תכנות בשפת פייתון

ברק גונן

המרכז לחינוך סייבר
CYBER EDUCATION CENTER

תכנות בשפת פייתון

Python Programming / Barak Gonen

גרסה 2.01 ספטמבר 2020

<u>כתיבה</u>:

ברק גונן

<u>עריכה</u>:

עומר רוזנבוים

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

מהדורה ראשונה תשע"ז 2017

מהדורה שנייה תש"פ 2020

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

http://www.cyber.org.il

תוכן עניינים

הקדמה והתקנות נדרשות 8

התקנות נדרשות 8

אייקונים 18

תודות 18

תוכן עניינים מצגות פייתון 19

פרק 1 – מבוא ללימוד פייתון 20

מהי שפת סקריפטים? 20

עבודה דרך command line 23

מספרים בבסיסים שונים 24

Help 26

הסימן _ (קו תחתון) 28

הרצת תוכניות פייתון דרך ה-command line 28

סיכום 29

פרק 2 – סביבת עבודה **PyCharm** 30

פתיחת קובץ פייתון 30

קובץ הפייתון הראשון שלנו 32

סדר ההרצה של פקודות בסקריפט פייתון 33

התרעה על שגיאות 35

הרצת הסקריפט ומסך המשתמש 37

דיבוג עם PyCharm 39

העברת פרמטרים לסקריפט 42

סיכום 43

פרק 3 – משתנים, תנאים ולולאות 44

סוגי משתנים בפייתון 44

תנאים 46

תנאים מורכבים 47

שימוש ב-is 49

בלוק 49

תנאי else, elif 51

לולאת while 52

לולאות for 54

pass 56

פרק 4 – מחרוזות 58

הגדרת מחרוזת 58

ביצוע format ל-print 59

חיתוך מחרוזות – string slicing 60

פקודות על מחרוזות 62

dir, help 63

צירופי תווים מיוחדים ו-raw string 64

קבלת קלט מהמשתמש – raw_input 65

פרק 5 – פונקציות 69

כתיבת פונקציה בפייתון 69

return 72

None 73

scope של משתנים 73

פייתון מתחת למכסה המנוע (הרחבה) 79

id, is 81

העברת פרמטרים לפונקציה 83

סיכום 85

פרק 6 – List, Tuple 86

הגדרת List 86

Mutable, immutable 88

פעולות על רשימות 90

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

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

קבלת קלט מהמקלדת 189
השמעת צלילים...... 189
PyGame מתקדם – שילוב OOP 191
מבוא...... 191
הגדרת class...... 192
הוספת מתודות accessors ו-mutators שימושיות 193
הגדרת אובייקטים בתוכנית הראשית 193
sprite.Group()...... 195
יצירת אובייקטים חדשים...... 196
הזזת האובייקטים 197
בדיקת התנגשויות...... 198
סיכום 203
פרק 12 – מילונים...... 204
מילונים, מתחת למכסה המנוע (הרחבה) 207
סוגי מפתחות 209
סיכום 210
פרק 13 – Magic Functions, List Comprehensions 211
List Comprehensions 211
Lambda 214
Map 215
Filter 216
Reduce 217
סיכום 217

הקדמה והתקנות נדרשות

ברוכים הבאים לשפת פייתון! אם אתם קוראים ספר זה כנראה שאתם עושים את צעדיכם הראשונים במגמת הגנת סייבר. מדוע לומדים דווקא פייתון? שפת פייתון נבחרה ללימוד כיוון שהיא נמצאת בשימוש רחב בתעשייה, באקדמיה וגם בקרב היחידות הטכנולוגיות בצה"ל. חומרי הלימוד הבאים של מגמת הסייבר מבוססים על שפת פייתון ולכן נדרשת שליטה בשפה כבסיס ללימוד יתר התכנים.

הספר כולל את החומר הנדרש כבסיס ללימוד רשתות מחשבים, מומלץ ללמוד מן הספר נושאים לפי הצורך – לדוגמה, סביבת העבודה, שימוש במחרוזות ורשימות נדרשים ללימוד רשתות ולכן יש ללמוד אותם תחילה. לעומת זאת, תכנות מונחה עצמים ניתן לדחות את הלימוד של נושא זה להמשך. בנוסף כולל הספר נושאי הרחבה למעוניינים לשלוט בשפה יותר לעומק.

הספר לא מניח כמעט ידע מוקדם, אך הוא מניח היכרות עם רעיונות בסיסיים בתכנות כגון משתנים, תנאים ולולאות. הוא נועד לאפשר לימוד עצמאי, והוא פרקטי וכולל תרגילים רבים. כדי לשלוט בשפת פייתון, כמו בכל שפת תכנות, אי אפשר להסתפק בידע תיאורטי. הספר אינו מתיימר לכסות את כל הנושאים בשפת פייתון, אלא להתמקד בנושאים שסביר שתזדקקו להם במהלך לימודיכם. אחד הנושאים עליהם הספר אינו מרחיב רבות הוא אודות השימוש במודולים, ספריות שפותחו בפייתון ומאפשרות לבצע פעולות מורכבות בקלות יחסית. הוויתור על ההסברים המפורטים אודות מודולים הוא ראשית מכיוון שקשה מאוד להקיף את כל המודולים החשובים בספר לימוד, ושנית מכיוון שמטרתנו היא להעניק לכם את הבסיס ללימוד עצמי של חומרים מתקדמים. כך, כאשר תסיימו את הלימוד מהספר, תוכלו בקלות למצוא מידע באינטרנט אודות כל נושא שתרצו ולשלב אותו בהצלחה בתוכנה שאתם כותבים.

התקנות נדרשות

כאמור ספר הלימוד מבוסס על שפת פייתון. ישנן התקנות רבות של פייתון, לכן נרצה להמליץ על סביבת עבודה ושימוש נכון בסביבת העבודה. להלן פירוט כלל ההתקנות הנדרשות הן ללימוד פייתון והן ללימוד רשתות מחשבים. גרסת הפייתון של ההתקנה היא 3.8. למעוניינים ללמוד רק פייתון, ללא רשתות, אפשר לוותר על התקנת Wireshark, Scapy ו-npcap.

שימו לב:

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

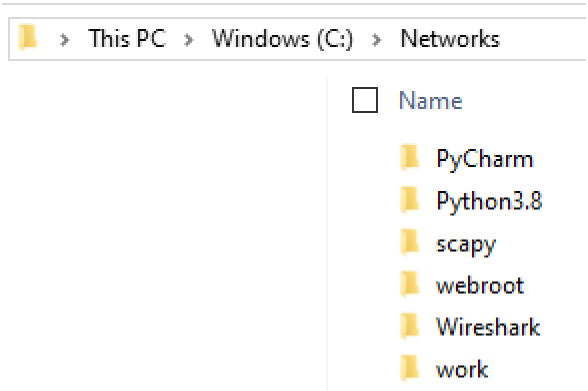
התוכנות הן (לפי סדר ההתקנה):

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

קובץ התקנה מרוכז נמצא בכתובת:

https://data.cyber.org.il/networks/installs.zip

אפשר להתקין את התוכנות בכל תיקיה, אך מנסיון רב שנים התקנה "מפוזרת" של התוכנות גורמת אינספור בעיות ללומדים. לדוגמה, חלק מהתוכנות לא מסוגלות לזהות תיקיות שיש בהן עברית. חלק מהתוכנות דורשות התאמה של משתני סביבה. אם תפעלו לפי ההוראות הבאות, תחסכו לעצמכם בעיות. שימו לב למבנה התיקיות המומלץ בסיום ההתקנות:



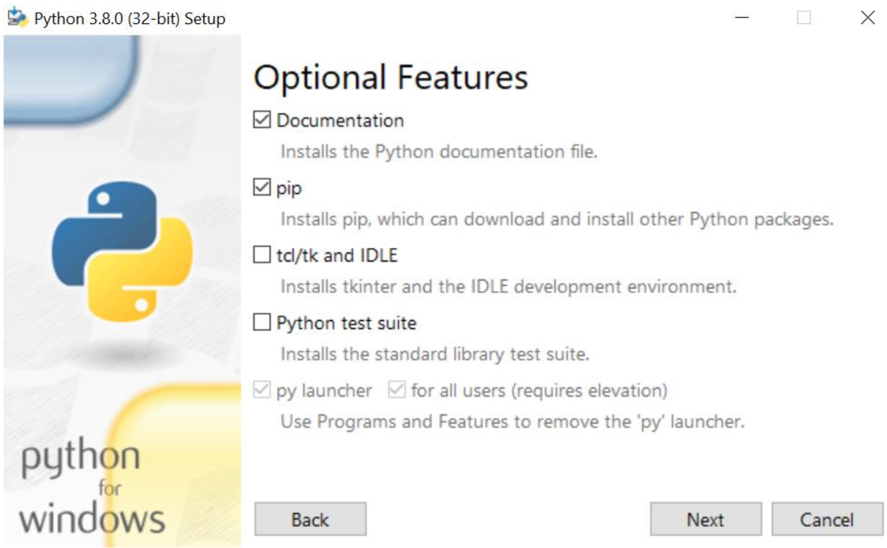
התיקיה הראשית היא Networks, שם נשים את כל תתי התיקיות. ישנן חמש תיקיות המיועדות להתקנות ותיקיה נוספת בשם Work המיועדת לקבצי הפייתון שלכם.

א. <u>התקנת פייתון</u>

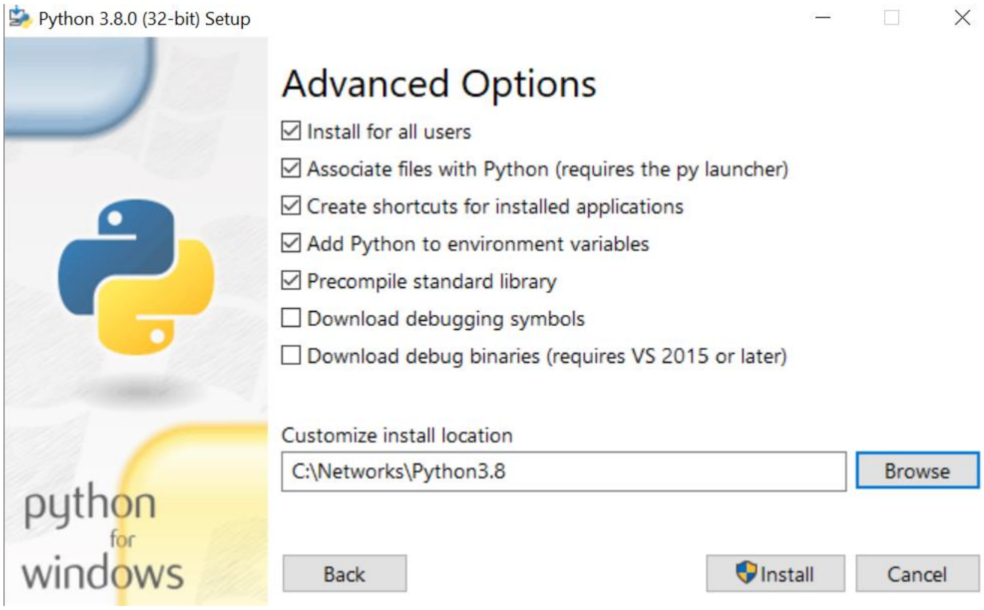
הקליקו על הקובץ python-3.8.0.exe, יופיע מסך ההתקנה הבא:

בחרו באפשרות "customize installation".

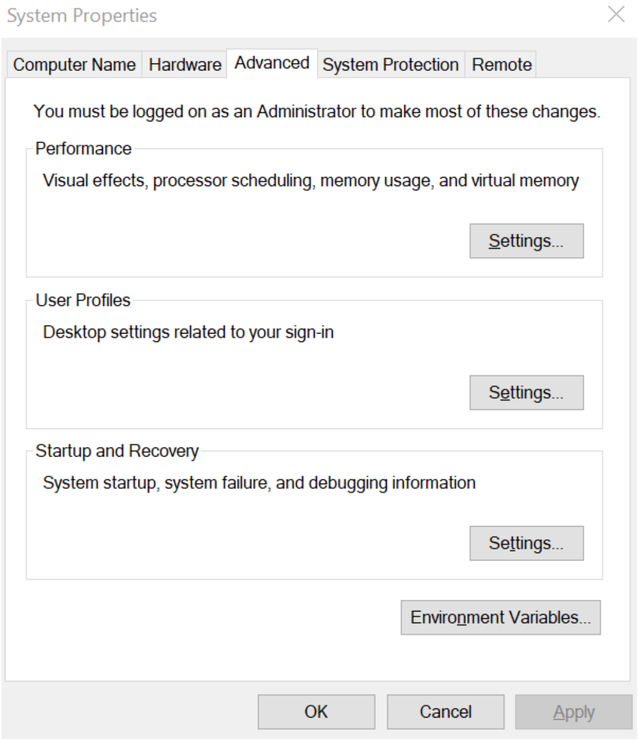
בחרו באפשרויות הבאות:



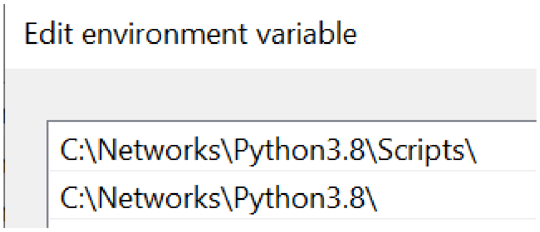
ובמסך הבא בחרו את האפשרויות המסומנות, והזינו בתור תיקיית התקנה את c:\Networks\Python3.8:



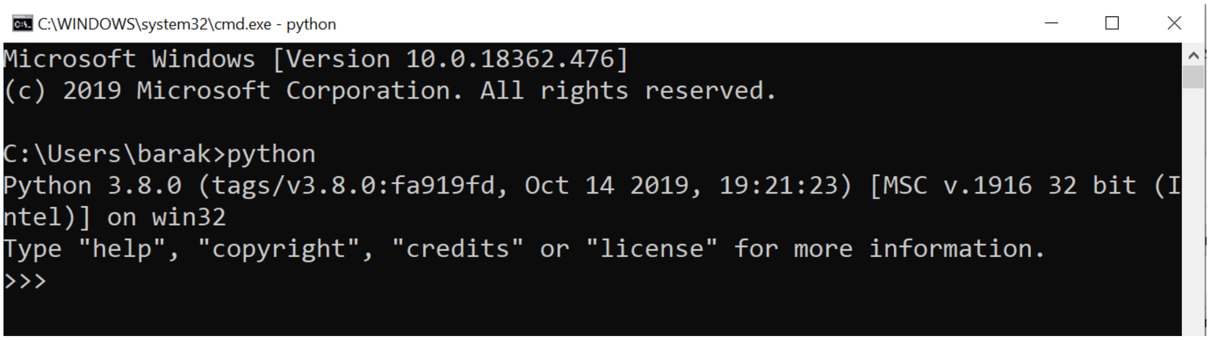
כעת בידקו שפייתון מוגדר בתוך ה-PATH של משתני הסביבה שלכם. במסך החיפוש של windows הקלידו env ובחרו באפשרות Edit The System Environment Variables. הקליקו על הלחצן Environment Variables.



וודאו שבתוך המשתנה PATH יש את שתי הכניסות הבאות. אם הן אינן, הוסיפו אותן ידנית (בהנחה שהתקנתם את פייתון בתוך c:\networks\python3.8)



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

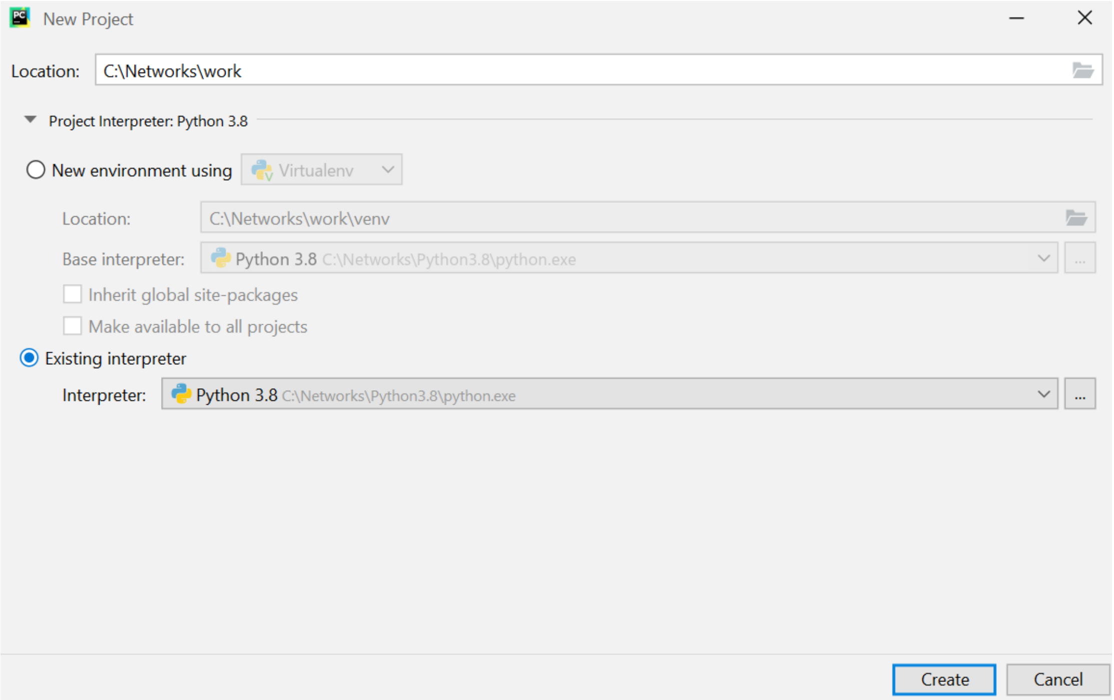


ב. PyCharm

הקליקו על קובץ ההתקנה, היא תתבצע מעצמה.

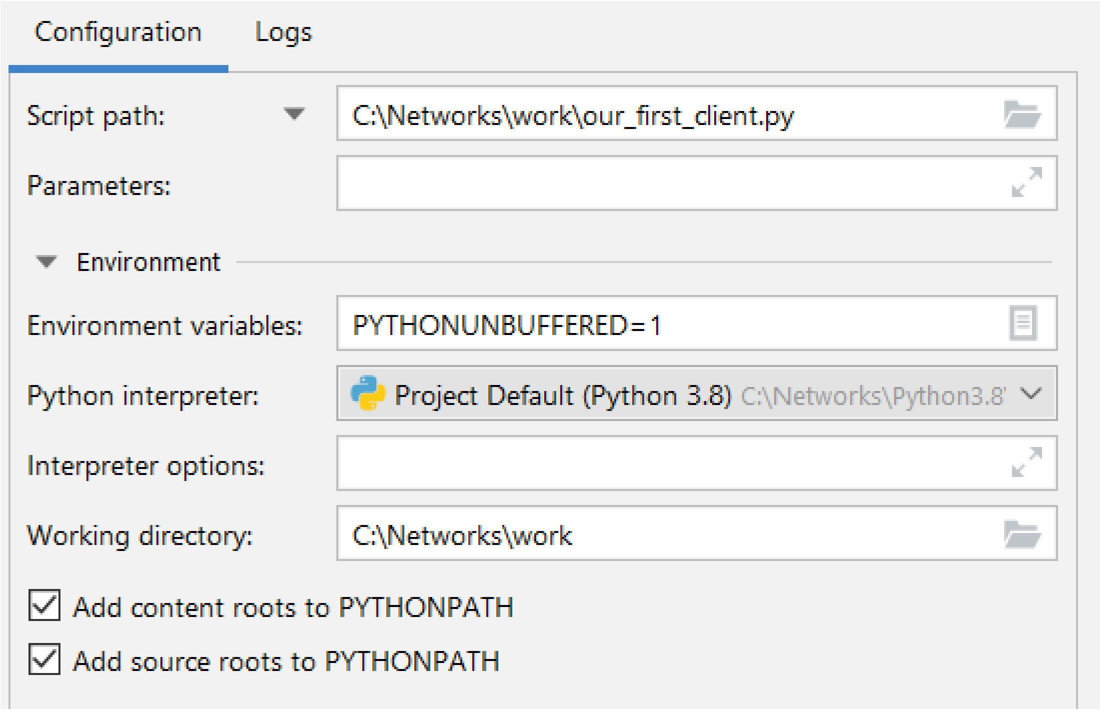
עם ההפעלה, בחרו באפשרות “Create new project” ופיתחו פרוייקט בספריה c:\networks\work. תצטרכו להגדיר היכן נמצא ה-python interpreter שלכם, בצעו זאת כך:

- בחרו באפשרות Existing Interpreter
- כאשר תקליקו על הלחצן שיש עליו ציור של שלוש נקודות, ייפתח לפניכם מבנה התיקיות במחשב שלכם. הגיעו אל התיקיה c:\networks\Python3.8 ובחרו את הקובץ python.exe:



אפשרות נוספת היא להגדיר את ה- Interpreter בעצמכם, עבור כל קובץ אותו אתם מריצים. מתי נכון לעשות זאת? אם קבלתם הודעת שגיאה "Invalid Python Interpreter selected for the project". ההודעה מופיעה שורות הקוד של הקובץ אותו אתם מנסים להריץ:

הקליקו על "Configure Python Interpreter" בצד הימני של ההודעה ותעברו למסך ההגדרות הבא:

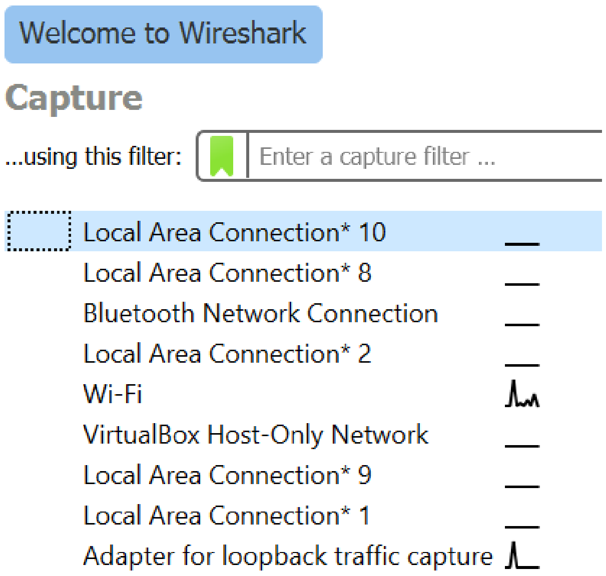


שנו את הערך של ה-interpreter Python כך שיצביע על הקובץ Python.exe שבתיקיה c:\networks\work\python3.8, כעת הקליקו על כפתור ה-OK והבעיה נעלמה.

ג. <u>Wireshark ו-npcap</u>

הקליקו על תוכנת ההתקנה של Wireshark. התוכנה תשאל אם ברצונכם להתקין על הדרך את npcap, בחרו באפשרות הזו.

הקליקו על האייקון של Wireshark. התוכנה תגלה באופן אוטומטי את כל ממשקי הרשת שלכם. כרטיס רשת פעיל יראה גרף עולה ויורד, בהתאם לרמת הפעילות.



במקרה זה, המחשב מחובר דרך Wi-Fi. בחרו בממשק הפעיל והתחילו הסנפה.

ה. scapy

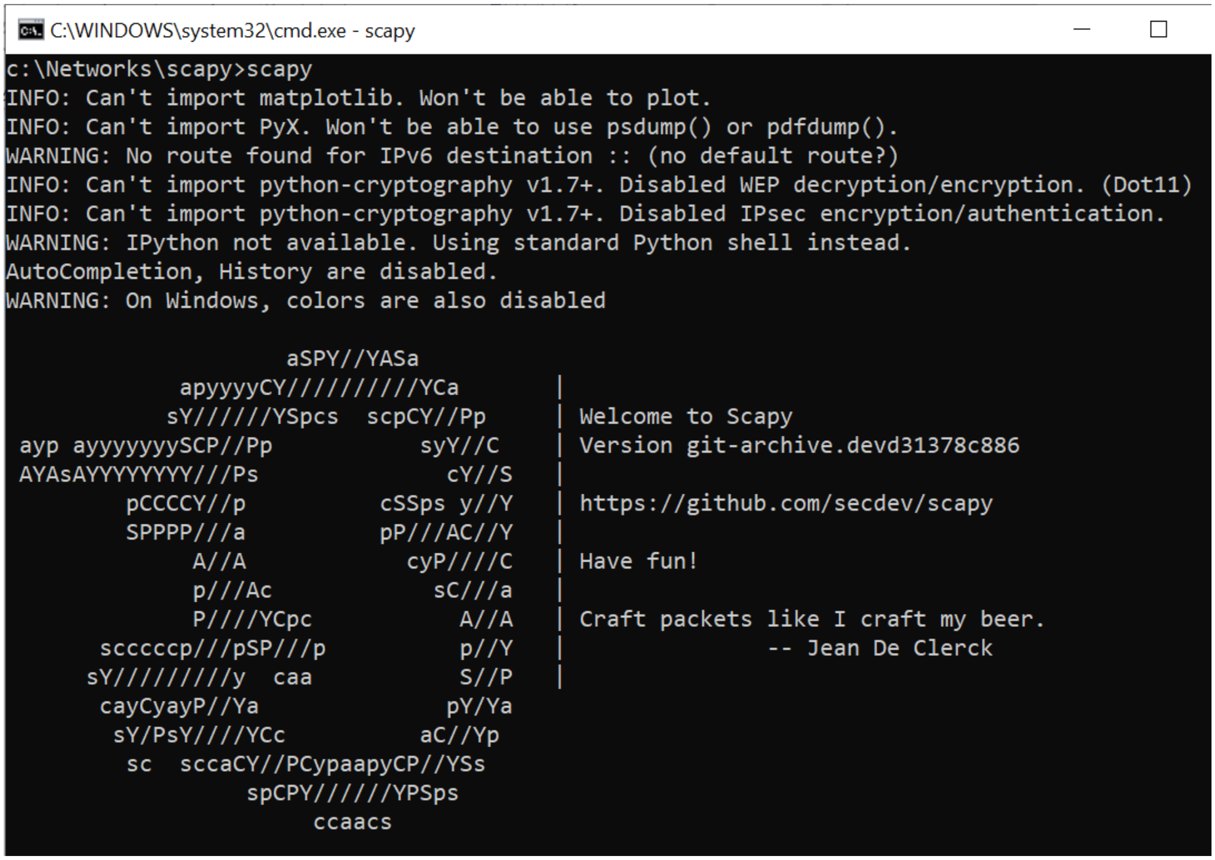
העתיקו את התוכן של התיקיה scapy-master מקובץ ה-zip אל תיקיית c:\networks\scapy.

פיתחו cmd והקלידו

cd c:\networks\scapy

כדי להתקין הקלידו python setup.py install

כדי לבדוק אם ההתקנה הצליחה, הקלידו scapy. תקבלו את המסך הבא, אין מה להיות מוטרדים מהודעות השגיאה- הן שייכות למודולים שאין לנו צורך בהם.



כדי לוודא שה-scapy שלכם עובד כשורה, הסניפו באמצעותו 2 פקטות והציגו אחת מהן:

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

זהו, סיימנו. מוכנים להתחיל לעבוד!

אייקונים

בספר, אנו משתמשים באייקונים הבאים בכדי להדגיש נושאים ובכדי להקל על הקריאה:

שאלה

הגדרה למונח

הדגשת נקודה חשובה

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

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

פתרון מודרך לתרגיל אותו היה עליכם לפתור בעצמכם

רקע היסטורי, או מידע העשרתי אחר

הפניה לסרטון

תודות

ספר זה לא היה נכתב אלמלא תרומתו של עומר רוזנבוים, שסייע הן 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 בצד התרגילים.

ברק גונן

תוכן עניינים מצגות פייתון

להלן לינקים למצגות הלימוד של פייתון, אשר עשויות לסייע לכם במהלך לימוד הפרקים השונים.

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

פרק 1 – מבוא ללימוד פייתון

מהי שפת סקריפטים?

ברוכים הבאים לשפת פייתון! שפת פייתון היא שפה שימושית וקלה לשימוש. לאחר שתשלטו בבסיס השפה תוכלו לכתוב תוכניות מסובכות בקלות יחסית. לדוגמה, במספר שורות קוד תוכלו לגרום לשני מחשבים לשלוח הודעות אחד לשני. בפחות ממאתיים שורות קוד תוכלו לפתח משחק מחשב, עם גרפיקה וצלילים.

שפת פייתון פותחה ב-1990 כשפת סקריפטים. מהי שפת סקריפטים? כדי להבין מהי שפת סקריפטים נצטרך להבין קודם כל כיצד עובדת שפה שאינה שפת סקריפטים, לדוגמה שפת C. כל שפת תוכנה צריכה להפוך בדרך כלשהי לשפת מכונה, כדי שהמעבד של המחשב יוכל להריץ אותה. ההבדל בין שפת סקריפטים לשפה שאיננה שפת סקריפטים הוא במסלול שעובר הקוד עד שהוא הופך לשפת מכונה. קוד שנכתב בשפת C צריך לעבור שני שלבים לפני שהמעבד של המחשב יכול להריץ אותו: השלב הראשון נקרא קומפילציה והוא מבוצע על ידי תוכנה שנקראת קומפיילר. הקומפיילר ממיר את הקוד משפת C לשפת אסמבלי. אסמבלי היא שפה שנמצאת רמה אחת מעל שפת מכונה וכדי לתכנת בה יש צורך לעבוד ישירות עם החומרה של המחשב. אם אתם רוצים לדעת יותר על שפת אסמבלי, תוכלו פשוט לפתוח את ספר לימוד האסמבלי של גבהים. השלב השני מבוצע על ידי תוכנה שנקראת אסמבלר. האסמבלר ממיר את הקוד משפת אסמבלי לשפת מכונה, כלומר לשפה שהמעבד מבין. בעקבות ההמרה האחרונה נוצר קובץ הרצה בעל סיומת exe (קיצור של executable). הנקודה החשובה לזכירה היא שבסוף התהליך נוצר קובץ שמכיל את כל הפקודות שכתבנו, כאשר הן מתורגמות 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 מדפיסה "Hello world", ונוסיף לה שורה חסרת משמעות – blablabla:

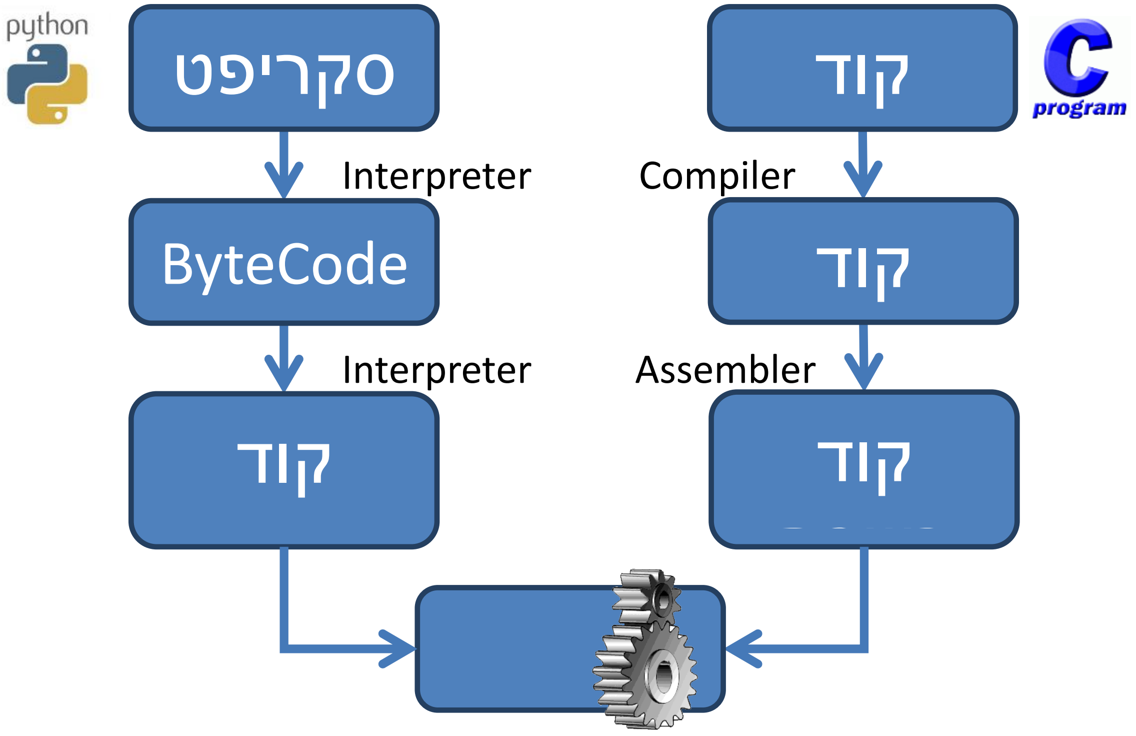
```c
#include <stdio.h>

int main()
{
    printf("Hello, World!\n");
    blablabla
    return 0;
}

```

כאשר ננסה לבצע קומפילציה, נקבל הודעת שגיאה:

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

במילים אחרות, לא נוכל להדפיס "Hello world", למרות שהשגיאה בתוכנית נמצאת בשורה שאחרי פקודת ההדפסה. הסיבה היא שהתוכנית שלנו נכשלת כבר בשלב ההמרה לשפת אסמבלי, ואינה עוברת כלל המרה לשפת מכונה.

כעת נעבור לדון במתרחש בשפת פייתון, שכאמור הינה שפת סקריפטים. על סקריפט פייתון פועלת תוכנה שנקראת interpreter ("פרשן"). ה-interpreter עובד בצורה אחרת לגמרי מאשר הקומפיילר והאסמבלר בהם משתמשים כדי לתרגם את שפת C לשפת מכונה. הוא אינו יוצר קובץ אסמבלי וגם אינו יוצר קובץ הרצה. במקום זאת, כל פקודה שכתבנו בשפת פייתון מתורגמת לשפת מכונה **רק בזמן הריצה**. תוך כדי תהליך הפירוש, נוצר קובץ עם סיומת pyc, שמכיל bytecode – הוראות שונות של ה-interpreter – אך כאמור זה אינו קובץ בשפת מכונה, כלומר, מעבד לא מסוגל להריץ את הקובץ הזה.



שפת סקריפטים (פייתון) לעומת שפת C

מדוע חשוב לנו לדעת את שלבי ההמרה? כי כעת אנחנו יכולים להבין את הניסוי הבא. הפעם, ניקח תוכנית תקינה בשפת פייתון, שמדפיסה "Hello world" ונוסיף גם לה blablabla:

```
print 'Hello world'
blablabla
```

***** שימו לב *****

דוגמאות הקוד בגרסה זו (2.0) של הספר עדיין מבוססות פייתון 2.7, בה פקודת ההדפסה היא ללא סוגריים. בפייתון גרסה 3, שהינה הגרסה שהתקנו, לאחר פקודת print יש לשים סוגריים ובתוכם להכניס את הטקסט המודפס, עם גרשיים.

לדוגמה:

print('Hello world')

בהמשך יעודכנו דוגמאות הקוד, עד אז הוסיפו סוגריים היכן שצריך.

