לשחק ולתכנת בשפת Python

כרך ראשון: מה זה תכנות?

עומר רוזנבוים ■ שי סדובסקי

מהדורת ניסוי

גבהים
חינוך • מצוינות • סייבר

לשחק ולתכנת בשפת Python

כרך א' – מה זה תכנות?

גרסה 1.1

כתיבה ועריכה

עומר רוזנבוים

שי סדובסקי

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 ממטה הסייבר הצה"לי.

© כל הזכויות על החומרים ממקורות חיצוניים ששובצו בספר זה שמורות לבעליהן.

© תשע"ה – 2015. כל הזכויות שמורות למטה הסייבר הצה"לי ולקרן רש"י.

הודפס בישראל.

http://www.cyber.org.il

תוכן עניינים

תוכן עניינים 2
מבוא 3
פרק 1 - תחילת מסע – תכנות עם רובוטים 7
פרק 2 – תנאים, לולאות ועולמות 27
פרק 3 – רובוטים מורכבים 51
פרק 4 – להתראות Guido 57
פרק 5 – מתחילים עם פייתון 67
פרק 6 – סביבת עבודה PyCharm 89
פרק 7 רשימות, לולאות וטיפוסים 91
פרק 8 – קונבנציות 99
פרק 9 – מחרוזות 100

מבוא

וואו, אתם ניגשים ללמוד תכנות. איזה כיף! לתכנת זו ככל הנראה אחת היכולות הכי חשובות שאתם עומדים לרכוש לאורך החיים, ולא רק אם אתם חנוני מחשבים. באמצעות תכנות תוכלו לעשות דברים רבים העולים בדמיונכם, וגם דברים שעדיין לא. בספר זה תרכשו את הכלים הבסיסיים שישמשו אתכם בכל דרככם בעולם התכנות. באמצעותם, תוכלו בעתיד לכתוב משחקים, לפתח אתרי אינטרנט, אפליקציות לטלפון הסלולרי, תוכנות שיעזרו לכם לפתור את שיעורי הבית במקצועות אחרים, ועוד המון דברים.

למעשה, בקרוב מאוד תגלו שאתם יכולים לתכנת בעצמכם משחקים שהמצאתם. בחוברת זו נתחיל לענות על השאלה הגדולה – "מה זה תכנות?". בהמשך השנה, ובחוברות הבאות, נכיר לעומק את שפת התכנות Python, שתלווה אותנו במסעינו בעולם התכנותי. עד סוף שנת הלימודים, אתם צפויים לרכוש את כל הכלים הדרושים בכדי לתכנן ולפתח משחק מחשב, לגמרי בעצמכם! אנו מקווים שאתם מוכנים למסע המאתגר והמרגש שעומד בפניכם.

קהל היעד של הספר

הספר מיועד לשני קהלי יעד עיקריים:

1. תלמידים ומורים הלומדים במסגרת חלופת הגנת סייבר במגמת הנדסת תוכנה.
2. כל מי שמעוניין ללמוד לתכנת בכלל, ובשפת פייתון בפרט, באופן עצמאי.

שיטת הלימוד של הספר

ספר זה ככל הנראה שונה מספרי לימוד אחרים שהכרתם. הספר נועד לאפשר לימוד עצמאי, והוא **פרקטי** מאוד וכולל תרגול רב. תכנות הינו נושא עצום ומורכב. מטרת ספר זה היא להקנות את הבסיס, כך שתוכלו להיות מתכנתים מצוויינים בהמשך.
לאחר למידה משמעותית של החומר, בעוד מעט מאוד זמן, תוכלו לכתוב בעצמכם תוכנות מעניינות – כגון משחקי מחשב עם גרפיקה, תוכנות שיעזרו לכם בפתרון שיעורי בית ועוד. עם זאת, מטרת הספר היא לא להפוך אתכם למומחים בשפת פייתון, וגם לא למתכנתי על. הספר מתמקד בהקניית הרגלים נכונים ואת הבסיס החשיבתי לעולם התכנות.

במהלך הלימוד בספר, עליכם לתפקד כ**לומדים פעילים** - כלומר, לא רק לקרוא ולהבין את החומר, אלא גם לתרגל אותו. בספר משולבים תרגילים רבים, חלקם מודרכים וחלקם לביצוע עצמי. בכדי לרכוש שליטה בחומר הלימוד, יש לבצע את כל התרגילים ולוודא שאתם מבינים אותם. התרגילים המודרכים נבנו כך שהם מפורקים למשימות מובנות המתבססות זו על זו ומניחות שהלומד מבצע בפועל את ההנחיות. אל תסתפקו בקריאת התרגילים המודרכים, והקפידו לבצע אותם שלב אחרי שלב.

צעדים להמשך

כחלק משיטת הלימוד של הספר, המעודדת למידה עצמאית, בסוף חלק מהפרקים הוספנו סעיף "צעדים להמשך". סעיף זה נועד לתלמידים סקרנים המעוניינים להרחיב את הידע, וכולל מקורות מידע נוספים ותרגילים מתקדמים.

סרטונים

לאורך הספר שובצו מספר סרטוני הדרכה שנועדו להקל על הלמידה. אתם מוזמנים להיעזר בהם לאורך הקריאה. בכל פעם שיהיה קישור לסרטון, יופיע גם קוד QR שיאפשר להגיע אליו בקלות.

ידע מקדים נדרש

ספר זה אינו מניח ידע מוקדם כלל.

