

תכנות בשפת פיתון

ברק גובן

תכנות בשפת פיתון

Python Programming / Barak Gonen

גרסה 1.01

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.

© כל הזכויות שמורות למרכז לחינוך סייבר, קרן רש"י. מהדורה ראשונה תשע"ז 2017

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

תוכן עניינים

8.....	הקדמה מעשית.....
8	התכונות נדרשות.....
10	אייקונים.....
11	תודות
12.....	תוכן עניינים מצגות פיתון.....
13.....	פרק 1 – מבוא ללימוד פיתון
13	מהו שפת סקריפטים?.....
15	עבודה דרך command line
17	מספרים בבסיסים שונים
18	Help
20	הסימן _ (קו תחתון).....
20	הרצה תוכניות פיתון דרך ה-command line
21	סיכום
22.....	פרק 2 – סביבת עבודה PyCharm
22	פתיחת קובץ פיתון.....
24	קובץ הפיתון הראשון שלנו.....
24	סדר ההרצה של פקודות בסקריפט פיתון
26	התרעה על שגיאות
28	הרצה הסקריפט ומסך המשתמש
30	דיבוג עם PyCharm
32	העברה פרמטרים לסקריפט
33	סיכום
34.....	פרק 3 – משתנים, תנאים ולולאות
34	סוגי משתנים בפייתון
36	תנאים
37	תנאים מורכבים

39	שימוש ב-is
39	בלוק
40	תנאי else, elif
41	לולאת while
43	לולאות for
45	pass
47	פרק 4 – מחרוזות
47	הגדרת מחרוזת
48	ביצוע print-format
49	היתוך מחרוזות – string slicing
51	פקודות על מחרוזות
52	dir, help
53	צירופי תווים מיוחדים ו-string-raw
54	קבלת קלט מהמשתמש – raw_input
58	פרק 5 – פונקציות
58	כתיבת פונקציה בפייטון
61	return
62	None
62	scope של משתנים
68	פייטון מתחת למכסה המנווע (הרחבה)
69	id, is
71	העברת פרמטרים לפונקציה
73	סיכום
74	פרק 6 – List, Tuple
74	הגדרת List
76	Mutable, immutable
78	פעולות על רשימות

78	in
78	append
78	pop
79	sort
81	split
82	join
82	Tuple
84	סיכום
85	פרק 7 – כתיבת קוד נכונה
86	PEP8
90	חלוקת קוד לפונקציות
91	פתרון מודרך
96	assert
99	סיכום
101	פרק 8 – קבצים ופרמטרים לסקריפטים
101	פתיחה קובץ
102	קריאה מקובץ
103	כתיבה לקובץ
103	סגירת קובץ
105	קבלת פרמטרים לתוכנית
109	סיכום
110	פרק 9 – Exceptions
111	try, except
114	סוגים של Exceptions
115	finally
118	with
124	פרק 10 – תכנות מונחה עצמים – OOP

124	מבוא – למה OOP ?
125	אובייקט – object
126	מחלקה – class
128	כתיבת class בסיסי
128	<code>__init__</code>
129	הוספה מתודות
130	Members
130	יצירת אובייקט
132	כתיבת class משופר
132	יצירת "מוסתרים" members
135	שימוש ב-accessor ו- <code>mutator</code>
137	יצירת מודולים ושימוש ב- <code>import</code>
140	אתחול של פרמטרים
140	קביעת ערך ברירת מהדול
141	<code>__str__</code>
141	<code>__repr__</code>
142	יצירת אובייקטים רבים
144	ירושה – inheritance
148	פולימורפיזם
149	<code>isinstance</code> הפונקציה
153	פרק 11 – OOP מתקדם (תכנות משחקים באמצעות PyGame)
154	כתיבת שלד של PyGame
155	שינויי רקע
159	הוספה צורות
161	תזוזה של ארכיפקה
163	ציור Sprite
165	קבלת קלט מהעכבר

168	קבלת קלט מהמקלדת
168	השימוש צלילים.....
170	מתוך PyGame – שילוב OOP
170	מבוא.....
171	הגדרת class.....
172	הוספה מתודות mutators-ו-accessors שימושיות.....
172	הגדרת אובייקטים בחוכנות הראשית
174	sprite.Group()
175	יצירת אובייקטים חדשים.....
176	הזאת האובייקטים
177	בדיקות התנגשויות.....
181	סיכום.....
182	פרק 12 – מיליוןים.....
185	מיליונים, מתחת למכסה המנווע (הרחבה)
187	סוגי מפתחות.....
188	סיכום.....
189	Magic Functions, List Comprehensions – 13
189	List Comprehensions.....
192	Lambda.....
193	Map.....
194	Filter.....
194	Reduce.....
195	סיכום.....

הקדמה מעשית

ברוכים הבאים לשפת פיתון! אם אתם קוראים ספר זה כנראה שאתם עושים את צעדיכם הראשונים בмагמת הגנת סייבר. מדוע לומדים דוקא פיתון? שפת פיתון נבחרה ללימודיו כיון שהיא נמצאת בשימוש רחב בתעשייה, באקדמיה וגם בקרב הייחוד הטכנולוגיות בזאה". חומר הלימוד הבאים של מגמת הסייבר מבוססים על שפת פיתון וכך נדרש שתילה בשפה כבסיס ללמידה יתר התכנים.

הספר כולל את החומר הנדרש כבסיס להשתות מחשבים, חומר הנלמד בכיתה י"א, והן למחקר תוכנה ופרוייקטי סיום, חומר הנלמד בכיתה י"ב. מומלץ ללמידה מן הספר נושאים לפי הצורך – לדוגמה, סביבת העבודה, שימוש במחרוזות ורשימות נדרשים ללמידה רשותות וכן יש ללמידה בתחלת י"א. לעומת זאת, תכונות מונחה עצמים נדרש בכיתה י"ב וכן ניתן לדוחות את הלימוד של נושא זה להמשך. בנוסף כולל הספר נושאי הרחבה לתלמידים המעניינים לשלוט בשפה יותר לעומק.

הספר לא מניח כמעט ידע מוקדם, אך כיון שהוא מיועד לכיתות י"א הוא מניח הিירות עם רעיונות בסיסיים בתכנות כגון משתנים, תנאים ולולאות. הוא נועד לאפשר למידה עצמאו, והוא פרקטטי וכובל מרגלים רבים. כדי לשנות בשפת פיתון, כמו בכל שפת תכנות, אי אפשר להסתפק במידע תיאורטי. הספר אינו מתיימר לכוסות את כל הנושאים בשפת פיתון, אלא להתמקד בנושאים שסביר שתזדקקו להם במהלך לימודכם. אחד הנושאים עליהם הספר אינו מרחב רבות הוא אודות השימוש במודלים, ספריות שפותחו בפייתון ומאפשרות לבצע פעולות מורכבות בקלות יחסית. הוויתור על ההסברים המפורטים אודות מודולים הוא ראשית מכיוון שקשה מאוד להקיף את כל המודלים החשובים בספר ללמידה, ושנית מכיוון שמטרתנו היא להעניק לכם את הבסיס ללמידה עצמי של חומרם מתקדמים. כך, כאשר תסימנו את הלימוד מהספר, תוכלו بكلות למצוא מידע באינטרנט או דוחות כל נושא שתרצו ולשלב אותו בהצלחה בתוכנה שאתם כתובים.

התכוות נדרשות

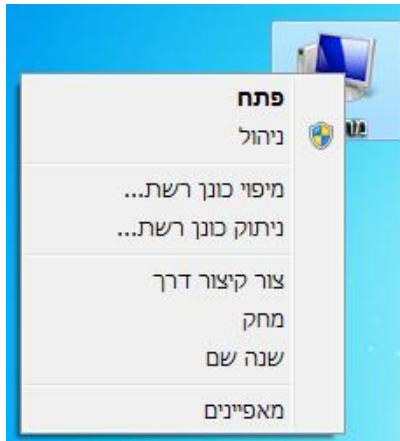
כאמור, ספר הלימוד מבוסס על שפת פיתון. ישנן התקנות רבות של פיתון, لكن נרצה להמליץ על סביבת עבודה ושימוש נכון בסביבת העבודה.

מומלץ להתקין פיתון כחלק מסביבת ההתקנות של גביהם, שפותחה על ידי אורי לוי ושמצאת בקישור:

www.cyber.org.il/workspace/gvahim_64bit.exe

www.cyber.org.il/workspace/gvahim_32bit.exe

איך אפשר לדעת אם מערכת הפעלה שלכם היא 32 או 64 ביט? כדי לוודא זאת, הקליקו קליק ימני על איקון "המחשב שלי" שב-desktop ובחרו "מאפיינים" (או properties).



במסך שיפתח תוכלו לראות תחת "סוג מערכת" האם המערכת שלכם היא 32 או 64 ביט:

מערכת	
דרוג: 4.6	מדד חוויה Windows
מעבד: Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz	3.10 GHz
זיכרון מותקן (RAM):	4.00 GB
סוג מערכת:	מערכת הפעלה של 64 סיביות
עת ומגע:	אין קלט עט או קלט מגע החומים עברו צג זה

על מנת להתקין את הסביבה יש לבצע את הצעדים הבאים:

- צרו תיקיה חדשה, מומלץ `c:\gvahim`.
- הריצו את הקובץ `gvahim.exe`. קובץ זה יחלץ את קבצי ההתקנה אל התיקיה שנבחרה, `c:\gvahim`.
- בתיקיה שנוצרה נמצא קובץ `install.install`. יש להריץ אותו כמנהל.
- לשם כך, השתמשו בקליק ימני על העכבר ובחרו באפשרות "הרץ כמנהל" או "run as administrator". לאחר מכן, יפתח חלון `cmd` ויתחיל להריץ את ההתקנה באופן שקט.
- אחת ההתקנות תצטרך את התערבותכם. יש ללחוץ `Next`<-`Install`<`Agree`<`Next`, מכאן ואילך הסקריפט ימשיך לבד. בסיום הריצת סקריפט ההתקנה, הוא דואג לנוקוט את הסביבה.
- לסיום, הפעילו מחדש את המחשב.

גרסת הפיתון של ההתקנה היא 2.7 – בהתאם לספר הלימוד. התקינה זו כוללת את כל הכלים הנוספים שנצורך להשתמש בהם במהלך הלימוד (כגון `wireshark`, `scapy`).

את כל התוכנות המותקנות – `wireshark`, `PyCharm`, `python` – ניתן למצוא בספריית ההתחלה או בKİצורי הדריך בשולחן העבודה.

לאחר ההתקנה יהיו בראשותכם שתי שיטות שונות להריצת פיתון. נסקור אותן:

- דרך ה-command line: שיטה זו מתחילה בבדיקה דברים קטנים בפייתון, כגון פעולות מתמטיות, ביצוע help או dir. מאוד קשה ולא מומלץ לתוכנת קוד פיתון של יותר משורות בודדות בדרך זו.
- דרך PyCharm: סביבת עבודה שמצויכה לימוד והתנסות. היתרונות המרכזים שלה – debugging באמצעות breakpoints וחוויי על שגיאות תכנות (כגון שכחה של נקודותים, טעות בשמות פונקציות, כמוות לא נוכנה של פרמטרים לפונקציה וכו'). זו סביבת העבודה המומלצת ללימוד ולכתיבת תוכניות. על מנת להשתמש ב-PyCharm היטב, מומלץ להיעזר במצבה הדרוכה. לינקים לכל מצגות פיתון ורשומות נמצאים בהמשך.

אייקונים

בספר, אנו משתמשים באיקונים הבאים כדי להציג נושאים ובכדי להקל על הקראיה:



שאלה



הגדרה למונח



הדגשת נקודה חשובה



"תרגיל מודרך". עלייכם לפתור תרגילים אלו תוך תוך כדי קריאת ההסבר



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



פתרון מודרך לתרגיל אותו היה עלייכם לפתור בעצמכם



רקע היסטורי, או מידע העשרה אחר



הפניה לסרטון

תודות

ספר זה לא היה נכתב אלףלא תרומתו של עומר רוזנבוים, שסייע לו בגיבוש תוכנית הלימודים בפייתון והן ביצע את העריכה של הספר. כמו כן רבים מהתרגילים בספר נכתבו על ידי שי סדובסקי ועומר רוזנבוים, וניתן להם קרדיט לצד התרגילים.

ברק גונן

תוכן עניינים מצגות פיתון

להלן לינקים למצגות הלימוד של פיתון, אשר עשוות לשיער לכם במהלך לימוד הפרקים השונים.

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

פרק 1 – מבוא ללימוד פיתון

מהי שפת סקריפטים?

ברוכים הבאים לשפת פיתון! שפת פיתון היא שפה שימושית וקלת לשימוש. לאחר שהשתלטו בסיס השפה תוכלו לכתוב תוכניות מסוימות בקלות יחסית. לדוגמה, במספר שורות קוד תוכלו ליצור שני מחשבים לשלה והודעות אחד לשני. בפחות ממאתיים שורות קוד תוכלו לפתח משחק מחשב, עם גרפיקה וצלילים.

שפת פיתון פותחה ב-1990 כשפת סקריפטים. מהי שפת סקריפטים? כדי להבין מהי שפת סקריפטים נctrיך להבין קודם כל כיצד עבדת שפה שאינה שפת סקריפט, לדוגמה שפת C. כל שפת תוכנה צריכה להפוך בדרך כלשהי לשפת מכונה, כדי שהמעבד של המחשב יוכל להריץ אותה. ההבדל בין שפת סקריפט לשפה שאינה שפת סקריפט הוא במסלול שעובר הקוד עד שהוא הופך לשפת מכונה. קוד שנכתב בשפת C נדרש שני שלבים לפני שהמחשב יוכל להריץ אותו: השלב הראשון נקרא קומpileציה והוא מבוצע על ידי תוכנה שנקראת קומפיילר. הקומפיילר ממיר את הקוד לשפת C לשפת אסמלטי. אסמלטי היא שפה שנמצאת רמה גבוהה יותר מעלה שפת מכונה וכי לתוכנתה יש צורך לעבוד ישירות עם החומרה של המחשב. אם אתם רוצים לדעת יותר על שפת אסמלטי, תוכלו פשוט לפתיחת ספר לימוד האסמלטי של גבהים. השלב השני מבוצע על ידי תוכנה שנקראת אסמלר. האסמלר ממיר את הקוד לשפת אסמלטי לשפת מכונה, ככלומר לשפה שהמעבד מבין. בעקבות ההמרה الأخيرة נוצר קובץ הרצה בעל הסיומת exe (קייזור של executable). הנקודה החשובה לזכירה היא שבסיום התהליך נוצר קובץ שמכיל את כל הפקודות שתכתבנו, כאשר הן מתורגם לשפת מכונה.

את השלבים הבאים נוכל לבחון 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 מדפסת "Hello world", ונוסף לה שורה חסרת משמעות – :blablabla

```

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

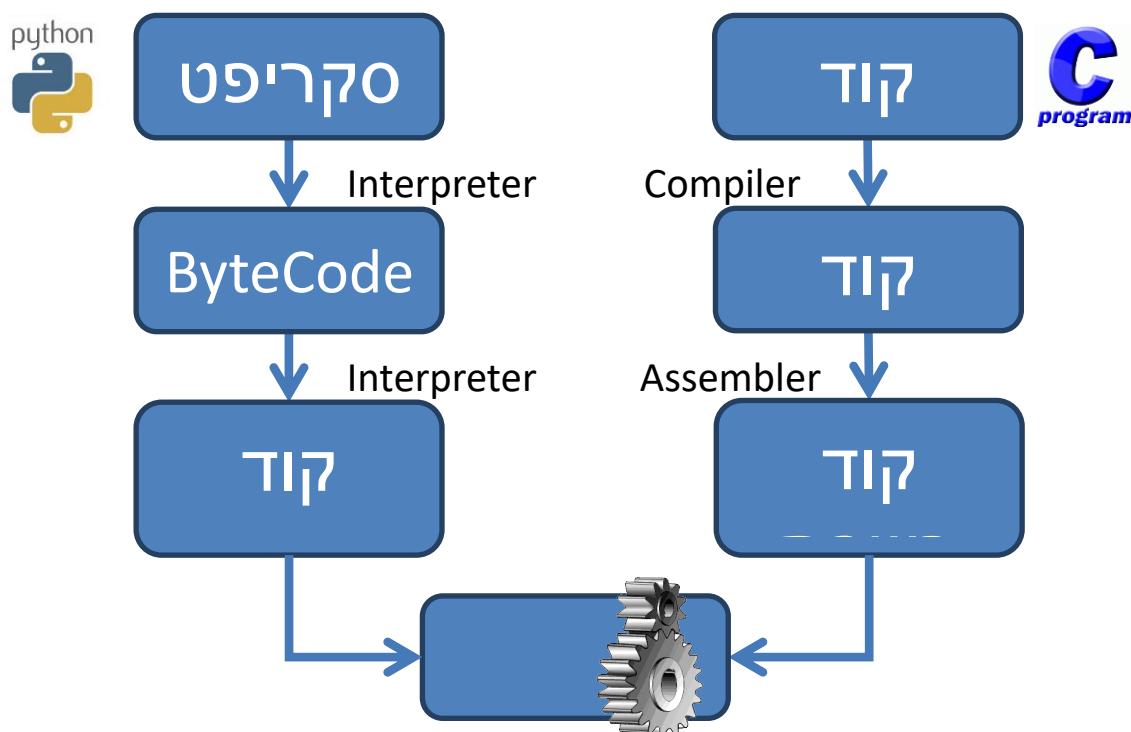
```

כאשר ננסה לבצע קומpileציה, נקבל הודעה שגיאה:

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

במילים אחרות, לא נוכל להדפיס "Hello world", למרות שהשגיאה בתוכנית נמצאת בשורה שאחריה פקודה הדפסה. הסיבה היא שהתוכנית שלנו נכשלת כבר בשלב ההמרה לשפת אסמבלי, ואינה עוברת כלל המרה לשפת מוגנה.

כעת ניבור לדון במתරחש בשפת פיתון, שכאמור הינה שפת סקריפטים. על סקריפט פיתון לפעולת תוכנה שנקראת **interpreter** ("פרשן"). ה-**interpreter** עובד בצורה אחרת למרי מאשר הקומפイルר והאסמלר בהם משתמשים כדי לתרגם את שפת C לשפת מוגנה. הוא אינו יוצר קובץ אסמבלי וגם אינו יוצר קובץ הרצה. במקום זאת, כל פקודה שתכתבו בשפת פיתון מתורגמת לשפת מוגנה רק בזמן הריצה. תוך כדי תהליך הפירוש, נוצר קובץ עם סיומת **.pyc**, המכיל **bytecode** – הוראות שונות של ה-**interpreter** – אך כאמור זה אינו קובץ בשפת מוגנה, ככלומר, מעבד לא מסוגל להרים את הקובץ הזה.



שפת סקריפטים (פיתון) לעומת שפת C

מדובר חשוב לנו לדעת את שלבי ההמרה? כי עצה אנחנו יכולים להבין את הניסוי הבא. הפעם, ניקח תוכנית תקינה בשפת פיתון, שמדפסה "Hello world" ונוסיף גם לה `:blablabla`

```

1 print 'Hello world'
2 blablabla

```

כאשר נריץ את התוכנית יתקבל הפלט הבא:

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

מה קיבלנו? השורה הראשונה, שמדפסה Hello world למסך, בוצעה בהצלחה. לאחר מכן מכן הטענו תקין את השורה blablabla, ניסיון שהסתיים בשגיאת הריצה. הנקודה המעניינת היא שהתוכנית רצתה באופן תקין עד שארעה השגיאה, וזאת בגיןו לתוכנית זהה בשפת C, שכלל לא רצתה. הסיבה שהצלהנו להציג לפיתון לנוקודה שהתקל מהתוכנית רצתה, היא לבדוק בגל תהליך ההמרה השונה. בפייתון לא נוצר קוד בשפת אסםביי וגם לא קובץ הריצה, ולכן השגיאה לא התקלטה עד לנוקודה שבה היה צורך לתרגם את blablabla חסר המשמעות לשפת מכונה. רק אז הבין ה-*interpreter* שיש כאן בעיה ועזר את ריצת התוכנית.

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 כתוב 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 מנגן כמו הקומפイルר של C, שמנע מאייתנו לכתוב קוד שלא עומד בכללי התהբיר של השפה ולבן הסיכוי לעיוות בזמן ריצה הוא קטן יותר. התرسוקות של קוד תוך כדי ריצה היא חמורה לאין שיעור מאשר שגיאת קומפילציה, אותה ניתן לגלוות ולדבג לפני הריצה. שנית, העובדה שקוד בשפת C מתורגם לשפת מכונה לא שורה אחר שורה אלא קובץ אחד, מאפשרת לבצע הילכתי ייעול (אופטימיזציה) של הקוד, כך שהוא עשוי לזרז יותר מהר ולצרוך פחות זיכרון מאשר קוד מקביל בשפת פיתון. זה דבר חשוב למי שרווצים להריץ אפליקציות "כבדות", כגון גרפיקה מורכבת או הצפנה.

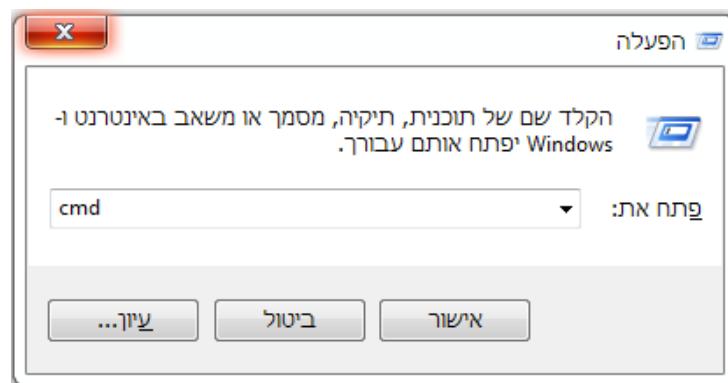
לסיכום, אפשר לומר שפיתון היא שפה נוחה וקלת כתיבתה, שמאפשרת להציג תוכניות עובדות, גם אם הדבר בא על חשבון מהירות או יעילות. בשל כך זכתה פיתון למפתחים נלהבים רבים, שדגאו לכתב מודולים – קטעי קוד שמבצעים משימות שונות – ולהפין אותם. אחד היתרונות העיקריים של פיתון הוא שניתן לקבל מן המוכן קוד בשפת פיתון שמבצע משימות רבות: חישובים מתמטיים, גרפיקה, ניהול קבצים במחשב וכל דבר שנייתן להעלות על הדעת.

עבודה דרך command line

נפעיל את פיתון בדרך הקצירה והמהירה. 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 זמנית על מקש ה-"*R*:



ויפוי מסך "הפעלה" או באנגלית `ctrl`, ובתוכו כתבו את שם התוכנה שאתם רוצים להפעיל. במקרה זה – `.cmd`.



עם לחיצה על מקש `enter` הגיעו למסך ה-*command line* `command`. במסך יהיה כתוב שם התקינה בה אתם נמצאים, לדוגמה `c:\cyber`. כתע כתבו `python` והקישו `enter`. ברוכם הבאים לפיתון!

```
c:\\>python
Python 2.7.13 (v2.7.13:a06454b1afaf, Dec 17 2016, 20:53:40) [MSC v.1500 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

אנו יכולים לכתוב כל תרגיל חשבוני שאנו רוצים, לדוגמה $4+5$, ונקבל מידית את התשובה. אנו יכולים אפילו להגיד דברים אחרים, למשל $a+b=5$, וכאשר נכתוב $a+b$ נקבל את הסכום שלהם. נסו זאת!



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

מספרים בסיסיים שונים

ניתן כאן הסבר מוקוצר מאוד לשיטות הבינאריות והקסדצימליות. הנושא זהה מכוסה בהרחבה בספר האסמבלי של גבאים, ואם איןכם שולטים בסיסי ספירה מומלץ להניח מעט לספר הפיתון וללמוד בסיסי ספירה לפני המשיכו.

אנו בני האדם סופרים בסיס 10, וזה דבר טבעי בעיני כיון שיש לנו עשר אצבעות. מחשבים סופרים בסיס 2, ביןاري. בסיס זה קיימות רק הספרות 0 ו-1. אין במחשבים שום ספרות חזק מאשר 0 ו-1.



כיוון שלבני אנוש מסובך מעט לקרוא רצפים ארוכים של אחדות וঅস্ফিম, מקובל להציג מידע ששומר בזיכרון המחשב בספרות הקסדצימליות – בסיס 16. זאת משומש שכל ספרה הקסדצימלית מייצגת 4 ספרות בינאריות (שימו לב ש-16 הינו 2 בחזקת 4, ולכן כל זה מתקיים). לדוגמה את הרצף:

1011 0001 1000 0010

ניתן לכתוב בספרות הקסדצימליות:

B182

...חרבה יותר קצר לקריאה!

אם נרצה לכתוב בפייתון מספרים בינאריים, עליהם להתחיל ב-0b. לדוגמה 0b11 הינו 3, ו-0b111 הינו 7. כדי לכתוב מספרים הקסדצימליים בפייתון, נוסיף להם תחילית x.כך לדוגמה, 0x1A הינו 26, ו-0x2B הינו 43. אפשר לראות זאת בклות אם נכתבו אותם בסביבת הפיתון:

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

Help



תרגיל מודרך

כיתבו בפייתון 2 בחזקת 7.

לא יודעים איך לכתוב חזקה בפייתון? כתבו `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>
```

נוכל לבחור בעזרה בנושאים הבאים: **Topics, Keywords, Modules** ונקבל את הרשימה הבאה:

אם נכתוב **POWER** (המילה האנגלית לחזקה) נקבל את התיאור המלא של **POWER**, שם נגלה ש כדי להעלות בחזקה עליינו 2^{**7} הינו 2^{**7} בחזקת 7. לדוגמה, 2^{**7} בסימן $*$.

כדי לצאת מ-`help` נכתוב `quit` ונגיע חזרה להלון הפקודה שלנו. נסו כעת להשתמש במקרה של מדרנו!

```
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 התיכון)

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



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

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

כל? כתעתשוו את הביטוי הבא:

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

דרך אחת היא פשוט להעתיק את כל התרגיל ממחתולת. אולם כפי שהבחנתם, מדובר במשתנה, מדויב למשהו באותו תרגיל פרט לכך שהוא בחזקת 2. כדי למצאו את הפתרון באופן אלגנטי ובלי הקלדות מיותרות פשוט רישמו את הביטוי הראשון, לחצו *enter* ובסורה הבאה כתבו:

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

פירושו של הקו התיכון – "כח את התוצאה האחרונה שהיישבת". באופן זה נוכל להסוך זמן כתיבה ולבצע חישובים בקלות. שימו לב שהטריך 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 נctrarך תוכנת עריכה כלשהי. בטור התחלה מומלץ להשתמש ב-*notepad++*, אשר ניתן להורדה בחינם מהאינטרנט. לא נשמש בו הרבה, אך שם אתם מудיפים לא להתקין אותו תוכלו להשתמש בתוכנת *notepad* שmaguya עם הלוונוט. איך מפעלים? בדוק כמו שלמדנו להפעיל כל תוכנה. ליהצו על *R*, *winkey+R* כתבו *notepad* בחלון הפעלה זהה.

את הקובץ שיצרתם שימרו עם סימת *py*, ככלומר 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 מתיק ה- command line להריץ את קובץ הפיתון שלנו. ראשית יש להציג אל התקינה שבתוכה שמרנו אותו. לדוגמה, אם שמרנו את הקובץ בתוך c:\cyber או עליינו כתוב:

```
cd c:\
```

```
cd cyber
```

לאחר שהגענו לתיקיה הנכונה, נכתב:

```
python hello.py
```

והקובץ שלנו יורץ מיד ☺

סיכום

בפרק זה רכשנו את הבסיס לתוכנות פיתון. אנחנו מבירנים שפיטון היא שפת סקריפטים, אנחנו יודעים להריץ פקודות פיתון פשוטות גם דרך ה-command line וגם דרך קבצי פיתון שיצרנו. אנחנו יודעים איך לחפש בפייטון עזרה על כל נושא שנרצה. למעשה, עם הידע שהקימים אצלנו בשלב זה כבר אפשר להסתדר לא רע – הרי בסופו של דבר אנחנו יודעים לכתוב תוכניות ולהריץ אותן, וכמובן גם יודעים לחפש עזרה בנושאים לא מוכרים. מעכשיו, נוכל למצוא הדרך 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. חסרי לנו סביבת עבודה שמאפשרת להריץ את הקוד מתוך הסביבה וכוללת דיבאגר. הכוונה בדייגר היא תוכנה שמסוגלת להריץ את הקוד שלנו פקודה אחריה פקודה, תוך כדי הצגת ערכי המשתנים השונים. זו יכולה להיות שימושית להבין מדוע תוכנית שכתבנו לא עובדת באופן תקין.

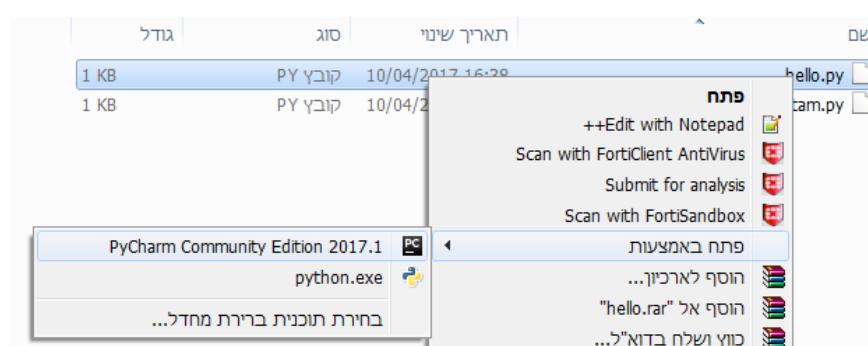
בחרנו להציג בפניכם את סביבת העבודה **PyCharm**. היכולות שיש给她 של PyCharm הן:

- כתבן (תחליף ל-**notepad++**).
- איתור שגיאות אוטומטי – כן, PyCharm מוצא בשbillינו את השגיאות בקוד שלנו. אין הכוונה לכך ש-**PyCharm** יודע לתקן עבורנו באגים באלגוריתם, אך PyCharm ביכולת יגלה לנו אם שכחנו לשים נקודותים, או שהשתמשנו במשתנה שכחנו לתת לו ערך בהתאם.
- דיבאגר – כאמור, יכולה להריץ את הקוד שלנו פקודה אחר פקודה (**step by step**), תוך הצגת ערכי המשתנים.
- יכולה לאותה בקהלת תוכנית לאחר קריסה או באג – נוכל לחסוך זמן שימושי כאשר התוכנית שלנו מתרסקת, על ידי כך שבמוקם לאותה התוכנה פשוט נורה ל-**PyCharm** להריץ אותה שוב.

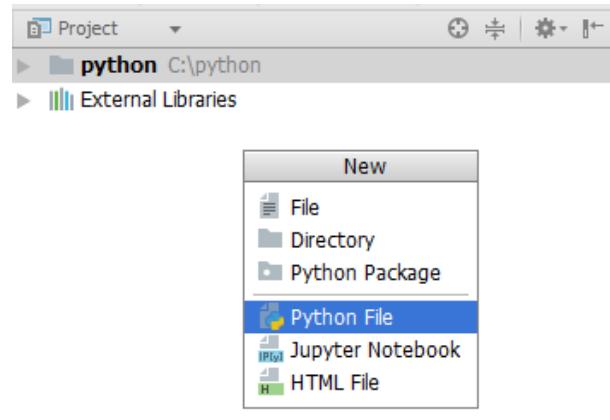
כעת נלמד להשתמש בסביבת העבודה **PyCharm**.

פתיחת קובץ פייתון

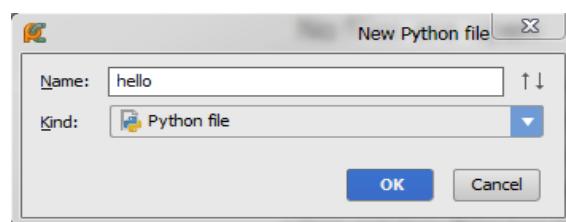
אם הקובץ כבר קיים, כל שנצחטרך לעשות הוא להקליק עליו קלייק ימני ולחזור באפשרות **open with** ולאחר מכן **.Community Edition**



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



לאחר מכן ניתן לקרוא החדש שם:



וכעת ניתן לעורר את הקובץ החדש שיצרנו:

```

PC python - [C:\python] - C:\Gvahim\barak\hello.py - PyCharm Community Edition 2017.1
File Edit View Navigate Code Refactor Run Tools VCS Window Help
C: > Gvahim > barak > hello.py
Project python C:\python
External Libraries

hello.py x
main()
1 __author__ = 'Barak'
2
3
4 def main():
5     print 'Hello'
6
7
8 if __name__ == '__main__':
9     main()
10

```

The screenshot shows the PyCharm interface with a project named 'python' containing a file 'hello.py'. The code editor displays the following Python script:

```

def main():
    print 'Hello'

if __name__ == '__main__':
    main()

```

קובץ הפיתון הראשון שלנו

נסקרו את השורות בקובץ הפיתון הראשון שלנו:

```

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

```

בשורה 1 נכתב את שם המחבר. זה חשוב מאוד – 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 שכבר מהשם שלו, זו הפונקציה הראשית של התוכנית שלנו. המשמעות היא שזו הפונקציה הראשונה שנקראת (מיד נבין מי קורא לה) ולכן כל מה שאנו רוצים שיבוצע צריך להיות כתוב בתוך `main`, או לחלופין ש-`main` תקרה לו.

בשורה 8 ושורה 9 יש תנאי מעט מסובך, שבמהמשך הלימוד נבין מה פירושו. בקצרה – אנחנו יכולים להריץ סקריפט פיתון בשתי דרכים. האחת, היא פשוט להריץ התוכנה, היא להריץ סקריפט אחר שקורא לסקריפט שלנו. הדרך השנייה מתבצעת בעזרת פקודה שנראית `import` ונלמד עליה בהמשך. נניח שתכתבנו סקריפט פיתון שמכיל כמה פונקציות מעניינות, וחבר שלנו רוצה להשתמש בהן. אם החבר יקרא לסקריפט שלנו על ידי פקודת `import`, אז עלול להוזר מצב שבו כל הקוד שתכתבנו רץ. כמובן, במקומות שהתוכנית שלו פשוט תזכיר 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 לפיתון "שמע פיתון", יכול להיות שהסקריפט הזה יצורף לסקריפט אחר. לכן קודם כל תבדוק אם מי שמריז את הסקריפט זה לא מישחו אחר שעשה לו `import`, ורק אם הסקריפט אינו מזמן-`import` או תקרא לפונקציה `main` שתחל את ריצת הסקריפט".

סדר ההרצה של פקודות בסקריפט פיתון

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

ה-interpreter של פיתון, שתפקידו לתרגם את פקודות הפייטון לשפת מכונה, מփש את הפקודה הראשונה שצמודה לצד שמאל והוא בודק אם לא מדובר בהגדירה של פונקציה. לכן, בשורה מספר 1 – שצמודה לצד שמאל – פיתון יבצע את הפוקודה. לאחר מכן בדקה מספר 4 פיתון ילמד שינוי פונקציה בשם main. פיתון יוסיף את main לרשימת הפונקציות המוכנות לו. ככלומר בשלב זה אפשר לקרוא ל-`main`, אך אין זה אומר שהפייטון מרים את `main` או אפילו בודק שהקוד של `main` הוא תקין. כל מה שידוע לפיתון – ישנה פונקציה `main`.

רק בשורה 8 נתקל פיתון בפקודה שעליו להריץ ושמבוצעת ממשו. אם התנאי שבשורה מתקיים, פיתון ממשיך אל שורה 9 ושם נאמר לו להריץ את הפונקציה `main`. בשלב זה פיתון יקפוץ אל `main` ויחל לבצע את מה שנכתב בה. מאידך, אם התנאי שבשורה 8 אינו מתקיים, פיתון הגיע לסוף הסקריפט ויריצה הסקריפט תסתיים.

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

שםו לב לכך, שזו היא תוכנית פיתון הגיונית וצורת הכתיבה של הקוד לא משקפת בשום אופן צורת כתיבה מומלצת של קוד. המטרה של הקוד הבא היא רק להמחיש את סדר שלבי ריצת סקריפט הפייטון. היה בכך הרבה יותר לנכון לכתוב את כל פקודות הדפסה שלנו בתוך פונקציית `main`. אם כך, מה יהיה סדר הפקודות 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__`.

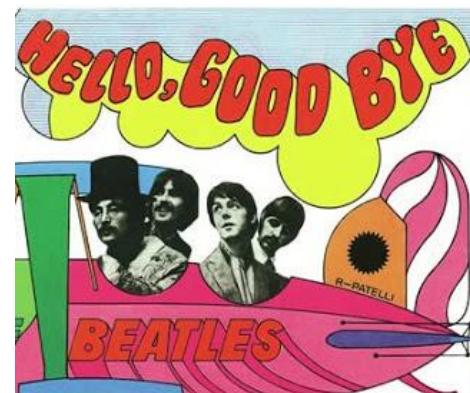
שורה 2 תדפיס למסך את הטקסט "You say.". שמתם לב לכך שיש פסיק בסוף פקודת הדפסה? בלי הפסיק, לאחר `print`, לאחר מכן תבוצע רידת שורה. הפסיק קובע שלאחר הדפסה יהיהתו אחד של רווח.

לאחר מכן תבוצע שורה 8, יודפס "Goodbye", עם רידית שורה בסופה.

בעקבות התקיימות התנאי בשורה 10, יבוצעו שורות 11 ו-12. שורה 11 תדפיס למסך I say Hello. שורה 12 תקרה לפונקציה main. בתוך הפונקציה main תבוצע הדפסה של Hello.

לאחר שהפונקציה main תסימן את ריצתה, אליה היא נקרה משורה 12, היא אמורה להזור ולהריץ את המשך התוכנית – אילו היה זה – אחרי שורה 12. כיוון שורה 12 היא סוף התוכנית, בכך יסימן 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

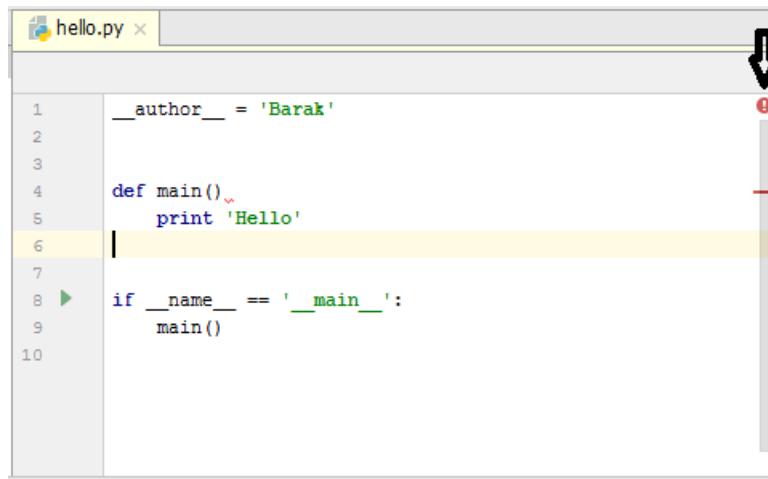
```

python - [C:\python] - C:\Gvahim\barak\hello.py - PyCharm Community Edition 2017.1
File Edit View Navigate Code Refactor Run Tools VCS Window Help
C: > Gvahim > barak > hello.py
Project python C:\python
External Libraries
hello.py
main()
1 __author__ = 'Barak'
2
3
4 def main():
5     print 'Hello'
6
7
8 if __name__ == '__main__':
9     main()
10

```

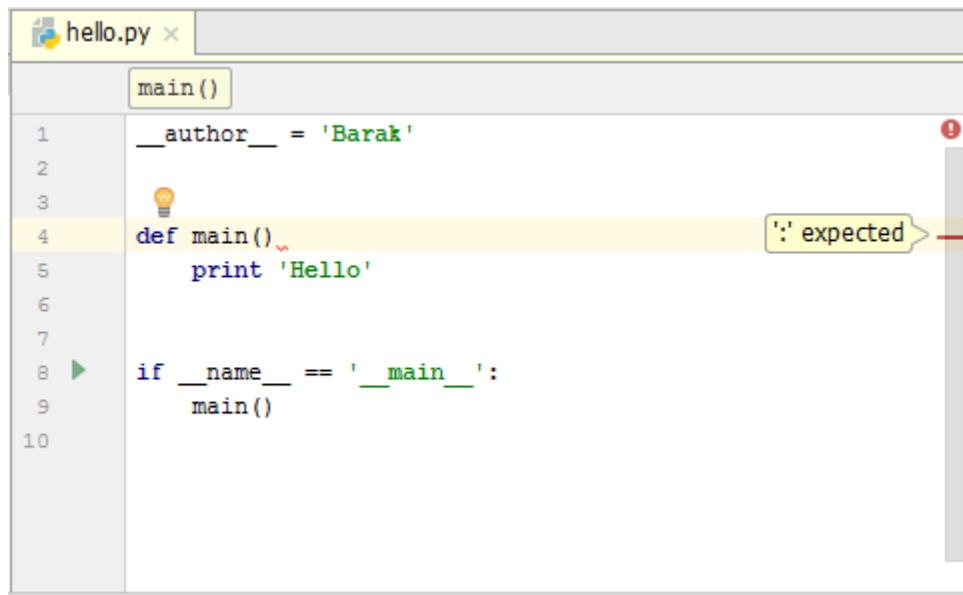
זהו סימן שהתוכנית שכתבנו אינה מכילה שגיאות. שימוש לב מה קורה כאשר אנחנו מוחקים את סימן הנקודותים שאחרי המילה

:main



```
hello.py x
1 __author__ = 'Barak'
2
3
4 def main():
5     print 'Hello'
6
7
8 if __name__ == '__main__':
9     main()
```

קיבלנו סימנים אדומים במספר מקומות בmseך. ראשית, במקום הנקודותים שהקנו מופיע קו תחתי אדום. שנית, ה-V הירוק הפוך לסתימן קריאה לאדום. שלישית, אם נעמוד עם העכבר על הקו האדום שבצד ימין, נקבל הסבר מה הבעה בקוד שלנו.



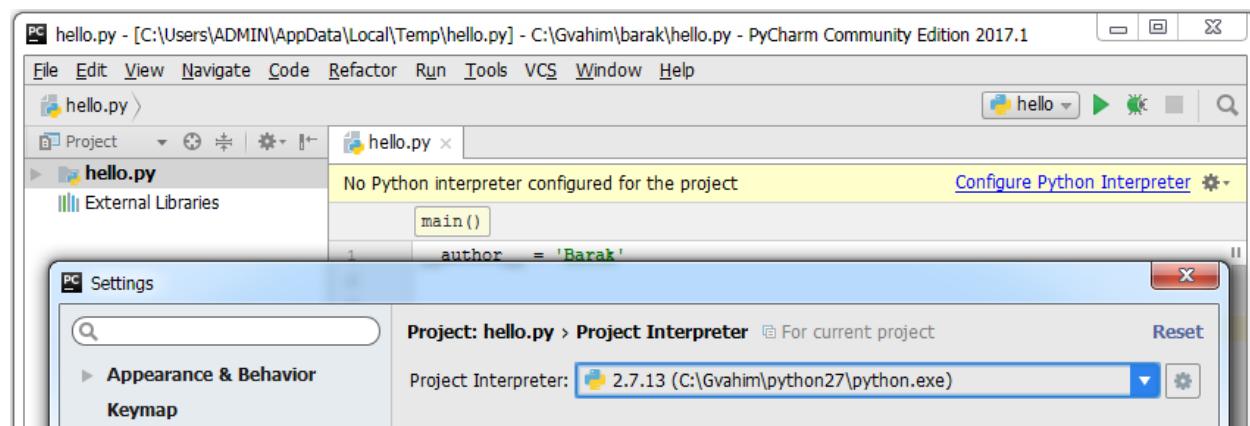
```
hello.py x
main()
1 __author__ = 'Barak'
2
3
4 def main():
5     print 'Hello'
6
7
8 if __name__ == '__main__':
9     main()
```

קליק שמאלiy על הקו האדום מימין גם יוביל אותנו בדיק לשורה הביעיתית. בKİצ'ור, כל מה שאנו צריכים בשביב להימנע!
משגיאות קוד ולפתור אותן בקלות!

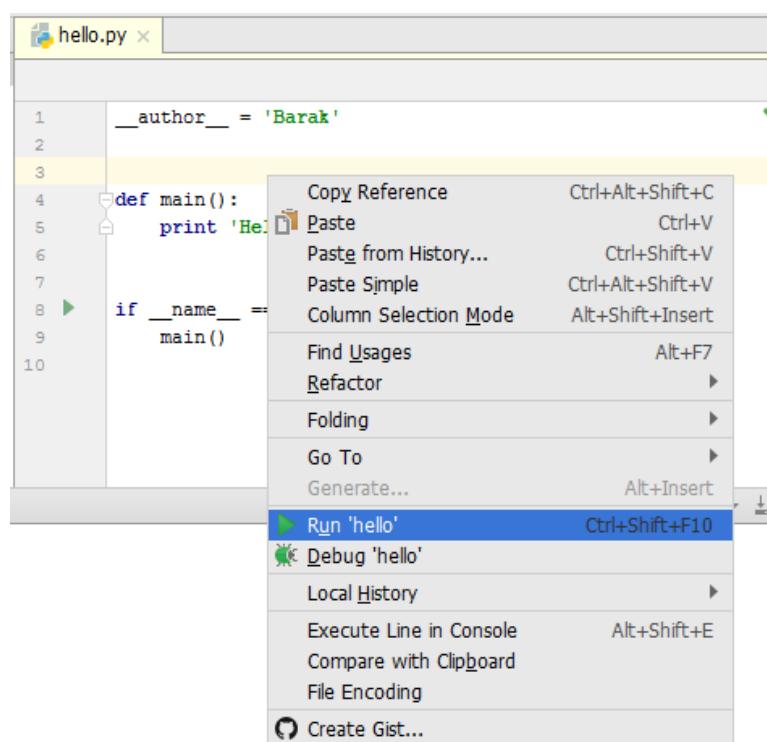
הרצה הסקריפט ומסך המשחמש

סבירת ההתקנות של גבהים אמורה להתקין ולאפשר לכם לעבוד מיידית בלי צורך בהגדרות נוספת, אולם אם מופיעה לכם שגיאה מסוג "No Python interpreter configured for the project", כפי שאפשר לראות מודגש בצד שמאל בתמונה, להלן הדרך לסדר את ההגדרות. ראשית צריך להציג PyCharm-Interpreter ל- PyCharm Configure Interpreter, זאת מכיוון שיש גרסאות שונות של שפת פיתון, וייתכן שעל אותו המחשב מותקנות גרסאות שונות. לכן, נקליק הכיתוב

Configure Python Interpreter ולאחר מכן נבחר את גרסת הפיתון שモתקנת במחשב שלנו, אשר מוגיעה עם ההתקנה של סביבת גבהים:

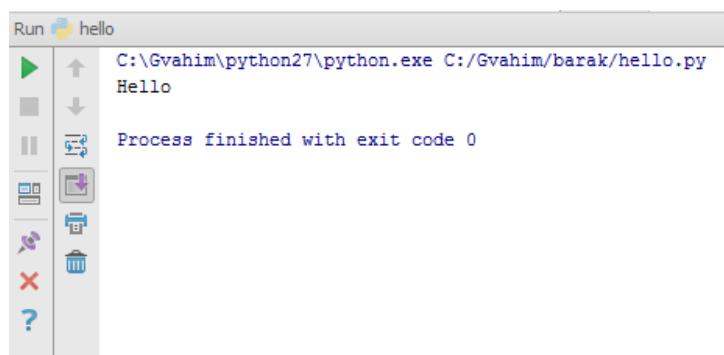


כדי להריץ את הסקריפט שכתבנו, נלחץ על עכבר ימני ונבחר באפשרות חוץ:

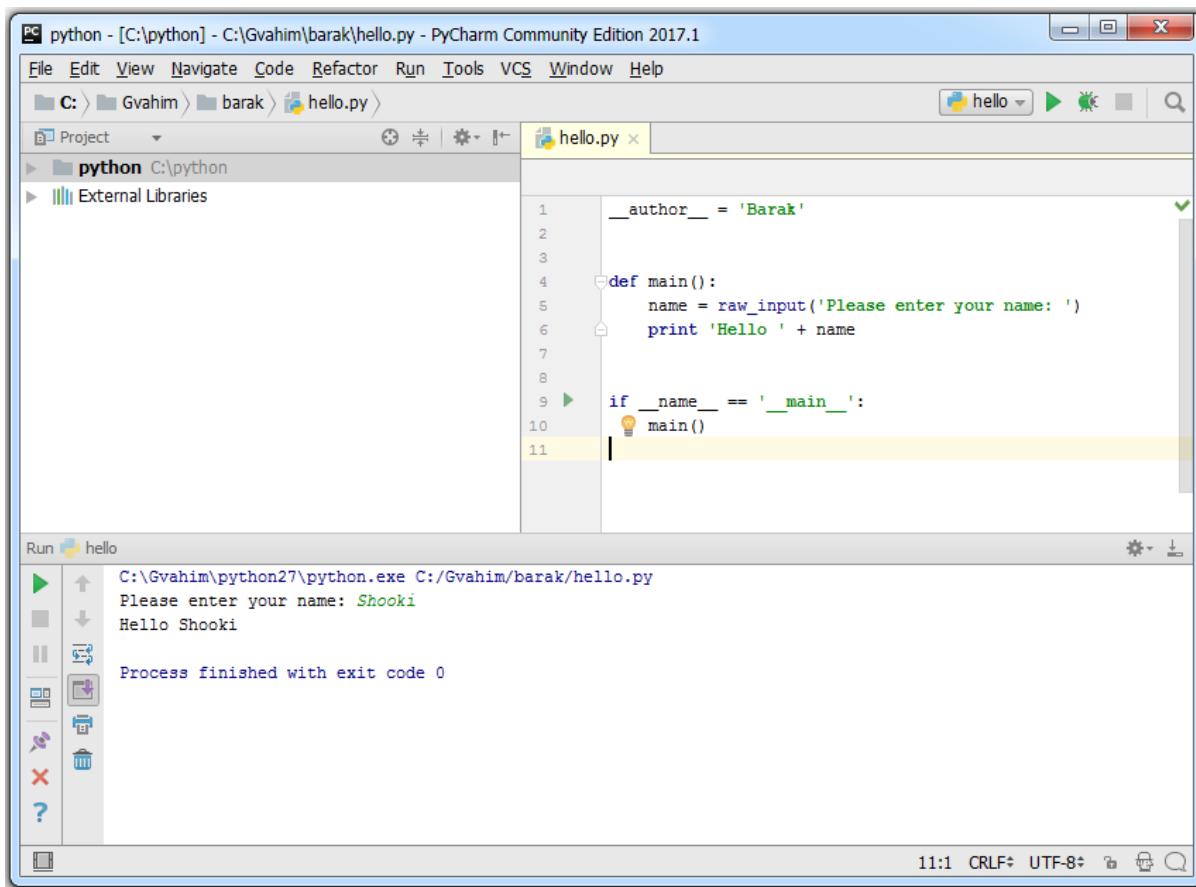


בקבות הלחיצה על run יופיע מסך משתמש, שם יודפס כל מה שהסקריפט שלנו מדפיס למסך. רואים את ה-"Hello" שעשינו

לו print ?



מסך המשתמש משמש גם לקבלת קלט מהמשתמש. שימו לב לפקודה שבסורה 5 :raw_input ,5



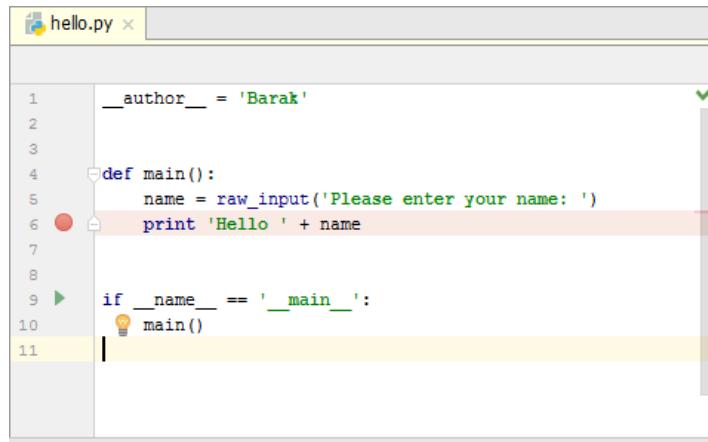
פקודה זו מדפיסה למסק את מה שנמצא בסוגרים, וכל מה שהמשתמש מקליד נכנס לתוך המשתנה, במקרה זה קרנו לו name .

בשורה 6 מתבצעת הדפסה של Hello ולאחר מכן הערך שנמצא בתוך המשתנה name . התבוננו בחלק התיכון של המסק ובידקו שאתם מבינים את תוצאת הריציה.

דיבוג עם PyCharm

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

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



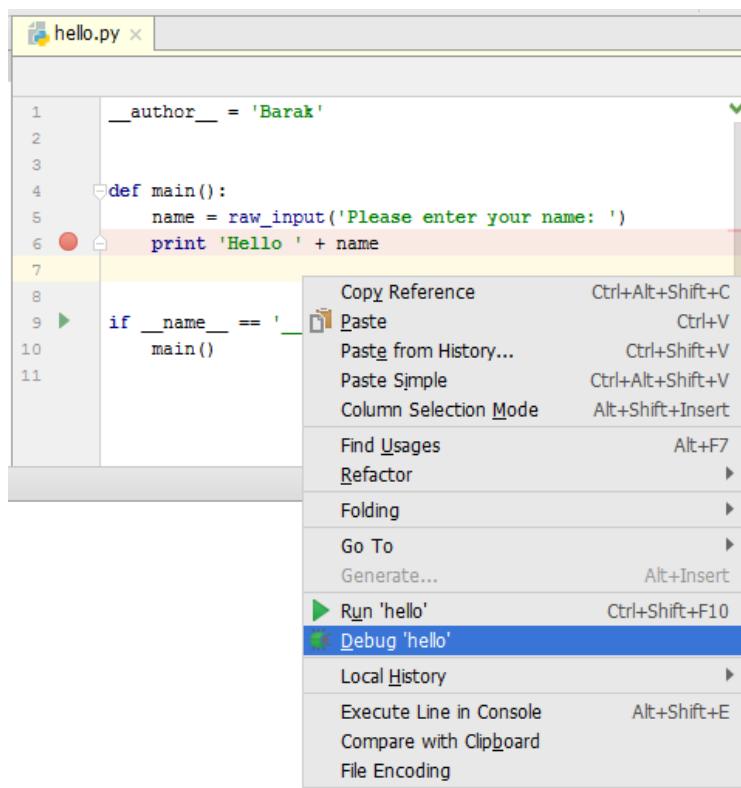
```
hello.py
```

```

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:



```
hello.py
```

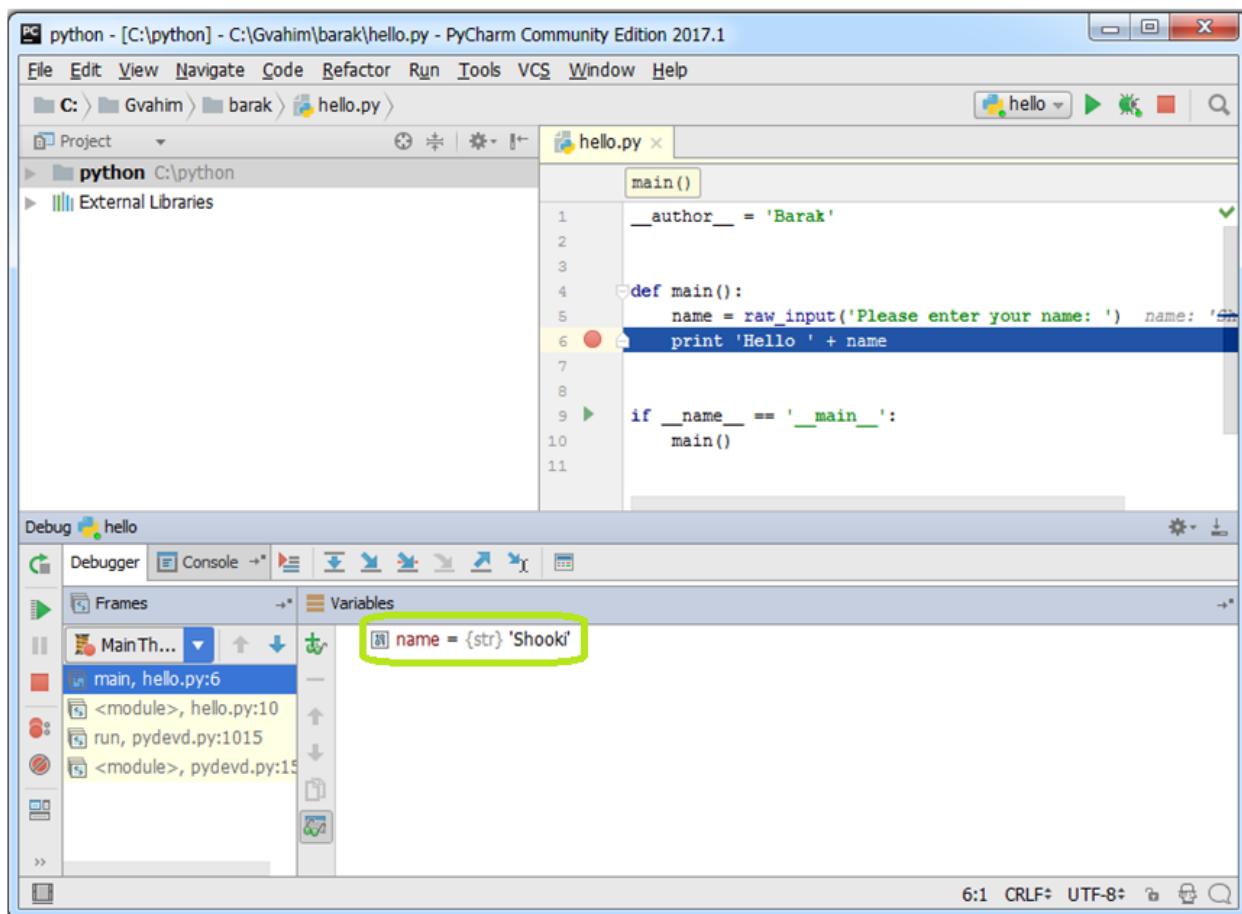
```

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

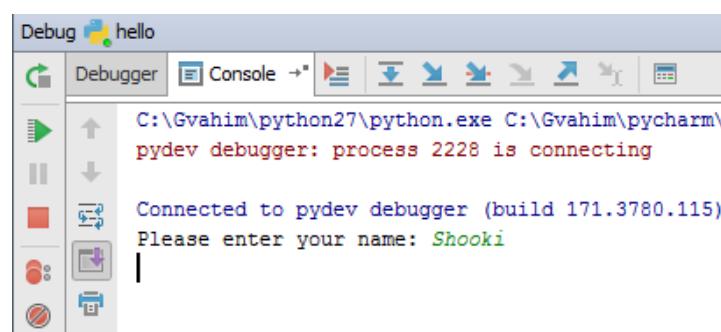
```

Copy Reference Ctrl+Alt+Shift+C
 Paste Ctrl+V
 Paste from History... Ctrl+Shift+V
 Paste Simple Ctrl+Alt+Shift+V
 Column Selection Mode Alt+Shift+Insert
 Find Usages Alt+F7
 Refactor ▾
 Folding ▾
 Go To ▾
 Generate... Alt+Insert
 Run 'hello' Ctrl+Shift+F10
Debug 'hello'
 Local History ▾
 Execute Line in Console Alt+Shift+E
 Compare with Clipboard
 File Encoding

לאחר מכן נראה כי התרחשו מספר דברים. ראשית, התוכנית רצתה עד לשורה עם הנקודה האדומה, ה-breakpoint, ושם עצמה. השורה סומנה בכחול. שנית, נפתח לנו חלון ה-debugger, אשר מכיל מידע על המשתנים שוגדרים בתוכנית שלנו. לדוגמה, המשתנה name הוא מסוג str (במהשך, נבין את המשמעות של סוגי המשתנים השונים) וערךו הוא Shooki, הערך שהמשמש הוזן לתוךו בשורת קוד מספר 5. ניתן לראות את הערך של name בחלון ה-variables, לנוכח הנסיבות הוא מודגש במסגרת צהובה:



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



נחזיר אל חלון הדיבאגר. כפי שניתן לראות ישנו היצים מסווגים שונים. להיצה על היצים תקדם את התוכנית שלנו. אם נעמוד עם העכבר על היצ כלשהו יופיע כיתוב שմסבירות מה היצ עושה. בשלב זה מומלץ להשתמש בחץ Step Over, השמאלי ביותר. נוח מאד להשתמש בקיצור המקלדת שלו – F8. היצ זה מרים שורת קוד אחת בכל פעם, שורה אחר שורה. כאשר נכתוב פונקציות, נרצה להשתמש לעתים הקרובות גם בחץ השני משמאלו, Step Into. קיצור המקלדת שלו הוא F7.



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



מומלץ לשЛОט בפעולות אלו, מכיוון שבבוא העת שימוש נכון בהן מאפשר לכם לגלוות בקלות בעיות בתוכנה שלכם!

העברה פרמטרים לסקריפט

دلגו על החלק הזה, והזירו אליו רק לאחר שתלמידו על sys.argv.

כדי להעביר פרמטרים לסקריפט, ככלומר מידע שהמשתמש מעביר לסקריפט לפני תחילת הריצה (לדוגמא, שם של קובץ :edit configurations שסקריפט צריך לעשות בו שימוש), נקליק על שם הקובץ שנמצא מצד ימין למעלה, ואז נבחר

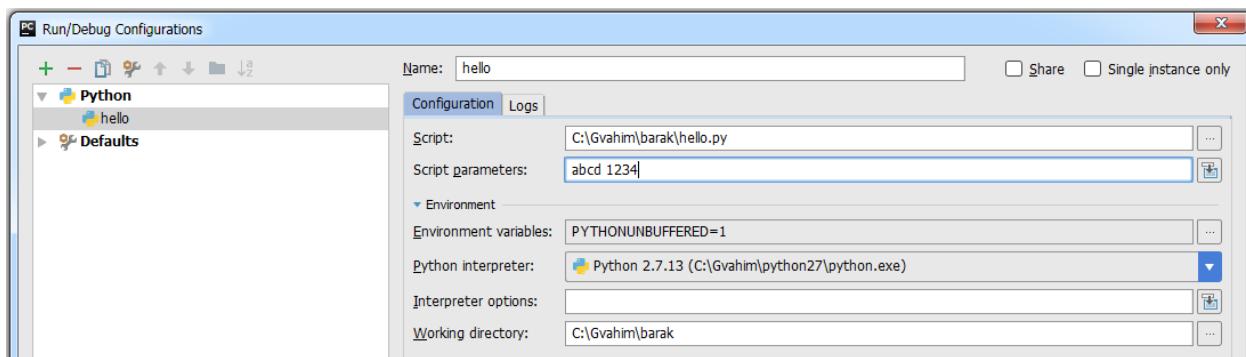
```

hello.py x
main()
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

```

כעת יפתח מסך שבו ניתן להזין לתוכו כמה פרמטרים שנרצה, עם סימן רווח בין:

פרמטר אחד לשני:



סיכום

בפרק זה סקרנו את סביבת העבודה PyCharm, שתשתמש אותו בהמשך. למדנו כיצד לטעון קובץ פיתון, לעורך אותו, להריץ אותו ולדbg 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.

כיתבו ב-`interpreter`:

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

אפשר גם לכתוב בקיצור:

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

בפייתון יש סוגים של משתנים רבים, לא רק `int` כਮובן, אנחנו נסקרו אותם אחד אחד, כאשר בפרק זה נתחיל מטיפוסי המשתנים הפחותים ביותר ובפרקים הבאים נסכח טיפוסי משתנים נוספים.

שימוש לב לסוגי המשתנים 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 שחייבים לשמר על האות הגדולה בתחילת המילה, אם נכתב true או false נקבל שגיאה. מכאן אפשר להסיק שפיטון היא שפה שבדילה בין אותיות גדולות לקטנות ("case sensitive" מה שנקרה"). לעומת זאת, המשתנה  a שונה מהמשתנה A.

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

True + 1

פיתון ידפיס לנו את התשובה – 2.

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

משתנה מסוג int, float: מספר הוא או מtipos int (אם הואשלם) או מtipos 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` אי דיק זעיר, שנובע מכך שבסתופו של דבר הם נשמרים בזיכרון המחשב באמצעות חזוקות 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 חלקי 7 באמצעות סופית של ביטים. אי הדיק זהה גורם לכך שאם נעשה פעולה חשבונית שיגרמו להצברות השגיאה, לבסוף נקבל תוצאה שהיא ברור שהיא אינה מדויקת. הנה, כאשר נחבר 0.1 עם עצמו מספר פעמים, ניווכח שההתוצאות אינן "עגולות" כפי שהיא צפוי:

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

זהו, סיימנו את הדיון 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 שלאחר ה-`if`, ואם ערך הביטוי הוא `False`, אז יבוצע דילוג. מיד נבין עד להיכן מבוצע הדילוג. בינהיים, נסקור את הדרכים השונות לבדוק את היחס בין משתנה לביטוי כלשהו.

שווין – הסימן **==** (פערמים =) פירושו "האם צד שמאל של הביטוי שווה לצד ימין". הקוד

בדוגמה הבאה ידפיס!: Yay!

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

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

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

גודול / קטן / גודול שווה / קטן שווה – כל אחד מהסימנים **>**, **<**, **>=**, **<=** בודק אם התנאים מקיימים את היחס שモגדר 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** או את שם המשתנה:

```
x = True
if x:
    print 'Got it? :)'
```

תנאים מורכבים

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

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

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

ואפשר גם לכתוב את אותו ביטוי עם סוגרים, לעיתים יותר קל לקרוא אותו כך:

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

כמובן שאט התנאי האחרון אפשר לכתוב גם בצורה מקוצרת, באופן הבא:

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

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

```
x = 31
if x == 21 or x > 30:
    print 'Yay!'
```

תנאי `not` מבצע היפוך. מקובל להשתמש בו יחד עם `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-`==`

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

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

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

בפירטונו הבא יש הדוגמה של ההבדל בין `so` לבין `==`:

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

בלוק

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

```
x = 21
if x == 21:
    print 'Hi!'
    print 'X is....'
print '21'
```

כל שלוש הפקודות שללאחר ה-`if` נמצאות בהזחה – אינדנטציה – של טאב אחד, או ארבעה רווחים יחסית לתנאי `if`. כיוון שגם נמצאות באותו הזוג, הן יבוצעו ככל אם התנאי יתקיים. שימושו לב להבדל בין הקוד לעיל לבין הקוד הבא:

```
x = 20
if x == 21:
    print 'Hi!'
    print 'X is....'
print '21'
```

השורה האחורונה, שמדפסה 21, תרוץ בכל מקרה – בין שה坦אי מתקיים ובין שהוא אינו מתקיים. זאת מכיוון שהיא נמצאת מחוץ לבלוק של תנאי `if`.

נושא ההזחה הוא קרייטי בפייתון, מכיוון שבניגוד לשפות אחרות בהן יש סימונים שונים של "סוף בЛОק", לדוגמה סוגרים מסולסלים, בפייתון סוף הבלוק נקבע רק לפי ההזחה. על כן הכרחי גם שהזחה תהיה עקבית – אנחנו לא יכולים לעשות לפעמים הזחה של שני רוחחים ולפעמים של ארבעה רוחחים. חשוב לציין גם שתוווי רוחה וטאב שונים זה מזה. ככלומר, גם אם לנו נראה ששורה שיש בה טאב נמצאת בהזחה זהה לשורה אחרת שיש בה הזחה של ארבעה חוווי רוחה, מבחינה פיזית אין אלו תווים שונים לגמריי. לכן, אם התחלנו לעשות הזחה עם טאים, רצוי שנמשיך רק עם טאים.

לא רק מה שנמצא בהזחה של טאב אחד שייך לבLOCK. בתוך BLOCK יכולים להיות עוד טאים, לדוגמה עקב שימוש בתנאי if נוספים. כל הפקודות שנמצאות בהזחה של לפחות טאב אחד מתנאי ה-if שלנו שייכות לאותו BLOCK. 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 :)'
```

シמו לב לכך ש-**PyCharm** מסיע ו"מסמן" את הבלוקים השונים באמצעות פסים אפורים דקים. מה יודפס כתוצאה מ竞选 הблוק? יורצו כל השורות חזן מאשר אלו שנמצאות בבלוק של `x < 21` או, כיוון שתנאי 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. במקום שנצרך לבדוק פעמים – פעם אם התנאי מתקיים ופעם אם הוא אינו מתקיים – אפשר להשתמש בפקודה `else`. אם התנאי שנמצא ב-`if` אינו מתקיים, יבוצע הקוד בבלוק של `else`.

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

מה אם יש תנאי נוסף ווותיקים לבודוק? לדוגמה, אנחנו מתקשרים לחבר כדי לבוא איתנו לסרט. אם הוא לא יכול, אנחנו מתקשרים לאמא ושותאים מה יש לאכול בבית. אם אמא לא עונה – אנחנו נשארים בבית וצופים בשידור החוזר 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 שבסיום אכלנו עוגת תפוחים של אמא. ברגע שתנאי ה-if-
התקיים, התוכנית לא נכנסת אל תוך תנאי ה-else. נצפה בטעות רק אם גם תנאי ה-if וגם תנאי ה-elif לא יתקיימו.

תרגיל



כתבו סקריפט עם תנאים על המשתנה `age`. אם `age` שווה 18, הסקריפט ידפיס 'Congratulations'. אם `age` קטן מ-18 ידפס 'We love old people', 'You are so young' או 'You are so young'après. בדקו שהסקריפט שלכם עובד היטב באמצעות קביעה שונים ל-`age` ובדיקה.

לולאת while

עד כה ראיינו דרכים לכתוב תנאי "חד פעמי", כלומר תנאי שפייתון מרים פעם אחת בלבד. לעיתים רצחה לבצע פעולה כלשהי כל עוד תנאי מסוימים מתקיים. לדוגמה, רצחה לקרוא קלט מהמשתמש ולבצע את הוראות המשמש, כל עוד המשתמש לא כתב 'Exit'.
במקרים אלו שימוש ב-if ו-else הוא לא מספיק טוב, כיון שהוא לא יכולם לדעת כמה פעמים התנאי שלנו צריך לזרז. אולי הממשש כתוב 'Exit' כבר בהוראה הראשונה? אולי בהוראה העשירה? במקרים אלו משתמש בלולאת `while`.

לולאת `while` מתחילה צפוי בהוראה `while`, לאחריה יבוא תנאי אותו נגדיר. לאחר מכן יש בлок של פקודות אשר יבוצעו בו אחר זו, ובסוף הблок תהיה חוזה אל בדיקת התנאי.

