

תכנות בשפה פיתון

ברק גonen

המרכז לחינוך סייבר
CYBER EDUCATION CENTER

תכנות בשפת פיתון

Python Programming / Barak Gonen

גרסה 2.01 ספטמבר 2020

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

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

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

תוכן עניינים

8	הקדמה והתקנות נדרשות
8	התקנות נדרשות
18	アイיקונים
18	תודות
19	תוכן עניינים מצגות פיתון
20	פרק 1 – מבוא ללימוד פיתון
20	מהו שפת סקריפטים?
23	עבודה דרך command line
24	מספריים בסיסים שונים
26	Help
28	הסימן _ (קו תחתון)
28	הרצה תוכניות פיתון דרך ה-command line
29	סיכום
30	פרק 2 – סביבת עבודה PyCharm
30	פתיחת קובץ פיתון
32	קובץ הפייתון הראשון שלנו
33	סדר ההרצה של פקודות בסקריפט פיתון
35	התרעה על שגיאות
37	הרצה הסקריפט ומסך המשתמש
39	דיבוג עם PyCharm
42	העברה פרמטרים לסקריפט
43	סיכום
44	פרק 3 – משתנים, תנאים ולולאות
44	סוגי משתנים בפייתון
46	תנאים
47	תנאים מורכבים

49	שימוש ב-is
49	בלוק
51	תנאי else, elif
52	לולאת while
54	לולאות for
56	pass
58	פרק 4 – מחרוזות
58	הגדרת מחרוזת
59	ביצוע print-format
60	חיתוך מחרוזת – string slicing
62	פוקודות על מחרוזות
63	dir, help
64	צירופי תווים מיוחדים ו-string
65	קבלת קלט מהמשתמש – raw_input
69	פרק 5 – פונקציות
69	כתיבה פונקציה בפייתון
72	return
73	None
73	scope של משתנים
79	פייתון מתחת למכסה המנווע (הרחבה)
81id, is
83	העברת פרמטרים לפונקציה
85	סיכום
86	פרק 6 – List, Tuple
86	הגדרת List
88	Mutable, immutable
90	פעולות על רשימות

90	in
90	append
91	pop
91	sort
93	split
94	join
95	Tuple
97	סיכום
98	פרק 7 – כתיבת קוד נכונה
99	PEP8
103	חלוקת קוד לפונקציות
105	פתרון מודרך
110	assert
114	סיכום
116	פרק 8 – קבצים ופרמטרים לסקריפטים
116	פתיחת קובץ
117	קריאה מקובץ
118	כתיבה לקובץ
118	סגירת קובץ
120	קבלת פרמטרים לתוכנית
125	סיכום
126	פרק 9 – Exceptions
127	try, except
130	סוגים של Exceptions
131	finally
134	with
141	פרק 10 – תכנות מונחה עצמים – OOP

141	מבוא – למה OOP ?
142	אובייקט – object
143	מחלקה – class
145	כתיבת class בסיסי
146	<code>__init__</code>
147	הוספה מתודות
147	Members
148	יצירת אובייקט
149	כתיבת class משופר
150	יצירת "מוסתרים" members
152	שימוש ב-accessor ו- mutator
154	יצירת מודולים ושימוש ב-import
158	אתחול של פרמטרים
158	קביעת ערך בירית מודול
159	<code>__str__</code>
159	<code>__repr__</code>
160	יצירת אובייקטים מרובים
162	ירושה – inheritance
166	פולימורפיזם
167	<code>isinstance</code>
171	פרק 11 – OOP מתקדם (תכנות משחקים באמצעות PyGame)
172	כתיבת שלד של PyGame
173	שינוי רקע
178	הוספה צורות
181	תזוזה של גרפייקה
184	ציור Sprite
186	קבלת קלט מהעכבר

189	קבלת קלט מהמקלדת
189	הש舅ת צלילים.....
191	OOP מתקדם – שילוב PyGame
191	מברא.....
192	הגדרת class.....
193	הוספה מתודות mutators-ו accessors שימושיות.....
193	הגדרת אובייקטים בחוכנות הראשית
195	sprite.Group()
196	יצירת אובייקטים חדשים.....
197	הזאת האובייקטים
198	בדיקות התנגשויות.....
203	סיכום
204	פרק 12 – מיליוןים.....
207	מיליוןים, מתחת למיכסה המנווע (הרחבה)
209	סוגי מפתחות
210	סיכום
211	Magic Functions, List Comprehensions – 13
211	List Comprehensions
214	Lambda
215	Map
216	Filter
217	Reduce
217	סיכום

הקדמה והתקנות נדרשות

ברוכים הבאים לשפט פיתון! אם אתם קוראים ספר זה כנראה שאתם עושים את צעדיכם הראשונים בмагמת הганת סייבר. מדוע לומדים דואק פיתון? שפט פיתון נבחרה ללימוד כיון שהיא נמצאת בשימוש רחב בתעשייה, באקדמיה וגם בקרב הichידות 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 ולכן יש ללמידה אוטם תחיליה. לעומת זאת, התקנות מונחה עצמים ניתנת 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 לכוסות את כל הנושאים בשפט פיתון, אלא להתמקדב בנושאים 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. הויתור על ההסברים המפורטים אודות מודולים הוא ראשית מכיוון שקשה מאוד להקיף את כל המודולים החשובים בספר ללמידה, 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 הן ללמידה פיתון והן ללמידה רשות מחשבים. גרסת הפיתון של התקנה היא 3.8.

שימוש לב:

התוכנות הן עבור מערכת הפעלה Windows 10, עבור מערכות הפעלה אחרות ישן גרסאות ספציפיות של התקנות ויש להוריד אותן באופן עצמאי. יתר התקנה צפוי להיות דומה.

התוכנות הן (לפי סדר ההתקנה):

1. **פייטון, גרסה 3.8.0** – נדרש ללימוד פייטון

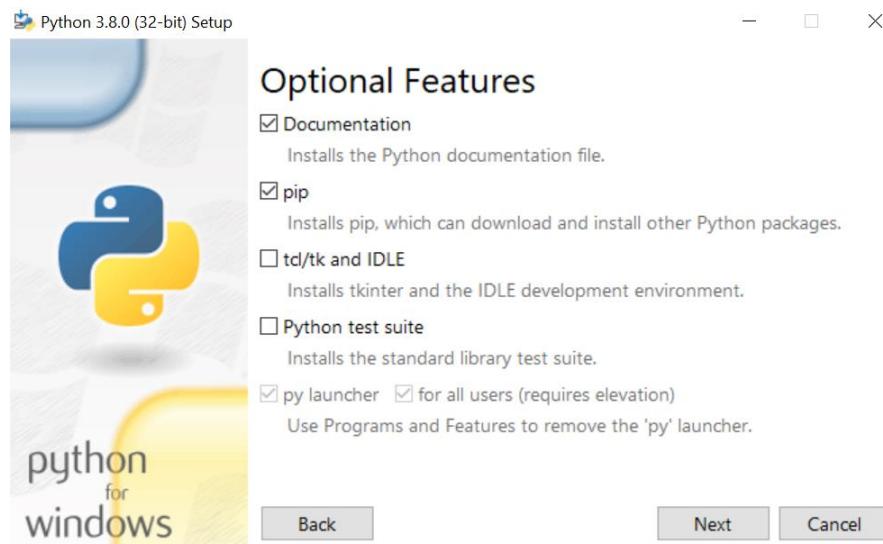
א. התקנת פיתון

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

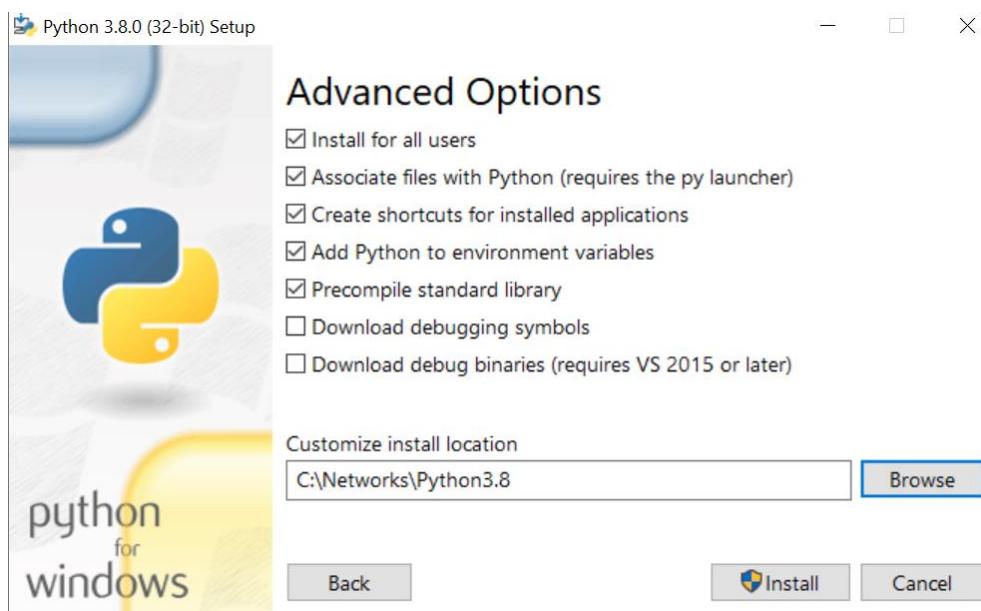


בחרו באפשרות "customize installation".

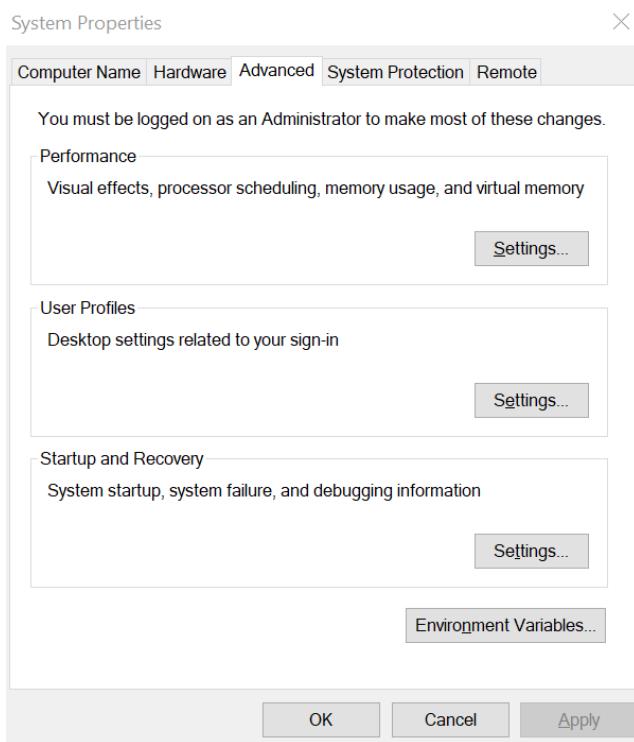
בחרו באפשרויות הבאות:



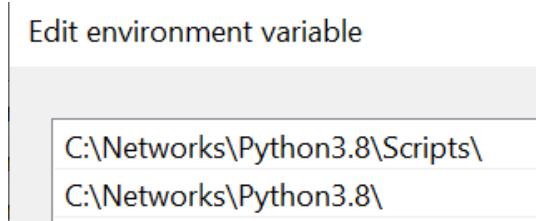
ובמסך הבא בחרו את האפשרויות המסומנות,
המייקם של ההורדה אינו קרייטי, פה הוא מסומן בתווים networks



כעת בידקו שפייתון מוגדר בתוך PATH של משתני הסביבה שלכם. במסך החיפוש של windows הקלידו env
.Environment Variables. הקliquו על הלחץן Edit The System Environment Variables



וודאו שבתוך המשתנה PATH יש את שתי הכנסיות הבאות. אם הן אינן, הוסיפו אותן ידנית (בהתנה שהתקנותם את פירוטן בטור `C:\networks\python3.8\Scripts\`) המיקום לא משתנה



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

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


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



רקע היסטורי, או מידע העשרתי אחר



הפניה לסרטון

תודות

ספר זה לא יהיה נכתב אל מולו 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 קרדיט בצד התרגילים.

ברק גון

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>

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: מומלץ 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 פייתון

מהי שפת סקריפטים?

ברוכים הבאים לשפת פייתון היא שפה שימושית וקלת לשימוש. לאחר שתשלטו בסיס השפה תוכל לכתוב תוכניות מסוימות בקלות יחסית. לדוגמה, במספר שורות קוד תוכל לגזור לשני מחשבים לשЛОח הודעות אחד לשני. לפחות ממאתיים שורות קוד תוכל לפתח משחק מחשב, עם גרפיקה וצללים.

שפת פייתון 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 להפוך 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 לעבור שני שלבים לפני שהמעבד יוכל להריץ אותו: השלב הראשון נקרא קומpile'ז'ה 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 לתוכנת בה 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 אסמבילר. האסambilר 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 מתורגמות לשפת מכונה.

את השלבים הבאים נוכל 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

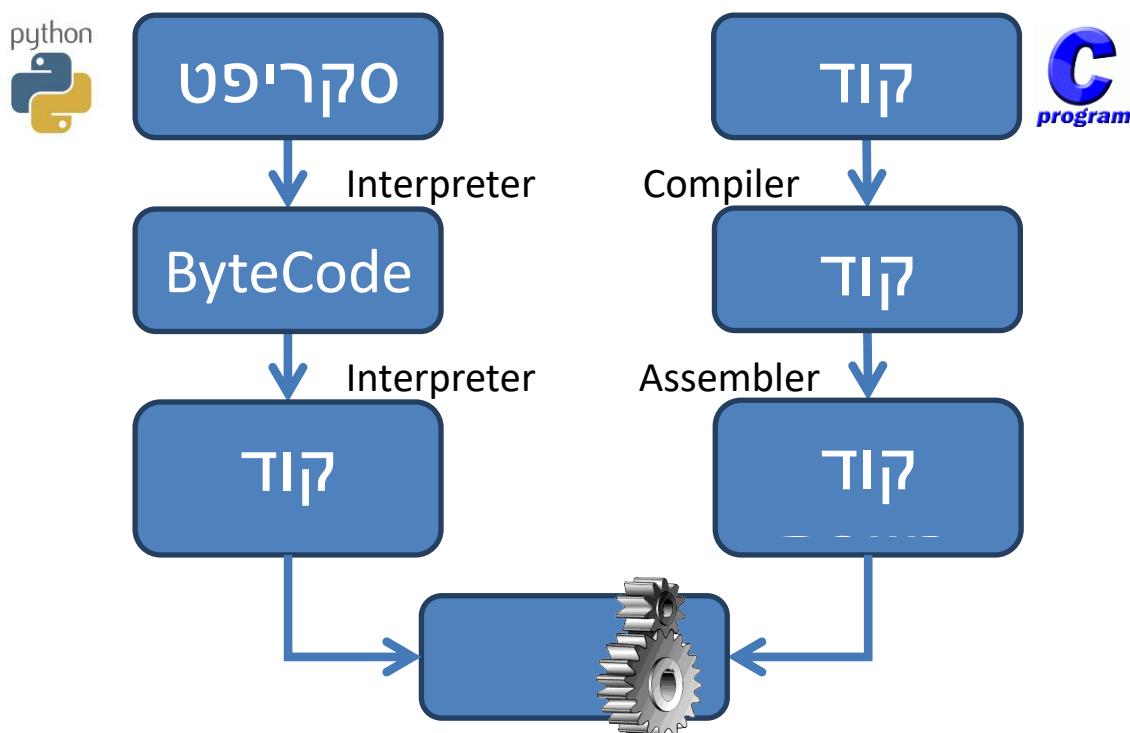
```

כasher nnesha libatzu kompiletsia, nakbel houdat shagiya:

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

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

עתנו נعود לדון במתוך שפה פיתון, שכזכור הינה שפת סקריפטים. על סקריפט פיתון פועלת תוכנה שנקראת interpreter ("פרשן"). interpreter עובד בצורה אחרת לגמרי מאשר הקומפיילר והאסמבילר בהם משתמשים כדי לתרגם את שפת C למוגנה. הוא אינו יוצר קובץ אסמבלי וגם אינו יוצר קובץ הריצה. במקום זאת, כל פקודה שתכתבו בשפת פיתון מתורגמת לשפת מוכנה רק בזמן הריצה. תוך כדי תהליך הפירוש, נוצר קובץ עם סיומת .pyc, שמכיל bytecode – הוראות שונות של interpreter – אך כאמור זה אינו קובץ בשפת מוכנה, 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 תקינה בשפת פיתון, שמדפסה "Hello world" ונוסף גם לה blablabla

```
1 print 'Hello world'
2 blablabla
```

*** שימוש לב ***

דוגמאות הקוד בגרסתו 2.0 (2.0) של הספר עדין מבוססות פיתון 2.7, בה פקודת הדפסה היא ללא סוגרים. בפייטון גרסה 3, שהינה הגרסה שהתקנו, לאחר פקודת print יש לשימוש סוגרים ובתוכם להכניס את הטקסט המודפס, עם גרשימים.

לדוגמה:

`print("Hello world")`

בالمושך יעדכנו דוגמאות הקוד, עד איז הוסיף סוגרים היכן שצרכי.

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 ניסיון להריץ את השורה blablabla, ניסיון שהסתיים בשגיאת הריצה. הנקודה המעניינת היא שהתוכנית רצתה באופן תקין עד שארעה השגיאה, וזאת בניגוד לתוכנית זהה בשפת C, שככל לא רצתה. הסיבה שהצלחנו להגיע בפייטון לנקודה שחלהק מהתוכנית רצתה, היא בדיק בಗל תחיליה ההמרה השונה. בפייטון לא נוצר קוד בשפת אסמבלי וגם לא קובץ הריצה, ולכן השגיאה לא התגלתה עד לנקודה 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 סלחנית לשגיאות מאשר שפות אחרות. שימוש לב גם עד כמה הקוד בשפת פיתון קצר יותר מאשר בשפת C. לכן, הדעה הרווחת היא שקל יותר ללמידה 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 מאייתנו לכתוב קוד שלא עומד בכלל התחביר של השפה וכן הściי לבעיות בזמן ריצה הוא קטן יותר. התרסקות של קוד תוך כדי ריצה היא חמורה לאין שיעור מאשר שגיאת קומPILEציה, אותה ניתן לגלוות ולדבג לפני הריצה. שנית, העובדה שקוד בשפת

C מתרגם לשפט מכונה לא שורה אחר שורה אלא קובץ אחד, מאפשרת לבצע תהליכי "יעול" (אופטימיזציה) של הקוד, כך שהוא עשוי לזרע יותר מהר ולצרוך פחות זיכרון מאשר קוד מקביל בשפט פיתון. זה דבר חשוב למי שרצוים להריץ אפליקציות "כבדות", כגון גרפיקה מורכבת או הצפנה.

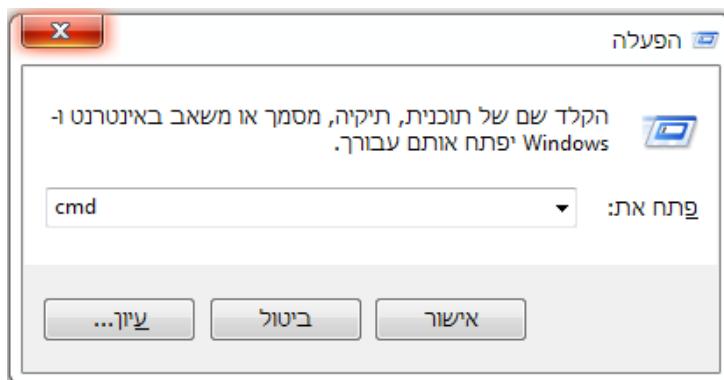
לסיום, אפשר לומר שפייתון היא שפה נוחה וקלת כתיבתה, שמאפשרת להגיע במהירות לתוכניות עובדות, גם אם הדבר בא על חשבון מהירות או יעילות. בשל כך זכתה פיתון למפתחים רבים, שדAGO לכתב מודולים – קטעי קוד SMBצעים שימושיות שונות – ולהפיץ אותם. כתוצאה לכך, אחד ה יתרונות העיקריים של פיתון הוא שניתן לקבל מן המוכן קוד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

עבודה דרך command line

נפעיל את פיתון בדרך הקצרה והירה. 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":



ויפיע מסך "הפעלה" או באנגלית Chat, ובתוכו כתבו את שם התוכנה שאתם רוצים להפעיל. במקרה זה – cmd.



עם לחיצה על מקש **enter** הגיעו למסקן **command-line**. במסך יהיה כתוב שם התקינה בה אתם נמצאים, לדוגמה `cyber:c.` כתעת כתבו `python` והקישו `enter`. ברוכים הבאים לפיתון!

```
c:\>python
Python 2.7.13 |v2.7.13:a06454b1afaf1, Dec 17 2016, 20:53:40| [MSC v.1500 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

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



נסו לבצע את התרגיל $3/4$. מה קיבלתם? כפי ששמתם לב, בחלוקת של מספרים שלמים זה זהה, התוצאה היא תמיד מספרשלם, ולכן פיתון מעגל אותה. נסו לכתוב $0.4/3.0$. נסו $3.0/4.0$. מה אפשר להסיק?

מספרים בסיסים שונים

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

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



כיוון שלבני אנוש מסובך מעט לקרוא רצפים ארוכים של אחדות ופסים, מקובל להציג מידע מיידע ששמור בזיכרון המחשב בספרות הקסדצימליות – בסיס 16. זאת מושם שככל ספרה הקסדצימלית מייצגת 4 ספרות בינהיות (שים 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 לקריאה!

אם נרצה לכתוב בפייתון מספרים בינהירים, עליהם להתחיל ב-b. לדוגמה 0b11 0b11 הינו 3, 0b1111 הינו 7. כדי לכתוב מספרים הקסדצימליים בפייתון, נוסיף להם תחילית x. כך לדוגמה, 0xA1 הינו 26, ו-0x2B הינו 43. אפשר לראות זאת בקלוות אם נכתוב אותן 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:

```
>>> help()

Welcome to Python 2.7!  This is the online help utility.

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/2.7/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, or topics, type "modules",
"keywords", or "topics".  Each module also comes with a one-line summary
of what it does; to list the modules whose summaries contain a given word
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:

אם נכתוב **POWER** (המוני האנגל' 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** נכתוב **quit** ונגיע חזרה לחילון הפוקודה שלנו. נסו כעת להשתמש במה שלמדנו!