משחקים

לאורך חוברת זו, ולמעשה לאורך שנת הלימודים כולה, נכתוב משחקים. כן, משחקי מחשב. בהתחלה, אלו יהיו משחקים פשוטים כגון משחק טריוויה בו נציג למשתמשת שאלות ונאפשר לה לענות עליהן ולדעת מתי היא צדקה ומתי טעתה. לאורך תכנית הלימודים, המשחקים יתפתחו, יהפכו לצבעוניים, יפים ומגוונים. במהלך השנה נכתוב משחקים של ניחוש מספר, משחקים דמויי Quest בהם יש דמויות ועלילה, משחק Hangman ועוד בסופו של דבר, תחשבו ותיצרו משחק עליו חשבתם בעצמכם.

התקנות נדרשות

עבור חוברת זו מומלץ להתקין את מערכת ההתקנות של "גבהים", המכילה את כל התוכנות הנדרשות. מערכת ההתקנות נמצאת בכתובת: http://tinyurl.com/portableheights. כשתכנסו לקישור זה, יופיע מדריך התקנה.

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.

נשמח לשמוע מכם!

ספר זה מיועד להיות בתהליך שיפור מתמיד. אנו נשמח לשמוע את חוות דעתכם על התוכן ו/או על צורת הכתיבה, שאלות שעלו לכם או כל הערה או הארה שתסייע לשפר את הספר.

אתם מוזמנים לשלוח התייחסויות לתיבת המייל pythongvahim@gmail.com.

תודות

אנשים רבים סייעו ותרמו לכתיבת חוברת לימוד זו. ליאת כרמון-קולט ועידן פדלברג תרמו לנו רבות מנסיונם בכיתות ומהידע שצברו, ונתנו לנו משוב רב על תכני הלימוד. ברק גונן היה ועודנו שותף פעיל ותורם בתכנון תכנית הלימודים. כרמל פונדק מינץ סייעה רבות במתן משוב על חוברת זו. לכל המורים, התלמידים והחברים שהשפיעו וסייעו בתהליך יצירת הספר – תודה רבה.

בפרקים העוסקים ב-Guido, הסתמכנו ונעזרנו רבות בשיעורים שכתב Roger Frank ועובדו בידי Jeffrey Elkner. אנו מודים להם ולכלל צוות הכותבים והמפתחים של Guido van Robot.

עומר רוזנבוים

שי סדובסקי

פרק 1 - תחילת מסע – תכנות עם רובוטים

בפרק המבוא דיברנו על הסיבות שבגינן נרצה ללמוד לתכנת. השתכנענו כי אין דבר מעניין וחשוב יותר מללמוד לתכנת, ועכשיו אנו מוכנים להתחיל במשימה.
אך בטרם נכיר את שפת Python, נתחיל את מסענו בתכנות לרובוטית בשם Guido.

מדוע להתחיל עם רובוטית?

קודם כל, זה כיף! Guido היא רובוטית חכמה מאוד, ונוכל ללמד אותה לבצע הרבה פעולות, ובכך ללמוד את העקרונות הבסיסיים של תכנות באופן ויזואלי ומהנה.
שפות תכנות כמו Python, Java, C ואחרות, דורשות לא מעט ידע והבנה בכדי להתחיל להשתמש בהן. כל דבר שנכתוב חייב להיות מדוייק, ושגיאה קטנה ביותר עלולה למנוע מאיתנו להריץ את התכנית. לעומת זאת, כאשר נעבוד עם Guido, נעבוד בסביבה "מוגנת" ונוחה, שמאפשרת לנו ללמוד בקלות מהשגיאות שלנו. למעשה, התוכנה שבה נשתמש, Guido van Robot, נכתבה בדיוק למטרה זו – בכדי ללמוד לתכנת.

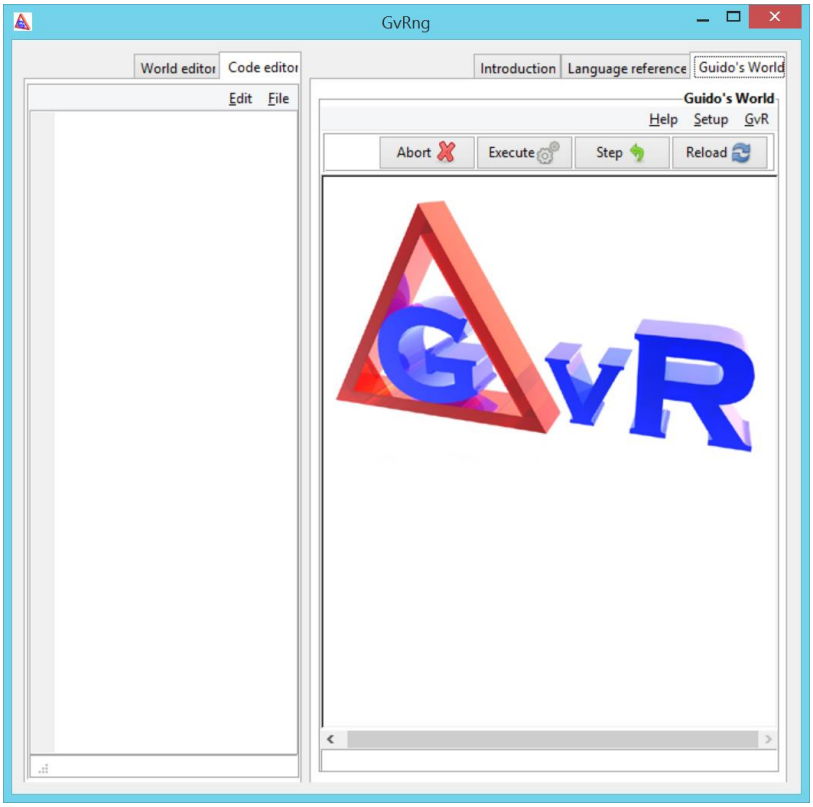
בפרק זה נכתוב פקודות ל-Guido, רובוטית שחיה בעולם קטן ופשוט. כך נוכל לראות את האפקט של הקוד שנכתוב – באופן ויזואלי. בתחילה, נלמד כיצד לגרום ל-Guido לנוע במרחב. בהמשך, נוכל גם לבצע פעולות מורכבות יותר, באמצעות תנאים ולולאות. נלמד גם כיצד לבנות עבור Guido עולמות שונים ולשחק בהם. בסוף פרק זה, נהיה מוכנים לפגוש בשפת תכנות של "העולם האמיתי" כשאנו מצויידים בידע רב על העקרונות החשובים מאחוריה.