```
while condition:
    # do something
    # return to the while condition
```

シומו לב – בדיקת התנאי מתבצעת רק לפני הכניסה לבlok. 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 יודפס כלום, מכיוון שבתוך הלולאה, בשורה 3, הערך של `age` מועלה ל-18 ואז התנאי שבודק אם `age` קטן מ-18 כבר לא מתקיים ושורה 4 לא תבוצע. אך שורה 4 כן תבוצע, מכיוון שברגע שנכנסים לתוך הблוק מרצים את `Not yet 18` פעמיים. הבדיקה של ה-`while` מתבצעת פעם אחת, בכניסה לבlok, ולא כל פקודה מחדש. לכן המשפט `Not yet 18` יודפס פעמיים. באירועיזציה השנייה של הלולאה התנאי כבר לא יתקיים ולא תהיה כניסה לתוך הблוק.

מה לדעכם יקרה אם נריץ את הלולאה הבאה?

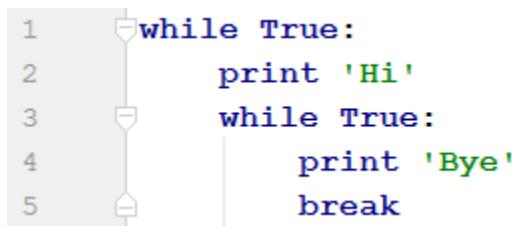
```
while True:
    print 'Hi'
```

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

```
1     while True:
2         print 'Hi'
3         break
4     print 'Bye'
```

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

מה ידפיס הסקריפט הבא?



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

תרגיל – Take a Break



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

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

lolאות for

lolאות for הן שימושות מיוחדים כאשר יש לנו אוסף של איברים שאנו רוצים לבצע עליהם פעולה כלשהי, בניגוד לlolאות while שהן שימושות כאשר מרצים מספר לא ידוע של פעמים. בשפת פיתון, lolאות for נכתבות בצורה מעט שונה משופות תכנות אחרות.

בפייתון, lolאת for מקבלת אוסף של איברים. לדוגמה, מספרים מ-1 עד 10, או ארבעה שמות של ילדים. בכל איטרציה – מעבר על הלולאה – אחד האיברים מאוסף האיברים נתען לערך המשתנה שעליו רצתה הלולאה. לאחר סיום האיטרציה, נתען האיבר הבא מאוסף האיברים וכך הלאה עד סיום כל האיברים באוסף.

lolאת for מתחילה במילה for, ולאחר מכן יבוא שם של המשתנה כלשהו שעליו רצתה הלולאה (נקרא "איטרטור" iterator), לאחר מכן המילה in ואחר מכן אוסף של איברים. בדוגמה הבאה אנחנו שמים כמה מספרים בתוך סוגרים מרובעים, מה שהוא אומר שם הופכים ל"רשימה", טיפוס של המשתנה שנלמד עליו בהמשך:

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

פלט ההרצאה יהיה:

```
0
2
4
```

כאמור, אפשר לייצר לולאות שרצות על כל מיני אוספים 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 נרצה לעשות לולאת `for` שעוברת על סדרה של מספרים? די לא נוח כתוב את כל המספרים אחד אחריו השני... הפקנץיה `range` מייצרת סדרות של מספרים. היא מקבלת בתור פרמטרים התחלת, סוף וקפייצה 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 כמה סדרות:

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

תוצאת ההרצאה:

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

הסבר: השורה הראשונה כאמור יוצרת רשימה של מספרים שמתחלים ב-3 ועוצרים 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 מזנת קפייצה, אז נעשה שימוש בקפייצה של 1. אם לא מזנת התחלת, מתחלים מ-0.

ישנה פונקציה דומה לפקודה `range`, אשר נראית `xrange`. פקודה זו מבצעת את אותו הדבר כמו `range`, אך לא נוצרת רשימת איברים בזיכרון המחשב. למה זה טוב? דמיינו שאתם רוצים למצוא את כל המספרים הראשוניים בין אפס למליארד. אתם כותבים קוד שבודק אם מספר הוא ראשוני, ומכניםים אותו לתוך לולאת `for` שרצה על (1000000000). מה שקרה הוא שתיתווצר רשימה של מליארד מספרים בזיכרון המחשב. לא דבר טוב בכלל, כיון שהוא צפוי להאט את העבודה המחשב. לעומת זאת, אם נעשה את אותה לולאת `for` עם (1000000000) `xrange` או נקבל את אותם המספרים אבל הם יתכלו בלי להישמר בזיכרון המחשב. אפשר לסכם ולומר שפקודת `xrange` היא שימושית כאשר ייצור סדרת מספרים שיישמשו אותנו מעבר לlolאת `for`. בשביל לlolאת `for` בלבד, מוטב להשתמש ב-`xrange`.

תרגיל 6 בום (เครดיט: עומר רוזנבוים, שי סדובסקי)



כיתבו לlolאת `for` שמדפיסת את כל המספרים מ-1 עד 40 (כולל).

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



הდפיסו למסך את כל המספרים בין 0 ל-100 שמתחולקים ב-7 ללא שארית, או שמכילים את הספרה 7, לפי הסדר. השתמשו רק בפעולות החשבון! עזרה: פעולת מודולו – החזרת השארית מחלוקת – נכתבת בפייתון בעזרת סימן %. לדוגמה:

14 % 4

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

pass

מה אם נרצה שלולאה שכתנו לא תבצע כלום? רגע אחד, למה נרצה לכתוב קוד שלא מבצע כלום? הדבר שימושי בתהליך פיתוח קוד. אנחנו כותבים את שלד התוכנית, שמכיל לולאות ופונקציות, אבל משאירים אותן ריקות. כך אנחנו מסיימים במיריות את השלד ויכולים לבדוק שהקוד שלנו הוא נכון, לפני שאנו מתחילהム להתעסק עם השימוש עצמו. זה די שימושי כאשר כותבים תוכניות מורכבות. לכן קיימת הפקודה `pass`, פקודה שאומרת – אל תעשה כלום.

לדוגמה:

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

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

תרגיל מסכם (เครดיט: עומר רוזנבוים, שי סדובסקי)



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

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

פרק 4 – מהרווזת



הגדרת מהרווזת

מהרווזת הינה אוסף של תווים. מגדירים מהרווזת באמצעות גרש יחיד או גרשאים. כך לדוגמה ניתן להגיד מחרוזת משתנה בשם

greeting אשר שווה למהרווזת Hello בשתי צורות, עם גרש יחיד או עם גרשאים:



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

והתוצאה היא זהה בשני המקרים.

למה טוב לזכור שאפשר להגיד מהרווזות בשתי הדרכים? כי לפעמים נרצה להגיד מהרווזת שיש בה את אחד הסימנים הללו. לדוגמה, אם נרצה להגיד את המהרווזת up 's what לא נוכל לתחום אותה בגרש יחיד. עם זאת, נוכל לכתוב:

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

נקודה נוספת שנוגעת למהרווזות היא שנותים לעיתים לבלב מהרווזות של ספרות עם המספר עצמו. לכן נבהיר זאת – המהרווזת '1234' אינה שווה למספר 1234. הבלבול נובע בכך שם נעשה print בשני המקרים ונקלט אותו הדבר – יודפס למסך 1234, ואולם הסיבה לכך היא שכאשר מבקשים להדפיס מספר, פיתרון מתרגם אותו מאחוריו הקליים למהרווזות וואז מדפיס אותה. ההבדל בין המהרווזת 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 תוצאת 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 פקודה `print` ולאחריה את מה שרצינו להדפיס. זו שיטה מאד נוחה לשימוש להדפס ערך ייחיד, אבל היא נהיית מעט מסורבלת לשימושים להדפסים אוסף של ערכים. דבר זה נכון במיוחד אם חילק המעריכים הם מסוג `str` וחלק מהם מטיפוסים אחרים, כגון `int`. נכון, אפשר להדפיס משהו שהוא כולל `str`-ו-`int` בצורה הבאה, על ידי המرة של ה-`int`-ל-`str`, אבל כאמור זה קצת מסורבל:

```

>>> greet = 'My age is: '
>>> age = 20
>>> greet + ' ' + str(age)
'My age is: 20'

```

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

ונכל להשתמש בפקודת `format`, ובמקום כל משתנה שאנחנו רוצים להדפיס, נשים 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

```

היתוך מחרוזות – string slicing

אפשר לחתוך חלקים מחרוזות באמצעות הפקודה:

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

החיתוך דומה לפונקציית `range` אותה הכרנו כבר.

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

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

הפרמטר `delta` מציין כמה אינדקסים קופצים, בירית המוחלט היא 1.

שים לב שגם ערכים שליליים מתקבלים באופן תקין! לדוגמה, אם נגידיר מחרוזת או נראה שכל איבר יש גם אינדקס שלילי –

כלומר 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, ככלمر 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
```

הסבר:

שורה 2 מדפסה את האיבר באינדקס מס' 1 במחרוות. שימוש לב שזהו 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 מאיןדקס 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 ברירה מחדול, 1. למעשה זהה למשה זויה המחרוזות 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 אחרונה. במלילים אחרות, מתחילה מהאיבר האחרון והולכים אחריה עד שמגיעים לאיבר הראשון.

תרגיל (เครดיט: עומר רוזנבוים, שי סדובסקי)



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

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

- 'lo ynm sl'

פקודות על מחרוזות

אפשרון מאפשרת לנו לבצע בקשות פעולהות שונות. כתה נראה דוגמה נחמדה לכך. גניה שיש לנו שתי מחרוזות ואנחנו רוצים ליצור מן מחרוזות חדשה, צירוף של שתיהן. הדרך לעשות זאת היא פשוט להשתמש בסימן '+'. יש למחרוזות מתודות שונות, נדגים כתה כמה מן.

```

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

```

נציג את פלט התוכנית לפני שנעביר 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 המחרוזות. בדוגמה זו, האורך של 'Hello' (シומו לב לרווחה בסוף המילה) ביחיד 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 עצם פונקציה שモגדרת בתחום חלקה, אבל נשמר זו את לפרק שדן בחלוקת. בכלל אופן, בשלב זה נשים לב בעיקר להבדל בצורת הכתיבה בין המתודה `upper` לפונקציה `len`. המתודה `upper` מגיעה אחרי סימן נקודה:

`message.upper()`

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

len(message)

הmethod `upper` מוחזירה את המחרוזת, כאשר כל התווים שלה הן אותיות גדולות. בשורה 5 אנחנו מדפיסים את `message` כדי להמחיש `sh-upper` לא שינתה את ערכו של המשתנה `message` – כמובן, האותיות נותרו קטנות.

בשורה 6 אנחנו מכירים method בשם `find`, אשר מקבלת כפרמטר תו ומוחזירה 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 מethodות מגייע כתה: היכן אפשר למצוא את כל המethodות של מחרוזות? בambilם אחרות, אם אנחנו רוצים לעשות פעולה עם מחרוזות, האם יש משהו שאנו יכולים לעשות חזן מאשר מה שם הפעולה?

ובכן, נזכיר את הפונקציה המובנית `dir`. פונקציה זו מוחזירה לנו את כל המethodות של משתנה. לדוגמה, אם נגידר מחרוזת בשם `message` וنعsha 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, כתה אנחנו יודעים מה שמוטות כל המethodות שניתנו להרץ על מחרוזת. אבל כיצד נוכל לדעת, לדוגמה, מה עושה method `'count'`? כאן מגיעה לעזרתנו הפונקציה המובנית `help`. נבצע `help count` על המודה `message` של `message` ונקבל את התיעוד של המודה:

```
>>> 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 הימיניות מופיעות בתוך מחרוזת. לדוגמה, כאשר נעשה count עם תחילה המחרוזת 'el' קיבל ערך 1, וזאת מכיוון ש-'el' מופיע פעם אחת בתחום 'Hello world'.

הדבר החשוב שלמדוינו אינו המethode count, אלא הידיעה שבכל פעם כשנרצה לבצע פעולה – על מחרוזות ועל טיפוסי משתנים אחרים – נדע היכן לחפש אותם ואיך לקבל עליהם מידע.

צירופי תווים מיוחדים ו-raw string

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

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

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

אבל מה אם יש לנו טקסט שכולל צירוף תווים מיוחד אנחנו רוצים שהוא יודפס כמו שהוא? לדוגמה, יש לנו במחשב תקיה בשם c:\cyber\number1\b52. שמו לב איך התיקיה זו תודפס באופן "רגיל":

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

```
c:\cyber
umber52
```

לא בדיק מה שרצינו שיודפס...

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

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

ההכפלת אומרת לפיתון 'אנחנו באמת מתכוונים לסימן \'.'

האפשרות השנייה היא לכתוב לפני תחילת המחרוזת את התו '\', שמנדר לפיתון שהכוונה היא ל-raw string. כמובן, עליו לקחת את התווים כמו שם ולא לנסות לחפש צירופי תווים בעלי משמעות.

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

קלט קלט מהמשתמש – raw_input

את הפונקציה `raw_input` הכרנו בקצתה בפרק הקודם. פונקציה זו משתמשת לקבלת קלט מהמשתמש. הפונקציה מדפיסה למסך מחרוזת לפי בחירתנו, ואת מה שהמשתמש מקליד היא מכניסה לתוך מחרוזת.

לדוגמה:

```
username = raw_input('Please enter your name\n')
print 'Hello {}'.format(username)
```

שים לב לכך שהמחרוזת המודפסת שבחרנו להדפיס למשתמש כוללת את התו '\n', וזאת כדי שקלט המשתמש יהיה בשורה חדשה, דבר זה נעשה מטумני נוחות. הtoutאה:

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

תרגיל – אבניבי

זוכרים את שפת הבינית? כתבו קוד שקולט משפט (באנגלית) מהמשתמש ומתרגם אותו לשפת הבינית. תוכורת: אחרי האותיות你需要 ציריך להוסיף ס ולחכפל את האות. לדוגמה, עברו הקלט `otach ohev ani` יודפס:

`Abanibi obohebev obotabach`

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

`if 'a' in my_str:`



תרגילי strings (מתוך google class strings, בעיבוד גבהים)



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

[לינק להורדת קובץ הפיתון של התרגיל:](http://data.cyber.org.il/python/ex_string.py)

לדוגמה, תרגיל `donuts`:



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

בתרגיל זה אנו מתבקשים להחזיר מחרוזת ש כוללת את מספר ה-`donuts` שקיבלו כפרמטר לפונקציה (בתוך המשתנה `count`), ואולם אם `count` הינו גדול מ-10,علינו להחזיר פשוט '`many`'.

פתרון:

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

תוצאת ההרצאה:

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

תרגיל מסכם – ז'אן ולז'אן (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

דוגמה לরיצה תקינה:

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

הדרכה: בתרגיל זה אנחנו נדרשים לבצע המרה בין סוגי טיפוסים שונים. זיכרו, שהפונקציה `raw_input` מחזירה מחרוזות של תווים. כלומר אם המשמש הוזן 1234, תוחזר המחרוזת '1234'. אם אנחנו מעוניינים להשתמש במחרוזות כמספר, علينا לבצע קודם לכן המרה מטיפוס מחרוזות לטיפוס `int`. לשם כך נשתמש בפונקציה `int`. לדוגמה:

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

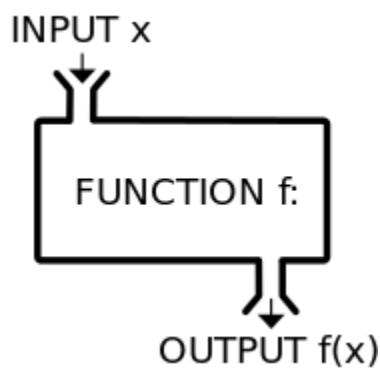
הערה: ניתן להניח שהשימוש הוזן קלט תקין.

פרק 5 – פונקציות

כתיבת פונקציה בפייתון

פונקציה היא קטע קוד שיש לו שם ואפשר להפעיל אותו על ידי קריאה לשם. לקטע הקוד אפשר להעביר מידע, שעליו יבוצעו פעולות, ומובן 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 לחת לפונקציה שם בעל משמעות, כך שגם מי שקורא את הקוד שלנו יוכל להבין מה רצינו.
- בתחילת הפונקציה יש לכתוב תיוז, שמתאר מה הפונקציה עושה. לתיוז בתחילת פונקציות יש שם מיוחד – `docstring`, קיצור של `documentation string`, מהרוות תיוז. כתבים `docstring` בתוך שלושה סימני גרשימים `'''` כפולים רצופים, כמו בדוגמה. מדוע זה חשוב? מכיוון שם נעשה `help` על הפונקציה, נקבל בחזרה את ה-`docstring` שלו. דבר מועיל מאוד כאשר אנחנו רוצים לשתף קוד!

```
help(hello)
```

Help on function hello in module __main__:

```
hello()
Print Hello
```

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

```
my_func()
```

דוגמה לפונקציה עם פרמטר אחד:

```
my_func(x)
```

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

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

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

שימוש לב לאופן התיעוד, שכולל גם הסבר אודוט הfrmטטים שמקבלת כל פונקציה. מדובר בתיעוד הם נקראים **Args**? זה קיצור של "ארגומנטים" ארגומנט הוא המשתנה כפי שהוא נקרא על ידי מי שקורא לפונקציה. פרטט הוא המשתנה כפי שהוא נקרא בתוך הפונקציה.

כעת נראה איך קוראים 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`, שזוהה לשם הפרטט כפי שמוגדר בתוך הפונקציה. מאידך, הפונקציה `print_messages` נקראת עם ארגומנטים בשם `mes` ו-`sage`, שאינם לשם הפרטטים שמוגדרים בתוך הפונקציה – `msg1` ו-`msg2`. שתי האפשרויות חוקיות. בתוך הפונקציה `print_messages` מתבצעת העתקה של הארגומנט `mes` לתוך הפרטט `msg1` ושל הארגומנט `sage` לתוך הפרטט `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 = seven()` 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



כיתבו פונקציה אשר מקבלת שני ערכים ומחזירה את המכפלת שלהם.

כיתבו פונקציה אשר מקבלת שני ערכים ומחזירה את המנה שלהם. זיכרו שהילוק באפס אינו חוקי, במקרה זה הזרו הודעה "Illegal".

None

הבה נבחן את הפונקציה הבאה:

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

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

```
k = stam()
```

נסו לעשות `print k` ותקבלו שהערך של k הוא `None`. הערך `None`, או "כלום", הוא ערך ריק. זהו מילה שמורה בפייתון. משתחנה יכול להיות שווה לערך ריק, והוא הוא שווה `None`.

יש 3 מצבים בהם פונקציה מוחזרת ערך `None`:

- לפונקציה אין `return` כלל.

- לפונקציה יש `return` אבל בלי שום ערך או משתנה. הכוונה לשורה שבה כתוב רק `return`.

- לפונקציה יש `return None`

של משתנים scope

מה תבצע התוכנית הבאה?

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

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

כפי שואלי 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 הרצתה? הרי הפונקציה `main` קראת לפונקציה `speak`, אשר בה הוגדר המשתנה `word` וAAF הודפס למסך. מדוע `main` לא מכירה את המשתנה `word`?

הסיבה היא שהמשנה `word` הוגדר בפונקציה `speak` והוא קיים רק בה. ברגע שיצאנו מהפונקציה `speak`, המשתנה `word` פשוט נמחק ואיןו קיים יותר. מכאן שה-`scope` של המשתנה `word` הוא אך ורק בתחום הפונקציה `speak`.

נحدد את ההסבר ותווך כדי נבין את ההבדל בין המשתנה גלובלי למשתנה מקומי, מקומי. נניח שכותבנו את הקוד הבא, שימו לב למשתנה החדש `:name`:

```
name = 'Shooki'
```

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

def main():
    speak()
```

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

נתקדם עוד一步 – כתעת אנחנו מגדרים את המשתנה `word` פעמיים. אחת כגלובלי ואחת כлокלי... מה ידפיס הקוד הבא?

```

word = 'bye'

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

def main():
    speak()

```

ובכן, יודפס hi. מודיע՞ ? הרו אמרנו שמשתנה גלובלי מוכגם בתוך פונקציה? נכון, אלא שבפונקציה speak אנחנו "דורסים" את המשתנה הגלובלי word עם משתנה מקומי בעל אותו שם. כתעת כאשר נפנה בתוך speak למשתנה word, הוא כבר לא יכיר את המשתנה הגלובלי, אלא רק את מה שהוגדר מקומיית.

ניסוון נוסף... מה ידפיס הקוד כתה?

```

word = 'bye'

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

def main():
    speak()
    print word

```

יודפס:

hi
bye

מדובר? את ההדפסה של `hi` כבר הבנו. כאשר `speak` מסיימת את הרציצה שלה, המשתנה המקומי `word` נמחק, וכעת קורה משאו מעניין – מסתבר שהמשתנה הגלובלי `word` לא נמחק, אלא פשוט נשמר בצד. לפיתון יש מידרג של עדיפויות: כאשר פונים למשתנה בתוך פונקציה, קודם כל פיתון מחפש אם קיים משתנה מקומי כזה, ולאחר מכן אם קיים משתנה גלובלי. ההדפסה השנייה מתרחשת מתוך הפונקציה `main`, שמכירה רק את המשתנה הגלובלי `word`.

ניסו איזור... מה ידפיס הקוד הבא?

```
word = 'love'

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

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

שגיאה! כדי שהתשובה 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 מדובר? הרי הגדרנו המשתנה גלובלי בשם זה? הסיבה היא, שכאשר אנחנו מבצעים פעולה שמשנה את ערכו של המשתנה בתוך פונקציה, כמו פעולה חיבור, פיתון מניח של משתנה שלנו יש עותק מקומי והוא מנסה לפעול עליון.

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

```

word = 'love'

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

def main():
    speak()
    print word

```

בuckות הריצה יודפס:

```

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

```

בעקבות הrincha יודפס:

```
love you
love
```

כעת, מדוע האפשרות הראשונה אינה מומלצת? כי שמהם לב, האפשרות הראשונה משנה את ערכו של המשתנה word גם מחוץ לפונקציה. הודפס פעמיים you. כמובן, הפונקציה שינה את ערכו של המשתנה. דמיינו שאתם כתובים את הפונקציה `main` ומتنנת אחר כותב את הפונקציה `speak`. כעת תארו לעצמכם את ההפטעה, שהפונקציה שינה ערך של משתנה בלי שהיא לכמ מושג שהוא עשתה זאת! אם הייתם רוצים לאפשר לפונקציה לשנות את הערך של המשתנה, היהתם מעבירים לה אותו כפרמטר ודווגים להציב את ערך החורה של הפונקציה במשתנה, כך:

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

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

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

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

תרגילים



- כתבו פונקציה בשם `factorial` שמחזירה את התוצאה של $5!$ (5 ערך). אין צורך להשתמש בברקורסיה.
- כתבו פונקציה בשם `beep` שמקבלת מחרוזת ומחזירה את המחרוזת ועוד beep בסופה.
- כתבו פונקציה בשם `mul_2nums` שמקבלת שני מספרים ומחזירה את המכפלה שלהם, או 0 אם התוצאה שלילית.
- כתבו פונקציה בשם `sum` יותר מפעם אחת בפונקציה. שימו לב, אפשר לכתוב `return` יותר מפעם אחת בפונקציה.

פיתון מתחת למכסה המנוע (הרחבה)



בחלק זה נפרט שני נושאים הקשורים לפרק הלימוד עד כה:

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

זכור פיתון היא שפת סкриיפטים, ולכן לא מתבצעת קומPILEציה לקוד. במקרים אחרים, כל שורה קוד שאנו כתבים מומרת לשפת מכונה רק כאשר מגיע תורה של שורה הקוד להיות מורצת. נמchia את העיקרון הזה בעזרת תוכנית קטנה.

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

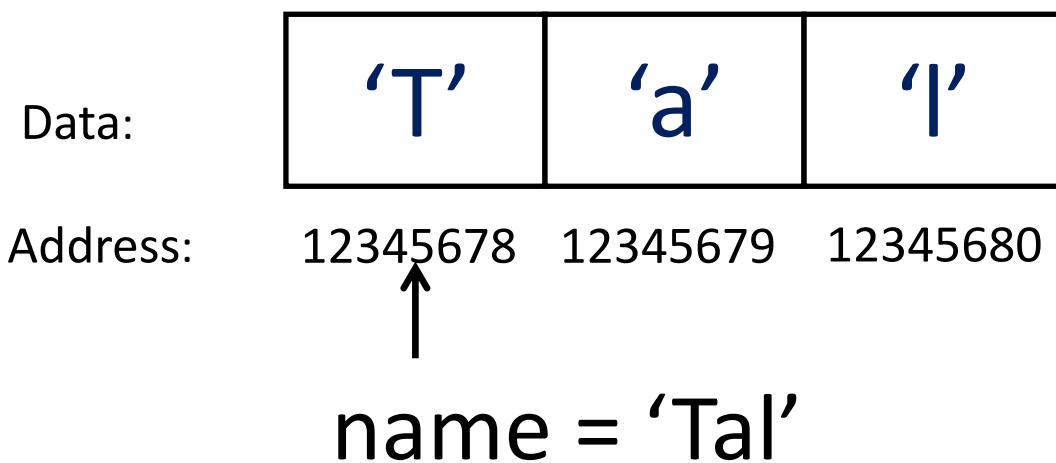
```
def main () :
    check (1)
```

כפי שאפשר לראות, הקוד מכיל קריאה לפונקציה בשם bla עם ארגומנט bla. הן הפונקציה והן הארגומנט אינם מוגדרים. לעומת זאת, אם נريץ את התוכנית נקבל תמיד 'OK'. הסיבה לכך היא שתנאי ה-if מתקיים תמיד, ולכן הקוד שבתוך תנאי ה-else לעולם אינו מגיע להיות מושך. זהה המחשבה של העובדה שבפייתון כל שורת קוד מפורשת ומתורגת לשפת מוכנה רק כאשר מגיע זמנה להיות מושכת.

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

id, is

זכרנו המחשב מכיל את כל המשתנים שמוגדרים בתוכנית שלנו. כל משתנה נמצא בכתובת מוגדרת, כמו למשל כאשר אנחנו מגדירים משתנה הוא מקבל כתובת מתוך טווח הכתובות שモקצת לריצת התוכנית שלנו. עבור כל משתנה שמוגדר בתוכנית פיתון, שם המשתנה משמש כדי להצביע על כתובת בזיכרון. לדוגמה, אנחנו יכולים להגדיר משתנה בשם name והוא יצביע על כתובת 12345678. בכתובת 12345678 יאותן הבית הראשון של המשתנה, אם המשתנה שלנו הוא בגודל של יותר מבית אחד אז יתר הבתים שלו מאוחסנים בכתובות העוקבות, לדוגמה 12345679, 12345680 ועוד הלאה.



באמצעות הפונקציה id אנחנו יכולים לחשב את הכתובת בויכרנו 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, מסמגת שהוא מספר מטיפוס Long, כלומר כוח המוצג על ידי 64 ביט.

הישבו: 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 כתובות בזיכרון. נראה דוגמה:

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

הגדכנו משתנה בשם name_copy וקבענו שהוא שווה למשתנה name. בכך, גרמנו לו למעשה להצביע לאותה הכתובת בזיכרון מציין המשתנה name. לאחר מכן, בדקנו שהערך של name_copy הוא גם '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. ראינו שאפשר לבדוק בעזרת is אם משתנה הוא True או לא. לדוגמה:

if result is True:

...

if results is not True:

...

כעת אנחנו יכולים להבין טוב יותר מהו `is`. האופרטור `is` בודק האם `id` של שני משתנים הוא זהה. אם כן – נקבל `True`, אחרת – `False`. שימושו לב שוגם ל-`True`, גם ל-`False` וגם ל-`None`.

```
>>> 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 האסטטוטי של גביהם. בקצרה, ישנן שתי שיטות להעברת פרמטרים לפונקציה.

Pass by value -

Pass by reference -

בשיטת `pass by value` מועבר לפונקציה העתק של הפרמטר. העתק נמצא על אזור בזיכרון שנקרא מחסנית, או `stack` ומשמש פונקציות. דמיינו שהמורה מזוויק דף נייר שכותב עליו המספר 10. המורה ניגש למכתנת הצלום ומצביע 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

במוקום 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 שהיה לפני כן.

בשיטת pass by reference מועברת לפונקציה – גם כן דרך המחסנית – הכתובת בזיכרון שבו נמצא המשתנה. ערך המשתנה לא מועבר לפונקציה, ואם ברצונה לקרוא את הערך עליו לגשת לזריזון בכתובת שנמסרה. דמיינו שכעת המורה שלנו מניח את הדף עם המספר 10 בתא שלו בחדר המורים, ובמקום לגשת למוכנת הצלום הוא ניגש למנעולן ומשכפל לכל אחד מתלמידיו 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 רוצחים – גם לשנות אותו. שינוי שיבצעו התלמידים ישפיע על הדף המקורי שבידי המורה. שימו לב שהפעם לא נוצרו לדף עותקים, יש רק עותק מקור.

או מה קורה בשפת פיתון? האם פרמטרים מועברים בשיטת pass by value או בשיטת pass by reference? הנה נזכיר פונקציה ונעביר 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 מהרזהות בשם x. בשורה 9 אנחנו מדפיסים את (x)id רק כדי שנוכל להשוות אותו לפני ואחרי הכניסה לפונקציה. בשורה 10 אנחנו קוראים לפונקציה ומעבירים לה את x כפרמטר. בתוך הפונקציה אנחנו בודקים את (x)id, לאחר מכן משנים את ערכו של x ואז בודקים שוב את (x)id. לאחר היציאה מהפונקציה, בשורה 11, שוב בודקים את (x)id כדי לראות אם הה-id היה בתוך הפונקציה נשמר.

והנה מה שקיבלנו:

```
37058640
```

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

בשורת הדרישה השנייה אנחנו רואים שלפונקציה הועבר ה-`id` של `x`, שהרי ה-`id` מחזק לפונקציה ובתוך הפונקציה הם זהים. האם זה אומר שהפרמטר הועבר `by reference`?

נראה כך, אבל אז מאיימת שורת הדרישה השלישי, ואני רואים שהעובדת שהפונקציה שינתה את `x` שינתה גם את `(x).id`. בambilם אחרות, נוצר בתחום הפונקציה משתנה חדש בשם `x`, שיש לו `id` שונה מאשר ה-`id` שנמסר לפונקציה. ואכן, בשורת הדרישה הרביעית, שמתרכשת מחזק לפונקציה, אנחנו רואים שערכו של `(x).id` חוזר להיות כפי שהיא לפני הקראיה לפונקציה.

לסיכום: פiython מעבירה לפונקציות את הכתובת של הפרמטרים, אך ברגע שפונקציה מנסה לשנות אותם נוצר העתק, כך שהערך של המשתנה המקורי לא ישנה.

רגע לפני שנסכם, נציין שככל מה שתכתבו כעת נכון למשתנים מסוימים, שנקרא `immutable`, שעד עכשו עסוקו בהם בלי לקרוא להם כך. מהם המשתנים מסווג `immutable`? ומהם המשתנים מסווג `mutable`? על כך נרחיב כשנלמד על משתנים מסווג `list`.

סיכום

בפרק זה למדנו אודות פונקציות בפייתון. למדנו להגדיר פונקציה, להעביר לה פרמטרים ו גם לקבל מהם ערכים. לאחר מכן ראיינו של משתנים יש `scope` שבו הם מוגדרים, כלומר משתנה מוכר רק בתחום שהוא מוגדר. סקרונו אפשרותויות שונות של שימוש במשתנים גלובליים בתחום פונקציה והגענו למסקנה שתמיד כדאי להעביר משתנים כפרמטרים לפונקציה, ובכל מקרה עדיף להמנע משינוי של משתנים בתחום פונקציה בלי שהקוד שקורא לפונקציה מודע לכך.

לאחר מכן, במסגרת הרחבה, ראיינו איך פiython עובד "מתחת למכסה המנע", כיצד מועברים פרמטרים לפונקציה. כעת אנחנו יכולים לכתוב פונקציות ולהשתמש בהם בצורה חופשית.

פרק 6 – List, Tuple

הגדרת List

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

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

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

בין כל שני איברים ישנו פסיק מפריד. שימו לב שבתוך ה-list יכול להיות אוסף של איברים מסוגים שונים. בדוגמה לעיל יש לנו:

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

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

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

:יודפס

```
11
aaaa
36.5
True
```

כמובן שצורך להיות שיטה ייעלה יותר להדפיס את כל איברי הרשימה, נכון? לו לאת `for` צועדת יד ביד עם `list`. כל מה שצדיק לעשوت הוא להוסיף את המילה `חן`, ולבחר שם של משתנה שהוא "איטרטור", ככלומר יקבל כל פעם ערך של איבר אחר ברשימה. כך:

```
for element in stam:
    print element
```

הלוואת תרוץ 4 פעמים, כמספר האיברים ב-`stam`. בכל ריצה – איטרציה – של הלוואה, האיטרטור `element` יקבל ערך מתוך `stam`, לפי הסדר שבו האיברים נמצאים בתחום `stam`. בסופו של דבר תוצאה ההדפסה תהיה זהה לתוצאה ההדפסה איבר איבר.

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

```
stam[2:]
```

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

תרגיל



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

תרגיל (เครดיט: עומר רוזנבוים, שי סובסקי)



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

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

תחזיר את הסכום – 37.5

עליכם למשה פונקציה שהיא כמו `sum` אבל לא בדיק. נקרא לה `summer`. המירוד ב-`summer`, הוא שהוא יכולה לעבוד עם כל סוגי של משתנה, לא רק `int` או `float`, אלא גם משתנים מטיפוס `string` ואף `list`. כן, אפשר לחבר רשימות! נסו להגיד שתי רשימות ולחבר אותן באמצעות סימן `+`.

לדוגמה, אם נכתוב:

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

נקבל:

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

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

Mutable, immutable

כעת נבין מה פירוש שני המושגים הללו, שהוזכרו בפרק הקודם. נחשב על ההבדלים בין מהירותם לרשימה. ראיינו ש כדי לגשת לאיבר ברשימה או במחוזות, כל מה צריך לעשות זה להכנס 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 לאייר פועלת באופן זהה ברשימה ובמחרוזת. אבל מה עם גישה לכתיבת של אייר? מה יקרה אם ננסה לשנות את האיר 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 מחרוזות אי אפשר לשנות – קיبلנו שגיאה. עכשו אפשר להבין את המושגים שבכותרת. משחו שאפשר לשנות אותו נקרא mutable, מהמילה "מוטציה". משחו שאי אפשר לשנותו הוא immutable.

עם זאת, ראיינו שאפשר לשנות מחרוזות. ככל מר, אפשר להגדיר מחרוזות ואו להגדיר אותה שוב עם ערך אחר. איך זה אפשרי?

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

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

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

בוצע שינוי ברשימה, ונីוכה שה-id נותר זהה:

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

פעולות על רשימות

in

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

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

כאשר כתבים ביטוי מהצורה 'איבר ח'ו רשימה', התוצאה תהיה או `True` או `False`, מה שהופך את השימוש לנוח במיוחד עבור משפט `if` כמו בדוגמה. בהזדמנות זאת נזכיר שפיטון מתייחס לאותיות גדולות וקטנות, כלומר אם נחפש `apples` במיילון נקבל `False`.

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

append

הmethodה `append` תמיד מוסיפה איבר בסוף הרשימה. כמובן, אי אפשר להוסיף באמצעות איבר לאמצע הרשימה. המmethodה מקבלת את האיבר שרוצים להוסיף. לדוגמה:

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

pop

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

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

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

sort

מה לגבי מין רשיימה? המתודה `sort` מטפלת בזוה. כל רשיימה ש-`sort` תפעל עליה תהפוך להיות ממוינית. אם נשימוש ב-`sort` בלי להזין לתוכה פרמטרים, מספרים ימוינו לפי הגודל ומחוזות ימוינו לפי סדר הופעתם במלון:

```
>>> 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 לנגיד, מהגדול לפחות, לפי סדר האלף בית... ובכן, הבה נחקור את `sort`. לפנינו כן, נזכיר بما שלמדנו כאשר ביצענו חיתוך מחוזות. אם יש לנו מחוזות, אנחנו יכולים ליצור מהם מחוזות שונה באמצעות סוגרים מרובעים, אשר בתוכם נמצא אינדקס התחלת, אינדקס סיום ועל כמה אינדקסים מודגמים. לדוגמה, התחלת באינדקס 2, סיום לפנוי 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 נקודותים. לדוגמה:

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

במקרה שאנו לא מזינים דילוג, מתבצע דילוג של 1. כמובן, יש ערך ברירת מחדל – אם אנחנו לא מזינים ערך אחר, כאילו הינו .1

cutת נחזיר למתודה `sort`. נעשה עליה :`help`

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

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

נראה שהיא מקבלת שלושה פרמטרים. הפרמטר האחרון נקרא `reverse` וערך בירית המודול שלו הוא `False`, כלומר אין "היפוך". נשנה 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']
```

קיבלנו את המילון בסדר הפוך. מספרים מהגדול לקטן ומהחרוזות הפוך מסדר הופעתן במילון.

נحمد, אבל מה אם נרצה לעשות מילון יותר יצרתי? לא רק מהקטן לגודל או מהגדול קטן, אלא כל מילון שנרצה? המתוודה `sort` מקבלת בתור פרמטר את המפתח למילון, פרמטר אשר נקרא `key`. בתור מפתח אנחנו יכולים לקובע איזו פונקציה שאחנהנו רוצhim. נראה דוגמה. יש לנו רשימה שכוללת כמה מהחרוזות. אנחנו רוצhim לבצע מילון לפי אורך המחרוזות. כאמור, המחרוזות 'zz', 'zz', שהאורך שלה הוא 2, צריכה להופיע לפני המחרוזות 'aaa' שהאורך שלה הוא 3 ואחרי המחרוזות 'c' שהאורך שלה הוא 1 בלבד. כדי לנו, הפונקציה `len` מוחירה אורך של מהחרוזות. 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 משלהנו ולהעביר אותה בתור מפתח. שימו לב, שהפונקציה צריכה להחזיר ערך כלשהו, שלפיו יבוצע המילון, כמו שהפונקציה `len` מוחירה מספר שהוא אורך המחרוזות.

בצעו מילון של מהחרוזות לפי התו האחרון בחרוזות. לדוגמה, `love` צריכה להיות לפני `cat` משום שהאות `e` מופיעה לפני התו `t`.

ראשית, נctrיך להגיד פונקציה שמקבלת מהחרוזות ומוחירה את התו האחרון שלה:

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

לאחר מכן נגדיר רשימה ונקרא ל-`sort` עם `:key=last`

```
def main():
    benjamins = ['p.daddy', 'lil.kim', 'dr.dre', 'n.big']
    benjamins.sort(key=last)
    print benjamins
```

ותוצאה ההדפסה שנתקבל:

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

split

לעתים נקבל מחרוזות ונרצה להפריד אותה למחרוזות קטנות יותר ולשמור אותן ברשימה. הדוגמה הקלסית היא מחרוזת שכוללת משפט, ו אנחנו רוצים להפריד אותה למלילים בודדות. לשם כך קיימת המתוודה `split`. כאמור, `split` פועלת על משרות מחרוזת, אך כיוון שהיא מחזירה רשימה נוכל עצת להבין יותר טוב את אופן הפעולה שלה.

המתוודה `split` מקבלת כפרמטר תו או מחרוזת שלפיהם תבוצע 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 רשימה בה כל איבר יהיה שם של סרט. למשלנו הרשימה מכילה את התו '#' בתור תו מפריד בין שמות הסרטים. לכן נקרא ל-`split` עם פרמטר '#':

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

וקיבלנו רשימה:

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

למתוודה `split` יש גם ערך ברירת מחדל, שהוא סימן רווח. במקרים אחרים, אם לא נעביר ל-`split` שום פרמטר, כל הרווחים במחרוזת יוסרו והמחרוזות שבין הרווחים יוכנסו לרשימה. לדוגמה:

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

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

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

join

זהי הפעולה הפוכה ל-split. יש לנו רשימה של מחרוזות ואנחנו רוצים לחבר אותן יחד למחרוזת אחת. כורת הכתיבה כאן היא גם הפוכה ל-split – ראשית יבוא התו המפריד בין המחרוזות השונות (הגוני שזה יהיה סימן רווח), לאחר מכן נקודה ו-join עם שם הרשימה בסוגרים. כך:

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

כעת אם נדפיס את rule נקבל את המחרוזת המקורי:

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

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



הורידו את סקריפט הפייתון שב קישור data.cyber.org.il/python/ex_list.py והשלימו את הקוד החסר בפונקציות.

Tuple

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, אי אפשר לשנות את הערכים שבו. לשם מה זה שימושי? ובכן, הוא דרך נוחה להחזיר ערכים רבים מפונקציה. לדוגמה הפונקציה הבאה מחזירה 2 ערכים:

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

אם נציב את ערכיו הזרה שלה בtuple משתנה כלשהו, יוכל לראות שה משתנה זה הוא מסוג tuple. כאשר נעשה התוצאה תודפס בתוך סוגרים עגולים:

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

תוצאות ההדפסה:

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

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

שימוש לב לדרך אלגנטית לקלוט מספר ערכים מפונקציה שהחזיר מספר ערכים:

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

כעת כל אחד מה משתנים first, second הוא מחזיר שמייל את אחד הערכים שהוחזר מפונקציה.

נשאל – מדוע בכלל יש צורך ב-tuple? הרי הינו יכולים לבצע את אותן הפעולות באמצעות רשימה. לדוגמה, יכולנו ליצור פונקציה שהחזיר רשימה של ערכים ולא tuple. כאשר הינו קוראים לה ערכי הזרה היו נתונים למשתנים.

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

```
a, b = stam()
```

אם כך, מדוע ?tuple

הסיבה המעניינת קשורה לכך שבפה פיתון מקצת זיכרונו לרישומות. פיתון מניה שם הגדנו רשיימה, יתכן שנרצה להוסיף אליה ערכים באמצעות `append` או לשנות את הערכים השמורים בה. לכן פיתון צריך לדאוג שהיו לרישימה מלביעים שמאפשרים להוסיף איברים ולערוך איברים קיימים – יש לכך עלות מסוימת בזיכרון. לעומת זאת, tuple הוא טיפוס שלא ניתן להוסיף לו ערכים וגם לא לשנות ערכים קיימים, שכן פיתון מקצת בדוק את כמות המקומות שהגדנו. כמובן, אם אנחנו יודעים שאנו לא מתכוונים להוסיף איברים, הטיפוס tuple הוא חסוני יותר במקומו בזיכרון. כאשר אנחנו מוחזירים ערכים מפונקציה, אנחנו יודעים כמה ערכים החזינו, שכן השימוש בtuple הוא מתקש.

סיכום

בפרק זה למדנו אודות שני טיפוסי משתנים שימושיים – `list` ו-`tuple`. סקרנו מתודות ופונקציות שימושיות של `list`: בדיקה אם איבר נמצא ברשימה, הוספת והוצאת איבר, מינון רגיל, מינון לפי מפתח מיוחד. ראיינו כיצד בעזרת `split` ו-`join` אפשר להמיר מהירותם לרישימה ולהיפך, דבר שימושי כשרצה לנתח טקסט ולעבד בו מילים בודדות.

במהלך הפרק התודענו למושגים `mutable` ו-`immutable`. ראיינו שרישימה היא משתנה `mutable` – ניתן לשינוי – בעוד מהירותם היא `immutable`. לסימן ראיינו כיצד tuple משמש להזרת ערכים מרובים מפונקציה.

פרק 7 – כתיבת קוד נכונה

עד כה למדנו לכתוב קוד פיריטון בסיסי, אך עשינו זאת בעלי הקפדה רבה על איות הקוד שאנחנו כתובים. מדוע חשוב לכתוב קוד פיריטון "נכון"? הרי אנחנו יודעים למה אנחנו התוכנו כאשר כתבנו את התוכנית...? הבעה היא שבעולם ה"אמתית" סביר מאוד שהקוד שכתבנו יהיה רק חלק ממערכת גדולה 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 שהוא יהיה להם קל להשתמש בו. בנוסף, שמירה על כללי כתיבת קוד איקוטי תסייע לנו למנוע כתיבה של שגיאות, ובביחוד שגיאות שהיה לנו קשה במיוחד לתתקן. כמו כן, כתיבת קוד איקוטית תסייע לנו במידה כתיבת הקוד ותמנע שכפול של פעולות קיימות, ובכך תחסוך לנו זמן יקר.

מהו בכלל קוד איקוטי? בפרק זה נתמקד בכך ככלים:

א. הקוד צריך לעבוד. כאמור, אם קוד נכתב במטרה לבצע פעולה מסוימת, עליו לבצע אותה בצורה נכונה. לדוגמה, אם נכתב משחק קטן, המשחק צריך לזרוץ באופן חלק בלי להיתקע.

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

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

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

ה. הקוד צריך להיות מתייעוד. כאמור, יש לכתוב `docstring` בתחילת כל פונקציה וייש להוסיף תיעוד שמסביר מה עשינו, אך יש להמנע מתייעוד יתר, שرك מצין את המובן מאליו ואינו מוסיף מידע חשוב לקורא. דוגמה לתייעוד יתר כזה:

```
print my_str      # print the contents of my_str
```

באופן כללי, תיעוד נדרש להסביר למה הקוד נראה כפי שהוא נראה, ולא איך או מה הקוד מבצע.

ו. הקוד צריך לתואם לקובננציות, או בעברית "מוסכמות". CAN הסיבה היא פשוט נוחות של מי שמנסה לקרוא את הקוד שלכם. לדוגמה, שמות של קבאים ייכתבו באותיות גדולות. אך, אם נקרא קוד של מישחו ונמצא שם אותיות גדולות, קיבלנו מידע חשוב בלי ממש.

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

- כתיבת קוד פython לפי קובננציה PEP8

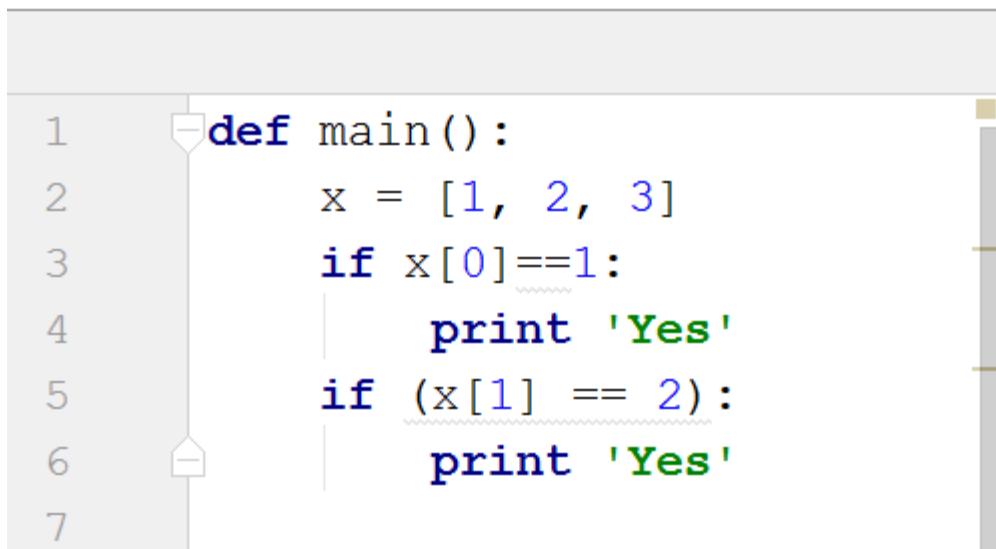
- חלוקה נכונה של קוד לפונקציות

- בדיקת תקינות קוד ומרקם קצה באמצעות assert

PEP8

אוסף כללי כתיבה נפוץ של קוד פython נקרא PEP8. מן הסתם לא נverbור כאן על כל הכללים – מדריך קצר מעת עומר רוזנביום ושלי סדובסקי נותנים מושג לגבי עיקרי הכללים: www.cyber.org.il/networks/PEP8.pdf

נתעכבר על השימוש ב- PyCharm על מנת לאתר טעויות PEP8 (כלומר, קוד שלא עומד בקובננציות שהוגדרו לפי PEP8 ולתכן אותו). 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 למעלה הינו בצבע צהוב, לא יירוק אבל גם לא אדום. הדבר מראה לנו שהקוד יירוץ בצורה תקינה, אבל יש בו טעויות PEP8. קל למצוא את הטעויות – האם אתם מבחינים בשני הקווים הצהובים שמתוחת לריבוע? נעמוד על אחד מהם:

```

1 def main():
2     x = [1, 2, 3]
3     if x[1]: PEP 8: missing whitespace around operator
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 שהסרים רוחחים לפני ואחרי סימן '='.

יתכן ששפתם לב גם לסייע הנוראה הצהובה שזולקת לצד שורת הקוד הבעייתית. לחיצה שמאלית עלייה תפתח לנו תפריט, ואם נבחר באפשרות Reformat file הבעיה מתוקן בלבד!

```

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

```

אפשרות נוספת לגלוות בעיות PEP8 בקוד היא באמצעות סימן אפור דק מתחת לטוים 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

```

לסיכום נושא ה-PEP8, נתמקד במספר דברים שחשוב שתשים לב אליהם.



- א. תיעוד: כל פונקציה שאתם כתבים צריכה להכיל `docstring`, כפי שראינו בתחילת הפרק אודות פונקציות. מומלץ לזכור על הדוגמאות שמספרות כיצד לכתוב `docstring`.
- ב. שימוש בקבועים: אחת הטיעויות הנפוצות של מתכוונים מתחילה 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 מורכב 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 חוזר על עצמו מספר פעמים בקוד – מכיוון שאנו רוצים להדפיס כמה דברים שונים, וכל אחד מהם – עשר פעמים. במקרה כזה נצטרך לשנות את המספר 10 שוב ושוב. עם זאת, יכול להיות שבחלק מהמקרים המספר 10 יהיה ל蹶ה אחר – למשל במקרים בהם רוצים לקרוא מידע מהמשתמש, ויכול להיות שדווקא שם נרצה לשמר על המספר 10 כשלעצמו. ככל שהתוכניות שלנו יהיו מורכבות יותר, כך הזמן שנצרך להקדיש להזداد.

האפשרות הטובה יותר היא שימוש קבועים. לדוגמה:

```
TIMES_TO_PRINT = 10

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

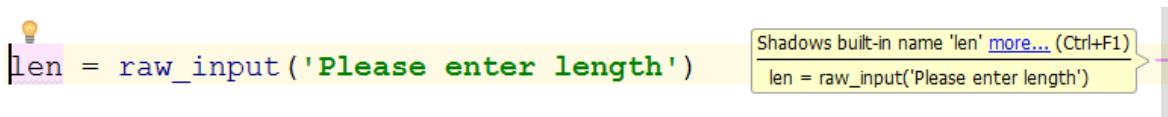
הגדרנו בתחלת התוכנית קבוע (שימו לב לאותיות הגדולות!), שرك מקראת השם שלו אנחנו כבר יודעים מה הוא עושה.Cut אם נרצה להדפיס מספר שונה של פעמים, כל שנצטרך לעשות הוא לשנות את הקבוע, סימנו.

שימו לב שפיתון לא מודאג עבורנו שהקבוע שלנו לא ישנה. ככלומר, עדין ניתן לבצע שינוי לערך הקבוע שלנו:

```
TIMES_TO_PRINT = 5
```

במקרה זה, הקונבנצייה אמורה לשמור עליו מפני עצמו – המתקנים. אנו מצינים את שם הקבוע באמצעות גדלות כדי לזכור שהוא קבוע – ולא לשנות אותו בקוד שלנו.

ג. לא לדרוםשמות מובנים בפייתון: לפיתון יש שמות מובנים – built-in names – שהפרשן שלו מכיר גם בלי שהגדרנו אותם. נניח שאנחנו מעוניינים לקלוט מהמשתמש מספר שמייצג אורך של משה. לדוגמה, אורך של ספר בעמודים, או אורך של משחק כדורים בדקות. מאוד מפתה להגיד את המשתנה בשם len – קיצור של length. הבעה היא שהשם 'len' הוא שם מובנה בפייתון, ככלומר יש לפיתון כבר פונקציה שנקראת len והיא יודעת להחזיר אורך של פרמטרים שהיא מקבלת. מה יקרה אם נדרום אותה? כמובן, פרט לכך שיתור לא יוכל להשתמש בה... לא ל恒mid, אבל כל עוד הסקריפט שלנו רץ.



חלוקת קוד לפונקציות

כאשר נכתב קוד,orcheshuv 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 שעבודת תקלות, ברור לקרוא וניתן בקלות להתחמה למשימות אחרות. מדוע זו מטרתנו? משום שמעט מקרים נדירים, בהן המערכת שלנו רצתה במאגרות מסוימות, הרבה יותר סביר שיש לנו משאבי מחשב רבים אך אנחנו מוגבלים בכמות הזמן שיש לנו לטובות כתיבת קוד, או שהמחיר שלאג במערכת הוא גבוהה מאוד, או שאנו נדרשים לכתוב קוד בצורה שצוות או קהילת מתכנתים יוכל להבין ולהשתמש בו. מכל הסיבות האלה, קוד איקוטי הוא קוד שאין בו באגים, שהוא קל להבנה ושהוא בклות להשתמש בו למשימות נוספות.

הבה ניקח רשימה פשוטה יחסית ונדגים עליה צורות שונות של כתיבת קוד. בתור רשימה ניקח את הבעייה הבאה: ברצוננו לקבל מהמשתמש רשימה של מספרים, ולבודק אם היא מקיימת חוקיות מסוימת. לדוגמה, שככל 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, ולכן נציג את הפתרון הכללי לביעיה התכנותית). נניח שהמשתמש מכניס רשימת מספרים, וכאשר הוא מעוניין לסיים את הכנסת הרשימה הוא מכניס 'STOP' בתור קלט. לדוגמה:

1

3

5

7

STOP

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


פתרון מודרך

הדרך הראשונה לפתור את התרגיל היא פשוט לקלוט את המספרים אחד אחריוiani, ובכל מספר שנקלט לבדוק אם מתקיים התנאי שהמספר האמצעי מקיים את התנאי שהוגדר (במקרה זה, ממוצע המספר שלפניו והמספר שאחריו). כזכור שתתני המספרים הראשונים נקלוט ללא בדיקה – אי אפשר לבדוק אם התנאי מתקיים עליהם לפני שקלטנו את המספר השלישי. הקוד הבא מבצע את המשימה, אך המשיכו הלאה להסביר ורק אחר כך קוראו 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 פעם אחת בלבד, ואם מתרברר שהנתןיא לא מתקיים או יתר המספרים כלל לא נקלטים. אם כך, האם זה קוד טוב? כלל וכלל לא. נפרט מדוע זה אינו קוד טוב ולא מומלץ כלל לקחת ממנו דוגמה.

ראשית, ספר הלימוד בכוונה אינו מסביר כיצד עובד הקוד. נסו להבין בעצמכם איך הקוד מבצע את המשימה שהוא-Amor לביצוע. דמיינו שאתם מתכוונים במצוות וחברכם לצוות השair لكم את הקוד הזה ויצא לחופשה. לבטא תצלicho להבין את הקוד, אבל המשימה צפואה לגוזל מכם זמן רב יחסית לך לא ארוך. שנייה, אחת הסיבות המרכזיות שהקוד הזה קשה לקרואה (וגם לכתיבת ולדיבוג!) 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 מתחבṭא גם בזמן שלוקח לדבג את הקוד. במילים אחרות, אם יש לכם באג בקוד שנראה כך, צפו לערב ארוך מול המחשב...

גנסה לשפר את הקוד שלנו. בניסיון השני, נפשת את הקוד שלנו באמצעות הפרדה בין המשימות בקוד. בעוד שבקוד של הניסיון הראשון פעולת קליטת המספרים הייתה משולבת בפועלות הבדיקה, בניסיון השני נפריד את קטעי הקוד – יהיה לנו קטע קוד שאחראי לקליטת כל המספרים (שורות 8 עד 14) וקטע קוד אחר שאחראי לבדוק אם התנאי מתקיים על סדרת המספרים (שורות 15 עד 24). בכך, אפשר לטעון שהקוד ביצع את המשימה בצורה יותר מהירה, אבל כזכור המטרה שלנו אינה להציג את מהירות הריצה של הקוד אלא לצמצם את כמות הזמן שלוקח לכתוב, לדבג ולתזוק את הקוד. שימו לב כיצד עצם החלקה למשימות שונות מורידה 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. וудין זה איןנו קוד מוצלח. מדוע? ראשית, הקוד אינו מתווד. בתוכו הקוד אין הסברים לדרך שבה הקוד פותר את הבעיה. אולם, 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 ממשימות נוספות יותר כך ייחסר לנו יותר תיעוד. שנית, קשה להשתמש בקוד זהה למגוון מקרים אחרים. נסביר: נניח שהחבר שעבוד איתנו נתקל גם הוא בעיה בה הוא צריך לעבור על רשימה מספרים ולבדוק אם מתקיימת הטעות הנ"ל. לחבירנו לעובדה כבר יש תוכנית אחרת שדוגמת לקליטת המספרים, כך שלא מתאים לו להעתיק את כל התוכנית שלנו. מה שהחבר יצטרך לבצע הוא להעתיק את קטע הקוד שלנו – שורות 15 עד 24 – ולהתאים את שמות המשתנים לשמות שלנו. מסובך! בשביל זה יש פונקציות. שלישי, האם מוזהם אפשרות כלשהו שהקוד שלנו יקרוט תוך כדי ריצה? מה לדעתכם יקרה אם המשתמש לא נעה להוראות שלנו והזין ערכים שאינם ספורות? בשורה 14, מתבצעת המرة של קלט המשתמש מחרוזת `int`. אם המחרוזת אינה ברת המرة `int` (לדוגמה, נסו להמיר את `'table'` ל-`int`...) או הקוד יקרוט תוך כדי ריצה. אנחנו צריכים להבטיח שהקוד שלנו לא יקרוט גם אם המשתמש מאתגר אותו.

לúcט, אנחנו צריכים להכין 3 שיפורים בקוד:

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

להלן הגרסה השלישית של הקוד, שטפלת בשיפורים הנדרשים. שימו לב, ברגע שהקוד יהיה ארוך יותר, אך בוואו נראה אםicut כל יותר להבין מה הקוד עושה. כיצד לקרוא את הקוד ולהבין אותו במינימום זמן? מומלץ להתחיל מפונקציית ה-main ולנסות להבין מה היא עושה. כדי להבין זאת, תוכלו להעזר בשמות הפונקציות – לעיתים אפשר להבין מה פונקציה עשויה בלי לקרוא אותה כלל. לאחר מכן תתקדמו אל הפונקציות, קיראו קודם כל את ה-docstring שלהם ורק לאחר מכן את הפונקציות עצמן. השוו בין כמות הזמן שהlöקחת לכם הקריאה כתעת לעומת הגרסאות הקודמות. מה יותר מהיר?

```

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

```

```

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 הדרישה, הוא שונכל להשתמש בפונקציות גם בתוכניות אחרות. זאת ממש שהפונקציות לא עוסחות שימוש במשתנים גלובליים – כל מה שהן צרכות מועבר להן בתור פרמטרים. התיעוד המפורט עוזר לנו להבין כיצד לקרוא לכל פונקציה כדי להשתמש בה. דבר זה יחסוך לנו גם זמן כתיבה בעתיד.

assert

מה דעתכם, האם יש עוד מה לשפר בגרסת האחרונה של הקוד? נשאל את עצמנו – איך אנחנו יודעים שהפונקציות שתכתבו אכן עובדות? כמובן, יכול להיות שהן עובדות ברוב מקרים, אך במקרה קצהן הן קורסות. דמיינו פונקציה שמבצעת חילוק – יכול להיות שהיא עובדת כמעט תמיד, אך קורסת אם המכנה הוא אפס...

בנוסף, יכול להיות מצב בו הפונקציה שלנו עובדת, גם במקרה קצהן, אך ביצענו בה שיפור. השיפורגורם לכך שבמקרה קצה הפונקציה שלנו כבר לא עובדת היטב, או קורסת.

נרצה למצוא דרך להרין בבדיקות בדיקות על פונקציה, בין שאנחנו כתבנו אותה ובין שאנחנו משתמשים בפונקציה שמשייתה אחר כתוב. הבדיקות יאפשרו לנו לגלות גם אם הפונקציה רצתה באופן תקין וגם אם גרמו לה בעיה בעקבות שינוי שעשינו בקוד. ישנו מספר דרכים לבדוק את הקוד שלנו, ואנו נזכיר את הדרך פשוטה ביותר – `assert`.

כדי להשתמש ב-`assert`, כותבים `assert`, לאחר מכן כתישו, שיכל להיות `True` או `False`. במקרה שלנו נכתב שם של פונקציה, ולאחר מכן סוגרים עם קלט לפונקציה, ולאחר מכן את הפלט הצפוי מהפונקציה עבור הקלט הנ"ל. ניקח לדוגמה את הפונקציה `list_is_nums`, אשר מקבלת רשימה ובודקת אם כל האיברים בה הינם מחרוזות שניתנות להמרה למספרים. אם כן, `list_is_nums` מוחזרת `True`, אחרת `False`. נוטיף לתוכנית שלנו שני `assert`ים מיד בתחילת ה-`main`:

```
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 שימושיים לפונקציה `list_is_nums`. הקלט הראשון הוא קלט תקין – ארבע מחרוזות שכל אחת מהן ניתנת להמרה למספר. לכן ה-`assert` בודק האם ערך החזרה הוא `True`. הקלט השני מכיל את המחרוזת '`hi`', שמדובר אינה ניתנת להמרה למספר. ה-`assert` מודא שבמקרה זה ערך החזרה מהפונקציה הוא `False`.

החלק העיקרי ב-`assert` הוא להכניס קלטים מיזדים, על מנת לבדוק מקרה קצה. לדוגמה, מה יקרה אם הפונקציה מקבלת רשימה ריקה? במקרה זה נרצה לוודא שיווצרו לנו `False`, לא כך? נבדוק:

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

בשלב הרצת התוכנית נתקבל את הפלט הבא:

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

מה קרה כאן? קיבלנו `AssertionError`. שגיאה שאומרת כך – "ביקשתם שנודיע אם הפונקציה הנבדקה מחזירה ערך שונה מהערך שציפיתם לו, ואכן זה מה שקרה". זאת מכיוון שהפונקציה `list_is_nums` מוחזרה `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 הזו דורשת קלט משתמש. המשמש לא צריך להיות מודע לכך שהתוכנה שלנו כוללת בדיקות. הרי אין זה הגיוני שנבקש מהמשתמש להזין קלטי בדיקה בכל פעם שהוא מרים את התוכנית שלנו...

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

- הפונקציות שלנו לא כוללות הדפסות. זאת מכיוון שאין אפשרות לבדוק הדפסות בעזרת `assert`, שבודק רק ערכיהם שהפונקציה מוחזרה. לכן, הדרך המקובלת היא שהפונקציה מוחזרה ערך, והקוד שקרא לה מדפיס מה צריך לפי הערך שהוחזר. בדוק כמו בדוגמה, בה הפונקציה `list_is_average` מוחזרת `True / False` וההדפסה בוצעה בשורות הקוד שאחרי הקריאה לפונקציה.

זהו! סיימנו את הדיוון במשימה שהוצגה בתחילת הפרק. עברנו ארבע גרסאות קוד שונות והגענו לקוד אליו שאפנו – עובד, נוח לקרוא ובדוק.

עד כה ראיינו שימוש ב-`assert` לבדיקת פונקציות שמחזירות רק `True` או `False`. כמובן שאפשר להשתמש ב-`assert` לבדיקת כל פונקציה. לשם המחשה, נגיד 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 התחלת נבדוק שהפונקציה שלנו עובדת היטב במקרים ה"רגילים":

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

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

כעת תורכם – אילו עוד פעולות `assert` כדאי לעשות על `my_div`? נסו להשוו עלי מקרים קצחים.

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

בנוסף, מה יקרה אם נעביר ל-`my_div` ערכים שאינם מספרים? גם כאן, היינו רוצים לקבל בחזרה מהפונקציה הודעתה כגון 'Parameters are not numbers'.

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

תרגיל מסכם – **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 האחרונה)
- את סכום הספרות של המספר

רגע, מה זה? ראייתי כבר את התרגיל הזה! יש לי דזה וו...

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 שנקבל 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 לקרוא בשם אוףן. הקפידו על כל הדברים הללו של הקוד לפונקציות, שימוש ב-**PEP8**, **טייעוד** ובדיקת פונקציות על ידי **assert**.

סיכום

פרק זה התקדם בשדרוג יכולות התוכנות שלנו. התחילנו מנושא שנראה טכני למדי במבט ראשון – שמירה על קונבנציות לפי **PEP8** – אבל ראיינו את החשיבות של נושא זה לכנתית קוד קרייא, שמתכוונים אחרים יכולים להבין בקלות ולעשות בו שימוש.

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

לבסוף וראינו איך **assert** עוזר לנו לבדוק שהקוד שלנו תקין ועובד היטב. למදנו שכאשר כותבים פונקציה, צריך לחשב על כל מקרי הקצה ולבדק אותן באמצעות קריאה מהתוכונית הראשית.

פרק 8 – קבצים ופרמטרים לסקרייפטים

בפרק זה נלמד שני נושאים שימושיים ביותר כTİת תוכניות פיתון. הראשון, שימוש בקבצים – נראה איך פותחים קובץ לקריאה ולכתיבה. השני, העברת פרמטרים לסקרייפט – ככלומר היכולת להריץ סкриיפט עם מידע שיגרום לסקרייפט לזרז בצווחה 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

בפייתון ישנה פונקציה מובנית בשם `open`, אשר מקבלת בתור פרמטר שם של קובץ. שם הקובץ צריך להיות מהרווז, כאשר מומלץ לשים לפניה את הסיימון `,`, שכפי שמלבדנו מסמן מהרווז `raw`, כך שם הקובץ יוזן כמו שהוא ולא תבוצע המרה לתווים מיוחדים. הייבים להבהיר לפונקציה גם את אופן פתיחת הקובץ, לדוגמה אנחנו יכולים לפתוח קובץ לקרואיה או לפתוח קובץ לכתיבה. אופן הפתיחה נקרא `mode`. נסקרו חלק מה-`mode`ים השונים (תיאור מלא נמצא ב-<https://docs.python.org/2/tutorial/inputoutput.html> בסעיף 7.2):

- לכתיבה של טקסט נשתמש ב-`w`, קישור של `write`
- לקריאה של טקסט נשתמש ב-`r`, קישור של `read`
- אם נרצה לכתוב מידע לקובץ בלי לזרוס את המידע הנוכחי, נשתמש ב-`a`, קישור של `append`. אם לא נפתח כך את הקובץ, אלא נשתמש ב-`w`, כל כתיבה שנכתוב לקובץ תחליל מתחלת הקובץ – ולמעשה התוכן של הקובץ יימחק בכל פעם שנרצה לכתוב אליו.

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

- לכתיבה של מידע ביןארי נשתמש ב-`wb`, קישור של `write binary`
- לקריאה של מידע ביןארי נשתמש ב-`rb`, קישור של `read binary`

דוגמה לשימוש ב-`open` לפתיחת קובץ טקסט:

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

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

קריאה מקובץ

המתודה `read` פועלת על אובייקטים מסוג `file`. בטור ברירת מחדל, המתודה קוראת את כל הקובץ ושמירת אותו בטור מחוזה בטור משנה שהוגדר על ידי המתכנת. לדוגמה:

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

תוצאת פקודת הדפסה:

"Dear Prudence" / The Beatles

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

קל ופשוט. החסרון של שיטה זו הוא שככל הקובץ נקרא בכת 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 אחרת כדי לקרוא קובץ, שורה אחר שורה.

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

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

מה יקרה אם הגענו לסוף הקובץ? במקרה זה הערך שנקבל מ-`readline` יהיה "",(Clomer מחזרות ריקה. דוגמה לקטיע קוד קצר שמדפיס את הקובץ שורה אחר שורה):

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

שימושו לב לכך שבסוף השורה האחורונה 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 רוחה של שורה נוספת נוספת בין כל שתי שורות מודפסות.

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

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

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

בנוסף, שימושו לבן שבו בשם `line` כדי לייצג כל שורה – כך ברור מה המשתנה זהה כולל בכל קריאה. כל יותר לקרוא קו, במקרה אם הוא ארוך, כשלש שמות המשתנים הם בעלי משמעות. זהו אחד מעקרונות כתיבת הקוד הנכון אותו הזכרנו בפרק הקודם.

כתבה לקובי

ראשית אנחנו צריכים לפתח את הקובי לכתיבה (בנהנה שהוא סגור – מיד נראה איך סוגרים קובי). כיוון שהקובץ כבר מכיל מידע, נרצה לפתוח אותו ב-`mode` של `append`. לאחר מכן נשתמש במתודה `write` על מנת לכתוב מידע לתוך הקובי:

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

סגירת קובי

לאחר שימושים את הטיפול בקובץ מומלץ לסגור אותו. אמנם קובי שפותחנו ייסגר אוטומטית ברגע שתסתומים ריצת התוכנית שלנו, אבל אי סגירה של קובי יכולה לגרום לתוכנית שלנו להתנהג בצורה לא צפואה וקשה מאוד לדיבוג. נמחיש על ידי דוגמה. התוכנית הבאה קוראת לפונקציה שפותחת קובי לכתיבה בלי לסגור אותו, וכי להמחייש שזה אינו תכונת נכון, הפונקציה קוריה `open_without_closing`. הפונקציה משנה את תוכן הקובי. לאחר מכן נפתח אותו קובי שוב, הפעם לקריאה. המצביעים לקובי נקראים כאן `fd`, קישור 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 לדעתכם תהיה תוצאה הדרישה? ובכן, בואפן מפתח הדרישה לא כולל 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 שסימנו להשתמש 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, הדרישה שנבעה מתוך פונקציית ה-main תדפיס כמו שצרכיך גם את השורה الأخيرة שהוספנו.

יש אפשרות נוספת יותר, שמאפשרת לנו לפתח קבצים בלי לדאוג לעשות להם `close`. פתיחת קובץ עם הפקודה `with` דואגת לסתירת 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 באוטו אופן שבו הדפסנו אותו קודם. מתי יסגר הקובץ? הקובץ ישאר פתוח רק כל עוד אנחנו נמצאים בבלוק של `with`. ברגע שה בלוק יגמר, יסגר הקובץ אוטומטית.

לשימוש `open` יתרון נוסף: נניה שהשתמשנו ב-`close`, אבל לפני שפיטון הגיע ל-`close` הוא נתקל בשגיאה והתוכנית הפסיקה לroz עט שגיאה. כתוצאה לכך הקובץ שלנו נותר פתוח, למרות שבתוכנית הורינו לסגור אותו. ההוראה `with` גורמת לכך שתבוצע סגירה של הקובץ לפני שהתוכנית מפסיקה לroz ומהזירה שגיאה. כך אנחנו יכולים להיות בטוחים שהקובץ שלנו נסגר בכל מקרה. איך עובדת `with` ואיך היא מצליחה לסגור את הקובץ למרות שרעה שגיאה בדרך? על כך – כשנלמד `.exceptions`.

עד כאן למדנו כיצד להשתמש בקבצים – כיצד לקרוא מהם, לכתוב אליהם ולהוסיף להם מידע. כמו כן ראיינו את החשיבות שבສגירות 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 שלאחר סיום הריצה הקבצים יוכל את אותו טקסט.

קבלת פרמטרים לתוכנית

דמיינו שאתם משתמשים בסקריפט פיתרון שבודק מהهو על המחשב שלכם. לדוגמה, אם תיikit קבצים כלשהי מכילה קבצי פיתון, בעלי הסיומת `.py` או `.cuk`. כאשר אתם מרכיבים את הסקריפט, אתם מתבקשים להזין את שם התיקיה אותה אתם מעוניינים לבדוק. זה בסדר, אבל מהופה מיותר: מילא כאשר אתם מרכיבים את הסקריפט אתם יודיעים על איזו תיקיה תרצו לפעול. למה צריך שהסקריפט ידפיס הודעה ויבקש ממכם להזין קלט? למה שלא תעבירו את שם התיקיה לסקריפט כבר ברגע ההרצה? אם לסקריפט הדמיוני שלנו קוראים `findpy.py`, או נרצה לכתוב ב-`cmd` פקודה כזו:

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

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

בצורה זו, ניתן גם לבדוק בклות הרבה יותר את הסкриיפט. אפשר להריץ אותו עם ערכים שונים ולבודק שקיבלנו תוצאות נכונות. מבחינה טכנית אנחנו רוצים שהקובץ `findpy.py` יהיה כתוב כך שהוא יוכל לקבל פרמטר את שם התיקיה שבה הוא צריך לבדוק האם קיימים קבצי פיתון. הנה נראה איך עושים זאת.