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, שמדפיסה Hello world למסך, בוצעה בהצלחה. לאחר מכן התבצע ניסיון להריץ את השורה blablabla, ניסיון שהסתיים בשגיאת הרצה. הנקודה המעניינת היא שהתוכנית רצה באופן תקין עד שארעה השגיאה, וזאת בניגוד לתוכנית זהה בשפת C, שכלל לא רצה. הסיבה שהצלחנו להגיע בפייתון לנקודה שחלק מהתוכנית רצה, היא בדיוק בגלל תהליך ההמרה השונה. בפייתון לא נוצר קוד בשפת אסמבלי וגם לא קובץ הרצה, ולכן השגיאה לא התגלתה עד לנקודה שבה היה צריך לתרגם את blablabla חסר המשמעות לשפת מכונה. רק אז הבין ה-interpreter שיש כאן בעיה ועצר את ריצת התוכנית תוך דיווח על שגיאה.

מה אפשר להסיק ממה שלמדנו עד כה? קודם כל, ששפת פייתון היא הרבה יותר סלחנית לשגיאות מאשר שפות אחרות. שימו לב גם עד כמה הקוד בשפת פייתון קצר יותר מאשר בשפת C. לכן, הדעה הרווחת היא שקל יותר ללמוד לכתוב קוד בשפת פייתון. עם זאת, גם לשפת C יש יתרונות על פייתון: ראשית, אם נכתוב קוד לא זהיר בשפת פייתון הוא יתרסק תוך כדי ריצה. אין מנגנון כמו הקומפיילר של C, שמונע מאיתנו לכתוב קוד שלא עומד בכללי התחביר של השפה ולכן הסיכוי לבעיות בזמן ריצה הוא קטן יותר. התרסקות של קוד תוך כדי ריצה היא חמורה לאין שיעור מאשר שגיאת קומפילציה, אותה ניתן לגלות ולדבג לפני ההרצה. שנית, העובדה שקוד בשפת

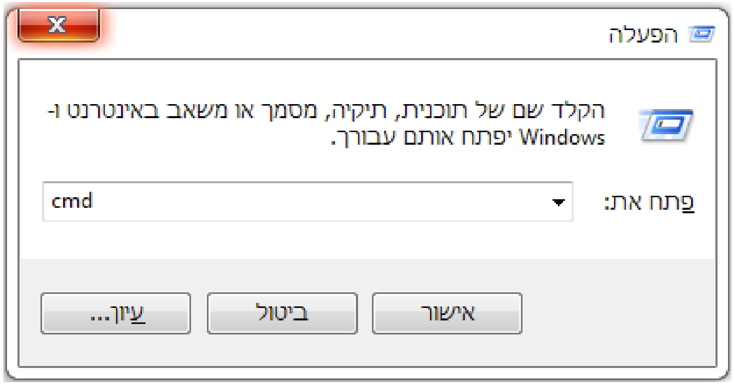
C מתורגם לשפת מכונה לא שורה אחר שורה אלא כקובץ אחד, מאפשרת לבצע תהליכי ייעול (אופטימיזציה) של הקוד, כך שהוא עשוי לרוץ יותר מהר ולצרוך פחות זיכרון מאשר קוד מקביל בשפת פייתון. זה דבר חשוב למי שרוצים להריץ אפליקציות "כבדות", כגון גרפיקה מורכבת או הצפנה.

לסיכום, אפשר לומר שפייתון היא שפה נוחה וקלה לכתיבה, שמאפשרת להגיע במהירות לתוכניות עובדות, גם אם הדבר בא על חשבון מהירות או יעילות. בשל כך זכתה פייתון למפתחים נלהבים רבים, שדאגו לכתוב מודולים – קטעי קוד שמבצעים משימות שונות – ולהפיץ אותם. כתוצאה מכך, אחד היתרונות העיקריים של פייתון הוא שניתן לקבל מן המוכן קוד בשפת פייתון שמבצע משימות רבות: חישובים מתמטיים, גרפיקה, ניהול קבצים במחשב וכל דבר שניתן להעלות על הדעת.

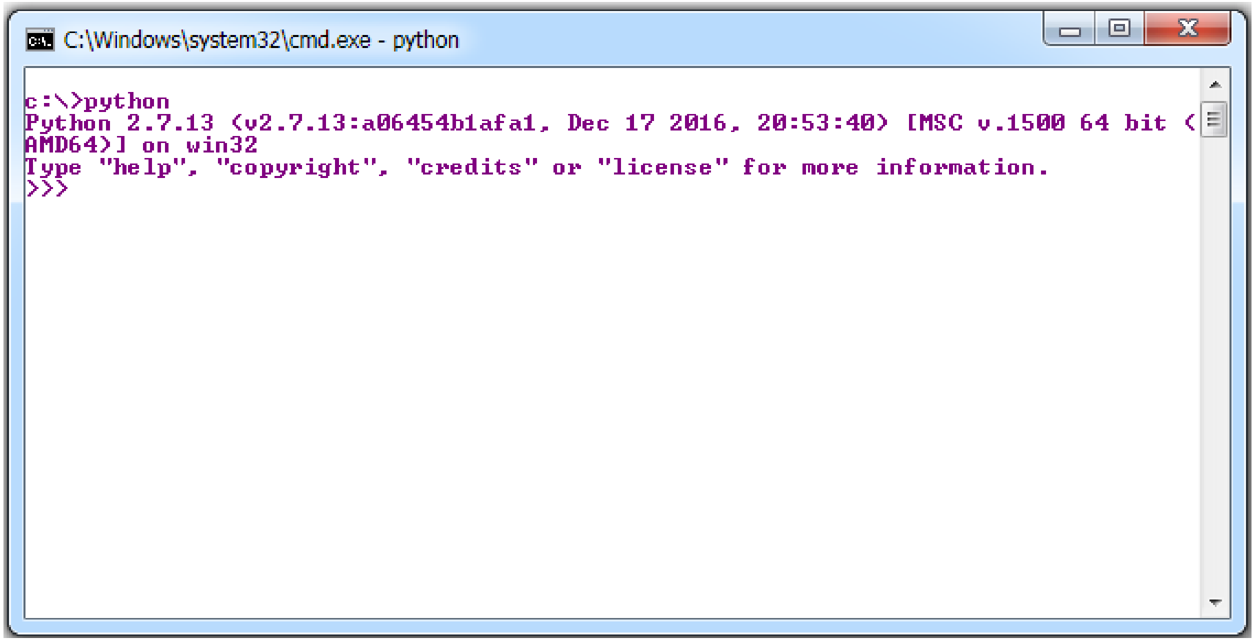
עבודה דרך command line

נפעיל את פייתון בדרך הקצרה 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":

יופיע מסך "הפעלה" או באנגלית run, ובתוכו כיתבו את שם התוכנה שאתם רוצים להפעיל. במקרה זה – cmd.



עם לחיצה על מקש ה-enter תגיעו למסך ה-command line. במסך יהיה כתוב שם התיקיה בה אתם נמצאים, לדוגמה c:\cyber. כעת כיתבו python והקישו enter. ברוכים הבאים לפייתון!



אנחנו יכולים לכתוב כל תרגיל חשבוני שאנחנו רוצים, לדוגמה 4+5, ונקבל מיידית את התשובה. אנחנו יכולים אפילו להגדיר משתנים, לדוגמה a=4 ו-b=5, וכאשר נכתוב a+b נקבל את הסכום שלהם. נסו זאת!

תרגיל מודרך

נסו לבצע את התרגיל 3/4. מה קיבלתם? כפי ששמתם לב, בחלוקה של מספרים שלמים זה בזה, התוצאה היא תמיד מספר שלם, ולכן פייתון מעגל אותה. נסו לכתוב 3.0/4.0. נסו 3/4.0. נסו 3.0/4. מה אפשר להסיק?

מספרים בבסיסים שונים

ניתן כאן הסבר מקוצר מאוד לשיטות הבינאריות וההקסדצימליות. הנושא הזה מכוסה בהרחבה בספר האסמבלי של גבהים, ואם אינכם שולטים בבסיסי ספירה מומלץ להניח מעט לספר הפייתון וללמוד בסיסי ספירה בטרם תמשיכו.

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

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

1011 0001 1000 0010

ניתן לכתוב בספרות הקסדצימליות:

B182

...הרבה 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 0x. כך לדוגמה, 0x1A הינו 26, ו-0x2B הינו 43. אפשר לראות זאת בקלות אם נכתוב אותם בסביבת הפייתון:

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

Help

תרגיל מודרך

כיתבו בפייתון 2 בחזקת 7.

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

```
>>> help()

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

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/2.7/tutorial/.

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

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

help>
```

נוכל לבחור בעזרה בנושאים הבאים: Keywords, Topics, Modules. נבקש את רשימת כל הנושאים ב-Topics ונקבל את הרשימה הבאה:

אם נכתוב POWER (המונח האנגלי לחזקה) נקבל את התיאור המלא של POWER, שם נגלה שכדי להעלות בחזקה עלינו להשתמש בסימן **. לדוגמה, 2**7 הינו 2 בחזקה 7.

כדי לצאת מ-help נכתוב quit ונגיע חזרה לחלון הפקודה שלנו. נסו כעת להשתמש במה שלמדנו!

```
help> topics

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

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

הסימן _ (קו תחתון)

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

תרגיל מודרך

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

4*(a+b)+3

קל? כעת חשבו את הביטוי הבא:

(4*(a+b)+3)**2

דרך אחת היא פשוט להעתיק את כל התרגיל מההתחלה. אולם כפי שהבחנתם, מדובר למעשה באותו תרגיל פרט לכך שהוא בחזקת 2. כדי למצוא את הפתרון באופן אלגנטי ובלי הקלדות מיותרות פשוט רישמו את הביטוי הראשון, לחצו enter ובשורה הבאה כיתבו:

_**2

פירושו של הקו התחתון – "קח את התוצאה האחרונה שחישבת". באופן זה נוכל לחסוך זמן כתיבה ולבצע חישובים בקלות. שימו לב שהטריק הזה עובד ב-interpreter, אך לא בסביבת העבודה אותה נלמד בהמשך.

הרצת תוכניות פייתון דרך ה-command line

כעת נכתוב את תוכנית הפייתון הראשונה שלנו! ראשית נצטרך תוכנת עריכה כלשהי. בתור התחלה מומלץ להשתמש ב-notepad++, אשר ניתן להורדה בחינם מהאינטרנט. לא נשתמש בו הרבה, כך שאם אתם מעדיפים לא להתקין אותו תוכלו להשתמש בתוכנת notepad שמגיעה עם חלונות. איך מפעילים? בדיוק כמו שלמדנו להפעיל כל תוכנה. ליחצו על winkey+R, כיתבו notepad בחלון ההפעלה וזהו.

את הקובץ שיצרתם שימרו עם סיומת py, כלומר קובץ פייתון. לדוגמה hello.py. כיתבו את הפקודה הבאה:

hello.py

```
print 'Hello cool cyber student!'
```

ושימרו את הקובץ.

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

```
cd c:\

cd cyber
```

לאחר שהגענו לתיקיה הנכונה, נכתוב:

```
python hello.py
```

והקובץ שלנו יורץ מיד ☺

סיכום

בפרק זה רכשנו את הבסיס לתכנות פייתון. אנחנו מבינים שפייתון היא שפת סקריפטים, אנחנו יודעים להריץ פקודות פייתון פשוטות גם דרך ה-command line וגם דרך קבצי פייתון שיצרנו. אנחנו יודעים איך לחפש בפייתון עזרה על כל נושא שנרצה. למעשה, עם הידע שקיים אצלנו בשלב זה כבר אפשר להסתדר לא רע – הרי בסופו של דבר אנחנו יודעים לכתוב תוכניות ולהריץ אותן, ואנחנו גם יודעים לחפש עזרה בנושאים לא מוכרים. מעכשיו, נוכל למצוא הדרכה על כמעט כל נושא שנלמד ולהסתדר לבד, אם נעדיף שלא לקרוא את המשך הספר...

פרק 2 – סביבת עבודה PyCharm

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

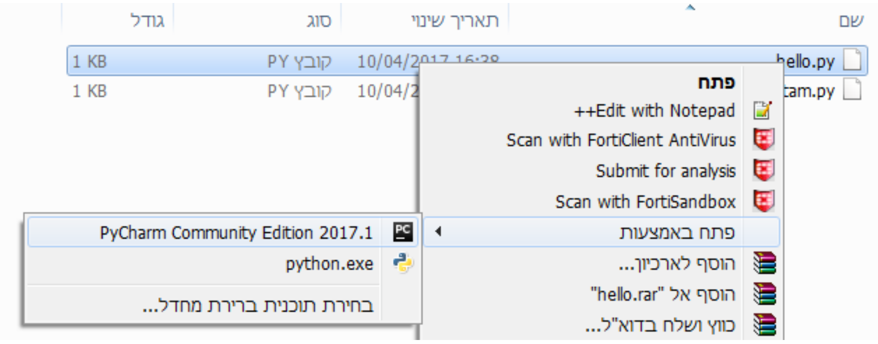
בחרנו להציג בפניכם את סביבת העבודה PyCharm. היכולות שסביבת עבודה זו מעניקה לנו הם:

- כתבן (תחליף ל-++notepad).
- איתור שגיאות אוטומטי – כן, PyCharm מוצא בשבילנו את השגיאות בקוד שלנו. אין הכוונה לכך ש-PyCharm יודע לתקן עבורנו באגים באלגוריתם, אך PyCharm בהחלט יגלה לנו אם שכחנו לשים נקודותיים, או שהשתמשנו במשתנה ששכחנו לתת לו ערך התחלתי.
- דיבאגר – כאמור, יכולת להריץ את הקוד שלנו פקודה אחר פקודה (step by step), תוך הצגת ערכי המשתנים.
- יכולת לאתחל בקלות תוכנית לאחר קריסה או באג – נוכל לחסוך זמן משמעותי כאשר התוכנית שלנו מתרסקת, על ידי כך שבמקום לאתחל את התוכנה פשוט נורה ל-PyCharm להריץ אותה שוב.

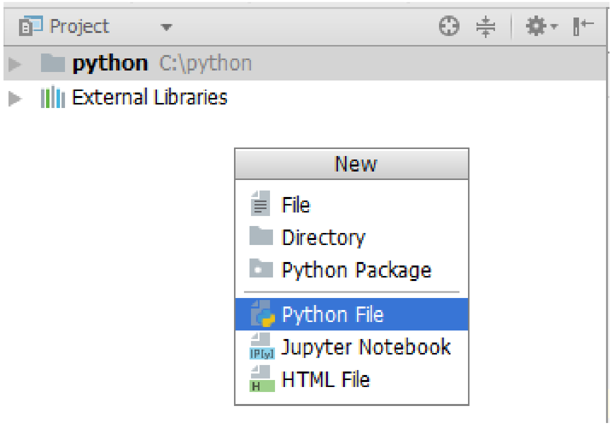
כעת נלמד להשתמש בסביבת העבודה PyCharm.

פתיחת קובץ פייתון

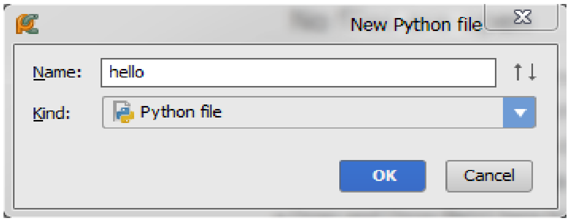
אם הקובץ כבר קיים, כל שנצטרך לעשות הוא להקליק עליו קליק ימני ולבחור באפשרות open with ולאחר מכן PyCharm Community Edition.



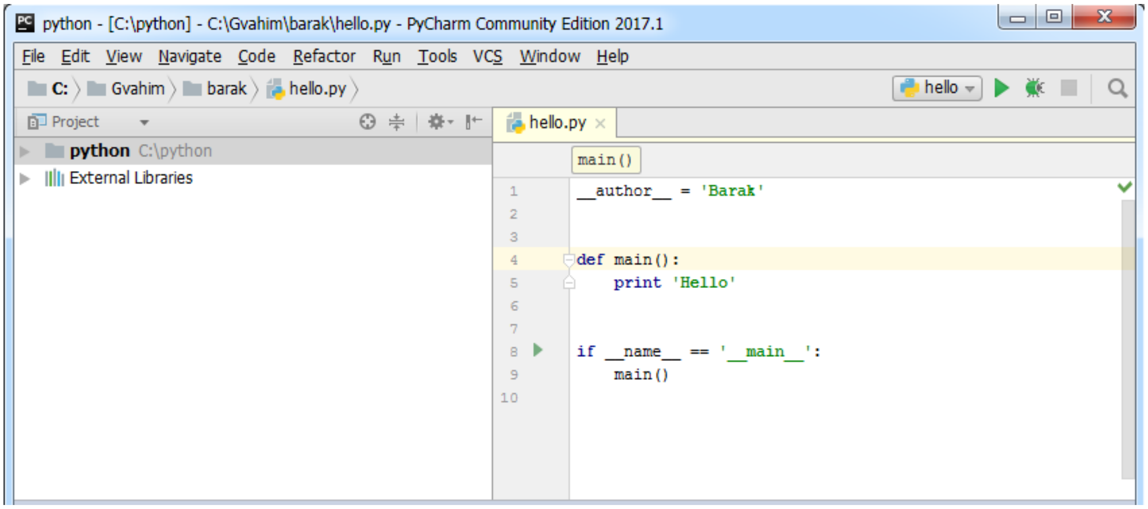
אם נרצה לפתוח קובץ חדש, אז נקליק על האייקון של PyCharm ומהתפריט נחבר ב-file, לאחר מכן new ואז מבין האפשרויות נבחר ב-python file:



לאחר מכן ניתן לקובץ החדש שם:



וכעת ניתן לערוך את הקובץ החדש שיצרנו:



קובץ הפייתון הראשון שלנו

נסקור את השורות בקובץ הפייתון הראשון שלנו:

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

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

בשורה 4 מוגדרת פונקציה בשם main. כפי שברור מהשם שלה, זו הפונקציה הראשית של התוכנית שלנו. המשמעות היא שזו הפונקציה הראשונה שנקראת (מיד נבין מי קורא לה) ולכן כל מה שאנחנו רוצים שיבוצע צריך להיות כתוב בתוך main, או לחלופין ש-main תקרא לו.

בשורה 8 ושורה 9 יש תנאי מעט מסובך, שבהמשך הלימוד נבין מה פירושו. בקצרה – אנחנו יכולים להריץ סקריפט פייתון בשתי צורות. האחת, היא פשוט להריץ אותו. השניה, היא להריץ סקריפט אחר שקורא לסקריפט שלנו. הדרך השניה מתבצעת בעזרת פקודה שנקראת import ונלמד עליה בהמשך. נניח שכתבנו סקריפט פייתון שמכיל כמה פונקציות מעניינות, וחבר שלנו רוצה להשתמש בהן. אם החבר יקרא לסקריפט שלנו על ידי פקודת import, אז עלול להווצר מצב שבו כל הקוד שכתבנו רץ. כלומר, במקום שהתוכנית שלו פשוט תכיר את הפונקציות שלנו ותוכל לקרוא להן, התוכנית של החבר פשוט מפעילה את כל התוכנית שלנו. זה כמובן לא מצב רצוי. לכן, אנחנו מוסיפים את שורות 8 ו-9 בקוד שלנו. שורות אלו אומרות לפייתון "שמע פייתון, יכול להיות שהסקריפט הזה יצורף לסקריפט אחר. לכן קודם כל תבדוק אם מי שמריץ את הסקריפט זה לא מישהו אחר שעשה לו import, ורק אם הסקריפט אינו מורץ מ-import אז תקרא לפונקציה main שתחל את ריצת הסקריפט".

סדר ההרצה של פקודות בסקריפט פייתון

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

ה-interpreter של פייתון, שתפקידו לתרגם את פקודות הפייתון לשפת מכונה, מחפש את הפקודה הראשונה שצמודה לצד שמאל ואז הוא בודק אם לא מדובר בהגדרה של פונקציה. לכן, בשורה מספר 1 – שצמודה לצד שמאל – פייתון יבצע את הפקודה. לאחר מכן בשורה מספר 4 פייתון ילמד שישנה פונקציה בשם main. פייתון יוסיף את main לרשימת הפונקציות המוכרות לו. כלומר בשלב זה אפשר לקרוא ל-main, אך אין זה אומר שפייתון מריץ את main או אפילו בודק שהקוד של main הוא תקין. כל מה שידוע לפייתון – ישנה פונקציה main.

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

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

שימו לב לכך, שזוהי אינה תוכנית פייתון 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 הפייתון. היה נכון הרבה יותר לכתוב את כל פקודות ההדפסה שלנו בתוך פונקציית ה-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 ההדפסה? בלי הפסיק, לאחר print תבוצע ירידת שורה. הפסיק קובע שלאחר ההדפסה יהיה תו אחד של רווח.

לאחר מכן תבוצע שורה 8, יודפס "Goodbye", עם ירידת שורה בסופו.

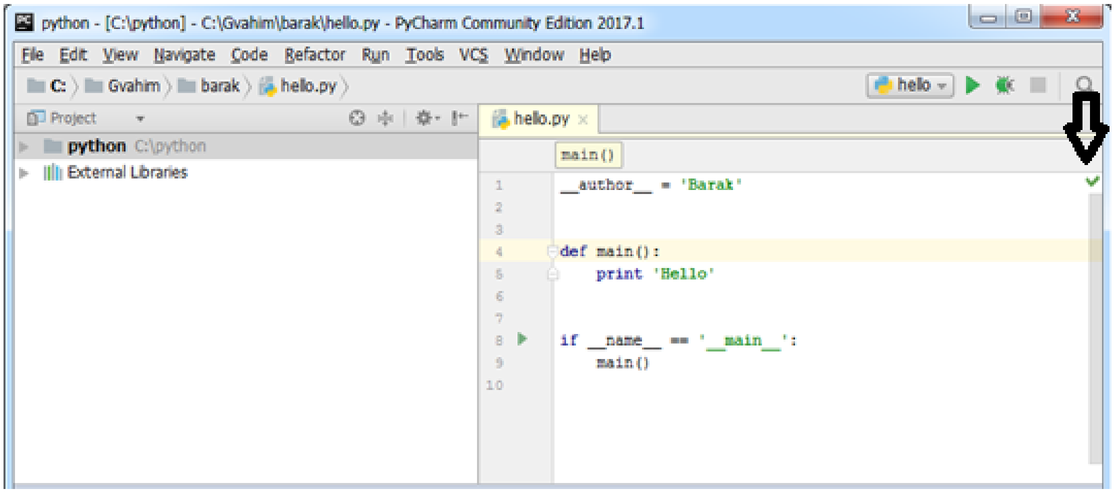
בעקבות התקיימות התנאי בשורה 10, יבוצעו שורות 11 ו-12. שורה 11 תדפיס למסך I say עם רווח. שורה 12 תקרא לפונקציה main. בתוך הפונקציה main תבוצע הדפסה של Hello.

לאחר שהפונקציה main תסיים את ריצתה, אליה היא נקראה משורה 12, היא אמורה לחזור ולהריץ את המשך התוכנית – אילו היה כזה – אחרי שורה 12. כיוון ששורה 12 היא סוף התוכנית, בכך יסיים הסקריפט את ריצתו.

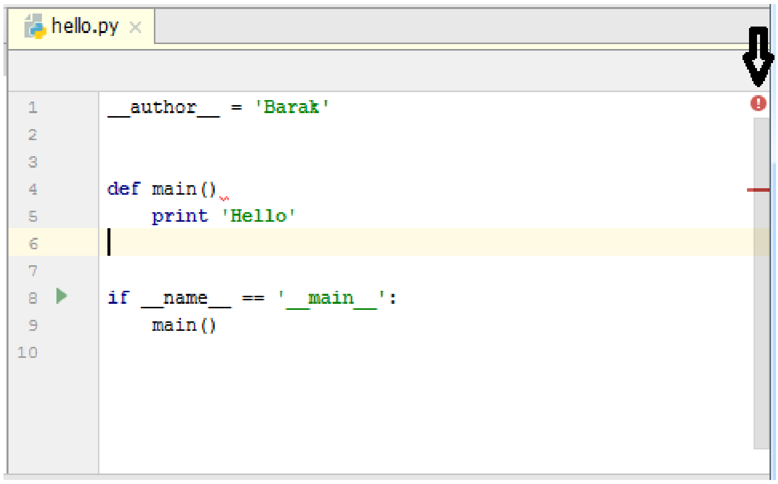
```
You say Goodbye
I say Hello
```

התרעה על שגיאות

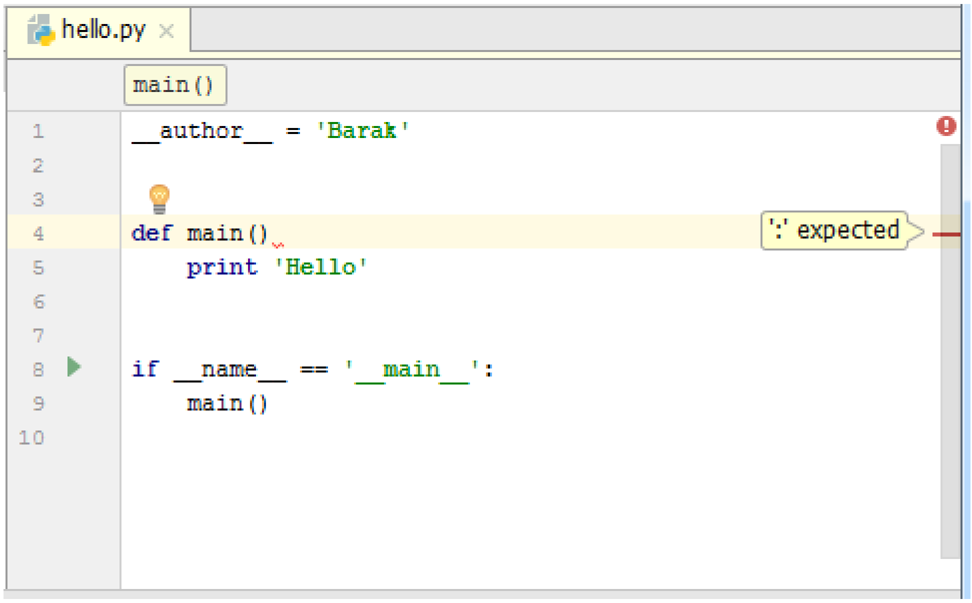
האם אתם מבחינים בסימן ה-V הירוק, שנמצא בחלק הימני העליון של PyCharm?



זהו סימן שהתוכנית שכתבנו אינה מכילה שגיאות. שימו לב מה קורה כאשר אנחנו מוחקים את סימן הנקודתיים שאחרי המילה main:



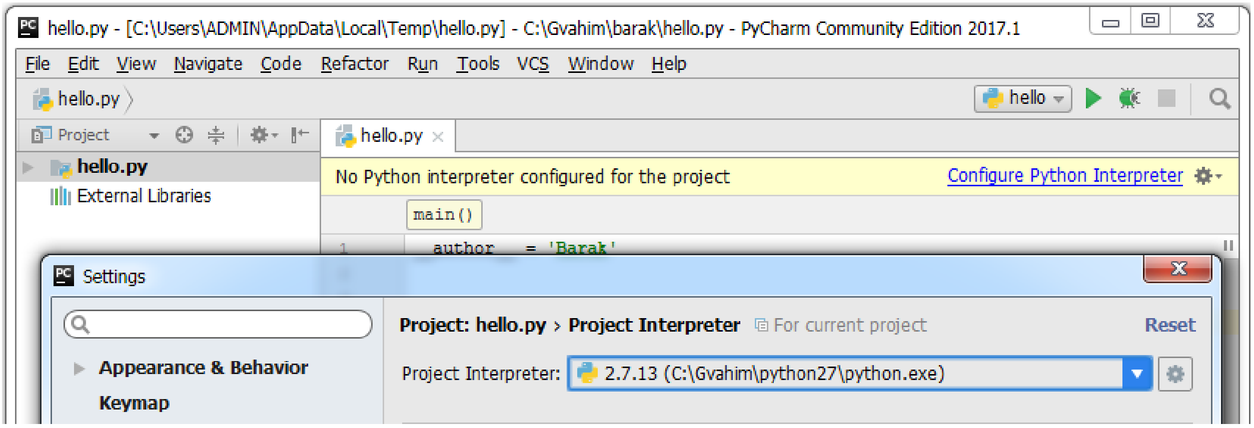
קיבלנו סימנים אדומים במספר מקומות במסך. ראשית, במקום הנקודתיים שמחקנו מופיע קו תחתי אדום. שנית, ה-V הירוק הפך לסימן קריאה לאדום. שלישית, אם נעמוד עם העכבר על הקו האדום שבצד ימין, נקבל הסבר מה הבעיה בקוד שלנו.



קליק שמאלי על הקו האדום מימין גם יוביל אותנו בדיוק לשורה הבעייתית. בקיצור, כל מה שאנחנו צריכים בשביל להימנע משגיאות קוד ולפתור אותן בקלות!

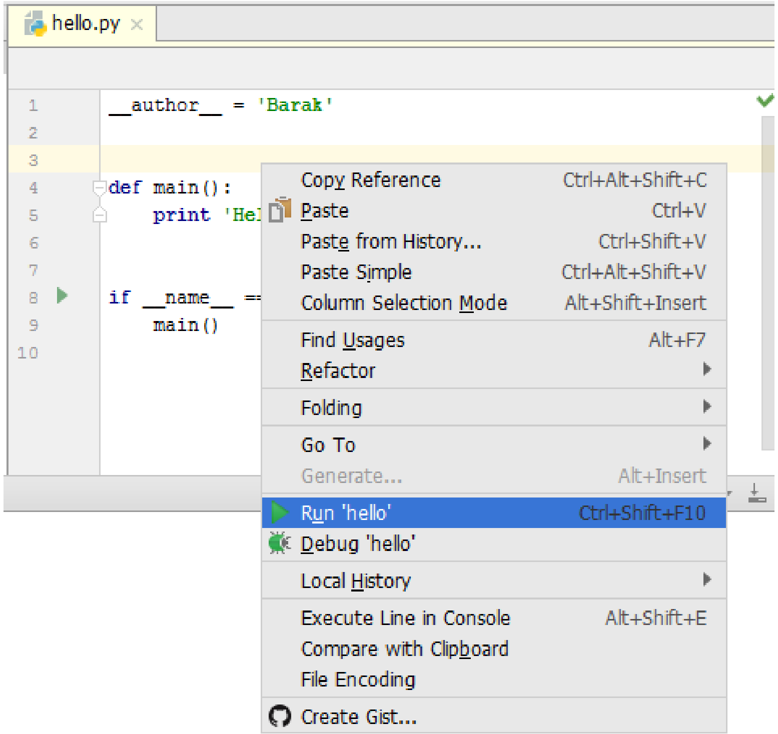
הרצת הסקריפט ומסך המשתמש

סביבת ההתקנות של גבהים אמורה להתקין ולאפשר לכם עבודה מיידית בלי צורך בהגדרות נוספות, אולם אם מופיעה לכם שגיאה מסוג "No Python interpreter configured for the project", כפי שאפשר לראות מודגש בצהוב בתמונה, להלן הדרך לסדר את ההגדרות. ראשית צריך להגדיר ל-PyCharm איפה ניתן למצוא את ה-interpreter של שפת פייתון, זאת מכיוון שיש גרסאות שונות של שפת פייתון, וייתכן שעל אותו המחשב מותקנות גרסאות שונות. לכן, נקליק הכיתוב Configure Python Interpreter ולאחר מכן נבחר את גרסת הפייתון

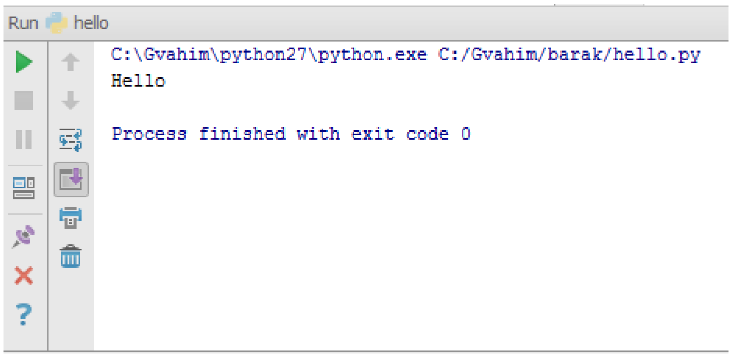


שמותקנת במחשב שלנו, אשר מגיעה עם ההתקנה של סביבת גבהים:

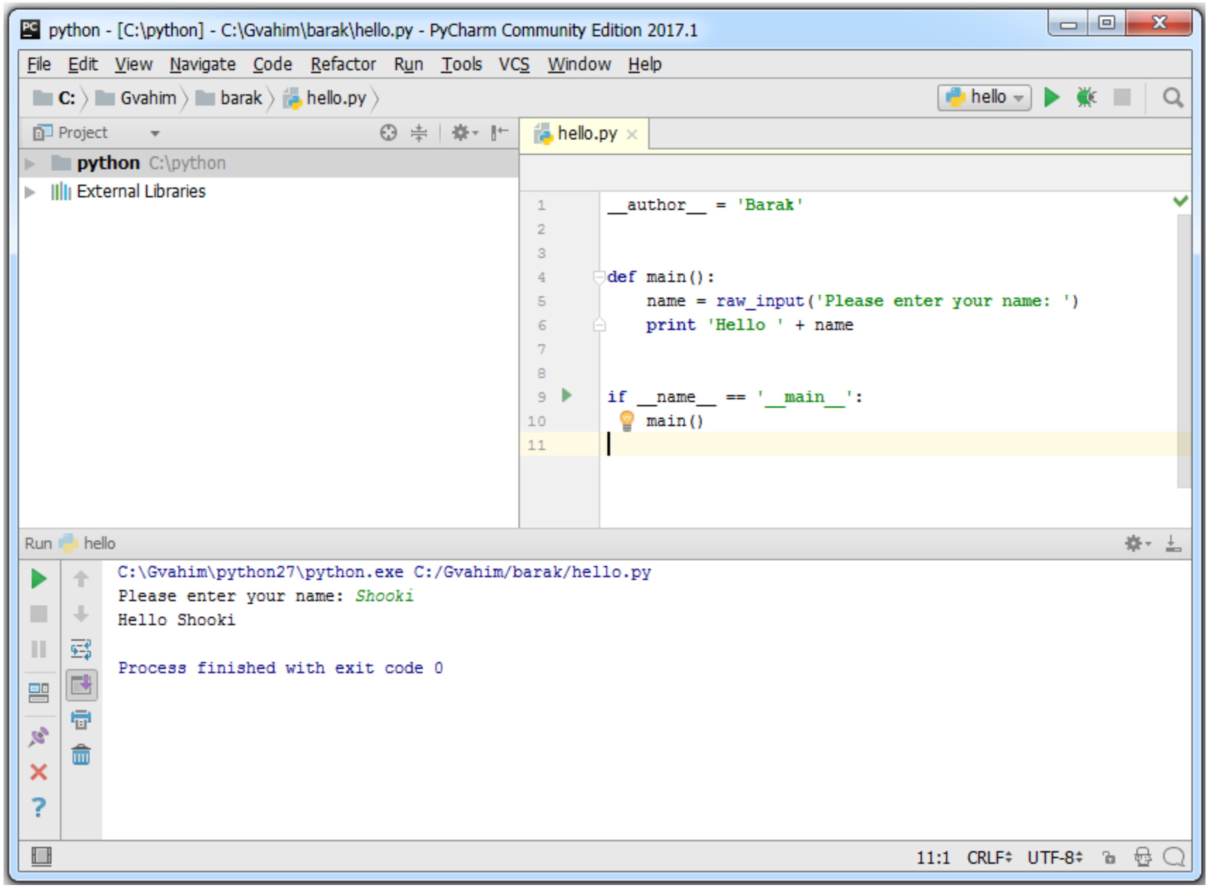
כדי להריץ את הסקריפט שכתבנו, נלחץ על עכבר ימני ונבחר באפשרות run:



בעקבות 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 שבשורה 5, raw_input:



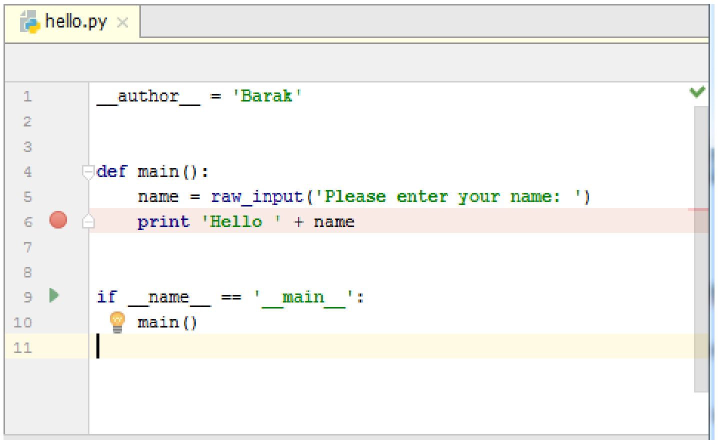
פקודה זו מדפיסה למסך את מה שנמצא בסוגריים, וכל מה שהמשתמש מקליד נכנס לתוך משתנה, במקרה זה קראנו לו name.

בשורה 6 מתבצעת הדפסה של Hello ולאחר מכן הערך שנמצא בתוך המשתנה name. התבוננו בחלק התחתון של המסך ובידקו שאתם מבינים את תוצאת הריצה.

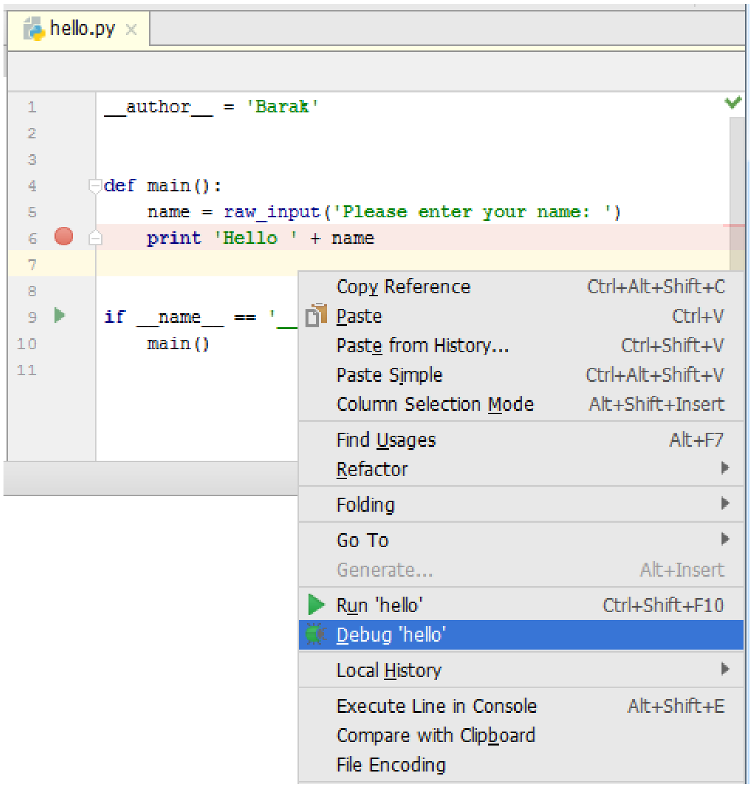
דיבוג עם PyCharm

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

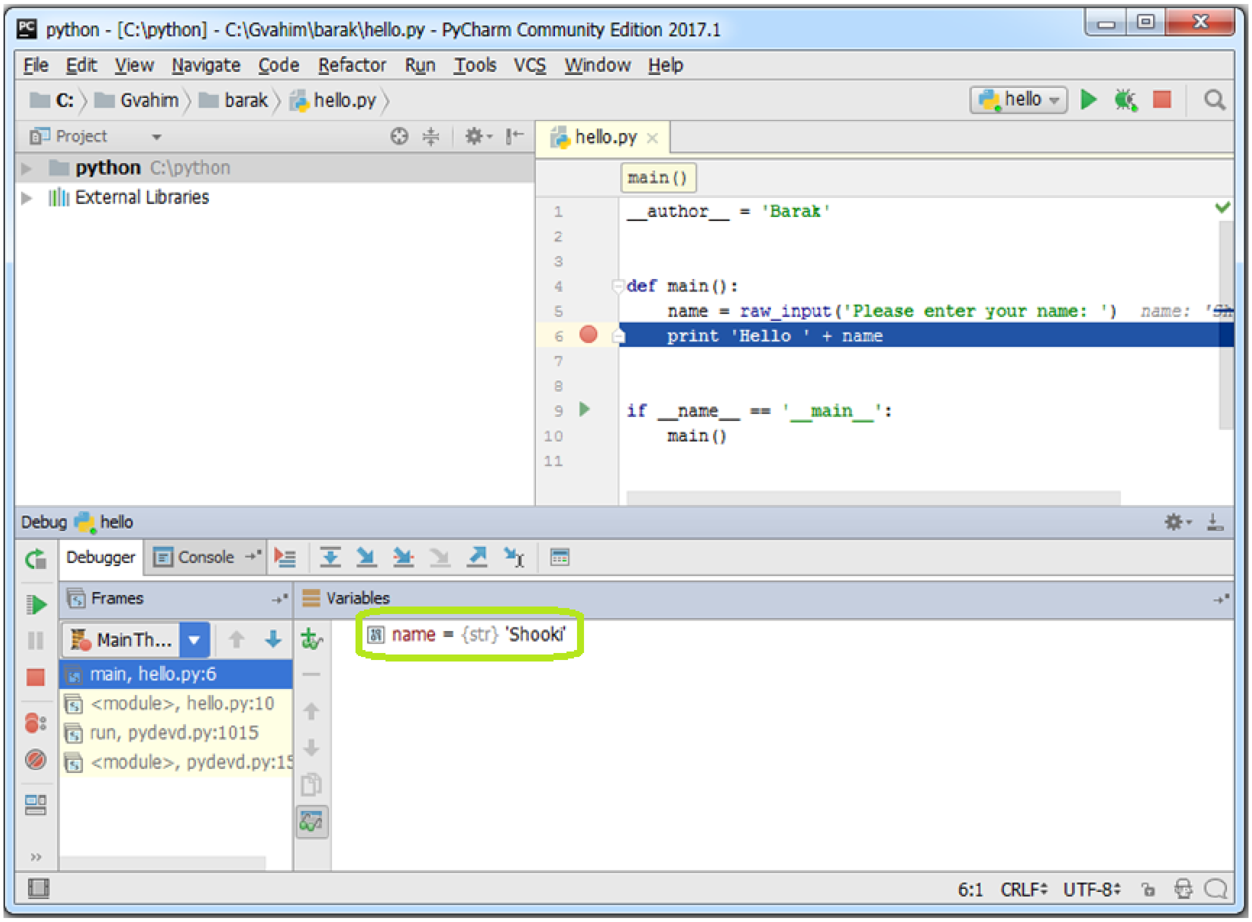
בשלב הראשון יש לשתול breakpoint בקוד. דבר זה מתבצע על ידי לחיצה שמאלית על העכבר, באזור האפור שליד מספרי השורות. בעקבות כך, תופיע נקודה אדומה ליד מספר השורה, כפי שניתן לראות ליד שורה 6 בקטע הקוד הבא:



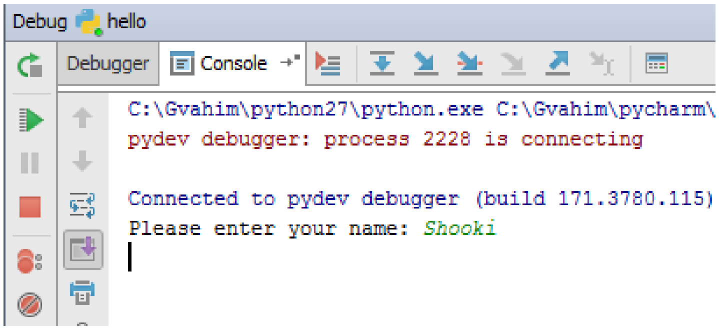
בשלב הבא צריך להריץ את התוכנית במוד debug, על ידי קליק ימני ובחירת האפשרות debug מבין האפשרויות:



לאחר מכן נראה כי התרחשו מספר דברים. ראשית, התוכנית רצה עד לשורה עם הנקודה האדומה, ה-breakpoint, ושם עצרה. השורה סומנה בכחול. שנית, נפתח לנו חלון ה-debugger, אשר מכיל מידע על המשתנים שמוגדרים בתוכנית שלנו. לדוגמה, המשתנה name הוא מסוג str (בהמשך, נבין את המשמעות של סוגי המשתנים השונים) וערכו הוא Shooki, הערך שהמשתמש הזין לתוכנו בשורת קוד מספר 5. ניתן לראות את הערך של name בחלון ה-variables, לנוחות ההתמצאות הוא מודגש במסגרת צהובה:

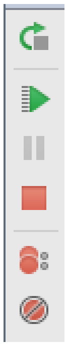


לחיצה על לשונית ה-Console תעביר אותנו אל מסך המשתמש (קלט / פלט):



נחזור אל חלון הדיבאגר. כפי שניתן לראות ישנם חיצים מסוגים שונים. לחיצה על החיצים תקדם את התוכנית שלנו. אם נעמוד עם העכבר על חץ כלשהו יופיע כיתוב שמסביר מה החץ עושה. בשלב זה מומלץ להשתמש בחץ Step Over, השמאלי ביותר. נוח מאוד להשתמש בקיצור המקלדת שלו – F8. חץ זה מריץ שורת קוד אחת בכל פעם, שורה אחר שורה. כאשר נכתוב פונקציות, נרצה להשתמש לעיתים קרובות גם בחץ השני משמאל, Step Into. קיצור המקלדת שלו הוא F7.

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

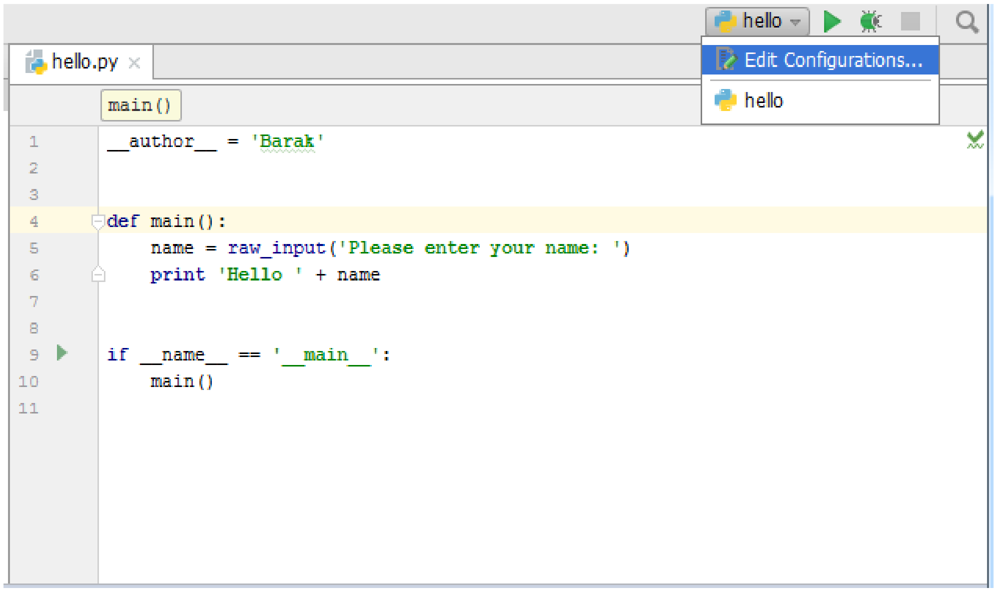


מומלץ לשלוט בפעולות אלו, מכיוון שבבוא העת שימוש נכון בהן יאפשר לכם לגלות בקלות בעיות בתוכנה שלכם!

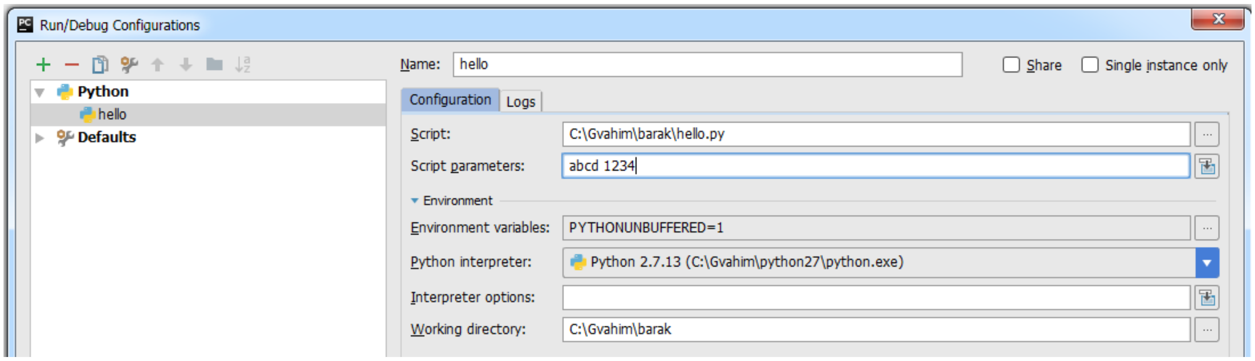
העברת פרמטרים לסקריפט

דלגו על החלק הזה, וחיזרו אליו רק לאחר שתלמדו על sys.argv.

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



כעת ייפתח מסך ובו ניתן יהיה לקבוע Script Parameters. ניתן להזין לתוכו כמה פרמטרים שנרצה, עם סימן רווח בין פרמטר אחד לשני:



סיכום

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

פרק 3 – משתנים, תנאים ולולאות

סוגי משתנים בפייתון

בפרק הראשון ראינו איך יוצרים משתנה בשם a שערכו שווה למספר שלם. סוג המשתנה שמכיל מספר שלם 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 סוגי משתנים רבים, לא רק int כמובן, ואנחנו נסקור אותם אחד אחד, כאשר בפרק זה נתחיל מטיפוסי המשתנים הפשוטים ביותר ובפרקים הבאים נכסה טיפוסי משתנים נוספים.

שימו 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'>
```