```
help> topics
```

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

ASSERTION	DEBUGGING	LITERALS	SEQUENCEMETHODS2
ASSIGNMENT	DELETION	LOOPING	SEQUENCES
ATTRIBUTEMETHODS	DICTIONARIES	MAPPINGMETHODS	SHIFTING
ATTRIBUTES	DICTIONARYLITERALS	MAPPINGS	SLICINGS
AUGMENTEDASSIGNMENT	DYNAMICFEATURES	METHODS	SPECIALATTRIBUTES
BACKQUOTES	ELLIPSIS	MODULES	SPECIALIDENTIFIERS
BASICMETHODS	EXCEPTIONS	NAMESPACES	SPECIALMETHODS
BINARY	EXECUTION	NONE	STRINGMETHODS
BITWISE	EXPRESSIONS	NUMBERMETHODS	STRINGS
BOOLEAN	FILES	NUMBERS	SUBSCRIPTS
CALLABLEMETHODS	FLOAT	OBJECTS	TRACEBACKS
CALLS	FORMATTING	OPERATORS	TRUTHVALUE
CLASSES	FRAMEOBJECTS	PACKAGES	TUPLELITERALS
CODEOBJECTS	FRAMES	POWER	TUPLES
COERCIONS	FUNCTIONS	PRECEDENCE	TYPEOBJECTS
COMPARISON	IDENTIFIERS	PRINTING	TYPES
COMPLEX	IMPORTING	PRIVATE NAMES	UNARY
CONDITIONAL	INTEGER	RETURNING	UNICODE
CONTEXTMANAGERS	LISTLITERALS	SCOPING	
CONVERSIONS	LISTS	SEQUENCEMETHODS1	

הסימן _ (קו תחתון)

כפי שראינו, אפשר להגיד בפייתון משתנים בקיל' קלות, פשוט באמצעות כתיבת שם המשתנה, הסימן '=' ולאחר מכן הערך שהמשתנה מקבל. קיימים בפייתון סוגים רבים של משתנים, עליהם נלמד בהמשך. כדי להגיד משתנה בשם a, אשר ערכו הוא 17, פשוט כתובים $a=17$.



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

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

קל? כתע חשבו את הביטוי הבא:

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

דרך אחת היא פשוט להעתיק את כל התרגילים מההתחלתה. אולם כפי שהבחנתם, מדובר למשעה באותו תרגיל פרט לכך שהוא חזקתו 2. כדי למצוא את הפתרון באופן אלגנטי ובלי הקלדות מיותרות פשוט רישמו את הביטוי הראשוני, לחזו `enter` ובשורה הבאה כתבו:

$$_{-}^{**2}$$

פירשו של הקוו התחתון – "קח את התוצאה الأخيرة שחייבת". באופן זה נוכל לחסוך זמן כתיבה ולבצע חישובים בקלות. שימו לב שהטריק הזה עובד ב-`interpreter`, אך לא בסביבת העבודה אותה נלמד בהמשך.

הרצת 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 משתמש בו הרבה, כך שאם אתם מעדיפים לא להתקין אותו תוכלו להשתמש בתוכנת `notepad` שmaguu עם חלונות. איך מפעלים? בדיק כómo שלמדנו להפעיל כל תוכנה. ליחזו על `R+winkey`, כתבו `notepad` בחלון הפעלה זהה.

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

```
hello.py
1 print 'Hello cool cyber student!'
2
```

ושימרו את הקובץ.

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

```
cd c:\
```

```
cd cyber
```

לאחר שהגענו לתקיה הנכונה, נכתוב:

```
python hello.py
```

והקובץ שלנו יורץ מיד ☺

סיכום

בפרק זה רכשנו את הבסיס לתוכנות פיתון. אנחנו מבינים שפיתון היא שפת סקריפטים, אנחנו יודעים להריץ פקודות פיתון פשוטות גם דרך command וגם דרך קבצי פיתון שייצורנו. 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,我们知道 גם יודעים לחפש עזרה בנושא לא מוכרים. מעכשיו, יוכל למצוא הדרכה על כמעט כל נושא שנלמד ולהסתדר לבד, 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 אחריו פקודה, תוך כדי הציג ערכיו המשתנים השונים. זו יכולת קריטית לשוכרים להבין מדוע תוכנית שכתבנו לא עובדת באופן תקין.

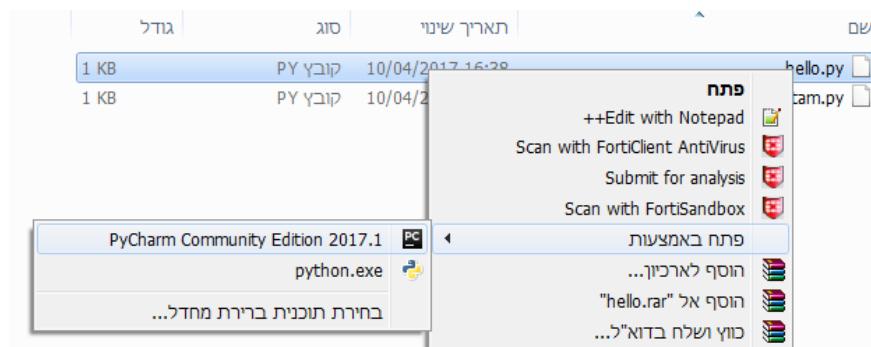
בחרנו להציג בפניכם את סביבת העבודה PyCharm. היכולות שיש给她 של PyCharm זו מעניקה לנו הם:

- כתבן (תחליף `notepad++`).
- איתור שגיאות אוטומטי – כן, PyCharm מוצא בשבילינו את השגיאות בקוד שלנו. אין הכוונה לכך ש- PyCharm יודע לתקן עבורנוágים באლגוריתם, 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 קrise או באג – תוכל לחסוך זמן שימושי כאשר התוכנית שלנו מתרסקת, על ידי כך שבמקרה לאתחל את התוכנה פשוט נוריה `PyCharm` להריץ אותה שוב.

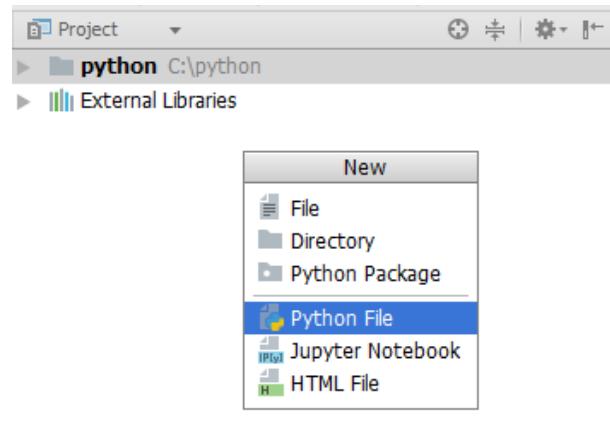
כעת נלמד להשתמש בסביבת העבודה PyCharm.

פתיחה קובץ פייתו

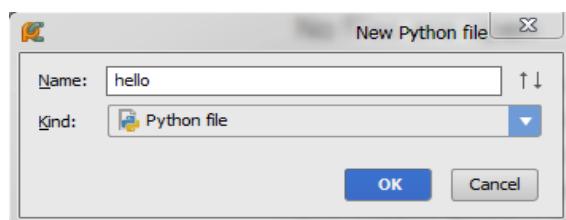
אם הקובץ כבר קיים, כל שנצרך לעשות הוא להקליק עליו קליק ימני ולחזור אפשרות `open with` ולאחר מכן `PyCharm Community Edition`.



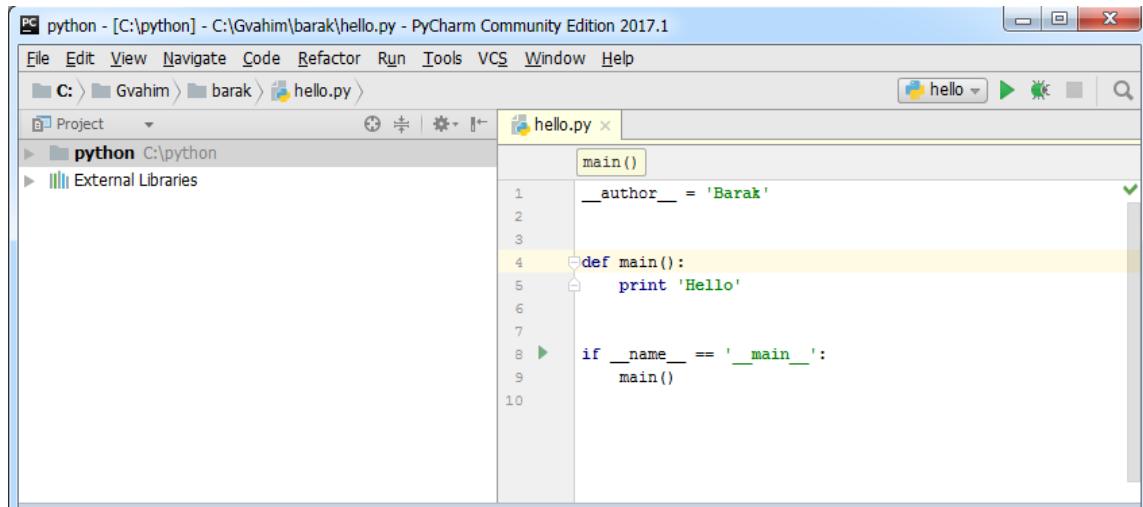
אם נרצה לפתח קובץ חדש, אז נקליק על האיקון של PyCharm ומהתפריט נחבר ב-file, לאחר מכן New וaz מבין האפשרויות נבחר ב-file:python file



לאחר מכן ניתן לקרוא החדש שם:



וכעת ניתן לעורק את הקובץ החדש שיצרנו:



קובץ הפיתון הראשון שלנו

נסקרו את השורות בקובץ הפיתון הראשון שלנו:

```

__author__ = 'Barak'

def main():
    print 'Hello'

if __name__ == '__main__':
    main()

```

בשורה 1 נכתב את שם המחבר. זה חשוב מאד – כשןקרא קוד של מתכנת אחר חשוב שנדע מי המחבר. כך, אם אנחנו עובדים בצוות, נדע אל מי לפנות במקרה של בעיה.

בשורה 4 מוגדרת פונקציה בשם `main`. כפי שברור מהשם שלה, זו ההפונקציה הראשית של התוכנית שלנו. המשמעות היא שגם ההפונקציה הראשונה שנקראת (מיד בין מי קורא לה) וכן כל מה שאנו רוצים שיבוצע צריך להיות כתוב בתוך `main`, או לחלופין ש-`main` תקרה לו.

בשורה 8 ושורה 9 יש תנאי מעט מסובך, שבמהלך הלימוד נבין מה פירשו. בקצרה – אנחנו יכולים להריץ סקריפט פ'ייטון בשתי צורות. האחת, היא פשוט להריץ אותו. השנייה, היא להריץ סקריפט אחר שקורא לסקריפט שלנו. הדרך השנייה מתבצעת בעזרת פ'קודה שנקראת `import` ומלמד עליה בהמשך. נניח שתכתבנו סקריפט פ'ייטון שמכיל כמה פונקציות מעניות, וחבר לנו רוצה להשתמש בהן. אם החבר יקרא לסקריפט שלו על ידי פ'קודה `import`, אז עלול להווצר 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 שורות 8 ו-9 בקוד שלנו. שורות אלו אומרות לפ'ייטון "שמע פ'ייטון", יכול להיות שהסקריפט הזה יourcef לסקריפט אחר. לכן קודם כל לבדוק אם מי שמריז את הסקריפט זה לא מישוה אחר שעשה לו `import`, ורק אם הסקריפט אינו מורץ מ-`import` אז תקרה לפונקציה `main` שתחל את ריצת הסקריפט.

סדר ההרצה של פ'קודות בסקריפט פ'ייטון

התבוננו שוב בסקריפט `udHello.py`. מה יהיה סדר ההרצה של הפ'קודות? כאמור, איזו פ'קודה תרוץ ראשונה?

ה-`interpreter` של פ'ייטון, שתפקידו לתרגם את פ'קודות הפ'ייטון לשפת מכונה, מחפש את הפ'קודה הראשונה שצמודה לצד שמאל ואז הוא בודק אם לא מדובר בהגדה של פונקציה. לכן, בשורה מספר 1 – שצמודה לצד שמאל – פ'ייטון יבצע את הפ'קודה. לאחר מכן ילמד משנה פונקציה בשם `main`. פ'ייטון יוסיף את `main` לרשימה הפונקציות המוכרות לו. לעומת, זה אפשר לקרוא ל-`main`, אך אין זה אומר שפ'ייטון מריז את `main` או אפילו בודק שהקוד של `main` הוא תקין. כל מה שידעו לפ'ייטון – ישנה פונקציה `main`.

רק בשורה 8 נתקל פ'ייטון בפקודה שעליו להריץ ושמבצעת משזה. אם התנאי שבשורה מתקיים, פ'ייטון ממשיר אל שורה 9 ושם נאמר לו להריץ את הפונקציה `main`. בשלב זה פ'ייטון יקפוּץ אל `main` ויחל לבצע את מה שנכתב בה. מאידך, אם התנאי שבשורה 8 אינו מתקיים, פ'ייטון הגיע לסוף הסקריפט וריצת הסקריפט תסתיים.

כעת שמו לב לסקריפט הבא – מה יהיה סדר הפ'קודות שיבוצע?

שיםו לב לכך, שזו הינו תוכנית פ'ייטון 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 הפ'ייטון. היה נכון הרבה יותר לכתוב את כל פ'קודות ההדפסה שלנו בתוך פונקציית `main`. אם כן, מה יהיה סדר הפ'קודות שיבוצע?

```

1     __author__ = 'The Beatles'
2     print 'You say',
3
4
5     def main():
6         print 'Hello'
7
8         print 'Goodbye'
9
10    if __name__ == '__main__':
11        print 'I say',
12        main()

```

שורה 1 קובעת את ערך המשתנה `__author__`.

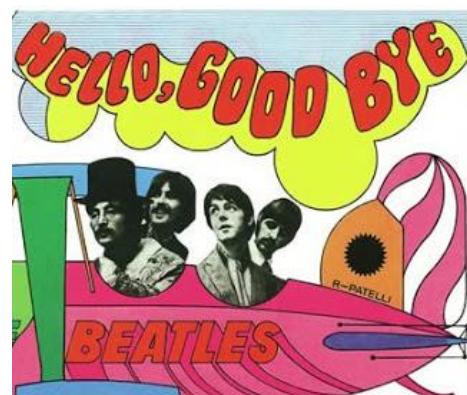
שורה 2 תדפיס למסך את הטקסט "You say". שמתמם לב לkr שיש פסיק בסוף פקודת הדפסה? בלי הפסק,
לאחר `print` תבוצע ירידת שורה. הפסיק קובע שלאחר הדפסה יהיהתו אחד של רווח.

לאחר מכן תבוצע שורה 8, יודפס "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` תבוצע הדפסה של Hello.

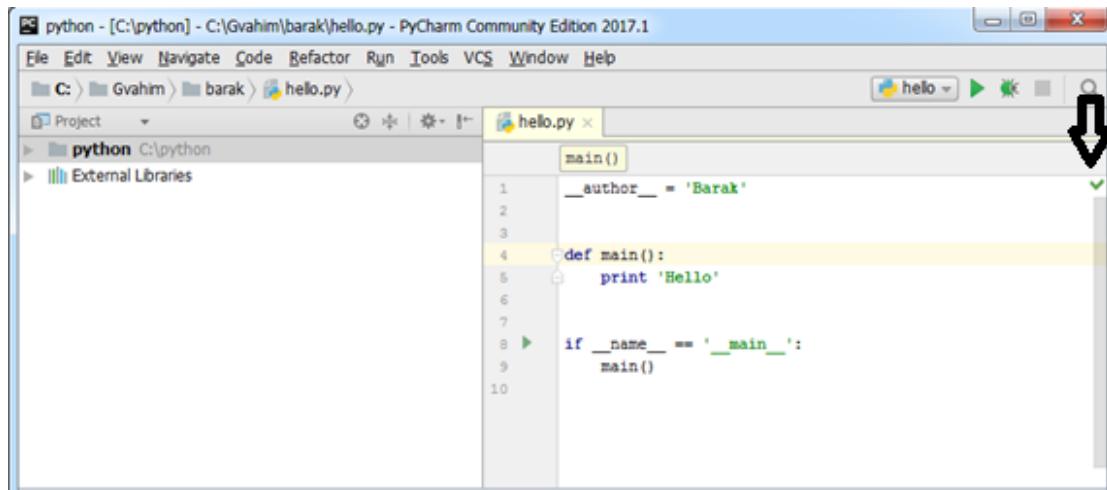
לאחר שהפונקציה `main` תסתיים את ריצתה, אליה היא נקראת משורה 12, היא אמורה לחזור ולהריץ את המשך התוכנית – אילו היה זה – אחרי שורה 12. כיוון שורה 12 היא סוף התוכנית, בkr יסימם הסקריפט את ריצתו.

You say Goodbye
I say Hello

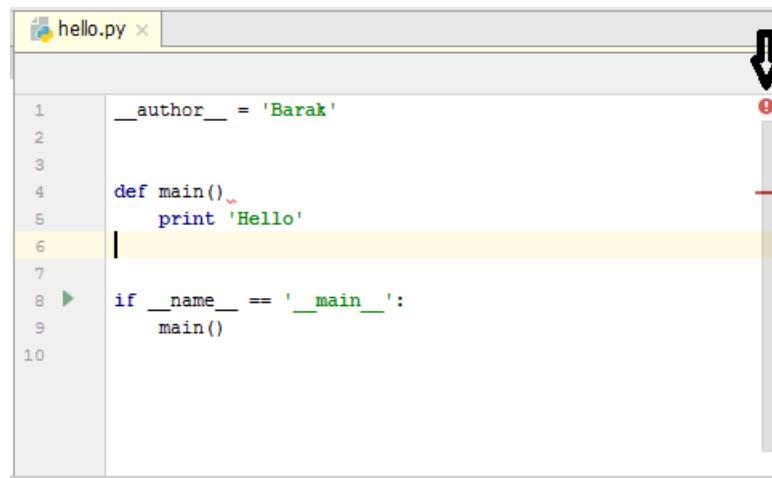


התרעה על שגיאות

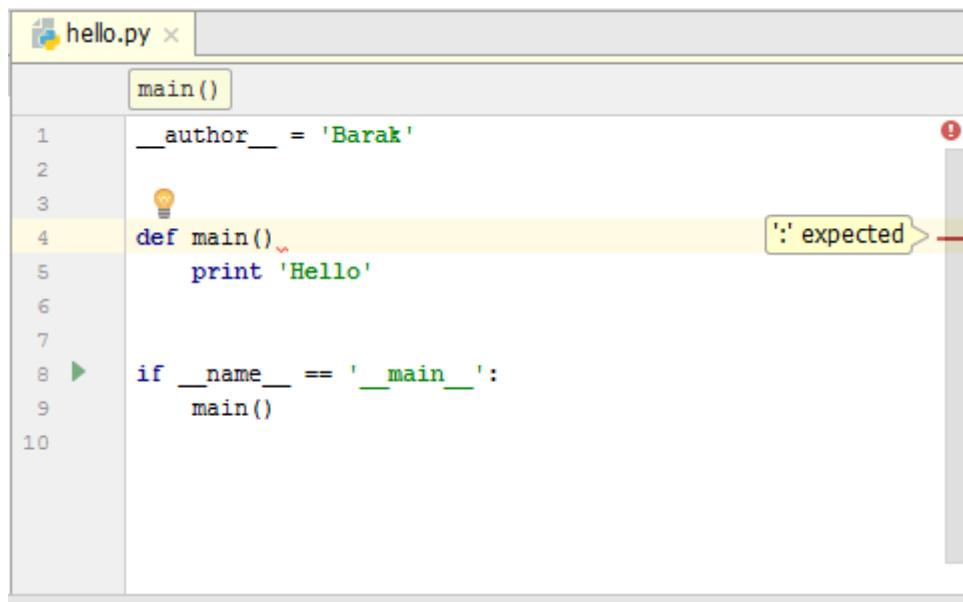
אם אתם מבחינים בסימן ה-V הירוק, שנמצא בחלק הימני העליון של PyCharm?



זהו סימן שהתוכנית שכתבנו אינה מכילה שגיאות. שימושו לב מה קורה כאשר אנחנו מוחקים את סימן הנקודתיים לאחרי המילה `main()`:



קיבלו סימנים אדומים במספר מקומות במסך. ראשית, במקום הנקודתיים שמחקנו מופיע קו תחתוי אדום. שנית, ה-V הירוק הפך לסימן קריאה לאדום. שלישיית, אם נעמוד עם העכבר על הקוו האדום שבצד ימין, קיבל הסבר מה הבעיה בקוד שלו.



The screenshot shows a PyCharm code editor window for a file named 'hello.py'. The code contains the following Python script:

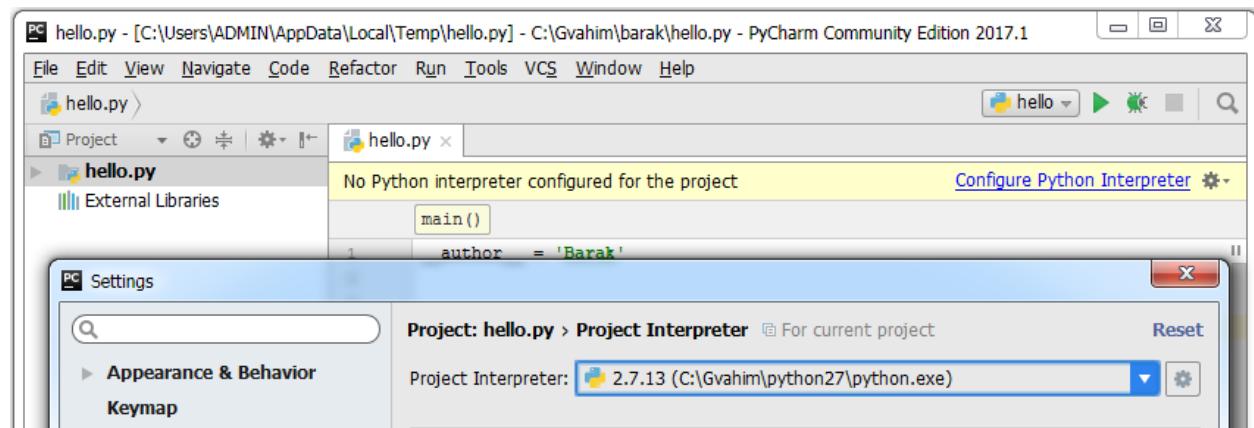
```
1  main()
2
3
4  def main():
5      print 'Hello'
6
7
8  if __name__ == '__main__':
9      main()
10
```

A red exclamation mark icon is positioned at the top right of the code editor. A tooltip '':' expected' is displayed near the end of the line 'def main():'. A small yellow lightbulb icon is also visible near the start of the 'def' line.

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

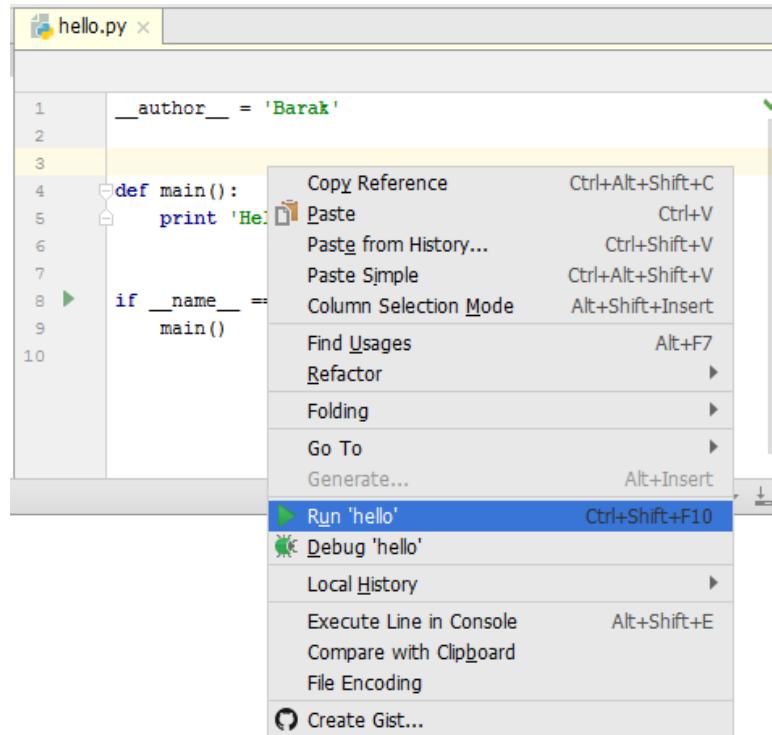
הרצה הסקריפט ו疎ך המשתמש

סביבת התקנות של גבהים אמורה להתקין ולאפשר לכם לעבוד מיידית בלי צורך בהגדרות נוספת, אולם אם מופיעעה לכם שגיאה מסוג "No Python interpreter configured for the project", כפי שאפשר לראות מודגש בצד שמאל בתמונה, להלן הדרך לסדר את ההגדרות. ראשית צריך להגדיר ל- PyCharm Aiha ניתן למצוא את ה- interpreter של שפת פיתון, זאת מכיוון שיש גרסאות שונות של שפת פיתון, ויתכן שעל אותו המחשב מותקנות גרסאות שונות. לכן, נקליק הכיתוב Configure Python Interpreter ולאחר מכן נבחר את גרסת הפיתון

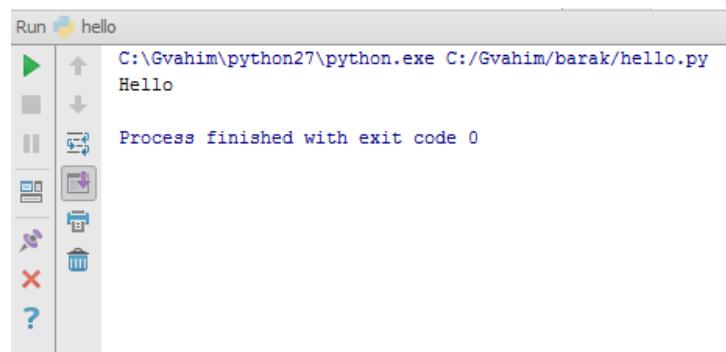


שモתקנת במחשב שלנו, אשר מגיעה עם התקינה של סביבת גבהים:

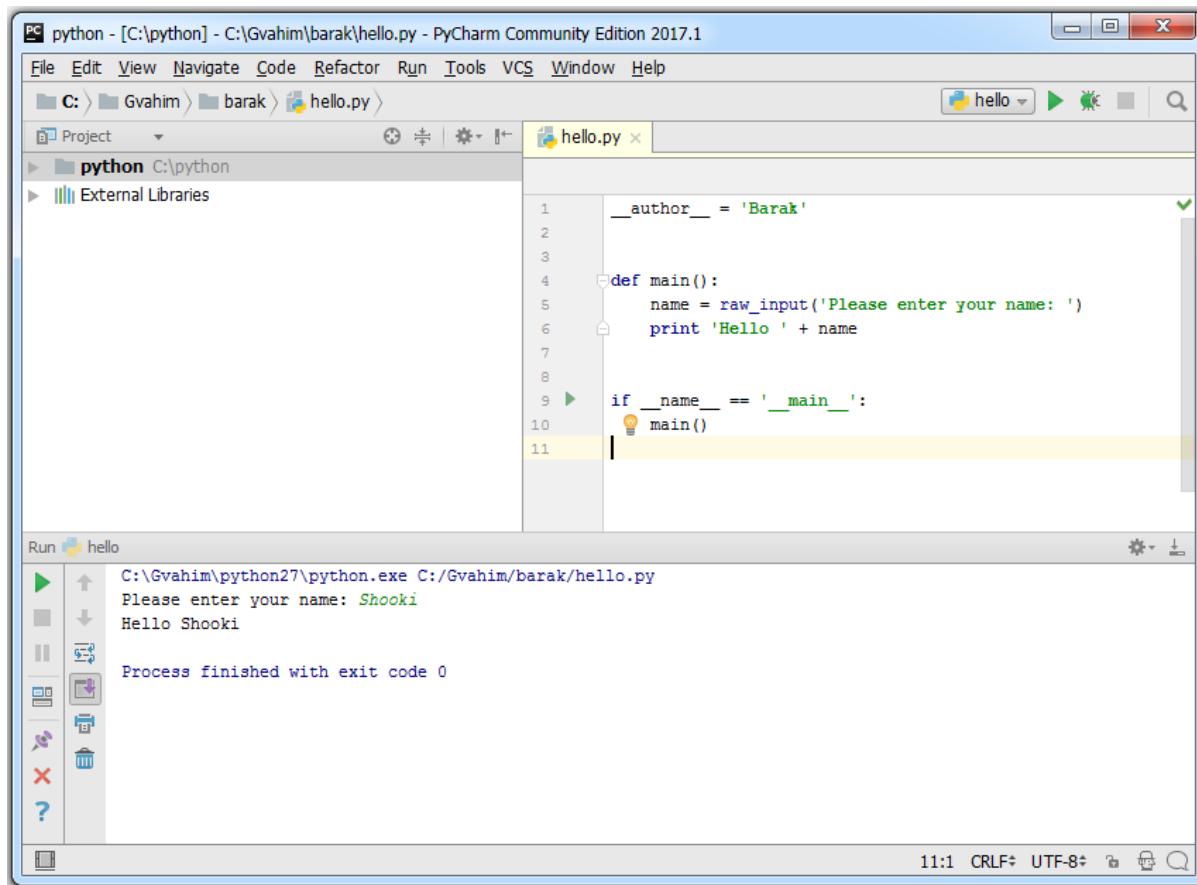
כדי להריץ את הסקריפט שכתבנו, נלחץ על עכבר ימני ונבחר באפשרות חוץ:



בעקבות הלחיצה על run יופיע מסך משתמש, שם יודפס כל מה שהסקריפט שלנו מדפיס למסך. רואים את ה- "Hello" שעשינו לו?



מסך המשתמש משמש גם לקבלת קלט מהמשתמש. שימושו לב לפקודה שבסורה 5 :



פקודה זו מדפיסה למספר את מה שנמצא בסוגרים, וכל מה שהמשתמש מקליך נכנס לתוך משתנה, במקרה זה קראנו לו `name`.

בשורה 6 מתבצעת הדפסה של Hello ולאחר מכן הערך שנמצא בתוך המשתנה same. התבוננו בחלק התיכון של המסר ובידקו שגם מובנים את תוצאת הריצעה.

PyCharm עם דיבוג

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

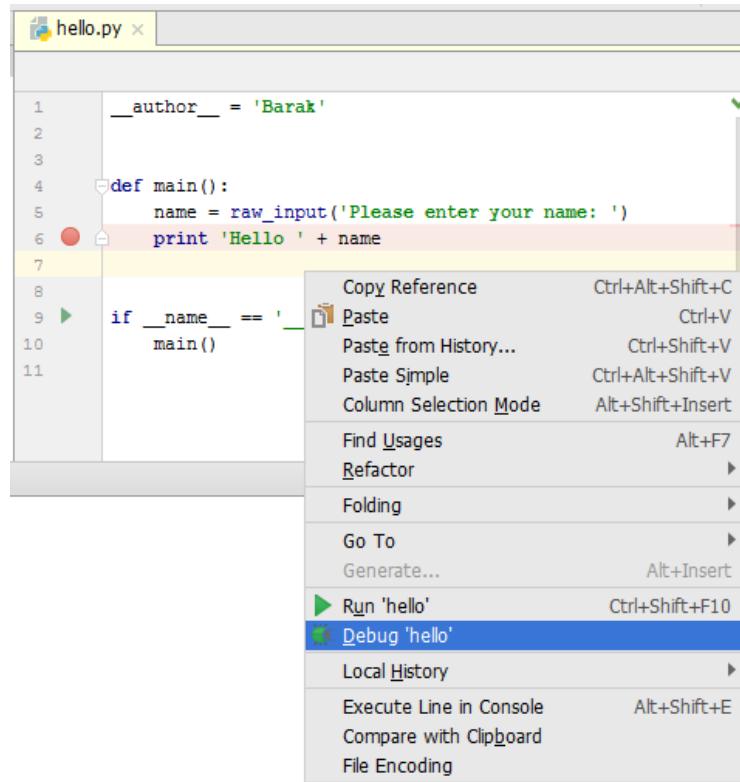
בשלב הראשון יש לשתול breakpoint בקוד. דבר זה מtbody על ידי לחיצה שמאלית על העכבר, באזור האפור שליד מספרי השורות. בעקבות כך, תופיע נקודה אדומה ליד מספר השורה, כפי שנitin לראות ליד שורה 6 בקטע הקוד הבא:

```

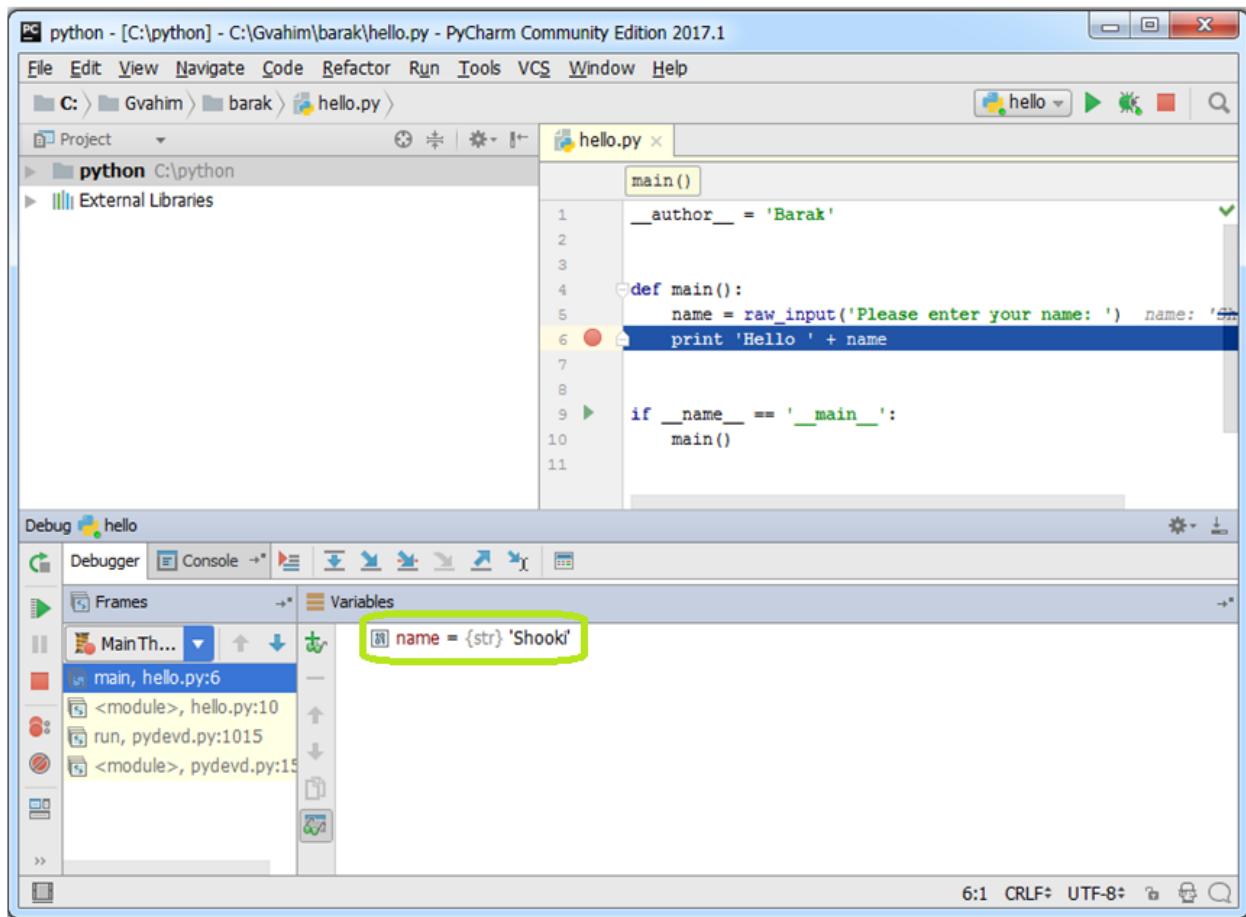
1 __author__ = 'Barak'
2
3
4 def main():
5     name = raw_input('Please enter your name: ')
6     print 'Hello ' + name
7
8
9 if __name__ == '__main__':
10    main()
11

```

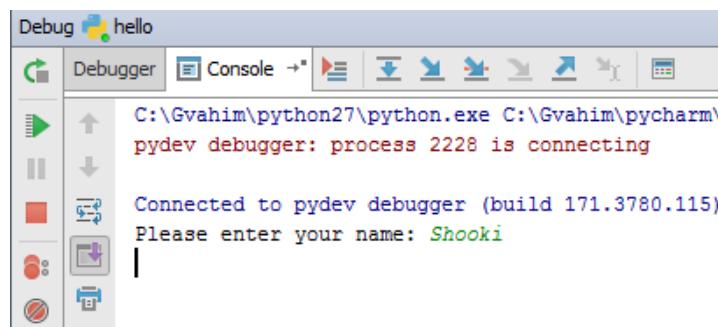
בשלב הבא צריך להריץ את התוכנית במוד debug, על ידי קליק ימני ובחירה האפשרות debug מבין האפשרויות:



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 debugger, אשר מכיל מידע על המשתנים שמוגדרים בתוכנית שלנו. לדוגמה, המשתנה `name` הוא מסוג `str` (בהמשך, נבין את המשמעות של סוגי המשתנים השונים) וערך הוא `Shooki Shooki`, הערך שהמשתמש הזין לתוכו בשורת קוד מספר 5. ניתן לראות את הערך של `name` בחלון variables, לנוחות ההתמצאות הוא מודגש במסגרת צהובה:



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



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



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

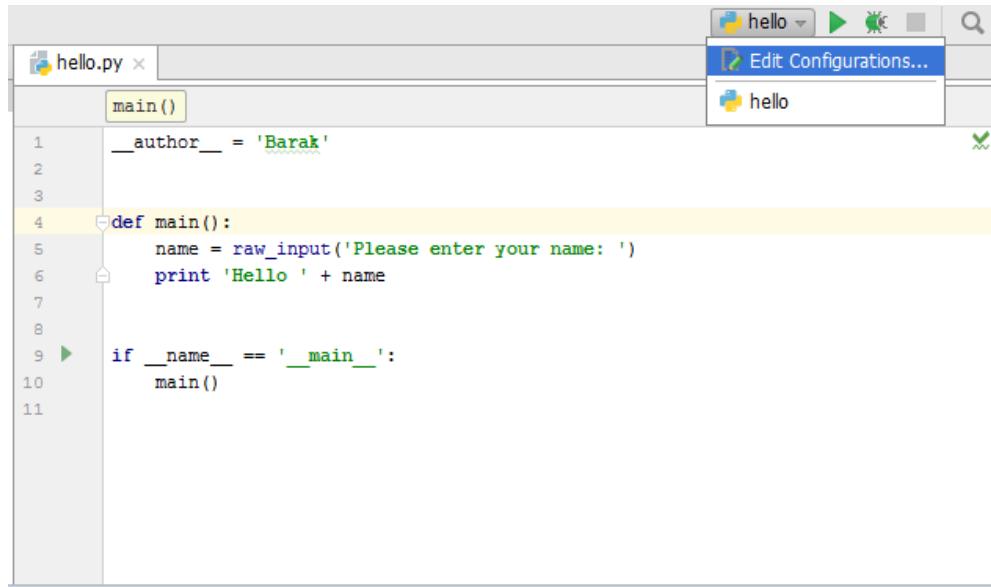


מומלץ לשנות בפעולות אלו, מכיוון שבבוא העת שימוש נכון בהן יאפשר לכם לגלוות בעיות בתוכנה שלכם!

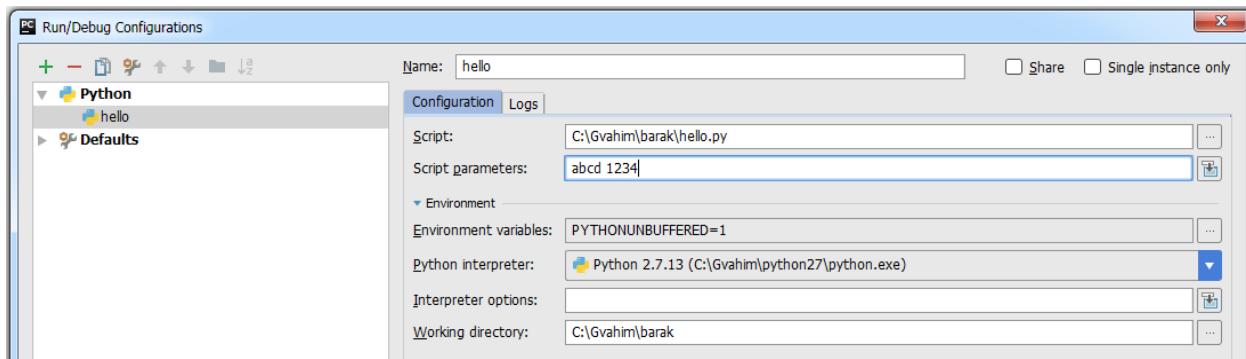
העברה פרמטרים לסקריפט

دلגו על החלק זהה, וחיזרו אליו רק לאחר שתלמידו 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 למעלה, ואז נבחר `:configurations`



cutת יפתח מסך שבו ניתן יהיה לקבוע **Script Parameters**. ניתן להזין לתוכו כמה פרמטרים שנרצה, עם סימן רווח בין פרמטר אחד לשני:



סיכום

בפרק זה סקרנו את סביבת העבודה PyCharm, שתשתמש איתה 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 ובפרקים 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 שפייתון 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

פייתון ידפיס לנו את התשובה – 2.

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

משתנה מסוג int, float, bool: 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 שבוטופו 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 כמות סופית של ביטים. איז הדיק זהה גורם לכך שם נעשה פעולות חשבוניות שיגרמו להצטברות השגיאה, לבסוף קיבל 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, סיימנו את הדין גם ב-int וב-float. ליתר טיפוסי המשתנים – יוקדשו פרקים נפרדים.

תנאים

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

שווין – הסימן `==` 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 שוגדר בסימן. לדוגמה:

```
x = 20
if x <= 21:
    print 'Yay!'
```

שווין למשתנה בוליאני – כאשר מושווים 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 מקוצר – 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. למשל, יש לבצע משחו 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 מ-not, or, and.

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

```
x = 21
if 20 < x and x < 22:
    print 'Yay!'
```

אפשר גם לכתוב אותו ביטוי עם סוגרים, לעתים יותר קל לקרוא אותו כך:

```
x = 21
if (20 < x) and (x < 22):
    print 'Yay!'
```

כמובן שתתנאי האחרון אפשר לכתוב גם בצורה קצרה, באופן הבא:

```
x = 21
if 20 < x < 22:
    print 'Yay!'
```

תנאי `or` משמעו "או". לדוגמה, כדי שהתנאי הבא יתקיים, מספיק ש-`x` יהיה שווה `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 `is`. לדוגמה:

```
x = 5
if x is not 5:
    print 'Not true!'
```

כאשר מדובר בערך בוליאני, הדרך המקובלת היא לרשום `not` לפני שם המשתנה:

```
x = False
if not x:
    print 'Yes!'
```

שימוש ב-`is`

בדוגמאות הקודמות ראיינו שאפשר לכתוב תנאי עם `==` וגם את אותו תנאי עם הביטוי `is`. מה ההבדל ביניהם?

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

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

בסירטון הבא יש הדוגמה של ההבדל בין `is` 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 אחת? כמובן! את מושג הבלוק הכי פשוט להבין מוצפיה 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 כלן אם התנאי יתקיים. שימושם לבן להבדל בין הקוד לעיל:

הקוד הבא:

```

x = 20
if x == 21:
    print 'Hi!'
else:
    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 מסולסים, בפייתון סוף הבלוק נקבע רק לפני ההזחה. על כן הכרחי גם שהهزחה תהיה עקבית – אנחנו לא יכולים לעשות לפעמים הזחה של שני רוחים ולפעמים של ארבעה רוחים. חשוב לציין גם שתווי רווח וטאב שונים זה מזה. כלומר, גם אם לנו נראה שזרה שיש בה טאב נמצאת בהזחה זויה לשורה אחרת שיש בה הזחה של ארבעהתווי רווח, מבחינת פיתון אלו תווים שונים לגמרי. לכן, אם התחלנו לעשות הזחה עם טאבים, רצוי שנמשיך רק עם טאבים.

לא רק מה שנמצא בהזחה של טאב אחד שייר לבлок. בתוך בלוק יכולים להיות עוד טאבים, לדוגמה עקב שימוש בתנאי או נספחים. כל הפקודות שנמצאות בהזחה של פחות 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 :)'

```

שימוש לבן-crash- PyCharm מסיע ו"מסמן" את הבלוקים השונים באמצעות סימני אפורים דקים. מה יודפס כתוצאה מריצת הבלוק? יורצו כל השורות חוץ מאשר אלו שנמצאות בבלוק של `x < 21` if, כיוון שהתנאי זה אינו מתקיים. פלט 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 שניצטרח לבדוק פעמיים – פעם אם התנאי מתקיים ופעם אם הוא אינו מתקיים – אפשר להשתמש בפקודה `else`. אם התנאי שנמצא ב-`if`-ו אינו מתקיים, יבוצע הקוד בבlok של `else`.

```
x = 20
if x == 21:
    print '21!'
else:
    print 'Something else'
```

מה אם יש לנו נוסף תנאי רצימ לבודק? 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 apple pie'
else:
    print 'Watching "Friends" on TV'
```

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



תרגיל

כתבו 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 `if age > 100`. לעומת זאת, ניתן ליצור תנאי מושך יותר, כמו `while age < 18`. בדוגמה הבאה, נרצה לקרוא קלט מהמשתמש ולבצע את הוראות המשמש, כל עוד המשמש לא כתוב 'Exit'. במקרים אלו שימוש ב-`if`-`else` הוא לא מספק טוב, כיון שאנו רוצים לדעת כמה פעמים התנאי שלנו צריך לזרז. 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 נגידיר. לאחר מכן יש בлок של פקודות אשר יבוצעו בזוו אחר זו, ובסיוף הבלוק תהיה חזרה אל בדיקת התנאי.

```
while condition:
    # do something
    # 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):

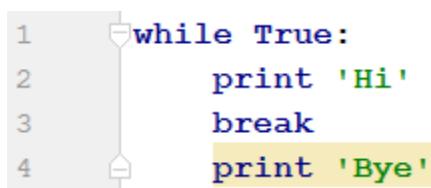
```
1     age = 17
2     while age < 18:
3         age += 1
4     print 'Not yet 18'
```

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

מה לדעתכם יקרה אם נריץ את הלולאה הבאה?

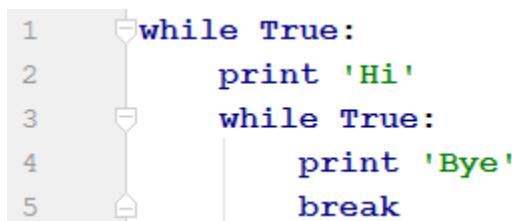
```
while True:
    print 'Hi'
```

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



שורה 2 תדפיס `Hi`. בשורה 3 אנחנו מורים למחשב לצאת מהlolאה, لكن שורה 4 לעולם לא תורץ (ואכן, שימוש לב-`sh-Charms` מסמן אותה בצלע בולט, ומדגיש לנו שכתבנו קוד שלולום לא יורץ).

מה ידפיס הסקריפט הבא?



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

תרגיל – Take a Break



הדיםו את כל המספרים בסדרת פיבונצ'י אשר ערכם קטן מ-10,000. חובה להשתמש ב-`while True!` תוכלם 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%D7%95%D7%99

לולאות for

לולאות `for` הן שימושיות במיוחד כאשר יש לנו אוסף של איברים שאנחנו רוצים לבצע עליהם פעולה כלשהי, בנגדוד 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 עליון 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" iterator), לאחר מכן המילה `in` ולאחר מכן אוסף של איברים. בדוגמה הבאה אנחנו שמים כמה מספרים בתוך סוגרים מרובעים, מה שאומר שהם הופכים ל"רשימה", טיפוס של משתנה שנלמד עליון בהמשך:

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

פלט ההרצה יהיה:

```
0
2
4
```

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

```
for i in ['Ben Gurion', 'Sharet', 'Begin', 'Rabin']:
    print i + ' was the prime minister of Israel'
```

פלט הריצה יהיה:

```
Ben Gurion was the prime minister of Israel
Sharet was the prime minister of Israel
Begin was the prime minister of Israel
Rabin was the prime minister of Israel
```

מה אם רצאה לעשות לולאת `for` שעוברת על סדרה של מספרים? די נוח לכתוב את כל המספרים אחד אחריו השני... הפונקציה `range` מייצרת סדרות של מספרים. היא מקבלת בתוך פרמטרים התחלת, סוף וקפיצה ויוצרת סדרת מספרים. לדוגמה:

`range(3, 5, 1)`

תיצור סדרה של מספרים אשר מתחילה ב-3, קטנים מ-8, ומתקדמים בקפיצות של 1.

דוגמה לסקריפט שמדפיס כמו סדרות:

```
print range(3, 8, 1)
print range(3, 8, 2)
print range(5)
```

תוצאת הרצה:

```
[3, 4, 5, 6, 7]
[3, 5, 7]
[0, 1, 2, 3, 4]
```

הסבר: השורה הראשונה כאמור יוצרת רשימה של מספרים שמתחילה ב-3 ועצרים לפני 8, בקפיצות של 1. השורה השנייה היא זהה, פרט לכך שהקפיצות הן של 2. השורה השלישית מדגימה את ערכי ברירת המחדל של `range`: כאשר היא מקבלת רק פרמטר אחד, היא מניחה שיש צורך להתחיל מ-0 ולהתකדם בקפיצות של 1. במקרים אחרים, יש לפונקציה ערכי ברירת מחדל. אם לא מוזנת קפיצה, אז נעשה שימוש בקפיצה של 1. אם לא מוזנת התחלתה, מתחילה מ-0.

אקסטרה `range` באמת נוצרת רשימת איברים בזיכרון המחשב. למה זה טוב? דמיינו שאתם רוצים למצוא את כל המספרים הראשוניים בין אפס למיליארד. אתם כתובים קוד שבודק אם מספר הוא ראשוני, ומכוונים אותו לטור לולאת `for` שרצה על (`range(1000000000)`). מה שיקרה הוא שתיזכר רשימה של מיליארד מספרים בזכרון המחשב. לא דבר טוב בכלל, כיון שהוא צפוי לבצע את העבודה המחשב. לעומת זאת, אם נעשה את אותה לולאת `for` עם (`range(1000000000)`) אז נקבל את אותם המספרים אבל הם יתקבלו בלי להישמר בזכרון המחשב.

תרגיל (เครดיט: עומר רוזנבוים, שי סדובסקי)



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



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

הדףiso 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 בפייתון בעזרת סימן %. לדוגמה:

14 % 4

תחזיר 2 (14 % 4 שווה ל-3 עם שארית 2).

pass

מה אם נרצה שלולאה שכותבנו לא תבצע כלום? רגע אחד, لماذا נרצה לכתוב קוד שלא מבצע כלום? הדבר שימושי בתחום פיתוח קוד. אנחנו כותבים את שלד התוכנית, שמכיל לולאות ופונקציות, אבל משאירים אותן ריקות. כך אנחנו מסיימים במהירות את השלד ויכולים לבדוק שהתוכנון של הקוד שלנו הוא נכון, לפני שאנו מתחילה להתעסק עם המימוש עצמו. זה די שימושי כאשר כותבים תוכניות מורכבות. لكن קיימת הפקודה pass, פקודה שאומרת – אל תעשה כלום.

לדוגמה:

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

הלוולאה הזאת 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 (เครดיט: עומר רוזנבוים, שי סדובסקי)

הדףiso למסך את כל המספרים מ-0 עד 5, בקפיצות של 0.1. אבל שימושו לב – את המספרים השלמים צריך להדפיס ללא נקודה עשרונית! לדוגמה:

0
0.1
0.2
...
1
1.1
...
...
5

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

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

נקודה נוספת שנוגעת למחרוזות היא שנטוים לבלבל 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 אז מדפיס אותה. ההבדל בין המחרוזת למספר יתבהיר לנו ברגע שננסה 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` ל-`format`

עד כה בכל פעם שרצינו להדפיס משהו, היינו כתבים פקודות `print` ולאחריה את מה שרצינו להדפיס. זו שיטה מאוד נוחה לשימושים פשוטים וררים ייחיד, אבל היא נהיית מעט מסורבלת לשימושים אחרים או סוף של ערכיהם. דבר זה נכון במיוחד אם חילק מערךם 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
>>> greet + ' ' + str(age)
'My age is: 20'

```

פקודת `print` מסורבלת מכיוון שאנחנו צריכים קודם כל לחבר עם `+` את כל החלקים של המחרוזת החדשה, שנאנו מייצרים, להוסיף 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 שנאנו רוצים להדפיס, נשטים סוגרים מסולסלים. לאחר מכן נכניס הכל לתוך `format`. לדוגמה:

```

>>> print '{} {}'.format(greet, age)
My age is: 20

```

או לדוגמה:

```
>>> print 'Hello! {} {}'.format(greet, age)
Hello! My age is: 20
```

חיתוך מחרוזת – string slicing

אפשר לחתוך חלקים ממחרוזות באמצעות הפונקציה `range`:

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

החיתוך דומה לפונקציית `range` אותה הכרנו כבר.