בثور התחלת נכיר בקצתה את המודול `sys`. מודול (או ספריה) הוא קובץ שמכיל פונקציות, ובפרקים הבאים נכיר מודולים נוספים. המודול `sys` מכיל פונקציות שמאפשרות פעולות מערכת שונות, כגון הגדרות של הדפסה למסך וכמו כן – 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` נגיש לנו, כלומר אנחנו יכולים להשתמש בו. כדי להשתמש במבנה שמוגדר במודול, אנחנו צריכים קודם כל לכתוב את שם המודול, לאחר מכן נקודה ואו את שם המבנה. לדוגמה, כדי להשתמש ברשימה `args` שמוגדרת ב-`sys`, צריך לרשום `sys.argv`, בדוק כמו בדוגמה שבkode.

בהמשך הספר, בפרק על OOP, נלמד על שיטות נוספות להשתמש בפונקציות ובמבנה נוספים שנמצאים בתוך מודולים. `args` מאפשרת לנו לקבל את הארגומנטים שהועברו לסקרייפט שלנו. באינדקס ה-0 של הרשימה נמצא שם הסкриיפט שאנו

מראיצים, וביתר האיברים נמצאים הפרמטרים שהעבכנו לסקרייפט (אם העברנו כלה, אחרית הרשימה מכילה רק איבר אחד – שם

הקובץ).

נשדרג מעט את הסкриיפט שלנו ונהפוך אותו לסקרייפט שמקבל פרמטר 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 PyCharm, ולא באמצעות ה-cmd. כיצד מעבירים את הפרמטר ל-PyCharm? זה זמן טוב לזכור לפרך אשר דן ב-[PyCharm](#) ולקרוא את החלק אודות [העברית](#) [פרמטרים לסקריפט](#). קבענו כפרמטר את המחרוזת 'Shooki' וכתזאה מההרצה הודפס למסך:

Hello Shooki

תרגיל – Printer



כיתבו סקריפט שמקבל כפרמטר שם של קובץ ומדפיס את התוכן שלו למסך.



רגע אחד – מה יקרה אם ננסה להעביר לסקריפט שם של קובץ שאינו קיים? נסו זאת. סביר שהסקריפט שלכם יקרוס עם הודעה שגיאה. זו בעיה, מכיוון שגם אם המשתמש שגה והזין שם קובץ שאינו קיים, לא נרצה שהקובץ יקרוס. במקרה זה, עדיף להחזיר למשתמש שגיאה שסבירה לו שהקובץ אינו קיים. כדי לתקן את הבעיה, נזכיר את המודול `os`. מודול `shutil` נוסף זה, כייזור של `Operating System`, מספק יכולות שונות של מערכת הפעלה. בתור דוגמה, נראה איך מציגים שמות של קבצים בתיקייה.

נתחילה נזכיר מ-`os import`. ביצוע `dir` על `os` יראה לנו את כל הֆונקציות ששייכות ל-`os`, ביניהן נמצאת הֆונקציה החביבה `listdir`. בעת 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 לתוכו `directory` את הפרמטר שקיבל הסקריפט שלנו, אז אנחנו מעבירים את `listdir` על מנת לקבל את רישימת הקבצים.

תרגיל – `os.path`



הבעיה בקוד שלנו, כמו בקוד שכתבתם כפתרון לתרגיל `printer`, היא שעדין נהיה בבעיה אם לסקרייפט יועבר שם של תיקיה שאינה קיימת. עלייכם לפתור 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` מכיל מתודה שמקבלת `path` לתיקיה ובודקת אם היא קיימת. תוכלו לקבל את רישימת כל המתוודות באמצעות `(os.path) dir` ותוכלו להשתמש ב-`help` על מנת לקרוא מה עשויה כל מותודה, עד שתמצאו את המתודה המתאימה.

תרגיל מסכם – Lazy Student



קיבלתם כשיעור בית קובץ עם תרגילים חכובן. 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 פתרונים. לדוגמה:

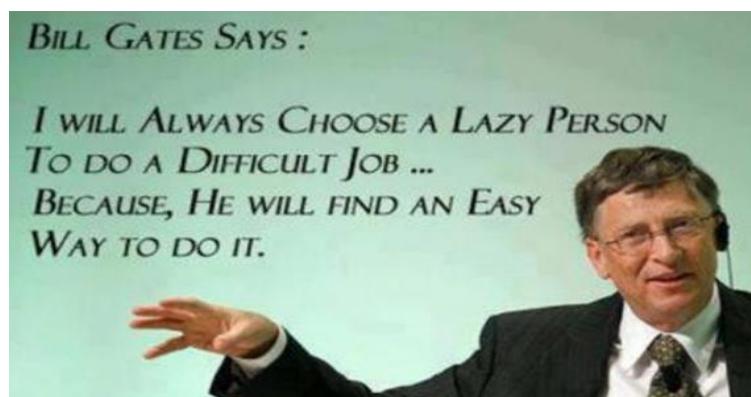
$$46 + 19 = 65$$

$$15 * 3 = 45$$

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

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

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



סיכום

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

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

פרק 9 – Exceptions

בפרק זה נלמד להגן על הקוד שלנו מהתресות בזמן ריצה באמצעות שימוש ב-`exceptions`. כמו שבמוכנית יש גם הגורה בטיחות וגם כריות אויר, כך גם במקרה שמספר אמצעים שנעמדו להגן עליו מקריסה, הם אמצעי בטיחות נוספים.



פגשנו כבר ב-`exceptions` בפרקם הקודמים – בכל פעם שהקוד שלנו "עף", הופך למסך טקסט שכולל 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 שמוקף באדום הוא שם ה-`exception`, והטעסט שכותב לפניו הוא תיאור מפורט של ה-`exception`. אפשר לראות שבכל פעם שכתבנו משהו שהוא-interpreter של פיתון לא הצליח להתמודד איתו, קיבלנו exception. אפשר גם לראות, שקיים exceptions מסווגים שונים, כך שאם ניסינו לחבר מחרוזת עם מספר קיבלנו شيئا שונא מאשר במקרה שבו ניסינו לגשת אל משתנה שאינו קיים.

try, except

הפקודה `try` והפקודה התאומה שלה `except` הן פקודות מיוחדות במיןן, שמאפשרות לנו להריץ כל קוד שאנחנו רוצחים ולדאוג למקרי הקצה בחלק הקוד אחר. כך הקוד שלנו הופך לנקי הרבה יותר. הרעיון של פקודת `try` הוא זה: "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 שלי."
`except`



נראתה דוגמה שימושית. זכרנו שכתבנו סקריפט שמקבל מהמשתמש שם של תיקיה ומדפיס את כל הקבצים שנמצאים בה? החשנו מצב בו המשתמש מכניס שם של תיקיה שאינה 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 "מסוכנות". הראשונה היא הפקודה שקוראת את מה שנמצא ב-[PATH].sys.argv[PATH]. אם המשמש לא הכנס כלל פרמטרים, הקוד יקרוס עקב פניה לאינדקס שאינו קיים. הפקודה השנייה היא כמובן הקריאה ל-dir.os.listdir הכנס, שכאמור תקרוס אם התיקיה לא קיימת. נכניס את שתיהן: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-ifים. הבדיקה הראשונה תהיה על אורך הרשימה argv והבדיקה השנייה על ערך החזרה של פונקציה שבודקת אם התיקיה קיימת. היתרונו של try, הוא שנחסכת מאייתנו כתיבת קוד – אפשר לבדוק את כל המקרים שעולים להוביל לкриיסת קוד באמצעות פקודה יחידה.

מה הבעה בצורת כתיבה זו? אם ארצה שגיאה לא נדע לבדוק מה השגיאה – ישנן שתי אפשרויות. לכן בקרוב נראה איך אנחנו כותבים את ה-`except` באופן שם ניתן לנו מושג אודות השגיאה. אך קודם כל נזכיר עוד קצת את `try` ו-`except`.

לפניכם פונקציה שעלולה להגיע ל-`except`. בידקו את עצמכם: באיזה מקרה הפונקציה תגיע לקוד שנמצא בתוך ה-`except`?

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

אםנו, אם איןנו מחרוזת, הפונקציה תדפיס 'Error'.

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

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

שגיאה נפוצה היא לומר שיודפס 0. הטענה שגורסת שיודפס 0 היא "בתחילת התוכנית x מקבל את הערך 0. הריצה של `try` לא מתבצעת עקב ה-`exception`, לכן ערכו של `x` נותר 0 כאשר הוא מגיע ל-`except`". הטעות היא בכך שהחלק מה-`try` דוקא כן מבוצע. כל מה שקדם לקפיצה ל-`except` בהחלט יבוצע. במקרה זה, מתבצעת השורה בה מושמים ב-`x` את הערך 1, ולאחר זה ערכו כאשר הקוד קופץ ל-`except`. כמובן שהתוכנית אינה מגיעה לשורת הקוד בה `x` הינו 2.

השורה התהודה היא, גם אם התרחש `exception`, כל הפקודות שכבר הטענו עדין תקפות. המעבד אינו "חזר אחורה" ...

סוגים של Exceptions

נזהר אל הפונקציה שהצגנו 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', ישנו שתי אפשרויות לבעה:

- המשמש לא הזין כלל פרמטר, וכך הניתה ל-sys.argv[PATH] המבוקש לא吟דקם לא חוקי ברשימה.
- המשמש הזין תיוקה שגוייה, וכך os.listdir יחזיר שגיאה.

הגינוי שנרצה לדוחה למשתמש מה הייתה הבעה. 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` משמעותה "הכנס את הודעת השגיאה של ה-exception לתוך המשתנה `e`". כפי שראינו לכל exception יש הודעת שגיאת יהודית, המשתנה `e` יכול אותה. בדוגמה הבאה אפשר לראות איך אנחנו יכולים שונים ומטפלים בכל אחד מהם בנפרד. אם הייתה שגיאה שלא מסוג ZeroDivisionError או TypeError, היא תיתפס על ידי ה-exception האחרון:

```

|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 (מאוחר יותר נלמד תכונות מונחה עצמים), אך `e` הוא אובייקט שמכיל את כל השדות של `Exception`, כולל הودעת השגיאה.

`finally`

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

הקריאה הראשונה ל-.`finally` תרוץ ללא בעיות מיוחדות. אך יורץ הקוד שנמצא בין `try` וב-`finally`. הקריאה השנייה תגרום מיד לזריקת `exception`, וכך מעבר ל-`except` ולאחר מכן מכוון לביצוע `finally`. תוצאת הריצה תהיה ההזפסה הבאה:

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

נסקרו דוגמה 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))
```

תוצאת ההריצה היא:

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

בקריאה הראשונה לפונקציה, ה-`try` מסתיים בהזרת ערך. כאן מתרחש משהו מעניין – ה-`interpreter` של פיתון מזהה שאחנו 'Final' וומודים ליצאת מהפונקציה ולכн הוא בודק אם ישנו קוד ב-`finally` שעליו לבצע לפני כן. לאחר ביצוע פקודת הדפסה של 'Final' מותבצעת חזרה אל בלוק ה-`try` ומשם מתקבל ערך החזרה 'OK'.

בקריאה השנייה לפונקציה מtbody תהליך דומה, קופיצה ל-`finally` וחזרה, אלא שהוא מtbody מה-`except`.

כעת 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 הגיע אל ה-`except`, שם כפי שראינו בדוגמה הקודמת ה-`interpreter` של פיתון יקפוּן אל `finally`. ההבדל מהדוגמה הקודמת, הוא שבמקרה זה מכילה הוראת חזרה ולכн בכך תסתיימים ריצת הפונקציה. תוצאה הדפסה:

123 returns: Final

with

וכך, כאשר למדנו על פתיחת קוביּץ באמצעות `open`, אמרנו שגם אם מתרחשת שגיאה כלשהי במהלך הריצה עדין הקוביּץ ייסגר. כיצד זה מתרחש?

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

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

שколо לקוד הבא:

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

כפי שאנו רואים, `with` כולל בתוכו `finally`, אשר דואג לסגירת הקובץ בכל מקרה – גם אם ארצה שגיאה במקומם כלשהו בבלוק שישיר 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, חלקה באפס) התוכנית תדפיס הודעה שגיאה מדויקת ולא רק שארעה שגיאה כלשהי.

קיבלו כישורי בית קובץ עם תרגילים שונים. כל תרגיל הוא בפורמט הבא: מספר-רואה-פעולה-רואה-מספר. לדוגמה:

$46 + 19$

$15 * 3$

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

$46 + 19 = 65$

$15 * 3 = 45$

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

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

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

תרגיל מסכם פיתון בסיסי – LogPuzzle



(เครดיט: google class, התאמת לגביהם: דנה אבן חיים, 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 תצטרכו להרכיב תמונה מחלקים אשר פוזרו באינטרנט. בטור התחלה, נראה כיצד אפשר להוריד כל קובץ שנמצא באינטרנט באמצעות פיתון, תוך שימוש במודול `urllib`.

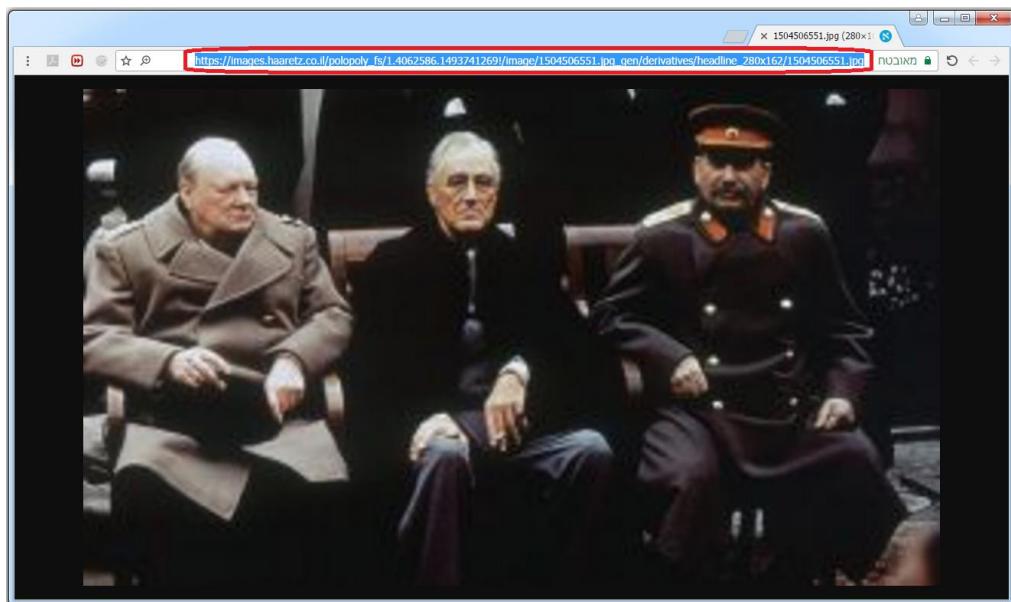
ראשית – מהו URL? אלו ראשי תיבות של Universal Resource Locator. לכל משאב באינטרנט יש URL מסויל. לשם המחשה, יכול להיות שרת האינטרנט שכתו www.cyber.com. בתוך השרת זהה שמור קובץ pdf בשם `file.pdf` ועמו html בשם `page.html`. ניתן יהיה להגיע לכל משאב לפי ה-URLים הבאים:

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

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

כעת אנחנו בשלים להבין את השימוש ב-`urllib`. מודול זה כולל פונקציה שנקרא `urlretrieve` – `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, או על "פתח תמונה בכרטיסיה חדשה". לאחר שהתמונה תיפתח, עיברו אל הכרטיסיה החדשה שנפתחה והעתיקו את ה-URL של התמונה משדה הכתובת של הדפסן.



.import urllib וכתיבתו

את ה-URL הדיבקו כך שישמש בתור הפקטור הראשון לפונקציה `urlretrieve` והוסיפו שם של קובץ אליו יש לשמר את הקובץ בהתאם מוריידם. כמובן שצריך לדאוג לכך שם הקובץ יהיה בעל סימות זהה לקובץ שאתם מוריידם. לדוגמה, אם אנחנו מוריידים קובץ עם סימות jpg או שם הקובץ אליו אנחנו שומרים צריך גם הוא להסתיים ב-jpg. סימת הקובץ שאנחנו מוריידים נמצאת תמיד בתווים האחרונים (הימניים) של ה-URL.

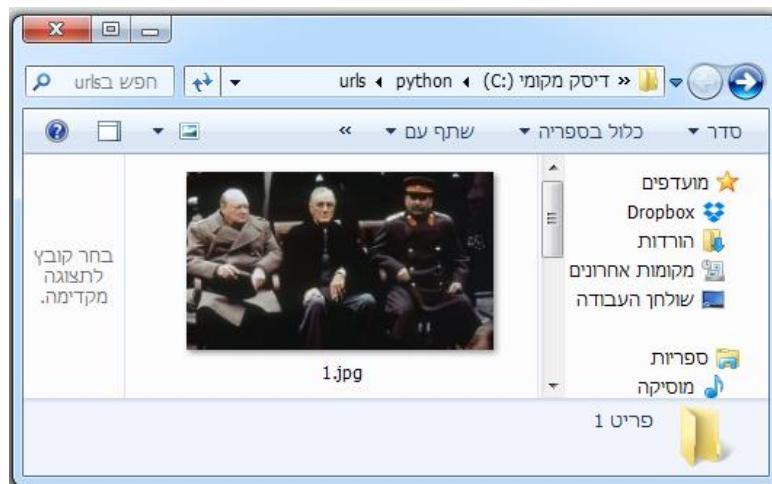
לדוגמא:

```
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 שנצטרכן ללמידה כדי לפתור את הפאל – חיבור תמונות לתמונה אחת. לשם כך נשתמש בקובץ `html.html`. כאשר רושמים בתחום הקובץ את שמות התמונות, הפעלת הקובץ מציגה אותן זו לצד זו.

כל מה שאנו צריכים לדעת כדי לכתוב את קובץ ה-`html` שנדרש בשביל לפטור את הפאל, הוא כיצד להשתמש בתבנית הבא:



```

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

```

הקלקה על הקובץ שיצרנו תפתח דפדף שבו תוצג התמונה שהורדנו – פעמיים.



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

לאחר הקדמה קצרה זו, ניגש לתרגיל עצמו. בリンク הבא נמצאים שני קבצי לוג. בכל קובץ לוג ישנו טקסט, שמתהבאים בו אweis 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/loqpuzzle.zip>

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

מה ה-URL המלא של הקבצים שאתה מוריידים?

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

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

איך מותבצע המיון?

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

בלינק ישנו קובץ בשם logpuzzle.py, שהוואה את קובץ השلد לפתרון התרגיל. הוא כולל את הפקנציות שנדרשות לביצוע המשימה, ועליכם להשלים את הקוד החסר בפקנציות.

כעת תורכם – פיתרו את הפازל. בהצלחה!

פרק 10 – תכונות מונחה עצמים – OOP

בפרק זה נלמד כיצד מביצעים תכונות מונחה עצמים בפייתון. ייתכן ששמעתם על המושג "תכונות מונחה עצמים", או באנגלית Object Oriented Programming בפרקם הקודמים. הנושא הוא רחוב, ולכן הפרק יתחלק ל-5 חלקים:

א. **מבוא – מדוע בכלל צריך תכונות מונחה עצמים, class, object**

ב. **כתבת class בסיסי**

ג. **כתבת class משופר**

ד. **ירושה – inheritance**

ה. **פולימורפיזם**



מבוא – מה OOP?

לפני שאחנו לומדים נושא חדש, צפוי שנשאלו את עצמנו מה נרויה מזו. הלא עד כה הצלחנו לפטור את כל המשימות שקיבנו. התשובה היא – ידע בתכונות מונחה עצמים יאפשר לנו לכתוב קוד הרבה יותר מהר מאשר אל מולו היה לנו את הידע הזה. נמיהיש על ידי דוגמא.

אנו רוצים ל כתוב תוכנית שתשתמש בית ספר. יש צורך שנשמר את שמות כל התלמידים ואת הציוןיהם שלהם במקצתות השונות. אנחנו יכולים לעשות זאת כך – לכל תלמיד ולכל מקצוע של תלמיד נקצתה משתנה:

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

```
talmid1_english = 90
```

```
talmid1_math = 95
```

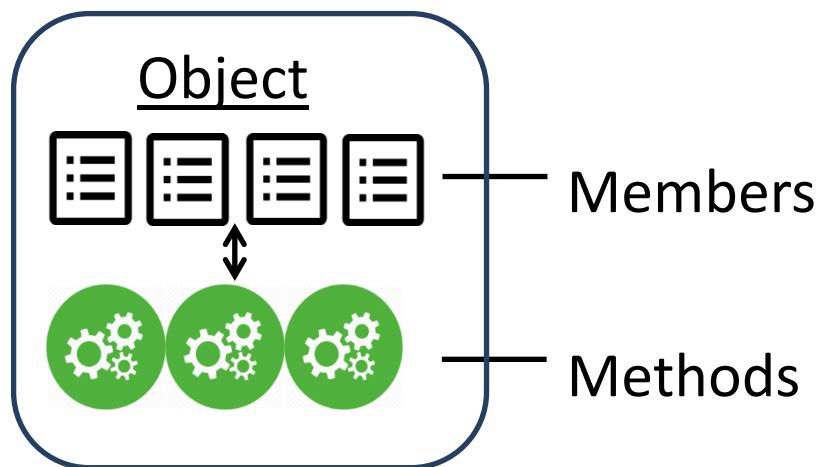
```
talmid1_geography = 85
```

אם יש לנו 200 תלמידים ו-10 מקצועות לכל תלמיד, בלי OOP נצטרך להזכיר שם של משתנה לכל תלמיד ולכל מקצוע של כל תלמיד. אנחנו בדרכ הבטוחה להציגו לפני משתנים... גם רשימה של ציונים עבור כל תלמיד אינה רעיון מוצלח, כיון שנצטרך לעבוד עם אינדקסים כדי להציג את המקצועות השונים. באמצעות OOP נוכל לזמן את כמות המשתנים שלנו. נלמד לבנות `class` של תלמיד, ובאמצעות 200 אובייקטים מסוג תלמיד – מיד נלמד גם מהו אובייקט – נזקיק את כל המידע על התלמידים. נציגנו את כמות המשתנים שלנו מאלפים למאות. נחמד? ניתן לטעון בצדק שאנו הציגו שאמנם היצום מאלפים למאות הוא נפלא, אבל גם לבצע פעולות על מאות אובייקטים יגרום לנו לתוכנת קוד די ארוך. ובכן, נראה שאפשר לעבור על כל האובייקטים עם לולאת `for` וכך לkürר עוד יותר את הקוד שלנו.

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

אובייקט – object

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



אובייקט מורכב מידע ומETHODות שפועלות על המידע

נתקלנו כבר במתודות והסבירנו שהה סוג מיוחד של פונקציות. אם כך, עכשו אנהנו יכולים להגיד את ההסבר – מתודה היא פונקציה, אבל לא סתם פונקציה אלא פונקציה שמוגדרת כחלק מאובייקט ופועלת על האובייקט. כמובן, רק האובייקט מכיר אותה ורק הוא יודע איך לפעול אליה. לדוגמה, ראיינו שקובץ הוא אובייקט. לאובייקט מסווג `file` יש מתודה שקוראים לה `read`. אנהנו לא קוראים לה בצורה זו:

`read(filename)`

הרי `read` לא יודעת לקבל שום דבר שהוא לא `file`. מעבר לכך, בקריאה כזו – פיתון חושב שאנו מתייחסים לפונקציה כללית בשם `read`, ולא מתודה של אובייקט ספציפי. במקום השימוש לעיל, אנהנו משתמשים בה כך, עם נקודה:

`filename.read()`

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

לסיום, אובייקט מכיל גם מידע וגם מתודות. ראיינו איך ניתן למתודה של אובייקט, מיד נראה איך "מייצרים" אובייקט שככללו גם מתודות וגם מידע.

מחלקה – `class`

מחלקה, או `class`, הוא קטע קוד שמנדרט את כל ה-`members` והמתודות של אובייקט ואשר משותפים לו וליתר האובייקטים של אותה מחלקה. נמחיש זאת. ניקח את כוכב הלכת שבתאי. זה כוכב לכט אחד מתוך מספר כוכבי לכט במערכת השמש שלנו. למרות שכל אחד מהם הוא שונה, לכולם יש מסה, רדיוס, מרחק מהשמש, זמן הקפה של השימוש וכו'. אפשר להגיד מחלקה של כוכבי לכט, לדוגמה בשם `planet`, אשר תכיל את כל המאפיינים המשותפים לכל כוכבי הלכת.



בכל פעם שנרצה להגדיר כוכב לכת, פשוט נשתמש במחלקה `planet` – היא מכילה כבר את התכנית לשימורת כל המידע. אפשר לדמיין ש-`planet` הוא דף מידע שמכיל את כל השודות שצורך בשבייל להגדיר כוכב לכת.

Planet

שם הכוכב:

מסה:

מרחק מהשמש:

זמן הקפה:

טמפרטורה:

או מה ההבדל בין אובייקט למחלקה? מחלקה מתארת את המאפיינים של כל האובייקטים שישיכים למחלקה. כשהחוכנות רצתה, בכל פעם שנגידיר אובייקט מסוים, יוקצה זיכרון בהחשב בהתאם לכמות המידע שצורך 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 לייצר עוגיות. בכל פעם שנשתמש בתבנית, חתיכה של בזק מקבל את הצורה של התבנית זו. כמות הבזק שנקצתה לעוגיה נקבעת על ידי התבנית, בדומה לכך שכמויות הזיכרון שמקצתה לאובייקט נקבעת על ידי המחלקה.



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

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

כabitת class בסיסי

נדמיין שאנו מפעילים מגדל פיקוח, ששולט בתנועת המטוסים סביבה שדה תעופה. תפקידנו למנוע התנagesיות מטוסים באוויר. בכל רגע נתון, יש באוויר מספר מטוסים, והבעיה היא שהטייסים השתגעו וטסים בכיוונים אקרים. תפקידנו הוא לעקוב אחריו. תנועת המטוסים באוויר.



כדי לתכנת את הפתרון, ראשית אנחנו צריכים ליזור 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 להלכים.

שם ה-`class` הוא `CrazyPlane`. שמו לב לכך שהשם מתחילה באות גדולה – זה הקובנציה לקביעת שמות של מחלקות. מיד נסביר מהם ה-`__init__` וה-`self`. לאחר מכן מוגדרים שני `members` (זוכרים שאמרנו שאובייקט מורכב ממתקודות ומ-`members`). הראשון הוא `x` והשני הוא `y`. כל אחד מהם מקבל ערך התחלתי – 0.

__init__

זהוethodה הראשונה שנזכיר. השם `init` נזר מ-`initializer`, או אתחול. אפשר לקרוא לה גם `constructor` או בניין. בתוכה `__init__` נהוג לשים את האתחול של כל ה-`members` של המחלקה, כפי שראינו בדוגמה. זהו methodה 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 פיתتون יודע על איזה אובייקט להפעיל את המתודה? במקרים אחרים, אנחנו יכולים ליצר כמה אובייקטים ששוויכים לאותו `class`, לדוגמה כמו זהה תבנית עוגיות שימושת לייצור עוגיות רבות. אם אנחנו רוצים להפעיל מתודה על עוגיה מסוימת, איך פיתتون יודע על איזה עוגיה הוא צריך לפעול?

התשובה היא `self`. אנחנו מעבירים למתודה מצביע לאובייקט שלו היא אמורה לפעול. אנחנו אומרים למתודה – "הפעיל על העוגיה הזו" בזמן שהאצבע שלנו מצביעה על עוגיה מסוימת. אי לך, נעביר את `self` כפרמטר לכל מתודה של המחלקה שצפוייה לזרז על האובייקט הזה.



הוספת מתודות

לאחר שכתבנו את המתודה הראשונה, `__init__`, הגיע הזמן להוסיף מתודות שעשוות פעולה שאין אתה. כיוון שמדובר במטוסים, 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` פשוט מחזירה את מיקום המטוס.שוב, נשים לב לכך שככל מתודה צריכה לקבל את `self` בתור פרמטר. כמו כן, נעשה שימוש במודול `random` וכן נדרש לבצע `import random` בתחילת הקובץ.

Members

כפי שŁמדנו זה עתה, ברגע שניצור אובייקט מסווג `CrazyPlane` יוצרו לנו x ו-y ששייכים 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 בינו לבין x,y:

```
class CrazyPlane:

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

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

המתודה `count_down` סופרת מ-10 ומטה, ומוגדר בה `i`. המשתנה `i` "חי" רק בתחום לולאת ה-`for` של המתודה. ברגע שהלולאה מסתיימת, `i` אינו קיים יותר ולא ניתן לגשת אליו. זאת לעומת `x,y`, שקיים 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 יוצר אובייקט בשם `plane1` ומשתמש באובייקט על מנת לקבל את מיקום המטוס:

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

חישבו – 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` ואת `y` `plane1.y` ומאתחלת את ערכיהם ל-0. המתוודה `get_position` פועלת על האובייקט `plane1` ומחזירה את ערכי ה-`x` וה-`y` שלו, כמובן 0, 0. נטו זאת בעצמכם – צרו אובייקט ובידקו שקיבלותם את ערכי ההתחלה שקבעתם.

נעשה בתוכנית שלנו שינוי קטן. במקום `posx` נקבע `x` ובמקום `posy` נקבע `y`. האם התוכנית תעבוד כעת, או שתתקבל שגיאה?

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

התשובה היא שהתוכנית תעבוד ללא כל בעיה. חשוב להבין ש-`y`, `x` שהגדכנו בפונקציה `main` `plane1` אינם `y`, `x` ששיכים ל-`plane1`. במילים אחרות, `x` אינו `plane1.y` ו-`y` אינו `plane1.x`. אין להם את אותו `id`. גם אם נשנה את `x`, ערכו של `plane1.x` לא ישתנה.

תרגיל



כתבו `class` של חיה אהובה עליכם (לדוגמא `Cat`, `Dog` וכו').

- הוסיפו מתודה `__init__` שתכלול את שם החיה (לדוגמא `Kermit`) ואת גיל החיה

- הוסיפו מתודה `birthday`, שתעללה את גיל החיה ב-1

- הוסיפו מתודה `get_age` שתחזיר את גיל החיה

שימוש לב בשלב הזה כל חייה שתיצרו באמצעות התבנית של המחלקה יקבלו את אותו שם אשר מופיע ב-`__init__`. בקרוב נלמד כיצד ניתן לכל חייה שם שונה בשלב ההתחול.



כתיבת 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 שנייתן לייבא על ידי import**
- **לקבוע ערכיהם ההתחלתיים לאובייקט חדש**
- **לייצור פקודת הדפסה מיוחדת ל-class שלנו**
- **לייצור מתודות accessor ו-mutator**

יצירת members "מוסתרים"

קטע הקוד הבא דורס את נתוני המיקום של המטוס:

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

המתודה `get_position` תחזיר את הערכים 10, 10. בעולם האמיתי, תוצאה של קוד כזה עלולה להיות התנגדות בין מטוסים. علينا למצוא דרך "להסתיר" את נתוני המיקום של המטוס, כך שהתוכנה שעושה בהם שימוש לא תנסה אותן בטעות.



בפייתון, הסתירה של `members` מתבצעת באמצעות תחילתית `__` (קו תחתי 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 הכפול גורם לכך שרק מетодות של `CrazyPlane` מכירות את ה-`members` הללו. מי שמשתמש ב-`class` שלנו כבר לא יוכל לראות שם קיימים. שימו לב להבדל: לפני ההסתירה, כתיבה של "plane1." הולטה אפשרויות שונות, ביניהן `x` ו-`y`.

```
plane1 = CrazyPlane()
plane1.|
```

x, y = f x
 ⚡ get_position(self) CrazyPlane
 ⚡ update_position(self) CrazyPlane
 f y CrazyPlane

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

```
plane1 = CrazyPlane()
plane1.|
```

x, y = ⚡ get_position(self) CrazyPlane
 ⚡ update_position(self) CrazyPlane

האם זה אומר שכבר אי אפשר לגשת אל members מסוימים? לא. פיתון מאפשר לנו גם אפשרות זו. בשפות עיליות י箇נו מושג שנקרא "private", שמשמעותו משתנים או מתודותuai איפואן יכול ל-*class*. בפייתון אין private, יש רק הסתרה. אם מתעקשים, אפשר לגשת אל members אפילו אם הם מסוימים.

זכור פונקציית `dir` מחזירה את כל המתודות וה-members של אובייקט מסוים. אם כך, נעשה `dir(plane1)` ונקבל:

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

שםו לב לשני האיברים הראשונים בראשיה, אשר מסתירים ב-"`x`" ו-"`y`". כו, נראה דומה למה שאנחנו מחפשים. נסח לפנות אליהם:

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

הקוד רץ ללא שגיאות וכאשר מדפיסים את הערך שמחזירה `get_position` מקבל (5,5), מה שמעיד על כך שאנו שינו את הערכים.

האם מדובר בBegan של פיתון? לא, אך פיתון תוכננה. הרעיון הוא לאפשר גמישות מקסימלית לתוכנותם. פיתון אומרת "ראו, יש סיבה שהמשתנים הללו מוסתרים. מי שכתב את הקוד לא רצה שתוכנית חיצונית תיגש ותשנה אותם. אבל אם אתם יודעים מה אתם עושים, והקצתם את הזמן להתגבר על ההסתירה, אז בבקשה – שנו כל מה שתרצו". יכולות זו שימושית רק במקרים נדירים, אבל במקרה הצורך היא אפשררת לנו, לדוגמה, לבצע שינויים בספריות פיתון שיש בהם אגפים, בלי להזדקק לקוד המקור.

אם כך, משתנים מוסתרים בפייתון הם למעשה דרך שלן Ci אין לשנות את תוכן המשתנה שלא על ידי קוד של המחלקה עצמה. עם זאת, זה רק סימון – ומשתמש חיצוני יוכל לשנות את ערכיו המשתנה המוסתר, אם יבחר בכך.

שימוש ב-mutator וב-accessor

סקרנו שתי שיטות גישה ל-members של מחלקה.

השיטה הראשונה, היא לגשת אל ה-members יישירות. לדוגמה `x1.plane` היא גישה ישירה ל-member `plane` של `x1`. אם מי שתוכנת את המחלקה `CrazyPlane` דאג להסדיר את `x`, עדין יוכל לגשת אליו בעזרת `x._CrazyPlane`.

השיטה השנייה, היא באמצעות שימוש בethodות שנמצאות במחלקה ומשמשות במיוחד לטובת קרייה ושינויי של members. לדוגמה, `get_position` היא דוגמה לmethod כזו. ראיינו שאם נקרא ל-`plane1.get_position()` נקבל את ערכו המיקום, למרות שהם מוסתרים. למtodoה שמאפשרת קרייה של members של מחלקה קוראים accessor ונהוג שהיא מתחילה ב-`get`.

מזהה שמאפשרת שינוי ערכיהם של members נקראת mutator, אם נרצה לאפשר שינוי המיקום של המטוס, נגידר פונקציה בשם `set`, שתקבל מיקום כפרמטר ותשנה בהתאם את מיקומו המטו.

יש דיון נרחב האם שימוש ב-accessors ו-mutators (ashr 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 שביכולתו של המתכנת לשלוט בערכיהם המוכנסים ולבודא תקינות. לדוגמה, אם המתכנת יודע שיש מקום בעייתי שאסור שימוש יימצא בו הוא יכול למנוע זאת באמצעות קוד מתאים. הקוד הבא לא מאפשר להזין למטרוס מיקום שלילי או את מיקום 4,4 כיון שהוא אינו מאפשר מעבר מטוסים:

```
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'
```

מайдן, 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 לפטוק בשאלת מהי השיטה העדיפה. נציין שהשווים להכיר את שתי השיטות מכיוון ששתייהן נפוצות בקוד בו אתם עושים לhitkak בעtid, והשוב לדעת להשתמש ב-accessors ו-mutators, ولو בשbill לקלבל 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? משומש שם נרצה להשתמש במחלקה שלנו בתוכנית אחרת, נדרש להעתיק את כל הקוד שלנו ואז לשנות מה שצורך. לא פתרון נוח. היינו רוצים שתוכנית אחרת תוכל "לייבא" רק את המחלקה שהגדכנו, ואולי כמה קבועים שקשורים אליה – וזהו.