<u>משתנה בוליאני bool</u>: משתנה בוליאני יכול לקבל שני ערכים בלבד – אמת או שקר, True או False. מגדירים משתנה בוליאני באופן הבא:

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

שימו לב שחייבים לשמור על האות הגדולה בתחילת המילה, אם נכתוב true או false נקבל שגיאה. מכאן אפשר להסיק שפייתון היא שפה שמבדילה בין אותיות גדולות לקטנות (מה שנקרא "case sensitive"). כלומר, המשתנה a שונה מהמשתנה A.

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

True + 1

פייתון ידפיס לנו את התשובה – 2.

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

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

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

שימו לב לכך שיש במספרים מסוג float אי דיוק זעיר, שנובע מכך שבסופו של דבר הם נשמרים בזיכרון המחשב באמצעות חזקות של 2, ויש כמות מוגבלת של ביטים שבה נשמר כל משתנה. לכן, רוב המספרים מסוג float נשמרים עם "עיגול" מסויים. לדוגמה, אין דרך לייצג את התוצאה של 1 חלקי 7 באמצעות כמות סופית של ביטים. אי הדיוק הזה גורם לכך שאם נעשה פעולות חשבוניות שיגרמו להצטברות השגיאה, לבסוף נקבל תוצאה שיהיה ברור שהיא אינה מדוייקת. הנה, כאשר נחבר 0.1 עם עצמו מספר פעמים, ניווכח שהתוצאות אינן "עגולות" כפי שהיה 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 – String, List, Tuple, Dictionary – 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. בינתיים, נסקור את הדרכים השונות לבדוק את היחס בין משתנה לביטוי כלשהו.

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

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

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

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

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

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

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

```
x = True
if x is True:
    print 'The truth!'
```

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

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

תנאים מורכבים

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

תנאי and משמעו "וגם". לדוגמה, התנאי הבא יתקיים רק אם מתקיים גם x גדול מ-20 וגם x קטן מ-22:

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

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

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

כמובן שאת 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-is

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

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

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

בסירטון הבא יש הדגמה של ההבדל בין is ל == :

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

בלוק

בדוגמאות שסקרנו תמיד היתה שורת קוד אחת לאחר תנאי ה-if. האם אפשר לשים יותר משורת קוד אחת? כמובן! את מושג הבלוק הכי פשוט להבין מצפייה בקוד:

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

כל שלוש הפקודות שלאחר ה-if נמצאות בהזחה – אינדנטציה – של טאב אחד, או ארבעה רווחים יחסית לתנאי ה-if. כיוון שכולן נמצאות באותה הזחה, הן יבוצעו כולן אם התנאי יתקיים. שימו לב להבדל בין הקוד לעיל לבין הקוד הבא:

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

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

נושא ההזחה הוא קריטי בפייתון, מכיוון שבניגוד לשפות אחרות בהן יש סימונים שונים של "סוף בלוק", לדוגמה סוגריים מסולסלים, בפייתון סוף הבלוק נקבע רק לפי ההזחה. על כן הכרחי גם שההזחה תהיה עקבית – אנחנו לא יכולים לעשות לפעמים הזחה של שני רווחים ולפעמים של ארבעה רווחים. חשוב לזכור גם שתווי רווח וטאב שונים זה מזה. כלומר, גם אם לנו נראה ששורה שיש בה טאב נמצאת בהזחה זהה לשורה אחרת שיש בה הזחה של ארבעה תווי רווח, מבחינת פייתון אלו תווים שונים לגמרי. לכן, אם התחלנו לעשות הזחה עם טאבים, רצוי שנמשיך רק עם טאבים.

לא רק מה שנמצא בהזחה של טאב אחד שייך לבלוק. בתוך בלוק יכולים להיות עוד טאבים, לדוגמה עקב שימוש בתנאי if נוספים. כל הפקודות שנמצאות בהזחה של <u>לפחות</u> טאב אחד מתנאי ה-if שלנו שייכות לאותו בלוק. לדוגמה:

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

שימו לב לכך ש-PyCharm מסייע ו"מסמן" את הבלוקים השונים באמצעות פסים אפורים דקים. מה יודפס כתוצאה מריצת הבלוק? יורצו כל השורות חוץ מאשר אלו שנמצאות בבלוק של if x < 21, כיוון שתנאי זה אינו מתקיים. פלט התוכנית יהיה:

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

תנאי else, elif

נניח שאנחנו רוצים להריץ קוד כלשהו אם תנאי מתקיים, וקוד אחר אם התנאי אינו מתקיים. במקום שנצטרך לבדוק פעמיים – 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 שאנחנו רוצים לבדוק? לדוגמה, אנחנו מתקשרים לחבר כדי לבוא איתנו לסרט. אם הוא לא יכול, אנחנו מתקשרים לאמא ושואלים מה יש לאכול בבית. אם אמא לא עונה – אנחנו נשארים בבית וצופים בשידור 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 שבסוף אכלנו עוגת תפוחים של אמא. ברגע שתנאי ה-elif התקיים, התוכנית לא נכנסה אל תוך תנאי ה-else. נצפה בטלויזיה רק אם גם תנאי ה-if וגם תנאי ה-elif לא יתקיימו.

תרגיל

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

לולאת while

עד כה ראינו דרכים לכתוב תנאי "חד פעמי", כלומר תנאי שפייתון מריץ פעם אחת בלבד. לעיתים נרצה לבצע פעולה כלשהי כל עוד תנאי מסויים מתקיים. לדוגמה, נרצה לקרוא קלט מהמשתמש ולבצע את הוראות המשתמש, כל עוד המשתמש לא כתב 'Exit'. במקרים אלו שימוש ב-if ו-else הוא לא מספיק טוב, כיוון שאנחנו לא יכולים לדעת כמה פעמים התנאי שלנו צריך לרוץ. אולי המשתמש כתב 'Exit' כבר בהוראה הראשונה? אולי בהוראה העשירית? במקרים אלו נשתמש בלולאת while.

לולאת while מתחילה כצפוי בהוראה while, לאחריה יבוא תנאי אותו נגדיר. לאחר מכן יש בלוק של פקודות אשר יבוצעו בזו אחר זו, ובסוף הבלוק תהיה חזרה אל בדיקת התנאי.

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

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

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

טעות נפוצה היא לומר שלא 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 הבלוק מריצים את כל הפקודות שלו. הבדיקה של ה-while מתבצעת פעם אחת, בכניסה לבלוק, ולא כל פקודה מחדש. לכן המשפט Not yet 18 יודפס פעם אחת. באיטרציה השניה של הלולאה התנאי כבר לא יתקיים ולא תהיה כניסה לתוך הבלוק.

מה לדעתכם יקרה אם נריץ את הלולאה הבאה?

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

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

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

שורה 2 תדפיס Hi. בשורה 3 אנחנו מורים למחשב לצאת מהלולאה, לכן 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?

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

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

תרגיל – Take a Break

הדפיסו את כל המספרים בסדרת פיבונצ'י אשר ערכם קטן מ-10,000. חובה להשתמש ב-while True! 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

לולאות for

לולאות for הן שימושיות במיוחד כאשר יש לנו אוסף של איברים שאנחנו רוצים לבצע עליהם פעולה כלשהי, בניגוד ללולאות while שהן שימושיות כאשר מריצים מספר לא ידוע של פעמים. בשפת פייתון, לולאות for נכתבות בצורה מעט שונה משפות תכנות אחרות.

בפייתון, לולאת 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 נטען לתוך משתנה שעליו רצה הלולאה. לאחר סיום האיטרציה, נטען האיבר הבא מאוסף האיברים וכך הלאה עד סיום כל האיברים באוסף.

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

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

פלט ההרצה 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 לכתוב את כל המספרים אחד אחרי השני... הפונקציה range מייצרת סדרות של מספרים. היא מקבלת בתור פרמטרים התחלה, סוף וקפיצה ויוצרת סדרת מספרים. לדוגמה:

range(3, 5, 1)

תיצור סדרה של מספרים אשר מתחילים 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 ההרצה:

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

הסבר: השורה הראשונה כאמור יוצרת רשימה של מספרים שמתחילים ב-3 ועוצרים לפני 8, בקפיצות של 1. השורה השניה 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 ולהתקדם בקפיצות של 1. במילים אחרות, יש לפונקציה ערכי ברירת מחדל. אם לא מוזנת קפיצה, אז נעשה שימוש בקפיצה של 1. אם לא מוזנת התחלה, מתחילים מאפס.

ישנה פונקציה דומה לפקודת range, אשר נקראת 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 למיליארד. אתם כותבים קוד שבודק אם מספר הוא ראשוני, ומכניסים אותו לתוך לולאת for שרצה על range(1000000000). מה שיקרה הוא שתיווצר רשימה של מיליארד מספרים בזיכרון המחשב. לא דבר טוב בכלל, כיוון שזה צפוי להאט את עבודת המחשב. לעומת זאת, אם נעשה את אותה לולאת for עם xrange(1000000000) אז נקבל את אותם המספרים אבל הם יתקבלו בלי להישמר בזיכרון המחשב. אפשר לסכם ולומר שפקודת range היא שימושית כאשר נרצה ליצור סדרת מספרים שישמשו אותנו מעבר ללולאת for. בשביל לולאת for בלבד, מוטב להשתמש ב-xrange.

תרגיל (קרדיט: עומר רוזנבוים, שי סדובסקי)

כיתבו לולאת for שמדפיסה את כל המספרים מ-1 עד 40 (כולל).

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

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

14 % 4

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

pass

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

לדוגמה:

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

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

תרגיל מסכם (קרדיט: עומר רוזנבוים, שי סדובסקי)

הדפיסו 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
...
...
5
```

פרק 4 – מחרוזות

הגדרת מחרוזת

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

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

והתוצאה היא זהה בשני המקרים.

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

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

נקודה נוספת שנוגעת למחרוזות היא שנוטים לפעמים לבלבל מחרוזות של ספרות עם המספר עצמו. לכן נבהיר זאת – המחרוזת '1234' אינה שווה למספר 1234. הבלבול נובע מכך שאם נעשה print בשני המקרים נקבל אותו הדבר – יודפס למסך 1234, אולם הסיבה לכך היא שכאשר מבקשים להדפיס מספר, פייתון מתרגם אותו מאחורי הקלעים למחרוזת ואז מדפיס אותה. ההבדל בין המחרוזת למספר יתבהר 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 רצה בהצלחה והדפיסה את תוצאת החישוב למסך. הפקודה השניה כבר לא. קיבלנו הודעת שגיאה שאומרת "לא ניתן לחבר משתנה מטיפוס str עם משתנה מטיפוס int". זיכרו זאת בהמשך ☺

ביצוע format ל-print

עד כה בכל פעם שרצינו להדפיס משהו, היינו כותבים פקודת print ולאחריה את מה שרצינו להדפיס. זו שיטה מאוד נוחה כשרוצים להדפיס ערך יחיד, אבל היא נהיית מעט מסורבלת כשרוצים להדפיס אוסף של ערכים. דבר זה נכון במיוחד אם חלק מהערכים הם מסוג str וחלק מהם מטיפוסים אחרים, כגון int. נכון, אפשר להדפיס משהו שכולל str ו-int בצורה הבאה, על ידי המרה של ה-int ל-str, אבל כאמור זה קצת מסורבל:

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

פקודת 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. במלים אחרות, זהו פתרון אפשרי אבל לא נוח.

נוכל להשתמש בפקודת format, ובמקום כל משתנה שאנחנו רוצים להדפיס, נשים סוגריים מסולסלים. לאחר מכן נכניס הכל לתוך format. לדוגמה:

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

או לדוגמה:

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

חיתוך מחרוזות – string slicing

אפשר לחתוך חלקים ממחרוזות באופן הבא:

my_str[start:stop:delta]

החיתוך דומה לפונקציית range אותה הכרנו כבר.

אינדקס ההתחלה הוא start, ברירת המחדל שלו היא 0.

אינדקס הסיום הוא stop, ברירת המחדל היא סוף המחרוזת.

הפרמטר delta מציין בכמה אינדקסים קופצים, ברירת המחדל היא 1.

שימו לב שגם ערכים שליליים מתקבלים באופן תקין! לדוגמה, אם נגדיר מחרוזת אז נראה שלכל 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, כלומר אחורה. 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?

השוו את מה שחשבתם שיתקבל עם התוצאות הבאות:

```
h
hr
he
Shrek
Shre
kerhS
```

הסבר:

שורה 2 מדפיסה את האיבר באינדקס מספר 1 במחרוזת. שימו לב שזהו התו השני, כיוון שהאינדקסים מתחילים מ-0.

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

שורה 4 מדפיסה את האיברים שמתחילים מאינדקס 1, מסתיימים בברירת המחדל (סוף המחרוזת) ומתקדמים בקפיצות של 2.

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

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

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

-1, כלומר אחורה. במילים אחרות, מתחילים מהאיבר האחרון והולכים אחורה עד שמגיעים לאיבר הראשון.

תרגיל (קרדיט: עומר רוזנבוים, שי סדובסקי)

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

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

פקודות על מחרוזות

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

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

נציג את פלט התוכנית לפני שנעבור לדון בכל שורת קוד:

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

בשורה 1 מודגם החיבור של שתי מחרוזות באמצעות +.

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

בשורה 3 אנחנו פוגשים את המתודה upper. מתודות הן כמו פונקציות, אבל של סוג משתנה ספציפי. בהמשך נלמד שמתודה היא בעצם פונקציה שמוגדרת בתוך מחלקה, אבל נשמור זאת לפרק שדן במחלקות. בכל 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 סוגריים את הפרמטר:

```
len(message)
```

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

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

dir, help

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

ובכן, נכיר את הפונקציה המובנית dir. פונקציה זו מחזירה לנו את כל המתודות של משתנה. לדוגמה, אם נגדיר מחרוזת בשם message ונעשה לה dir, נקבל את התוצאה הבאה:

```
>>> message = 'Hello World'
>>> dir(message)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__', '__for
mat__', '__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', '
translate', 'upper', 'zfill']
```

נחמד, כעת אנחנו יודעים מה שמות כל המתודות שניתן להריץ על מחרוזת. אבל כיצד נוכל לדעת, לדוגמה, מה עושה המתודה 'count'? כאן מגיעה לעזרתנו הפונקציה המובנית help. נבצע help על המתודה count של message ונקבל את התיעוד של המתודה:

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

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

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

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

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

הדבר החשוב שלמדנו אינו המתודה 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 '\' הוא תו מיוחד אשר קרוי "escape character" והוא מאפשר ליצור מחרוזות עם צירופי תווים מיוחדים. ישנם תוים, שאם נשים אותם אחרי ה-'\' הם לא יודפסו כמו שהם, אלא יקבלו משמעות אחרת. אחד הצירופים הידועים ביותר הוא \n. אם נכתוב \n במחרוזת 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*
\\x*hh...*	ASCII character with hex value *hh...*

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

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

```
c:\cyber
umber52
```

לא בדיוק מה שרצינו שיודפס...

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

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

ההכפלה אומרת לפייתון 'אנחנו באמת מתכוונים לסימן \ '.

האפשרות השניה היא לכתוב לפני תחילת המחרוזת את התו r, שמגדיר לפייתון שהכוונה היא ל-raw string. כלומר, עליו לקחת את התווים כמו שהם ולא לנסות לחפש צירופי תווים בעלי משמעות.

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

קבלת קלט מהמשתמש – raw_input

את הפונקציה raw_input הכרנו בקצרה בפרק הקודם. פונקציה זו משמשת 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)
```

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

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

תרגיל – אבניבי

זוכרים את שפת הבי"ת? כיתבו קוד שקולט משפט (באנגלית) מהמשתמש ומתרגם אותו לשפת הבי"ת. תזכורת: אחרי האותיות aeiou צריך להוסיף b ולהכפיל את האות. לדוגמה, עבור הקלט ani ohev otach יודפס:

Abanibi obohebev obotabach

טיפ לייעול הקוד: כדי לדעת אם תו נמצא במחרוזת ניתן להשתמש בפקודה in. לדוגמה:

if 'a' in my_str:

הלילה בו זכתה ישראל בתחרות האירוויזיון

„זה משגע", מילמל בלילה הזמר יזהר כהן בפריז - לאחר שלהקתו ניצחה ב„אירוויזיון"

א־ב־ני־בי

השיר

תרגילי strings (מתוך google class, בעיבוד גבהים)

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

לינק להורדת קובץ הפייתון 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 שכוללת את מספר ה-donuts שקיבלנו כפרמטר לפונקציה (בתוך המשתנה count), אולם אם count הינו גדול מ-10, עלינו להחזיר פשוט 'many'.

פתרון:

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

תוצאת ההרצה:

```
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 (קרדיט: 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

- את סכום הספרות של המספר

דוגמה לריצה תקינה:

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

הדרכה: בתרגיל זה אנחנו נדרשים לבצע המרה בין סוגי טיפוסים שונים. זיכרו, שהפונקציה raw_input מחזירה מחרוזת של תווים. כלומר אם המשתמש הזין 1234, תוחזר המחרוזת '1234'. אם אנחנו מעוניינים להשתמש במחרוזת כמספר, עלינו לבצע קודם לכן המרה מטיפוס מחרוזת לטיפוס int. לשם כך נשתמש בפונקציה int. לדוגמה:

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

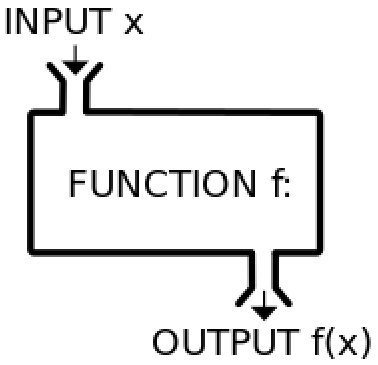
הערה: ניתן להניח שהמשתמש הזין קלט תקין.

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, וכמובן שאפשר גם לקבל חזרה ערכים מהפונקציה.

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



בואו נכתוב את הפונקציה הראשונה שלנו, שמדפיסה Hello:

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

מתחילים עם המילה השמורה def. לאחר מכן שם הפונקציה, ולאחר מכן סוגריים שבתוכם אפשר לשים פרמטרים.

שימו לב לדגשים הבאים:

- מומלץ מאוד לתת לפונקציה שם בעל משמעות, כך שגם מי שקורא את הקוד שלנו יבין מה רצינו.
- בתחילת הפונקציה יש לכתוב תיעוד, שמתאר מה הפונקציה עושה. לתיעוד בתחילת פונקציות יש שם מיוחד – docstring, קיצור של documentation string, מחרוזת תיעוד. כותבים docstring בתוך שלושה סימני גרשיים כפולים רצופים, כמו בדוגמה. מדוע זה חשוב? מכיוון שאם נעשה help על הפונקציה, נקבל בחזרה את ה-docstring שלה. דבר מועיל מאוד כאשר אנחנו רוצים לשתף קוד!

help(hello)

```
Help on function hello in module __main__:

hello()
    Print Hello
```

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

my_func()

דוגמה לפונקציה עם פרמטר יחיד:

my_func(x)

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

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


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

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

כעת נראה איך 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 נקראה עם ארגומנט בשם message, שזהה לשם הפרמטר כפי שמוגדר בתוך הפונקציה. מאידך, הפונקציה print_messages נקראה עם ארגומנטים בשם mes ו-sage, שאינם זהים לשמות הפרמטרים שמוגדרים בתוך הפונקציה – msg1 ו-msg2. שתי האפשרויות חוקיות. בתוך הפונקציה print_messages מתבצעת העתקה של הארגומנט mes לתוך הפרמטר msg1 ושל הארגומנט sage לתוך הפרמטר msg2.

```
print_messages(mes, sage)

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

המחשה של העברת ארגומנטים לפונקציה עם פרמטרים בעלי שמות שונים

return

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

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


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

השורה print a 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 11 7.

תרגיל

- כיתבו פונקציה אשר מקבלת שני ערכים ומחזירה את המכפלה שלהם.

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

None

הבה נבחן את הפונקציה הבאה:

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

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

k = stam()

נסו לעשות print k ותקבלו שהערך של k הוא None. הערך None, או "כלום", הוא ערך ריק. זוהי מילה שמורה בפייתון. משתנה יכול להיות שווה לערך ריק, ואז הוא שווה None.

יש 3 מצבים בהם פונקציה מחזירה ערך None:

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

scope של משתנים

מה תבצע התוכנית הבאה?