מתחילים - התקנה

ראשית, עלינו להתקין את התוכנה Guido van Robot (בקיצור – GvR). נוכל להתקין אותה מאתר הפרוייקט, כאן:

http://sourceforge.net/projects/gvr/files/latest/download?source=files

לאחר שנתקין את התוכנה, אנו צפויים לראות מסך דומה לזה:



בשלב זה, אין צורך להבין מה אומר כל חלק בתוכנה. נכיר זאת עם הזמן.

תרגיל מודרך 1.1 – בניית עולם בסיסי

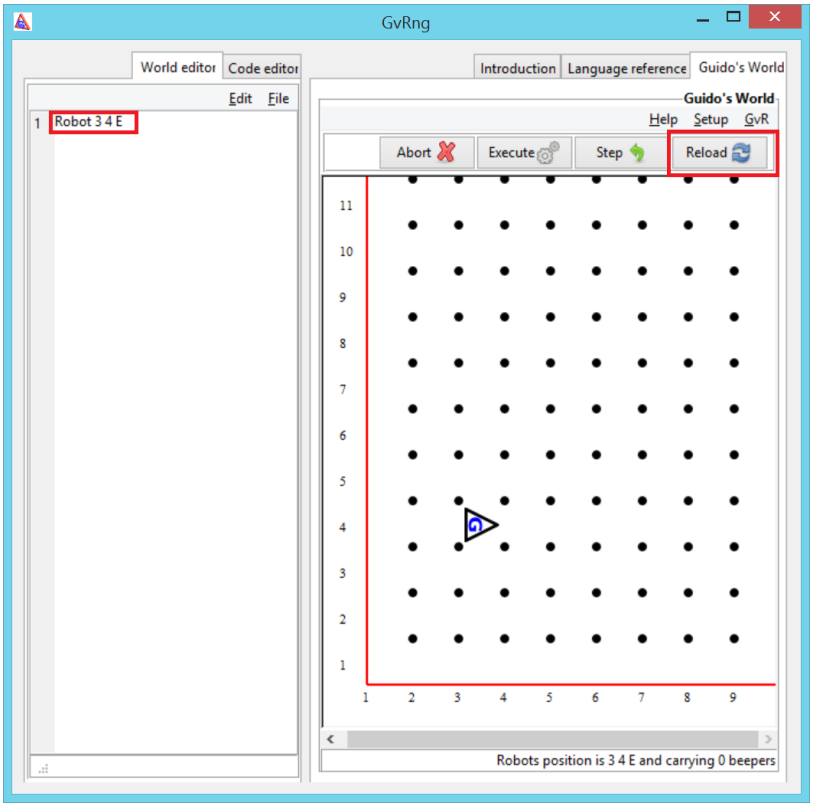
הגיע הזמן להביא את Guido אל המסך, ואם תרצו – להקים אותה לתחיה[1]! לשם כך, נלחץ על הלשונית (tab) בשם **World editor**. בחלון זה, נרשום את הפקודה הבאה[2]:

```
Robot 3 4 E
```

בקרוב מאוד נבין את המשמעות של פקודה זו. בינתיים, נלחץ על הלשונית בשם **Guido's World**, ואז על כפתור ה-**Reload**. בהנחה שביצענו את הפעולות כמו שצריך, אנו צפויים לראות מסך כזה:

[1] האם חשבתם שכבר בדקות הראשונות בהן תלמדו לתכנת תיצרו עולמות ותקימו רובוטים לתחיה? אנו מנחשים שלא. אתם יכולים לחייך מעט בטרם תמשיכו בקריאה.

[2] תזכורת: עלינו להיות לומדים פעילים, ולכן נרשום בעצמנו את הפקודות ונבצע את הצעדים המוסברים בתרגילים מודרכים, ולא נסתפק בלקרוא אותם.