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

נשמר קובץ פיתון בשם planes.py. לתוכה 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'
```

cut גדרת תוכנית שמשתמשת במחלקה שנמצאת בקובץ :plane.py

```

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()

```

בשורה הראשונה אנחנו מבצעים import לקובץ planes. כעת זו הוזמנות נחרמת 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 המקורי – יש לנו `main` שלנו שנחנו עובדים אליו. אנחנו רוצים רק את ההגדרות של המחלקות, וקובעים כלשהם אם הם קיימים, לדוגמה מקום המגדל. תנאי ה-`if` הנ"ל מודיע על פיתון "אם הגעת לקובץ זה תור כדי ריצת התוכנית ולא בעקבות `import`, תריץ את `main`" – בדיק מה שרצינו שקרה.

לאחר שביצענו `import`, אנחנו יכולים להשתמש שיבאנו אל התוכנית שלנו. השימוש במחלקות מצרך שינוי קטן בקוד – לפני שאנו מגדירים אובייקט מסווג `CrazyPlane` צריך לסמן שהאובייקט שייך לקובץ `planes` שיבאנו, כפי שניתן לראות בשורה השלישית.רגע – איך פיתון לא יודע בעצם ש-`CrazyPlane` הוא מחלקה שמודדרת ב-`planes`? ובכן, פיתון יודע, אבל פיתון מניה שיכל להיות שיסנה עוד מחלוקת בשם זה בקובץ אחר. מקרה זה עולב בהחלט להתרחש, במקרה שיש מתכנתים רבים שכותבים מודולים בנפרד. לכן, פיתון דורש שנציגו בפירוש לאיזה קובץ אנחנו מתכוונים. ואם בכלל זאת אנחנו רוצים לוותר על ציון שם הקובץ? אפשר לוותר על כך בשינוי קטן. שמו לב לזרה הבאה של `import`:

```
from planes import CrazyPlane
```

```
def main():
    plane1 = CrazyPlane()
```

עתה פיתון יודע שם אנחנו כתובים `CrazyPlane` אנחנו מתכוונים רק למחלוקת הנ"ל מתוך `planes`, ולכן אפשר לוותר על ציון שם הקובץ. שמו לב לכך שם אכן היה קיימן 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 מתיקן הקובץ – ולשם כך הוספנו לקובץ עוד מחלוקת – גורם לכך שכעת רק המחלוקת `CrazyPlanes` יובאה. המחלוקת השנייה, `NormalPlane`, נותרה לא מוכרת לתוכנית הראשית.

לעתים תיתקלו ביבוא של כל המחלקות והקבועים של הקובץ באופן הבא:

```
from planes import *
```

```
def main():
    plane1 = CrazyPlane()
    plane2 = NormalPlane()
```

צורה זאת של ייבוא עלולה לגרום לבאגים. מדוע? מכיוון שיכל 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 * import, מה יקרה כאשר נרצה ליצור אובייקט מסווג ?CrazyPlane אם מה שמוגדר ב-planes או ב-?airports התשובה היא שהקובץ שייבאנו אחרון "דורס" את ההגדרות הקודומות. לכן, שיטה זו אינה מומלצת.

אתחול של פרמטרים

אפשרויות הטישה של המטוסים השתנו, כאשר נפתח לרווחת הציבור שדה תעופה נוספת. כעת אנחנו רוצים לאפשר לחלק מהמטוסים להמריא משדה התעופה החדש. נניח שהמקום של שדה התעופה הוא (5, 5). כמובן, אנחנו צריכים להעביר את קואורדינטות המרਆ 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 ממריאים משדה התעופה הישן, היינו רוצים שברירת המחדל למטוס חדש שמריא 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
```

כעת, פיתון בודק אם כאשר אנחנו יוצרים מטוס חדש אנחנו מעבירים את המיקום כפרמטר. אם כן – המיקום שהעבכנו קבוע, אחרת – מיקום ברירת המחדל קבוע. נוכל להגדיר מטוסים חדשים באופן הבאים:

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

האובייקט `elal` קיבל את ערכי המיקום של ברירת המחדל – כלומר 0, 0, ואילו האובייקט `american` קיבל את ערכי ההתחלתה שהעבכנו לו – מיקום שדה התעופה של ניו יורק.

המתודה `__str__`

ננסה לעשות שימוש בMETHOD `print` להציגו, באמצעות הפקודה `:print american`

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

איך פיתון יודע מה להדפיס כאשר אנחנו עושים `print` לאובייקט מסוים? לכל אובייקט ישנה מתודה `__str__`. אנחנו יכולים לדרוס את `__str__` ולהחליפּ אותה בכל צורת הדפסה שנרצה. הנה נבצע זאת:

```
def __str__(self):
    return 'Plane position: {}'.format(self.get_position())
```

הסביר: ראשית, המתודה צריכה להחזיר ערך, שהואurreturn. לכן ישנו במתודה `return`. הערך שmorphozero הוא מהרוות, שחלקה קבוע וחולקה מקבל את הערך שמחזירה המתודה `get_position` ומעביר אותו לפורתם מהרוות. אם עכשו נבצע `print`, נקבל:

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

זהו. יותריפה ממה שהחזירה פקודת הדפסה לפני השינוי? ☺

המתודה `__repr__`

המתודה `repr`, קיצור של `represent`, היא דרך נוספת המשמשת להדפסה של אובייקטים. אך בעוד `__str__` משמשת לצורך הצגיה יפה של נתונים למשתמש, המתודה `__repr__` משמשת להצגת נתונים למתקנת, על מנת לסייע בדיבוב.

נמחיש זאת באמצעות הדוגמה הבאה (เครดיט לרענון – <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`, באוטומטי נקראת המתודה `__str__` של `datetime`, אשר בה מוגדר הזמן בפורמט נוח לקרוא. לעומת זאת, כאשר בדקנו מה ערכו של `t` קיבלנו את התוצאה של `__repr__` של `datetime`, אשר נתנה לנו יותר מידע על מה שמתאר "מזהורי הקלעים". ניתן לכתוב את `repr` עבור כל מחלקה שאחננו כתובים, על ידי הגדרה של `__str__`, בדומה לכך שבהגדכנו את `__repr__`.

יצירת אובייקטים מרובים

עד כה יצרנו אובייקט אחד מהמחלקה שלנו. אפשר ליצור מה"מבנה" כמה אובייקטים שאחננו רוצים, רק צריך לדאוג שלכל אובייקט יהיה שם ייחודי, אחרת נדרוס אובייקטים שכבר הגדכנו. נגיד ארבעה מטוסים:

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

מeos אללול מרים מנקודות ברירת המחדל שלנו ואילו היתר מרים ממיקומים שונים שמועברים כפרמטרים.

אם אנחנו רוצים לבצע פעולה כלשהי על כל המטוסים, נכניס אותם לרשימה (אם אנחנו רוצים לרשימה) או מוסיפים – לאחרת (tuple) וואז נוכל לעבור על כלם בלאלה:

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



תרגיל סיכום בינוי – כתיבת class משופר



שפרו את ה-class של החיה שיצרתם:

העבירו את ה-class `klass` לקובץ `chizoni` ועשו לו `import`

הסתירו את שם החיה ואת הגיל שלה (תזכורת: `(__)`)

אפשרו לקבע את שם החיה בזמן ייצירת האובייקט

אפשרו לשנות את שם החיה (`set` מתודה)

אפשרו לקרוא את שם החיה ואת גיל החיה (`get` מתודות)

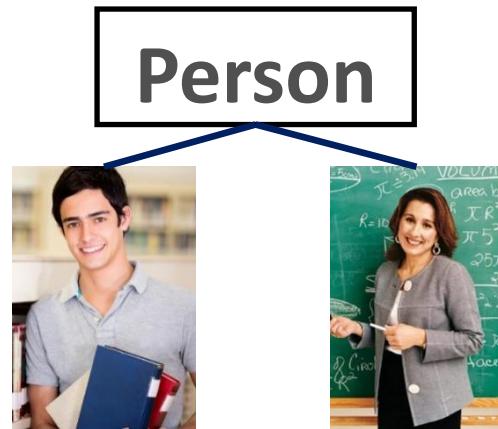
אפשרו הדפסת פרטי החיה על ידי קרייה ל-`print` (מתודת `__str__`)



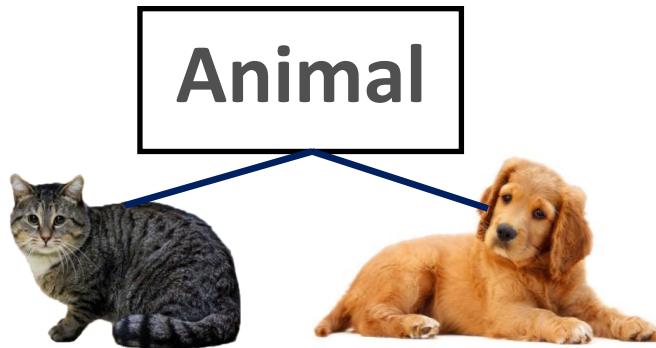
ירושה – inheritance

לפעמים מחלוקת היא סוג ספציפי של מחלוקת אחרת. לדוגמה – חתול הוא דוגמה ספציפית של חייה, ומוטס הוא דוגמה ספציפית של כלי תחבורה. הרעיון של ירושה מאפשר לנו ללקח מחלוקת קיימת וליצור ממנה מחלוקת חדשה, שכוללת את כל התכונות שקיימות בחלוקת שירשנו ממנה ועוד תכונות נוספות, שהן מיוחדות רק למחלוקת החדשה שיצרנו. המחלוקת החדשה, זו שיורשת מהחלוקת הקיימת, נקראת **superclass**. המחלוקת שממנה ירשנו נקראת **subclass**. לדוגמה:

"Person" superclass של ה-**subclasses** Student ו- Teacher -



"Animal" superclass של ה-**subclasses** Dog ו- Cat -



כדי להציג איך 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 נומכה מ-3 (גרסת הפיתון שלנו היא 2.7).
בגרסה 3 פיתון כבר מניח מראש שהירושות הוא מ-object ולכן אין צורך לכתוב זאת במפורש.

כעת ניצור בית ספר. בבית ספר יש מורים ותלמידים.

למורים יש:

שם -

גיל -

שכר -

לתלמידים יש:

שם -

גיל -

- ממוצע ציונים

יש לנו כבר את המחלקה Person. בואו נשתמש בה. נזכיר שהמחלקות חדשות יורשות מ-Person – שימוש לב שמותינו זהה 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 יצרנו אובייקט מסווג Teacher, פיתון מחפש את ה-`__init__` של Teacher. כיוון שלא הגדרנו לו `__init__`, פיתון הולך אל המחלקה מהמנה Teacher יורש, כלומר Person, ומפעיל את ה-`__init__` של Person. בהתאם של Person נקבעים מחדש בירית מחדל ו-`teacher1` יורש אותם. נזכיר שגם ל-`Teacher` כן היה `__init__`, לא היה מופעל אוטומטית ה-`__init__` של המחלקה מהמנה הוא ירש. לגבי פקודת הדפסה, למרות שלא הגדרנו שום מетодה ל-`Teacher`, הוא יורש את `__str__` מ-`Person` ולכן יודפס:

Person Tal is 20 years old

כפי שהחלטנו, צריך שלכל מורה יהיה שכר. לכן נוסף לו מתודה אתחול. המתודה צריכה לגרום לכך שם שאפשר נירש מ-`Person` ומה שטיפצפי למורה, נגידר. 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`. במתודה ה-`__init__` הפעלנו את מתודה האתחול של `Person` עם פרמטרים ששלחנו לה (`name, age`). כיוון ש-`Person` לא יודעת מה לעשות עם פרמטר ה-`salary`, הגדרנו לו-`Teacher` משתנה נוספת, משותה זה הוא כל מה שבדיל כרגע בין `Person` ל-`Teacher`. הגדרנו פשוט ש-`Teacher` הוא `Person` עם שכר.

אם אנחנו רוצים לקרוא למתודות של המחלקה ממנה ירשנו, ניתן להשתמש ב-`super`. פונקציה זו מוצאת את המחלקה ממנה ירשנו. הקוד הבא קורא לפונקציה 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 למתודה 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 שמבוצעים `__init__` של כל המחלקות מכאן ירשנו.

מה יודפס אם נrint 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 – ירושה



צרו את המחלקה Student שירושת מ-`Person`. עליכם:

- לקבוע ערכי ברירת מחדל כרצונכם
- להוסיף לאתחול משתנה נסתר שישמר את ממוצע הציונים
- הוסיפו לממוצע הציונים מתודות `mutator` ו-`accessor`

פולימורפיזם

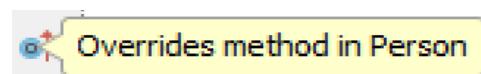
הסיבה שהגדכנו ל-`Person` מתודה בשם `say` היא לבדוק לטובות חלק זה. נפעיל את `say` על אובייקט המורה שלנו, ונקבל את המחרוזת "Hi". אבל נרצה שהמורה יאמר משהו ייחודי, לדוגמה, "Good morning cyber students!", וכמו כן שיירש את כל התכונות של `Person`.

כדי לגרום למethode `say` לעבוד כמו שאנו רוצים, נגידר אותה מחדש בתוך `:Teacher`:



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

שימוש לב לסימן העיגול הכהול לצד המתוודה. אם נעמוד עליו עם העכבר, נקבל הסבר:



במילים אחרות – דרסנו את המתוודה say של Person.



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

אנו יכולים להגיד מהירות נוספת מ-`Person`, לדוגמה `teacher`. כמו של-`Student` יש מתודה say ייחודית, גם ל-`Student` אפשר להגיד say. כתה אפשר להבין מדוע קוראים לפעולה זו "פולימורפיזם". "Poly" משמעתו "הרביה" ואילו "morph" משמעתו "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 רוצחים subclass יפה למשתנים מסוימים של superclass שלו. חשוב לדעת שימושים הינם מסוימים גם מהירושים. לכן, אם אנחנו רוצחים לשימוש ב-members של superclass, ניתן לבחור לעשות זאת באמצעות методות accessor (כזכור יש דיון נרחב בשאלת האם אכן רצוי לעשות זאת). לדוגמה, נעדכן את מתודה str של Teacher (כזכור יש דיון נרחב בשאלת האם אכן רצוי לעשות זאת).

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

הפונקציה isinstance

בבית הספר המעליה של נווה חמציצים נפתחה מגמת סייבר. נגידר תלמיד שולמד במגמה בתור CyberStudent, ככלומר אובייקט שירש מ-`Student` אך כזה 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. הגדרנו subclass של Student. אך כעת, מנהל בית הספר מבקש שכל תלמיד שהציוון שלו בסיביר טוב יקבל הודעה "Wow!".

מתכנתה הכנס לרשימה בשם students את האובייקטים של כל התלמידים, גם אלו שנמצאים במגמת סייבר וגם אלו שאינם. לאחר מכן המתכנת כתוב את הקוד הבא. האם הקוד יעבד באופן תקין?

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

תשובה:

כאשר נרים את הקוד, נקבל שגיאת הרצה -

AttributeError: Student instance has no attribute 'get_cyber_grade'

הסיבה היא שאנו לא-`CyberStudent` מושג `get_cyber_grade`, אבל ל-`Student` אין מתודה כזו. לכן, האובייקט הראשון מסוג `Student` שנמצא ברשימה יגרום לשגיאת ההרצה הנ"ל. על מנת לפתור את הבעיה, הפונקציה `isinstance` מוגיעה לעזרתנו. הפונקציה מקבלת שם של אובייקט ושם של מחלקה, ובודקת אם האובייקט שייך למחלקה או יורש מהמחלקה זו. אם כן – מוחזר `True`, אחרת – `False`. חשוב לציין 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 השלישית תחזיר `True` מכיוון ש-`noam` הוא `Student`, כלומר `noam` הוא subclass של `Person`.

הבדיקה הרביעית תחזיר `False`, מכיוון 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 (เครดיט: שי סDOBסקי)



הציגו מחלקה בשם `BigThing`, אשר מקבלת כפרמטר בזמן היצירה משתנה כלשהו (המשתנה יכול להיות כל דבר – מחרוזת, רשימה, מספר וכו'). למחלקה יש Method בשם `size`, אשר עובדת כך:

- אם המשתנה הוא מספר – המתודה מחזירה את המספר

- אם המשתנה הוא רשימה / מילון / מחרוזת – המתודה מחזירה את `len` של המשתנה

לדוגמה, עבור ההגדלה:

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

התוצאה של `size(my_thing)` הייתה 5, כאשר המחרוזת 'table'.

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

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

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

- אחרת יחזיר "OK"

לדוגמה, עבור ההגדרה:

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

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

פרק 11 – OOP מתקדם (תכנות משחקים באמצעות PyGame)



בחלק זה נלמד לכתוב משחקים באמצעות מודול PyGame של פיתון.

בשלב הראשון נעשו שימוש בפונקציות בסיסיות של PyGame – ניצור מסך עם גרפייה נעה, אשר מגיבה למקלדת ולעכבר ומשמיע צלילים. בשלב השני, בשלב OOP (תכנות מונחה עצמים), על מנת ליצור משחקים המבוססים על שכפול של עצמים ובדיקה אם שני עצמים נוגעים זה זהה (לדוגמה משחקי יריות, משחקי כדור ומשחקי מבוכים).

תוכן העניינים של פרק זה הינו כדלקמן:

- א. כתיבת שדר של PyGame
- ב. הוספת תמונה רקע
- ג. הוספת צורות גאומטריות
- ד. הזות צורות על המסך
- ה. הוספת דמויות Sprites
- ו. קבלת קלט מהעכבר
- ז. קבלת קלט מהמקלדת
- ח. השמעת צלילים
- ט. שילוב OOP
- י. בדיקת התנגשויות בין עצמים

כhibaת שלד של PyGame

```

1 import pygame
2
3 # Constants
4 WINDOW_WIDTH = 700
5 WINDOW_HEIGHT = 500
6
7 # Init screen
8 pygame.init()
9 size = (WINDOW_WIDTH, WINDOW_HEIGHT)
10 screen = pygame.display.set_mode(size)
11 pygame.display.set_caption("Game")
12
13 pygame.quit()

```

ראשית עליינו לבצע import לモול PyGame.

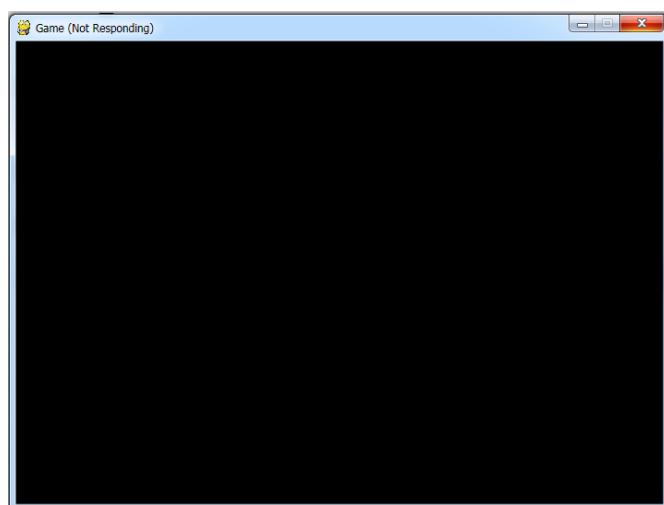
נגידו בטור קבועים את גודל החלון – כמות הפיקסלים לרוחב ולגובה החלון.

לאחר מכן נאחל את PyGame על ידי קרייה לפונקציה init, כך שנוכל להתחיל להשתמש בפונקציות שונות שלו.

יצור מסך בגודל שקבענו ונקבע לו את השם "Game".

מיד לאחר מכן תבצע הפוקודה () .PyGame.quit()

כאשר נרין את התוכנית, יופיע לרגע קצר מסך של PyGame ואז המסך ייסגר.



אם נרצה שהמסך יישאר, נכנס לפניה שורה 13 לולאה אינסופית, אך שימו לב שכרגע – באופן זמני – הדרך היחידה לסגור את התוכנית היא באמצעות לחיצה העצור של PyCharm או באמצעות ניהול המשימות. לחץ סגירת החלון שבקצתה העליון של המסך עדין אינו עובד. מיד נסדר את זה ☺

נרצה לגרום למצב בו לחיצה על כפתור הסירה של מסך המשחק שלנו מבצעת את מה שהיא אמורה לעשות, וסוגרת את המסך. לשם כך, עלינו לחתת לתוכנית שלנו הוראה מה לעשות במקרה שהיראה שיש לחיצה על כפתור הסירה:

```
13 finish = False
14 while not finish:
15     for event in pygame.event.get():
16         if event.type == pygame.QUIT:
17             finish = True
```

כל פעולה שהמשתמש מבצע במסך המשחק מגיעה אל רשיימה, וכל פעולה כזו מוצמד סוג של אירוע – event.type – להזמנה, לחיצה על כפתור סגירת המסך היא סוג אירוע בשם PyGame.event.get(). המתוודה PyGame.QUIT מספקת לנו את הרשימה של כל הפעולות שהמשתמש ביצע מאז הפעם האחרון שקרהנו לה. הקוד לעיל עובר על כל הפעולות שהמשתמש ביצע ובודק אם אחת מהן היא PyGame.QUIT. אם כן – הקוד י יצא מהלולאה האינסופית.

שינויי רקע

הרקע השחור אמן נחמד, אך לעיתים נרצה לצבעו את המסך שלנוצבע אחר, לדוגמה בצביע לבן.

נצטרך להגדיר את הצביע הלבן בתורת tuple של 3 מספרים -

WHITE = (255, 255, 255)

אלו ערכי RGB (Red, Green, Blue) של הצביע הלבן. כל צבע מוגדר על ידי שלישית מספרים שונים. דוגמה:

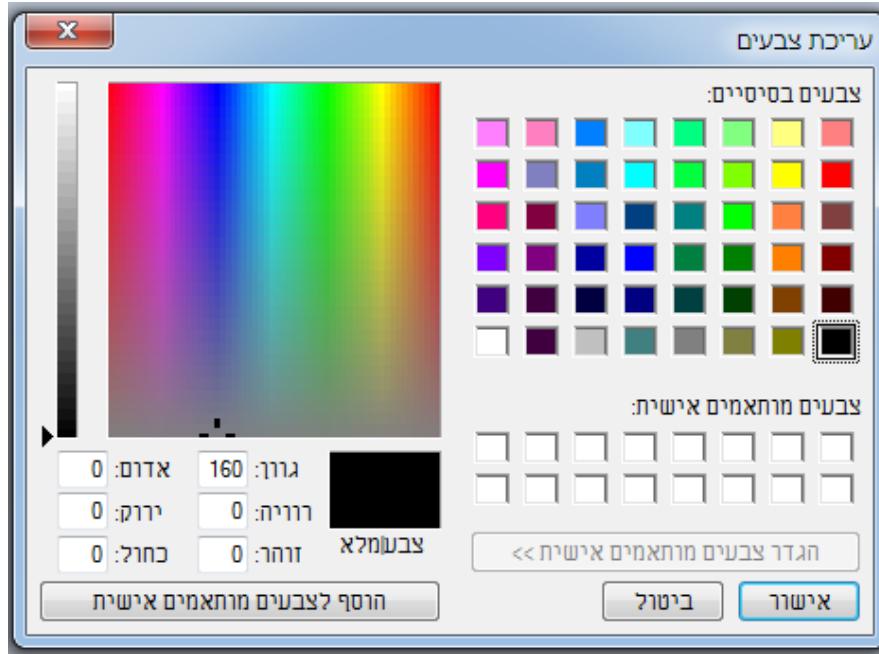
RED = (255, 0, 0)

וכמובן:

BLACK = (0, 0, 0)

אפשר למצוא את שלישית ה-RGB של כל צבע שתרצו באמצעות פונקציית `paint` (צייר) הבסיסית שמאפשרה עם Windows,

תחת תפריט "עריכת צבעים":



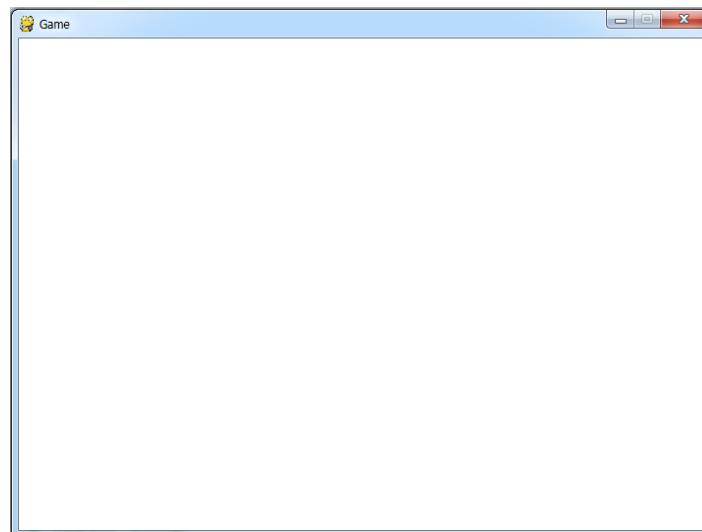
כעת נגידר בתוכנית שהצבע של המסך שלנו הוא לבן. לפני הכניסה לולאה, נוסיף את השורות 15 ו-16:

```

14 # Fill screen and show
15 screen.fill(WHITE)
16 pygame.display.flip()
17
18 finish = False

```

והתוצאה:

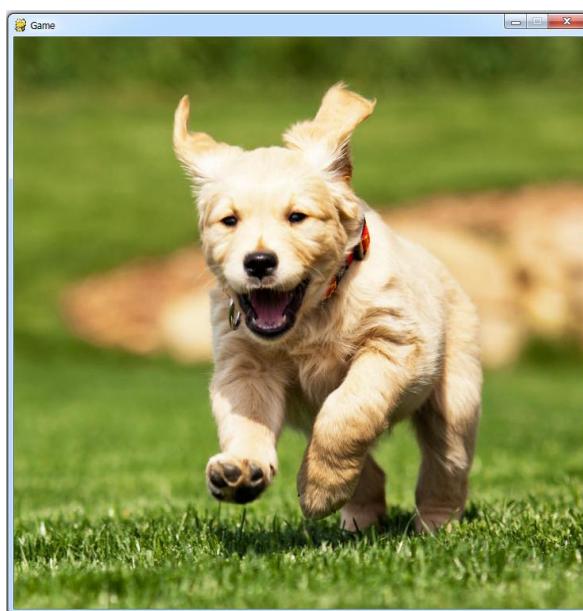


מה מבצעת שורה 16 ?

מסך המחשב קורא את המצב של כל פיקסל ממוקם מוגדר בזיכרון המחשב (שנקרא "video memory"). בשורה 15 אנחנו עדין לא משנים את המצב של הזיכרון ממנו המסך מציג, אלא רק את האובייקט screen. בשורה 16, אנו מורים לתוכנית שלנו לבצע "flip", כלומר להחליף את המידע ב-video memory במידע ששמרנו בתוך האובייקט screen. רק ביצוע ה-flip הוא משנה בפועל את מצב המסך שלנו.

במילים אחרות, אנחנו יכולים לעירר את האובייקט screen כפי רצוננו, אך ללא flip כל השינויים שנבצע יהיו נסתרים למשתמש ולא יוצגו על המסך.

רקע לבן הוא משעם קצר, בואו נעלם תמונה רקע!



ראשית נבחר תמונה כרצוננו. שימו לב לגודל התמונה בפיקסלים. אפשר למצוא את הגודל באמצעות ה-properties של הקובץ. במקרה זה, הגודל הוא 720 על 720 פיקסלים.

<p>example.jpg</p> <p>תמונה מסוג JPEG</p> <p>תאריך צילום: ציון תאריך צילום</p> <p>תגים: חסף-tag</p> <p>דירוג: ★★★★☆</p> <p>מידדים: 720 x 720</p> <p>גודל: 69.6 KB</p> <p>סוטרת: חסף סוטרת</p> <p>מחברים: חסף מחבר</p>	A photograph of a golden retriever puppy running towards the camera on a grassy field. The puppy is light brown with a red collar and has its mouth open as if barking or panting. The background is blurred green grass and some brown dirt.
--	---

שנו את הקבועים שנמצאים בראש התוכנית – גובה ורוחב החלון – על מנת להתאים אותו לגודל התמונה שבחורתם. לאחר ששיםינו את גודל המסך כדי שייתאים לתמונה, נטען אותה כתמונה רקע.

ראשית נגדיר קבוע בשם IMAGE שיכיל את שם הקובץ שלנו, לדוגמה:

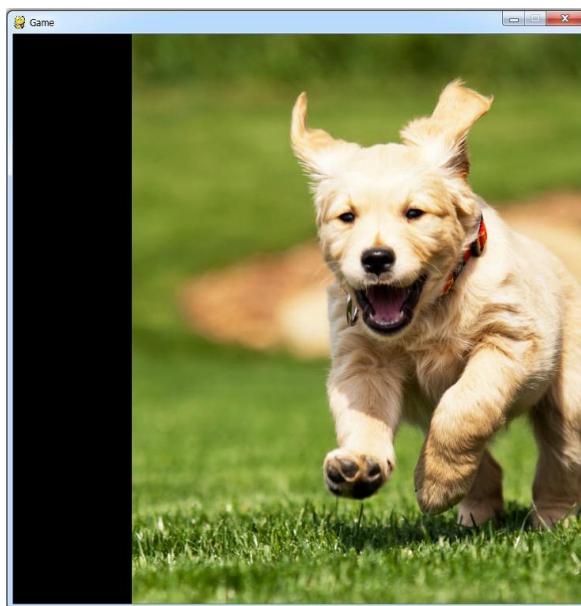
```
IMAGE = 'example.jpg'
```

וכעת נטען את התמונה כתמונה רקע:

```
15 # Fill screen and show
16 img = pygame.image.load(IMAGE)
17 screen.blit(img, (0, 0))
18 pygame.display.flip()
```

שורה 17 טעונה את התמונה בתוך האובייקט screen, החל ממיקום (0, 0). זהה הפינה השמאלית העליונה של המסך. כיוון שדאגנו מראש שגודל המסך יהיה שווה לגודל התמונה, תמונה הרקע תופסת כעת את כל גודל המסך.

シומו לב מה היה קורה אילו היינו מנוטים לטעון את התמונה ממקום התחלתי (0, 150):



כלומר ככל שאנחנו עולמים בפרמטר הראשון אנחנו זרים ימינה על המסך. ככל שאנחנו עולמים בפרמטר השני אנחנו זרים למיטה. אם הגדרנו מסך בגודל 720 על 720, היפיקסל הימני התחתון הוא במספר (719, 719).

הוספה צורות

כעת נוסיף כמה צורות על הרקע שלנו. ספרייה PyGame מאפשרת לנו לצייר קוים, עיגולים, מלבנים, אליפסות וקשתות.

נתחיל בכך שנצייר קו. לשם כך, נשתמש במתודה `PyGame.draw.line`. מתודה זו מקבלת בתור פרמטרים:

- משטח – אובייקט של PyGame עליו י הצייר הקו

- צבע הקו

- נקודת התחלה

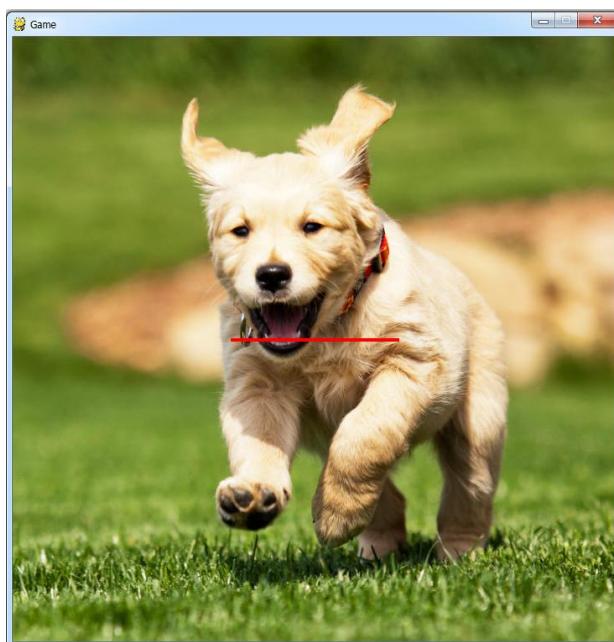
- נקודת סיום

- עובי הקו (ברירת מחדל – 1)

כך:

```
PyGame.draw.line(surface, color, start_pos, end_pos, width=1)
```

כלומר, כדי לצייר קו אדום מנקודה [260, 360] אל נקודה [460, 360] ובעובי 4:

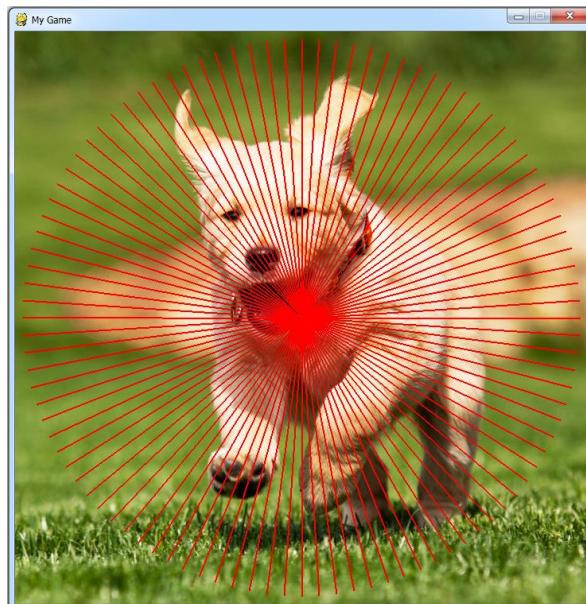


```
19 pygame.draw.line(screen, RED, [260, 360], [460, 360], 4)
20 pygame.display.flip()
```

תרגיל



צרו בעצמכם את הדוגמה הבאה, המורכבה מ-100 קוויים באורך 350 היוצאים ממרכז התמונה. טיפ: השתמשו ב-`math.cos` כדי לחשב את נקודת הסיום של כל קו, בעזרת הfonקציות `cos` ו-`sin`.

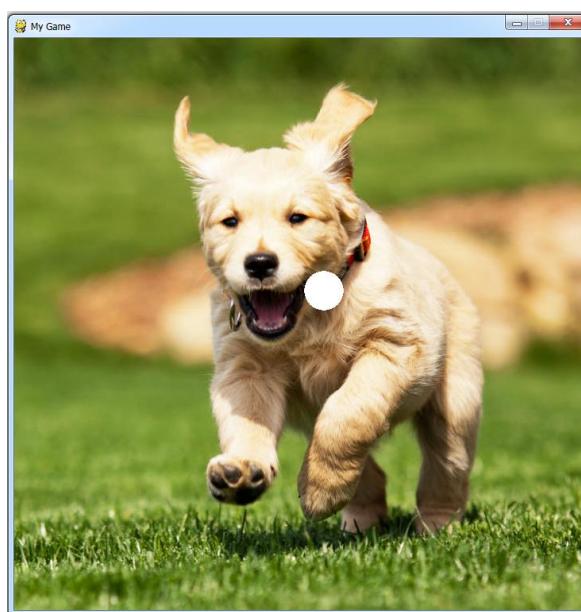


תרגיל



באמצעות התיעוד שנמצא בקישור הבא, ציררו עיגול במקום כלשהו על המסך.

<https://www.PyGame.org/docs/ref/draw.html>



תזוזה של גרפיקה

עד עכשיו ציירנו על המסך צורות שונות, מיד נלמד איך להזיז אותן. איך אפשר לגורום לצופה לחוש שצורה זוה על המסך?

מציריים צורה במקום מסוים. לאחר זמן מה מוחקם אותה, ומציריים אותה שוב במקום אחר, ליד המקום הקודם. כך נדמה לצופה שהצורה "זהה".