אינדקס ההתחלת הוא `start`, ברירת המחדל שלו היא 0.

אינדקס הסיום הוא `stop`, ברירת המחדל היא סוף המחרוזת.

הפרמטר `delta` מצין בכמה אינדקסים קופצים, ברירת המחדל היא 1.

שימוש לב שגמ ערכים שליליים מתקבלים באופן תקין! לדוגמה, אם נגידיר מחרוזת אז נראה שלכל 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 הקפיצות יכולות להיות שליליות, קלומר אחרת. להלן כמה דוגמאות:

```

1 name = 'Shrek'
2 print name[1]
3 print name[1:3]
4 print name[1::2]
5 print name[:]
6 print name[:-1]
7 print name[-1::-1]

```



מה לדעתכם יהיו תוצאות הריצה?



השו את מה שחשבתם שיתקבל עם התוצאות הבאות:

```

h
hr
he
Shrek
Shre
kerhS

```

הסביר:

שורה 2 מדפסה את האיבר באינדקס מספר 1 במחוזת. שימו לב שהוא התו השני, כיוון שהאינדקסים מתחילה מ-0.

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

שורה 4 מדפסה את האיברים שמתחלים מאיןדיקס 1, מסתויימים בברירת המחדל (סוף המחרוזת) ומתקדמים בקפיצות של 2.

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

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

שורה 7 מדפסה את האיברים מהאיבר באינדקס 1 - (האחרון) ועד לברירת המחדל (האיבר האחרון) בקפיצות של



1-, כלומר אחרת. במיילים אחרות, מתחלים מהאיבר האחרון והולכים אחרת עד למגעים לאיבר הראשון.

תרגיל (קדagit: עומר רוזנבוים, שי סדובסקי)

כתבו סקRYPT שמכיל משתנה מסוג str, בעל הערך 'Hello, my name is Inigo Montoya'. השתמשו רק ב-slice'ים על המחרוזת והדפיסו את המחרוזות הבאות:

- 'Hello'
- 'my name'
- 'Hlo ynm slioMnoa'
- 'lo ynm sl'

פקודות על מחרוזות

פייתון מאפשרת לנו לעשות בקלות פעולות שונות.icut נראתה 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 כמה מהן.

```

1 message = 'Hello ' + 'world'
2 print message
3 print len(message)
4 print message.upper()
5 print message
6 print message.find('o')

```

מציג 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`, אשר מחזירה את האורך של המחרוזת. בדוגמה זו, האורך של 'Hello' (שים לב לרווח בסוף המילה) ביחד עם 'world' הוא 11 תווים.

בשורה 3 אנחנו פוגשים את המתודה `upper`. מתודות הן כמו פונקציות, אבל של סוג משתנה ספציפי. בהמשך נלמד שמתודה היא בעצם פונקציה שמוגדרת בתוך מחלקה, אבל נשמר זאת לפרק שדן במחלקות. בכלל אופן, בשלב זה נשים לב בעיקר 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 `upper` מחזירה את המחרוזת, כאשר כל התווים שלה הן אותיות גדולות. בשורה 5 אנחנו מדפיסים את `message` כדי להמחיש `upper` לא שינוי את ערכו של המשתנה `message` – כלומר, האותיות נותרו קטנות.

בשורה 6 אנחנו מכירים מתודה בשם `find`, אשר מקבלת כפרמטרתו ומחזירה את המיקום הראשון שלו בתחום המחרוזת. המיקום הראשון של האות 'o' בתחום 'Hello world' הוא אינדקס מספר 4.

dir, help

הדבר החשוב ביותר שיש לזכור לגבי מתודות של מחרוזות מגע כתה: 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 הפעולה?

ובכן, נזכיר את הפונקציה המובנית `dir`. פונקציה זו מחזירה לנו את כל המתודות של המשתנה. לדוגמה, אם נגידיר מחרוזת בשם `message` ונעשה 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__', '__formatte
r_field_name_split', '__formatter_parser', 'capitalize', 'center', 'count', 'decode',
 'encode', 'endswith', 'expandtabs', 'find', 'format', 'index', 'isalnum', 'isalph
a', 'isdigit', 'islower', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower',
 'lstrip', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rspl
it', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 't
ranslate', 'upper', 'zfill']
```

נحمد, כעת אנחנו יודעים מה שמות כל המתודות שניתן להרץ על מחרוזת. אבל כיצד נוכל לדעת, לדוגמה, מה עשוה המתודה 'count'? כאן מגיעה לסייענו הפקציה 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, וזאת מכיוון שה-"el" מופיע פעם אחת בתוך "Hello world".

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

צירופי תווים מיוחדים ו-raw string

התו '\' הואתו מיוחד אשר קרי "escape character" והוא מאפשר ליצור מחרוזות עם צירופי תווים מיוחדים. שנות תווים, שם נשים אותם אחרי ה-'\' הם לא יודפסו כמו שהם, אלא יקבלו משמעות אחרת. אחד הצירופים הידועים ביותר הוא \\. אם נכתב ח\[בחרוזת וננסה להדפיס אותה נקבל... נס' זאת בעצמכם.

להלן טבלה של הצירופים המוחדים (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 תקין בשם c:\cyber\number1\b52. שמו לב איך התיקיה זו תודפס באופן "רגיל":

```
print 'c:\cyber\number1\b52'
```

```
c:\cyber
umber52
```

לא בדוק מה שרצינו שיודפס...

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

```
print 'c:\\cyber\\\\number1\\\\b52'
```

הכפלת אומרת לפיתון 'אנחנו באמת מתכוונים לשימן \'.'

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

```
print r'c:\cyber\number1\b52'
```

קלט קלט מהמשתמש – input

את הפונקציה `input` הכרנו בקצרה 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)
```

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

```
Please enater your name
Kalista
Hello Kalista
```

תרגיל – אבןיבי

זכירים את שפת הב"ת? כתבו קוד שקולט משפט (באנגלית) מהמשתמש ותרגמו אותו לשפת הב"ת. תזכורת: אחרי האותיות `so` צריך להוסיף `ה` ולהכפיל את האות. לדוגמה, עברו הקלט `ani ohev otach` וידפו:

`Abanibi obohebev obotabach`

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

```
if 'a' in my_str:
```



תרגיל מסכם – ז'אן ולז'אן (קרדייט: עומר רוזנבוים, שי סדובסקי)



כיתבו תוכנית שקוולתת מהמשתמש במספר בעל 5 ספרות ומדפייה:



- 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: בתרגול זה אנחנו נדרשים לבצע המרה בין סוגי טיפוסים שונים. זיכרו, שהפונקציה `input()` מחזירה מחרוזת של תווים. כולם אם המשמש הzin, 1234, תחזיר 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
```

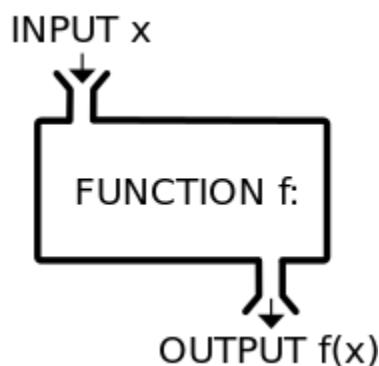
הערה: ניתן להניח שהמשמש הzin קלט תקין.

פרק 5 – פונקציות

כתיבה פונקציה בפייתן

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

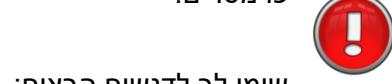
במילים אחרות פונקציה היא קטע קוד שיש לו קלט `input`, ופלט `output`.



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 עשו. לתייעוד בתחילת פונקציות יש שם מיוחד – `docstring`, קישור של `documentation string`, מחוץ לתיעוד. כתובים בתוך `docstring` שלושה סימני גרשימים כפולים רצופים, כמו בדוגמה. מדוע זה חשוב? מכיוון שם נעשה שימוש על הפונקציה, נקבל בחזרה את `docstring` שלה. דבר מעיל מאד 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 יחיד:

```
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 msg1, msg2
```

שימוש לב לאופן התייעוד, ש כולל גם הסבר אודוט הפרמטרים שמקבלת כל פונקציה. מודיע 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 `print_messages` מתבצעת העתקה של הארגומנט `mes` לתוך הפרמטר `msg1` ושל הארגומנט `sage` לתוך הפרמטר `msg2`.

```
print_messages(mes, sage)

        mes   sage
        ↓     ↓
def print_messages(msg1, msg2):
    """ Print two messages
Args:
    msg1 - a string
    msg2 - a string
"""
print msg1, msg2
```

המחשה של העברת ארגומנטים לפונקציה עם פרמטרים בעלי שמות שונים

`return`

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

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

```
def main():
    a = seven()
    print a
```

השורה `a = print` תגרום להדפסת המספר 7.

אפשר להחזיר יותר מערך אחד:

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



```
def main():
    var1, var2 = seven_eleven()
    print var1, var2
```

השורה `print var1, var2` תגרום להדפסת 7, 11.

תרגיל



- כתבו פונקציה אשר מקבלת שני ערכים ומוחזירה את המכפלה שלהם.

- כתבו פונקציה אשר מקבלת שני ערכים ומחזירה את המנה שלהם. זיכרו שהילוק 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()

נסו לעשות k print ותקבלו שהערך של k הוא None, או "כלום", והוא ערך ריק. זהוי מילה שמורה 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 שם ערך או משתנה. הכוונה לשורה שבה כתוב רק return.
- לפונקציה יש return None יש .return None

scope של משתנים

מה תבצע התוכנית הבאה?

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

```
def main():
    speak()
    print word
```

כפי שאלוי שמתם לב, 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 מכירה את המשתנה `word`?

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

נحدد את ההסבר ותוך כדי נבין את ההבדל בין משתנה גלובלי למשתנה מקומי, מקומי. נניח שכ כתבנו את הקוד הבא, שimeo לב למשתנה החדש `name`:

```

name = 'Shooki'

def speak():
    word = 'hi'
    print word, name

def main():
    speak()

```

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

נתקיים עוד שלב – כתת אנחנו מגדירים את המשתנה word פעמיים. אחת כגלובלי ואחת כлокלי... מה ידפיס הקוד הבא?

```

word = 'bye'

def speak():
    word = 'hi'
    print word

def main():
    speak()

```

ובכן, ידפס hi. מדוע? הרי אנחנו שימושו המשתנה גלובלי מוכר גם בתחום פונקצייה? נכון, אלא שבפונקציה speak אנחנו "דורסים" את המשתנה הגלובלי word עם משתנה לוקלי בעל אותו שם. כתת כאשר נפנה בתחום speak לממשנה word, הוא כבר לא יכיר את המשתנה הגלובלי, אלא רק את מה שהוגדר לוקלית.

ניסיין נוספת... מה ידפיאו הקוד הבא?

```
word = 'bye'
```

```
|def speak():
    word = 'hi'
)    print word
```

```
|def main():
    speak()
)    print word
```

ידפיאו:

```
hi
bye
```

מדוע? את ההדפסה של `hi` כבר הבנו. כאשר `speak` מסיימת את הריצה שלה, המשתנה המקומי `word` נמחק, וכעת קורה משהו מעניין – מסתבר שהמשתנה הגלובלי `word` לא נמחק, אלא פשוט נשמר בצד. לפיכך יש מידרג של עדיפויות: כאשר פונים למשתנה בתוך פונקציה, קודם כל פיקטן מחפש אם קיים משתנה מקומי זהה, ולאחר מכן אם קיים משתנה גלובלי. ההדפסה השנייה מתרכשת מתוך הפונקציה `main`, שמכירה רק את המשתנה הגלובלי `word`.

ניסיין אחרון... מה ידפיאו הקוד הבא?

```

word = 'love'

def speak():
    word += ' you'
    print word

def main():
    speak()
    print word

```

שגיאה! כדי שההתשובה לא תהיה קלה מדי, הורדנו את הקווים האדומים של PyCharm, אשר מסמנים שינוי שגיאה בקוד...

```

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 שימושה את ערכו של המשתנה בתוך פונקציה, כמו פעולה חיבור, פירוטן מניח של משתנה שלנו יש עותק מקומי והוא מסה לפעול עליו.

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

```

word = 'love'

def speak():
    global word
    word += ' you'
    print word

```

```

def main():
    speak()
    print word

```

בקבות הריצה יודפו:

```
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, כך:

```
word = 'love'
```

```
|def speak(word):
    word += ' you'
|    print word
```

```
|def main():
    speak(word)
|    print word
```

בקבות 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. הודפו פעמיים `you love`. כמובן, הפונקציה שינתה את ערכו של המשתנה. דמיינו שאתם כתובים את הפונקציה `main` ומתקנת אחר כתוב את הפונקציה `speak`. כתת תארו לעצמכם את ההפטעה, שהפונקציה שינתה ערך של משתנה בלי שהיא לכם מושג שהוא 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, כר:

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

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

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

```
|def main():
|    word = 'love'
|    word = speak(word)
|    print word
```

תרגילים

- כתבו פונקציה בשם `factorial` שמחזירה את התוצאה של $5!$ (5 עצרת). אין צורך להשתמש בברקורסיה.
- כתבו פונקציה בשם `beep` שמקבלת מחרוזת ומחזירה את המחרוזת ועוד `beep` בסופה.
- כתבו פונקציה בשם `sums_2nums` שמקבלת שני מספרים ומחזירה את המכפלת שלהם, או 0 אם התוצאה שלילית. שימו לב, אפשר לכתוב `return` יותר מפעם אחת בפונקציה.

פייטון מתחת למcosa המנווע (הרחבה)





בחלק זה נפרט שני נושאים הקשורים לפרק הלימוד עד כה:

- נמchiaш את עקרון התרגום של פיתון לשפת מכונה בזמן ריצה
- נכיר את הפונקציה `if` ואת האופרטור `is`
- נחקור כיצד פרמטרים מועברים לפונקציות

זכור פיתון היא שפת סקריפטים, ולכן לא מתבצעת קומpileציה לקוד. במקרים אחרות, כל שורת קוד שאנו נ כתבים מומרת לשפת מכונה רק כאשר מגע תורה של שורת הקוד להיות מorrectת. נמchiaש את העיקון הזה בעזרתו תוכנית קטנה.

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

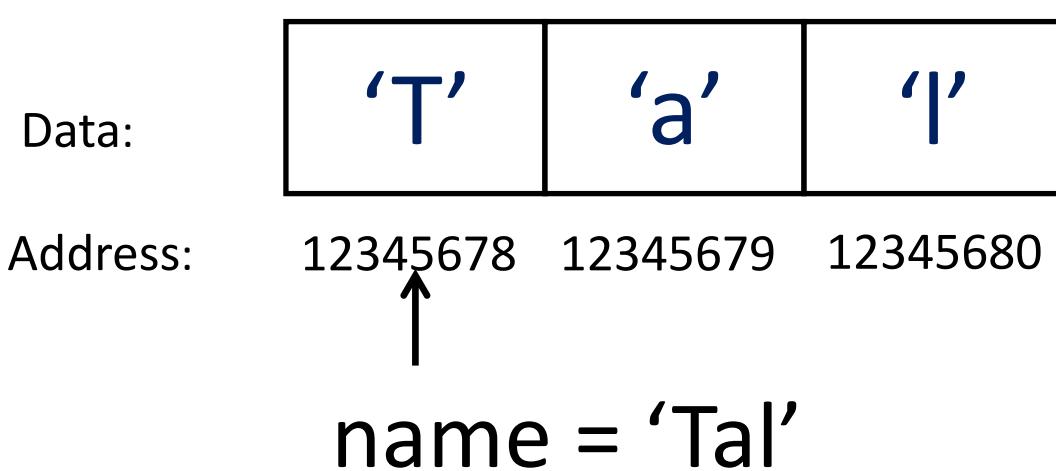
```
def main():
    check(1)
```

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

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

id, is

זכירנו המחשב מכיל את כל המשתנים שמדוברים בתוכנית שלנו. כל משתנה נמצא בכתובת מוגדרת, כמוyar כאשר אנחנו מגדירים משתנה הוא מקבל כתובת מתווך כתובות שמקצת לריצת התוכנית שלנו. עבור כל משתנה שמדובר בתוכנית פיתון, שם המשתנה משמש כדי לציין על כתובת בזיכרון. לדוגמה, אנחנו יכולים להגיד משתנה בשם `name` 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 12345680, 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 שזהו מספר מטיפוס `Long`, כלומר זה 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, גרמנו לו למעשה להציבו לאותה הכתובת בזיכרון שאליה מצביע המשתנה `name`. לאחר מכן, בדקנו שהערך של `name_copy` הוא גם כן 'Tal'. כפי שהיא הערך של `name`. בשלב האחרון בדקנו מה ה-`id` של `name_copy`. כפי שניתן לראות, הוא זהה ל-`id` של `name`.

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

`if result is True:`

...

if results is not True:

...

כעת 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 `True`, גם `False` וגם `None` יש `.id`.

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

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

העברת פרמטרים לפונקציה

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

Pass by value -

Pass by reference -

בשיטת `pass by value` מועבר לפונקציה **העתק** של הפרמטר. העתק נמצא על אזור בזיכרון שנקרא מחסנית, או `stack`, ומשמש פונקציות. דמיינו שהמורה מחזיק דף נייר שכתוב עליו המספר 10. המורה ניגש למוכנות הצילים 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 `pass by value`, היא למעשה לא משנה את ערך המשתנה עצמו, אלא העתק שלו. אי לכך, 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 עליו לגשת לזכרון בכתובת שנמסרה. דמיינו שכעת המורה שלנו מניח את הדף עם המספר 10 בתא שלו בחדר המורים, ובמקום לגשת למוכנות הצילים הוא ניגש למגנול 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 שביצעו התלמידים ישפיע על הדף המקורי שבידי המורה. שימוש לבשபעם לא נוצרו לדף עותקים, יש רק עותק מקור.

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

```

1 def func(x):
2     print 'id(x): {} x: {}'.format(id(x), x)
3     x = 'Bye'
4     print 'id(x): {} x: {}'.format(id(x), x)
5
6
7 def main():
8     x = 'Hi'
9     print id(x)
10    func(x)
11    print id(x)
```

בשורה 8 אנחנו קובעים מחזורת בשם `x`. בשורה 9 אנחנו מדפיסים את `(x)id` רק כדי שנוכל להשוות אותו לפני ואחרי הכניסה לפונקציה. בשורה 10 אנחנו קוראים לפונקציה וונבירים לה את `x` כפרמטר. בתוך הפונקציה אנחנו בודקים את `(x)id`, לאחר מכן משנים את ערכו של `x` וaz בודקים שוב את `(x)id`. לאחר היציאה מהפונקציה, בשורה 11, שוב בודקים את `(x)id` לראות אם הוא-

והנה מה שקיים:

```
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 ובתור הפונקציה הם זרים. האם 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*. בambilים 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 הקראיה 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 שכתבנו כתע נכוון למשתנים מסוימים, שנקראים *immutable*, שעד עכשוו עסוקנו 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 אודות פונקציות בפייתון. למדנו להגדיר פונקציה, להעביר לה פרמטרים וגם לקבל מהם ערכים. לאחר מכן רأינו שלמשתנים יש שוקס שבו הם מוגדרים, כלומר משתנה מוכר רק בפונקציה שבה הוא מוגדר. סקרנו אפשרות שונות של שימוש במשתנים גלובליים בתוך פונקציה והגענו למסקנה שתמיד כדאי להעביר משתנים כפרמטרים לפונקציה, ובכל מקרה עדיף להמנע משינוי של משתנים בתוך פונקציה בלי שהקובד שקורא לפונקציה מודיע לכך.

לאחר מכן, במסגרת ההרחבה, רأינו איך פיתון עובד "מתחת למcosa המנו", כיצד מועברים פרמטרים לפונקציה. כתע 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 ויכן מגדירים אותו? זהו אוסף של איברים, ועוד שאפשר להוסיף אליו איברים או להוציא ממנו איברים. כמו רשימת קניות – אפשר להוסיף לה מוצרים שברצוננו לרכוש או למחוק מהם מוצרים. מגדירים list באמצעות סוגרים מרובעים, כר:

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

בין כל שני איברים ישנו פסיק מיוחד. שימושו לב שבתוך ה-list יכול להיות אוסף של איברים מסוגים שונים. בדוגמה לעיל יש לנו float, string, int, boolean. אם נכתב print stam נקבל את הדפסה הבאה:

```
[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 לאת `for` צועדת 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` יקבל ערך מຕוך `stam`, לפי הסדר שבהם האיברים נמצאים בתחום `stam`. בסופו של דבר תוצאה ההדפסה תהיה זהה ל贤אצת ההדפסה איבר איבר.

דמיון נוסף בין `list` ובין `string`, היא היכולת לגשת לחלק מהאיברים באמצעות סוגרים מרובעים ונקודות – יכולות לשמש ב-slicing. לדוגמה, אם נרצה להשתמש באיברים של `stam` רק מהאיבר באינדקס 2 ואילך, נוכל לכתוב:

```
stam[2:]
```

יתר החוקים של slicing שראינו על מחזורות (כמו בירית מחדל לערכי התחלת וסיום, קפיצות וכו') תקפים גם לרשימות. לא נחזור עליהם, אך מומלץ לחזור ולקראן את החומר על חיתוך מחזורות ולנסות את הדברים על רשימה.



תרגיל

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

Mutable, immutable

כעת נبني 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. לדוגמה:

```
>>> 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 של 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, ושלא אחר הכתיבה הרשימה מכילה את הערך החדש. לעומת זאת מחרוזת אי אפשר לשנות – קיבלנו שגיאה. עכשו אפשר להבין את המושגים שכותרת. משהו שאפשר לשנות אותו נקרא mutable, מהמילה "מוטצייה". משהו 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

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

בעזרת ה-`id`, לפני ואחרי השינוי:

נבצע שינוי ברשימה, ונינוכח 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 מסוים נמצא בתוך רשימה. נניח שהגדירנו רשימת קניות בשם `shopping_list`. איך יוכל לדעת האם היא כבר כוללת `apples`? באמצעות שימוש במילה `חן`, אותה כבר הכרנו:

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

כאשר כותבים ביטוי מהצורה '`איבר חן רשימה`', התוצאה תהיה או `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 המשתנה, בנויגוד לפונקציות שמקבלות את שם המשתנה בסוגרים. נבין זאת יותר טוב כשנלמד תכונות מונחה עצמים OOP. השימוש ב-`pop` ו-`append` מתבצע כך:

`append`

הmethod `append` תמיד מוסיף איבר בסוף הרשימה. כאמור, אי אפשר להוסיף באמצעותה איבר לאמצע הרשימה. המethod מקבלת את האיבר שורצים להוסיף. לדוגמה:

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

pop

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

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

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

sort

מה לגבי מין רשימה? המethod `sort` מטפלת בכך. כל רשימה ש-`sort` תפעל עליה תהפוך להיות ממויינת. אם נשתמש ב-`sort` בלי להזין לתוכה פרמטרים, מספרים ימוינו לפי הגודל ומחרוזות ימוינו 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 לפחות, לפי סדר האלף בית... ובכן, הנה נחקור את `sort`. לפני כן, נזכיר כמה שלמדנו כאשר ביצענו חיתוך מחרוזות. אם יש לנו מחרוזת, אנחנו יכולים ליצור ממנה מחרוזת שונה באמצעות סוגרים מרובעים, אשר בתוכם נמצא אינדקס התחלת, אינדקס סיום ועל כמה אינדקסים מدلגים. 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 לשים בסוגרים המרובעים פרמטר אחד, שניים, או להשאיר פרמטר ריק לאחר נקודות. לדוגמה:

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

במקרה שאנו לא מזינים דילוג, מושג דילוג של 1. כלומר, יש ערך בሪית מהdoll – אם אנחנו לא מזינים ערך אחר, כאילו הזנו 1.