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


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

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

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

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

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

נחדד את ההסבר ותוך כדי נבין את ההבדל בין משתנה גלובלי למשתנה לוקלי, מקומי. נניח שכתבנו את הקוד הבא, שימו לב למשתנה החדש name:

```
name = 'Shooki'


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


def main():
    speak()
```

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

נתקדם עוד שלב – כעת אנחנו מגדירים את המשתנה word פעמיים. אחת כגלובלי ואחת כלוקלי... מה ידפיס הקוד הבא?

```
word = 'bye'


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


def main():
    speak()
```

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

ניסיון נוסף... מה ידפיס הקוד כעת?

```
word = 'bye'


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


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

יודפס:

```
hi
bye
```

מדוע? את ההדפסה של hi כבר הבנו. כאשר speak מסיימת את הריצה שלה, המשתנה הלוקלי word נמחק, וכעת קורה משהו מעניין – מסתבר שהמשתנה הגלובלי word לא נמחק, אלא פשוט נשמר בצד. לפייתון יש מידרג של עדיפויות: כאשר פונים למשתנה בתוך פונקציה, קודם כל פייתון מחפש אם קיים משתנה לוקלי כזה, ולאחר מכן אם קיים משתנה גלובלי. ההדפסה השניה מתרחשת מתוך הפונקציה main, שמכירה רק את המשתנה הגלובלי word.

ניסיון אחרון... 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 שישנה שגיאה בקוד...

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

המשתנה word אינו מוגדר. אבל מדוע? הרי הגדרנו משתנה גלובלי בשם זה? הסיבה היא, שכאשר אנחנו מבצעים פעולה שמשנה את ערכו של משתנה בתוך פונקציה, כמו פעולת חיבור, פייתון מניח שלמשתנה שלנו יש עותק מקומי והוא מנסה לפעול עליו.

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

```
word = 'love'


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


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

בעקבות הריצה יודפס:

```
love you
love you
```

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

האפשרות השניה, 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 הריצה יודפס:

```
love you
love
```

כעת, מדוע האפשרות הראשונה אינה מומלצת? כפי ששמתם לב, האפשרות הראשונה משנה את ערכו של המשתנה word גם מחוץ לפונקציה. הודפס פעמיים love you. כלומר, הפונקציה שינתה את ערכו של המשתנה. דמיינו שאתם כותבים את הפונקציה main ומתכנת אחר כותב את הפונקציה speak. כעת תארו לעצמכם את ההפתעה, שהפונקציה שינתה ערך של משתנה בלי שיהיה לכם מושג שהיא עשתה זאת! אם הייתם רוצים לאפשר לפונקציה לשנות את הערך של המשתנה, הייתם מעבירים לה אותו כפרמטר ודואגים להציב את ערך החזרה של הפונקציה במשתנה, כך:

word = speak(word)

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

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


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

תרגילים

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

פייתון מתחת למכסה המנוע (הרחבה)

בחלק זה נפרט שני נושאים הקשורים לפרקי הלימוד עד כה:

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

כזכור פייתון היא שפת סקריפטים, ולכן לא מתבצעת קומפילציה לקוד. במילים אחרות, כל שורת קוד שאנחנו כותבים מומרת לשפת מכונה רק כאשר מגיע תורה של שורת הקוד להיות מורצת. נמחיש את העיקרון הזה בעזרת תוכנית קטנה.

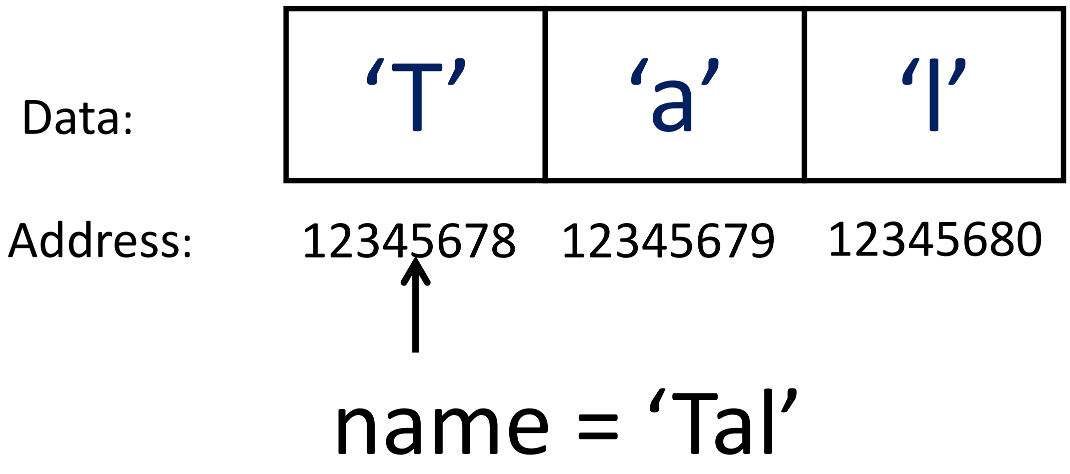
```
def check(num1):
    if True:
        print 'OK'
    else:
        bla(blabla)


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

כפי שאפשר לראות, הקוד מכיל קריאה לפונקציה בשם bla עם ארגומנט blabla. הן הפונקציה והן הארגומנט אינם מוגדרים. למרות זאת, אם נריץ את התוכנית נקבל תמיד 'OK'. הסיבה לכך היא שתנאי ה-if מתקיים תמיד, ולכן הקוד שבתוך תנאי ה-else לעולם אינו מגיע להיות מורץ. זוהי המחשה של העובדה שבפייתון כל שורת קוד מפורשת ומתורגמת לשפת מכונה רק כאשר מגיע זמנה להיות מורצת.

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

id, is

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



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

דוגמה לשימוש ב-id:

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

האות L בסוף הכתובת בזיכרון, מסמנת שזהו מספר מטיפוס Long, כלומר כזה המיוצג על ידי 64 ביט.

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

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

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

הגדרנו משתנה בשם name_copy וקבענו שהוא שווה למשתנה name. בכך, גרמנו לו למעשה להצביע לאותה הכתובת בזיכרון שאליה מצביע המשתנה name. לאחר מכן, בדקנו שהערך של name_copy הוא גם כן 'Tal', כפי שהיה 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. שימו לב שגם ל-True, גם ל-False וגם ל-None יש id.

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

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

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

העברת פרמטרים לפונקציה

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

- Pass by value
- Pass by reference

בשיטת pass by value מועבר לפונקציה **העתק** של הפרמטר. ההעתק נמצא על אזור בזיכרון שנקרא מחסנית, או stack, ומשמש פונקציות. דמיינו שהמורה מחזיק דף נייר שכתוב עליו המספר 10. המורה ניגש למכונת הצילום ומכין לכל אחד מתלמידי הכיתה העתק של דף הנייר עם הספרה 10 עליו. לכל תלמיד יש העתק של המספר. אם אחד התלמידים יכתוב על הדף שלו 11 במקום המספר 10, הדף של המורה לא ישתנה. באופן זהה, אם הפונקציה משנה את ערכו של משתנה שהועבר אליה בשיטת pass by value, היא למעשה לא משנה את ערך המשתנה עצמו, אלא העתק שלו. אי לכך, ביציאה מהפונקציה ערכו של המשתנה יהיה כפי 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 עליה לגשת לזיכרון בכתובת שנמסרה. דמיינו שכעת המורה שלנו מניח את הדף עם המספר 10 בתא שלו בחדר המורים, ובמקום לגשת למכונת הצילום הוא ניגש למנעולן ומשכפל לכל אחד מתלמידי הכיתה מפתח לתא שלו. המורה מחלק את המפתחות לתלמידים שלו. לאף תלמיד אין את הדף עם המספר 10, אבל הפעם התלמידים יכולים לגשת אל תא המורה, לקרוא את המספר 10 ואם הם רוצים – גם לשנות אותו. שינוי שיבצעו התלמידים ישפיע על הדף המקורי שבידי המורה. שימו לב שהפעם לא נוצרו לדף עותקים, יש רק עותק מקור.

אז מה קורה בשפת פייתון? האם פרמטרים מועברים בשיטת pass by value או בשיטת pass by reference?
הבה ניצור 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 מחרוזת בשם x. בשורה 9 אנחנו מדפיסים את id(x) רק כדי שנוכל להשוות אותו לפני ואחרי הכניסה לפונקציה. בשורה 10 אנחנו קוראים לפונקציה ומעבירים לה את x כפרמטר. בתוך הפונקציה אנחנו בודקים את id(x), לאחר מכן משנים את ערכו של x ואז בודקים שוב את id(x). לאחר היציאה מהפונקציה, בשורה 11, שוב בודקים את id(x) כדי לראות אם ה-id שהיה בתוך הפונקציה נשמר.

והנה מה שקיבלנו:

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

בשורת ההדפסה השניה אנחנו רואים שלפונקציה הועבר ה-id של x, שהרי ה-id מחוץ לפונקציה ובתוך הפונקציה הם זהים. האם זה אומר שהפרמטר הועבר by reference?

נראה כך, אבל אז מגיעה שורת ההדפסה השלישית, ואנו רואים שהעובדה שהפונקציה שינתה את x שינתה גם את id(x). במילים אחרות, נוצר בתוך הפונקציה משתנה חדש בשם x, שיש לו id שונה מאשר ה-id שנמסר לפונקציה. ואכן, בשורת ההדפסה הרביעית, שמתרחשת מחוץ לפונקציה, אנחנו רואים שערכו של id(x) חזר להיות כפי שהיה לפני הקריאה לפונקציה.

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

רגע לפני שנסכם, נציין שכל מה שכתבנו כעת נכון למשתנים מסוג מסויים, שנקרא immutable, שעד עכשיו עסקנו בהם בלי לקרוא להם כך. מהם משתנים מסוג immutable? ומהם משתנים מסוג mutable? על כך נרחיב כשנלמד על משתנים מסוג רשימה, list.

סיכום

בפרק זה למדנו אודות פונקציות בפייתון. למדנו להגדיר פונקציה, להעביר לה פרמטרים וגם לקבל ממנה ערכים. לאחר מכן ראינו שלמשתנים יש scope שבו הם מוגדרים, כלומר משתנה מוכר רק בפונקציה שבה הוא מוגדר. סקרנו אפשרויות שונות של שימוש במשתנים גלובליים בתוך פונקציה והגענו למסקנה שתמיד כדאי להעביר משתנים כפרמטרים לפונקציה, ובכל מקרה עדיף להמנע משינוי של משתנים בתוך פונקציה בלי שהקוד שקורא לפונקציה מודע לכך.

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

פרק 6 – List, Tuple

הגדרת List

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

מהו list וכיצד מגדירים אותו? זהו אוסף של איברים, אוסף שאפשר להוסיף אליו איברים או להוציא ממנו איברים. כמו רשימת קניות – אפשר להוסיף לה מוצרים שברצוננו לרכוש או למחוק ממנה מוצרים. מגדירים list באמצעות סוגריים 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 int, string, float ו-boolean. אם נכתוב print stam נקבל את ההדפסה הבאה:

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

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

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

ויודפס:

```
11
aaaa
36.5
True
```

כמובן שצריכה להיות שיטה יעילה יותר להדפיס את כל איברי הרשימה, נכון? לולאת for צועדת יד ביד עם list. כל מה שצריך לעשות הוא להוסיף את המילה in, ולבחור שם של משתנה שיהיה "איטרטור", כלומר יקבל כל פעם ערך של איבר אחר ברשימה. כך:

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

הלולאה תרוץ 4 פעמים, כמספר האיברים ב-stam. בכל ריצה – איטרציה – של הלולאה, האיטרטור element יקבל ערך מתוך stam, לפי הסדר שבהם האיברים נמצאים בתוך stam. בסופו של דבר תוצאת ההדפסה תהיה זהה לתוצאת ההדפסה איבר איבר.

דמיון נוסף בין list ובין string, היא היכולת לגשת לחלק מהאיברים באמצעות סוגריים מרובעים ונקודותיים – כלומר 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 מחרוזות (כגון ברירת מחדל לערכי התחלה וסיום, קפיצות וכו') תקפים גם לרשימות. לא נחזור עליהם, אך מומלץ לחזור ולקרוא את החומר על חיתוך מחרוזות ולנסות את הדברים על רשימה.

תרגיל

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

תרגיל (קרדיט: עומר רוזנבוים, שי סדובסקי)

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 לממש פונקציה שהיא כמו sum אבל לא בדיוק. נקרא לה summer. המיוחד ב-summer, הוא שהיא יכולה לעבוד עם כל טיפוס של משתנה, לא רק int או float, אלא גם משתנים מטיפוס string ואף list. כן, אפשר לחבר רשימות! נסו להגדיר 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. כלומר, כל האיברים מהרשימה ניתנים לחיבור זה עם זה.

Mutable, immutable

כעת נבין מה פירוש שני המושגים הללו, שהוזכרו בפרק הקודם. נחשוב על ההבדלים בין מחרוזת לרשימה. ראינו שכדי לגשת לאיבר ברשימה או במחרוזת, כל מה שצריך לעשות זה להכניס את האינדקס של האיבר לתוך סוגריים מרובעים. לדוגמה:

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

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

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

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

ראינו שאת הרשימה אפשר לשנות בלי בעיה, ושלאחר הכתיבה הרשימה מכילה את הערך החדש. לעומת זאת מחרוזת אי אפשר לשנות – קיבלנו שגיאה. עכשיו אפשר להבין את המושגים שבכותרת. משהו שאפשר לשנות אותו נקרא mutable, מהמילה "מוטציה". משהו שאי אפשר לשנותו הוא immutable.

עם זאת, ראינו שאפשר לשנות מחרוזות. כלומר, אפשר להגדיר מחרוזת ואז להגדיר אותה שוב עם ערך אחר. איך זה אפשרי?

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

במקרה הזה, חשוב להבין שהמחרוזת my_str כבר אינה מצביעה על אותו מיקום בזיכרון. אפשר לוודא את זה

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

בעזרת ה-id, לפני ואחרי השינוי:

נבצע שינוי ברשימה, וניווכח 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 שהגדרנו 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 כותבים ביטוי מהצורה 'איבר in רשימה', התוצאה תהיה או True או False, מה שהופך את השימוש לנוח במיוחד עבור משפטי if כמו בדוגמה. בהזדמנות זאת נזכיר שפייתון מתייחסת לאותיות גדולות וקטנות, כלומר אם נחפש apples במילון נקבל False.

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

append

המתודה append תמיד מוסיפה איבר בסוף הרשימה. כלומר, אי אפשר להוסיף באמצעותה איבר לאמצע הרשימה. המתודה מקבלת את האיבר שרוצים להוסיף. לדוגמה:

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

pop

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

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

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

sort

מה לגבי מיון רשימה? המתודה sort מטפלת בזה. כל רשימה ש-sort תפעל עליה תהפוך להיות ממויינת. אם נשתמש ב-sort בלי להזין לתוכה פרמטרים, מספרים ימוינו לפי הגודל ומחרוזות ימוינו לפי סדר הופעתם במילון:

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

אך מיון יכול להתבצע בהרבה אופנים. אפשר למיין מהקטן לגדול, מהגדול לקטן, לפי סדר האלף בית... ובכן, הבה נחקור את sort. לפני כן, ניזכר במה שלמדנו כאשר ביצענו חיתוך מחרוזות. אם יש לנו מחרוזת, אנחנו יכולים ליצור ממנה מחרוזת שונה באמצעות סוגריים מרובעים, אשר בתוכם נמצא אינדקס התחלה, אינדקס סיום ועל כמה אינדקסים מדלגים. לדוגמה, התחלה באינדקס 2, סיום לפני אינדקס 12 ודילוג של 3 אינדקסים בכל פעם:

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

כפי שראינו, למרות שישנם שלושה פרמטרים, לא חייבים לכתוב את שלושתם. אפשר לשים בסוגריים המרובעים פרמטר אחד, שניים, או להשאיר פרמטר ריק לאחר נקודותיים. לדוגמה:

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

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

כעת נחזור 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 ברירת המחדל 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 המיון בסדר הפוך. מספרים מהגדול לקטן ומחרוזות הפוך מסדר הופעתן במילון.

נחמד, אבל מה אם נרצה לעשות מיון יותר יצירתי? לא רק מהקטן לגדול או מהגדול לקטן, אלא כל מיון שנרצה? המתודה sort מקבלת בתור פרמטר את המפתח למיון, פרמטר אשר נקרא key. בתור מפתח אנחנו יכולים לקבוע

איזו פונקציה שאנחנו רוצים. נראה דוגמה. יש לנו רשימה שכוללת כמה מחרוזות. אנחנו רוצים לבצע מיון לפי אורך המחרוזות. כלומר, המחרוזת 'zz', שהאורך שלה הוא 2, צריכה להופיע לפני המחרוזת 'aaa' שהאורך שלה הוא 3 ואחרי המחרוזת 'c' שהאורך שלה הוא 1 בלבד. כידוע לנו, הפונקציה len מחזירה אורך של מחרוזת. לכן פשוט נעביר בתור מפתח את הפונקציה len, כך:

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

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

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

ראשית, נצטרך להגדיר פונקציה שמקבלת מחרוזת ומחזירה 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
```

ותוצאת ההדפסה שנקבל:

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

split

לעיתים נקבל מחרוזת ונרצה להפריד אותה למחרוזות קטנות יותר ולשמור אותן ברשימה. הדוגמה הקלאסית היא מחרוזת שכוללת משפט, ואנחנו רוצים להפריד אותה למילים בודדות. לשם כך קיימת המתודה split. כאמור, split פועלת על משתנים מטיפוס מחרוזת, אך כיוון שהיא מחזירה רשימה נוכל כעת להבין יותר טוב את אופן הפעולה שלה.

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

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

נרצה ליצור רשימה בה כל איבר יהיה שם של סרט. למזלנו 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']
```

למתודה split יש גם ערך ברירת מחדל, שהוא סימן רווח. במילים אחרות, 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()
```

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

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

join

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

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

כעת אם נדפיס את rule נקבל את המחרוזת המקורית:

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

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

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

Tuple

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

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

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

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

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

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

תוצאות ההדפסה:

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

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

שימו לב לדרך אלגנטית לקלוט מספר ערכים מפונקציה שמחזירה מספר ערכים:

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

כעת כל אחד מהמשתנים first, second הוא מחרוזת שמכילה את אחד הערכים שהוחזר מהפונקציה.

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

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

a, b = stam()
```

אם כך, מדוע tuple?

הסיבה המעניינת קשורה לדרך שבה פייתון מקצה זיכרון לרשימות. פייתון מניח שאם הגדרנו רשימה, ייתכן שנרצה להוסיף אליה ערכים באמצעות append או לשנות את הערכים השמורים בה. לכן פייתון צריך לדאוג שיהיו לרשימה מצביעים 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, לכן פייתון מקצה בדיוק את כמות המקומות שהגדרנו. כלומר, אם אנחנו יודעים שאנחנו לא מתכוונים להוסיף איברים, הטיפוס tuple הוא חסכוני

יותר במיקום בזיכרון. כאשר אנחנו מחזירים ערכים מפונקציה, אנחנו יודעים כמה ערכים החזרנו, לכן השימוש ב-tuple הוא מתבקש.

סיכום

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

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

פרק 7 – כתיבת קוד נכונה

עד כה למדנו לכתוב קוד פייתון בסיסי, אך עשינו זאת בלי הקפדה רבה על איכות הקוד שאנחנו כותבים. מדוע חשוב לכתוב קוד פייתון "נכון"? הרי אנחנו יודעים למה אנחנו התכוונו כאשר כתבנו את התוכנית...? הבעיה היא שבעולם ה"אמיתי" סביר מאוד שהקוד שכתבנו יהיה רק חלק מתוך מערכת גדולה יותר, שנכתבת על ידי צוות מתכנתים, מחלקת מתכנתים או אפילו קהילה של מתכנתים. כאשר אנשים אחרים יקראו את הקוד שכתבנו, הוא צריך להיות קריא מספיק על מנת שיהיה להם קל להשתמש בו. בנוסף, שמירה על כללי כתיבת קוד איכותי תסייע לנו למנוע כתיבה של שגיאות, ובייחוד שגיאות שיהיה לנו קשה במיוחד לאתר ולתקן. כמו כן, כתיבת קוד איכותית תסייע לנו במהלך כתיבת הקוד ותמנע שכפול של פעולות קיימות, ובכך תחסוך לנו זמן יקר.

מהו בכלל קוד איכותי? בפרק זה נתמקד בכמה כללים:

א. הקוד צריך לעבוד. כלומר, אם קוד נכתב במטרה לבצע פעולה מסויימת, עליו לבצע אותה בצורה נכונה. לדוגמה, אם נכתוב משחק מחשב קטן, המשחק צריך לרוץ באופן חלק בלי להיתקע.

ב. הקוד צריך להתחשב במקרי 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. המשתמש הקליד מספר שלילי. הפונקציה צריכה לא לקרוס. גם אם המשתמש הכניס ערך שאינו מספר, הפונקציה צריכה לא לקרוס.

ג. הקוד צריך להכתב עם חלוקה נכונה לפונקציות. כל פונקציה צריכה לטפל במשימה אחת ולעשות את מה שהיא אמורה לעשות ורק את מה שהיא אמורה לעשות. למה זה בכלל משנה? הרי אם הקוד עובד, אז הוא עובד? ראשית, בתוכניות גדולות חלוקה נכונה לפונקציות יכולה לחסוך זמן פיתוח רב. נאמר שפיתחנו תוכנה מורכבת ולא עשינו חלוקה לפונקציות. כאמור, אנחנו עובדים בצוות של מתכנתים. סביר שמתכנת שעובד איתנו יצטרך לקרוא ולהכיר את הקוד שלנו, וכתיבת כל הקוד בבלוק אחד תקשה עליו למדי. 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 עם חלוקה לפונקציות, סביר שהיינו צריכים לבצע שינוי רק בפונקציות בודדות.

ד. הקוד צריך לכלול שמות משמעותיים לפונקציות ומשתנים. לדוגמה, L12 אינו שם טוב למשתנה – למי שקורא אותו אין מושג מה הוא שומר בתוכו. לעומת זאת, salary (משכורת) הוא משתנה יותר ברור. הדבר תקף גם לגבי פונקציות. לפונקציה שמבצעת בדיקה אם קלט תקין אפשר לקרוא stam, או g1, שמות שמי שקורא אותם אינו מבין מהם מה הפונקציה מבצעת, או פשוט check_valid_input. מה יותר קריא?

ה. הקוד צריך להיות מתועד. כלומר, יש לכתוב docstring בתחילת כל פונקציה ויש להוסיף תיעוד שמסביר מה עשינו, אך יש להמנע מתיעוד יתר, שרק מציין את המובן מאליו ואינו מוסיף מידע חשוב לקורא. דוגמה לתיעוד יתר כזה:

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

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

ו. הקוד צריך לתאום לקונבנציות, או בעברית "מוסכמות". כאן הסיבה היא פשוט נוחות של מי שמנסה לקרוא את הקוד שלכם. לדוגמה, שמות של קבועים ייכתבו באותיות גדולות. כך, אם נקרא קוד של מישהו ונמצא שם אותיות גדולות, קיבלנו מידע חשוב בלי מאמץ.

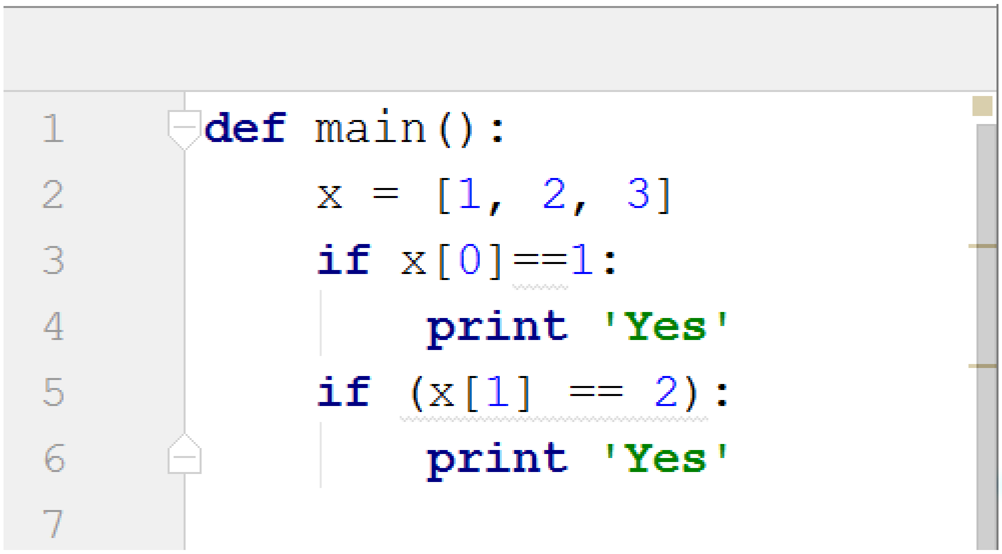
את הפרק הזה נקדיש לנושא כתיבת קוד נכונה. נחלק את הלימוד לתתי הנושאים הבאים:

- כתיבת קוד פייתון לפי קונבנציית PEP8
- חלוקה נכונה של קוד לפונקציות
- בדיקת תקינות קוד ומקרי קצה באמצעות assert

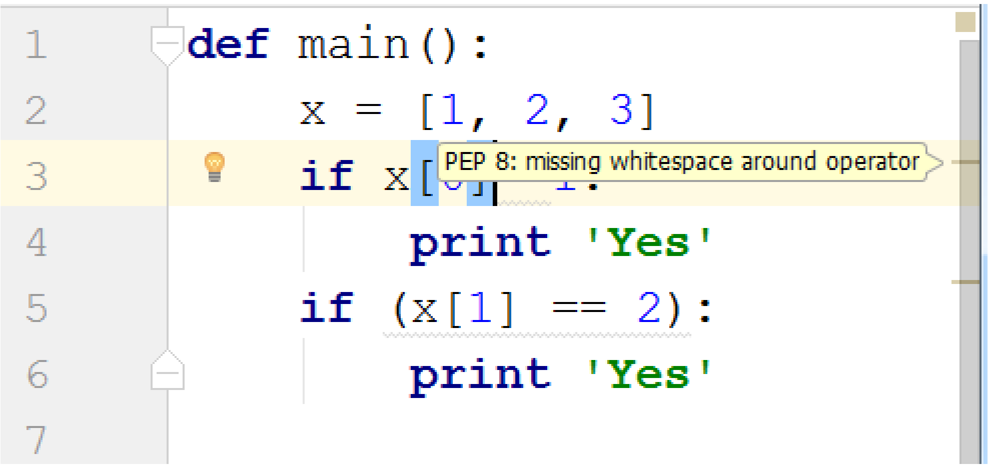
PEP8

אוסף כללי כתיבה נפוץ של קוד פייתון נקרא PEP8. מן הסתם לא נעבור כאן על כל הכללים – מדריך קצר מאת עומר רוזנבוים ושי סדובסקי נותן מושג לגבי עיקר הכללים: http://data.cyber.org.il/networks/PEP8.pdf

נתעכב על השימוש ב-PyCharm על מנת לאתר טעויות PEP8 (כלומר, קוד שלא עומד בקונבנציות שהוגדרו לפי PEP8) ולתקן אותן. שימו לב לקטע הקוד הבא:

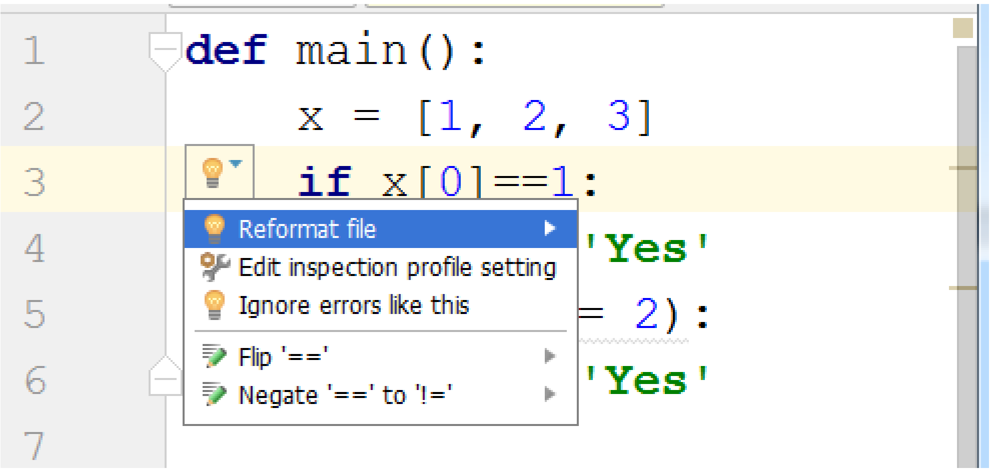


ניתן לראות שהריבוע בצד ימין למעלה הינו בצבע צהבהב, לא ירוק אבל גם לא אדום. הדבר מראה לנו שהקוד ירוק בצורה תקינה, אבל יש בו טעויות PEP8. קל למצוא את הטעויות – האם אתם מבחינים בשני הקווים הצהבהבים שמתחת לריבוע? נעמוד על אחד מהם:



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

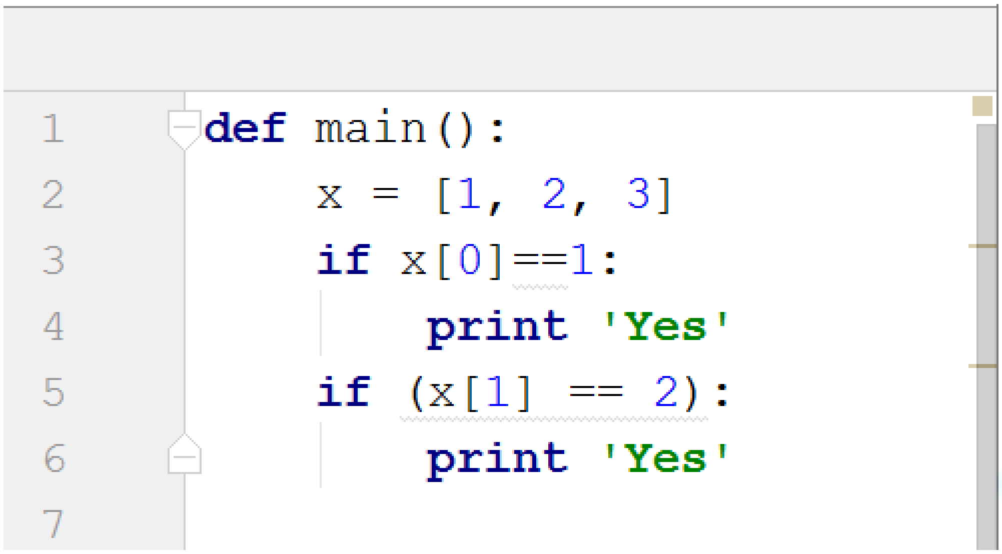
ייתכן ששמתם לב גם לסימן הנורה הצהובה שדולקת לצד שורת הקוד הבעייתית. לחיצה שמאלית עליה תפתח לנו תפריט, ואם נבחר באפשרות Reformat file הבעיה תתוקן לבד!



אפשרות נוספת לגלות בעיות PEP8 בקוד היא באמצעות סימן אפור דק מתחת לתווים הבעייתיים. דבר זה מסייע לנו להמנע מבעיות כבר בשלב הכתיבה.

תרגיל

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



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

א. תיעוד: כל פונקציה שאתם כותבים צריכה להכיל docstring, כפי שראינו בתחילת הפרק אודות פונקציות. מומלץ לחזור על הדוגמאות שמסבירות כיצד לכתוב docstring.

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

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

יש בצורת הכתיבה הזו שתי בעיות. ראשית, לא ברור מה מציין המספר 10 (אפשר להבין זאת מקריאת הקוד, אבל בקוד מורכב יותר זה ייקח זמן). שנית, אם נרצה שהתוכנית שלנו תדפיס 11 פעמים ולא 10, עלינו לחפש בקוד את המספר 10 ולשנות אותו ל-11. נאמר שהמספר 10 חוזר על עצמו מספר פעמים בקוד – מכיוון שאנחנו 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 יתייחס למקרה אחר – למשל כמות הפעמים שאנחנו רוצים לקרוא מידע מהמשתמש, ויכול להיות שדווקא שם נרצה לשמור על המספר 10 כשלעצמו. ככל שהתוכניות שלנו יהיו מורכבות יותר, כך הזמן שנצטרך להקדיש לזה יגדל.

האפשרות הטובה יותר היא שימוש בקבועים. לדוגמה:

```
TIMES_TO_PRINT = 10

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

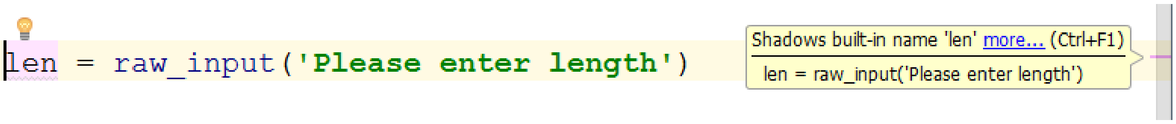
הגדרנו בתחילת התוכנית קבוע (שימו לב לאותיות הגדולות!), שרק מקריאת השם שלו אנחנו כבר יודעים מה הוא עושה. כעת אם נרצה להדפיס מספר שונה של פעמים, כל שנצטרך לעשות הוא לשנות את הקבוע, סיימנו.

שימו לב שפייתון לא מוודא עבורנו שהקבוע שלנו לא ישתנה. כלומר, עדיין ניתן לבצע שינוי לערך הקבוע שלנו:

```
TIMES_TO_PRINT = 5
```

במקרה זה, הקונבנציה אמורה לשמור עלינו מפני עצמנו – המתכנתים. אנו מציינים את שם הקבוע באותיות גדולות כדי לזכור שהוא קבוע – ולא לשנות אותו בקוד שלנו.

ג. לא לדרוס שמות מובנים בפייתון: לפייתון יש שמות מובנים – built-in names – שהפרשן שלו מכיר גם בלי שהגדרנו אותם. נניח שאנחנו מעוניינים לקלוט מהמשתמש מספר שמייצג אורך של משהו. לדוגמה, אורך של ספר בעמודים, או אורך של משחק כדורסל בדקות. מאוד מפתה להגדיר את המשתנה בשם len– קיצור של length. הבעיה היא שהשם 'len' הוא שם מובנה בפייתון, כלומר יש לפייתון כבר פונקציה שנקראת len והיא יודעת להחזיר אורך של פרמטרים שהיא מקבלת. מה יקרה אם נדרוס אותה? כלום, פרט לכך שיותר לא נוכל להשתמש בה... לא לתמיד, אבל כל עוד הסקריפט שלנו רץ.



חלוקת קוד לפונקציות

כאשר נכתוב קוד, נחשוב איך אפשר לחלק את הקוד לפונקציות שעושות דברים מוגדרים. בקוד איכותי, כל פונקציה תעשה בדיוק את מה שהיא צריכה לעשות – לא פחות ולא יותר. זיכרו – מטרתנו אינה לכתוב את הקוד הכי קצר או הכי יעיל, אלא קוד שעובד ללא תקלות, ברור לקריאה וניתן בקלות להתאמה למשימות אחרות. מדוע זו מטרתנו? משום שלמעט מקרים נדירים, בהן המערכת שלנו רצה במגבלות משאבים, הרבה יותר סביר שיש לנו משאבי מחשב רבים אך מאידך אנחנו מוגבלים בכמות הזמן שיש לנו לטובת כתיבת קוד, או שהמחיר של באג במערכת הוא גבוה מאוד, או שאנחנו נדרשים לכתוב קוד בצורה שצוות או קהילת מתכנתים יוכלו להבין ולהשתמש בו. מכל הסיבות האלה, קוד איכותי הוא קוד שאין בו באגים, שהוא קל להבנה ושאפשר בקלות להשתמש בו למשימות נוספות.