נתיחה לשלבים השונים:

מיהיקת צורה וצייר שלה מחדש: כל מה שעשינו לעשות כדי למחוק צורה, הוא פשוט לצייר את תמונה הרקע מעיליה. להזכירם, לצייר צורה מעל תמונה הרקע כבר לדנו.

המתנה של פרק זמן מוגדר בין המיהיקה לצייר: על מנת לעשות זאת נדרש לשעון (טימר), שיתזמין את פרק הזמן בו יש לחזור על פעולות המיהיקה והצייר מחדש.

כדי להוסיף שעון, נגיד:

```
clock = pygame.time.Clock()
```

נוסיף לקבועים שלנו את:

```
REFRESH_RATE = 60
```

כלומר, המסך מתעדכן 60 פעמים בשניה.

בנייה מחדש של הלולאה המרכזית של התוכנית:

```

30 while not finish:
31     for event in pygame.event.get():
32         if event.type == pygame.QUIT:
33             finish = True
34
35     screen.blit(img, (0, 0))
36     # update ball's position
37     ball_x_pos += 1
38     ball_y_pos += 1
39
40     pygame.draw.circle(screen, WHITE, [ball_x_pos, ball_y_pos], RADIUS)
41     pygame.display.flip()
42     clock.tick(REFRESH_RATE)

```

הכנסנו את שורה 35 לתוך הלולאה מכיוון שכעת אנחנו צריכים לצייר את הרקע בכל פעם מחדש (וזאת על מנת למחוק את הזרה שאנו מונינים להזיז).

במוקום שורות 37 ו-38 צריך Kbota קוד כלשהו שמיוזן את הזרה שלנו. לדוגמה, אפשר להזיז מעט את העיגול שציירנו בתרגיל הקודם. בדוגמה הקוד לעיל, אנו מזיזים את העיגול בפיקסל אחד למטה ובפיקסל אחד ימינה בכל איטרציה של הלולאה, רק כדי להמחיש את העקרון. שימו לב שאנו לא בודקים שמייקם העיגול הוא הגיוני (לדוגמה, שהעיגול נמצא בתחום המסך...).

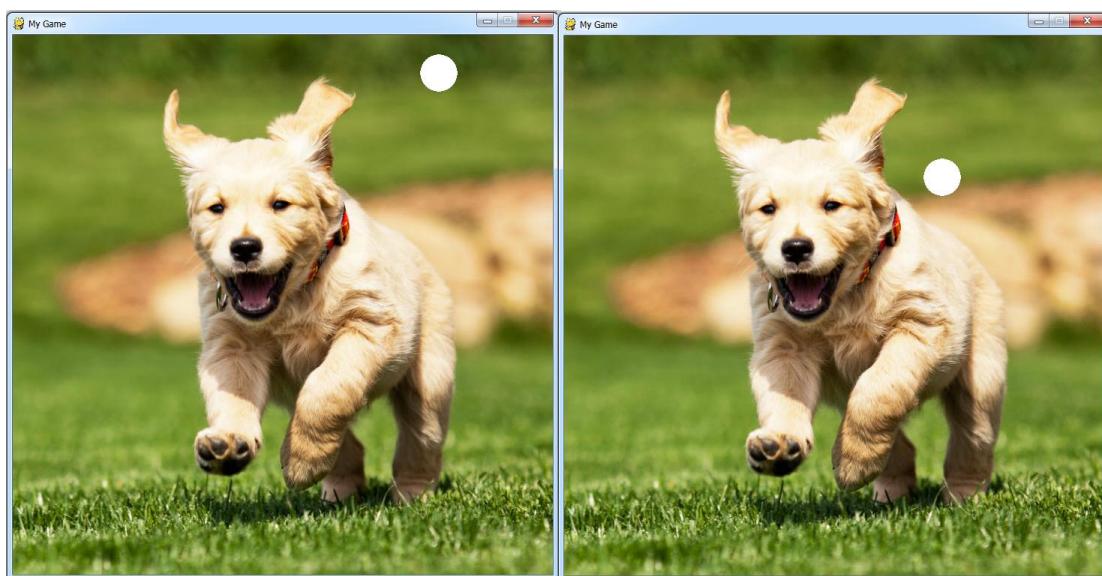
בשורה 40 אנחנו מציירים את העיגול במיקומו החדש.

שורה 41 גורמת לעדכון זיכרונו הויידאו של המסך, כפי שראינו קודם.

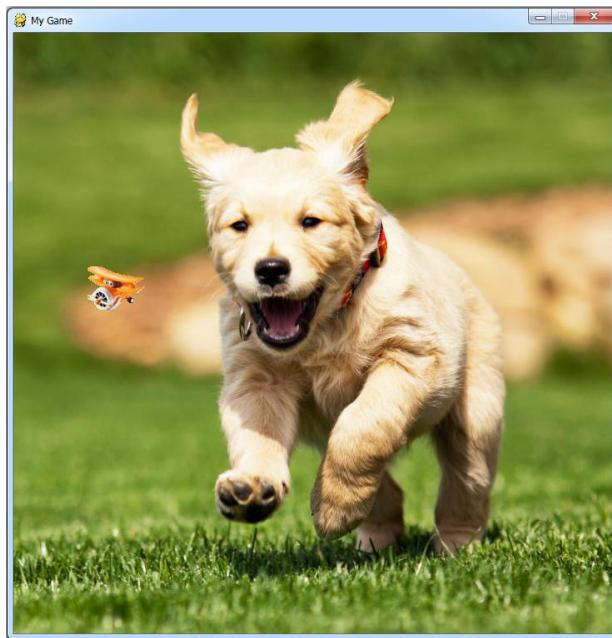
שורה 42 מגדירה לשעון שלנו להמתין פרק זמן לפני האיטרציה הבאה של לולאת ה-while שבשורה 30. כאמור, בלי המתנה זו, לא נוכל לשלוט במהירות ההתקומות של הגרפיקה שלנו על המסך.

תרגיל – פינג פונג

scalillo את התרגיל בו ציירתם עיגול על המסך, כך שהעיגול יתקדם על המסך מצד לצד. במידה והעיגול נוגע בשולי המסך, שנו את וקטור המהוות שלו כאילו שהוא כדור שניitz מושטה.



Sprite ציור



סיימנו להשתעשע עם צורות גאומטריות. כעת נניח על תמונה הרקע שלנו תמונה שחkn קטנה, שנראית Sprite, ובזיז אותה למקום.

בתוך תמונה שחkn נבחר קובץ תמונה קטן:



שימוש לב שהמוטס נמצא על רקע אחד. מיד נבין את החשיבות שכך.

כדי ליצור תמונה עם רקע אחד נפתח את תוכנת הצייר, paint, ונעתיק לתוכה כל תמונה שנרצה. כעת נגדיר צבע רקע באמצעות תפריט "עריכת צבעים" אותו הכרנו כבר. חשוב להגדיר צבע שאינו נמצא בתמונה שבחרנו. במקרה זה בחרנו את הצבע הורוד שה-RGB שלו הוא (255, 20, 147).

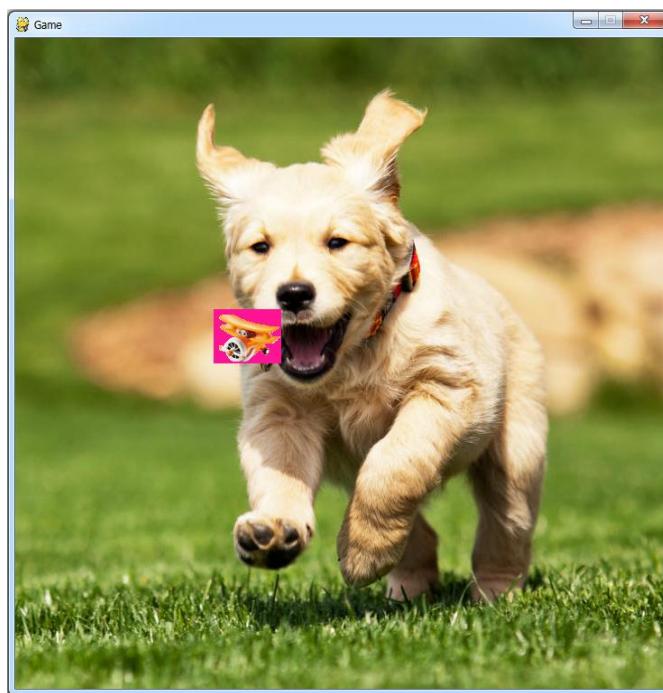
לאחר שסימנו, מומלץ לשמר את התמונה בפורמט `png`. זהו פורמט שאינו מעוות את הצלבים בתמונה וכך הצלב שהכרנו בהור רקע יישמר כמו שהוא.

בשלב הבא נטען את ה-`Sprite` שלנו לתוכנית:

```
17 img = pygame.image.load(IMAGE)
18 screen.blit(img, (0, 0))
19 player_image = pygame.image.load('plane.png').convert()
20 screen.blit(player_image, [220, 300])
21 pygame.display.flip()
```

השורות שנוספו הינן 19 ו-20. בשורה 19 אנחנו טוענים את קובץ התמונה שלנו, ובשורה 20 אנחנו קובעים את המיקום שלו על תמונה הרקע.

אם נריץ את התוכנית כעת, נקבל את התוצאה הבאה:



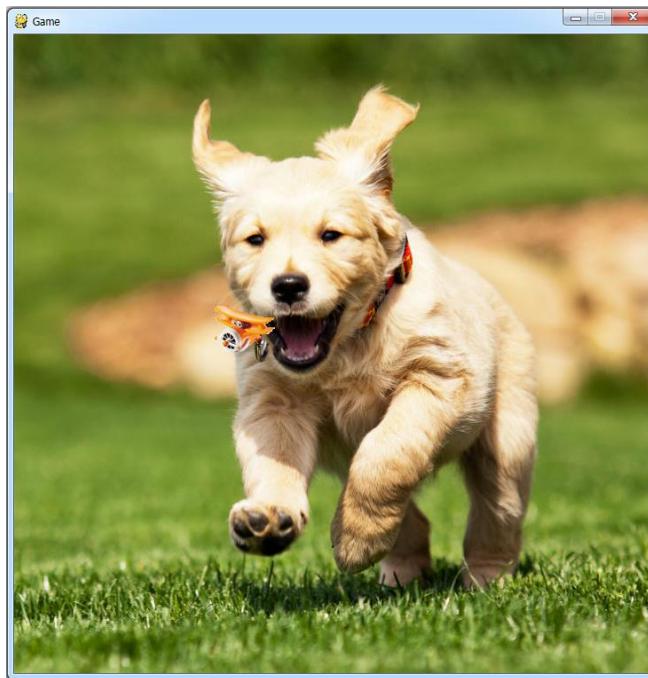
כאן נכנס לשימוש הרקע שהגדכנו ל-`Sprite` שלנו.

בתחילת התוכנית נגדיר קבוע את הצלב שאיתו צבענו את רקע ה-`Sprite` שלנו:

`PINK = (255, 20, 147)`

נגידר לתוכנית שלנו להציגו הצעה זו. כמובן, אם היא מזזה ב-**Sprite** שלנו פיקסל בצבע שהגדנו, היא תתייחס אליו כשקופ ולא תציג אותו מעל תמונה הרקע (שימו לב לשורה 21):

```
20 player_image = pygame.image.load('plane.png').convert()
21 player_image.set_colorkey(PINK)
22 screen.blit(player_image, [220, 300])
```



נפלא! כעת אנחנו יכולים להעלות כל תמונה מעל תמונה הרקע שלנו ולהזיז אותה.

קבלת קלט מהעכבר

כאשר אנו מקבלים קלט מהעכבר, אנחנו רוצים למשה לעדעת שני דברים:

- מה מיקום העכבר על המסך
- אילו כפתורים לחוצים

לשמהתנו, PyGame מספק לנו את המידע זהה די בקלות.

באמצעות המתודה `PyGame.mouse.get_pos()` אנחנו מקבלים את המיקום הנוכחי של העכבר. מידה שנרצה שה-Sprite ינוע בהתאם למיקום העכבר, כל מה שנותר לנו לעשות הוא להודיע למסך שלנו לצייר את ה-Sprite במיקום של העכבר ואז לבצע `flip` למסך.

```

34 while not finish:
35     for event in pygame.event.get():
36         if event.type == pygame.QUIT:
37             finish = True
38
39     screen.blit(img, (0, 0))
40     mouse_point = pygame.mouse.get_pos()
41     screen.blit(player_image, mouse_point)
42
43     pygame.display.flip()
44     clock.tick(REFRESH_RATE)

```

כפי ששמתם לב, האיקון של העכבר מופיע על ה-Sprite שלנו וזה לא נראה יפה במיוחד. לכן, בתחילת התוכנית נרים את שורה הקוד הבאה, שתעלים את האיקון של העכבר:

`PyGame.mouse.set_visible(False)`

כעת נרצה לבצע פעולות שונות על פי להיזות המשמש על העכבר.

ראשית נדריך את להציגו העכבר. המספרים המשווים לכל לחץן הן בהתאם לערכים של PyGame, אנחנו נתונים להם שמות כדי שהקוד יהיה יותר קרייא:

```

17 LEFT = 1
18 SCROLL = 2
19 RIGHT = 3

```

בכל פעם שהמשמש יקליק על הכפתור השמאלי של העכבר, נשמר את מיקום העכבר ברשימה שאנו נזכיר – `mouse_pos_list`. נוכל להשתמש ברשימה המיקומים של העכבר בשבייל לעשות כל דבר שנרצה.

נבחן את הקוד הבא:

```

34  while not finish:
35      for event in pygame.event.get():
36          if event.type == pygame.QUIT:
37              finish = True
38      elif event.type == pygame.MOUSEBUTTONDOWN \
39          and event.button == LEFT:
40          mouse_pos_list.append(pygame.mouse.get_pos())

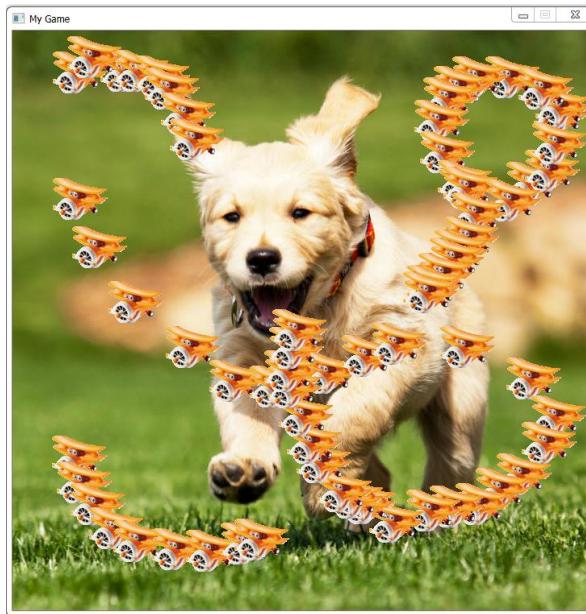
```

התנאי שהגדכנו מתחילה בבדיקה `event.type` – אנו בודקים האם מדובר בלחיצה על כפתור כלשהו בעכבר. שימוש לב שאין צורך להגדיר את הקבוע `MOUSEBUTTONDOWN`, משום שהקבוע זהה מופיע כבר בתחום ה-`event` המוגדרים של PyGame. לאחר שהוא ידנו שהתרחשה לחיצה על כפתור כלשהו בעכבר, אנו בודקים אם היא כפתור בדיקן נלחץ.

תרגיל – עכבר



כתבו תוכנית אשר בכל פעם שהעכבר נלחץ בקлик שמאל, תשאיר על המסך עותק של ה-`Sprite` שלנו. וודאו שניתן להקליק על העכבר ללא הגבלה.



קבלת קלט מהמקלדת

קיבלה קלט מהמקלדת דומה למדוי לקבלת קלט מהעכבר. גם כאן, אנו מחפשים eventים בתוך הרשימה שחוורת מקריאה ל-`PyGame.event.get()`, רק שהפעם נחפש ארועים שקשורים למקלדת.

ראשית, נבדוק שהיתה לחיצה על המקלדת, וזאת באמצעות ה-`event` שנקרא `KEYDOWN`:

```
if even.type == PyGame.KEYDOWN:
```

בנחיה שאכן הייתה לחיצה על המקלדת, נוכל לבדוק האם מקש ספציפי הוקש. כל המקשיים במקלדת מתחילה ב-`K_` ואחריו יש את המKeySpec. לדוגמה, מקש ה-`a` הוא `K_a`. נוכל לבדוק אם מקש ה-`a` הוקש על ידי:

```
if event.key == PyGame.K_a:
```

```
45 while not finish:
46     for event in pygame.event.get():
47         if event.type == pygame.QUIT:
48             finish = True
49         # User pressed a key
50     elif event.type == pygame.KEYDOWN:
51         if event.key == pygame.K_a: # key is 'a'
```

השمعת צלילים

ראשית ניצור קובץ שמכיל את הצליל שאנו רוצים להשמי. רוב הצלילים ניתנים למציאה ב-[youtube](#) ([youtube](#) (אפשר להפץ לדוגמה)). לאחר מכן נוריד את הקובץ ה-mp3 באמצעות אתר כגון:

<http://www.youtube-mp3.org/>

שימוש לבשהורדת מוזיקה שיש עליה זכויות יוצרים באמצעות אתר זה הינה בניגוד לתנאי השימוש של יוטיוב.

<http://www.youtube.com/watch?v=KMU0tzLwhbE>

Convert Video

כדי לחתוך רק קטע מסוים ממתריך קובץ ה-mp3 שהורדנו, השתמש באתר כגון:

<http://mp3cut.net/>



cut יש ברשותנו קובץ mp3 עם הצליל המבוקש.

העלאה והשمعה שלו מתבצעות כה:

```

29     SOUND_FILE = "kaboom.mp3"
30     pygame.mixer.init()
31     pygame.mixer.music.load(SOUND_FILE)
32     pygame.mixer.music.play()
```

תרגיל סיכום מבנים



צרו סקריפט PyGame אשר מבצע את הדברים הבאים:

- העלאה של תמונה רקע

- הוזה של Sprite כלשהו על תמונה הרקע

- אפשר יהיה להזין את ה-Sprite על ידי העבר

- אפשר יהיה להזין את ה-Sprite על ידי המקלדת

- לחיצה על קлик שמאלית בעברית תויר את הסימן של ה-Sprite במקומות שבו על המסך

- לחיצה על מקש ה-space תמחק את כל ה-Sprites שצויירו על הרקע

- לחיצה על המקש הימני של העברת שימוש אפקט קולי כלשהו

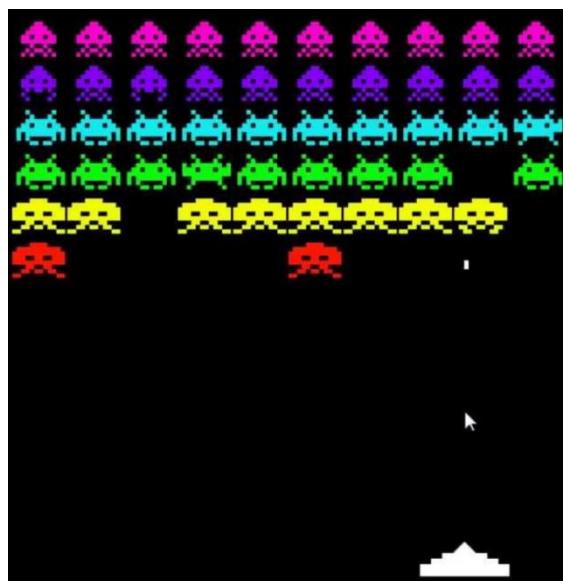
תנו!

OOP מתקדם – שילוב PyGame

מבוא

לאחר שלמדנו לכתוב OOP בפייתון, נשלב את הידע שלנו ב-OOP PyGame ליצירת משחקים מורכבים. מדוע אנחנו זוקים ל-OOP בשביל לכתוב משחקים בפייתון? הלא אנחנו יודעים כבר להעלות למסך כל צורה או Sprite שאנחנו רוצים ולהזין אותו?

נכון, אבל הבעיה היא שהשיטה שהשתמשנו בה עד כה טובה בשביל להזין צורה או שתיים. דמיינו שהמסך שלנו מלא ב-Sprites של Space Invaders שציריך להזין אותם... נצטרך להגדיר כמה שירות Sprites, לכל אחד מהם לטעון תמונה, לקבוע מיקום ולנהל את התנועה שלהם על המסך. אחת הביעות הגודלות שלנו תהיה לתת שמות למשתנים. נניח שנרצה להגדיר מיקום על המסך עבור שני invaders.space invader1 משנתה למיקום x של invaders.space invader2... וזה עוד לפני שהתחלנו לדבר על משתנים invaders1, משנתה למיקום x של invaders2 ועוד אחד למיקום y של invaders2. נראה לנו מיותר לרשום כל מיקום ותנועה שליהם (לא תמיד הם נעים באותה מהירות). לא נעים במיוחד לתכנת משחק כזה ☺



בנקודה זו מגע לזרתנו OOP. הרעיון הכללי הוא שנדריך מחלקה שכוללת את כל התוכנות של האובייקט שאנחנו צריכים, לדוגמה איך נראה invaders.space invader, מה המיקום שלו על המסך ומה מהירותו שלו, ובכל פעם שנרצה להוסיף invader חדש – פשוט ניצור instance נוסף של המחלקה שלנו. קידמה, מתחילה.

הגדרת class

בואו ניתן לכלבלב שלנו כמה כדורים לשחק בהם!

נפתח קובץ חדש, נקרא לו לדוגמה `shapes.py`. מאוחר יותר נעשה לו `import` לתוכנית הראשית שלנו.

נגידר בתוכו `class` בשם `Ball`. חשוב מאוד ש-`Ball` יירש מ-`Sprite`, כך שנוכל לרש את כל המתודות המועילות של `Sprite` מתודות אשר ישמשו אותנו בהמשך. כמובן, לא נשכח לעשות `import PyGame` בתחלת הקובץ. כתע נכתוב את ה-`constructor` שלנו.

```

7   class Ball(pygame.sprite.Sprite):
8
9       def __init__(self, x, y):
10          super(Ball, self).__init__()
11          self.image = pygame.image.load(MOVING_IMAGE).convert()
12          self.image.set_colorkey(PINK)
13          self.rect = self.image.get_rect()
14          self.rect.x = x
15          self.rect.y = y
16          self.__vx = HORIZONTAL_VELOCITY
17          self.__vy = VERTICAL_VELOCITY

```

בשורה 10, הדבר הראשון שאנו צריכים לבצע הוא להריץ את פונקציית `__init__` האתחלול של המחלקה ממנה `Ball` יורש.

שורות 11 ו-12 מוכנות לנו כבר, טעינת הציור.

בשורה 13 מוגדר משתנה בשם `rect` ששייך לאובייקט שלנו. המשתנה זה שומר את המיקום של הכדור שלנו על המסך, והוא מאוד חשוב בשביל לדעת מה הכדור שלנו עושה. לדוגמה, אולי אובייקטים אחרים הוא מתנגש. שימו לב שלמרות שאנו מציירים על המסך עיגול, הרוי `rect` הוא מרובע. זכרו שבעינו התוכנית התמונה שלנו אינה עיגול, אלא מרובע שביקשנו להציג צבע אחד ממנו בתור "שחור". לתוכנית אין דרך לדעת שאחרי שהציגנו את הצבע הורוד בטור שקור, נותרה צורה של עיגול. הקואורדינטות של `rect` הן הפינה השמאלית העליונה של הריבוע שלנו (כלומר הריבוע השלם, כולל החלקים השקופים), והגודל של `rect` הוא הגודל של הריבוע.

בשורות 14,15 נגידר שה-`Sprite` שלנו מקבל מיקום התחלתי בזמן הייצור שלו. משתמש בו מאוחר יותר לטובת הציור על המסך.

נותר לנו רק לקבוע את מהירותו של הכדור. שימו לב לכך שהמשתנים של מהירותם הם מוסתרים – מתחילהם ב-`__`. הינו יכולים לקבוע מהירות רנדומלית או לקבל את מהירות כפרמטר, נשאיר זאת להחלטתכם.

הוספה מתודות mutators ו-accessors שימושיות

בואו נזין את הcador שלנו על המסך.

```
def update_v(self, vx, vy):
    self.__vx = vx
    self.__vy = vy

def update_loc(self):
    self.rect.x += self.__vx
    self.rect.y += self.__vy

def get_pos(self):
    return self.rect.x, self.rect.y

def get_v(self):
    return self.__vx, self.__vy
```

יצרנו מספר מתודות שתפקידן למצוא את מיקום הcador בכל פעם שנרצה. מיקום הcador החדש ייחסב בהתאם המיקום הקודם של הcador, בתוספת מהירות הcador בכל אחד מהציריים.

זהו, התבנית של הcador שלנו מוכנה וכעת נוכל למסור לכלבלב שלנו כמה כדורים שנרצה.

הגדרת אובייקטים בתוכנית הראשית

בתוכנית הראשית נדרש לעשות import ל-shapes שיצרנו. כדי שלא נצטרך לציין שהצורה נלקחה מהמודול shapes בכל פעם, נעשה import באופן הבא:

```
from shapes import Ball
```

נדיר שני כדורים ונציג אותם על המסך:

```

26     ball1 = Ball(100, 100)
27     ball2 = Ball(200, 200)
28     screen.blit(ball1.image, ball1.get_pos())
29     screen.blit(ball2.image, ball2.get_pos())
30     finish = False
31
32     while not finish:
33         for event in pygame.event.get():
34             if event.type == pygame.QUIT:
35                 finish = True
36         clock.tick(REFRESH_RATE)
37         pygame.display.flip()

```

בשורות 26 ו-27 אנחנו יוצרים שני כדורים, לכל אחד מהם יש מקום התחלתי שאנו מביברים ל-.constructor. מקום התחלתי זה יועתק שם ל-.rect member של המחלקה Ball.

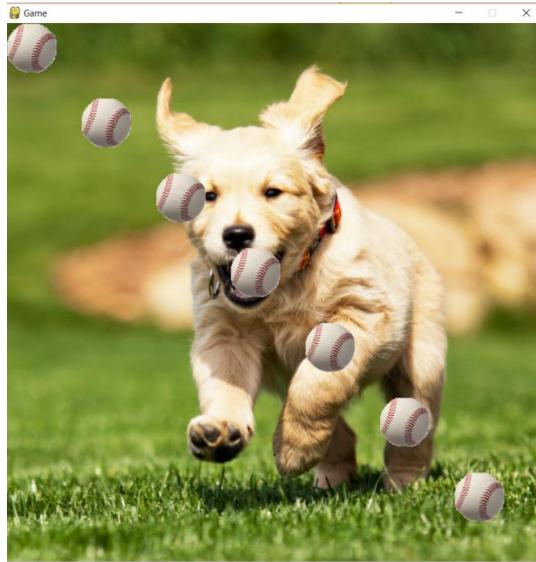
בשורות 28, 29 אנחנו מביברים למסך את הcadors. המתודה blit מקבלת כזוכה את התמונה שאנו רוצים להעלות ואות המיקום. התמונה היא ball.image (מעבר כל כדור), ואל המיקום אנחנו ניגשים בעזרת המתודה get_pos שיצרנו לשם כך.

שורות 30 והלאה הן הקוד הבסיסי המוכר לנו. נרץ ונקבל כדורים במקומות המתאימים על המסך:



sprite.Group()

עד כה הצלחנו להציג כדורים על המסך כרצוננו, מצוין. עם זאת, הדרך בהעשינו זאת מעט מסורבלת. דמיינו שאנחנו רוצחים לצייר שבעה כדורים ולא שניים, כמו בדוגמה הבאה:



האם הגיוני שבשביל להציג כמהות גדולה של כדורים נצטרך לכתוב שוב ושוב שורות כמו 23 ו-24? לא. האם אין דרך יותר פשוטה להציג את כל ה כדורים שלנו, ולא לכתוב שורת קוד נפרצת לכל כדור?... בוודאי שיש.

```

26     balls_list = pygame.sprite.Group()
27     for i in xrange(NUMBER_OF_BALLS):
28         ball = Ball(i*DISTANCE, i*DISTANCE)
29         balls_list.add(ball)
30
31     balls_list.draw(screen)
  
```

כדי לעבור על כל ה כדורים בבת אחת אנחנו צריכים לולאת `for`, ולולאת ה-`for` צריכה לעבור על משהו שהוא `iterable`. והוא `sprite.Group()`. אבל למה אנחנו מגדירים (`sprite.Group()`) סתם רשימה ריקה, []? בגלל שלרשימה מסוג () יש כל מיני מוגדרות שימושיות, הקשורות לנו לעבודה. בקרוב נכיר שתים מהן:

- מתודה שמדפיסה למסך בבת אחת את כל האובייקטים שברשימה

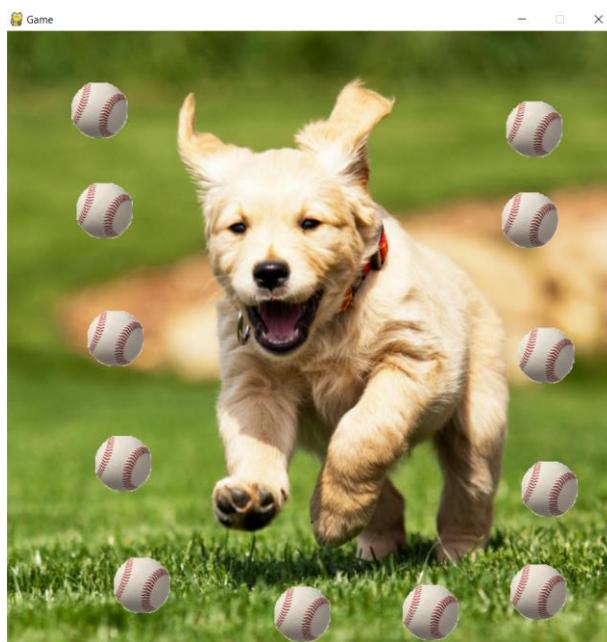
- מתודה שבודקת אם יש התנגשויות בין אובייקטים (נמצאים באותו מקום)

בכל איטרציה של הלולאה אנחנו מגדירים כדור חדש ולאחר מכן מוסיפים אותו לרשימה שלנו באמצעות Method `add`. נפלא, יש לנו כתה רשימה הכוללת שבעה כדורים. שימו לב לכך שהעובדת שאנו עושים משתמש בכל איטרציה באותו משתנה בשם `ball` אינה גורמת למחיקת הקיימים הקודמים: כל משתנה בשם `ball` הוא מצביע על אובייקט מסווג `Ball`. בכל פעם שהתוכנית מגיעה לשורה 28, נוצר אובייקט חדש מסוג `Ball` והמשתנה `ball` מצביע עליו. לעומת זאת רק המצביע המשתנה – ולא האובייקט עצמו. נוסף על כן, בשורה 29 המצביע על האובייקט שנוצר נשמר ברשימה, ולכן מוגלים לגשת אליו גם אחרי `ball` כבר מצביע על האובייקט הבא.

בשורה 31 אנחנו קוראים למתחודה שחווסכת לנו המון עובדה: `draw()`. המתחודה הזה פועלת על `sprite.Group` ומאפשרת להעיבר למסך בבת אחת את כל האובייקטים שיש ברשימה. פעולה זו שולחה לרוץ בלולאת `for` ולהדפיס כל אובייקט למסך באופן נפרד.

יצירת אובייקטים חדשים

כעת נרצה שכמות האובייקטים שלנו תהיה ניתנת לקביעה על ידי המשתמש. לדוגמה – שכל הקלקה על לחץ שמאלית בעברית תיצור כדור חדש על המסך.



נזכיר בכך שבודק אם הייתה הקלקה על הלחץ השמאלי של העכבר – שורה 30 (המשךה לתוך שורה 31), ומה היה מיקום העכבר – שורה 32:

```

26 while not finish:
27     for event in pygame.event.get():
28         if event.type == pygame.QUIT:
29             finish = True
30         elif event.type == pygame.MOUSEBUTTONDOWN \
31             and event.button == LEFT:
32             x, y = pygame.mouse.get_pos()
33             ball = Ball(x, y)
34             balls_list.add(ball)
35
36         screen.blit(img, (0, 0))
37         balls_list.draw(screen)
38         pygame.display.flip()
39         clock.tick(REFRESH_RATE)

```

לאחר שמצאנו את מיקום העכבר אנחנו מגדירים `ball` חדש במקומו זה ומוסיפים אותו לרשימה. את יתר הפקודות כבר הכרנו לפני הדפסת הרקע (36), הדפסת כל הcadורים (37), עדכון המסך (38) וקציבת זמן עדכון התוכנית (39).

הזוזת האובייקטים

אובייקטים זזים יותר מעוניינים מאובייקטים קבועים במקום. בואו ניתן לכל אובייקט מהירות ההתחלתית אקראית (חשוב לעשوت `:import random`

```

30         elif event.type == pygame.MOUSEBUTTONDOWN \
31             and event.button == LEFT:
32             x, y = pygame.mouse.get_pos()
33             ball = Ball(x, y)
34             vx = random.randint(-3, 3)
35             vy = random.randint(-3, 3)
36             ball.update_v(vx, vy)
37             balls_list.add(ball)

```

בשורות 36–34 אנחנו מוגבלים מהירות ההתחלתית וمعدכנים אותה בהתאם cadור בהתאם. בכל פעם שנריץ את המתוודה `ball.update_loc()` יעדכנו ערכי `rect.x` ו- `rect.y` שלcador, כפי שקבענו בהתחלה:

```

39 for ball in balls_list:
40     ball.update_loc()

```

זהו,cadors שלנו נעים על המסלול ☺



הבעיה במצב הנוכחי היא, שגם כאשר הבדורים מגיעים לקצה המסך הם ממשיכים לנוע ויזרים ממנו. לכן הוסיפו קוד, שבודק אם הבדור שלנו נוגע בשולי המסך ואם כן – מעדכן את מהירותו שלו כך שהבדור " קופץ" חזרה. טיפ: קפיצה הבדור חוזרת מתבצעת על ידי היפוך מהירות האופקית שלו (אם פגע בקצה הימני או השמאלי של המסך) או היפוך מהירות האנכית שלו (אם פגע בקצה העליון או התיכון של המסך).

בדיקות התנגשויות

ברוב משחקי המחשב נרצה לדעת אם שני אובייקטים עולמים זה על זה. לדוגמה, אם יריה נוגעת בדמות, או אם הפקמן שלנו נחפס על ידי שدون.

נרצה לשדרג את התוכנית שלנו כך שיהיה שני כדורים שמתנגשים זה בזה – יימחקו. כתת נראה איך אפשר לזהות בקלות התנגשויות בין אובייקטים.

הmethod `spritecollide` משמשת למטרה זו. המתודה מקבלת אובייקט ורשימה של אובייקטים ומוחירה את כל האובייקטים מהרשימה שמתנגשים באובייקט הנבדק. בשורת הקוד הבא אנו בודקים אם אובייקט מסוים בשם `ball` מתנגש בבדורים שמוררים ב-`:balls_list`:

```
balls_hit_list = pygame.sprite.spritecollide(ball, balls_list, False)
```

כיצד המתודה בודקת את ההתנגשויות? באמצעות ה-`rect` של כל אובייקט. כלומר, שני אובייקטים מתנגשים אם ה-`rect`ים שלהם חופפים זה עם זה.

בנוסף, המתודה מקבלת פרמטר בוליאני בשם `doKill` ("האם להרוג"). אם ערך הפרמטר הוא `True`, או כל אובייקט ברשימהת האובייקטים שמתנגש באובייקט הנבדק – יימחק.

במקרה שלנו אנחנו לא רוצים למחוק את האובייקט המתנגש. מדוע? מיד נראה.

```

49     new_balls_list.empty()
50     for ball in balls_list:
51         balls_hit_list = pygame.sprite.spritecollide \
52             (ball, balls_list, False)
53         if len(balls_hit_list) == 1:      # ball collides
54             # only with itself
55             new_balls_list.add(ball)
56
57     balls_list.empty()
58     for ball in new_balls_list:
59         balls_list.add(ball)

```

בשורה 50 אנחנו מדירים שאנו עוכרים על כל ה כדורים שבתו רישימת ה כדורים שלנו. ככלומר אנחנו בודקים בכמה כדורים בתוך הרשימה מתנגש כל כדור ברשימה. ברגע שההתוצאה תהיה תמיד 1, מכיוון שהכדור שבחרנו מהרשימה מתנגש עם עצמו (שכן ה rect שלו חופף זה של עצמו). זו גם הסיבה לכך שאנו לא מוחקים את ה כדור המתנגש מהרשימה, כי אחרת נישאר עם רישימת כדורים ריקה.