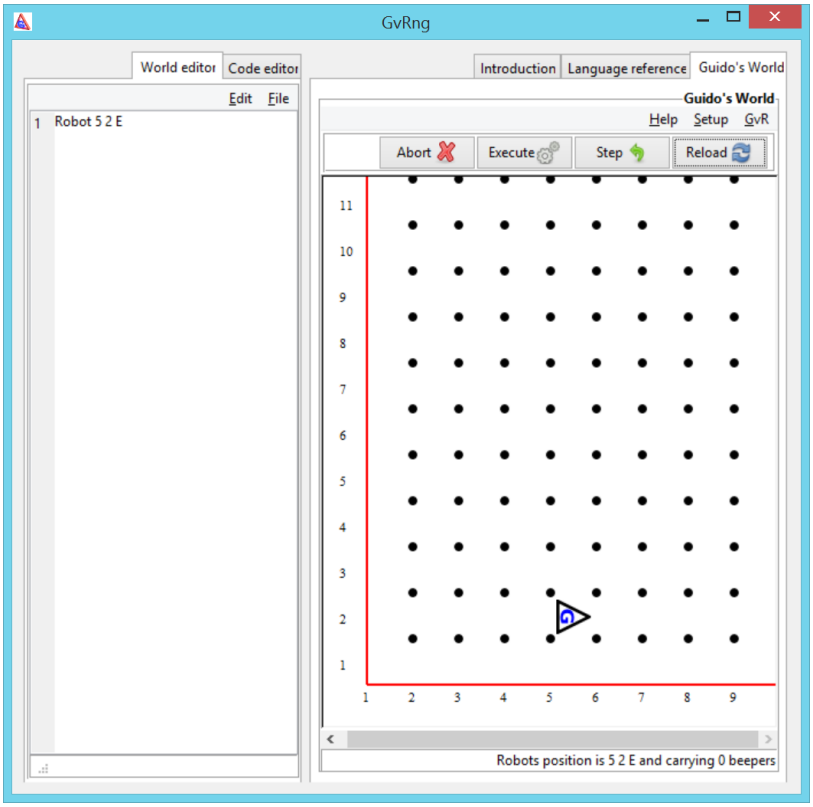
איזה יופי, Guido הגיעה אל המסך. היא מוצגת בתור משולש בו כתובה האות **G**. זה הזמן להכיר קצת את העולם שבו Guido חיה. העולם מחולק לטורים ושורות, ממש כמו בטבלה. גם השורות וגם הטורים ממוספרים, החל מ-1. כך העולם מחולק למעשה למשבצות. כרגע, Guido נמצאת במשבצת בטור השלישי, בשורה הרביעית.

נסתכל שוב בשורה שרשמנו בלשונית **World editor**:

Robot 3 4 E

בשלב זה, נוכל כבר לנחש את משמעות המספרים 3 ו-4: אלו מספר הטור ומספר השורה בהם Guido ממוקמת, בהתאמה. על מנת לבדוק זאת, ננסה לשנות אותם. נמחק את השורה הקודמת שכתבנו, וכעת נציג את Guido במשבצת שבטור החמישי, ובשורה השניה, כך:

Robot 5 2 E



בשלב זה בנינו עולם פשוט מאוד. הוא מכיל שורות וטורים, ללא תוספות מיוחדות. בעתיד הקרוב מאוד, נלמד כיצד לבנות עולמות מעניינים יותר.

כעת, נוכל לשמור את העולם שיצרנו. לשם כך, נוכל להשתמש בתפריט **File** הממוקם מעל לאיזור הפקודות. ניתן לבחור ב-**Save** או ב-**Save as**[3]. שמרו את קובץ העולם בשם: first_world.wld[4]. באופן כללי, כדאי להתרגל לשמור את קבצי העבודה שלנו באופן תדיר לצורך גיבוי. כך בכל מקרה לא נאבד קובץ שעבדנו עליו במשך זמן רב.

כעת כדאי שנלמד פרט חשוב על Guido: היא תמיד פונה לאחד מארבעה כיוונים: **N**orth (צפון), **S**outh (דרום), **E**ast (מזרח) או **W**est (מערב). בעולם בו יצרנו, Guido פונה לכיוון **E**ast (מזרחה). ניתן לדעת זאת אם נסתכל על המשולש שמייצג אותה – הוא "פונה" ימינה, כלומר לכיוון מזרח. נחזור שוב לשורה שמתארת עבורינו את העולם, האם תוכלו לנחש איזה חלק ממנה מורה ל-Guido לפנות מזרחה?

Robot 5 2 E

[3] נוכל גם להשתמש בקיצור המקלדת Ctrl+S. באופן כללי, קיצורי מקלדת עוזרים מאוד לקצר את זמן העבודה ומאפשרים לנו להתרכז בדברים המעניינים יותר בתכנות. נסו להתרגל להשתמש בהם באופן קבוע.
[4] נשתמש בסיומת הקובץ wld עבור קבצים שמתארים את העולם של Guido.

ניחשתם נכון – האות **E** מציינת כי Guido פונה לכיוון מזרח (**E**ast). באופן כללי, ניתן כעת להבין כי המבנה של הפקודה הינו כזה:

Robot <מספר טור> <מספר שורה> <כיוון>

כאשר הכיוון יכול להיות אחד מארבעת הערכים: N, S, E או W.

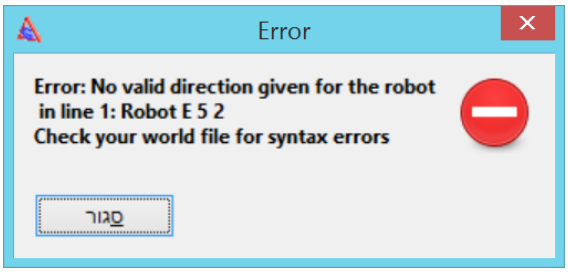
לכל ערך כזה של פקודה (מספר טור, מספר שורה או כיוון) קוראים **פרמטר**. לפקודה **Robot**, כפי שאנו מבינים עתה, יש שלושה **פרמטרים**.

בשלב זה נשים לב כי Guido פונה אך ורק לארבעת הכיוונים שצויינו לעיל. היא לא יכולה, למשל, לפנות לכיוון דרום-מזרח.