הבה ניקח משימה פשוטה יחסית ונדגים עליה צורות שונות של כתיבת קוד. בתור משימה ניקח את הבעיה הבאה: ברצוננו לקבל מהמשתמש רשימה של מספרים, ולבדוק אם היא מקיימת חוקיות מסויימת. לדוגמה, שכל מספר הוא ממוצע שני המספרים שצמודים לו. הרשימה -

1, 3, 5, 7

היא רשימה בה כל מספר הוא ממוצע שני המספרים שצמודים לו (נכון, רשימה זו היא בהכרח סדרה חשבונית, אך יכולנו לבחור כל חוקיות אחרת, ולכן נציג את הפתרון הכללי לבעיה 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 ולבדוק אם החוקיות מתקיימת. כלומר אם 3 הוא ממוצע של 1 ו-5, ואם 5 הוא ממוצע של 3 ו-7. את האיבר הראשון והאחרון אין צורך לבדוק, מן הסתם.

פתרון מודרך

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

```
__author__ = 'Barak'
# First attempt to solve the problem -
# Check if each number in a list is the
# average of the prior and next numbers


```

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

```

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

ראשית, ספר הלימוד בכוונה אינו מסביר כיצד עובד הקוד. נסו להבין בעצמכם איך הקוד מבצע את המשימה שהוא אמור לבצע. דמיינו שאתם מתכנתים בצוות וחברכם לצוות השאיר לכם את הקוד הזה ויצא לחופשה. לבטח תצליחו להבין את הקוד, אבל המשימה צפויה לגזול ממכם זמן רב יחסית לקוד לא ארוך. שנית, אחת הסיבות המרכזיות שהקוד הזה קשה לקריאה (וגם לכתיבה 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 מתבטא גם בזמן שלוקח לדבג את הקוד. במילים אחרות, אם יש לכם באג בקוד שנראה כך, צפו לערב ארוך מול המחשב...

ננסה לשפר את הקוד שלנו. בניסיון השני, נפשט את הקוד שלנו באמצעות הפרדה בין המשימות בקוד. בעוד שבקוד של הניסיון הראשון פעולת קליטת המספרים היתה משולבת בפעולת 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). נכון, אפשר לטעון שהקוד הקודם ביצע את המשימה בצורה יותר מהירה, אבל כזכור המטרה שלנו אינה להאיץ את מהירות הריצה של הקוד אלא לצמצם את כמות הזמן שלוקח לכתוב, לדבג ולתחזק את הקוד. שימו לב כיצד עצם החלוקה למשימות שונות מורידה את כמות התנאים בתוך תנאים. יש לנו כרגע לכל היותר 3 רמות אינדטציה, 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()
```

נחמד. ועדיין זה אינו קוד מוצלח. מדוע? ראשית, הקוד אינו מתועד. בתוך הקוד אין הסברים לדרך שבה הקוד פותר את הבעיה. אמנם, בדיקה אם מספר מקיים תנאי של להיות ממוצע של שני מספרים היא בדיקה מאוד פשוטה ואפשר להבין מתוך הקוד מה מתבצע, אולם ככל שנרצה לבצע משימות מסובכות יותר כך ייחסר לנו יותר תיעוד. שנית, קשה להשתמש בקוד הזה למשימות אחרות. נסביר: נניח שחבר שעובד איתנו נתקל גם הוא בבעיה בה הוא צריך לעבור על רשימת מספרים ולבדוק אם מתקיימת החוקיות הנ"ל. לחברנו לעבודה כבר יש תוכנית אחרת שדואגת 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 בתוכנית שלו. מסובך! בשביל זה יש פונקציות. שלישית, האם אתם מזהים אפשרות כלשהי שהקוד שלנו יקרוס תוך כדי ריצה? מה לדעתכם יקרה אם המשתמש לא נענה להוראות שלנו והזין ערכים שאינם ספרות? בשורה 14, מתבצעת המרה של קלט המשתמש ממחרוזת ל-int. אם המחרוזת אינה ברת המרה ל-int (לדוגמה, נסו להמיר את 'table'

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

לסיכום, אנחנו צריכים להכניס 3 שיפורים בקוד:

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

להלן הגרסה השלישית של הקוד, שמטפלת בשיפורים הנדרשים. שימו לב, ברור שהקוד יהיה ארוך יותר, אך בואו נראה אם כעת קל 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 עושה בלי לקרוא אותה כלל. לאחר מכן התקדמו אל הפונקציות, קיראו קודם כל את ה-docstring שלהן ורק לאחר מכן את הפונקציות עצמן. השוו בין כמות הזמן שלוקחת לכם הקריאה כעת לעומת

```python
__author__ = 'Barak'
# Third attempt to solve the problem -
# Check if each number in a list is the
# average of the prior and next numbers

END_INPUT = 'STOP'
MIN_LIST_LENGTH = 3


def receive_user_inputs(ending_value):
    """ Receive multiple inputs from the user and store in list
    Args:
        ending_value - once the user enters this value, stop receiveing
    Return value:
        list of all received values
    """
    input_list = []
    while True:
        user_input = raw_input('Enter num, {} to quit '.format(ending_value))
        if user_input == ending_value:
            break
        else:
            input_list.append(user_input)
    return input_list


```

הגרסאות הקודמות. מה יותר מהיר?

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


def list_is_average(input_list):
    """ Check if a list is according to the condition
    that each element is the average of the adjacent elemennts
    like: 1, 3, 5, 7
    Args:
        input_list - the list to be tested
    Return value:
        True / False
    """
    # Convert list from str elemnts to floats
    nums = []
    for element in input_list:
        nums.append(float(element))
    # Check if the nums are all averages
    for index in xrange(1, len(nums)-1):
        avg = float(nums[index-1] + nums[index+1])/2
        if nums[index] != avg:
            return False
    return True


def main():
    input_list = receive_user_inputs(END_INPUT)
    if len(input_list) >= MIN_LIST_LENGTH and list_is_nums(input_list):
        if list_is_average(input_list):
            print 'List is good'
        else:
            print 'List is not good'

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

דבר נוסף שהרווחנו מכתיבת הקוד באופן הזה, הוא שנוכל להשתמש בפונקציות גם בתוכניות אחרות. זאת משום שהפונקציות לא עושות שימוש במשתנים גלובליים – כל מה שהן צריכות 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 בגרסה האחרונה של הקוד? נשאל את עצמנו – איך אנחנו יודעים שהפונקציות שכתבנו אכן עובדות? כלומר, יכול להיות שהן עובדות ברוב במקרים, אך במקרי קצה 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 במקרי קצה, אך ביצענו בה שיפור. השיפור גורם לכך שבמקרי קצה הפונקציה שלנו כבר לא עובדת היטב, או קורסת.

נרצה למצוא דרך להריץ בקלות בדיקות על פונקציה, בין שאנחנו כתבנו אותה ובין שאנחנו משתמשים בפונקציה שמישהו אחר כתב. הבדיקות יאפשרו לנו לגלות גם אם הפונקציה רצה באופן תקין וגם אם גרמנו לבעיה בעקבות שינוי שעשינו בקוד. ישנן מספר דרכים לבדוק את הקוד שלנו, ואנו נכיר את הדרך הפשוטה ביותר – assert.

כדי להשתמש ב-assert, כותבים assert, לאחר מכן ביטוי כלשהו, שיכול להיות True או False. במקרה שלנו נכתוב שם של פונקציה, לאחר מכן סוגריים 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 מחזירה True, אחרת False. נוסיף לתוכנית שלנו שני assertים 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 שמועברים 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', שכמובן אינה ניתנת להמרה למספר. ה-assert מוודא שבמקרה זה ערך החזרה מהפונקציה הוא False.

החלק המעניין ב-assert הוא להכניס קלטים מיוחדים, על מנת לבדוק מקרי קצה. לדוגמה, מה יקרה אם הפונקציה list_is_nums תקבל רשימה ריקה? במקרה זה נרצה לוודא שיוחזר לנו False, לא כך? נבדוק:

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

בשלב הרצת התוכנית נקבל 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 הנבדקת מחזירה ערך שונה מהערך שציפיתם לו, ואכן זה מה שקרה". זאת מכיוון שהפונקציה list_is_nums מחזירה True אם היא מקבלת רשימה ריקה. אכן, היה עוד מה לשפר בתוכנית שלנו, וגילינו זאת באמצעות ה-assert. כדי לתקן זאת, נצטרך להוסיף לפונקציה list_is_nums בדיקה האם הפונקציה קיבלה רשימה ריקה.

נוסיף לקוד שלנו assertים שונים, במטרה לבדוק את הפונקציות. הנה ה-main שלנו לאחר התוספות:

```
def main():
    assert list_is_nums(['1', '2', '13', '14']) is True
    assert list_is_nums(['10', '11', 'hi']) is False
    assert list_is_nums([]) is False
    assert list_is_average(['1', '3', '5', '7']) is True
    assert list_is_average(['1', '2', '4', '7']) is False
    input_list = receive_user_inputs(END_INPUT)
    if len(input_list) >= MIN_LIST_LENGTH and list_is_nums(input_list):
        if list_is_average(input_list):
            print 'List is good'
        else:
            print 'List is not good'
```

מספר דגשים:

- את הפונקציה receive_user_input אנחנו לא בודקים עם assertים. מדוע? מכיוון שהפונקציה הזו דורשת קלט משתמש. המשתמש לא צריך להיות מודע לכך שהתוכנה שלנו כוללת בדיקות. הרי אין זה הגיוני שנבקש מהמשתמש להזין קלטי בדיקה בכל פעם שהוא מריץ את התוכנית שלנו...

- הפונקציה list_is_average לא כוללת assert עם רשימה ריקה. זאת מכיוון שאם הגענו לפונקציה הזו, זה 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 לכל הפחות, כדי שמי שישתמש בה בעתיד – אולי בתוכנית אחרת – יידע זאת.

- הפונקציות שלנו לא כוללות הדפסות. זאת מכיוון שאי אפשר לבדוק הדפסות בעזרת assert, שבודק רק ערכים שהפונקציה מחזירה. לכן, הדרך המקובלת היא שהפונקציה מחזירה ערך, והקוד שקרא לה מדפיס

מה שצריך לפי הערך שהוחזר. בדיוק כמו בדוגמה, בה הפונקציה list_is_average החזירה רק True / False וההדפסה בוצעה בשורות הקוד שאחרי הקריאה לפונקציה.

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

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

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

בתור התחלה 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'
```

פעולת ה-assert הראשונה בודקת אם התוצאה היא כצפוי. פעולת ה-assert השניה כוללת דבר נוסף – הודעת שגיאה שתודפס במקרה שהערך שיתקבל לא יהיה שווה לערך הצפוי. במקרה זה יודפס 'Not the expected result', אך כמובן שאפשר להדפיס כל הודעה. מומלץ כמובן שההודעה תכלול מידע אודות השגיאה, כך שמי שמריץ יוכל בקלות לדבג ולמצוא את מקור הבעיה.

כעת תורכם – אילו עוד פעולות assert כדאי לעשות על my_div? נסו לחשוב על מקרי קצה.

ובכן, הדבר הראשון שמומלץ לבדוק בחלוקה היא התמודדות עם חלוקה באפס. אם הפונקציה קורסת זו בעיה, והיינו רוצים שבמקרה של חלוקה באפס תוחזר לנו הודעת 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 הודעה כגון 'Parameters are not numbers'. באופן זה מי שקורא לפונקציה עם ערכים לא נכונים לא יגרום לריסוק התוכנית.

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

תרגיל מסכם – Deja Vu (קרדיט: 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? ראיתי כבר את התרגיל הזה! יש לי דז'ה וו...

נכון מאוד ☺ 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 לקרוס בשום אופן. הקפידו על כל הדברים שלמדנו בפרק זה – חלוקה נכונה של הקוד לפונקציות, שימוש ב-PEP8, תיעוד ובדיקת פונקציות על ידי assertים.

סיכום

פרק זה התמקד בשדרוג יכולות התכנות שלנו. התחלנו מנושא שנראה טכני למדי במבט ראשון – שמירה על קונבנציות לפי PEP8 – אבל ראינו את החשיבות של נושא זה לכתיבת קוד קריא, שמתכנתים אחרים יכולים להבין בקלות ולעשות בו שימוש.

לאחר מכן התמודדנו עם אחד המחסומים המרכזיים שעומדים בפני מתכנתים מתחילים: כתיבת תוכנית שלא רק מבצעת את מה שהיא צריכה לבצע, אלא גם מחולקת למשימות שמבוצעות 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 חוזר.

לבסוף ראינו איך assert עוזר לנו לבדוק שהקוד שלנו תקין ועובד היטב. למדנו שכאשר כותבים פונקציה, צריך לחשוב על כל מקרי הקצה ולבדוק אותם באמצעות קריאה מהתוכנית הראשית.

פרק 8 – קבצים ופרמטרים לסקריפטים

בפרק זה נלמד שני נושאים שימושיים ביותר עבור כתיבת תוכניות פייתון. הראשון, שימוש בקבצים – נראה איך פותחים קובץ לקריאה ולכתיבה. השני, העברת פרמטרים לסקריפטים – כלומר היכולת להריץ סקריפט עם מידע שיגרום לסקריפט לרוץ בצורה מוגדרת, מבלי לבקש מהמשתמש להזין ערך כלשהו תוך כדי ריצת הסקריפט. בתור תרגיל מסכם נשלב את שני הדברים יחד – נריץ סקריפט שמקבל כפרמטר שמות של קבצים ופועל עליהם.

פתיחת קובץ

בפייתון ישנה פונקציה מובנית בשם open, אשר מקבלת בתור פרמטר שם של קובץ. שם הקובץ צריך להיות מחרוזת, כאשר מומלץ לשים לפניה את הסימן r, שכפי שלמדנו מסמן מחרוזת raw, כך ששם הקובץ יוזן כמו שהוא ולא תבוצע המרה לתווים מיוחדים. חייבים להעביר לפונקציה גם את אופן פתיחת הקובץ, לדוגמה אנחנו יכולים לפתוח קובץ לקריאה או לפתוח קובץ לכתיבה. אופן הפתיחה נקרא mode. נסקור חלק מה-mode‏ים השונים (תיאור מלא נמצא ב-https://docs.python.org/2/tutorial/inputoutput.html בסעיף 7.2):

- לכתיבה של טקסט נשתמש ב-w, קיצור של write
- לקריאה של טקסט נשתמש ב-r, קיצור של read
- אם נרצה לכתוב מידע לקובץ בלי לדרוס את המידע הקיים, נשתמש ב-a, קיצור של append. אם לא נפתח כך את הקובץ, אלא נשתמש ב-w, כל כתיבה שנכתוב לקובץ תתחיל מתחילת 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 כתבן לא נוכל לקרוא את התוכן שלה. כדי לטפל בקבצים בינאריים יש צורות מיוחדות של קריאה וכתיבה:

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

דוגמה לשימוש ב-open לפתיחה של קובץ טקסט:

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

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

קריאה מקובץ

המתודה read פועלת על אובייקטים מסוג file. בתור ברירת מחדל, המתודה קוראת את כל הקובץ ושומרת אותו בתור מחרוזת בתוך משתנה שהוגדר על ידי המתכנת. לדוגמה:

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

תוצאת פקודת ההדפסה:

```
"Dear Prudence" / The Beatles

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

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

אפשרות אחרת היא להשתמש במתודה readline, שכפי שמרמז השם שלה קוראת שורה אחר שורה. לדוגמה:

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

מה יקרה אם הגענו לסוף הקובץ? במקרה זה הערך שנקבל מ-readline יהיה '', כלומר מחרוזת ריקה. דוגמה לקטע קוד קצר שמדפיס את הקובץ שורה אחר שורה:

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

שימו לב לכך שבסוף השורה האחרונה יש לנו פסיק. הפסיק מסמן שלא להוסיף ירידת שורה בסוף ההדפסה. הסיבה היא שבקובץ הטקסט ממילא יש ירידת שורה בסוף כל שורה, ואילו לא היינו שמים פסיק היה רווח של שורה נוספת בין כל שתי שורות מודפסות.

עקב השימושיות של הדפסת שורה אחר שורה, פייתון מאפשר לנו להשתמש בלולאת for רגילה עם איטרטור. בכל איטרציה מתבצעת למעשה קריאה של שורה אחת. כך, הקוד שלנו ניתן לכתיבה מקוצרת ונחמדה:

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

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

בנוסף, שימו לב לכך שבחרנו בשם line כדי לייצג כל שורה – כך ברור מה המשתנה הזה כולל בכל קריאה. קל יותר לקרוא קוד, במיוחד אם הוא ארוך, כשכל שמות המשתנים הם בעלי משמעות. זהו אחד מעקרונות כתיבת הקוד הנכון אותם הזכרנו בפרק הקודם.

כתיבה לקובץ

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

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

סגירת קובץ

לאחר שמסיימים את הטיפול בקובץ מומלץ לסגור אותו. אמנם קובץ שפתחנו ייסגר אוטומטית ברגע שתסתיים ריצת התוכנית שלנו, אבל אי סגירה של קובץ יכולה לגרום לתוכנית שלנו להתנהג בצורה לא צפויה וקשה מאוד לדיבוג. נמחיש על ידי דוגמה. התוכנית הבאה קוראת לפונקציה שפותחת קובץ לכתיבה בלי לסגור אותו, וכדי להמחיש שזה אינו תכנות נכון, הפונקציה קרויה open_without_closing. הפונקציה משנה את תוכן הקובץ. לאחר מכן נפתח אותו קובץ שוב, הפעם לקריאה. המצביעים לקובץ נקראים כאן fd, קיצור של file descriptor.

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


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


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

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

```
"Dear Prudence" / The Beatles

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

המסקנה היא שכדאי תמיד לסגור קבצים אחרי שסיימנו להשתמש בהם. כדי לעשות זאת משתמשים במתודה close. פשוט כך:

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

כעת, ההדפסה שנבצע מתוך פונקציית ה-main תדפיס כמו שצריך גם את השורה האחרונה שהוספנו.

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

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

לאחר הפקודה with, נכתוב open עם הפרמטרים הרגילים. לאחר מכן, נוסיף as ואת שם המשתנה שיכיל את המצביע לקובץ. שורות הקוד הבאות מדפיסות את הקובץ, בדיוק באותו אופן שבו הדפסנו אותו קודם. מתי ייסגר הקובץ? הקובץ יישאר פתוח רק כל עוד אנחנו נמצאים בבלוק של with. ברגע שהבלוק ייגמר, ייסגר הקובץ אוטומטית.

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

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

תרגיל – מכונת שכפול

צרו באמצעות סייר חלונות שני קבצי טקסט, האחד ריק והשני כולל טקסט כלשהו. כיתבו סקריפט אשר מוגדרים בו שמות שני קבצים. הסקריפט יעתיק את הטקסט אל הקובץ הריק, כך שלאחר סיום הריצה הקבצים יכילו את אותו טקסט.

קבלת פרמטרים לתוכנית

דמיינו שאתם משתמשים בסקריפט פייתון שבודק משהו על המחשב שלכם. לדוגמה, אם תיקיית קבצים כלשהי מכילה קבצי פייתון, בעלי הסיומת py או pyc. כאשר אתם מריצים את הסקריפט, אתם מתבקשים להזין את שם התיקיה אותה אתם מעוניינים לבדוק. זה בסדר, אבל משהו פה מיותר: ממילא כאשר אתם מריצים את הסקריפט אתם יודעים על איזו תיקיה תרצו לפעול. למה צריך שהסקריפט ידפיס הודעה ויבקש מכם להזין קלט? למה שלא

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

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

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

בצורה זו, ניתן גם לבדוק בקלות רבה יותר את הסקריפט. אפשר להריץ אותו עם ערכים שונים ולבדוק שקיבלנו תוצאות נכונות.

מבחינה טכנית אנחנו רוצים שהקובץ findpy.py יהיה כתוב כך שהוא יוכל לקבל כפרמטר את שם התיקיה שבה הוא צריך לבדוק האם קיימים קבצי פייתון. הבה נראה איך עושים זאת.

בתור התחלה נכיר בקצרה 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 וכמובן – קבלת פרמטרים לסקריפט. אפשר לקרוא על sys בלינק http://www.python-course.eu/sys_module.php.

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

```
import sys


def main():
    print sys.argv

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

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

sys.argv, בדיוק כמו בדוגמה שבקוד. בהמשך הספר, בפרק על OOP, נלמד על שיטות נוספות להשתמש בפונקציות ובמבנים נוספים שנמצאים בתוך מודולים.

argv מאפשרת לנו לקבל את הארגומנטים שהועברו לסקריפט שלנו. באינדקס ה-0 של הרשימה נמצא שם הסקריפט שאנחנו מריצים, וביתר האיברים נמצאים הפרמטרים שהעברנו לסקריפט (אם העברנו כאלה, אחרת הרשימה מכילה רק איבר אחד – שם הקובץ).

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

```
import sys
NAME = 1


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

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

כאשר עובדים עם PyCharm נוח להעביר פרמטר לסקריפט שלנו באמצעות PyCharm, ולא באמצעות ה-cmd. כיצד מעבירים את הפרמטר ל-PyCharm? זה זמן טוב לחזור לפרק אשר דן ב-PyCharm ולקרוא את החלק אודות העברת פרמטרים לסקריפט. קבענו כפרמטר את המחרוזת 'Shooki' וכתוצאה 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 הודעת שגיאה. זו בעיה, מכיוון שגם אם המשתמש שגה והזין שם קובץ שאינו קיים, לא נרצה שהקוד שלנו יקרוס. במקום זה, עדיף להחזיר למשתמש שגיאה שמסבירה לו שהקובץ אינו קיים. כדי לתקן את הבעיה, נכיר את המודול os. מודול שימושי נוסף זה, קיצור של Operating System, מספק יכולות שונות של מערכת הפעלה. בתור דוגמה, נראה איך מציגים שמות של קבצים בתיקיה.

נתחיל כמובן מ-import os. ביצוע dir על os יראה לנו את כל הפונקציות ששייכות ל-os, ביניהן נמצאת הפונקציה החביבה listdir. כעת 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 לתוך directory את הפרמטר שקיבל הסקריפט שלנו, ואז אנחנו מעבירים את directory ל-os.listdir על מנת לקבל את רשימת הקבצים.

תרגיל – os.path

הבעיה בקוד שלנו, כמו בקוד שכתבתם כפתרון לתרגיל printer, היא שעדיין נהיה בבעיה אם לסקריפט יועבר שם של תיקיה שאינה קיימת. עליכם 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 רשימת כל המתודות באמצעות dir(os.path) ותוכלו להשתמש ב-help על מנת לקרוא מה עושה כל מתודה, עד שתמצאו את המתודה המתאימה.

תרגיל מסכם – Lazy Student

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

46 + 19

15 * 3

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

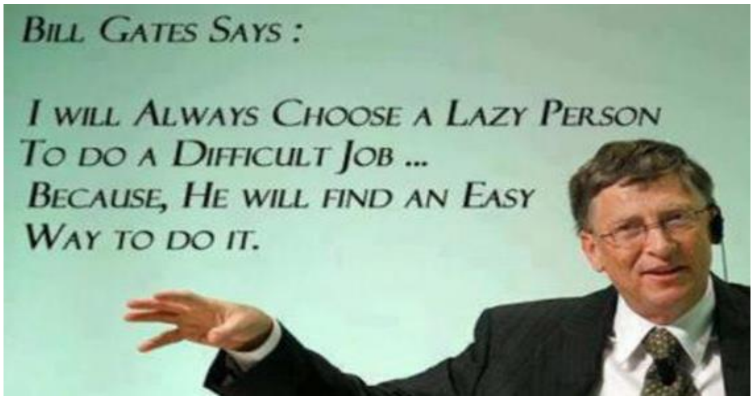
46 + 19 = 65

15 * 3 = 45

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

python lazy_student.py homework.txt solutions.txt

הסקריפט יקרא את התרגילים מתוך homework.txt וישמור את הפתרונות אל solutions.txt. הניחו כמובן שיש שגיאות בפורמט של חלק מהתרגילים, או בשמות הקבצים. אסור לסקריפט שלכם לקרוס בשום אופן! אם יש בעיה בתרגיל, כיתבו במקום המתאים בקובץ הפתרונות הודעת שגיאה והמשיכו לתרגיל הבא.



סיכום

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

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

פרק 9 – Exceptions

בפרק זה נלמד להגן על הקוד שלנו מהתרסקות בזמן ריצה באמצעות שימוש ב-exceptions. כמו שבמכונית יש גם חגורת בטיחות וגם כריות אויר, כך גם בקוד יש מספר אמצעים שנועדו להגן עליו מקריסה, ו-exceptions הם אמצעי בטיחות נוסף.

פגשנו כבר ב-exceptions בפרקים הקודמים – בכל פעם שהקוד שלנו "עף", הודפס 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
```

הטקסט שמוקף באדום הוא שם ה-exception, והטקסט שכתוב לפניו הוא תיאור מפורט של ה-exception. אפשר לראות שבכל פעם שכתבנו משהו שה-interpreter של פייתון לא הצליח להתמודד איתו, קיבלנו exception. אפשר גם לראות, שקיימים exceptions מסוגים שונים, כך שאם ניסינו לחבר מחרוזת עם מספר קיבלנו שגיאה שונה מאשר במקרה שבו ניסינו לגשת אל משתנה שאינו קיים.

try, except

הפקודה try והפקודה התאומה שלה expect הן פקודות מיוחדת במינן, שמאפשרות לנו להריץ כל קוד שאנחנו רוצים ולדאוג למקרי הקצה בחלק קוד אחר. כך הקוד שלנו הופך לנקי הרבה יותר. הרעיון של פקודת try הוא כזה: "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".

נראה דוגמה שימושית. זוכרים שכתבנו סקריפט שמקבל מהמשתמש שם של תיקיה ומדפיס את כל הקבצים שנמצאים בה? חששנו ממצב 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 try-except. יש בקוד הזה שתי פקודות "מסוכנות". הראשונה היא הפקודה שקוראת את מה שנמצא ב-sys.argv[PATH]. אם המשתמש לא הכניס כלל פרמטרים, הקוד יקרוס עקב פניה לאינדקס שאינו קיים. הפקודה השניה היא כמובן הקריאה ל-os.listdir עם התיקיה שהמשתמש הכניס, שכאמור תקרוס אם התיקיה לא קיימת. נכניס את שתיהן לתוך try:

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

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

היינו יכולים לבצע את אותו תהליך בלי try-except, רק בעזרת if-else. הבדיקה הראשונה תהיה על אורך הרשימה argv והבדיקה השניה על ערך החזרה של פונקציה שבודקת אם התיקיה קיימת. היתרון של try, הוא

שנחסכת מאיתנו כתיבת קוד – אפשר לבדוק את כל המקרים שעלולים להוביל לקריסת קוד באמצעות פקודה יחידה.

מה הבעיה בצורת כתיבה זו? אם ארעה שגיאה לא נדע בדיוק מה השגיאה – ישנן שתי אפשרויות. לכן בקרוב נראה איך אנחנו כותבים את ה-except באופן שגם ייתן לנו מושג אודות השגיאה. אך קודם כל נחקור 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'
```

אמנם, אם thing אינו מחרוזת, הפונקציה תדפיס 'Error'.

נסבך מעט את הפונקציה. מה לדעתכם יהיה הערך של x אם הקוד ייכנס ל-except? במילים 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 שחלק מה-try דווקא כן מבוצע. כל מה שקדם לקפיצה ל-except בהחלט יתבצע. במקרה זה, מתבצעת

השורה בה משימים ב-x את הערך 1, ולכן זה ערכו כאשר הקוד קופץ ל-except. כמובן שהתוכנית אינה מגיעה לשורת הקוד בה x הינו 2.

השורה התחתונה היא, שגם אם התרחש exception, כל הפקודות שכבר התבצעו עדיין תקפות. המעבד אינו "חוזר אחורה"...

סוגים של Exceptions

נחזור 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', ישנן שתי אפשרויות לבעיה:

- המשתמש לא הזין כלל פרמטר, ולכן הגישה ל-sys.argv[PATH] תהיה לאינדקס לא חוקי ברשימה.
- המשתמש הזין תיקיה שגויה, לכן os.listdir יחזיר שגיאה.

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

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

צורת הכתיבה של except משמעותה "הכנס את הודעת השגיאה של ה-exception לתוך המשתנה e". כפי שראינו לכל exception יש הודעת שגיאה ייחודית, המשתנה e יכיל אותה. בדוגמה הבאה אפשר לראות איך אנחנו תופסים exceptions שונים ומטפלים בכל אחד מהם בנפרד. אם היתה 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? Exception היא class של פייתון (מאוחר יותר נלמד תכנות מונחה עצמים), לכן e הוא אובייקט שמכיל את כל השדות של Exception, כולל הודעת השגיאה.

finally

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

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


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

נסקור דוגמה. חישבו מה ידפיס הקוד הבא?

הקריאה הראשונה ל-do_something תרוץ ללא בעיות מיוחדות. לכן יורץ הקוד שנמצא ב-try וב-finally.

הקריאה השניה תגרום מיד לזריקת exception, ולכן למעבר ל-except ולאחר מכן לביצוע finally. תוצאת הריצה תהיה ההדפסה הבאה:

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

נסקור דוגמה מעניינת 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 ההרצה היא:

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

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

בקריאה השניה לפונקציה מתבצע תהליך דומה, קפיצה ל-finally וחזרה, אלא שהוא מתבצע מה-except.

כעת נבחן דוגמה נוספת, שממחישה את פעולת 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 תגיע אל ה-except, שם כפי שראינו בדוגמה הקודמת ה-interpreter של פייתון יקפוץ אל finally. ההבדל מהדוגמה הקודמת, הוא שבמקרה זה finally מכילה הוראת חזרה ולכן בכך תסתיים ריצת הפונקציה. תוצאת ההדפסה:

```
123 returns: Final
```

with

כזכור, כאשר למדנו על פתיחת קובץ באמצעות with open, אמרנו שגם אם מתרחשת שגיאה כלשהי במהלך הריצה עדיין הקובץ ייסגר. כיצד זה מתרחש?

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

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

שקול לקוד הבא:

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

כפי שאנחנו רואים, with כולל בתוכו finally, אשר דואג לסגירת הקובץ בכל מקרה – גם אם ארעה שגיאה במקום כלשהו בבלוק ששייך לו.

תרגיל מסכם – lazy student 2

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

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

46 + 19

15 * 3

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

46 + 19 = 65

15 * 3 = 45

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

python lazy_student.py homework.txt solutions.txt

הסקריפט יקרא את התרגילים מתוך homework.txt וישמור את הפתרונות אל solutions.txt. הניחו כמובן שיש שגיאות בפורמט של חלק מהתרגילים, או בשמות הקבצים. אסור לסקריפט שלכם לקרוס בשום אופן! אם יש בעיה בתרגיל, כיתבו במקום המתאים בקובץ הפתרונות הודעת שגיאה והמשיכו לתרגיל הבא.

תרגיל מסכם פייתון בסיסי – LogPuzzle

(קרדיט: google class, התאמה לגבהים: דנה אבן חיים, אורי לוי)

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

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

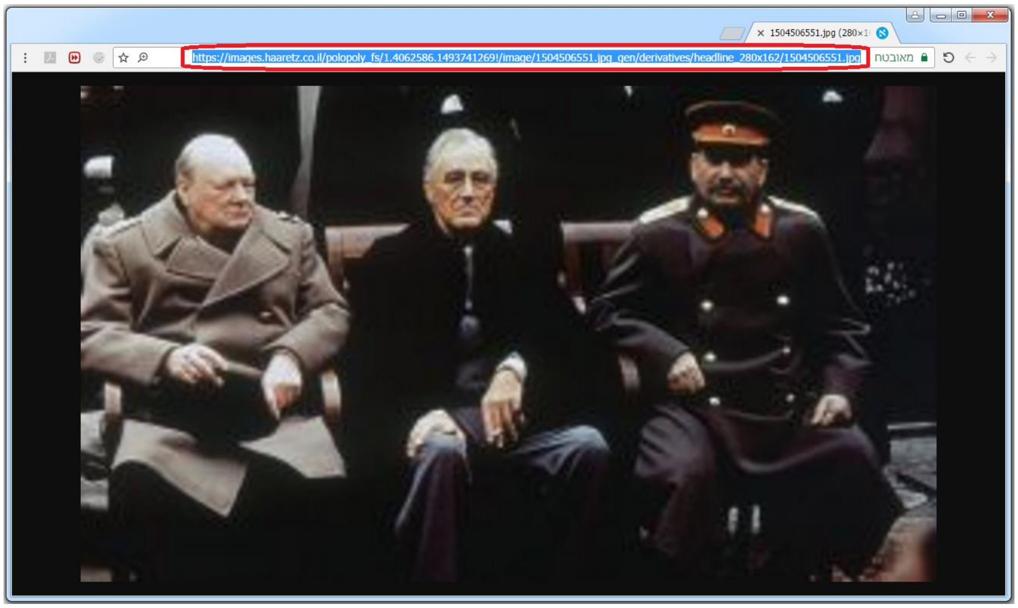
ראשית – מהו URL? אלו ראשי תיבות של Universal Resource Locator. לכל משאב באינטרנט יש URL משלו. לשם המחשה, יכול להיות שרת אינטרנט שכתובתו www.cyber.com. בתוך השרת הזה שמור קובץ pdf בשם file.pdf ועמוד html בשם page.html. ניתן יהיה להגיע לכל משאב לפי ה-URLים הבאים:

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

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

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

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



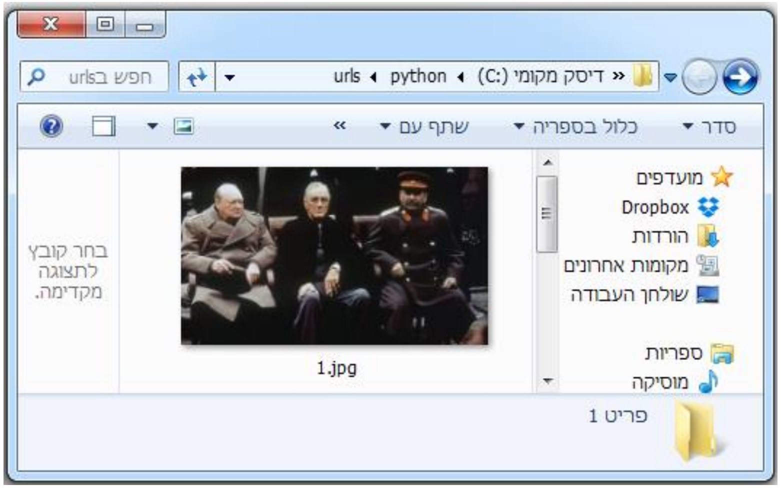
פיתחו סקריפט פייתון וכיתבו import urllib.

את ה-URL הדביקו כך שישמש בתור הפרמטר הראשון לפונקציה urlretrieve והוסיפו שם של קובץ אליו יש לשמור את הקובץ שאתם מורידים. כמובן שצריך לדאוג לכך ששם הקובץ יהיה בעל סיומת זהה לקובץ שאתם מורידים. לדוגמה, אם אנחנו מורידים קובץ עם סיומת jpg אז שם הקובץ אליו אנחנו שומרים צריך גם הוא להסתיים ב-jpg. סיומת הקובץ שאנחנו מורידים נמצאת תמיד בתווים האחרונים (הימניים) של ה-URL.

לדוגמה:

```
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 שנצטרך ללמוד כדי לפתור את הפאזל – חיבור תמונות לתמונה אחת. לשם כך נשתמש בקובץ html. כאשר רושמים בתוך הקובץ את שמות התמונות, הפעלת הקובץ מציגה אותן זו לצד זו.