בשורה 53 אנחנו בודקים האם ה כדור שלנו רק בכדור אחד ברשימה (כלומר, בעצם בלבד). אם כן, זה אומר שהכדור צריך להשאר על המסך. לכן נשמר אותו ברשימה ה שורדים "שורדים" new_balls_list. שימוש לבן לכך שזהו רישימה מסוג sprite.Group() ושבשורה 49 אנחנו מרוקנים אותה מכדורים, באמצעות המתוודה .empty()

כל שנותר לנו לעשות הוא להחזיר את ה כדורים השורדים לרישימת balls_list לפני שנמשיך ונציג אותם על המסך. לשם כך אנחנו מרוקנים את list balls_list בשורה 57 ולאחר מכן מעתיקים כל כדור מתוך רישימת השורדים אל תוך .balls_list.

שימוש לבן לכךuai אפשר לכתוב פשוט `balls_list = new_balls_list`. מה יקרה לדעתכם במקרה זה? תוכלו לעזור בדוגמה הבאה של שתי רישימות שמצוות על אותו מקום בזיכרון.

```

In[12]: list1 = [1, 2, 3]
In[13]: list2 = list1
In[14]: list1.pop(0)
Out[14]: 1
In[15]: list1.pop(0)
Out[15]: 2
In[16]: list1.pop(0)
Out[16]: 3
In[17]: list2
Out[17]: []

```

לסיום, הנה הקוד המלא של התוכנית שבודקת התנגשויות של כדורים:

```

1 # Pygame example program
2 # Sprite collision detection
3 # Author: Barak Gonen, 2017
4
5 import pygame
6 import random
7 from shapes import Ball
8
9 WIDTH = 720
10 HEIGHT = 720
11 LEFT = 1
12 REFRESH_RATE = 60 # times each second
13 BALL_SIZE = 55 # pixels
14
15 img = pygame.image.load('example.jpg')
16 pygame.init()
17 size = (WIDTH, HEIGHT)
18 screen = pygame.display.set_mode(size)
19 pygame.display.set_caption("Game")
20 screen.blit(img, (0, 0))
21 clock = pygame.time.Clock()
22
23 balls_list = pygame.sprite.Group()
24 new_balls_list = pygame.sprite.Group()
25 finish = False
26 while not finish:
27     for event in pygame.event.get():
28         # quit if window closed
29         if event.type == pygame.QUIT:
30             finish = True
31         # add ball each time user clicks mouse
32         elif event.type == pygame.MOUSEBUTTONDOWN \
33             and event.button == LEFT:
34             x, y = pygame.mouse.get_pos()
35             ball = Ball(x, y)
36             vx = random.randint(-3, 3)
37             vy = random.randint(-3, 3)
38             ball.update_v(vx, vy)
39             balls_list.add(ball)
40
41     # update balls locations, bounce from edges
42     for ball in balls_list:
43         ball.update_loc()
44         x, y = ball.get_pos()
45         vx, vy = ball.get_v()
46         if x + BALL_SIZE > WIDTH or x < 0:

```

```

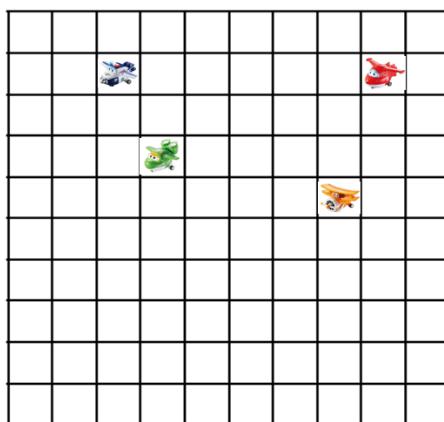
47         vx *= -1
48     if y + BALL_SIZE > HEIGHT or y < 0:
49         vy *= -1
50     ball.update_v(vx, vy)
51
52     # find which balls collide and should be removed
53     new_balls_list.empty()
54     for ball in balls_list:
55         balls_hit_list = pygame.sprite.spritecollide \
56             (ball, balls_list, False)
57         if len(balls_hit_list) == 1:      # ball collides
58             # only with itself
59             new_balls_list.add(ball)
60
61     balls_list.empty()
62     for ball in new_balls_list:
63         balls_list.add(ball)
64
65     # update screen with surviving balls
66     screen.blit(img, (0, 0))
67     balls_list.draw(screen)
68     pygame.display.flip()
69     clock.tick(REFRESH_RATE)
70
71 pygame.quit()

```

תרגיל מסכם – פיקוח אווירי



אתם יושבים במגדל הפיקוח האווירי ואחראים למנוע התנגשות בין מטוסים במרחב האווירי שלכם. המרחב האווירי הוא מטריצה בגודל 10 × 10 משבצות על 10 משבצות (כל משבצת היא בגודל של כמה פיקסלים שתחלito, לדוגמה 50 על 50 פיקסלים).



למרחב האווירי שלכם נכנסו 4 מטוסים מסוג **CrazyPlane** שונים בו אקראיות. בכל תור, כל אחד מהמטוסים יכול לזרז לכל משבצת צמודה למשבצת בה הוא נמצא (כולל אלכסון).

עליכם כתוב אלגוריתם שמנהל את תנועת המטוסים: בכל תור על האלגוריתם לבחור אם מתחת למוטס להמשיך לנעו אקראיות או להוראות לו לאייזו משבצת צמודה הוא צריך לפנות.

ניקוד: המטרה היא שהאלגוריתם שלכם ישלח כמה שפחות הוראות למטוסים. לכן, כל מטוס שנע למשבצת אקראיית מעניק לכם נקודה. כל מטוס שנע למשבצת עקב פוקודה קיבל מכם, מורייד לכם נקודה. המשחק נגמר כאשר שני מטוסים מתנגשים או לאחר 1000 תורות.

חישוב ניקוד לדוגמה:

ארבעת המטוסים שרדו 1000 תורות, כולמר נעו ביחד 4000 משבצות. בסך הכל המטוסים קיבלו 200 פקודות שינוי מיקום, כולמר הם נעו 3800 צעדים אקראיים (3800 נקודות). סך הכל הרוחות $200 - 3800 = 3600$ נקודות.

הצלחתם לגרום למטוסים שלכם לשרוד? יפה מאד! עכשיו שחקו את המשחק עם 10 מטוסים ☺ האם האלגוריתם שלכם עוזין עובד היטב? האם תוכלו לשפר אותו?

סיכום

בפרק זה למדנו לתוכנת מושקי מחשב באמצעות PyGame. ראיינו איך יוצרים מסך משתמש, מגדרים תמונה רקע, מעלים על המסך תמונה קטנה, מקבלים קלט מהעכבר ומהמקלדת ומשמיעים צלילים.

לאחר מכן ראיינו כיצד השימוש ב-OOP מאפשר לנו להגדיר בקלות כל תמונה שנרצה בתור אובייקט ולהשתמש במתודות מתאימות כדי להזין תמונות ולבזוק אם הן מתנגשות.

בצד ההנאה מתכנות משחק מחשב, המטרה היא להתנסות בכתיבה קוד OOP פיתוני ולראות את היתרונות שלו על פני קוד שאינו OOP.

פרק 12 – מיליון

בפרק זה נכיר טיפוס חדש של פיתון – **מילון** (dictionary). מהו מילון? זה אוסף של זוגות, כאשר כל זוג מכיל מפתח וערך. מפתח נקרא **key** וערך נקרא **value**. לדוגמה, אפשר להגיד מילון שמכיל ציונים, כאשר המפתח הוא תעודת זהות. מגדירים מילון באמצעות סוגרים מסולסים, כך:

```
students_grade = {}
```

כעת ניתן להזין לתוכו זוגות של מפתחות וערכים. לדוגמה:

```
students_grade['00001234'] = 85
students_grade['00003579'] = 95
students_grade['00002468'] = 65
```

בצד שמאל, בירוק, אלו המפתחות. במקרה זה בחרנו שהמפתחות יהיו מחרוזות, אך אפשר לבחור בטיפוסים אחרים משניים נוספים. לדוגמה `tuple`. בצד ימין, בכחול, אלו הערכים. במקרה זה בחרנו שהערכים יהיו מティיפוס `int`, אך אפשר להזין ערכים מכל טיפוס שנרצה.

אם נרצה לשלוף את אחד הערכים, נקרא למילון עם המפתח המתאים. לדוגמה:

```
print students_grade['00003579']
```

דפס את הערך .95

אם נרצה להגיד מילון שמאראש יש לו ערכים התחלתיים, נוכל לעשות זאת באמצעות הסימן נקודותיים:

```
students_grade = {'00001234': 85, '00003579': 95}
```

כיוון שמיילון הוא אוסף של איברים, ניתן לעבור על האיברים שבו באמצעות לולאת `for`. לדוגמה:

```
for key in students_grade:
    print 'Student ID: {}, grade: {}'.format(key, students_grade[key])
```

הסבר: כאשר אנחנו מביצעים לולאת `for` על מילון אנחנו רצים על אוסף כל המפתחות שקיים במילון. עברו כל מפתח, אנחנו מדפיסים את המפתח ואת הערך הצמוד אליו.

Get, in, pop, keys, values

מה יקרה אם נחפש במילון מפתח שאינו קיים? לדוגמה, נבקש את הציון שישיך לתעודת זהות שאינה קיימת במילון:

```
print students_grade['00009999']
```

נקבל exception:

```
Traceback (most recent call last):
  print students_grade['00009999']
KeyError: '00009999'
```

פקודת `get` מאפשרת לנו לבצע "בטעות". ככלומר, אם המפתח קיים יוחזר הערך `None`. נזכיר, כי הערך `None` משמעו "כלום". זו מילה שמרורה. להלן דוגמאות לשימוש ב-`get` עם מפתח קיים ומפתח שאינו קיים. הריצו אותן וצפו בתוצאות שמתקבלות:

```
print students_grade.get('00001234')
print students_grade.get('00009999')
```

שימוש ל-`get`, ש-`get` זורשת סוגרים עגולים, כיוון שהוא מתחזק (של אובייקט מטיבוס מילון).

אם אנחנו רק רוצים לדעת אם מפתח נמצא במילון, נוכל להשתמש באופרטור `in`. לדוגמה:

```
print '00001234' in students_grade
print '00009999' in students_grade
```

נקבל `True` או `False`.

אם נרצה לקבל את רשימת כל המפתחות שקיימים במילון, או של כל הערכים, נוכל להשתמש בMETHODS `keys` ו-`values`.
לדוגמה:

```
print students_grade.keys()
print students_grade.values()
```

ונקבל:

```
[ '00003579', '00002468', '00001234' ]
[ 95, 65, 85 ]
```

תרגיל – קניות



- א. צרו מילון בשם `prices`. המפתח הוא שם המוצר והערך הוא מחיר המוצר. הוסיפו כ-5 מוצרים למילון. לדוגמה –
בגינות, 10 ₪. תפוחים, 8 ₪. לחם, 7 ₪. גבינה, 20 ₪. מיץ, 15 ₪.

- ב. צרו מילון בשם `shopping_cart`. המפתח הוא שם המוצר והערך הוא כמות המוצרים מסווג זה בעגלת הקניות. הוסיפו מספר מוצרים למילון. לדוגמה – בנות, 2 ייחדות. לחם, 3 ייחדות. גבינה, 1 ייחדה.
- ג. הכנסו למשתנה בשם `total` את סכום הקניות בעגלת. בידקו שהסכום שמתකבל הוא נכון.
- ד. כתע, הזינו לתוך `shopping_cart` מוצר שאיןו קיים ברשימה המהירים. עליכם לוודא שהתוכנית אינה עפה על שגיאה אלא מדפסה הודעה מתאימה.

תרגיל מסכם מילונים – (google classes wordcount קרדיט: wordcount)



הורידו את קובץ התרגיל מהלינק הבא:

www.cyber.org.il/python/wordcount.zip

בתרגיל אתם נדרשם לקרוא את הקובץ `alice.txt` ולהדפיס למסך את כמות המופעים השונים של כל מילה.

טיפ: השתמשו ב-`split` על מנת להפריד בין מילים.

למתקדים: טפלו במקרים של מילים זרות, שנבדלות רק בסימני פיסוק שצמודים אליהן. לדוגמה – ‘her’ ו-‘.’

(עם נקודה בסוף). בדוגמה זו עליכם לԶוזות שמדובר במקרה מילה ולספר יחד את כל המופעים.



מילונים, מתחת למכסה המנווע (הרחבת)



בחלק זה נבין יותר לעומק איך עובד מילון. זה אינו חלק מעשי, אבל תלמידים סקרנים ימצאו בו עניין.

דמיינו שאתם עובדים בבית מלון בשם 'Hash Gardens'. זה מלון מיוחד מכיוון שבבעל המלון הוא מתמטיקאי מטורף שלא מוכן לשמר שום מידע על האורחיםם שלו, לא על מחשב ולא על פיסת נייר. לנכון במלון אין רשימה של מספרי החדרים של כל האורחיםם. כמובן, אין לכם דרך לדעת באיזה מספר חדר מתגורר כל אורחה. בוקר אחד הטלפון לדלק מצצל. מישחו מבקש לשוחח עם מר smith john. איך אתם מעבירים את השיחה לחדר שלו?

כל הנראה, תצטרכו לחשוף את מר סמית' בכל החדרים. לא קל, במיוחד אם המלון שלכם גדול מאוד. תבזבזו הרבה זמן באיתור מסטר החדר של כל אורחה שמתקשרים אליו.

אתם פונים לבבעל המלון ומתחננים שיתונן לכם לשמר במקום כלשהו את מספרי החדרים של האורחיםם, אבל הוא פשוט מהיר ונותן לכם עצה מתמטית. מעכשו, כל אורחה שmagiu למלון מקבל מכל מסטר חדר שיש קשר מתמטי בין שמו האורחה. כך, למרות שלא רשום לכם בשום מקום איזה חדר קיבל כל אורחה, כאשר מתקשרים ומבקשים לשוחח עם אורחה כלשהו, אתם יכולים לחשב במהירות את מסטר החדר המבוקש.

להלן דוגמה לפונקציה שמחשבת מסטר החדר על פי שם האורחה: נניח שבמלון יש 113 חדרים, אשר ממוספרים מ-0 ועד 112 (כפי שניתן לצפota מתמטי, כמות החדרים במלון היא מספר ראשוני). כל אורחה שmagiu למלון מקבל חדר שמספרו מחושב לפי הנוסחה הבאה: נכפול את ערך ה-ascii של כל התווים בשם האורחה וلتוצאה נבצע מודולו 113. באופן זה הנפק כל מהירותה למספר בין 0 ל-112. התוצאה תהיה מסטר החדר.

להלן קוד של פונקציה שמבצעת את החישוב הנ"ל:

```
ROOMS = 113
```

```
def find_room_number(guest_name):
    tmp = ord(guest_name[0])
    for letter in guest_name[1:] :
        tmp *= ord(letter)
    return tmp % ROOMS

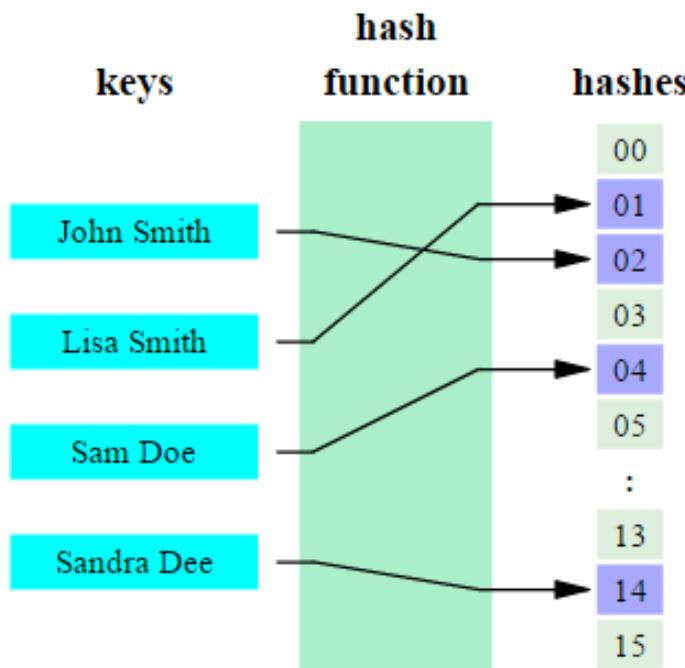
def main():
    room = find_room_number('john smith')
    print room
```

כאשר מגיע אלינו האורה smith john, הפונקציה תחשב את מספר החדר שלו – 61. אם מישהו יתקשר למר סמי', יוכל לחשב מחדש את מספר החדר שלו.

כפי שואלי ניחתמו, הפונקציה `find_room_number` היא בעצם פונקציית `hash`. מה מיוחד בה?

- א. היא מתאימה לכל מהרוות מספר בתחום קבוע מראש
- ב. למהרוות זהות מתקיים `hash` זהה. לא יכול להיות שאורהים עם שמות זהים יקבלו חדרים שונים
- ג. אי אפשר (או לפחות, קשה מאוד) לשחזר את שם האורה מתוך ה-`hash` של השם שלו. אם אנחנו יודעים שאורה נמצא בחדר 61, אי אפשר לגלוות שמו הוא smith john. במלילים אחרות, זו ה-`hash` פונקציה חד-כיוונית.

אך יש בעיה אחת עם פונקציית ה-`hash` שלנו: יכול להיות מצב שבו שני אורחים בעלי שמות שונים יקבלו את אותו חדר, מה שיצור התנגשות (collision). לכן, פונקציית `hash` טובה היא כזו שהסיכוי להתנגשות בה הוא מאוד קטן.



המחשה של פונקציית `hash`. מקור: ויקיפדיה

נזהור כתע מועלם המלונאות והתיירות אל עולם הפיתון. בפני מי שבנה את שפת פיתון עצמה בעיה של העובד במילון: איך מוצאים מהר את הערך שצמוד למפתח מסוים? כפי שרainer, חיפוש על פניו כל המפתחות הוא יקר בזמן, במיוחד אם יש במאילון שלנו הרבה מפתחות. הפתרון שמיילונים עושים בו שימוש הוא כMOVEDן פונקציית `hash`.

מילון בפייתון מתחילה בטור טבלה ריקה. כאשר אנחנו מכניםם למילון מפתח וערך, פיתון מחשב את ה-`hash` של המפתח ומתקבל מספר. המספר הזה יהיה האינדקס בטבלה שבו פיתון ישמר את המפתח ואת הערך הצמוד אליו. לדוגמה, אם ה-`hash` היה 5 האינדקס בטבלה יהיה 5.

מה קורה כאשר אנחנו רוצים לשולוף ערך מהמילון? אנחנו מעבירים לפיתון את המפתח. על בסיס המפתח פיתון מחשב את ה-`hash` וניגש אל המקום בטבלה שזה המספר שלו. באופן זה, חיפוש של ערך בתוך מילון לוקח זמן קצר מאד ומעט אינו תלוי בຄמות המפתחות שיש במילון.

עלול לקרות מצב שבו לשני מפתחות יש את אותו `hash`. ראוי לציין שפיתון משתמש בפונקציית `hash` ("טובה", כזו שדואגת שהסיכוי שנייה מפתחות יהיה עליי אותו `hash` הוא קטן. למצב של שני מפתחות יש אותו `hash` קוראים `collision` ויש דרך לטפל בו. הערך הראשון מוכנס לטבלה כרגע. כאשר פיתון רוצה להכניס את הערך השני לאותו מקום בטבלה, הוא בודק ומגילה שבטבלה כבר יש מפתח חדש שקיים בטבלה, סימן שקרה `collision`. המפתח החדש ישמר במקום הפניו הבא. לדוגמה, אם המקום 5 בטבלה כבר תפוס, פיתון יכול לנוסח לשומר באינדקס 6. אם רצחה לשולוף מהמילון את הערך של המפתח השני, פיתון יגש קודם כל לאינדקס 5. שם הוא יגלה שהמפתח אינו מתחאים ולכן הוא ימשיך לאינדקס הבא עד שיימצא את המפתח המבוקש.

סוגי מפתחות

ראינו שככל צמד שמכנס למילון מורכב ממספר מפתח וערך. בטור מפתח השתמשנו בשתיים מטיפוס מחרוזת. האם כל טיפוס משתנה יכול להיות מפתח?

כדי להסביר על זה, נזכיר בדיון שלנו בנושא `mutable` ו-`immutable`. אובייקט מסווג `mutable` הוא אובייקט שניתן לשנות, כדוגמת רשיימה. ככלמנו על רשימות ראיינו שאנו יכולים להגדיר רשיימה, לשנות את אחד האיברים בה ולאחר השינוי ה-`id` של הרשימה ישאר ללא שינוי. הערך של אחד האיברים השנתה אף זו עדין אותה רשיימה. לעומת זאת, אובייקטים מסווג `immutable` לא ניתנים לשינוי. למשל, הדרך היחידה לשנות אותם היא להגדיר מחדש או כMOVן משתנה ה-`id` שלהם.

הבה נניח שהיינו יכולים להשתמש במשתנה מסווג `mutable` בטור מפתח. לצורך הדיון, ניקח שתי רשיימות, שנזכיר שהן `:mutable`

```
a = ['apple']
```

```
b = ['banana']
```

שימוש לב שאלות רשיימות, אשר מכילות כל אחת איבר אחד מטיפוס מחרוזות.

כעת, נגידיר מילון:

```
fruits = {a:1, b:2}
```

פיתון חישב את ה-`hash` של `a` ושל `b` והכניס כל אחד מהם למקום המתאים לו בטבלה.

כעת אנחנו משנים את ערכו של `b`:

```
b[0] = 'apple'
```

אנחנו יכולים לעשות זאת כיון שמדובר ברשימה, שהיא `mutable`.

מה יקרה אם כעת ננסה לשולוף את `s` מהמילון?

פיתון יחשב את ה-`hash` של `s` ויקבל ערך מסוים, אך הערך לא יתאים למיקום `s-b` שומר בו. אם לא די בכך, ה-`hash` שיתקבל דוקא כן יתאים למיקום אחר בטבלה – המיקום אשר שומר את `a` ושיש לו מפתח זהה – וכך החיפוש יחזיר את הערך השגוי "1".

סיכום

בפרק זה למדנו אודות מבנה נתונים מיוחד. מילון מאפשר לנו לשמר נתונים ולמצוא אותם במהירות רבה, תוך שימוש במפתחות ובפונקציה מתמטית – `hash`. למדנו לצור מילון, להזין לתוכו ערכים ומפתחות ולהפוך ערכיהם במילון באמצעות מפתח. בהמשך סקרנו מספר תכונות של פונקציית `hash` וראינו כיצד תכונות אלו עוזרות לה להיות שימושית עבור מילוניים.

Magic Functions, List Comprehensions – 13

בפרק זה נלמד לנצל יכולות של פיתון על מנת לכתוב קוד בצורה יותר קצרה ו'פייטונית'.

List Comprehensions

נניח שאחננו רוצים ליצור רשימה של איברים שיש להם חוקיות מסוימת. לדוגמה, חזוקות של מספרים. רשימה כזו אי אפשר ליצור באמצעות `range`, מכיוון שהקפיצה בין מספרים אינה קבועה. מצד שני, אנחנו לא רוצים לכתוב באופן ידני את המספרים我们知道. אנחנו כן רוצים לנצל את העבודה שישנה חוקיות למספרים.

דרך אחת ליצור את הרשימה שלנו היא באמצעות `for`, לדוגמה, כך:

```
squares = []
for i in range(100):
    squares.append(i**2)
```

כך ייצרו רשימה שמכילה את האיברים $[0, 2, 4, \dots, 99^2]$

כעת נלמד **syntax** אלטרנטיבי, שיוצר את אותה הרשימה לעיל בשורה אחת בלבד. צורת כתיבה זו נהוגה מאוד בשפת פיתון, שכן לאחר שתרגלים אליה היא נוחה מאוד לקריאה ולכתיבת-code. **list comprehensions** זה קוראיםsyntax זה קוראים **list comprehensions**.

באמצעות **list comprehensions** אפשר ליצור את הרשימה לעיל בצורה הבאה:

```
squares = [i**2 for i in range(100)]
```

שימוש לב לשינוי בסדר הפקודות. בעוד את לולאת `for` קוראים "עבור כל אחד מהאיברים ב-`range(100)`" תבצע הפעלה בריבוע", את הביטוי שבסוגרים צריך לקרוא "תבצע את i^2 " עבור כל אחד מהאיברים ב-`range(100)`".

אנחנו יכולים להוסיף תנאים. נגיד, שמור רק את הריבועים של איברים זוגיים. שוב, נציג קודם את צורת הכתיבה עם לולאת `for`:

```
squares = []
for i in range(100):
    if i % 2 == 0:
        squares.append(i**2)
```

והכתב המקוצר, באמצעות **:list comprehensions**

```
squares = [i**2 for i in range(100) if i%2==0]
```

לא תמיד קל לקרוא קוד שכתוב בשורה אחת ארוכה. למען קלות הקריאה, אפשר לחלק את השורה הנ"ל במספר שורות:

```
squares = [i**2
           for i in range(100)
           if i%2==0]
```

בינתיים כתבנו לוලאות פשוטות, שניתן היה לכתוב ב-3-4 שורות ללא list comprehensions. לעיתים, נרצה לבצע דברים מורכבים יותר באמצעות list comprehensions.

לצורך התרגיל, נרצה ליצור רשימה של כל המספרים הראשוניים. כדי לעשות זאת, ניצור קודם כל רשימה של המספרים הלא ראשוניים. לאחר מכן, כל מספר שלא נמצא בראשימת הלא ראשוניים – יהיה מספר ראשוני. להלן קוד שמבצע זאת באמצעות :for לוולאות:

```
1 import math
2 LIMIT = 100
3
4 root = int(math.sqrt(LIMIT))
5 non_primes = []
6 for i in range(2, root):
7     for j in range(2*i, LIMIT, i):
8         non_primes.append(j)
9 non_primes.sort()
10
11 primes = []
12 for i in range(LIMIT):
13     if not i in non_primes:
14         primes.append(i)
```

הסביר: הקבוע LIMIT מגדיר מה הגבול העליון של המספרים הראשוניים שאנו מחפשים. בדוגמה זו, נרצה לקבל את הראשוניים עד 100. כדי למצוא את כל המספרים הלא ראשוניים עד 100 אין צורך לבדוק את המכפלות של כל המספרים, אלא רק עד שורש 100. לכן, מוגדר המשתנה root אשר ישמש את הלולאה החיצונית.

בלולאה הפנימית, בשורה 7, מוצאים את כל המכפלות של i הקטנות מ-LIMIT. לדוגמה, נניח ש- i הוא 5, אינדקס ההתחלה הוא $5^2=25$ ומתקדים בקפיצות של 5. כל מכפלה כזו מתווספת לרשימה בשורה 8. מיוון הרשימה בשורה 9 מתבצע רק לשם היופי.

בשורות 11–14 אנחנו יוצרים רשימה חדשה, לתוךה מוכנסים כל המספרים עד LIMIT אשר אינם מופיעים ברשימה של הלא ראשוניים.

וכעת, אותו קוד כפי שהוא כתוב עם list comprehensions

```

1 import math
2 LIMIT = 100
3
4 root = int(math.sqrt(LIMIT))
5 non_primes = [j
6             for i in range(2, root)
7             for j in range(2*i, LIMIT, i)]
8
9 primes = [i
10            for i in range(LIMIT)
11            if not i in non_primes]
```

תרגיל – קיצוצים (เครดיט: עומר רוזנבוים, שי סדובסקי)



כתבו את הפונקציה avg_diff, אשר מקבלת שתי רשימות של מספרים ומחזירה את ההפרש הממוצע ביניהן. לדוגמה, עבור הרשימות [1,2,3,4],[1,1,1,1] יוחזר 1.5. נשמע פשוט? אז זהו, שunkב קיצוצים בתקציב אנחנו נאלצים להטייל ולכתוב את הפונקציה בשורה אחת בלבד. זה אומר ש-:

- אסור לרדת שורה

- אסור להשתמש בתו ":"

אפשר להניח שתשתי הרשימות מכילות מספרים, שהן בעלות אותו אורך ושחן אין ריקות.

תרגיל – אנטיבי



כתבו את הפונקציה anti_bi אשר מקבלת מחרוזת ומחזירה אותה ללא מופעים של התו 'b'. זיכרו – הקיצוצים עדין

נמשכים! ☺

לסיכום, דרך דורך חזקה לכתוב בצורה קצרה מאוד קוד מורכב. הן יכולות להכיל גם תנאים מורכבים, כאשר מתרגלים אליו – **list comprehensions** – עוזרות לנו לכתוב קוד בצורה קריאה, יפה ו מהירה.

Lambda

פונקציות **lambda** הן פונקציות לשימוש חד פעמי – אנחנו רוצים לעשות פעולה לא מאוד מורכבת ונראה לנו מיותר להגיד פונקציה במיוחד בשביב זה.

שימוש לב לאופן הכתיבה הבא:

```
f = lambda x: 2*x +1
```

איך קוראים את זה? "הפונקציה **f** מקבלת כפרמטר **x** ומזהירה את $x^2 + 1$ ". כמובן הקוד הזה להגדרת הפונקציה הבאה:

```
|def f(x):
|    return 2*x + 1
```

אם נרצה לקרוא ל-**f** נעשה זאת בדיק כמה שקוראים לפונקציה רגילה. לדוגמה (**f(5)** יחזיר 11).

אפשר גם להגיד פונקציה **lambda** שמקבלת כמה פרמטרים. לדוגמה:

```
f = lambda x, y: x*y + x + y
```

לדוגמה, (**f(2, 3)** יחזיר 11).

מתי משתמש בפונקציית **lambda**? לדוגמה, כאשר אנחנו רוצים להעביר **key** לפונקציית **sort**, אפשר להגיד את ה-**key** בתווך פונקציית **lambda**. דוגמה נוספת אשר נלמד בעתיד – כאשר נלמד רשותות וכותבו קוד לסינון מידע שעובר ברשות, משתמש בפונקציית **lambda**.

תרגילים



- כתבו פונקציית **lambda** שמקבלת מספר ומזהירה את המספר פלוס 2.

- כתבו פונקציית **lambda** שמחזירה את הערך המוחלט של מספר, והשתמשו בפונקציה זו על מנת למיין באמצעות **sort** רשימה של מספרים, לדוגמה: [2, -8, 5, -6, -1, 3].

הfonקציות הבאות שנלמד, `filter`, `map`, `reduce` ו-`list comprehensions` אך חשוב להכיר אותן כיון שהן נמצאות לעיתים בקוד פיתון.

Map

fonקציית `map` מקבלת פונקציה ורשימה. נוצרת רשימה חדשה, שהיא התוצאה של הרצאת הפונקציה על כל אחד מאיברי הרשימה המקורית.

במילים אחרות, לכתוב `new_list = map(func, old_list)` זה כמו לכתוב:

```
new_list = []
for element in old_list:
    new_list.append(func(element))
```

כפי שבתча ניחשתם, את `func` אין צורך להגיד ממש, אלא ניתן להכניס `lambda` כרצוננו.

לדוגמה, ניקח רשימה מספרים ונרצה לכפול כל מספר פי 2 ולהוסיף 1:

```
old_list = [3, 6, 1, 7, 5]
print map(lambda x: 2*x +1, old_list)
```

ונקבל `[7, 13, 3, 15, 11]`.

אנחנו לא חייבים להסתפק ברשימה אחת. אפשר לכתוב פונקציית `lambda` שמקבלת יותר פרמטר אחד ולהעביר ל-`map` כמה מתחיינה של רישימות. לדוגמה, ישנן שתי רשימות ואנחנו מעוניינים לכפול אותן זו בזו (מה שנקרא "מכפלה וקטורית" במתמטיקה):

```
l1 = [1, 2, -5, 6]
l2 = [2, -1, 3, 4]
print map(lambda x, y: x*y, l1, l2)
```

ונקבל `[2, -2, -15, 24]`.

תרגיל – אנחנו על המפה (เครดיט: עומר רוזנבוים, שי סדובסקי)



כיתבו פונקציה שמקבלת מחרוזת ומחזירה מחרוזת חדשה, אשר כל האותיות בה מוכפלות. לדוגמה, עבור המחרוזת 'Cyber' יתקבל 'CCyybbeerr'. האתגר הוא כמפורט לעילו כתוב את הפונקציה בשורה אחת ☺ נסו את הפונקציה עם מגוון קלטים ובידקו שהתוצאה נכונה.

Filter

פונקציה זו נועדה לסנן (to filter) רק איברים רלבנטיים מתוך רשימה קיימת. לשם כך, הפונקציה `filter` מקבלת פונקציה ורשימה, בודקת על כל אחד מאיברי הרשימה האם הפונקציה מחזירה עליו `True` ומהזירה רשימה רק של האיברים שהחזירו `True`. כך, לדוגמה:

```
new_list = filter(func, old_list)
```

```
new_list = []
for element in old_list:
    if func(element) is True:
        new_list.append(element)
```

לדוגמה, נרצה לקחת רשימה וליצור ממנה רשימה חדשה רק של המספרים הזוגיים:

```
old_list = [2, 3, 4, 7, 8, 10]
print filter(lambda x: x%2 == 0, old_list)
```

הסבר: קודם כל אנחנו יוצרים פונקציית `lambda`, שהחזיר את הערך הבוליאני של הביטוי `x%2==0`. לאחר מכן אנחנו מכניסים לתוך `filter` את הפונקציה הנ"ל יחד עם רשימה.

תרגיל – את מסגנת אותו (เครดיט: עומר רוזנבוים, שי סדובסקי)



צרו פונקציה שמקבלת מספר ומחזירה רשימה של כל המספרים הקטנים ממנו אשר מתחלקים ב-3. לדוגמה, עבור המספר 10 תוחזר הרשימה `[3, 6, 9]`. כאמור, אורך הפונקציה חייב להיות שורה אחת בלבד ☺

Reduce

הפונקציה `reduce` מקבלת פונקציה ורשימה ומוחזירה ערך יחיד. הערך זה הוא התוצאה של ביצוע הפונקציה על איברי הרשימה שוב ושוב עד שנותר ערך יחיד. אפשר לחשב על `reduce` בתור `map` שימושי להתבצע כל עוד אורך הרשימה החדש שנותרת גדול מ-1.

לדוגמא, יש לנו רשימה ואנחנו רוצים ליחס את הסכום של האיברים שלה:

```
old_list = [2, 3, 4, 7, 8, 10]
print reduce(lambda x, y: x+y, old_list)
```

אפשר לדמיין שבכל שלב נוצרת רשימה חדשה, שבה האיבר הראשון הוא סכום שני האיברים ברשימה המקורית:

- [5, 4, 7, 8, 10] •
- [9, 7, 8, 10] •
- [16, 8, 10] •
- [24, 10] •
- [34] •

כאשר הגיענו לרשימה באורך 1 החישוב ייעצץ ויוחזר הערך שנותר בה.



תרגיל – שיעור ספרות (เครดיט: עומר רוזנבוים, שי סדובסקי)

עליכם ליצור פונקציה שמקבלת מספר ומחזירה את סכום הספרות שלו. לדוגמה, עבור 104 תקבל התוצאה 5. כרגע, רק שורה אחת של קוד ☺

סיכום

בפרק זה למדנו כיצד לחת קוד, שבדרך כלל היינו כתבים אותו באמצעות `for`, ולכתוב אותו בצורה קצרה ופешטה באמצעות `list comprehensions` או באמצעות פונקציות הקסם של פיתון: `reduce`-ו-`lambda`, `map`, `filter`. חלק מפונקציות אלו יהיו שימושיות בלימודי רשות. המשך לימוד מוצלח!