בתחילת הפרק, כשהסברנו מדוע אנו מתחילים את מסע הלמידה שלנו בתכנות מ-Guido, הזכרנו ששפות תכנות דורשות מאיתנו דיוק – עלינו לכתוב את הפקודות באופן מדויק, שתואם את דרישות השפה. נוכל לראות זאת כבר עכשיו. אם, למשל, ננסה להפוך את הסדר של הפקודה:

Robot E 5 2

וללחוץ על **Reload**, אנו צפויים לקבל מיד הודעת שגיאה:



חשוב שנתרגל כבר בשלב זה לראות שגיאות, ולנסות ללמוד מהן. במהלך תהליך התכנות, אנו צפויים להתקל בשגיאות לעתים תכופות, ועלינו לדעת לקרוא אותן. במקרה זה, נכתב לנו שהשגיאה ארעה בשורה 1 בקובץ העולם. מידע זה לא עוזר לנו רבות, בהינתן שכרגע קובץ העולם שלנו מכיל שורה אחת בלבד. עם זאת, הוא יהיה רלבנטי כאשר קובץ העולם שלנו יתארך.

בנוסף, אנו מקבלים המלצה לבדוק את קובץ העולם ולחפש בו שגיאות **syntax**.

הכוונה ב-**syntax** היא לסט החוקים של השפה.

במקרה זה, חרגנו מהסדר שבו יש לכתוב את הפקודה, ולכן היא לא עבדה כמו שצריך.

תרגיל 1.2 – בניית עולם פשוט

כעת אתם מצויידים בכל הידע הדרוש בשביל לבנות עולמות פשוטים. נסו לבנות את העולמות הבאים:

1) עולם בו Guido נמצאת בטור השני, בשורה השביעית. עליה לפנות לכיוון מערב.
2) עולם בו Guido נמצאת בפינה השמאלית התחתונה. עליה לפנות לכיוון צפון.

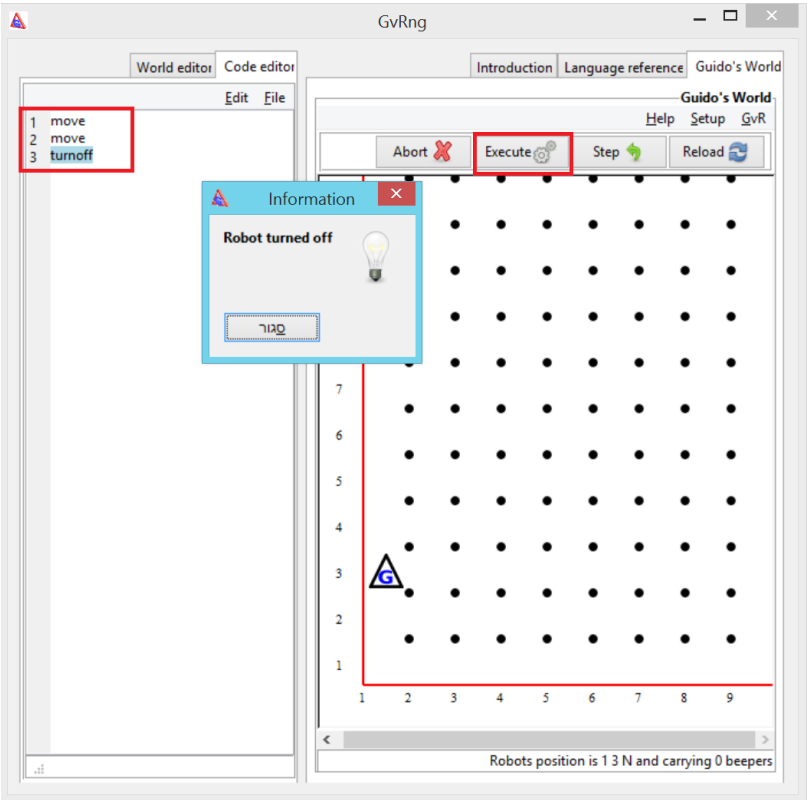
תרגיל מודרך 1.3 – תנועה

מצויין, הצלחנו ליצור עבור Guido עולם, וגם למקם אותה על המסך. אבל מה הרובוטית הזו יודעת לעשות בכלל? הדבר הראשון ש-Guido יודעת לעשות הוא לנוע בתוך העולם שלה. בכל שלב, Guido יכולה לנוע צעד אחד בלבד. היות ש-Guido יכולה לפנות רק לאחד מארבעה כיוונים (צפון, דרום, מזרח או מערב) - היא תנוע או בטור, או בשורה.

נשתמש בעולם האחרון שכתבתם בתרגיל 1.2 – עולם בו Guido נמצאת בפינה השמאלית התחתונה, כשהיא פונה לכיוון צפון. כעת, נכתוב את שורות הפקודה הראשונות שלנו שלא מתייחסות לעולם – אלא ל-Guido עצמה. כלומר, אלו יהיו פעולות ש-Guido תבצע, ולא פעולות שמגדירות איך העולם הכללי נראה.
לשם כך, נלחץ על הלשונית **Code editor**, ונכתוב את השורות הבאות:

```
move
move
turnoff
```

שימו לב שכל פקודה נמצאת בשורה נפרדת. כעת, נלחץ על Execute, והקוד שלנו ירוץ. במידה ומילאנו אחר ההוראות, אנו צפויים לראות את המסך הבא:

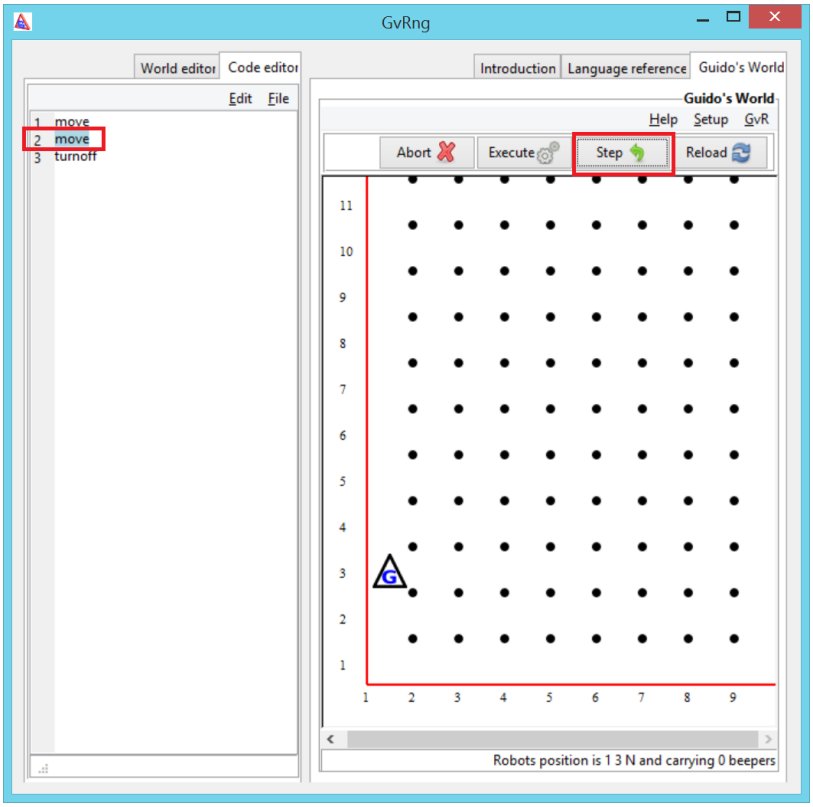


היות ש-Guido פנתה צפונה, ונתנו לה פעמיים את פקודת **move**, היא נעה שתי משבצות לכיוון צפון, אחת בכל פעם. לבסוף, היא נכבתה.

מה בעצם קרה כאן?

למעשה, הרצתם את **שורות הקוד** הראשונות שלכם. מזל טוב! כפי שניתן לראות בתמונה, השורות ממוספרות (במקרה זה מ-1 עד 3). כל שורה מורצת בנפרד, והשורות מורצות בזו אחר זו. כלומר, ראשית תרוץ פקודת ה-**move** הראשונה, לאחר מכן פקודת ה-**move** השניה, ולבסוף תרוץ פקודת **turnoff**.

Guido מאפשרת לנו לראות את הקוד שלנו רץ שורה אחר שורה. לשם כך, נלחץ על הכפתור **Reload**. כעת, בכל פעם שנלחץ על **Step** – נראה שורה שרצה. לאחר שתי לחיצות, אנו צפויים לראות את המסך הבא:



Guido ביצעה פעמיים פעולת move, והיא כרגע במשבצת בטור הראשון, בשורה השלישית. כמו שניתן לראות, בלשונית **Code editor**, מסומנת הפקודה האחרונה שרצה.

ממש כמו ששמרנו את העולם שיצרנו, ניתן לשמור גם את הקוד ש-Guido מריצה. נשמור אותו בשם first_code.gvr.[5]

תרגיל 1.4 – תנועה בעולם פשוט

אז אנחנו יודעים ליצור עולם פשוט, ואנחנו יודעים לנוע. צרו עולם בו Guido ממוקמת בטור החמישי, בשורה החמישית, ופונה דרומה. גירמו לה להתקדם עד שתגיע אל השורה הראשונה. לאחר מכן, כבו את Guido.

סיכום ביניים – עולם פשוט ותנועה

אז רק התחלנו ללמוד לתכנת, ואנחנו כבר יודעים לבנות עולמות פשוטים, כמו גם להזיז את Guido. את העולמות אנו יוצרים ב-**World editor**, ואת הקוד ל-Guido אנו כותבים ב-**Code editor**. למדנו שהעולם של Guido

[5] נשתמש בסיומת קבצים gvr כדי לתאר קבצי קוד ל-Guido עצמה.

מורכב ממשבצות, שממוקמות בשורות וטורים. נוכל לזהות כל משבצת באמצעות מספר הטור והשורה שבהם היא ממוקמת.

הכרנו את המונח **syntax**, וגם ראינו **syntax** של פקודות שונות. על מנת ליצור עולם, למשל, ראינו שיש את ה-**syntax** הבא:

Robot <מספר טור> <מספר שורה> <כיוון>

נוכחנו כי עלינו להיות מאוד מדוייקים, ולכתוב כל פרמטר במקום המתאים לו. אם ננסה לכתוב את הכיוון לפני מספר הטור, למשל, נקבל שגיאה.

ה-syntax של הפקודה **move**, מאידך, הרבה יותר פשוט. אנו כותבים רק:

move

בשורה בפני עצמה.

בסך הכל למדנו על שתי פקודות ש-Guido מכירה: **move** ו-**turnoff**. בקרוב מאוד נכיר כמה פקודות נוספות.

באופן כללי, אנחנו מתחילים להבין מה המשמעות של קוד – שורות של פקודות, שרצות בזו אחר זו. Guido מבצעת בדיוק את מה שנכתוב לה לבצע, ממש לפי הפקודות שנרשום. אם נרשום פקודות אחרות, היא תפעל אחרת. יחדיו, שורות הקוד האלו מהוות **תוכנית**.