כל מה שאנחנו צריכים לדעת כדי לכתוב את קובץ ה-html שנדרש בשביל לפתור את הפאזל, הוא כיצד להשתמש בתבנית הבאה:

```
<verbatim>
<html>
<body>
<img src="c:\python\urls\1.jpg"><img src="c:\python\urls\1.jpg">
</body>
</html>
```

הקלקה על הקובץ שיצרנו תפתח דפדפן ובו תוצג התמונה שהורדנו – פעמיים.

כלומר, כל מה שאנחנו צריכים לעשות זה להחליף את שמות הקבצים שבתבנית בשמות של התמונות, חלקי הפאזל, וכך נוכל להציג את התמונות זו לצד זו.

לאחר הקדמה קצרה זו, ניגש לתרגיל עצמו. בלינק הבא נמצאים שני קבצי לוג. בכל קובץ לוג ישנו טקסט, שמתחבאים בו url של תמונות jpg שצריך להוריד. לאחר ההורדה של התמונות ושמירתן לדיסק במקום שתבחרו, עליכם למיין אותן ואז להציג אותן זו לצד זו בהתאם לסדר המיון שנקבע בתרגיל.

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

כיצד לזהות את התמונות שצריך להוריד מקבצי 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 שאתם מורידים?

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

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

איך מתבצע המיון?

- עבור הקובץ logo, המיון הוא לפי סדר האלף-בית של הקבצים
- עבור הקובץ message, המיון הוא לפי סדר האלף-בית של המילה **השניה** בשמות הקבצים. לדוגמה הקובץ a-aaaa-zzzz.jpg יבוא **אחרי** הקובץ z-bbbb-cccc.jpg, כיוון שבמילון cccc קודם ל-zzzz.

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

כעת תורכם – פיתרו את הפאזל. בהצלחה!

פרק 10 – תכנות מונחה עצמים – OOP

בפרק זה נלמד כיצד מבצעים תכנות מונחה עצמים בפייתון. ייתכן ששמעתם על המושג "תכנות מונחה עצמים", או באנגלית Object Oriented Programming, אך הפרק אינו מניח ידע קודם בתכנות מונחה עצמים, רק שליטה בנושאים שנלמדו בפרקים הקודמים. הנושא הוא רחב, ולכן הפרק יתחלק ל-5 חלקים:

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

ב. כתיבת class בסיסי

ג. כתיבת class משופר

ד. ירושה – inheritance

ה. פולימורפיזם

מבוא – למה OOP?

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

אנחנו רוצים לכתוב תוכנית שתשמש בית ספר. יש צורך שנשמור את שמות כל התלמידים ואת הציונים שלהם במקצועות השונים. אנחנו יכולים לעשות זאת כך – לכל תלמיד ולכל מקצוע של תלמיד נקצה משתנה:

talmid1_name = 'Shimshon Gani'

talmid1_english = 90

talmid1_math = 95

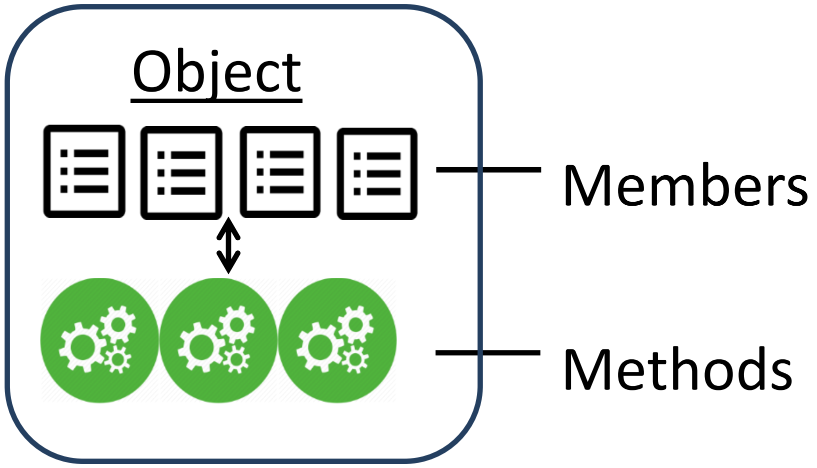
talmid1_geography = 85

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

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

אובייקט – object

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



אובייקט מורכב ממידע וממתודות שפועלות על המידע

נתקלנו כבר במתודות והסברנו שזה סוג מיוחד של פונקציות. אם כך, עכשיו אנחנו יכולים לחדד את ההסבר – מתודה היא פונקציה, אבל לא סתם פונקציה אלא פונקציה שמוגדרת כחלק מאובייקט ופועלת על האובייקט. כלומר, רק האובייקט מכיר אותה ורק הוא יודע איך לעבוד איתה. לדוגמה, ראינו שקובץ הוא אובייקט. לאובייקט מסוג file יש מתודה שקוראים לה read. אנחנו לא קוראים לה בצורה הזו:

```
read(filename)
```

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

```
filename.read()
```

בצורה זו אנחנו מנחים את ה-interpreter של פייתון – "גש אל האובייקט שנקרא filename. תמצא שיש לו מתודה בשם read. הפעל אותה על filename".

לסיכום, אובייקט מכיל גם מידע וגם מתודות. ראינו איך ניגשים למתודה של אובייקט, מיד נראה איך "מייצרים" אובייקט שכולל גם מתודות וגם מידע.

מחלקה – class

מחלקה, או class, הוא קטע קוד שמגדיר את כל ה-members והמתודות של אובייקט ואשר משותפים לו וליתר האובייקטים של אותה מחלקה. נמחיש זאת. ניקח את כוכב הלכת שבתאי. זהו כוכב לכת אחד מתוך מספר כוכבי לכת במערכת השמש שלנו. למרות שכל אחד מהם הוא שונה, לכולם יש מסה, רדיוס, מרחק מהשמש, זמן הקפה של השמש וכו'. אפשר להגדיר מחלקה של כוכבי לכת, 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 התבנית לשמירת כל המידע. אפשר לדמיין ש-planet היא דף מידע שמכיל את כל השדות שצריך בשביל להגדיר כוכב לכת.

Planet

שם הכוכב: ______________________________

מסה: ______________________________

מרחק מהשמש: ______________________________

זמן הקפה: ______________________________

טמפרטורה: ______________________________

אז מה ההבדל בין אובייקט למחלקה? מחלקה מתארת את המאפיינים של כל האובייקטים ששייכים למחלקה. כשהתוכנית רצה, בכל פעם שנגדיר אובייקט מסויים, יוקצה זיכרון במחשב בהתאם לכמות המידע שצריך כדי לשמור את כל המאפיינים של המחלקה. אפשר לומר שמחלקה היא כמו תבנית של עוגיות: התבנית אינה עוגיה ואי אפשר לאכול תבנית, אבל באמצעות התבנית אפשר ליצור עוגיות. בכל פעם שנשתמש בתבנית, חתיכה של בצק תקבל את הצורה של התבנית הזו. כמות הבצק שנקצה לעוגיה נקבעת על ידי התבנית, בדומה לכך שכמות הזיכרון שמוקצה 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 ממנה נוצר. לדוגמה, עוגיה היא instance של תבנית העוגיות ואילו שבתאי הוא instance של המחלקה planet.

בקרוב, נראה כיצד כותבים class, וכיצד יוצרים אובייקט שלה – כלומר instance של המחלקה הזו.

כתיבת class בסיסי

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

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

```python
class CrazyPlane:

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

נפרק את מה שכתוב לחלקים.

שם ה-class הוא CrazyPlane. שימו לב לכך שהשם מתחיל באות גדולה – זוהי הקונבנציה לקביעת שמות של מחלקות. מיד נסביר מהם ה-__init__ וה-self. לאחר מכן מוגדרים שני members (זוכרים שאמרנו שאובייקט מורכב ממתודות ומ-members?). הראשון הוא x והשני הוא y. כל אחד מהם מקבל ערך התחלתי – 0.

__init__

זוהי המתודה הראשונה שנכיר. השם init נגזר מ-initializer, או אתחול. אפשר לקרוא לה גם constructor או בנאי. בתוך __init__ נהוג לשים את האתחול של כל ה-members של המחלקה, כפי שראינו בדוגמה. זוהי מתודה מיוחדת בכך שהיא רצה באופן אוטומטי בכל פעם שנוצר instance של CrazyPlane בזכרון המחשב.

כפי שאנחנו רואים בדוגמה, המתודה מקבלת פרמטר שנקרא self. כדי להבין מהו, נשאל את עצמנו שאלה: איך פייתון יודע על איזה אובייקט להפעיל את המתודה? במילים אחרות, אנחנו יכולים לייצר כמה אובייקטים ששייכים לאותו class, לדוגמה כמו אותה תבנית עוגיות שמשמשת ליצירת עוגיות רבות. אם אנחנו רוצים להפעיל מתודה על עוגיה מסויימת, איך פייתון יודע על איזו עוגיה הוא צריך לפעול?

התשובה היא self. אנחנו מעבירים למתודה מצביע לאובייקט שעליו היא אמורה לפעול. אנחנו אומרים למתודה – "תפעלי על העוגיה הזו" בזמן שהאצבע שלנו מצביעה על עוגיה מסויימת. אי לכך, נעביר את self כפרמטר לכל מתודה של המחלקה שצפויה לרוץ על האובייקט הזה.

הוספת מתודות

לאחר שכתבנו את המתודה הראשונה, __init__, הגיע הזמן להוסיף מתודות שעושות פעולות שאינן אתחול. 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 המייצגים את מיקום המטוס ומעדכנת אותם רנדומלית (כמו שאמרנו, הטייסים השתגעו). המתודה get_position פשוט מחזירה את מיקום המטוס. שוב, נשים לב לכך שכל מתודה צריכה לקבל את self בתור פרמטר. כמו כן, נעשה שימוש במודול random ולכן נדרש לבצע import random בתחילת הקובץ.

Members

כפי שלמדנו זה עתה, ברגע שניצור אובייקט מסוג CrazyPlane יווצרו 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 בתוך לולאת ה-for של המתודה. ברגע שהלולאה מסתיימת, i אינו קיים יותר ולא ניתן לגשת אליו. זאת לעומת x, y שקיימים כל עוד האובייקט קיים. לכן x,y הם members.

יצירת אובייקט

לאחר שהגדרנו את המחלקה, הגיע הזמן להשתמש בתבנית שהגדרנו 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 שיצרנו אותו, אוטומטית מורצת פונקציית ה-__init__, אשר יוצרת את plane1.x ואת plane1.y ומאתחלת את ערכיהם ל-0. המתודה get_position פועלת על האובייקט plane1 ומחזירה את ערכי ה-x וה-y שלו, כלומר 0 ,0. נסו זאת בעצמכם – צרו אובייקט ובידקו שקיבלתם את ערכי ההתחלה שקבעתם.

נעשה בתוכנית שלנו שינוי קטן. במקום xpos נכתוב x ובמקום ypos נכתוב 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-x, y שהגדרנו בפונקציה main אינם x, y ששייכים ל-plane1. במילים אחרות, x אינו plane1.x ו-y אינו plane1.y. אין להם את אותו id(). גם אם נשנה את x, ערכו של plane1.x לא ישתנה.

תרגיל

כיתבו class של החיה האהובה עליכם (לדוגמה Cat, Dog וכו').

- הוסיפו מתודת __init__שתכלול את שם החיה (לדוגמה Kermit) ואת גיל החיה
- הוסיפו מתודת birthday, שתעלה את גיל החיה ב-1
- הוסיפו מתודת get_age שתחזיר את גיל החיה

שימו לב שבשלב הזה כל החיות שתיצרו באמצעות התבנית של המחלקה יקבלו את אותו שם אשר מופיע ב-__init__. בקרוב נלמד כיצד נותנים לכל חיה שם שונה בשלב האתחול.

כתיבת class משופר

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

- ליצור members "מוסתרים"
- להפוך את ה-class לקובץ שניתן לייבא על ידי import

- לקבוע ערכים התחלתיים לאובייקט חדש
- ליצור פקודת הדפסה מיוחדת ל-class שלנו
- ליצור מתודות accessor ו-mutator

יצירת members "מוסתרים"

קטע הקוד הבא דורס את נתוני המיקום של המטוס:

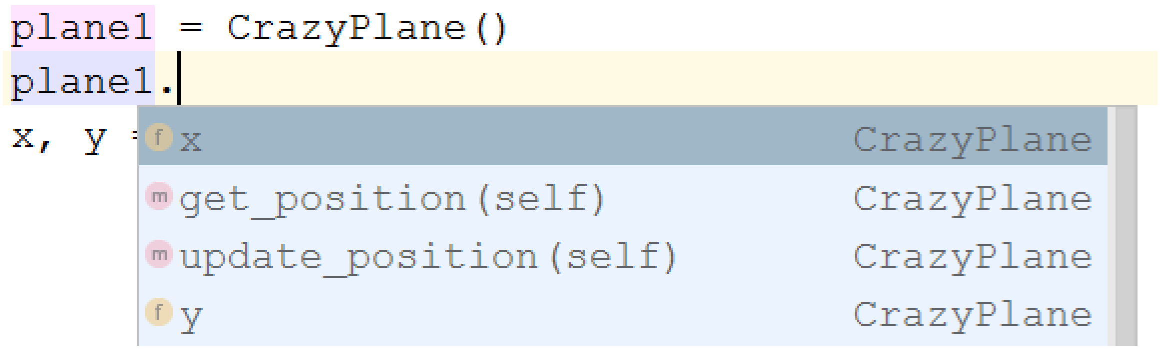
```
plane1 = CrazyPlane()
plane1.x = 10
plane1.y = 10
x, y = plane1.get_position()
```

המתודה get_position תחזיר את הערכים 10, 10. בעולם האמיתי, תוצאה של קוד כזה עלולה להיות התנגשות בין מטוסים. עלינו למצוא דרך "להסתיר" את נתוני המיקום של המטוס, כך שהתוכנה שעושה בהם שימוש לא תשנה אותם בטעות.

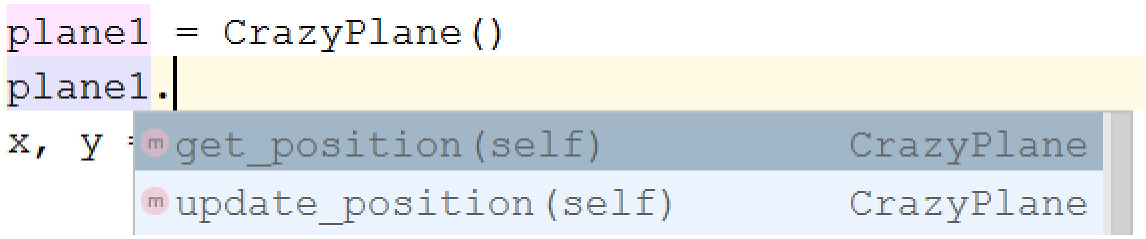
בפייתון, הסתרה של members מתבצעת באמצעות תחילית __ (קו תחתי כפול). נסתיר את המשתנים של CrazyPlane:

```python
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 מתודות של CrazyPlane מכירות את ה-members הללו. מי שמשתמש ב-class שלנו כבר לא יכול לראות שהם קיימים. שימו לב להבדל: לפני ההסתרה, כתיבה של "plane1." העלתה אפשרויות שונות, ביניהן x, y.



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



האם זה אומר שכבר אי אפשר לגשת אל members מוסתרים? לא. פייתון מאפשרת לנו גם אפשרות זו. בשפות עיליות ישנו מושג שנקרא "private", שמשמעותו משתנים או מתודות שאי אפשר לגשת אליהן מחוץ ל-class. בפייתון אין private, יש רק הסתרה. אם מתעקשים, אפשר לגשת אל members אפילו אם הם מוסתרים.

כזכור פונקציית dir מחזירה את כל המתודות וה-members ששייכים לאובייקט מסויים. אם כך, נעשה dir(plane1) ונקבל:

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

שימו לב לשני האיברים הראשונים ברשימה, אשר מסתיימים ב-"__x" וב-"__y". כן, נראה דומה למה שאנחנו מחפשים. ננסה 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 מתקבל (5,5), מה שמעיד על כך שאכן שינינו את הערכים.

האם מדובר בבאג של פייתון? לא, כך פייתון תוכננה. הרעיון הוא לאפשר גמישות מקסימלית למתכנתים. פייתון 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, והקדשתם את הזמן להתגבר על ההסתרה, אז בבקשה – שנו כל מה שתרצו". יכולת זו שימושית רק במקרים נדירים, אבל במקרה הצורך היא מאפשרת לנו, לדוגמה, לבצע שינויים בספריות פייתון שיש בהם באגים, בלי להזדקק לקוד המקור.

אם כך, משתנים מוסתרים בפייתון הם למעשה דרך שלנו לסמן כי אין לשנות את תוכן המשתנה שלא על ידי קוד של המחלקה עצמה. עם זאת, זהו רק סימון – ומשתמש חיצוני יוכל לשנות את ערכי המשתנה המוסתר, אם יבחר בכך.

שימוש ב-accessor וב-mutator

סקרנו שתי שיטות גישה ל-members של מחלקה.

השיטה הראשונה, היא לגשת אל ה-members ישירות. לדוגמה plane1.x היא גישה ישירה ל-member של plane1. אם מי שתכנת את המחלקה CrazyPlane דאג להסתיר את x, עדיין נוכל לגשת אליו בעזרת plane1._CrazyPlane__x.

השיטה השניה, היא באמצעות שימוש במתודות שנמצאות במחלקה ומשמשות במיוחד לטובת קריאה ושינוי של members. לדוגמה, get_position היא דוגמה למתודה כזו. ראינו שאם נקרא ל-plane1.get_position() נקבל את ערכי המיקום, למרות שהם מוסתרים. למתודה שמאפשרת קריאה של members של מחלקה קוראים accessor ונהוג שהיא מתחילה ב-get.

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

יש דיון נרחב האם שימוש ב-accessors וב-mutators (אשר נקראים גם getters, setters) הוא נכון, לא רק בשפת פייתון אלא באופן כללי בתכנות 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 הוא שביכולתו של המתכנת לשלוט בערכים המוכנסים ולוודא תקינות. לדוגמה, אם המתכנת יודע שיש מיקום בעייתי שאסור שמטוס יימצא בו הוא יכול למנוע זאת באמצעות קוד מתאים. הקוד הבא לא מאפשר להזין למטוס מיקום שלילי או את מיקום 4,4 כיוון שנמצא שם מגדל שאינו 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'
```

מאידך, טוענים כנגד 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 לפסוק בשאלה מהי השיטה העדיפה. נציין שחשוב להכיר את שתי השיטות מכיוון ששתיהן נפוצות בקוד בו אתם עשויים להיתקל בעתיד, וחשוב לדעת להשתמש ב-accessors ו-mutators, ולו בשביל לקבל בהמשך החלטה לא להשתמש בהם.

יצירת מודולים ושימוש ב-import

הקוד שלנו כרגע כולל גם את הגדרת המחלקה, גם את ה-main ואולי גם עוד פונקציות וקבועים שהגדרנו. מדוע כדאי לשנות את זה? משום שאם נרצה להשתמש במחלקה שלנו בתוכנית אחרת, נצטרך להעתיק את כל הקוד שלנו ואז לשנות מה שצריך. לא פתרון נוח. היינו רוצים שתוכנית אחרת תוכל "לייבא" רק את המחלקה שהגדרנו, ואולי כמה קבועים שקשורים אליה – וזהו.

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

נשמור קובץ פייתון בשם planes.py. לטובת ההמשך הגדרנו מחלקה נוספת, NormalPlane. להלן הקוד שבקובץ:

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


class CrazyPlane:

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

    def update_position(self):
        self.__x += random.randint(-1, 1)
        self.__y += random.randint(-1, 1)

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

    def set_position(self, x, y):
        if (x, y) == CONTROL_TOWER_LOCATION:
            print 'Location of the tower'
        elif x < 0 or y < 0:
            print 'Illegal location'
        else:
            self.__x = x
            self.__y = y
            print 'Position set'
```

```
class NormalPlane:

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

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

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


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


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

כעת נגדיר תוכנית שמשתמשת במחלקה שנמצאת בקובץ plane.py:

```
import planes


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


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

בשורה הראשונה אנחנו מבצעים import לקובץ planes. כעת זו הזדמנות נהדרת להסביר את התפקיד של if '__main__' == __name__. בקובץ planes.py קיימת פונקציה בשם main. כאשר אנחנו מייבאים את הקובץ, אנחנו לא מעוניינים להפעיל את main של הקובץ המיובא – יש לנו main משלנו שאנחנו עובדים איתו. אנחנו רוצים רק את ההגדרות של המחלקות, וקבועים כלשהם אם הם קיימים, לדוגמה מיקום המגדל. תנאי ה-if הנ"ל מודיע לפייתון "אם הגעת לקובץ הזה תוך כדי ריצת התוכנית ולא בעקבות import, תריץ את main" – בדיוק מה שרצינו שיקרה.

לאחר שביצענו import, אנחנו יכולים להשתמש במחלקות שייבאנו אל התוכנית שלנו. השימוש במחלקות מצריך שינוי קטן בקוד – לפני שאנחנו מגדירים אובייקט מסוג CrazyPlane צריך לסמן שהאובייקט שייך לקובץ planes שייבאנו, כפי שניתן לראות בשורה השלישית. רגע – איך פייתון לא יודע בעצמו ש-CrazyPlane הוא מחלקה שמוגדרת ב-planes? ובכן, פייתון יודע, אבל פייתון מניח שיכול להיות שישנה עוד מחלקה בשם זה בקובץ אחר. מקרה כזה עלול בהחלט להתרחש, במקרה שיש מתכנתים רבים שכותבים מודולים בנפרד. לכן, פייתון דורש שנציין בפירוש לאיזה קובץ אנחנו מתכוונים. ואם בכל זאת אנחנו רוצים לוותר על ציון שם הקובץ? אפשר לוותר על כך בשינוי קטן. שימו לב לצורה הבאה של import:

```
from planes import CrazyPlane


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

כעת פייתון יודע שאם אנחנו כותבים CrazyPlane אנחנו מתכוונים רק למחלקה הנ"ל מתוך planes, לכן אפשר לוותר על ציון שם הקובץ. שימו לב לכך שאם אכן היה קיים קובץ אחר שיש בו מחלקה בעלת אותו שם, כרגע "דרסנו" את שם המחלקה בקובץ האחר ולא נוכל להשתמש בו עוד.

ביצוע import רק למחלקה אחת מתוך הקובץ – ולשם כך הוספנו לקובץ עוד מחלקה – גורם לכך שכעת רק המחלקה CrazyPlanes יובאה. המחלקה השניה, NormalPlane, נותרה לא מוכרת לתוכנית הראשית.

לעיתים תיתקלו בייבוא 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 שיכול להיות שבשני קבצים שונים ישנן שתי מחלקות בעלות שם זהה. לדוגמה, המחלקה CrazyPlane מוגדרת גם בקובץ planes.py וגם בקובץ airports.py. אם נייבא את שני הקבצים באמצעות * import, מה יקרה כאשר נרצה ליצור אובייקט מסוג CrazyPlane? האם מה שמוגדר ב-planes או ב-airports? התשובה היא שהקובץ שייבאנו אחרון "דורס" את ההגדרות הקודמות. לכן, שיטה זו אינה מומלצת.

אתחול של פרמטרים

אפשרויות הטיסה של המטוסים השתנו, כאשר נפתח לרווחת הציבור שדה תעופה נוסף. כעת אנחנו רוצים לאפשר לחלק מהמטוסים להמריא משדה התעופה החדש. נניח שהמיקום של שדה התעופה הוא (5, 5). כלומר, אנחנו צריכים להעביר את קואורדינטות ההמראה 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 שממריא תהיה שדה התעופה הישן, שמיקומו (0,0). נעשה שינוי קל במתודת האתחול:

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

כעת, פייתון בודק אם כאשר אנחנו יוצרים מטוס חדש אנחנו מעבירים את המיקום כפרמטר. אם כן – המיקום שהעברנו קובע, אחרת – מיקום ברירת המחדל קובע. נוכל להגדיר מטוסים חדשים באופנים הבאים:

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

האובייקט elal יקבל את ערכי המיקום של ברירת המחדל – כלומר 0, 0, ואילו האובייקט american יקבל את ערכי ההתחלה שהעברנו לו – מיקום שדה התעופה של ניו יורק.

המתודה __str__

ננסה לעשות print למטוס שהגדרנו, באמצעות הפקודה print american:

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

איך פייתון יודע מה להדפיס כאשר אנחנו עושים print לאובייקט מסויים? לכל אובייקט ישנה מתודת __str__. אנחנו יכולים לדרוס את __str__ ולהחליף אותה בכל צורת הדפסה שנרצה. הבה נבצע זאת:

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

הסבר: ראשית, המתודה צריכה להחזיר ערך, שהוא הערך שיודפס. לכן ישנו במתודה return. הערך שמוחזר הוא מחרוזת, שחלקה קבוע וחלקה מקבל את הערך שמחזירה המתודה get_position ומעביר אותו לפורמט המחרוזת.

אם עכשיו נבצע print, נקבל:

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

זהו. יותר יפה ממה שהחזירה פקודת ההדפסה לפני השינוי? ☺

__repr__

המתודה repr, קיצור של represent, היא דרך נוספת שמשמשת להדפסה של אובייקטים. אך בעוד __str__ משמשת לצורך הצגה יפה של נתונים ל*משתמש*, המתודה __repr__משמשת להצגת נתונים ל*מתכנת*, על מנת לסייע בדיבוג.

נמחיש זאת באמצעות הדוגמה הבאה (קרדיט לרעיון – http://www.geeksforgeeks.org/str-vs-repr-in-python):

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

הסבר: כאשר ביצענו print, באופן אוטומטי נקראה המתודה __str__ של datetime, אשר בה מוגדר הזמן בפורמט נוח לקריאה. לעומת זאת, כאשר בדקנו מה ערכו של t קיבלנו את התוצאה של __repr__ של datetime, אשר נתנה לנו יותר מידע על מה שמתרחש "מאחורי הקלעים". ניתן לכתוב את repr עבור כל מחלקה שאנחנו כותבים, על ידי הגדרה של __repr__, בדומה לדרך שבה הגדרנו את __str__.

יצירת אובייקטים מרובים

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

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

מטוס אלעל ממריא מנקודת ברירת המחדל שלנו ואילו היתר ממריאים ממיקומים שמועברים כפרמטרים.

אם אנחנו רוצים לבצע פעולה כלשהי על כל המטוסים, נכניס אותם לרשימה (אם אנחנו רוצים לאפשר הוספת מטוסים – אחרת tuple) ואז נוכל לעבור על כולם בלולאה:

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

תרגיל סיכום ביניים – כתיבת class משופר

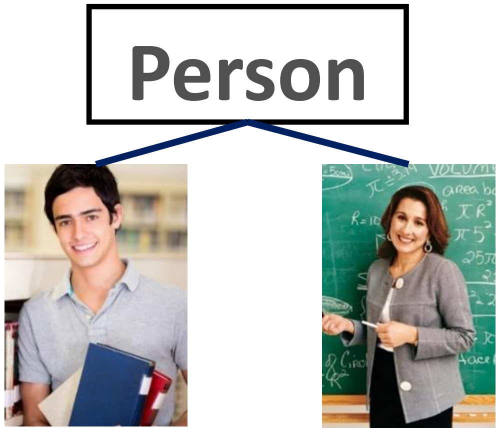
שפרו את ה-class של החיה שיצרתם:

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

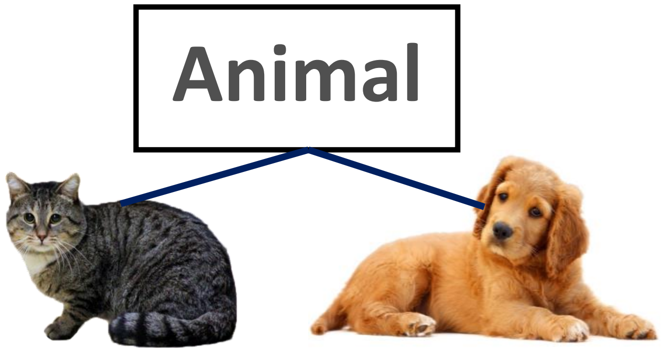
ירושה – inheritance

לפעמים מחלקה היא סוג ספציפי של מחלקה אחרת. לדוגמה – חתול הוא דוגמה ספציפית של חיה, ומטוס הוא דוגמה ספציפית של כלי תחבורה. הרעיון של ירושה מאפשר לנו לקחת מחלקה קיימת וליצור ממנה מחלקה חדשה, שכוללת את כל התכונות שקיימות במחלקה שירשנו ממנה ועוד תכונות נוספות, שהן מיוחדות רק למחלקה החדשה שיצרנו. המחלקה החדשה, זו שיורשת מהמחלקה הקיימת, נקראת subclass. המחלקה שממנה ירשנו נקראת superclass. לדוגמה:

- Teacher ו-Student הם subclasses של ה-superclass שנקרא "Person"



- Cat ו-Dog הם subclasses של ה-superclass שנקרא "Animal"



כדי להמחיש איך מייצרים subclass, בתור 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 יורש מ-object. זהו הנוהג בגרסת פייתון נמוכה מ-3 (גרסת הפייתון שלנו היא 2.7). בגרסה 3 פייתון כבר מניח מראש שהירושות הן 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. נגדיר שהמחלקות החדשות יורשות מ-Person – שימו לב שמציינים זאת בסוגריים ליד הגדרת המחלקה:

```
class Teacher(Person):

class Student(Person):
```

שאלה למחשבה: נניח שהגדרנו את Teacher באופן שאין בו כלום, כך:

```
class Teacher(Person):
    pass
```

מה יבצע קטע הקוד הבא?

```
teacher1 = Teacher()
print teacher1
```

תשובה: כאשר יצרנו אובייקט מסוג Teacher, פייתון מחפש את ה-__init__ של Teacher. כיוון שלא הגדרנו לו __init__, פייתון הולך אל המחלקה ממנה Teacher יורש, כלומר Person, ומפעיל את ה-__init__ של Person. באתחול של Person נקבעים משתני ברירת מחדל ו-teacher1 יורש אותם. נציין שאילו ל-Teacher כן היה __init__, לא היה מופעל אוטומטית ה-__init__ של המחלקה ממנה הוא ירש. לגבי פקודת ההדפסה, למרות שלא הגדרנו שום מתודה ל-Teacher, הוא יורש את __str__ מ-Person ולכן יודפס:

```
Person Tal is 20 years old
```

כפי שהחלטנו, צריך שלכל מורה יהיה שכר. לכן נוסיף לו מתודת אתחול. המתודה צריכה לגרום לכך שמה שאפשר נירש מ-Person ומה שספציפי למורה, נגדיר. נבצע זאת כך:

```
class Teacher(Person):

    def __init__(self, name, age, salary):
        Person.__init__(self, name, age)
        self.__salary = salary
```

נסביר מה ביצענו. הגדרנו מחלקה בשם Teacher שיורשת מ-Person. במתודת ה-__init__ הפעלנו את מתודת האתחול של Person עם פרמטרים ששלחנו לה (name, age). כיוון ש-Person לא יודעת מה לעשות עם פרמטר ה-salary, הגדרנו ל-Teacher משתנה נוסף. למעשה, משתנה זה הוא כל מה שמבדיל כרגע בין Teacher ל-Person. 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 לפונקציית ה-__init__ של המחלקה ש-Teacher יורשת ממנה, כלומר ל-Person:

```
class Teacher(Person):

    def __init__(self, name, age, salary):
        super(Teacher, self).__init__(name, age)
        self.__salary = salary
```

הסיבות המרכזיות לשימוש ב-super הן:

א. אנחנו רוצים לקרוא למתודה של מחלקה, אשר המחלקה שלנו דרסה עם מתודה בעלת שם זהה.

ב. במקרים של מחלקות שיורשות ממספר מחלקות, שימוש ב-super הוא הכרחי כדי לוודא שמבוצעים __init__ של כל המחלקות מהן ירשנו.

מה יודפס אם נריץ כעת את הקוד הבא?

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

תשובה:

```
Person Barak is 40 years old
```

וזאת מכיוון ש-Teacher יורש את ה-__str__ של Person.

תרגיל מסכם – ירושה

צרו את המחלקה Student שיורשת מ-Person. עליכם:

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

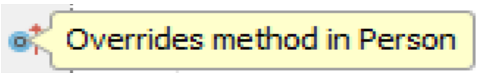
פולימורפיזם

הסיבה שהגדרנו ל-Person מתודה בשם say היא בדיוק לטובת חלק זה. נפעיל את say על אובייקט המורה שלנו, ונקבל את המחרוזת ":) Hi". אבל נרצה שהמורה יאמר משהו ייחודי, לדוגמה "Good morning cyber students!", וכמובן שיירש את כל התכונות של Person.

כדי לגרום למתודה say לעבוד כמו שאנחנו רוצים, נגדיר אותה מחדש בתוך Teacher:

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

שימו לב לסימן העיגול הכחול לצד המתודה. אם נעמוד עליו עם העכבר, נקבל הסבר:



במילים אחרות – דרסנו את המתודה say של Person.

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

אנחנו יכולים להגדיר מחלקה נוספת שתירש מ-Person, לדוגמה Student. כמו של-teacher יש מתודת say ייחודית, גם ל-Student אפשר להגדיר say. כעת אפשר להבין מדוע קוראים לפעולה זו "פולימורפיזם". "Poly" משמעותו "הרבה" ואילו "morph" משמעותו "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.

לעיתים אנחנו רוצים ש-subclass יפנה למשתנים מוסתרים של ה-superclass שלו. חשוב לדעת שמשתנים מוסתרים הינם מוסתרים גם מהיורשים. לכן, אם אנחנו רוצים להשתמש ב-members של superclass, ניתן לבחור לעשות זאת באמצעות מתודות accessor (כזכור יש דיון נרחב בשאלה האם אכן רצוי לעשות זאת). לדוגמה, נעדכן את מתודת ה-str של Teacher:

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

הפונקציה isinstance

בבית הספר המעולה של נווה חמציצים נפתחה מגמת סייבר. נגדיר תלמיד שלומד במגמה בתור CyberStudent, כלומר אובייקט שיורש 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 שהציון שלו בסייבר טוב יקבל הודעת "Wow!".

מתכנת הכניס לרשימה בשם students את האובייקטים של כל התלמידים, גם אלו שנמצאים במגמת סייבר וגם אלו שאינם. לאחר מכן המתכנת כתב את הקוד הבא. האם הקוד יעבוד באופן תקין?

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

תשובה:

כאשר נריץ את הקוד, נקבל שגיאת הרצה -

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

הסיבה היא שאמנם ל-CyberStudent יש מתודה בשם 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 מגיעה לעזרתנו. הפונקציה מקבלת שם של אובייקט ושם של מחלקה, ובודקת אם האובייקט שייך למחלקה או יורש מהמחלקה הזו. אם כן – מוחזר True, אחרת – False. חשוב לציין, שאובייקט תמיד שייך ל-superclass של המחלקה שלו.

כעת נוכל לכתוב מחדש את הלולאה שלנו, כך שרק אם student הוא מסוג CyberStudent תתבצע קריאה ל-get_cyber_grade:

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

נסיים בשאלה למחשבה. מוגדרים התלמידים הבאים:

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

מה תהיה התוצאה של כל אחת מהבדיקות הבאות?

```python
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 כמובן True.

הבדיקה השלישית תחזיר True מכיוון ש-noam הוא Student, כלומר subclass של Person.