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

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

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

נראה שהוא מקבל שלושה פרמטרים. הפרמטר האחרון נקרא `reverse` וערך בריית המdoll שלו הוא `False` – כלומר אין "היפוך". ננסה את ערכו ל-`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 המין בסדר הפוך. מספרים מהגדול לקטן ומחרוזות הפוך מסדר הופעתן במילון.

נحمد, אבל מה אם נרצה לעשות מין יותר 'יצירתי'? לא רק מהקטן לגודל או מהגדול לקטן, אלא כל מין שנרצה? המетодה `sort` מקבלת בתור פרמטר את המפתח למין, פרמטר אשר נקרא `key`. בתור מפתח אנחנו יכולים לקבוע

איזו פונקציה שאנו רוצים. נראה דוגמה. יש לנו רשימה שכוללת כמה מחרוזות. אנחנו רוצים לבצע מיון לפי אורך המחרוזות. למשל, המחרוזת 'zz', שהאורך שלה הוא 2, צריכה להופיע לפני המחרוזת 'aaa' שהאורך שלה הוא 3 ואחרי המחרוזת 'c' שהאורך שלה הוא 1 בלבד. כידוע 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']
```

אנחנו יכולים גם להגיד פונקציה משלנו ולהעביר אותה בתור מפתח. שימו לב, שהפונקציה צריכה להחזיר ערך כלשהו, שלפיו יבוצע המיון, כמו שהפונקציה `len`מחזירה מספר שהוא אורך המחרוזת.

בצעו מיון של מחרוזת לפי התו האחרון במחרוזת. לדוגמה, `love` צריכה להיות לפני `cat` משום שהאות e מופיעה לפני התו t.



הקלט

```
benjamins = ['p.daddy', 'lil.kim', 'dr.dre', 'n.big']
```

ותוצאת ההדפסה שנתקבל:

```
['dr.dre', 'n.big', 'lil.kim', 'p.daddy']
```

split

לעתים נקבל מחרוזת ונרצה להפריד אותה למחוזות קטנות יותר ולשמור אותן בראשימה. הדוגמה הקלאסית היא מחרוזת שכוללת משפט, ואנחנו רצим להפריד אותה למיללים בודדות. לשם כך קיימת המתודה `split`. כאמור, פועלת על משתנים מטיפוס מחרוזת, אך כיוון שהיא מחזירה ראשימה נוכלCut להבין יותר טוב את אופן הפעולה שלה.

המתודה `split` מקבלת כפרמטר תו או מחרוזת שלפיהם תתבצע ההפרדה. כדי להבין את רעיון ההפרדה, ניקח לדוגמה מחרוזת שכוללת שמות של מספר סרטים:

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

נרצה ליצור ראשימה בה כל איבר יהיה שם של סרט. למשלנו הראשימה מכילה את התו '#' בתורתו מפריד בין שמות הסרטים. لكن נקרא `split` עם פרמטר '#':

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

וקיבלנו ראשימה:

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

لامתודה `split` יש גם ערך ברירת מחדל, שהוא סימן רווח. במקרים אחרות, אם לא נעביר `split` שם פרמטר, כל הרוחים במחוזת יוסרו והמחוזות שבין הרוחים יוכנסו לראשימה. לדוגמה:

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

הפכנו את `rule` לראשימה. אם נדפיס את ששת האיברים הראשונים בראשימה נקבל:

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

join

זהי הפעולה הפוכה ל-`split`. יש לנו ראשימה של מחרוזות ואנחנו רצים לחבר אותן יחד למחרוזת אחת. צורת הכתיבה כאן היא גם הפוכה ל-`split` – ראשית יבוא התו המפריד בין המחרוזות השונות (הגינוי שזה יהיה סימן רווח), ולאחר מכן נקודה ו-`join` עם שם הראשימה בסוגרים. כך:

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

כעת אם נדפס את rule נקבל את המחרוזת המקורית:

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

Tuple

שימוש לב, בדוגמאות הbabot ישנו פונקציית tuple, התעלמו מבוזה הבינואה עיקריה העומדת מאחוריה. נכיר טיפוס נוסף של משתנה בפייתון – tuple. מגדירים tuple באמצעות סוגרים עגולים, כך:

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

הטיפוס tuple הוא כמו list, אבל immutable. כלומר, אין אפשרות לשנות את הערכים שבו. לשם מה זה שימושי? ובכן, tuple הוא דרך להחזיר ערכים רבים מפונקציה. לדוגמה הפונקציה הבא 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()
```

cut כל אחד מהמשתנים 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, מדוע `tuple`

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 שהייו 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 בדוק את כמות המקומות שהגדכנו. כלומר, אם אנחנו יודעים שאנו צריכים להוסיף איברים, הטיפוס `tuple` הוא חסוני

ויתר במקומם בזיכרון. כאשר אנחנו מוחזרים ערכים מפונקציה, אנחנו יודעים כמה ערכים החזרנו, אך השימוש בוtuple הוא מתבקש.

סיכום

בפרק זה למדנו אודות שני טיפוסי משתנים שימושיים – list ו-tuple. סקרנו מתודות ופונקציות שימושיות של list: בדיקה אם איבר נמצא ברשימה, הוספה 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 שרישמה היא משתנה mutable – ניתן לשינוי – בעוד מחרוזת היא immutable. לסיום רأינו כיצד tuple משמש להחזרת ערכים מרובים מפונקציה.

פרק 7 – כתיבת קוד נכונה

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

מהו בכלל קוד איות? בפרק זה נתמקד בכמה כללים:

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 לקרוס כתוצאה לכך. לדוגמה, כתבנו פונקציה שמחשבת את השורש של מספר כלשהו שהמשתמש מקליד. המשמש הקליד מספר שלילי. הפונקציה צריכה לא לקרוס. גם אם המשמש הכנס ערף שאינו מספר, הפונקציה צריכה לא לקרוס.

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 יצטרך 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 ג אלה 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` אינו שם טוב למשתנה – מי שקורא אותו אין מושג מה הוא שומר בתוכו. לעומת זאת, `salary` (משכורת) הוא משתנה 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 `check_valid_input`. מה יותר קרייא?

ה. הקוד צריך להיות מתוועד. כלומר, יש לכתוב docstring בתחילת כל פונקציה ויש להוסיף תייעוד שמסביר מה עשינו, אך יש להמנע מתייעוד יתר, שכן מצין את המובן שלנו והוא מוסיף מידע חשוב לקורא. דוגמה לתייעוד יתר כזה:

```
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 לקובננציות, או בעברית "מוסכמות". כאן הסיבה היא פשוט נוחות של מי שמנסה לקרוא את הקוד שלכם. לדוגמה, שמות של קבועים ייכתבו באותיות גדולות. כך, אם נקרא קוד של מישוא וונמצא שם אותיות גדולות, קיבלנו מידע חשוב בלי מאמץ.

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

- כתיבת קוד פיתון לפי קובננציות PEP8

- חלוקה נכון של קוד לפונקציות

- בדיקת תקיןות קוד ומקרי קצה באמצעות assert

PEP8

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

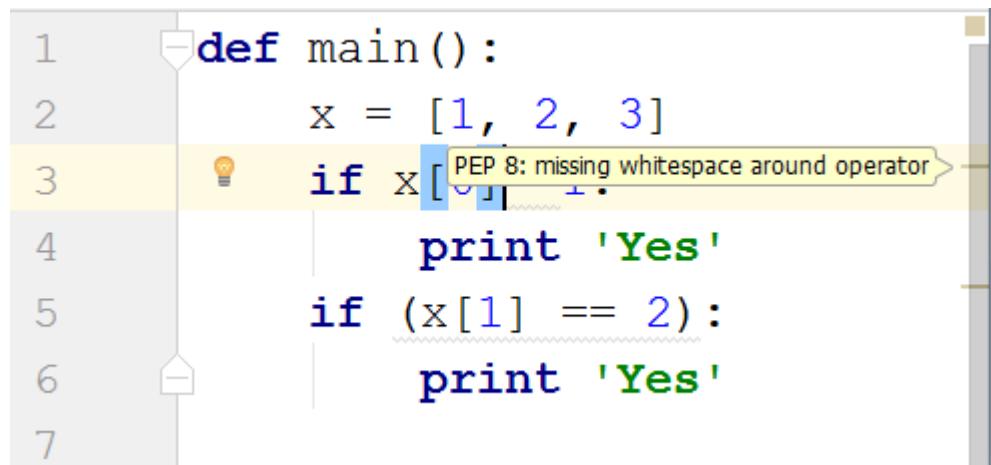
נתעכבר על השימוש ב-*PyCharm* על מנת לאתר טוויות PEP8 (כלומר, קוד שלא עומד בקובננציות שהוגדרו לפי PEP8) ולתקן אותן. שימושו לב לקטע הקוד הבא:

```

1 def main():
2     x = [1, 2, 3]
3     if x[0]==1:
4         print 'Yes'
5     if (x[1] == 2):
6         print 'Yes'
7

```

ניתן לראות שהריבוע בצד ימין מעלה הינו בצבע צהבהב, לא יירוק אבל גם לא אדום. הדבר מראה לנו שהקוד ירץ בצורה תקינה, אבל יש בו טעויות PEP8. קל למצוא את הטעויות – האם אתם מבחינים בשני הקווים הצהבהבים שמתוחת לריבוע? נعمוד על אחד מהם:



The screenshot shows a code editor with Python code. The third line, which contains the assignment operator '==' without a space, has a yellow background and a small lightbulb icon. A tooltip above the cursor says "PEP 8: missing whitespace around operator". The rest of the code is in its original state.

```

1 def main():
2     x = [1, 2, 3]
3     if x[0]==1:
4         print 'Yes'
5     if (x[1] == 2):
6         print 'Yes'
7

```

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

יתכן שתשים לב גם לסימן הנוראה הצהובה שדוילקט לצד שורת הקוד הביעית. לחיצה שמאלית עלייה תפתח לנו תפריט, ואם נבחר באפשרות **Reformat file** הבעיה תתוקן לבד!

```

1 def main():
2     x = [1, 2, 3]
3     if x[0]==1:
4         'Yes'
5     = 2):
6         'Yes'
7

```

A screenshot of PyCharm's code editor showing a context menu for a PEP8 violation. The menu items are:

- Reformat file
- Edit inspection profile setting
- Ignore errors like this
- Flip '=='
- Negate '==' to '!='

אפשרות נוספת לגלוות בעיות PEP8 בקוד היא באמצעות סימן אفور דק מתחת לתוכים הבעייתיים. דבר זה מסיע לנו להמנע מביעיות כבר בשלב הכתיבה.

תרגיל

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

```

1 def main():
2     x = [1, 2, 3]
3     if x[0]==1:
4         print 'Yes'
5     if (x[1] == 2):
6         print 'Yes'
7

```

לסייעם 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 לזכור 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 xrange(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). שנית, אם נרצה שהתוכנית שלנו תדפיס 11 פעמים ולא 10, علينا לחפש בקוד את המספר 10 ולשנות אותו ל-11. נאמר שהמספר 10 חוצר על עצמו מספר פעמים בקוד – מכיוון שאנו רוצים להדפיס כמה דברים שונים, וכל אחד מהם – עשר פעמים. במקרה זה נדרש לשנות את המספר 10 שוב ושוב. עם זאת, יכול להיות שבחלק מהמקרים המספר 10 יתיחס למקרה אחר – למשל במקרים מסוימים שאנו רוצים לקרוא מידע מהמשתמש, יוכל להיות שדוקא שם נרצה לשומר על המספר 10 כשלעצמו. ככל שהתוכניות שלנו יהיו מורכבות יותר, כך הזמן שנצרך להקדיש לזה יגדל.

האפשרות הטובה יותר היא שימוש בקבועים. לדוגמה:

```
TIMES_TO_PRINT = 10
```

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

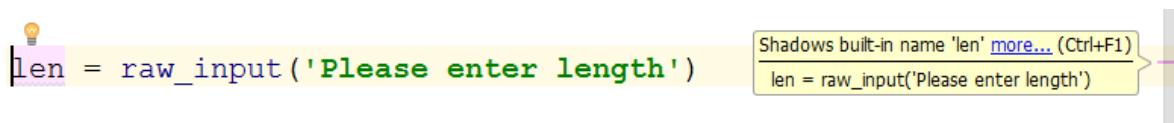
הגדרנו בתחילת התוכנית קבוע (שמו לב לאותיות הגדולות!), שרק מカリית השם שלו אנחנו כבר ידעים מה הוא עושה.-cut אם נרצה להדפיס מספר שונה של פעמים, כל שנצטרך לעשות הוא לשנות את הקבוע, סימנו.

שימוש לב שפיטון לא מודא עבורנו שהקובע שלנו לא ישנה. כמובן, עדין ניתן לבצע שינוי לערך הקבוע שלנו:

```
TIMES_TO_PRINT = 5
```

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

ג. לא לדחוס שמות מבנים בפייתון: לפיתון יש שמות מבנים – names-in-built – שהפרשן שלו מכיר גם בלי, שהגדכנו אותם. נניח שאנו ממעוניינים לקלוט מהמשתמש מספר שמייצג אורך של משה. לדוגמה, אורך של ספר בעמודים, או אורך של משחק כדורי בדקות. מאוד מפתחה להגדיר את המשתנה בשם `len` – קיצור של `length`. הבעייה היא שהשם '`len`' הוא שם מובנה בפייתון, כלומר יש לפיתון כבר פונקציה שנקראת `len` והיא יודעת להחזיר אורך של פרמטרים שהוא מקבלת. מה יקרה אם נדרוס אותה? כמובן, פרט לכך שייתר לא יוכל להשתמש בה... לא ל 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 וניתן בקלות להטאה למשימות אחרות. מדוע זו מטרתנו? משום שלמעט מקרים נדירים, בהן המערכת שלנו רצתה ב מגבלות מסוימים, הרבה יותר סביר שיש לנו משבבי מחשב רבים אך מאידך אנחנו מוגבלים בכמות הזמן שיש לנו לטובת כתיבת קוד, או שהמחיר של באג במערכת הוא גבוה מאד, או שאנו צריכים לכתוב קוד בצורה שצוות או קהילת מתקנים יכולים להבין ולהשתמש בו. מכל הסיבות האלה, קוד איקוטי הוא קוד שאין בו באגים, שהוא קל להבנה ואפשר ברקלות להשתמש בו למשימות נוספות.

הבה ניקח משימה פשוטה יחסית ונדגים עליה צורות שונות של כתיבת קוד. בתור משימה ניקח את הבעייה הבאה: ברצוננו לקבל מהמשתמש רשימה של מספרים, ולבדוק 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 לביעיה התיכונוטית). 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 מתקנית. כלומר 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.

פתרונות מודרך

הדרך הראשתונה לפתור את התרגיל היא פשוט לקלוט את המספרים אחד אחריו השני, ובכל מספר שנקלט לבדוק אם מתקיים התנאי שהמספר האמצעי מקיים את התנאי שהוגדר (במקרה זה, ממוצע המספר שלפניו והמספר שאחריו). 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:

```

1     __author__ = 'Barak'
2     # First attempt to solve the problem -
3     # Check if each number in a list is the
4     # average of the prior and next numbers
5
6

```

```

7     def main():
8         index = 0
9         ok = True
10        while True:
11            user_input = raw_input('Enter num, STOP to quit ')
12            if user_input == 'STOP':
13                break
14            else:
15                user_input = int(user_input)
16                if index == 0:
17                    before = user_input
18                elif index == 1:
19                    middle = user_input
20                else:
21                    avg = float(before + user_input) / 2
22                    if middle != avg:
23                        ok = False
24                        break
25                    before = middle
26                    middle = user_input
27                    index += 1
28            if ok:
29                print 'List is good'
30            else:
31                print 'List is not good'
32

```

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

ראשית, ספר הלימוד בכוונה אינו מסביר כיצד עובד הקוד. נסו להבין בעצמכם איך הקוד מבצע את המשימה שהוא-Amor לבצע. דמיינו שאתם מתכוונים במצוות וחברכם למצוות השאיר לכם את הקוד הזה ויצא לחופשה. לבתו תצליחו להבין את הקוד, אבל המשימה צפiosa לגזול ממקם זמן רב יחסית לקוד לא ארוך. שנית, אחת הסיבות המרכזיות שהקוד הזה קשה לקרואיה (וגם לכתיבתה ולדיבוגו) הוא כמות התנאים שנמצאים בתוך תנאים. שימו לב לשורות 23 ו-24 – הן נמצאות 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 מתייחס גם בזמן שלוקח לדבג את הקוד. במילים אחרות, אם יש לכם באג בקוד שנראה כך, צפו לערב ארוך מול המחשב...

נססה 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 קטיעי הקוד – יהיה לנו קטיעת קוד שאחראי לקלילת כל המספרים (שורות 8 עד 14) וקטיעת קוד אחר שאחראי לבדוק אם התנאי מתקיים על סדרת המספרים (שורות 15 עד 24). בכך, אפשר לטעון שהקוד הקודם ביצע את המשימה בצורה יותר מהירה, אבל כזכור המטרה שלנו אינה להציג את מהירות הריצה של הקוד אלא לצמצם את כמות הזמן שלוקח לכתוב, לדבג ולתחזק את הקוד. שימו לב כיצד עצם החלוקה למשימות שונות מוריידה את כמות התנאים בתוך תנאים. יש לנו כרגע לכל היוטר 3 רמות אינדנטציה, במקום 5:

```

1   __author__ = 'Barak'
2   # Second attempt to solve the problem -
3   # Check if each number in a list is the
4   # average of the prior and next numbers
5
6
7   def main():
8       nums_list = []
9       while True:
10           user_input = raw_input('Enter num, STOP to quit ')
11           if user_input == 'STOP':
12               break
13           else:
14               nums_list.append(int(user_input))
15       ok = True
16
17       for index in xrange(1, len(nums_list)-1):
18           avg = float(nums_list[index-1] + nums_list[index+1])/2
19           if nums_list[index] != avg:
20               ok = False
21               break
22       if ok:
23           print 'List is good'
24       else:
25           print 'List is not good'
26
27   if __name__ == '__main__':
28       main()

```

נحمد. ועדיין זה אינו קוד מוצלח. מדוע? ראשית, הקוד אינו מתועד. 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 מתוך הקוד מה מתבצע, אולם ככל שנרצה לבצע שימוש מסווגות יותר כך יחסר לנו יותר מידע. שנית, קשה להשתמש בקוד זהה למשימות אחרות. נסביר: נניח שחבר שעבוד איתנו נתקל גם הוא בעיה תיועוד. הוא צריך לבצע על רשימת מספרים ולבדוק אם מתקיימת הטענות הנ"ל. לחבומו לעבודה כבר יש תוכנית בה הוא צריך לבצע על רשימת המספרים ולבדוק אם מתקיימת הטענות הנ"ל. מה שהחבר יצרך לבצע אחרת שדווגת לקליטת המספרים, כך שלא מתאים לו להעתיק את כל התוכנית שלנו. מה שהחבר יצרך להעתיק הוא להעתיק את קטע הקוד שלנו – שורות 15 עד 24 – ולהתאים את שמות המשתנים לשמות בתוכנית שלו. מסובך! בשביל זה יש פונקציות. שלישיית, האם אתם מזהים אפשרות כלשהי שהקוד שלנו יקרוס תוך כדי ריצה? מה לדעתכם יקרה אם המשתמש לא גענה להוראות שלנו והזין ערכים שאינם ספורות? בשורה 14, מתבצעת המרה של קלט המשתמש ממחרוזת ל-int. אם המחרוזת אינה ברת המרה ל-int (לדוגמא, נסו להמיר את 'table'

ל-int...) אז הקוד יקרוס תור כד' ריצה. אנחנו צריכים להבטיח שהקוד שלנו לא יקרוס גם אם המשתמש מאתגר אותו.

לטיכום, אנחנו צריכים להכין 3 שיפורים בקוד:

- **תיעוד**
- **הוספת פונקציות שיבצעו קטעי קוד חשובים**
- **בדיקה שלכל המשמש תקין ולא תיגרם בשום אופן קריסה תור כד' ריצה**

להלן הגרסה השלישית של הקוד, שטיפלה בשיפורים הנדרשים. שימו לב, ברור שהקוד יהיה ארוך יותר, אך בוואנו נראה אםCut קיל' יותר להבין מה הקוד עושה. כיצד לקרוא את הקוד ולהבין אותו במינימום זמן? מומלץ להתחיל מפונקציית `main` ולנסות להבין מה היא עושה. כדי להבין זאת, תוכלן לעזור בשמות הפונקציות – לעתים אפשר להבין מה פונקציה עשויה בלי לקרוא אותה כלל. לאחר מכן מכוון אל הפונקציות, קיראו קודם כל את ה-

הגרסאות שלהן ורק לאחר מכן את הפונקציות עצמן. השוו בין כמות הזמן שלוקחת לכלם הקראיהCut לעומת docstring שלהן ורתק לאחר מכן את הפונקציות עצמן.

```

1     __author__ = 'Barak'
2     # Third attempt to solve the problem -
3     # Check if each number in a list is the
4     # average of the prior and next numbers
5
6     END_INPUT = 'STOP'
7     MIN_LIST_LENGTH = 3
8
9
10    def receive_user_inputs(ending_value):
11        """ Receive multiple inputs from the user and store in list
12        Args:
13            ending_value - once the user enters this value, stop receiving
14        Return value:
15            list of all received values
16        """
17        input_list = []
18        while True:
19            user_input = raw_input('Enter num, {} to quit '.format(ending_value))
20            if user_input == ending_value:
21                break
22            else:
23                input_list.append(user_input)
24        return input_list
25
26

```

הגרסאות הקודמות. מה יותר מהיר?

