה להגדרת הפונקציה הבאה:

```
def f(x):
    return 2*x+1
```

אם נרצה לקרוא ל-f נעשה זאת בדיק כמו שקוראים לפונקציה רגילה. לדוגמה (5) f יחזיר 11.

אפשר גם להגיד פונקציית lambda שמקבלת כמה פרמטרים. לדוגמה:

```
f = lambda x, y: x*y + x + y
```

לדוגמה, (2, 3) f יחזיר 11.

מתי נשמש בפונקציית lambda? לדוגמה, כאשר אנחנו רוצים להעביר key לפונקציית `os.path.join`, אפשר להגיד את key בתוך פונקציית lambda. דוגמה נוספת אשר נלמד בעtid – כאשר נלמד רשותות וכותבו קוד לשינון מידע שעובר ברשות, נשמש בפונקציית lambda.

תרגילים



- כתבו פונקציה lambda שמקבלת מספר ומחזירה את המספר פלוס 2.
- כתבו פונקציה lambda שמחזירה את הערך המוחלט של מספר, והשתמשו בפונקציה זו על מנת למיין באמצעות sort רשימה של מספרים, לדוגמה: [3, -1, -8, 5, -6, 2].

הfonקציות הבאות שנלמד, `map`, `reduce` ו-`filter`, הוחלפו למשה על ידי list comprehensions אך חשוב להכיר אותן כיוון她们 נמצאות לעיתים בקוד פיתון.

Map

fonקציית `map` מקבלת פונקציה ורשימה. נוצרת רשימה חדשה, שהיא התוצאה של הרצת הפונקציה על כל אחד מאיברי הרשימה המקורי.

במילים אחרות, לכתוב `new_list = map(func, old_list)` זה כמו לכתוב:

```
new_list = []
for element in old_list:
    new_list.append(func(element))
```

כפי שבטח ניחשתם, את `func` אין צורך להגיד ממש, אלא ניתן להכניס lambda כרצוננו.

לדוגמה, ניקח רשימת מספרים ונרצה לכפול כל מספר פי 2 ולהוסיף 1:

```
old_list = [3, 6, 1, 7, 5]
print(set(map(lambda x: 2*x+1, old_list)))
```

ונקבל {3, 7, 11, 13, 15}. שימושו לב שלצורך ההדפסה הפקנו את תוצאת המיפוי לאובייקט מטיפוס `set`, שנית להדפסה.

אנחנו לא חייבים להסתפק ברשימה אחת. אפשר לכתוב פונקציה lambda שמקבלת יותר מפרמטר אחד ולהעביר ל-`map` כמה מתאימה של רשימות. לדוגמה, ישן שתי רשימות ואנחנו מעוניינים לכפול אותן זו בזו (מה שנקרא "מכפלה וקטורית" במתמטיקה):

```
list1 = [1, 2, -5, 6]
list2 = [2, -1, 3, 4]

print(set(map(lambda x, y: x*y, list1, list2)))
```

ונקבל {2, 2, 15, 24}

תרגיל – אנחנו על המפה (קדיט: עומר רוזנבוים, שי סדובסקי)



כתבו פונקציה שמקבלת מחרוזת ומחזירה מחרוזת חדשה, אשר כל האותיות בה מוכפלות. לדוגמה, עבור המחרוזת 'Cyber' יתקבל 'CCyybbeerr'. האתגר הוא כותב את הפונקציה בשורה אחת ☺ נסו את הפונקציה עם מגוון קלטים ובזקן שההתוצאה נכונה.

Filter

פונקציה זו נועדה לסנן (to filter) רק איברים רלבנטיים מתוך רשימה קיימת. לשם כך, הפונקציה `filter` מקבלת פונקציה ורשימה, בודקת על כל אחד מאיברי הרשימה האם הפונקציה מחזירה עליו `True` ומחזירה רשימה רק של האיברים שהחזירו `True`. כך, ניתן –

`(new_list = filter(func, old_list))` מבצע פעולה זהה לקוד הבא:

```
new_list = []
for element in old_list:
    if func(element) is True:
        new_list.append(func(element))
```

לדוגמה, נרצה לחתך רשימה וליצור ממנה רשימה חדשה רק של המספרים הזוגיים:

```
old_list = [2, 3, 4, 7, 8, 10]
print(set(filter(lambda x: x % 2 == 0, old_list)))
```

הסבר: קודם כל אנחנו יוצרים פונקציית `lambda`, שמחזירה את הערך הבוליאני של הביטוי `x % 2 == 0`. לאחר מכן אנחנו מכניסים לתוך `filter` את הפונקציה הנ"ל יחד עם רשימה.

תרגיל – את מסגנת אוטי (קדיט: עומר רוזנבוים, שי סדובסקי)



צרו פונקציה שמקבלת מספר ומחזירה רשימה של כל המספרים הקטנים ממנו אשר מתחלקים ב-3. לדוגמה, עבור המספר 10 תחזיר הרשימה `[3, 6, 9]`. כמובן, אוריך הפונקציה חייב להיות שורה אחת בלבד ☺

Reduce

הערה: הפקציה זו הייתה קיימת בפייטון 2, אך אינה קיימת יותר בפייטון 3. מומלץ להשתמש בלולאת `for` במקום.

הפקציה `reduce` מקבלת פונקציה ורשימה ומוחירה ערך יחיד. הערך זהה הוא התוצאה של ביצוע הפקציה על איברי הרשימה שוב ושוב עד שנותר ערך יחיד. אפשר לחשב על `reduce` בתור `map` שימושי להתבצע כל עוד אוורך הרשימה החדש שנוצרת גדול מ-1.

לדוגמה, יש לנו רשימה ואנחנו רוצים לחשב את הסכום של האיברים שלה:

```
old_list = [2, 3, 4, 7, 8, 10]
print(reduce(lambda x, y: x+y, old_list))
```

אפשר לדמיין שבכל שלב נוצרת רשימה חדשה, שבה האיבר הראשון הוא סכום שני האיברים בראשימה המקורי:

- [5, 4, 7, 8, 10]
- [9, 7, 8, 10]
- [16, 8, 10]
- [24, 10]
- [34]

כאשר הגיעו לרשימה באורך 1 החישוב יעצר ויוחזר הערך שנשאר בה.

סיכום

בפרק זה למדנו כיצד לחתך קוד, שבדרכן כלל היינו כותבים אותו באמצעות לולאת `for`, ולכתוב אותו בצורה קצרה ופושטה באמצעות `list comprehensions` או באמצעות פונקציות הקסם של פייטון: `lambda`, `map`, `filter` ו-`reduce`. חלק מfonקציות אלו יהיו שימושיות בלימודי רשותת. המשך לימוד מוצלח!