אז אחרי שעשינו צעד קטן לרובוטית, וצעד גדול ליכולות התכנות שלנו – הגיע הזמן ללמוד דברים נוספים.

פניות וסיבובים

לנוע זה נחמד, אבל נכון לעכשיו – Guido מאוד מוגבלת. אם נגרום לה להתחיל כשהיא פונה לכיוון מערב – היא תוכל לנוע רק מערבה. זה לא מאפשר לנו הרבה מרחב תמרון. עם זאת, כפי שקורה לעתים, המפתחים של הרובוטית נתקלו בקשיים תקציביים והחליטו להעניק ל-Guido את היכולת לפנות שמאלה בלבד. כך, בעודה נשארת על אותה המשבצת בעולם, Guido יכולה לפנות ולהסתובב נגד כיוון השעון. בכל פעם, Guido תפנה כך לכיוון אחר. הפקודה שמבצעת זאת היא:

turnleft

רגע, מה? עדיין לא ניסיתם את זה? כנראה שאתם עדיין לא מספיק מתרגשים מהעניין.

תרגיל 1.5 – תנועה בעולם פשוט

קדימה, נסו לשחק עם הפקודה החדשה שלמדתם. צרו עולם שבו Guido נמצאת בפינה השמאלית התחתונה, כשהיא פונה מזרחה. גרמו ל-Guido לנוע שני צעדים לכיוון מזרח, ולאחר מכן להסתובב לכיוון צפון. גרמו לה לנוע

שני צעדים צפונה, ואז להסתובב לכיוון מערב. המשיכו וגרמו ל-Guido לנוע שני צעדים מערבה, ולבסוף להסתובב ולחזור לנקודת ההתחלה.

תזכורת: אל תשכחו לכבות את Guido בסוף התכנית שלכם. כל תכנית שנכתבת ל-Guido חייבת להסתיים בכיבוי. גם רובוטים צריכים לישון, אחרי הכל.

ביפ ביפ!

העולם של Guido לא היה מעניין כל כך לולא היו בו עצמים נוספים פרט ל-Guido עצמה. הפריט העיקרי שנמצא בעולם של Guido הם ביפרים (beepers). ביפר הוא מכשיר שיכול להשמיע קולות, ו-Guido יכולה לשמוע את הקולות האלו. היא יכולה לאסוף ביפרים שהיא מוצאת בדרך, והיא יכולה להניח ביפרים במשבצות שונות בעולם. Guido מצויידת מראש בתיק גב אופנתי[6], שבו היא יכולה לאכסן את הביפרים שהיא אוספת.

כאשר אנו בונים עולם עבור Guido, אנו מציינים את כמות הביפרים של-Guido יש בתיק בעת תחילת התכנית. על מנת לעשות זאת, נשתמש בחלק נוסף של הפקודה שיצא לנו להכיר קודם לכן:

Robot <מספר טור> <מספר שורה> <כיוון> <מספר ביפרים>

שימו לב לתוספת – מספר ביפרים. כך למשל, הפקודה הבאה:

Robot 2 3 N 4

תיצור עולם שבו Guido נמצאת בטור השני, בשורה השלישית, היא פונה צפונה, ול-Guido יש ארבעה ביפרים בתיק. נשים לב כי הביפרים נמצאים אצל Guido בשלב זה, ולא מפוזרים ברחבי העולם.

רגע, אבל אמרנו שלפקודה Robot יש רק שלושה פרמטרים! איך זה אפשרי?

לפקודה Robot יש פרמטר רביעי שהוא **פרמטר אופציונאלי**, כלומר – לא חייבים לציין אותו. במידה שלא נציין מספר ביפרים, ניצור עולם שאין בו ביפרים כלל.

בשלב זה נכיר שתי פקודות חדשות, אחת בכדי להרים ביפר שנמצא על משבצת בעולם, והשניה כדי להניח ביפר שנמצא בתיק הגב של Guido על משבצת:

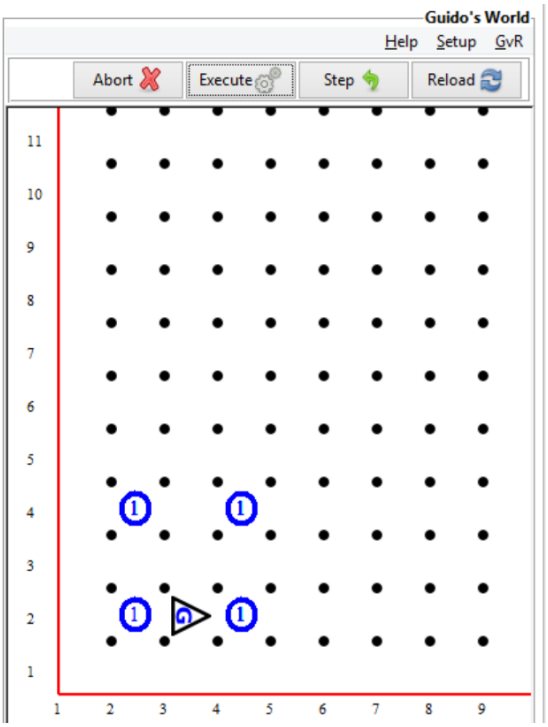
pickbeeper

putbeeper

[6] עם תמונות של R2-D2, Wall-E, רובוקופ ועוד "סלבס" מעולם הרובוטים.