```

27     def list_is_nums(input_list):
28         """ Check if all the elements of a list are nums (int or float)
29         Args:
30             input_list - the list to be checked
31         Return value:
32             True / False
33         """
34         for element in input_list:
35             if not element.isdigit():
36                 return False
37         return True
38
39
40     def list_is_average(input_list):
41         """ Check if a list is according to the condition
42             that each element is the average of the adjacent elements
43             like: 1, 3, 5, 7
44         Args:
45             input_list - the list to be tested
46         Return value:
47             True / False
48         """
49         # Convert list from str elements to floats
50         nums = []
51         for element in input_list:
52             nums.append(float(element))
53         # Check if the nums are all averages
54         for index in xrange(1, len(nums)-1):
55             avg = float(nums[index-1] + nums[index+1])/2
56             if nums[index] != avg:
57                 return False
58         return True
59
60
61     def main():
62         input_list = receive_user_inputs(END_INPUT)
63         if len(input_list) >= MIN_LIST_LENGTH and list_is_nums(input_list):
64             if list_is_average(input_list):
65                 print 'List is good'
66             else:
67                 print 'List is not good'
68
69     if __name__ == '__main__':
70         main()

```

דבר נוסף שהרוויחנו מכתיבת הקוד באופן זהה, הוא שnochל להשתמש בfonkцийות גם בתוכניות אחרות. זאת ממש שהfonקציות לא עושות שימוש במסתנים גלובליים – כל מה שהן צרכו מועבר להן בתור פרמטרים. התיעוד המפורט עוזר לנו להבין כיצד לקרו לכל פונקציה כדי להשתמש בה. דבר זה יחסור לנו גם זמן כתיבת עתידי.

assert

מה דעתכם, האם יש עוד מה לשפר בגרסה الأخيرة של הקוד? נשאל את עצמנו – איך אנחנו יודעים שהפונקציות שכתבנו אכן עובדות? כמובן, יכול להיות שהן עובדות ברוב במקרים, אך במקרה קצה הן קורסוטות. דמיינו פונקציה שמבצעת חילוק – יכול להיות שהיא עבדת כמעט תמיד, אך קורסוט אם המכנה הוא אפס....

בנוסף, יכול להיות מצב בו הפקציה שלנו עובדת, גם במקרים קצה, אך ביצענו בה שיפור. השיפור גורם לכך שבמקרים קצה הפקציה שלנו כבר לא עבדה היטב, או קורסת.

נרצה למצוא דרך לבדוק במדויק את הבדיקה שלבנו, וכן נזכיר את הדרך פשוטה ביותר – `assert`.

כדי להשתמש ב-`assert`, כותבים `assert`, לאחר מכן ביטוי כלשהו, שיכל להיות `True` או `False`. במקרה שלנו נכתוב שם של פונקציה, לאחר מכן סוגרים עם קלט לפונקציה, ולאחר מכן את הפלט הצפוי מהפונקציה עברו הקלט הנ"ל. ניקח לדוגמה את הפונקציה `sums_is_list`, אשר מקבלת רשימה ובודקת אם כל האיברים בה הינם מחרוזות שנitinoot להמרה למספרים. אם כן, `sums_is_list` מחזירה `True`, אחרת `False`. נסיף לתוכנית שלנו:

```
|def main():
    assert list_is_nums(['1', '2', '13', '14']) is True
    assert list_is_nums(['10', '11', 'hi']) is False
```

בדוגמה אנחנו רואים שני קלטי בדיקה שמעברים לפונקציה `list_is_nums`. הקלט הראשון הוא קלט תקין – ארבע מחרוזות שכל אחת מהן ניתנת להמרה למספר. אך ה-`assert` בודק האם ערך החזרה הוא `True`. הקלט השני מכיל את המחרוזת '`hi`', שסבירן אינה ניתנת להמרה למספר. ה-`assert` מודיע שבמקרה זה ערך החזרה מהפונקציה הוא `False`.

החלק המעניין ב-`assert` הוא להכניס קלטים מיוחדים, על מנת לבדוק מקרים קצח. לדוגמה, מה יקרה אם הפונקציה `sum_is_nums` מקבל רשימה ריקה? במקרה זה נרצה לוודא שיווצר לנו `False`, לא `Cr?` נבדוק:

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

בשלב הרצת התוכנית קיבל את הפלט הבא:

Traceback (most recent call last):

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

```
AssertionError
```

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

נוסיף לקוד שלנו `assert`'ים נוספים, במטרה לבדוק את הפונקציות. הנה ה-`main` שלנו לאחר התוספות:

```
61 def main():
62     assert list_is_nums(['1', '2', '13', '14']) is True
63     assert list_is_nums(['10', '11', 'hi']) is False
64     assert list_is_nums([]) is False
65     assert list_is_average(['1', '3', '5', '7']) is True
66     assert list_is_average(['1', '2', '4', '7']) is False
67     input_list = receive_user_inputs(END_INPUT)
68     if len(input_list) >= MIN_LIST_LENGTH and list_is_nums(input_list):
69         if list_is_average(input_list):
70             print 'List is good'
71         else:
72             print 'List is not good'
```

מספר דגשים:



- את הפונקציה `receive_user_input` אנחנו לא בודקים עם `assert`'ים. מדוע? מכיוון שהפונקציה הזאת דורשת קלט משתמש. המשתמש לא צריך להיות מודע לכך שהתוכנה שלנו כוללת בדיקות. הרי אין זה הגיוני שנבקש ממנו להשתמש להזין קלטי בדיקה בכל פעם שהוא מרים את התוכנית שלנו...

- הפונקציה `list_is_average` לא כוללת `assert` עם רshima ריקה. זאת מכיוון שם הגענו לפונקציה הזאת, זה אומר שעברנו כבר את הבדיקה שאריך הרשימה גדול-מ-MIN (קבוע ערכו 3). צריך להוסיף לтиיעוד של `list_is_average` שהיא מקבלת רשימה שאורכה 3 לפחות, כדי שמי שישתמש בה בעתיד יוכל 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 מדפיס

מה שצירר לפि הערך שהוחזר. בדיקן כמו בדוגמה, בה הפונקציה `list_is_average` החזירה רק `/ True` וההדפסה בוצעה בשורות הקוד שאחרי הקראיה לפונקציה `False`.

זהו! סיימנו את הדין ממשימה שהוצאה בתחלת הפרק. עברנו ארבע גרסאות קוד שונות והגענו לקוד אליו שאפנו – עובד, נוכן לקריאה ובודק.

עד כה ראיינו שימוש ב-`assert` לבדיקת פונקציות שמחזירות רק `True` או `False`. כמובן שאפשר להשתמש בו assert לבדיקת כל פונקציה. לשם המחשה, נגדיר פונקציה פשוטה שמבצעת חלוקה בין שני מספרים:

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

בתוך התחלתה נבדוק שהפונקציה שלנו עובדת היטב היטב במקרים ה"רגילים":

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

פעולות `assert` הראשונה בודקת אם התוצאה היא צפויה. פעולה `assert` השנייה כוללת דבר נוסף – הודעת שאינה שתודפס במקרה שהערך שיתקבל לא יהיה שווה לערך צפויה. במקרה זה יודפס `'Not the expected result'`, אך כמובן שאפשר להציג כל הودעה. מומלץ כמובן שההודעה תכלול מידע מידע>About השגיאה, כך שמי שמריץ יוכל בקלות לדבג ולמצוא את מקור הבעיה.

כעת תורכם – אילו עוד פעולות `assert` כדאי לעשות על `div_my`? נסו לחשב על מקרים קצה.

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

בנוסף, מה יקרה אם נubby ל-`div_my` ערכים שאינם מספרים? גם כן, היינו רוצים לקבל בחזרה מהפונקציה הודעה כגון `'Parameters are not numbers'`. בואפן זה מי שקורא לפונקציה עם ערכים לא נכונים לא יגרום לריסוק התוכנית.

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

תרגיל מסכם – Nu **Deja Vu** (קרדייט: 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 להכניס 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 בשום אופן. הקפידו על כל הדברים שלמדו בפרק זה – חלוקה נכונה של הקוד לפונקציות, שימוש ב-PEP8, תיעוד ובדיקת פונקציות על ידי `assert`.

סיכום

פרק זה הتمיקד בשדרוג יכולות התוכנות שלנו. התחלנו מנושא שנראה טכני למדי במבט ראשון – שמירה על קונבנציות לפי PEP8 – אבל ראיינו את החשיבות של נושא זה לצטיבת קוד קרי, שתוכננים אחרים יכולים להבין בקלות ולעשות בו שימוש.

לאחר מכון התמודדנו עם אחד המחסומים המרכזיים שעומדים בפני מתכנתים מתחילה: כתיבת תוכנית שלא רק מבצעת את מה שהוא צריך לבצע, אלא גם מחולקת למשימות שבוצעות כל אחת על ידי פונקציה אחרת. כתיבת קוד בדרך זו היא הדרך הארכית אבל המהירה. הקוד שלנו גם יותר קל לדיבוג וגם יותר קל לשימוש חוזר.

לבסוף ראיינו איך `assert` עוזר לנו לבדוק שהקוד שלנו תקין ועובד היטב. למדנו שכאשר כתבים 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, העברת פרמטרים לסקרייפטים – יכולת להריץ סкриיפט עם מידע שיגרום לסקרייפט לזרע בצוורה 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 מחוץ, כאשר מומלץ לשם לפני הטיון `z`, שכי' שלמדו מסמן מחזורת `raw`, כך שם הקובץ יוזן כמו שהוא ולא תבצע המרה לתווים מיוחדים. חייבים להעביר לפונקציה גם את אופן פתיחת הקובץ, לדוגמה אנחנו יכולים לפתוח קובץ לקריאה או לפתוח קובץ לכתיבה. אופן הפתיחה נקרא `mode`. נזכיר חלקי מה-`mode`'ים השונים (תיאור 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 נפתח כך 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 נוכל 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:

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

מהו סוג האובייקט שמחזירה הפונקציה `open`? על מנת לגלות, נוכל להשתמש בפונקציה `type` המוכרת לנו. נסו לכתוב (`input_file` – מה קיבלת?

קריאה מקובץ

הmethodה `read` פועלת על אובייקטים מסווג `file`. בטור ברירת מחדל, המתודה קוראת את כל הקובץ ושמירתו אותו בתוך מחזוזת בתוך משתנה שהוגדר על ידי המתכנת. לדוגמה:

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

תוצאה פקודת ה 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 ופשטן. החסרון של שיטה זו היא שכל הקובץ נקרא בבת אחת לתוך המשתנה שהגדנו. אם הקובץ גדול מאוד – זה בעייתי, כיוון שייגרם עומס גדול על הזיכרון וכתוואה מכך הקריאה מהקובץ תהיה איטית. לכן מומלץ להשתמש בשיטה קצרה יותר כדי לקרוא קובץ, שורה אחר שורה.

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

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

מה יקרה אם הגענו לסוף הקובץ? במקרה זה הערך שנתקבל מ-`readline` יהיה "", כלומר מחזוזת ריקה. דוגמה לקטע קוד קצר שמדפיס את הקובץ שורה אחר שורה:

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

שימוש לבן לב שבסוף השורה האחרונה יש לנו פסיק. הפסיק מסמן שלא להוסיף ירידת שורה בסוף הדפסה. הסיבה היא שבקובץ הטקסט מילא יש ירידת שורה בסוף כל שורה, ואילו לא היינו שמים פסיק היה רוח של שורה נוספת בין כל שתי שורות מודפסות.

עקב השימושות של הדפסת שורה אחר שורה, פיתון מאפשר לנו להשתמש בולולאת `for` ורגילה עם איטרטור. בכל איטרציה מתבצעת למעשה קריאה של שורה אחת. כך, הקוד שלנו יוכל לכתיבה מקוצרת ונחמדה:

```
for line in input_file:
    print line,
```

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

בנוסף, שימוש לבן לב שבחרנו בשם `line` כדי לייצג כל שורה – כך ברור מה המשתנה זהה כולל בכל קריאה. כך יותר לקרוא קוד, במיוחד אם הוא ארוך, ככל שמות המשתנים הם בעלי משמעות. זהו אחד מעקרונות כתיבת הקוד הנכון אותו הזכרנו בפרק הקודם.

כתיבה לקובץ

ראשית אנחנו צריכים לפתח את הקובץ לכתיבה (בנהנזה שהוא סגור – מיד נראה איך סגורים קובץ). כיוון שהקובץ `dear_prudence.txt` כבר מכיל מידע, נרצה לפתח אותו ב-`mode` של `append`. לאחר מכן נשתמש בMETHOD write על מנת לכתוב מידע לתוך הקובץ:

```
input_file = open(r'c:\python\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 יסגר אוטומטית ברגע שתסתה'ם ריצת התוכנית שלנו, אבל אי סגירה של קובץ יכולה לגרום לתוכנית שלנו להתנהג בצורה לא צפופה וקשה מאוד לדיבוג. נמחיש על ידי דוגמה. התוכנית הבאה קוראת לפונקציה שפותחת קובץ לכתיבה בלבד לסגור אותו, וכך להמיחס שזה אינו תכונת נכון, הפונקציה קוריה `open_without_closing`. הפונקציה משנה את תוכן הקובץ. לאחר מכן נפתח אותו קובץ שוב, הפעם לקריאה. המצביעים לקובץ נקראים כאן `fd`, קיצור של `file descriptor`.

```

FILENAME = r'c:\python\dear_prudence.txt'

def open_without_closing(filename):
    fd1 = open(filename, 'a')
    fd1.write('Dear Prudence open up your eyes\n')

def main():
    open_without_closing(FILENAME)
    fd2 = open(FILENAME, 'r')
    for line in fd2:
        print line,

```

מה לדעתם 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 שכדי תמיד לסגור קבצים אחרי שימושינו בהם. כדי לעשות זאת משתמשים בMETHOD close. פשוט כך:

```

fd1 = open(filename, 'a')
fd1.write('Dear Prudence open up your eyes\n')
fd1.close()

```

כעת, ההדפסה שנבצע מתוך פונקציית `chomd` תדפיס כמו שציריך גם את השורה الأخيرة שהוספה.

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

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

לאחר הפקודה `with`, נכתב `open` עם הפרמטרים הרגילים. לאחר מכן, נוסיף `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 יישאר פתוח רק כל עוד אנחנו נמצאים בבלוק של `with`. ברגע שהבלוק יגמר, יסגר הקובץ אוטומטית.

לשימוש ב-`open` `with` יתרון נוסף: נניח שהשתמשנו ב-`close`, אבל לפני שפינו הגיעו ל-`close` הוא נתקל בשגיאה והתוכנית הפסיקה לרווח עט שגיאה. כתוצאה לכך הקובץ שלנו נותר פתוח, למרות שבתוכנית הורינה לסגור אותו. ההוראה `with` גורמת לכך שתבוצע סגירה של הקובץ לפני שהתוכנית מפסיקה לרווח ומחזירה שגיאה. כך אנחנו יכולים להיות בטוחים שהקובץ שלנו נסגר בכל מקרה. אין עובדת `with` ואין היא מצליחה לסגור את הקובץ למרות שארעה שגיאה בדרך? על כך – כשנלמד exceptions.

עד כאן Learnedנו כיצד להשתמש בקבצים – כיצד לקרוא מהם, וכיצד לאליהם ולהוסיף להם מידע. כמו כן Learnedנו את החשיבות שבסגירת הקובץ בתום השימוש בו. כעת, נעבור לחלק השני של פרק זה.

תרגיל – מכונת שכפול

צרו באמצעות סייר חלונות שני קבצי טקסט, האחד ריק והשני כולל טקסט כלשהו. כתבו סקריפט אשר מוגדרים בו שמות שני קבצים. הסקריפט יעתיק את הטקסט אל הקובץ הריק, וכך לאחר סיום הרכזה הקבצים ייכלו את אותו טקסט.

קבלת פרמטרים לתוכנית

דמיינו שאתם משתמשים בסקריפט פיתון שבודק משאו על המחשב שלכם. לדוגמה, אם תייקית קבצים כלשהן מכילה קבצי פיתון, בעלי הסיומת `py` או `sc`. כאשר אתם מרכיבים את הסקריפט, אתם מתבקשים להזין את שם התיקייה אותה אתם מעוניינים לבדוק. זה בסדר, אבל משאו פה מיותר: מילא כאמור你们 את מרכיבים את הסקריפט אתם יודיעים על איזו תיקייה תרצו לפעול. למה צריך שהסקריפט ידפיס הודעה ויבקש ממך להזין קלט? למה שלא

תעבירו את שם התיקיה לסקרייפט כבר ברגע ההרצה? אם לסקרייפט הדמיוני שלנו קוראים `py.findpy`, אז נרצה לכתוב ב-`cmd` פקודה כגון:

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

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

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

מבחן טכנית אנחנו רוצים שהקובץ `findpy` יהיה כתוב כך שהוא יוכל לקבל פרמטר את שם התיקיה שבה הוא צריך לבדוק האם קיימים קבצי פייתון. הנה נראה איך עושים זאת.

בתוור התחלת נכיר בקצתה את המודול `sys`. מודול (או ספריה) הוא קובץ שמכיל פונקציות, ובפרקם הבאים נכיר מודולים נוספים. המודול `sys` מכיל פונקציות שמאפשרות פעולות מערכת שונות, כגון הגדרות של הדפסה למסך וכדומה – – קבלת פרמטרים לסקרייפט. אפשר לקרוא על `sys` בlienק 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
```

```
def main():
    print sys.argv

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

בעקבות השימוש ב-`import`, כל מבנה (פונקציה, משתנה או קבוע) שנכתב במודול `sys` נגישים לנו, כלומר אנחנו יכולים להשתמש בו. כדי להשתמש במבנה שモגדר במודול, אנחנו צריכים קודם כל לכתוב את שם המודול, לאחר מכן נקודה ואז את שם המבנה. לדוגמה, כדי להשתמש ברשימה `arg` שמוגדרת ב-`sys`, צריך לרשום `print`

args, בדיקן כמו בדוגמה שבקוד. בהמשך הספר, בפרק על OOP, נלמד על שיטות נוספות להשתמש בפונקציות ובמבנה נתונים נוספים שנמצאים בתוך מודולים.

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

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

```
import sys
NAME = 1

def main():
    print 'Hello {}'.format(sys.argv[NAME])

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

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

Hello Shooki

תרגיל – Printer



כתבו סקריפט שמקבל כפרמטר שם של קובץ ומדפיס את תוכן שלו למסך.



רגע אחד – מה יקרה אם ננסה להעביר לסקריפט שם שלקובץ שאינו קיים? נסו זאת. סביר שהסקריפט שלכם יק:rightros עם הודעת שגיאה. זו בעיה, מכיוון שגם אם המשתמש שגאה והזין שם קובץ שאינו קיים, לא נרצה שהקובץ שלנו יקרוא. במקום זה, עדיף להחזיר למשתמש שגיאה שמסבירה לו שהקובץ אינו קיים. כדי לתקן את הבעיה, נזכיר את המודול `os`. מודול שימושי נוסף זה, קיצור של `Operating System`, מספק יכולות שונות של מערכת הפעלה. בתור דוגמה, נראה איך מציגים שמות של קבצים בתיκיה.

נתחיל כמובן מ-`os import`. ביצוע `ziof os` יראה לנו את כל הפונקציות שישיות ל-`os`, ביניהן נמצאת הפונקציה החביבה `listdir`. עצה נשנה את הסקריפט שלנו בהתאם:

```

import sys
import os
PATH = 1

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

if __name__ == '__main__':
    main()

```

אנחנו טוענים לערך `directory` את הפרמטר שקיבל הסקריפט שלנו, ואז אנחנו מעבירים את ערך `1` ל-`directory`. `os.listdir` על מנת לקבל את רשימת הקבצים.



תרגיל – os.path

הבעיה בקוד שלנו, כמו בקוד שכתבתם כפתרון לתרגיל `printer`, היא שעדין נהיה בעיה אם לסקורייפט יועבר שם של תיקיה שאינה קיימת. עליו לפטור את הבעיה באמצעות בדיקה האם התיקיה קיימת ואם היא אינה קיימת – להדפיס שגיאה כגון "Directory not found", לפני שאתם מבקשים את הקבצים שנמצאים בה. טיפ: `os.path` מיל מתודה שמקבלת `path` לティקיה ובודקת אם היא קיימת. תוכל לקבל את רשותת כל המתוודות באמצעות `os.path.isdir()` ותוכלו להשתמש ב-`-h` על מנת לקרוא מה עשו כל מתוודה, עד שתמצאו את המתוודה המתאימה.

**תרגיל מסכם – Lazy Student**

קיבלתם כשיעור 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 סקורייפט שמקבל קובץ תרגילים, כאשר כל תרגיל בשורה נפרדת, ושומר לקובץ נפרד את כל התרגילים כשם פרטיים. לדוגמה:

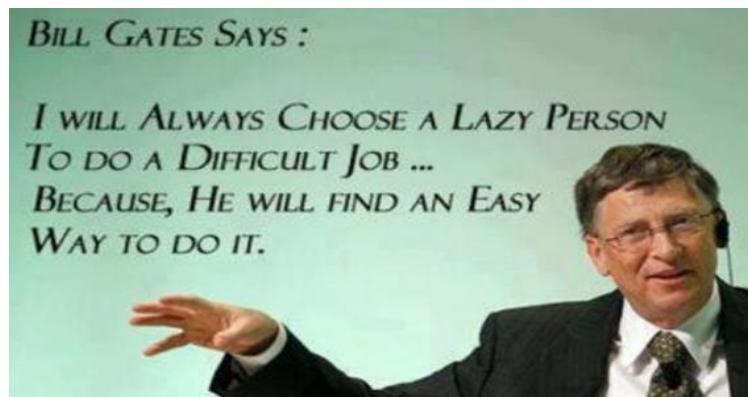
$46 + 19 = 65$

$15 * 3 = 45$

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

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

הסקורייפט יקרא את התרגילים מתוך `homework.txt` וישמר את הפתרונות אל `solutions.txt`. הניתן מבון שיש שגיאות בפורמט של חלק מהתרגילים, או בשמות הקבצים. אסור לסקורייפט 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, לקרוא וכתיבת. הכרנו את הפונקציות המובנות לעובדה עם קבצים: `read`, `open`, `write`. רأינו 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 להכין 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 שונים 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`".



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

```
]import sys
import os
PATH = 1

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

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

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

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

אם תהיה בעיה כלשהי, התוכנית תקפוץ 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

שנחסכת מאייתנו כתיבת קוד – אפשר לבדוק את כל המקרים שעלולים להוביל לкриיסת קוד באמצעות פקודה `יחידה`.

מה הבעיה במצבה כזו? 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`-`except`.

לפניכם פונקציה שעלולה להגיד להגעה ל-`except`. בידקו את עצמכם: באיזה מקרה הפונקציה תגיע לכך שנמצא בתוך `try`-`except`?

```
def do_something(thing):
    try:
        print 'Hello ' + thing
    except:
        print 'Error'
```

אמנם, אם אין מחרוזת, הפונקציה תדפיס 'Error'.

נסביר מעט את הפונקציה. מה לדעתם יהיה הערך של `x` אם הקוד ייכנס ל-`except`? במילים אחרות, אם לפונקציה יעבר 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
```

שגיאה נפוצה היא לומר שיודפס 0. הטענה שגורסת שיודפס 0 היא "בתחלת התוכנית `x` מקבל את הערך 0. הריצה של `try` לא מתבצעת עקב ה-`exception`, לכן ערכו של `x` נותר 0 כאשר הוא מגע ל-`except`". הטעות היא בכך שחלק מה-`try` דואקן מבוצע. כל מה שקדם ל-`except` בהחלט יבוצע. במקרה זה, מתבצעת

השורה בה מושאים ב-`x` את הערך 1, וכך זה 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 הטענו עדין תקפות. המעבד אינו "חזר אחורה" ...

סוגים של Exceptions

נחזיר אל הפונקציה שהציגנו לפני זמן קצר:

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

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

- המשמש לא חזן כלל פרמטר, ולכן הגישה ל-[`PATH`]`sys.argv` תהיה לאינדקס לא חוקי ברשימה.
- המשמש חזן תיוקה שגואה, אך `os.listdir` יחזיר שגואה.

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

```
def main():
    try:
        directory = sys.argv[PATH]
        print os.listdir(directory)
    except Exception, e:
        print e
```

צורת הכתיבה של `except` משמעotta "הכנס את הודעת השגיאה של ה-exception לתוך המשתנה e". כפי שראינו לכל `exception` יש הודעת שגיאה ייחודית, המשתנה `e` יכול אותה. בדוגמה הבאה אפשר לראות איך אנחנוTopics מטפלים שונים בכל אחד מהם בנפרד. אם הייתה שגיאה שלא מסוג exceptions היא תיתפס על ידי ה-exception `ZeroDivisionError` או `TypeError`:

```
def do_something_that_will_raise_exception(num):
    print num / 0

num = 12
try:
    do_something_that_will_raise_exception(num)
except ZeroDivisionError:
    print 'Zero division error'
except TypeError:
    print 'Type Error'
except Exception, e:
    print e
```

מהו `e`? הוא `class` של פיתון (מאחר יותר למד תכונות מונחה עצמים), אך `e` הוא אובייקט שמכיל את כל השדות של `Exception`, כולל הודעת השגיאה.

finally

הפקודה `finally` נמצאת לעיתים קרובות בשימוש יחד עם הצירוף `try-except`. פקודה זו שימושית כאשר נרצה שקוד כלשהו יירץ בכל מקרה, בין שארע `exception` ובין שלא. מן הסתם, לא הגיוני להעתיק את אותן שורות קוד הן ל-`try` והן ל-`except`. כאן מגיעה `finally` לסייעתנו. כל מה שנמצא בבלוק של `finally` יירץ בכל מקרה.

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

```
def main():
    do_something('Shooki')
    do_something(123)
```

נסקרו דוגמה. חישבו מה ידפיס היקוד 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 ולאחר מכן לביצוע finally. תוצאה הריצה תהיה ההדפסה הבאה:

```
Hello Shooki
Final
cannot concatenate 'str' and 'int' objects
Final
```

נסקור 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, e:
        print e
        return 'Error'
    finally:
        print 'Final'

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

תוצאה הריצה היא:

```
Hello Shooki
Final
"Shooki" returns: OK
cannot concatenate 'str' and 'int' objects
Final
123 returns: Error
```

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

בקראיה 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, e:
        return 'Error'
    finally:
        return 'Final'

def main():
    print '123 returns: {}'.format(do_something(123))
```

הפונקציה מגיע אל ה-*except*, שם כפי שראינו בדוגמה הקודמת ה-*interpreter* של פיתון יקفوץ אל *finally*. ההבדל מהדוגמה הקודמת, הוא שבמקרה זה מכילה הוראת חזרה ולכן בכך תסתיים ריצת הפונקציה. תוצאה ההדפסה:

123 returns: Final

with

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

למעשה, ניתן לתרגם את *with* לפקודות שכוללות *try*-*finally*. לדוגמה, הקוד הבא:

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

שקלול לќוד הבא:

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

כפי שאנו רואים, עם `with` בתוכו `finally`, אשר דואג לסגירת הקובץ בכל מקרה – גם אם ארצה שגיאה במקום כלשהו בבלוק ששייר לו.