הבדיקה הרביעית תחזיר False, כיוון ש-noam הוא Student, ואינו יורש מ-CyberStudent.

הבדיקה החמישית תחזיר True, מכיוון ש-daniel הוא CyberStudent, אשר יורש מ-Student.

תרגיל מסכם פולימורפיזם – BigCat (קרדיט: שי סדובסקי)

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

- אם המשתנה הוא מספר – המתודה מחזירה את המספר

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

לדוגמה, עבור ההגדרה:

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

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

כעת הגדירו מחלקה 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 יוחזר "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 פייתון.

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

תוכן העניינים של פרק זה הינו כדלקמן:

א. כתיבת שלד של PyGame

ב. הוספת תמונת רקע

ג. הוספת צורות גאומטריות

ד. הזזת 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

כתיבת שלד של PyGame

```
import pygame

# Constants
WINDOW_WIDTH = 700
WINDOW_HEIGHT = 500

# Init screen
pygame.init()
size = (WINDOW_WIDTH, WINDOW_HEIGHT)
screen = pygame.display.set_mode(size)
pygame.display.set_caption("Game")

pygame.quit()
```

ראשית עלינו לבצע import למודול PyGame.

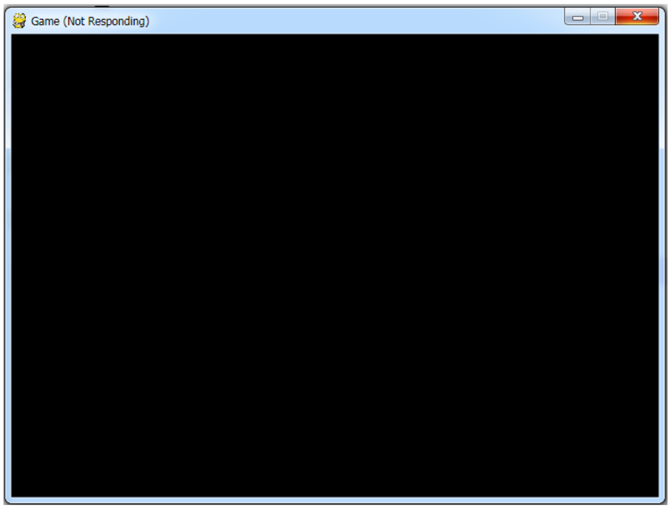
נגדיר בתור קבועים את גודל החלון – כמות הפיקסלים לרוחב ולגובה החלון.

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

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

מיד לאחר מכן תתבצע הפקודה ()PyGame.quit.

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



אם נרצה שהמסך יישאר, נכניס לפני שורה 13 לולאה אינסופית, אך שימו לב שכרגע – באופן זמני – הדרך היחידה לסגור את התוכנית היא באמצעות לחצן העצור של PyCharm או באמצעות מנהל המשימות. לחצן סגירת החלון שבקצה העליון של המסך עדיין אינו עובד. מיד נסדר את זה ☺

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

```
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 רשימה, ולכל פעולה כזו מוצמד סוג של ארוע – event.type. לדוגמה, לחיצה על כפתור סגירת המסך היא סוג ארוע בשם PyGame.QUIT. המתודה PyGame.event.get() מספקת לנו את הרשימה של כל הפעולות שהמשתמש ביצע מאז הפעם האחרונה שקראנו לה. הקוד לעיל עובר על כל הפעולות שהמשתמש ביצע ובודק אם אחת מהן היא PyGame.QUIT. אם כן – הקוד יוצא מהלולאה האינסופית.

שינוי רקע

הרקע השחור אמנם נחמד, אך לעתים נרצה לצבוע את המסך שלנו בצבע אחר, לדוגמה בצבע לבן.

נצטרך להגדיר את הצבע הלבן בתור tuple של 3 מספרים -

WHITE = (255, 255, 255)

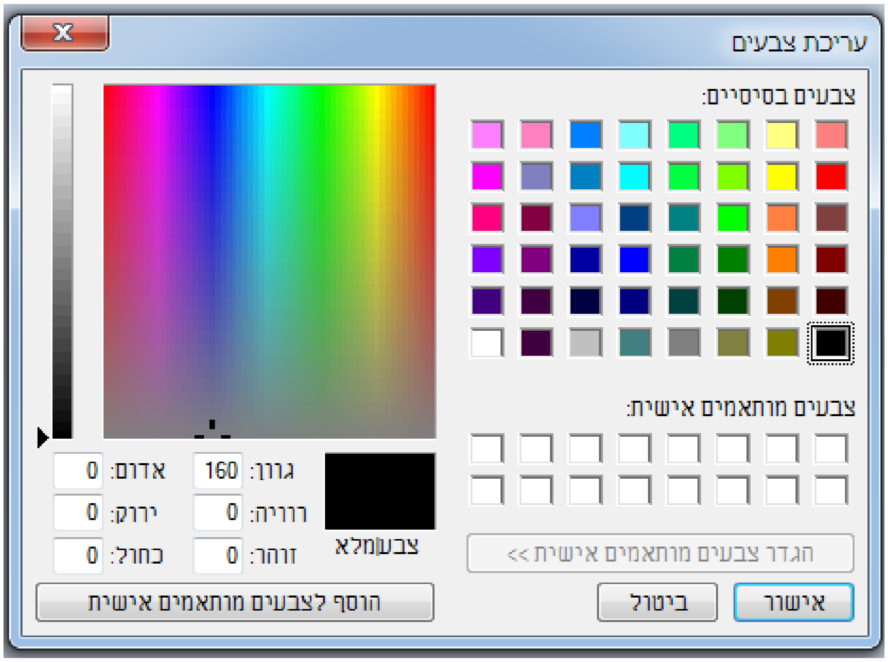
אלו ערכי RGB (Red, Green, Blue) של הצבע הלבן. כל צבע מוגדר על ידי שלישיית מספרים שונה. לדוגמה:

RED = (255, 0, 0)

וכמובן:

BLACK = (0, 0, 0)

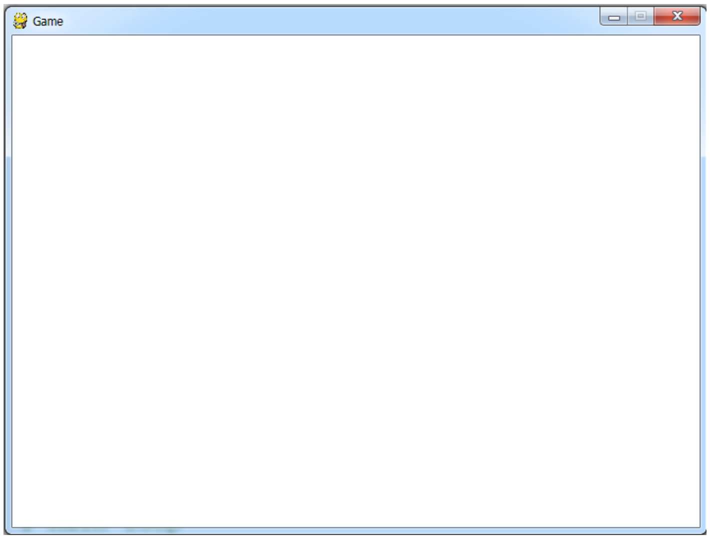
אפשר למצוא את שלישיית ה-RGB של כל צבע שתרצו באמצעות תוכנת paint (צייר) הבסיסית שמגיעה עם Windows, תחת תפריט "עריכת צבעים":



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

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

והתוצאה:

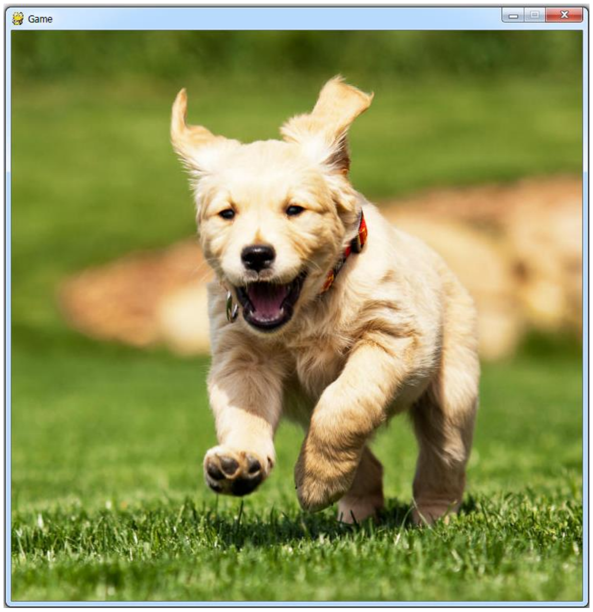


מה מבצעת שורה 16?

מסך המחשב קורא את המצב של כל פיקסל ממקום מוגדר בזיכרון המחשב (שנקרא "video memory"). בשורה 15 אנחנו עדיין 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 שמשנה בפועל את מצב המסך שלנו.

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

רקע לבן הוא משעמם קצת, בואו נעלה תמונת 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.

שנו את הקבועים שנמצאים בראש התוכנית – גובה ורוחב החלון – על מנת להתאים אותו לגודל התמונה שבחרתם. לאחר ששינינו את גודל המסך כדי שיתאים לתמונה, נטען אותה כתמונת רקע.

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

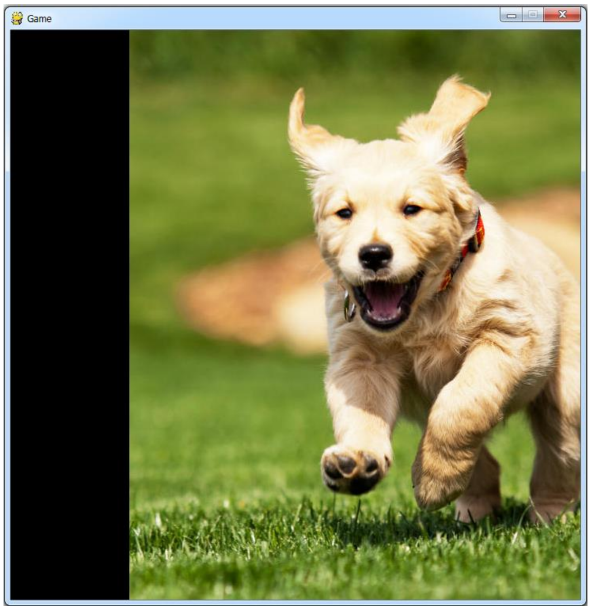
```
IMAGE = 'example.jpg'
```

וכעת נטען את התמונה כתמונת רקע:

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

שורה 17 טוענת את התמונה לתוך האובייקט screen, החל ממיקום (0, 0). זוהי הפינה השמאלית העליונה של המסך. כיוון שדאגנו מראש שגודל המסך יהיה שווה לגודל התמונה, תמונת הרקע תופסת כעת את כל גודל המסך.

שימו לב מה היה קורה אילו היינו מנסים לטעון את התמונה ממקום התחלתי (150, 0):



כלומר ככל שאנחנו עולים בפרמטר הראשון אנחנו זזים ימינה על המסך. ככל שאנחנו עולים בפרמטר השני אנחנו זזים למטה. אם הגדרנו מסך בגודל 720 על 720, הפיקסל הימני התחתון הוא במספר (719, 719).

הוספת צורות

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

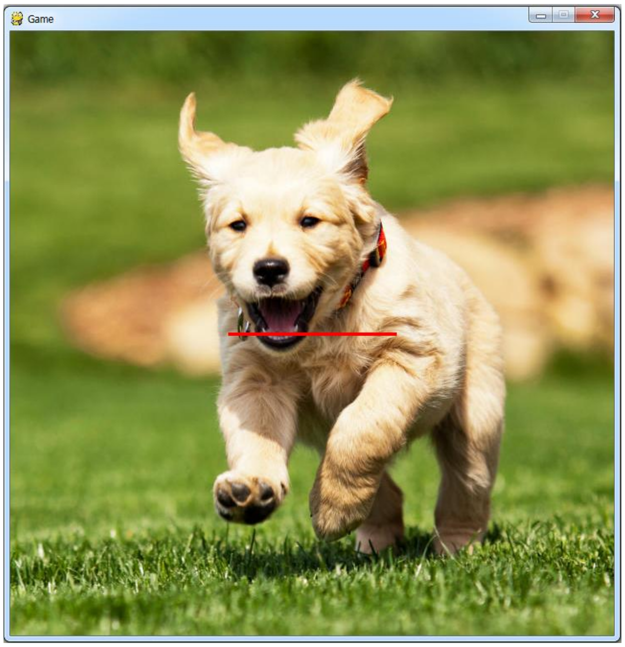
נתחיל בכך שנצייר קו. לשם כך, נשתמש במתודה PyGame.draw.line. מתודה זו מקבלת בתור פרמטרים:

- משטח – אובייקט של PyGame עליו יצויר הקו
- צבע הקו
- נקודת התחלה
- נקודת סיום
- עובי הקו (ברירת מחדל – 1)

כך:

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

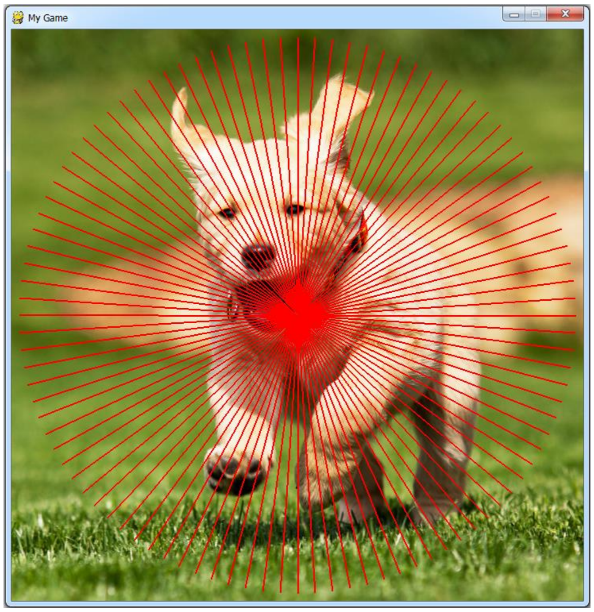
כלומר, כדי לצייר קו אדום מנקודה [360 ,260], אל נקודה [360 ,460] ובעובי 4:



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

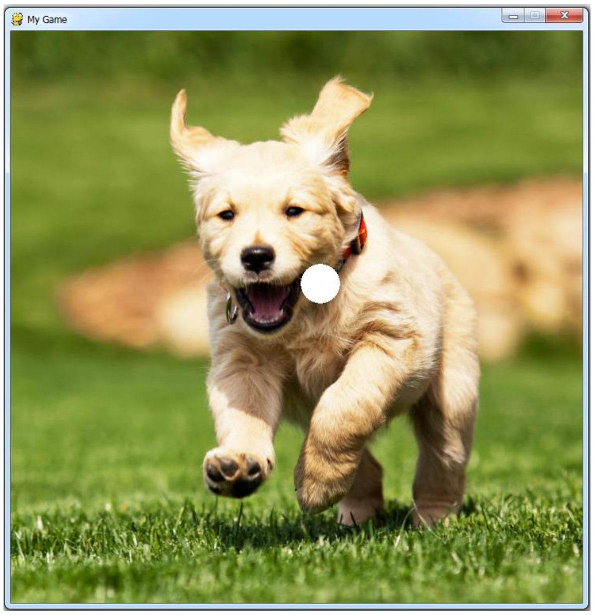
תרגיל

צרו בעצמכם את הדוגמה הבאה, המורכבת מ-100 קווים באורך 350 היוצאים ממרכז התמונה. טיפ: השתמשו ב- import math כדי לחשב את נקודת הסיום של כל קו, בעזרת הפונקציות sin ו-cos.



תרגיל

באמצעות התיעוד שנמצא בקישור הבא, ציירו עיגול במקום כלשהו על המסך.
https://www.PyGame.org/docs/ref/draw.html


My Game

תזוזה של גרפיקה

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

מציירים צורה במקום מסויים. לאחר זמן מה מוחקים אותה, ומציירים אותה שוב במקום אחר, ליד המקום הקודם. כך נדמה לצופה שהצורה "זזה".

נתייחס לשלבים השונים:

מחיקת צורה וציור שלה מחדש: כל מה שעלינו לעשות כדי למחוק צורה, הוא פשוט לצייר את תמונת הרקע מעליה. להזכירכם, לצייר צורה מעל תמונת הרקע כבר למדנו.

המתנה של פרק זמן מוגדר בין המחיקה לציור: על מנת לעשות זאת נזדקק לשעון (טיימר), שיתזמן את פרק הזמן בו יש לחזור על פעולות המחיקה והציור מחדש.

כדי להוסיף שעון, נגדיר:

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

נוסיף לקבועים שלנו את:

```
REFRESH_RATE = 60
```

כלומר, המסך מתעדכן 60 פעמים בשניה.

נבנה מחדש את 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 הצורה שאנחנו מעוניינים להזיז).

במקום שורות 37 ו-38 צריך לבוא קטע קוד כלשהו שמזיז את הצורה שלנו. לדוגמה, אפשר להזיז מעט את העיגול שציירנו בתרגיל הקודם. בדוגמת הקוד לעיל, אנו מזיזים את העיגול בפיקסל אחד למטה ובפיקסל אחד ימינה בכל איטרציה של הלולאה, רק בכדי להמחיש את העקרון. שימו לב שאנחנו לא בודקים שמיקום העיגול הוא הגיוני (לדוגמה, שהעיגול נמצא בתוך המסך...).

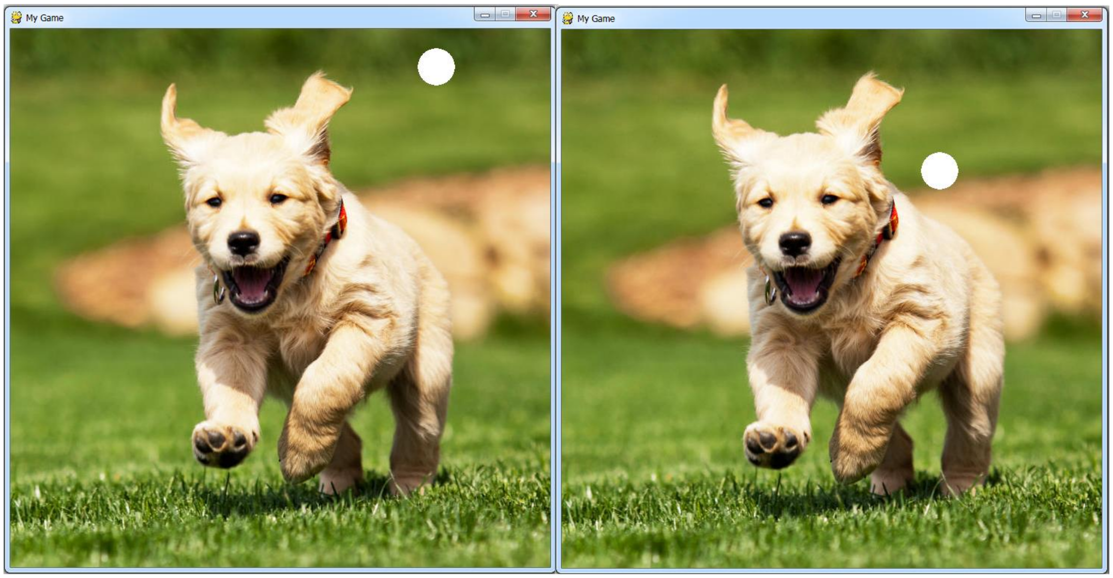
בשורה 40 אנחנו מציירים את העיגול במיקומו החדש.

שורה 41 גורמת לעדכון זיכרון הווידאו של המסך, כפי שראינו קודם.

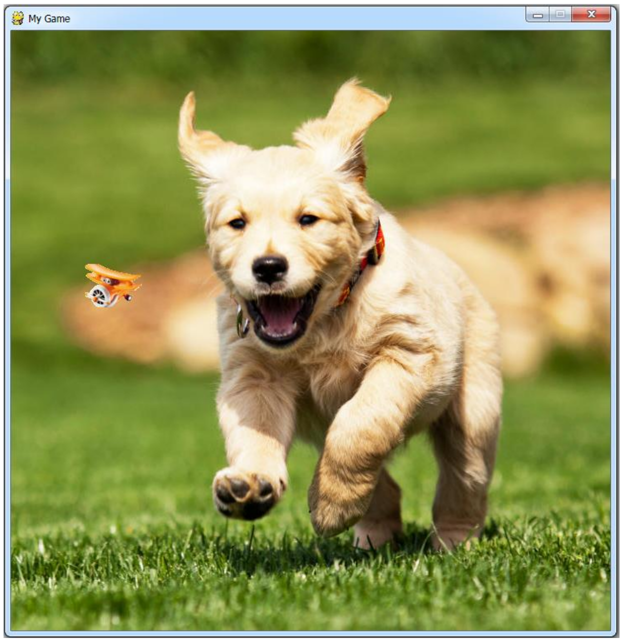
שורה 42 מגדירה לשעון שלנו להמתין פרק זמן לפני האיטרציה הבאה של לולאת ה-while שבשורה 30. כאמור, בלי המתנה זו, לא נוכל לשלוט במהירות ההתקדמות של הגרפיקה שלנו על המסך.

תרגיל – פינג פונג

שכללו את התרגיל בו ציירתם עיגול על המסך, כך שהעיגול יתקדם על המסך מצד לצד. במידה והעיגול נוגע בשולי המסך, שנו את וקטור המהירות שלו כאילו שהוא כדור שניתז ממשטח.


My Game
My Game

ציור Sprite



סיימנו להשתעשע עם צורות גאומטריות. כעת נניח על תמונת הרקע שלנו תמונת שחקן קטנה, שנקראת Sprite, ונזיז אותה ממקום למקום.

בתור תמונת שחקן נבחר קובץ תמונה קטן:

שימו לב שהמטוס נמצא על רקע אחיד. מיד נבין את החשיבות שבכך.

כדי ליצור תמונה עם רקע אחיד 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).

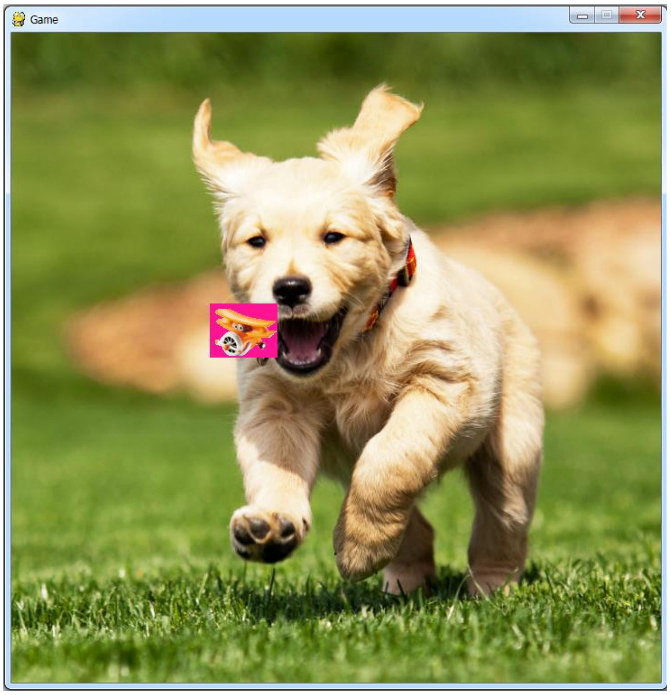
לאחר שסיימנו, מומלץ לשמור את התמונה בפורמט png. זהו פורמט שאינו מעוות את הצבעים בתמונה וכך הצבע שבחרנו בתור רקע יישמר כמו שהוא.

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 תמונת הרקע.

אם נריץ את התוכנית כעת, נקבל את התוצאה הבאה:



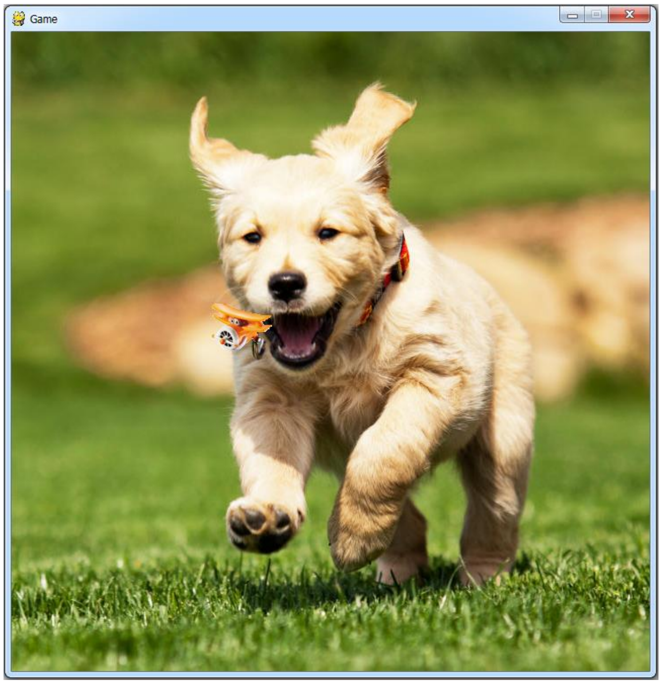
כאן נכנס לשימוש הרקע שהגדרנו ל-Sprite שלנו.

בתחילת התוכנית נגדיר כקבוע את הצבע שאיתו צבענו את רקע ה-Sprite שלנו:

PINK = (255, 20, 147)

נגדיר לתוכנית שלנו להתעלם מהצבע הזה. כלומר, אם היא מזהה ב-Sprite שלנו פיקסל בצבע שהגדרנו, היא תתייחס אליו כשקוף ולא תצבע איתו מעל תמונת 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 תמונת הרקע שלנו ולהזיז אותה.

קבלת קלט מהעכבר

כאשר אנו מקבלים קלט מהעכבר, אנחנו רוצים למעשה לדעת שני דברים:

- מה מיקום העכבר על המסך
- אילו כפתורים לחוצים

לשמחתנו, PyGame מספק לנו את המידע הזה די בקלות.

באמצעות המתודה ()PyGame.mouse.get_pos אנחנו מקבלים את המיקום הנוכחי של העכבר. במידה 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.

```
while not finish:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            finish = True

    screen.blit(img, (0, 0))
    mouse_point = pygame.mouse.get_pos()
    screen.blit(player_image, mouse_point)

    pygame.display.flip()
    clock.tick(REFRESH_RATE)
```

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

PyGame.mouse.set_visible(False)

כעת נרצה לבצע פעולות שונות על פי לחיצות המשתמש על העכבר.

ראשית נגדיר את לחצני העכבר. המספרים המשוייכים לכל לחצן הן בהתאם לערכים של PyGame, אנחנו נותנים להם שמות כדי שהקוד יהיה יותר קריא:

```
LEFT = 1
SCROLL = 2
RIGHT = 3
```

בכל פעם שהמשתמש יקליק על הכפתור השמאלי של העכבר, נשמור את מיקום העכבר ברשימה שאנחנו ניצור – mouse_pos_list. נוכל להשתמש ברשימת המיקומים של העכבר בשביל לעשות כל דבר שנרצה.

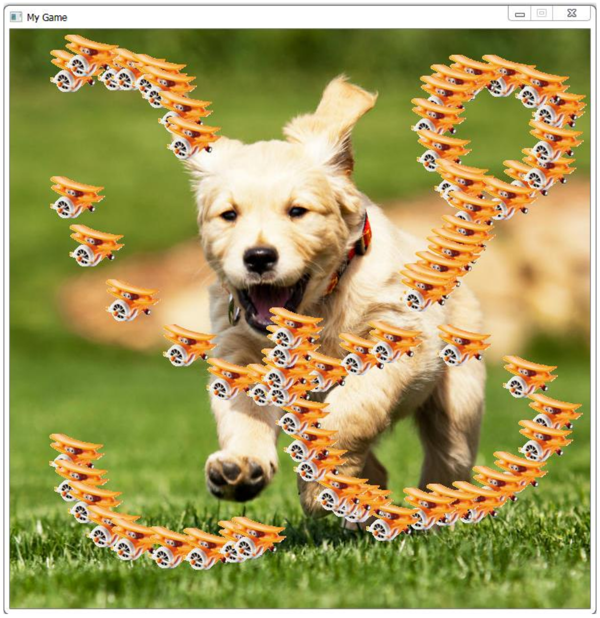
נבחן את הקוד הבא:

```
while not finish:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            finish = True
    elif event.type == pygame.MOUSEBUTTONDOWN \
            and event.button == LEFT:
        mouse_pos_list.append(pygame.mouse.get_pos())
```

התנאי שהגדרנו מתחיל בבדיקת event.type – אנו בודקים האם מדובר בלחיצה על כפתור כלשהו בעכבר. שימו לב שאין צורך להגדיר את הקבוע MOUSEBUTTONDOWN, משום שהקבוע הזה מופיע כבר בתוך ה-event־ים המוגדרים של PyGame. לאחר שווידאנו שהתרחשה לחיצה על כפתור כלשהו בעכבר, אנו בודקים איזה כפתור בדיוק נלחץ.

תרגיל – עכבר

כיתבו תוכנית אשר בכל פעם שהעכבר נלחץ בקליק שמאלי, תשאיר על המסך עותק של ה-Sprite שלנו. וודאו שניתן להקליק על העכבר ללא הגבלה.



קבלת קלט מהמקלדת

קבלת קלט מהמקלדת דומה למדי לקבלת קלט מהעכבר. גם כאן, אנו מחפשים event בתוך הרשימה שחוזרת 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:

בהנחה שאכן הייתה לחיצה על המקלדת, נוכל לבדוק האם מקש ספציפי הוקש. כל המקשים במקלדת מתחילים ב-_K ואחריו יש את המקש. לדוגמה, מקש ה-a הוא K_a. נוכל לבדוק אם מקש ה-a הוקש על ידי:

if event.key == PyGame.K_a:

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

השמעת צלילים

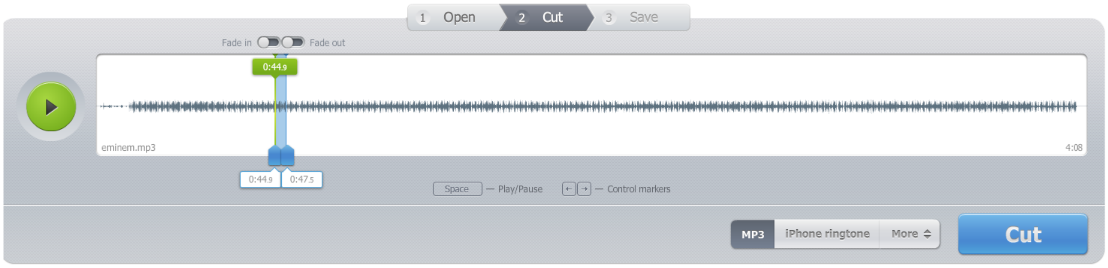
ראשית ניצור קובץ שמכיל את הצליל שאנחנו רוצים להשמיע. רוב הצלילים ניתנים למציאה ב-youtube (אפשר לחפש לדוגמה laser beam sound effect). לאחר מכן נוריד את קובץ ה-mp3 באמצעות אתר כגון:

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

שימו לב שהורדת מוזיקה שיש עליה זכויות יוצרים באמצעות אתר זה הינה בניגוד לתנאי השימוש של יוטיוב.

כדי לחתוך רק קטע מסויים ממתוך קובץ ה-mp3 שהורדנו, נשתמש באתר כגון:

http://mp3cut.net/



כעת יש ברשותנו קובץ mp3 עם הצליל המבוקש.

העלאה והשמעה שלו מתבצעות כך:

```
SOUND_FILE = "kaboom.mp3"
pygame.mixer.init()
pygame.mixer.music.load(SOUND_FILE)
pygame.mixer.music.play()
```

תרגיל סיכום ביניים

צרו סקריפט PyGame אשר מבצע את הדברים הבאים:

- העלאה של תמונת רקע
- הזזה של Sprite כלשהו על תמונת הרקע
- אפשר יהיה להזיז את ה-Sprite על ידי העכבר
- אפשר יהיה להזיז את ה-Sprite על ידי המקלדת
- לחיצה על קליק שמאלי בעכבר תותיר את הסימן של ה-Sprite במיקום שלו על המסך
- לחיצה על מקש ה-space תמחק את כל ה-Sprites שצוירו על הרקע
- לחיצה על המקש הימני של העכבר תשמיע אפקט קולי כלשהו

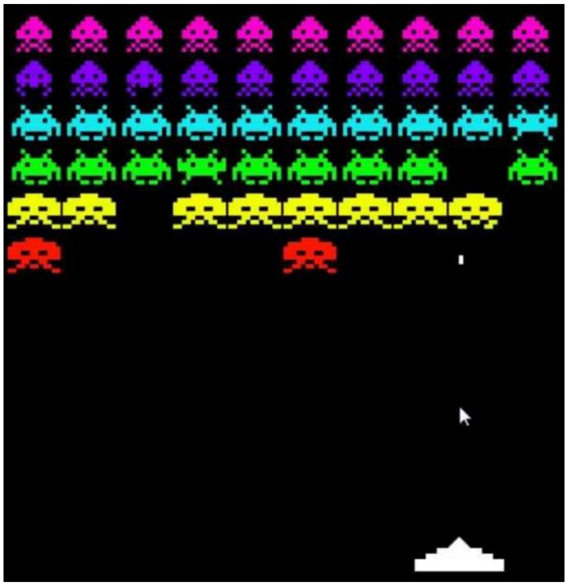
תהנו!

PyGame מתקדם – שילוב OOP

מבוא

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

נכון, אבל הבעיה היא שהשיטה שהשתמשנו בה עד כה טובה בשביל להזיז צורה או שתיים. דמיינו שהמסך שלנו מלא ב-Sprites של Space Invaders שצריך להזיז אותם... נצטרך להגדיר כמה עשרות Sprites, לכל אחד מהם לטעון תמונה, לקבוע מיקום ולנהל את התזוזה שלהם על המסך. אחת הבעיות הגדולות שלנו תהיה לתת שמות למשתנים. נניח שנרצה להגדיר מיקום על המסך עבור שני space invaders. נזדקק למשתנה שישמור את המיקום x של invader1, משתנה למיקום y של invader1, משתנה למיקום x של invader2 ועוד אחד למיקום y של invader2... וזה עוד לפני שהתחלנו לדבר על משתנים למהירויות שלהם (לא תמיד הם נעים באותה מהירות). לא נעים במיוחד לתכנת משחק כזה ☺



בנקודה זו מגיע לעזרתנו OOP. הרעיון הכללי הוא שנגדיר מחלקה שכוללת את כל התכונות של האובייקט שאנחנו צריכים, לדוגמה איך נראה space invader, מה המיקום שלו על המסך ומה המהירות שלו, ובכל פעם שנרצה להוסיף invader חדש – פשוט ניצור instance נוסף של המחלקה שלנו. קדימה, מתחילים.

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.

נגדיר בתוכו class בשם Ball. חשוב מאוד ש-Ball יירש מ-Sprite, כך שנוכל לרשת את כל המתודות המועילות של Sprite, מתודות אשר ישמשו אותנו בהמשך. כמובן, לא נשכח לעשות import PyGame בתחילת הקובץ. כעת 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 שאנחנו צריכים לבצע הוא להריץ את פונקציית האתחול של המחלקה ממנה Ball יורש.

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

בשורה 13 מוגדר משתנה בשם rect ששייך לאובייקט שלנו. משתנה זה שומר את המיקום של הכדור שלנו על המסך, והוא מאוד חשוב בשביל לדעת מה הכדור שלנו עושה. לדוגמה, באילו אובייקטים אחרים הוא מתנגש. שימו לב שלמרות שאנחנו מציירים על המסך עיגול, הרי ש-rect הוא מרובע. זיכרו שבעיני התוכנית התמונה שלנו איננה עיגול, אלא מרובע שביקשנו להציג צבע אחד ממנו בתור "שקוף". לתוכנית אין דרך לדעת שאחרי שהצגנו את הצבע הורוד בתור שקוף, 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 נגדיר שה-Sprite שלנו מקבל מיקום התחלתי בזמן היצירה שלו. נשתמש בו מאוחר יותר לטובת הציור על המסך.

נותר לנו רק לקבוע את מהירות הכדור. שימו לב לכך שהמשתנים של המהירות הם מוסתרים – מתחילים ב-__ . היינו יכולים לקבוע מהירות רנדומלית או לקבל את המהירות כפרמטר, נשאיר זאת להחלטתכם.

הוספת מתודות accessors ו-mutators שימושיות

בואו נזיז את הכדור 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 הכדור בכל פעם שנרצה. מיקום הכדור החדש יחושב בתור המיקום הקודם של הכדור, בתוספת מהירות הכדור בכל אחד מהצירים.

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

הגדרת אובייקטים בתוכנית הראשית

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

```
from shapes import Ball
```

נגדיר שני כדורים ונצייר 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()
```

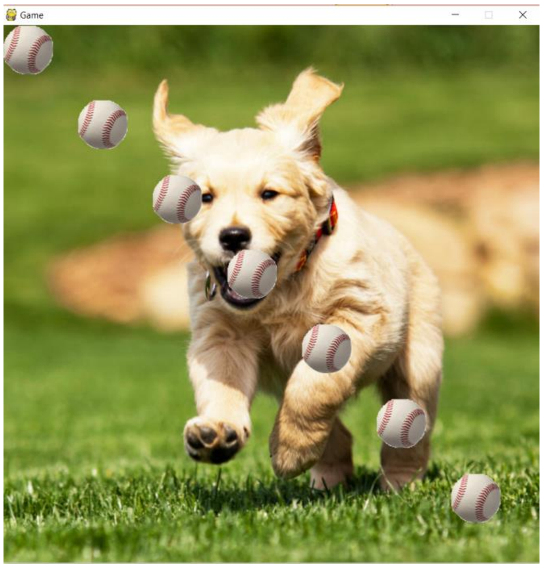
בשורות 26 ו-27 אנחנו יוצרים שני כדורים, לכל אחד מהם יש מיקום התחלתי שאנחנו מעבירים ל-constructor. מיקום התחלתי זה יועתק שם ל-member rect של המחלקה Ball.