תרגיל מסכם – lazy student 2



כתבו מחדש את הפתרון לתרגיל lazy student, אך תוך שימוש ב-`try-except` במקום הנדרשים. הקפידו על כך שבמידה וארעה שגיאה כלשהי (קובץ לא נפתח, חלוקה באפס) התוכנית תדפיס הודעה שגיאת מדוייקת ולא רק שארעה שגיאה כלשהי.

קיבתם 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 סקירהפט שמקבל קובץ תרגילים, כאשר כל תרגיל בשורה נפרדת, ושומר לקובץ נפרד את כל התרגילים כשם פטורים. לדוגמה:

46 + 19 = 65

15 * 3 = 45

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

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

הסקריפט יקרא את התרגלים מתיקhomework.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 הודעת שגיאה והמשיכו לתרגיל הבא.

תרגיל מסכם פיתון בסיסי – LogPuzzle



(เครดיט: [google class](#), התאמת לגבאים: דנהabenheim, אורית לוי)

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

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

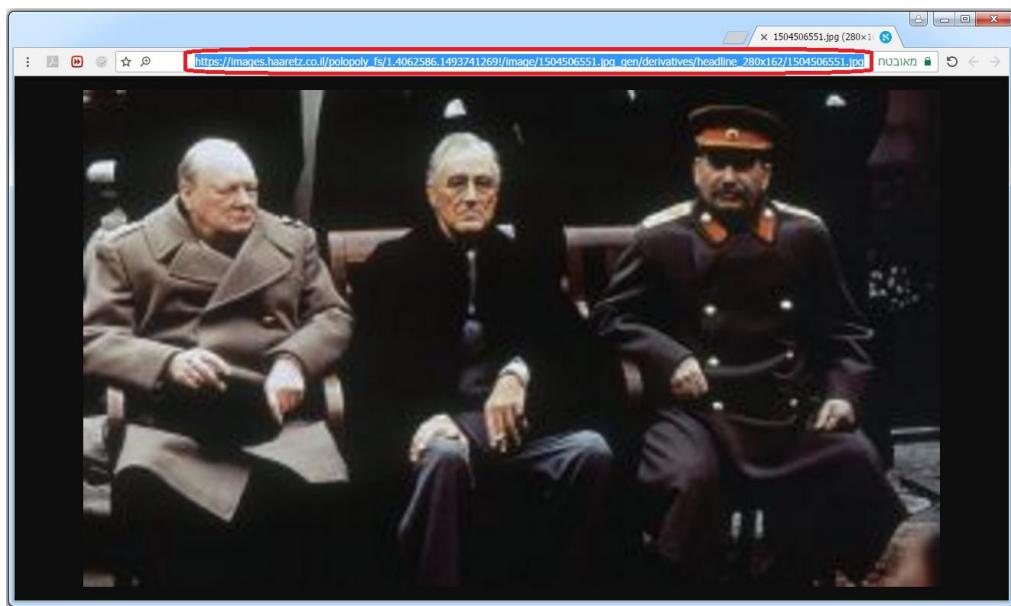
ראשית – מהו URL? אלו ראשי תיבות של Universal Resource Locator. לכל משאב באינטרנט יש URL משלו. לשם המחשב, יכול להיות שרת האינטרנט שכותבו www.cyber.com. בתוך השרת זהה שמור קובץ pdf בשם `file.pdf` ועמוד html בשם `page.html`. ניתן יהיה לכל משאב לפי ה-URLים הבאים:

<http://www.cyber.com/file.pdf>

<http://www.cyber.com/page.html>

כעת אנחנו בשלים להבין את השימוש ב-`urllib`. מודול זה כולל פונקציה שנазטרך – `urlretrieve`. הפונקציה מקבלת שני פרמטרים: הראשון – URL של קובץ שרוצים להוריד ולשמור במחשב שלנו. השני – שם הקובץ אליו ישמר הקובץ שיורד. נתרgal את השימוש בו:

גילשו לאתר האינטרנט 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 וכיתבו `import urllib`.

את ה-URL הדבוקן כך שיישמש בתור הפעמטר הראשון `urlretrieve` לפונקציה `open` והוא ישמור שם של קובץ אליו יש לשמר את הקובץ שאתם מורידים. כמובן שצריך לדאוג לכך שם הקובץ יהיה בעל סימנת זהה לקובץ שאתם מורידים. לדוגמה, אם אנחנו מורידים קובץ עם סימנת `.jpg` אז שם הקובץ `.jpg` איזשהו שומרם צריך גם הוא להסתיים ב-`.jpg`. סימנת הקובץ שאנו מורידים נמצאת תמיד בתווים האחרונים (הימניים) של ה-URL.

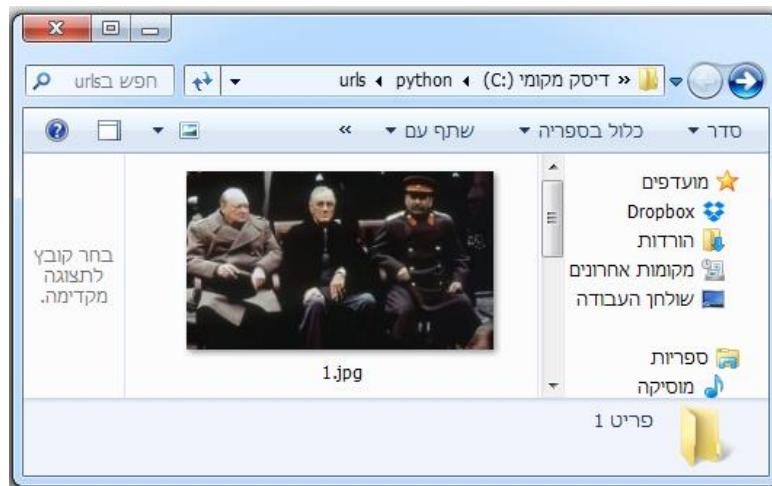
לדוגמא:

```
import urllib

def main():
    urllib.urlretrieve('https://images.haaretz.co.il/polopoly_fs/1.4062586.1493741269!/'
                       'image/1504506551.jpg_gen/derivatives/headline_280x162/1504506551.jpg',
                       r'c:\python\urls\1.jpg')

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

והתוצאה:

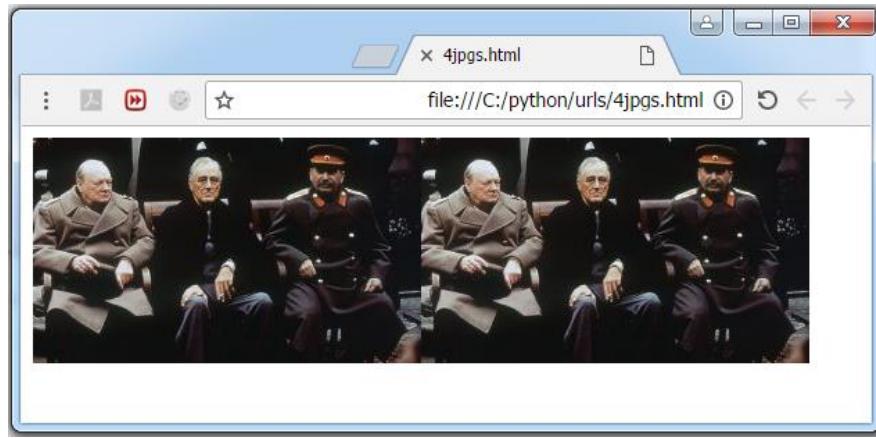


כעת משאנו יודעים איך להוריד תמונות מהאינטרנט, יש פרט נוסף שנוצר כדי לפתור את הפאל – חיבור תמונות לתמונה אחת. לשם כך נשתמש בקובץ `1.html`. כאשר רושמים בתוך הקובץ את שמות התמונות, הפעלת הקובץ מציגה אותן זו לצד זו.

כל מה שאנו צריכים לדעת כדי לכתוב את קובץ `1.html` שנדרש בשבייל לפתור את הפאל, הוא כיצד להשתמש בתבנית הבאה:

```
4jps.html
1 <verbatim>
2 <html>
3 <body>
4 
5 </body>
6 </html>
```

הקלקה על הקובץ שייצורו תפתח דף ובו תוצג התמונה שהורדנו – פעמיים.



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

לאחר הקדמה קצרה זו, ניגש לתרגיל עצמו. בLINIK הבא נמצאים שני קבצי לוג. בכל קובץ לוג ישנו טקסט שמתחלבים בו ארבעים של תמונות jpg שציריך להוריד. לאחר ההורדה של התמונות ושמירתן לדיסק במקום שתבחרו, עלייכם למיין אותן ואז להציג אותן זו לצד זו בהתאם לסדר המין שנקבע בתרגיל.

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

כיצד לזהות את התמונות שציריך להוריד מקובצי הלוג? ה-URL של כל תמונה נמצא בטקסט שמופיע החל מסימן פקודות ה-'GET' ועד סוף ה-'jpg'. כמו כן, ה-URLים כוללים את השם שרת גבהים.

מה ה-URL המלא של הקבצים שאתם מודדים?

עליכם לקחת את שם קובץ jpg ולהוסיף לו את שם קובץ הלוג ואת הכתובת של שרת גבהים. לדוגמה, הקובץ עליכם למכור את שם קובץ jpg וליחסיו לו את שם קובץ הלוג ואת הכתובת של שרת גבהים. ה-URL המלא הוא:

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

איך מתבצע המיון?

- עברו הקובץ logo, המיון הוא לפני סדר האלף-בית של הקבצים
- עבור הקובץ message, המיון הוא לפני סדר האלף-בית של המילה השנייה בשמות הקבצים. לדוגמה הקובץ jpg-a-aaaaa-zzzzz.jpg, יבוא אחרי הקובץ jpg-ccccccc-z, כיוון שבמילוןcccc קודם ל-zzzzz.

בלינק ישנו קובץ בשם `uy.logpuzzle`, שמהווה את קובץ השלד לפתרון התרגיל. הוא כולל את הפונקציות שנדרשות לביצוע המשימה, ועליכם להשלים את הקוד החסר בפונקציות.

כעת תורכם – פיתרו את הפАЗל. בהצלחה!

פרק 10 – תכונות מונחה עצמים – OOP

בפרק זה נלמד כיצד מבצעים תכונות מונחה עצמים בפייתון. יתכן ששמעתם על המושג "תכונות מונחה עצמים", או באנגלית 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 נרויח 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 זהה. נמחיש על ידי דוגמא.

אנחנו רוצים לכתב תוכנית שתשתמש בית ספר. יש צורך לשמור 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
```

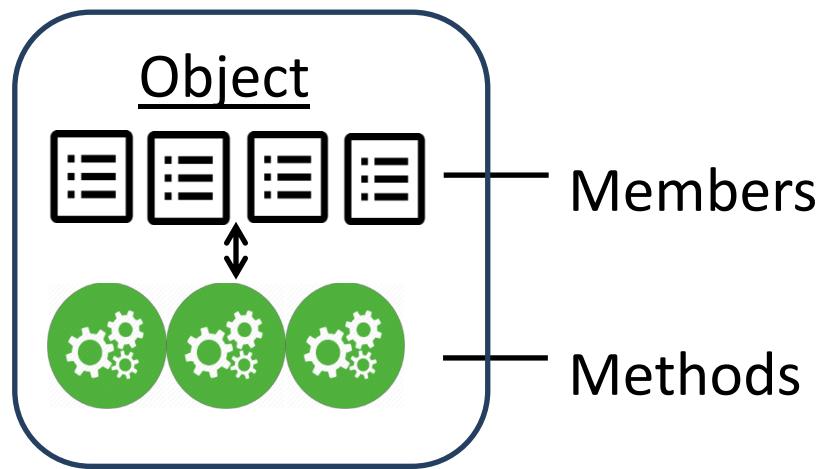
```
talmid1_geography = 85
```

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 בדרך 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 למצום את כמות המשתנים שלנו. נלמד לבנות `class` של תלמיד, ובאמצעות 200 אובייקטים מסווג תלמיד – מיד נלמד גם מהו אובייקט – נחזק את כל המידע על התלמידים. מצמנו את כמות המשתנים שלנו מאלפיים למאות. נחמד? ניתן לטעון בצדק שאנו המצומצם מאלפיים למאות הוא נפלא, אבל גם לבצע פעולות על מאות אובייקטים יגרום לנו לתכנת קוד די ארוך. ובכן, נראה שאפשר לבצע על כל האובייקטים עם לולאת `for` וכן לkürר עוד יותר את הקוד שלנו.

במהשך 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 ומethods שפועלות על המידע

נתקלנו כבר בmethods והסבירנו שהה סוג מיוחד של פונקציות. אם כך, עכשו אנחנו יכולים לחזור את ההסבר – methods היא פונקציה, אבל לא סתם פונקציה אלא שמודרנת כחלק מאובייקט ופעולה על האובייקט. כמובן, רק האובייקט מכיר אותה ורק הוא יודע איך לعباد איתה. לדוגמה, ראיינו שקובץ הוא אובייקט. לאובייקט מסוג file יש方法ה שקוראים לה `read()`. אנחנו לא קוראים לה בצורה זו:

`read(filename)`

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

`filename.read()`

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

לסיום, אובייקט מכיל גם מידע וגם methods. ראיינו איך ניתן לmethods של אובייקט, מיד נראה איך "מייצרים" אובייקט שככל גם methods וגם מידע.

מחלקה – class

מחלקה, או `class`, הוא קטע קוד שגדיר את כל members והmethods של אובייקט ואשר משתפים לו וליתר האובייקטים של אותה מחלקה. נניח שזאת. ניקח את כוכב הלכת שבתאי. זהו כוכב לכט אחד מתוך מספר כוכבי לכת במערכת השמש שלנו. למרות שכל אחד מהם הוא שונה, כוללם יש מהה, רדיוס, מרחק המשמש, זמן הקפה של השמש וכו'. אפשר להגיד מחלקה של כוכבי לכת, 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 השדות שצריך בשביל להגיד כוכב לכט.

Planet

שם הכוכב:

סזה:

מרחק מהמשמש:

זמן הקפה:

טמפרטורה:

از מה ההבדל בין אובייקט למחלקה? מחלקה מתארת את המאפיינים של כל האובייקטים שישיכים למחלקה. כשהתוכנית רצה, בכל פעם שנגידר אובייקט מסוים, יוקצה 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 ששנשתמש בתבנית, חתיכה של בץ תקבל את הצורה של התבנית זו. כמוות הבץ שנקצתה לעוגיה נקבעת על ידי התבנית, דומה לכך שכמות הזיכרון שמקצתה 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`.

בkrabob, נראה כיצד כתבים `class`, וכייז יוצרים אובייקט שלו – קלומר `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 CrazyPlane:

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

נפרק את מה שכתוב לחלקים.

שם ה-class הוא `CrazyPlane`. שמו לב לכך שהשם מתייחס לאות גדולה – זהה ה konkונבציית הקביעת שמות של מחלקות. מיד נסביר מהם ה-`__init__` וה-`self`. לאחר מכן מוגדרים שני members (זכירים שאמרנו שאובייקט מורכב ממethodות ו-members?). הראשון הוא `x` והשני הוא `y`. כל אחד מהם מקבל ערך התחלתי – 0.

`__init__`

זו היא המתודה הראשונה שנכיר. השם `__init__` נגזר מ-`initializer`, או אתחול. אפשר לקרוא לה גם constructor או בניאי. בתוך `__init__` נהוג לשים את האתחול של כל ה-members של המחלקה, כפי שראינו בדוגמה. זהה מетодה מיוחדת בכך שהיא רצה באופן אוטומטי בכל פעם שנוצר `instance` של `CrazyPlane` בזיכרון המחשב.

כפי שאנו רואים בדוגמה, המתודה מקבלת פרמטר שנקרא `self`. כדי להבין מהו, נשאל את עצמנו שאלה: איך פיתון יודע על איזה אובייקט להפעיל את המתודה? במקרים אחרים, אנחנו יכולים ליצור כמה אובייקטים ששיכים לאותו `class`, לדוגמה כמו אותה תבנית עוגיות שימושת לייצור עוגיות רבות. אם אנחנו רוצים להפעיל מетодה על עוגיה מסוימת, איך פיתון יודע על איזו עוגיה הוא צריך לפעול?

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



הוספה מתודות

לאחר שכתבנו את המתודה הראשונה, `__init__`, הגיע הזמן להוסיף מתודות שעשויות לפעולת שאין אתחול. כמו שמדובר במשתנים, נרצה לאפשר לעדכן מיקום ולקבל את המיקום:

```
import random

class CrazyPlane:

    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` המיצגים את מיקום המטוס ומעדכנת אותם רנדומלית (כמו שאמרנו, הטיסים השתגעו). המתודה `get_position` פשוטמחזירה את מיקום המטוס.שוב, נשים לב לכך שכל מתודה צריכה לקבל את `self` בתור פרמטר. כמו כן, נעשה שימוש במודול `random` ולכן נדרש לבצע `import random` בתחילת הקובץ.

Members

כפי שלמדנו זה עתה, ברגע שניצור אובייקט מסווג `CrazyPlane` 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 `y,x`:

```
class CrazyPlane:

    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 `i`. המשתנה `i` "חיה" רק בתוך לולאת ה-`for` של methodה. ברגע שהלולאה מסתיימת, `i` אינו קיים יותר ולא ניתן לגשת אליו. זאת לעומת `x`, `y` שקיימים כל עוד האובייקט קיים. לכן `x,y` הם `members`.

יצירת אובייקט

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

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

�ישבו – מה יהיו ערכיהם של `xpos, ypos` ?

...יצרנו אובייקט בשם `plane1`, שהוא `instance` של `CrazyPlane`. ברגע שייצרנו אותו, אוטומטית מורצת פונקציית `__init__`, אשר יוצרת את `x` `plane1.x` ואת `y` `plane1.y` ומתחילה את ערכיהם ל-0. methodה `get_position` פועלת על האובייקט `plane1` ומהזירה את ערכי `x` וה-`y` שלו, כלומר 0, 0. נסו זאת בעצמכם – צרו אובייקט ובידקו שקיבלו את ערכי ההתחלה שקבעתם.

נעשה בתוכנית שלנו שינוי קטן. במקומות `sosx` כתוב `x` ובמקומות `sosy` כתוב `y`. האם התוכנית תעבור כעת, או שתתקבל שגיאה?

```
def main():
    plane1 = CrazyPlane()
    x, y = plane1.get_position()
```

התשובה היא שהתוכנית תעבור ללא כל בעיה. חשוב להבין ש-`y`, `x` שהגדכנו בפונקציה `choose` אינם `y`, `x` ששווים ל-`plane1`. במילים אחרות, `x` אינו `x`. `plane1` ו-`y` אינו `y`. `plane1`. אין להם את אותו ()`p`. גם אם נשנה את `x`, ערכו של `x` לא ישתנה.

תרגיל



כתבו `class` של חיה האהובה עליכם (לדוגמה `Cat`, `Dog` וכו').

- הוסיפו מתודת `__init__` שתcalcul את שם החיה (`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__`. בקורס נלמד כיצד נתונים לכל חיה שם שונה בשלב האתחול.



כתיבה `class` משופר

בחלק זה נלמד לשפר את ה-`class` שכתבנו בחלק הקודם. לשם כך נלמד לבצע כמה דברים:

- לייצור `members` "מוסתרים"
- להפוך את ה-`class` לקובץ שנitin ליבא על ידי `import`

- לקבוע ערכים התחלתיים לאובייקט חדש

- ליצור פקודת הדפסה מיוחדת ל-class שלנו

- ליצור מethodות mutator-accessor

יצירת members "מוסתרים"

קטע הקוד הבא דורס את נטווי המיקום של המטוס:

```
plane1 = CrazyPlane()
plane1.x = 10
plane1.y = 10
x, y = plane1.get_position()
```

הmethodה `get_position` תחזיר את הערכים 10, 10. בעולם האמיתי, תוצאה של קוד זהה עלולה להיות הרגשות בין מטוסים. علينا למצוא דרך "להסתיר" את נתוני המיקום של המטוס, כך שהתוכנה שעושה בהם שימוש לא תשנה אותם בטעות.



בפייטון, הסתירה של `members` מתבצעת באמצעות תחילה `__` (קו תחתית כפול). נסתרר את המשתנים של `:CrazyPlane`

```

class CrazyPlane:

    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 התחתי הכהן גורם לכך שרק מתודות של `CrazyPlane` מכירות את ה-members הללו. מי שמשתמש ב-`class` שלנו כבר לא יכול לראות שהם קיימים. שימו לב להבדל: לפני ההסתירה, כתיבת של "plane1" העלה אפשרויות שונות, בין היתר `x`, `y`.

```

plane1 = CrazyPlane()
plane1.|
```

The screenshot shows an IDE interface with code completion for the variable 'plane1.'. A dropdown menu lists the following options:

- `x` (highlighted in orange)
- `get_position(self)`
- `update_position(self)`
- `y` (highlighted in orange)

The items are grouped by class: 'CrazyPlane' contains 'x', 'get_position(self)', 'update_position(self)', and 'y'. The background of the code editor is yellow.

לאחר ההסתירה, אפשר באמצעות המושלים האוטומטי לראות רק את המתודות הבאות:

```

plane1 = CrazyPlane()
plane1.|
```

The screenshot shows an IDE interface with code completion for the variable 'plane1.'. A dropdown menu lists the following options:

- `get_position(self)` (highlighted in orange)
- `update_position(self)`

The items are grouped by class: 'CrazyPlane' contains 'get_position(self)' and 'update_position(self)'. The background of the code editor is yellow.

האם זה אומר שכבר א' אפשר לגשת אל members מוסתרים? לא. פ'יטון מאפשרת לנו גם אפשרות זו. בשפות עליות י'נו מושג שנקרא "private", שימושו משתנים או מתודות שא' אפשר לגשת אליהן מחוץ ל-class. בפייטון אין private, יש רק הסתרה. אם מתעקשים, אפשר לגשת אל members אפילו אם הם מוסתרים.

זכור פונקציית dir ממחזירה את כל המתודות וה-members ששייכים לאובייקט מסוים. אם כך, נעשהdir(Plane1) ונקבל:

```
['_CrazyPlane__x', '_CrazyPlane__y', '__doc__', '__init__', '__module__', 'get_position', 'update_position']
```

שים לב לשני האיברים הראשונים ברשימה, אשר מסתיימים ב-"x" ו-"y". כן, נראה דומה למה שאנו מוחפשים. ננסה לפנות אליהם:

```
def main():
    plane1 = CrazyPlane()
    plane1._CrazyPlane__x = 5
    plane1._CrazyPlane__y = 5
    print(plane1.get_position())
```

הקוד רץ ללא שגיאות וכאש מדפיס את הערך שמחזירה get_position מתקבל (5,5), מה שמעיד על כך שakan שינו את הערכים.

האם מדובר בbag של פ'יטון? לא, כך פ'יטון תוכננה. הרעיון הוא לאפשר גמישות מקסימלית למוכנותים. פ'יטון אומרת "ראו, יש סיבה שהמשתנים הללו מוסתרים. מי שכתב את הקוד לא רצה שתוכנית חיונית תיגש ותשנה אותם. אבל אם אתם יודעים מה אתם עושים, והקצתם את הזמן להתגבר על ההסתירה, אז בבקשה – שנו כל מה שתרצו". יכולת זו שימושית רק במקרים נדירים, אבל במקרה הצורך היא אפשרית לנו, לדוגמה, לבצע שינויים בספריות פ'יטון שיש בהם בגים, בלי להזדקק לקוד המקורי.

אם כך, משתנים מוסתרים בפייטון הם למעשה דרך שלנו לסמן כי אין לשנות את תוכן המשתנה שלא על ידי קוד של המחלקה עצמה. עם זאת, זה רק סימן – ומשתמש חיוני יכול לשנות את ערכי המשתנה המוסתר, אם יבחר כך.

שימוש ב-accessor ו-**mutator**

סקרנו שתי שיטות גישה ל-members של מחלקה.

השיטה הראשונה, היא לגשת אל ה-members ישירות. לדוגמה `x1.get_plane` היא יכולה לגשת ל-member `plane` של `members`. אם מי שתכנת את המחלקה `CrazyPlane` דאג להסתייר את `x`, עדין יוכל לגשת אליו באמצעות `x1._CrazyPlane_x`.

השיטה השנייה, היא באמצעות שימוש בMETHOD שנותן גישה למחלקה ומשמשות במיוחד לטובת קריאה ו שינוי של members. לדוגמה, `get_position` היא דוגמה לMETHOD כזו. ראיינו שם נקרא `-(get_position)` קיבל `plane1` את ערכו המיקום, למרות שהם מוסתרים. לMETHOD שמאפשרת קריאה של members של מחלקה קוראים accessor ונהוג שהוא מתחילה ב-`get`.

METHOD שמאפשרת שינוי ערכים של members נקראת mutator ונהוג שהוא מתחילה ב-`set`. לדוגמה, אם נרצה לאפשר שינוי המיקום של המטוס, נגדיר פונקציה בשם `set_position`, שתקבל מיקום כפרמטר ותשנה בהתאם את מיקום המטוס.

יש דיון נרחב האם שימוש ב-methods accessors וב-mutators (אשר נקראים גם 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 בערכיהם המוכנסים ולבודא תקינות. לדוגמה, אם המתכונת יודע שיש מקום בעייתי שאסור שימושו יימצא בו הוא יכול למנוע זאת באמצעות קוד מתאים. הקוד הבא לא מאפשר להזין למטרס מקום שלילי או את מקום 4,4 כיוון שנמצא שם מגדל שאינו מאפשר מעבר מטוטלים:

```
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, טוענים נגד accessors, mutators טענות רבות, בין היתר שהם מסבכים את כתיבת הקוד. לדוגמה, אם יש אובייקט בשם a ו-member בשם b, אז במקום לכתוב כך:

a.b += 1

אנחנו נאלצים לכתוב, פחות או יותר, כך:

a.set_b(a.get(b) + 1)

ספר זה אינו שואף לפ██וק בשאלת מהי השיטה העדיפה. נציין שחשוב להכיר את שתי השיטות מכיוון שתיהן נפוצות בקוד בו אתם עשויים ליתקל בעתיד, וחשוב לדעת להשתמש ב-accessors-mutators ו-**get**-**set**, ولو 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 שהגדכנו. מדוע כדאי לשנות את זה? משום 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 שהגדכנו, ואולי כמה קבועים קשורים אליה – וזהו.

לשם כך, נבצע שתי פעולה: ראשית נפריד את המחלקה לקובץ עצמאי. שנייה, נבצע בתוכנית הראשית `import` למחלקה. נראה איך זה מתבצע.

נשמר קובץ פיתון בשם `planes.py`. לטובת המשך הגדרנו מחלקה נוספת, `NormalPlane`. להלן הקוד

שבקובץ:
`import random`
`CONTROL_TOWER_LOCATION = (4, 4)`

```
class CrazyPlane:

    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 NormalPlane:

    def __init__(self):
        self.__x = 0
        self.__y = 0

    def update_position(self):
        self.__x += 1
        self.__y += 1

    def get_position(self):
        return self.__x, self.__y

def main():
    print 'This main is not reached if the file is imported'

if __name__ == '__main__':
    main()

```

cut גדרת תוכנית משתמשת במחלקה שנמצאת בקובץ `plane.py`:

```
import planes
```

```

def main():
    plane1 = planes.CrazyPlane()
    plane1.set_position(3, 4)
    print plane1.get_position()

```

```

if __name__ == '__main__':
    main()

```

בשורה הראשונה אנחנו מבצעים `import planes`. כעת זו הגדמנות נהדרת להסביר את התפקיד של `if __name__ == '__main__'`. בקובץ `planes.py` קיימת פונקציה בשם `main`. כאשר אנחנו מיבאים את הקובץ, אנחנו לא מעוניינים להפעיל את `main` של הקובץ המייבא – יש לנו `main` משלהנו שאנו שואבים עובדים אליו. אנחנו רוצים רק את ההגדרות של המחלקות, וקבועים כלשהם אם הם קיימים, לדוגמה מיקום המגדל. תנאי ה-`if` הנ"ל מודיע לנו "אם הגעת לקובץ זהה תוך כדי ריצת התוכנית ולא בעקבות `import`, תריץ את `main`" – בבדיקה מה שרצינו שיקרה.

לאחר שביצענו `CrazyPlane` , אנחנו יכולים להשתמש במחלקות שיבאנו אל התוכנית שלנו. השימוש במחלקות מציר `planes` שינו קטן בקוד – לפנינו שאנו מגדירים אובייקט מסווג `CrazyPlane` ציר לסמן שהאובייקט שיר לקובץ `CrazyPlane` שהוא מחלקת שיבאנו, כפי שניתן לראות בשורה השלישי. רגע – איך פיתון לא יודע בעצמו ש- `CrazyPlane` הוא מחלקת שוגדרת ב- `planes` ? ובכן, פיתון יודע, אבל פיתון מניח שיכול להיות שישנה עוד מחלוקת בשם זה בקובץ אחר. מקרה כזה עלול בהחולט להתרחש, במקרה שיש מתכוונים רבים שכותבים מודולים בנפרד. 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. שימוש לב לצורה הבאה של `import` :

```
from planes import CrazyPlane
```

```
def main():
    planel = CrazyPlane()
```

cut פיתון יודע שאם אנחנו כתבים `CrazyPlane` אנחנו מתכוונים רק למחלוקת הנ"ל מתוך `planes` , אך אפשר לוותר על ציון שם הקובץ. שימוש לב לכך שאם אין היה קיימם קובץ אחר שיש בו מחלוקת בעלת אותו שם, כרגע "דרסנו" את שם המחלוקת בקובץ الآخر ולא נוכל להשתמש בו עוד.

ביצוע `import` רק למחלוקת אחת מתוך הקובץ – ולשם כך הוספנו לקובץ עוד מחלוקת – גורם לכך שכעת רק המחלוקת `CrazyPlanes` יובאה. המחלוקת השנייה, `NormalPlane` , נותרה לא מוכרת לתוכנית הראשית.

לעתים תיתקלו בייבוא של כל המחלוקות והקביעים של הקובץ באופן הבא:

```
from planes import *
```

```
def main():
    planel = CrazyPlane()
    plane2 = NormalPlane()
```

צורה זאת של יבוא עלולה לגרום לבאגים. מדוע? מכיוון שניתן להיות שבשני קבצים שונים ישנן שתי מחלוקות בעלות שם זהה. לדוגמה, המחלוקת `CrazyPlane` מוגדרת גם בקובץ `planes.py` וגם בקובץ `airports.py` . אם ניבא את שני הקבצים באמצעות `*` , מה יקרה כאשר נרצה ליצור אובייקט מסווג `CrazyPlane` ? האם מה שוגדר ב- `planes` או ב- `airports` ? התשובה היא שהקובץ שיבאנו אחרון "דורס" את ההגדרות הקודמות. לכן, שיטה זו אינה מומלצת.

אתחול של פרמטרים

אפשרויות הטישה של המטוסים השתנו, כאשר נפתח לרווחת הציבור שדה תעופה נוספת.Cut אם אנחנו רוצים לאפשר לחלק מהמטוסים להMRIIA משדה התעופה החדש. נניח שהמקום של שדה התעופה הוא (5, 5). כלומר, אנחנו צריכים להעביר את קואורדינטות המראה של המטוס כאשר אנחנו יוצרים מטוס חדש, מה שכמובן נכון לבצע ב- `__init__`:

```
def __init__(self, x, y):
    self.__x = x
    self.__y = y
```

אם אנחנו רוצים להגדיר מטוס חדש, אנחנו צריכים להגדיר אותו מראש עם הפרמטרים המבוקשים. לדוגמה:

```
from planes import CrazyPlane
NEW_YORK_X = 5
NEW_YORK_Y = 5
```

```
]def main():
    american = CrazyPlane(NEW_YORK_X, NEW_YORK_Y)
```

קביעת ערך ברירת מחדל

כיוון שרוב המטוסים ממראים משדה התעופה הישן, היינו רוצים שברירת המחדל למטוס חדש שמראה תהיה שדה התעופה הישן, שמיומו (0,0). נעשה שינוי קל במתודת האתחול:

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

Cut, פיתון בודק אם כאשר אנחנו יוצרים מטוס חדש אנחנו מעבירים את המקום כפרמטר. אם כן – המקום שהערכנו קבוע, אחרת – מקום ברירת המחדל קבוע. נוכל להגיד מטוסים חדשים באופן הבא:

```
elal = CrazyPlane()
american = CrazyPlane(NEW_YORK_X, NEW_YORK_Y)
```

האובייקט `elal` קיבל את ערכיו המקוריים של ברירת המחדל – כלומר 0, 0, ואילו האובייקט `american` קיבל את ערכיו ההתחלתיים – מיקום שדה התעופה של ניו יורק.

הmethod `__str__`

נסה לעשות `print` למתוס `shagdrano`, באמצעות הפקודה `:print american`

```
<planes.CrazyPlane instance at 0x0000000002330148>
```

air פיתון יודע מה להדפיס כאשר אנחנו עושים `print` לאובייקט מסוים? לכל אובייקט ישנה מתודת `__str__`. אנחנו יכולים לדرس את `__str__` ולהחליף 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 שידפס. לכן ישנו במתודה `return`. הערך שמוחזר הוא מחרוזת, שחלקיה קבוע וחלקיה מקבל את הערך שמחזירה המתודה `get_position` ומעביר אותו לפורמט המחרוזת.

אם עכšíי נבצע `print`, נקבל:

```
Plane position: (5, 5)
```

זהו. יותר יפה مما שהציגה פקודת ההדפסה לפני השינוי? ☺

__repr__

הmethod `repr`, קיצור של `represent`, היא דרך נוספת שימושית להדפסה של אובייקטים. אך בעוד `__str__` משמשת לצורך הצגה יפה של נתונים למשתמש, המתודה `__repr__` משמשת להצגת נתונים לתוכנת, על מנת לשיע בדיבוג.

נמchioש זאת באמצעות הדוגמה הבאה (เครดיט לרעיון – <http://www.geeksforgeeks.org/str-vs-repr-in-python/>):

```
In[5]: import datetime
In[6]: t = datetime.datetime.now()
In[7]: print t
2017-07-08 15:32:05.471000
In[8]: t
Out[8]: datetime.datetime(2017, 7, 8, 15, 32, 5, 471000)
```

הסביר: כאשר ביצענו `print`, באופן אוטומטי נקראת המетодה `__str__` של `datetime`, אשר בה מוגדר הזמן בפורמט נוח לקריאה. לעומת זאת, כאשר בדקנו מה ערכו של `t` קיבלנו את התוצאה של `__repr__` של `datetime`, אשר נתנה לנו יותר מידע על מה שתרחש "מאחוריו הקלעים". ניתן לכתוב `repr` עבור כל מחלקה שאנו חנכו כתובים, על ידי הגדרה של `__repr__`, בדומה לדרך שבה הגדרנו את `__str__`.

יצירת אובייקטים מרובים

עד כה יצרנו אובייקט אחד מהמחלקה שלנו. אפשר ליצור מה"tabnitt" כמה אובייקטים שאנו רוצים, רק צריך לדאוג שלכל אובייקט יהיה שם ייחודי, אחרת נדרס אובייקטים שכבר הגדרנו. נגדיר ארבעה מטוסים:

```
elal = CrazyPlane()
american = CrazyPlane(NEW_YORK_X, NEW_YORK_Y)
british = CrazyPlane(LONDON_X, LONDON_Y)
lufthansa = CrazyPlane(BERLIN_X, BERLIN_Y)
```

מテו אילע מרים מנוקדת בירית המבדל שלנו ואילו היתר מרים 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 colum בלאלה:

```
fleet = [elal, american, british, lufthansa]
for plane in fleet:
    print plane
```



תרגיל סיכום בניינים – כתיבת class משופר



שפרו את ה-class של החיה שיצרתם:

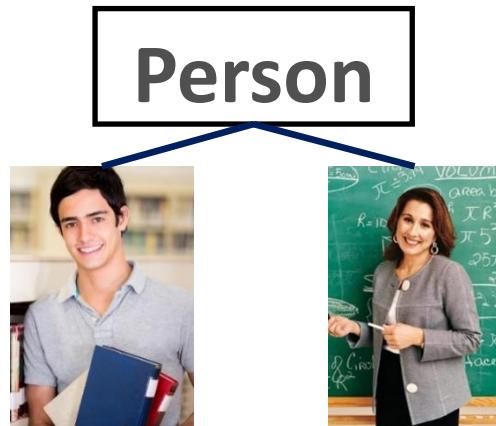
- העבירו את ה-class לקובץ חיצוני ועשו לו `import`
- הסתרו את שם החיה ואת הגיל שלה (תזכורת: _)
- אפשרו לקבוע את שם החיה בזמן יצירת האובייקט
- אפשרו לשנות את שם החיה (METHOD `set`)
- אפשרו לקרוא את שם החיה ואת גיל החיה (METHOD `get`)
- אפשרו הדפסת פרטי החיה על ידי קראיה ל-`print` (METHOD `__str__`)



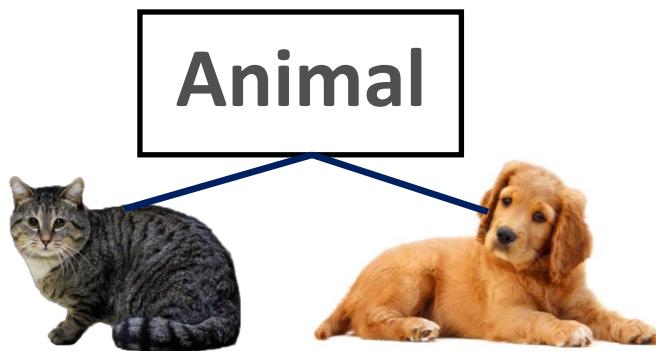
ירושא – inheritance

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 ללקח 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 נקראת **המחלקה הירשתית**, או **sub class**. המחלוקת שירשנו נקראת **super class**. לדוגמה:

"Person" superclass שלם subclasses הם Student ו Teacher -



"Animal" superclass שלם subclasses הם Dog ו Cat -



כדי להמחייב איר מייצרים subclass, בטור התחלה ניצור מחלקה בשם Person, עם כמה מתודות בסיסיות:

```
class Person(object):

    def __init__(self, name='Tal', age=20):
        self.__name = name
        self.__age = age

    def say(self):
        print '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
```

בשורה הראשונה מוגדר ש-Person יירש מ-object. זהו הנוהג בגרסת פיתון נמכה מ-3 (גרסת הפיתון שלנו היא 2.7). בגרסה 3 פיתון כבר מניח מראש שהירושות הן מ-object ולכן אין צורך לכתוב זאת במפורש.

cutut ניצור בית ספר. בבית ספר יש מורים ותלמידים.

למורים יש:

שם -

גיל -

- שכר

لتלמידים יש:

- שם

- גיל

- ממוצע ציונים

יש לנו כבר את המחלקה Person. בואו נשתמש בה. נגדיר שהמחלקות החדשות יורשות מ-Person – שימוש לבשנים 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 מסווג Teacher, פיתון מחפש את ה-`__init__` שלTeacher. כיוון שלא הגדרנו לו `__init__`, פיתון הולך אל המחלקה ממנה Teacher יורש, כלומר Person, ומפעיל את ה-`__init__` של Person. בהתאם של Person נקבעים משתני ברירת מחדל ו-`teacher1` יורש אותם. נציין שאליה ל-`Teacher` כן היה `__init__`, לא היה מופעל אוטומטית ה-`__init__` של המחלקה ממנה הוא ירש. לגבי פקודת ההדפסה, למרות שלא הגדרנו שום מתודה ל-`Teacher`, הוא יורש את `__str__` מ-`Person` ולכן ידפס:

Person Tal is 20 years old

כפי שהחלטנו, צריך שלכל מורה יהיה שכר. לכן 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` שיורשת מ-`Person`. במתודת ה-`__init__` הפעלנו את מתודת האתחול של `Person` עם פרמטרים שלחננו לה (`name, age`). כיוון ש-`Person` לא יודעת מה לעשות עם פרמטר ה-`salary`, המשנה זה הוא כל מה שմבדיל ברגע בין `Teacher` ל-`Person`. הגדרנו פשוט ש-`Teacher` הוא `Person` עם שכר.

אם אנחנו רוצים לקרוא למетодות של המחלקה ממנה ירשו, ניתן להשתמש ב-`super`. פונקציה זו מוצאת את המחלקה ממנה ירשו. הקוד הבא קורא לפונקציית ה-`__init__` של המחלקה ש-`Teacher` יורשת 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 למетодה של מחלקה, אשר המחלקה שלנו דרשה עם מתודה בעלת שם זהה.
- ב. במקרים של מחלקות שיורשות מספר מחלקות, שימוש ב-`super` הוא הכרחי כדי לוודא שמבצעים `__init__` של כל המחלקות מהן ירשו.

מה יודפס אם נrint כעת את הקוד הבא?

```
barak = Teacher('Barak', 40, 25)
print barak
```

תשובה:

Person Barak is 40 years old

וזאת מכיוון ש-Teacher יירש את ה-`__str__` של Person

תרגיל מסכם – ירשה



צרו את המחלקה Student שירושת מ-Person. עלייכם:

- לקבוע ערכי בריית מחדל כרצונכם

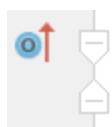
- להוסיף לאתחול משתנה נסתר שישמר את ממוצע הציונים

- הוסיפו לממוצע הציונים מתודות mutator-accessor

פולימורפיזם

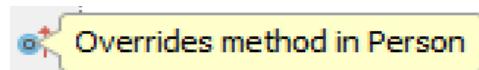
הסיבה שהגדכנו ל-Person מתודה בשם say היא כדי לבדוק האם חלק זה נפעיל את say על אובייקט המורה שלנו, ונקבל את המחרוזת "(:) Hi". אבל נרצה שהמורה יאמר משהו "חוד'", לדוגמה `Good morning cyber students!`, וכמו כן שיירש את כל התכונות של Person.

כדי לגרום למétoda say לעבוד כמו שאנו רוצים, נגדיר אותה מחדש בתוך :Teacher



```
def say(self):
    print 'Good morning cyber students! '
```

שימוש לב לסייע היגול הכהן לצד המתודה. אם נעמוד עליו עם העכבר, נקבל הסבר:



במילים אחרות – דרשו את המתודה say של Person.



מה יקרה אם נפעיל את say על אובייקט מסווג Teacher? נקבל את המשפט שרצינו.

אנחנו יכולים להגיד מחלוקת נוספת שתרиш מ-Person, לדוגמה teacher. כמו של-teacher יש מתודת say יחודית, גם ל-Student אפשר להגיד say. כתה אפשר להבין מדוע קוראים לפעולה זו "פולימורפיזם". משמעו "הרבה" ואילו "morph" ממשמעותו "צורה". הרבה צורות. ואכן, לעיתים 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 יפנה למשתנים מוסתרים של ה-superclass שלו. חשוב לדעת שימושים מוסתרים הינם מוסתרים גם מהיורשים. לכן, אם אנחנו רוצים להשתמש ב-members של superclass, ניתן לבחור לעשות זאת באמצעות מתודות accessor (כזכור יש דיון נרחב בשאלת האם אכן רצוי לעשות זאת).
לדוגמא, נעדכן את מתודת str של Teacher:

```
def __str__(self):
    return 'Teacher {} is {} years old, salary {} per hour'.\
        format(self.get_name(), self.get_age(), self.__salary)
```

הfonקציה `isinstance`

בבית הספר המעליה של נווה חמציצים נפתחה מגמת סייבר. נגיד תלמיד שלומד במגמה בתור CyberStudent, קלומר אובייקט שיירש מ-Student אך זהה שכולל גם ציון סייבר:

```
| class CyberStudent (Student):
| |
| |     def __init__ (self, name, age, grade, cyber_grade):
| |         Student.__init__(self, name, age, grade)
| |         self.__cyber_grade = cyber_grade
```

עד כה אין שום דבר חדש במה שעשינו. האדרנו subclass של Student. אך כעת, מנהל בית הספר מבקש שכיל תלמיד שהצין שלו בס"בר טוב קיבל הודעה "Wow!".

מתכונת הכניס לרשימה בשם students את האובייקטים של כל התלמידים, גם אלו שנמצאים במגמת סייבר וגם אלו שאינם. לאחר מכן המתכונת כתוב את הקוד הבא. האם הקוד עובד באופן תקין?

```
for student in students:
    if student.get_cyber_grade () >= GOOD_GRADE:
        print 'Wow!'
```

תשובה:

כאשר נורץ את הקוד, נקבל שגיאת הרצה -

`AttributeError: Student instance has no attribute 'get_cyber_grade'`

הסיבה היא שאנו לא-`CyberStudent` מודעה בשם `get_cyber_grade`, אבל ל-`Student` אין מודעה כזו. לכן, האובייקט הראשון מסוג `Student` שנמצא ברשימה יגרום לשגיאת ההרצה הנ"ל. על מנת לפתור את הבעיה, הפונקציה `isinstance` מקבלת שם של אובייקט ושם של מחלקה, ובודקת אם האובייקט שייר למחלקה או יורש מהמחלקה זו. אם כן – מוחזר `True`, אחרת – `False`. חשוב לציין, שאובייקט תלמיד שייר ל-`superclass` של המחלקה שלו.

כעת נוכל לכתוב מחדש את הלולאה שלנו, כך שרק אם student הוא מסוג `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 למחשבה. מוגדרים התלמידים הבאים:

```
noam = Student('Noam', 16, 93)
daniel = CyberStudent('Daniel', 17, 95, 90)
```

מה תהיה התוצאה של כל אחת מהבדיקות הבאות?

```
print isinstance(noam, Student)
print isinstance(daniel, CyberStudent)
print isinstance(noam, Person)
print isinstance(noam, CyberStudent)
print isinstance(daniel, Student)
```

תשובה:

שתי הבדיקות הראשונות יחזירו כਮובן True.

הבדיקה השלישייה תחזיר True מכיוון ש-noam הוא Student, כלומר subclass 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 פולימורפיזם – BigCat (קדagit: שי סדובסקי)



הגדירו מחלקה בשם BigThing, אשר מקבלת כפרמטר בזמן היצירה משתנה קלשו (המשתנה יכול להיות כל דבר – מחרוזת, רשימה, מספר וכו'). למחלקה יש מתודה בשם size, אשר עבדת כך:

- אם המשתנה הוא מספר – המתודה מחזירה את המספר

- אם המשתנה הוא רשימה / מילון / מחזוזת – המתוודה מחזירה את `len` של המשתנה

לדוגמה, עבור ההגדה:

```
my_thing = BigThing('table')
```

התוצאה של `size(my_thing)` יהיה 5, מאורע המחרוזת 'table'.

cutet הגדרו מחלוקת בשם BigCat, אשר יורשת מ-`BigThing` ומתקבלת כפרמטר בזמן היצירה גם משקל.

- אם המשקל גדול מ-15, המתוודה `size` תחזיר "Fat"

- אם המשקל גדול מ-20, המתוודה `size` תחזיר "Very fat"

- אחרת יחזיר "OK"

לדוגמה, עבור ההגדה:

```
latif = BigCat('latif', 22)
```

התוצאה של `size(latif)` תהיה "Very fat"