בשורות 28, 29 אנחנו מעבירים למסך את הכדורים. המתודה blit מקבלת כזכור את התמונה שאנחנו רוצים להעלות ואת המיקום. התמונה היא ball.image (עבור כל כדור), ואל המיקום אנחנו ניגשים בעזרת המתודה get_pos שיצרנו לשם כך.

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

sprite.Group()

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



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

```
balls_list = pygame.sprite.Group()
for i in xrange(NUMBER_OF_BALLS):
    ball = Ball(i*DISTANCE, i*DISTANCE)
    balls_list.add(ball)

balls_list.draw(screen)
```

כדי לעבור על כל הכדורים בבת אחת אנחנו צריכים לולאת for, ולולאת ה-for צריכה לעבור על משהו שהוא iterable. sprite.Group() הוא iterable. אבל למה אנחנו מגדירים sprite.Group() ולא סתם רשימה ריקה, [ ]? בגלל שלרשימה מסוג sprite.Group() יש כל מיני מתודות שימושיות, שחוסכות לנו עבודה. בקרוב נכיר שתיים 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 מתודת add. נפלא, יש לנו כעת רשימה הכוללת שבעה כדורים. שימו לב לכך שהעובדה שאנחנו משתמשים בכל איטרציה באותו משתנה בשם ball אינה גורמת למחיקת הכדורים הקודמים: כל משתנה בשם ball הוא מצביע על אובייקט מסוג Ball. בכל פעם שהתוכנית מגיעה לשורה 28, נוצר אובייקט חדש מסוג Ball והמשתנה ball מצביע עליו. כלומר, רק המצביע משתנה – ולא האובייקט עצמו. נוסף על כך, בשורה 29 המצביע על האובייקט שנוצר נשמר ברשימה, ולכן אנחנו מסוגלים לגשת אליו גם אחרי ש-ball כבר מצביע על האובייקט הבא.

בשורה 31 אנחנו קוראים למתודה שחוסכת לנו המון עבודה: draw. המתודה הזו פועלת על sprite.Group() ומאפשרת להעביר למסך בבת אחת את כל האובייקטים שיש ברשימה. פעולה זו שקולה ללרוץ 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 לחצן שמאלי בעכבר תיצור כדור חדש על המסך.

ניזכר בקוד שבודק אם היתה הקלקה על הלחצן השמאלי של העכבר – שורה 30 (שממשיכה 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 במיקום זה ומוסיפים אותו לרשימה. את יתר הפקודות כבר הכרנו לפני: הדפסת הרקע (36), הדפסת כל הכדורים (37), עדכון המסך (38) וקציבת זמן עדכון התוכנית (39).

הזזת האובייקטים

אובייקטים זזים יותר מעניינים מאובייקטים שקבועים במקום. בואו ניתן לכל אובייקט מהירות התחלתית אקראית (חשוב לעשות 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 34–36 אנחנו מגרילים מהירות התחלתית ומעדכנים את מהירות הכדור בהתאם. בכל פעם שנריץ את המתודה ball.update_loc() יעודכנו ערכי ה-rect.x וה-rect.y של הכדור, כפי שקבענו בהתחלה:

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

זהו, הכדורים שלנו נעים על המסך ☺

תרגיל bounce

הבעיה במצב הנוכחי היא, שגם כאשר הכדורים מגיעים לקצה המסך הם ממשיכים לנוע ויוצאים ממנו. לכן הוסיפו קוד, שבודק אם הכדור שלנו נוגע בשולי המסך ואם כן – מעדכן את המהירות שלו כך שהכדור "קופץ" חזרה. טיפ: קפיצת הכדור חזרה מתבצעת על ידי היפוך המהירות האופקית שלו (אם פגע בקצה הימני או השמאלי של המסך) או היפוך המהירות האנכית שלו (אם פגע בקצה העליון או התחתון של המסך).

בדיקת התנגשויות

ברוב משחקי המחשב נרצה לדעת אם שני אובייקטים עולים זה על זה. לדוגמה, אם יריה נוגעת בדמות, או אם הפקמן שלנו נתפס על ידי שדון.

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

המתודה spritecollide משמשת למטרה זו. המתודה מקבלת אובייקט ורשימה של אובייקטים ומחזירה את כל האובייקטים מהרשימה שמתנגשים באובייקט הנבדק. בשורת הקוד הבאה אנו בודקים אם אובייקט מסויים בשם ball מתנגש בכדורים ששמורים 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-rectים שלהם חופפים זה עם זה.

בנוסף, המתודה מקבלת פרמטר בוליאני בשם doKill ("האם להרוג"). אם ערך הפרמטר הוא True, אז כל אובייקט ברשימת 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 מגדירים שאנחנו עוברים על כל הכדורים שבתוך רשימת הכדורים שלנו. כלומר אנחנו בודקים בכמה כדורים בתוך הרשימה מתנגש כל כדור ברשימה. ברור שהתשובה תהיה תמיד לפחות 1, מכיוון שהכדור שבחרנו מהרשימה מתנגש עם עצמו (שכן ה-rect שלו חופף את זה של עצמו). זו גם הסיבה לכך שאנחנו לא מוחקים את הכדור המתנגש מהרשימה, כי אחרת נישאר עם רשימת כדורים ריקה.

בשורה 53 אנחנו בודקים האם הכדור שלנו התנגש רק בכדור אחד ברשימה (כלומר, בעצמו בלבד). אם כן, זה אומר שהכדור צריך להשאר על המסך. לכן נשמור אותו ברשימת הכדורים ה"שורדים" new_balls_list. שימו לב לכך שזוהי רשימה מסוג ()sprite.Group ושבשורה 49 אנחנו מרוקנים אותה מכדורים, באמצעות המתודה empty.

כל שנותר לנו לעשות הוא להחזיר את הכדורים השורדים לרשימת balls_list לפני שנמשיך ונציג אותם על המסך. לשם כך אנחנו מרוקנים את balls_list בשורה 57 ולאחר מכן אנחנו מעתיקים כל כדור מתוך רשימת השורדים אל תוך balls_list.

שימו לב לכך שאי אפשר לכתוב פשוט balls_list = new_balls_list. מה יקרה במקרה זה? תוכלו להעזר בדוגמה הבאה של שתי רשימות שמצביעות על אותו המקום 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]: []
```

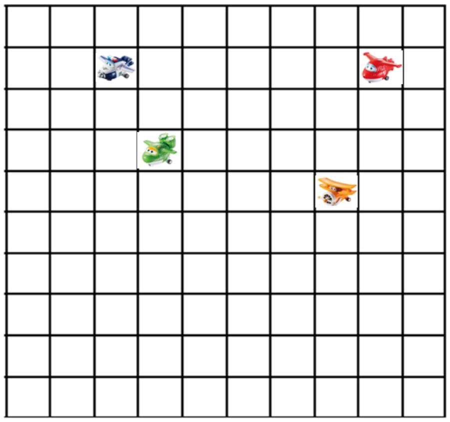
לסיכום, הנה הקוד המלא של התוכנית שבודקת התנגשויות של כדורים:

```python
# Pygame example program
# Sprite collision detection
# Author: Barak Gonen, 2017

import pygame
import random
from shapes import Ball

WIDTH = 720
HEIGHT = 720
LEFT = 1
REFRESH_RATE = 60  # times each second
BALL_SIZE = 55  # pixels

img = pygame.image.load('example.jpg')
pygame.init()
size = (WIDTH, HEIGHT)
screen = pygame.display.set_mode(size)
pygame.display.set_caption("Game")
screen.blit(img, (0, 0))
clock = pygame.time.Clock()

balls_list = pygame.sprite.Group()
new_balls_list = pygame.sprite.Group()
finish = False
while not finish:
    for event in pygame.event.get():
        # quit if window closed
        if event.type == pygame.QUIT:
            finish = True
        # add ball each time user clicks mouse
        elif event.type == pygame.MOUSEBUTTONDOWN \
                and event.button == LEFT:
            x, y = pygame.mouse.get_pos()
            ball = Ball(x, y)
            vx = random.randint(-3, 3)
            vy = random.randint(-3, 3)
            ball.update_v(vx, vy)
            balls_list.add(ball)

    # update balls locations, bounce from edges
    for ball in balls_list:
        ball.update_loc()
        x, y = ball.get_pos()
        vx, vy = ball.get_v()
        if x + BALL_SIZE > WIDTH or x < 0:
```

```
            vx *= -1
        if y + BALL_SIZE > HEIGHT or y < 0:
            vy *= -1
        ball.update_v(vx, vy)

    # find which balls collide and should be removed
    new_balls_list.empty()
    for ball in balls_list:
        balls_hit_list = pygame.sprite.spritecollide \
            (ball, balls_list, False)
        if len(balls_hit_list) == 1:    # ball collides
                                        # only with itself
            new_balls_list.add(ball)

    balls_list.empty()
    for ball in new_balls_list:
        balls_list.add(ball)

    # update screen with surviving balls
    screen.blit(img, (0, 0))
    balls_list.draw(screen)
    pygame.display.flip()
    clock.tick(REFRESH_RATE)

pygame.quit()
```

תרגיל מסכם – פיקוח אווירי

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



למרחב האווירי שלכם נכנסו 4 מטוסים מסוג CrazyPlane שנעים בו אקראית. בכל תור, כל אחד מהמטוסים יכול לזוז לכל משבצת שצמודה למשבצת בה הוא נמצא (כולל באלכסון).

עליכם לכתוב אלגוריתם שמנהל את תנועת המטוסים: בכל תור על האלגוריתם לבחור אם לתת למטוס להמשיך לנוע אקראית או להורות לו לאיזו משבצת צמודה הוא צריך לפנות.

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

חישוב ניקוד לדוגמה:

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

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

סיכום

בפרק זה למדנו לתכנת משחקי מחשב באמצעות PyGame. ראינו איך יוצרים מסך משתמש, מגדירים תמונת רקע, מעלים על המסך תמונה קטנה שזזה, מקבלים קלט מהעכבר ומהמקלדת ומשמיעים צלילים.

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

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

פרק 12 – מילונים

בפרק זה נכיר טיפוס חדש של פייתון – מילון (dictionary). מהו מילון? זהו אוסף של זוגות, כאשר כל זוג מכיל מפתח וערך. מפתח נקרא key וערך נקרא value. לדוגמה, אפשר להגדיר מילון שמכיל ציונים, כאשר המפתח הוא תעודת זהות. מגדירים מילון באמצעות סוגריים מסולסלים, כך:

```
students_grade = {}
```

כעת ניתן להזין לתוכו זוגות של מפתחות וערכים. לדוגמה:

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

בצד שמאל, בירוק, אלו המפתחות. במקרה זה בחרנו שהמפתחות יהיו מחרוזות, אך אפשר לבחור בטיפוסי משתנים נוספים, לדוגמה tuple. בצד ימין, בכחול, אלו הערכים. במקרה זה בחרנו שהערכים יהיו מטיפוס int, אך אפשר להזין ערכים מכל טיפוס שנרצה.

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

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

ידפיס את הערך 95.

אם נרצה להגדיר מילון שמראש יש לו ערכים התחלתיים, נוכל לעשות זאת באמצעות הסימן נקודותיים:

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

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

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

הסבר: כאשר אנחנו מבצעים לולאת for על מילון אנחנו רצים על אוסף כל המפתחות שקיימים במילון. עבור 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 ששייך לתעודת זהות שאינה קיימת במילון:

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

נקבל exception:

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

פקודת get מאפשרת לנו לבצע חיפוש "בטוח". כלומר, אם המפתח קיים יוחזר הערך הנכון, אך אם המפתח אינו קיים יוחזר הערך None. נזכיר, כי הערך None משמעותו "כלום". זו מילה שמורה. 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')
```

שימו לב, ש-get דורשת סוגריים עגולים, כיוון שהיא מתודה (של אובייקט מטיפוס 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 במתודות 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 – בננות, 10 ₪. תפוחים, 8 ₪. לחם, 7 ₪. גבינה, 20 ₪. מיץ, 15 ₪.

ב. צרו מילון בשם shopping_cart. המפתח הוא שם המוצר והערך הוא כמות המוצרים מסוג זה בעגלת הקניות. הוסיפו מספר מוצרים למילון. לדוגמה – בננות, 2 יחידות. לחם, 3 יחידות. גבינה, 1 יחידה.

ג. הכניסו למשתנה בשם total את סכום הקניות בעגלה. בידקו שהסכום שמתקבל הוא נכון.

ד. כעת, הזינו לתוך shopping_cart מוצר שאינו קיים ברשימת המחירים. עליכם לוודא שהתוכנית אינה עפה על שגיאה אלא מדפיסה הודעה מתאימה.

תרגיל מסכם מילונים – wordcount (קרדיט: google classes)

הורידו את קובץ התרגיל מהלינק הבא:

http://data.cyber.org.il/python/wordcount.zip

בתרגיל אתם נדרשים לקרוא את הקובץ alice.txt ולהדפיס למסך את כמות המופעים השונים של כל מילה.

טיפ: השתמשו ב-split על מנת להפריד בין מילים.

למתקדמים: טפלו במקרים של מילים זהות, שנבדלות רק בסימני פיסוק שצמודים אליהן. לדוגמה – ‘her’ ו-‘her.’ (עם נקודה בסוף). בדוגמה זו עליכם לזהות שמדובר באותה מילה ולספור יחד את כל המופעים.

מילונים, מתחת למכסה המנוע (הרחבה)

בחלק זה נבין יותר לעומק איך עובד מילון. זה אינו חלק מעשי, אבל תלמידים סקרנים ימצאו בו עניין.

דמיינו שאתם עובדים בבית מלון בשם 'Hash Gardens'. זהו מלון מאד מיוחד מכיוון שבעל המלון הוא מתמטיקאי מטורף שלא מוכן לשמור שום מידע על האורחים שלו, לא על מחשב ולא על פיסת נייר. לכן במלון אין רשימה של מספרי החדרים של כל האורחים. כלומר, אין לכם דרך לדעת באיזה מספר חדר מתגורר כל אורח. בוקר אחד הטלפון בדלפק מצלצל. מישהו מבקש לשוחח עם מר john smith. איך אתם מעבירים את השיחה לחדר שלו?

ככל הנראה, תצטרכו לחפש את מר סמית' בכל החדרים. לא קל, במיוחד אם המלון שלכם גדול מאוד. תבזבזו הרבה זמן באיתור מספר החדר של כל אורח שמתקשרים אליו.

אתם פונים לבעל המלון ומתחננים שייתן לכם לשמור במקום כלשהו את מספרי החדרים של האורחים, אבל הוא פשוט מחייך ונותן לכם עצה מתימטית. מעכשיו, כל אורח שמגיע למלון מקבל מכם מספר חדר שיש קשר מתימטי בינו לבין שם האורח. כך, למרות שלא רשום לכם בשום מקום איזה חדר קיבל כל אורח, כאשר מתקשרים ומבקשים לשוחח עם אורח כלשהו, אתם יכולים לחשב במהירות את מספר החדר המבוקש.

להלן דוגמה לפונקציה שמחשבת מספר חדר על פי שם האורח: נניח שבמלון יש 113 חדרים, אשר ממוספרים מ-0 ועד 112 (כפי שניתן לצפות ממתימטיקאי, כמות החדרים במלון היא מספר ראשוני). כל אורח שמגיע למלון מקבל חדר שמספרו מחושב לפי הנוסחה הבאה: נכפול את ערכי ה-ascii של כל התווים בשם האורח ולתוצאה נבצע מודולו 113. באופן זה נהפוך כל מחרוזת למספר בין 0 ל-112. התוצאה תהיה מספר החדר.

להלן קוד של פונקציה שמבצעת את החישוב הנ"ל:

```
ROOMS = 113


def find_room_number(guest_name):
    tmp = ord(guest_name[0])
    for letter in guest_name[1:]:
        tmp *= ord(letter)
    return tmp % ROOMS


def main():
    room = find_room_number('john smith')
    print room
```

כאשר מגיע אלינו האורח john smith, הפונקציה תחשב את מספר החדר שלו – 61. אם מישהו יתקשר למר סמית', נוכל לחשב מחדש את מספר החדר שלו.

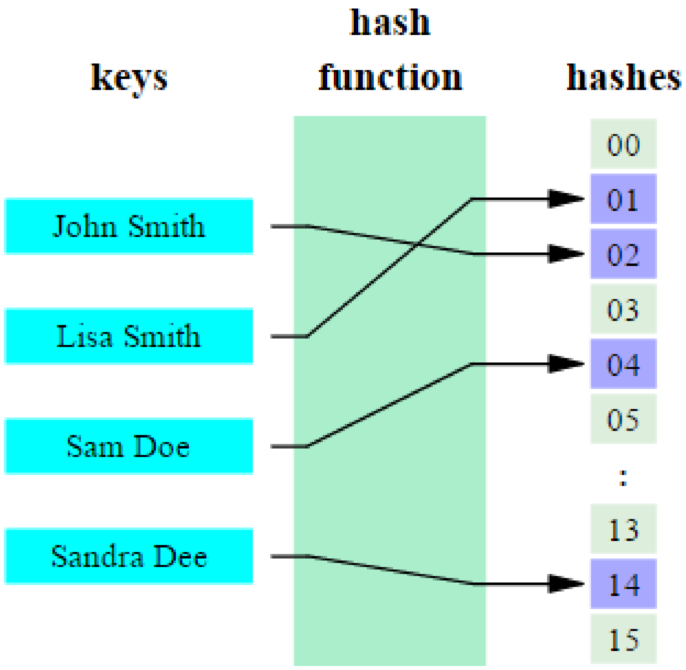
כפי שאולי ניחשתם, הפונקציה find_room_number היא בעצם פונקציית hash. מה מיוחד בה?

א. היא מתאימה לכל מחרוזת מספר בתחום קבוע מראש

ב. למחרוזות זהות מתקבל hash זהה. לא יכול להיות שאורחים עם שמות זהים יקבלו חדרים שונים

ג. אי אפשר (או לפחות, קשה מאוד) לשחזר את שם האורח מתוך ה-hash של השם שלו. אם אנחנו יודעים שאורח נמצא בחדר 61, אי אפשר לגלות ששמו הוא john smith. במילים אחרות, זוהי 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 שייצור התנגשות (collision). לכן, פונקציית hash טובה היא כזו שהסיכוי להתנגשות בה הוא מאוד קטן.



המחשה של פונקציית hash. מקור: ויקיפדיה

נחזור כעת מעולם המלונאות והתיירות אל עולם הפייתון. בפני מי שבנה את שפת פייתון עמדה אותה בעיה של העובד במלון: איך מוצאים מהר את הערך שצמוד למפתח מסויים? כפי שראינו, חיפוש על פני כל המפתחות הוא יקר בזמן, במיוחד אם יש במילון שלנו הרבה מפתחות. הפתרון שמילונים עושים בו שימוש הוא כמובן פונקציית hash.

מילון בפייתון מתחיל בתור טבלה ריקה. כאשר אנחנו מכניסים למילון מפתח וערך, פייתון מחשב את ה-hash של המפתח ומקבל מספר. המספר הזה יהיה האינדקס בטבלה שבו פייתון ישמור את המפתח ואת הערך הצמוד אליו. לדוגמה, אם ה-hash יהיה 5, האינדקס בטבלה יהיה 5.

מה קורה כאשר אנחנו רוצים לשלוף ערך מהמילון? אנחנו מעבירים לפייתון את המפתח. על בסיס המפתח פייתון מחשב את ה-hash וניגש אל המקום בטבלה שזה המספר שלו. באופן זה, חיפוש של ערך בתוך מילון לוקח זמן קצר מאד וכמעט אינו תלוי בכמות המפתחות שיש במילון.

עלול לקרות מצב שבו לשני מפתחות יש את אותו hash. ראוי לציין שפייתון משתמש בפונקציית hash "טובה", כזו שדואגת שהסיכוי ששני מפתחות יהיו בעלי אותו hash הוא קטן. למצב שלשני מפתחות יש אותו hash קוראים collision ויש דרך לטפל בו. הערך הראשון מוכנס לטבלה כרגיל. כאשר פייתון רוצה להכניס את הערך השני לאותו מקום בטבלה, הוא בודק ומגלה שבטבלה כבר יש מפתח במקום המבוקש. אם המפתח החדש שונה מהמפתח שקיים בטבלה, סימן שקרה collision. המפתח החדש יישמר במקום הפנוי הבא. לדוגמה, אם המקום באינדקס 5 בטבלה כבר תפוס, פייתון יכול לנסות לשמור באינדקס 6. אם נרצה לשלוף מהמילון את הערך של המפתח השני, פייתון ייגש קודם כל לאינדקס 5. שם הוא יגלה שהמפתח אינו מתאים ולכן הוא ימשיך לאינדקס הבא עד שיימצא את המפתח המבוקש.

סוגי מפתחות

ראינו שכל צמד שמוכנס למילון מורכב ממפתח וערך. בתור מפתח השתמשנו במשתנים מטיפוס מחרוזת. האם כל טיפוס משתנה יכול להיות מפתח?

כדי להשיב על זה, ניזכר בדיון שלנו בנושא mutable ו-immutable. אובייקט מסוג mutable הוא אובייקט שניתן לשנות, כדוגמת רשימה. כשלמדנו על רשימות ראינו שאנחנו יכולים להגדיר רשימה, לשנות את אחד האיברים בה ולאחר השינוי ה-id של הרשימה יישאר ללא שינוי. הערך של אחד האיברים השתנה אך זו עדיין אותה רשימה. לעומת זאת, אובייקטים מסוג immutable לא ניתנים לשינוי. כלומר, הדרך היחידה לשנות אותם היא להגדיר אותם מחדש ואז כמובן משתנה ה-id שלהם.

הבה נניח שהיינו יכולים להשתמש במשתנה מסוג mutable בתור מפתח. לצורך הדיון, ניקח שתי רשימות, שנזכיר שהינן mutable:

```
a = ['apple']

b = ['banana']
```

שימו לב שאלו רשימות, אשר מכילות כל אחת איבר אחד מטיפוס מחרוזת.

כעת, נגדיר מילון:

```
fruits = {a:1, b:2}
```

פייתון חישב את ה-hash של a ושל b והכניס כל אחד מהם למיקום המתאים לו בטבלה.

כעת אנחנו משנים את ערכו של b:

```
b[0] = 'apple'
```

אנחנו יכולים לעשות זאת כיוון שמדובר ברשימה, שהיא mutable.

מה יקרה אם כעת ננסה לשלוף את b מהמילון?

פייתון יחשב את ה-hash של b ויקבל ערך מסויים, אך הערך לא יתאים למיקום ש-b שמור בו. אם לא די בכך, ה-hash שיתקבל דווקא כן יתאים למיקום אחר בטבלה – המיקום אשר שומר את a ושיש לו מפתח זהה – ולכן החיפוש יחזיר את הערך השגוי "1".

סיכום

בפרק זה למדנו אודות מבנה נתונים מיוחד. מילון מאפשר לנו לשמור נתונים ולמצוא אותם במהירות רבה, תוך שימוש במפתחות ובפונקציה מתימטית – hash. למדנו ליצור מילון, להזין לתוכו ערכים ומפתחות ולחפש ערכים במילון באמצעות מפתח. בהמשך סקרנו מספר תכונות של פונקציית hash וראינו כיצד תכונות אלו עוזרות לה להיות שימושית עבור מילונים.

פרק 13 – Magic Functions, List Comprehensions

בפרק זה נלמד לנצל יכולות של פייתון על מנת לכתוב קוד בצורה יותר קצרה ו"פייתונית".

List Comprehensions

נניח שאנחנו רוצים ליצור רשימה של איברים שיש להם חוקיות מסויימת. לדוגמה, חזקות של מספרים. רשימה כזו אי אפשר לייצר בעזרת range, מכיוון שהקפיצה בין מספרים אינה קבועה. מצד שני, אנחנו לא רוצים לכתוב באופן ידני את המספרים ואנחנו כן רוצים לנצל את העובדה שישנה חוקיות למספרים.

דרך אחת ליצור את הרשימה שלנו היא באמצעות לולאת for, לדוגמה, כך:

```
squares = []
for i in range(100):
    squares.append(i**2)
```

כך יצרנו רשימה שמכילה את האיברים [0, 2, 4, … 99**2]

כעת נלמד syntax אלטרנטיבי, שיוצר את אותה הרשימה לעיל בשורה אחת בלבד. צורת כתיבה זו נהוגה מאוד בשפת פייתון, שכן לאחר שמתרגלים אליה היא נוחה מאוד לקריאה ולכתיבה. ל-syntax זה קוראים list comprehensions.

באמצעות list comprehensions אפשר ליצור את הרשימה לעיל בצורה הבאה:

```
squares = [i**2 for i in range(100)]
```

שימו לב לשינוי בסדר הפקודות. בעוד את לולאת ה-for קוראים "עבור כל אחד מהאיברים ב-range(100) תבצע העלאה בריבוע", את הביטוי שבסוגריים 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)
```

והכתיב המקוצר, באמצעות list comprehensions:

```
squares = [i**2 for i in range(100) if i%2==0]
```

לא תמיד קל לקרוא קוד שכתוב בשורה אחת ארוכה. למען קלות הקריאה, אפשר לחלק את השורה הנ"ל למספר שורות:

```
squares = [i**2
           for i in range(100)
           if i%2==0]
```

בינתיים כתבנו לולאות פשוטות, שניתן היה לכתוב ב-3-4 שורות ללא list comprehensions. לעתים, 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 ברשימת הלא ראשוניים – ייחשב מספר ראשוני. להלן קוד שמבצע זאת באמצעות לולאות for:

```
import math
LIMIT = 100

root = int(math.sqrt(LIMIT))
non_primes = []
for i in range(2, root):
    for j in range(2*i, LIMIT, i):
        non_primes.append(j)
non_primes.sort()

primes = []
for i in range(LIMIT):
    if not i in non_primes:
        primes.append(i)
```

הסבר: הקבוע LIMIT מגדיר מה הגבול העליון של המספרים הראשוניים שאנחנו 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 2*5 ומתקדמים בקפיצות של 5. כל מכפלה כזו מתווספת לרשימה בשורה 8. מיון הרשימה בשורה 9 מתבצע רק לשם היופי.

בשורות 14–11 אנחנו יוצרים רשימה חדשה, לתוכה מוכנסים כל המספרים עד LIMIT אשר אינם מופיעים ברשימה של הלא ראשוניים.

וכעת, אותו קוד כפי שהוא כתוב עם list comprehensions:

```python
import math
LIMIT = 100

root = int(math.sqrt(LIMIT))
non_primes = [j
              for i in range(2, root)
              for j in range(2*i, LIMIT, i)]

primes = [i
          for i in range(LIMIT)
          if not i in non_primes]
```

תרגיל – קיצוצים (קרדיט: עומר רוזנבוים, שי סדובסקי)

כיתבו את הפונקציה avg_diff, אשר מקבלת שתי רשימות של מספרים ומחזירה את ההפרש הממוצע ביניהן. לדוגמה, עבור הרשימות [1,1,1,1],[1,2,3,4] יוחזר 1.5. נשמע פשוט? אז זהו, שעקב קיצוצים בתקציב אנחנו נאלצים להתייעל ולכתוב את הפונקציה בשורה אחת בלבד. זה אומר ש-:

- אסור לרדת שורה
- אסור להשתמש בתו ":"

אפשר להניח ששתי הרשימות מכילות מספרים, שהן בעלות אותו אורך ושהן אינן ריקות.

תרגיל – אנטיבי

כיתבו את הפונקציה anti_bi אשר מקבלת מחרוזת ומחזירה אותה ללא מופעים של התו 'b'. זיכרו – הקיצוצים עדיין נמשכים! ☺

לסיכום, list comprehensions מהוות דרך חזקה לכתוב בצורה קצרה מאוד קוד מורכב. הן יכולות להכיל גם תנאים מורכבים, וכאשר מתרגלים אליהן – list comprehensions עוזרות לנו לכתוב קוד בצורה קריאה, יפה ומהירה.

Lambda

פונקציות lambda הן פונקציות לשימוש חד פעמי – אנחנו רוצים לעשות פעולה לא מאוד מורכבת ונראה לנו מיותר להגדיר פונקציה במיוחד בשביל זה.

שימו לב לאופן הכתיבה הבא:

```
f = lambda x: 2*x +1
```

איך קוראים את זה? "הפונקציה f מקבלת כפרמטר x ומחזירה את 2*x+1". כלומר הקוד זהה להגדרת הפונקציה הבאה:

```
def f(x):
    return 2*x + 1
```

אם נרצה לקרוא ל-f נעשה זאת בדיוק כמו שקוראים לפונקציה רגילה. לדוגמה f(5) יחזיר 11.

אפשר גם להגדיר פונקציית lambda שמקבלת כמה פרמטרים. לדוגמה:

```
f = lambda x, y: x*y + x + y
```

לדוגמה, f(2, 3) יחזיר 11.

מתי נשתמש בפונקציית lambda? לדוגמה, כאשר אנחנו רוצים להעביר key לפונקציית sort, אפשר להגדיר את ה-key בתור פונקציית lambda. דוגמה נוספת אשר נלמד בעתיד – כאשר נלמד רשתות ונכתוב קוד לסינון מידע שעובר ברשת, נשתמש בפונקציות lambda.

תרגילים

- כיתבו פונקציית lambda שמקבלת מספר ומחזירה את המספר פלוס 2.
- כיתבו 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].

הפונקציות הבאות שנלמד, map, reduce ו-filter, הוחלפו למעשה על ידי list comprehenstions אך חשוב להכיר אותן כיוון שהן נמצאות לעיתים בקוד פייתון.

Map

פונקציית map מקבלת פונקציה ורשימה. נוצרת רשימה חדשה, שהיא התוצאה של הרצת 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 שבטח ניחשתם, את func אין צורך להגדיר ממש, אלא ניתן להכניס lambda כרצוננו.

לדוגמה, ניקח רשימת מספרים ונרצה לכפול כל מספר פי 2 ולהוסיף 1:

```
old_list = [3, 6, 1, 7, 5]
print 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 מפרמטר אחד ולהעביר ל-map כמות מתאימה של רשימות. 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 (קרדיט: עומר רוזנבוים, שי סדובסקי)

כיתבו פונקציה שמקבלת מחרוזת ומחזירה מחרוזת חדשה, אשר כל האותיות בה מוכפלות. לדוגמה, עבור המחרוזת 'Cyber' יתקבל 'CCyybbeerr'. האתגר הוא כמובן לכתוב 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 ומחזירה רשימה רק של האיברים שהחזירו True. כך, לכתוב –
new_list = filter(func, old_list) מבצע פעולה זהה לקוד הבא:

```
new_list = []
for element in old_list:
    if func(element) is True:
        new_list.append(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, שמחזירה את הערך הבוליאני של הביטוי x%2==0.לאחר מכן אנחנו מכניסים לתוך filter את הפונקציה הנ"ל יחד עם רשימה.

תרגיל – את מסננת אותי (קרדיט: 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]. כמקודם, אורך הפונקציה חייב להיות שורה אחת בלבד ☺

Reduce

הפונקציה reduce מקבלת פונקציה ורשימה ומחזירה ערך יחיד. הערך הזה הוא התוצאה של ביצוע הפונקציה על איברי הרשימה שוב ושוב עד שנותר ערך יחיד. אפשר לחשוב על reduce בתור map שממשיך להתבצע כל עוד אורך הרשימה החדשה שנוצרת גדול מ-1.

לדוגמה, יש לנו רשימה ואנחנו רוצים לחשב את הסכום של האיברים שלה:

```
old_list = [2, 3, 4, 7, 8, 10]
print reduce(lambda x, y: x+y, old_list)
```

אפשר לדמיין שבכל שלב נוצרת רשימה חדשה, שבה האיבר הראשון הוא סכום שני האיברים ברשימה המקורית:

- [5, 4, 7, 8, 10]
- [9, 7, 8, 10]
- [16, 8, 10]
- [24, 10]
- [34]

כאשר הגענו לרשימה באורך 1 החישוב ייעצר ויוחזר הערך שנותר בה.

תרגיל – שיעור ספרות (קרדיט: עומר רוזנבוים, שי סדובסקי)

עליכם ליצור פונקציה שמקבלת מספר ומחזירה את סכום הספרות שלו. לדוגמה, עבור 104 תתקבל התוצאה 5. כרגיל, רק שורה אחת של קוד ☺

סיכום

בפרק זה למדנו כיצד לקחת קוד, שבדרך כלל היינו כותבים אותו באמצעות לולאת for, ולכתוב אותו בצורה קצרה ופשוטה באמצעות list comprehensions או באמצעות פונקציות הקסם של פייתון: lambda, map, filter ו-reduce. חלק מפונקציות אלו יהיו שימושיות בלימודי רשתות. המשך לימוד מוצלח!