תרגיל 1.6 – ריבוע של ביפרים

צרו עולם בו Guido נמצאת במשבצת בטור השני, בשורה השניה. נועו, בצעו פניות והניחו ביפרים כך שבסוף הריצה של התוכנית, העולם יראה כך:



המלצה: היעזרו ביכולת הנהדרת של Guido לראות את הקוד שלכם רץ, פקודה אחר פקודה, באמצעות הכפתור **Step**. כך תוכלו לזהות טעויות בקוד שלכם, ולהבין מה קורה בכל שלב.

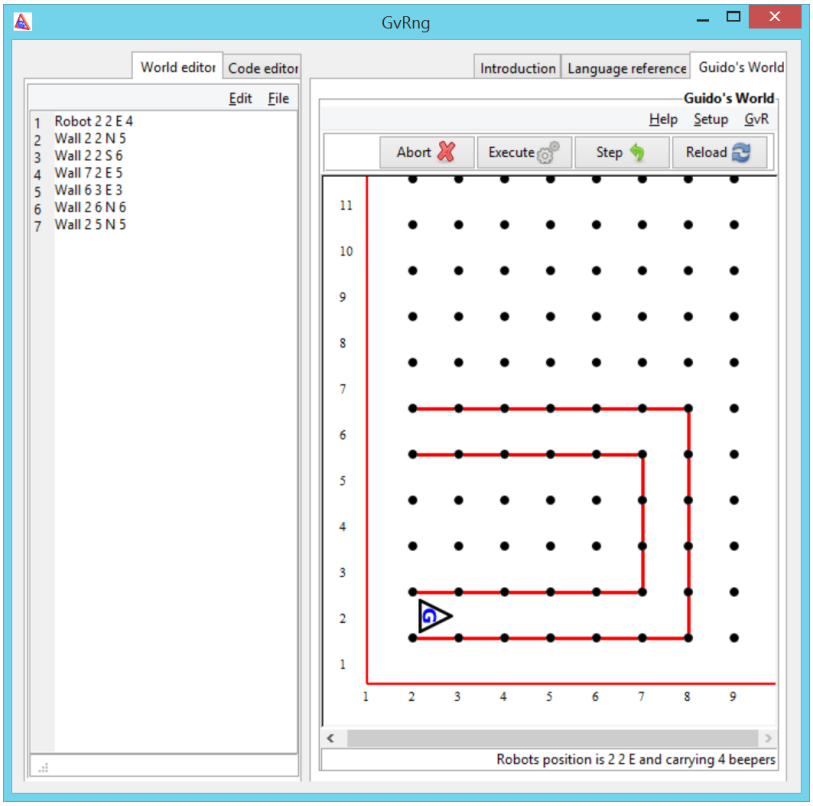
חומות של תקווה

Guido ראתה פעם את הסרט חומות של תקווה, וכמעט התרגשה (זה הרבה, בשביל רובוטית). גם בעולם שלה, יש לעתים חומות. בשלב זה לא נלמד איך ליצור חומות בעצמנו, אך נכיר כיצד Guido פועלת בעולם עם חומה.

נכנס אל לשונית ה-**World editor**, ונדביק לתוכה את הפקודות הבאות:

```
Robot 2 2 E 4
Wall 2 2 N 5
Wall 2 2 S 6
Wall 7 2 E 5
Wall 6 3 E 3
Wall 2 6 N 6
Wall 2 5 N 5
```

לאחר לחיצה נוספת על כפתור ה-**Reload**, נקבל עולם שנראה כך:

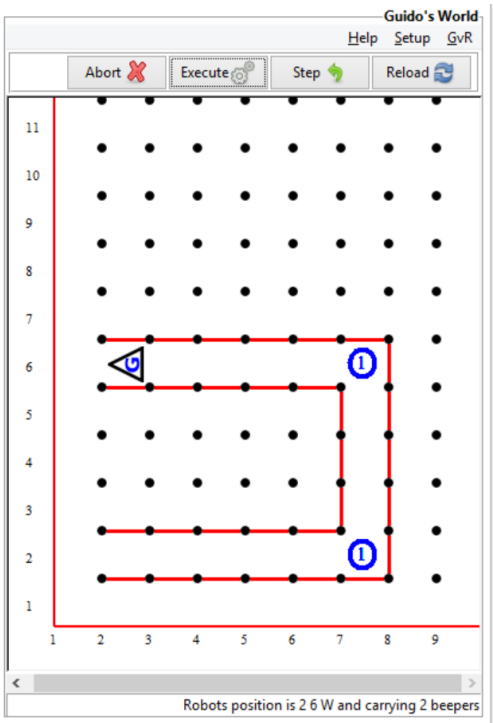


הקווים האדומים העלו מתארים חומות. הם אמנם נראים קטנים, אך Guido מאוד לא אוהבת להתקל בהם. אם נכתוב לה, באופן מרושע למדי, תכנית שמבצעת שש פעמים את הפקודה **move**, נקבל את האזהרה הבאה:

בשלב זה, הקוד שלנו יפסיק לרוץ, ו-Guido תתעצב. אי לכך, נשאף להמנע מפגיעה בחומות!

תרגיל 1.7 – התחמקות מחומה

בעולם שזה עתה ראינו, עליכם לכתוב קוד שיגרום ל-Guido להגיע אל המשבצת בטור השני, בשורה השישית, ולהניח בדרכה שני ביפרים, כך שבסוף ריצת התכנית העולם יראה באופן הבא:


Guido's World
Help Setup GvR
Abort
Execute
Step
Reload
11
10
9
8
7
6
5
4
3
2
1
1 2 3 4 5 6 7 8 9
Robots position is 2 6 W and carrying 2 beepers

פניה ימינה – ופונקציות

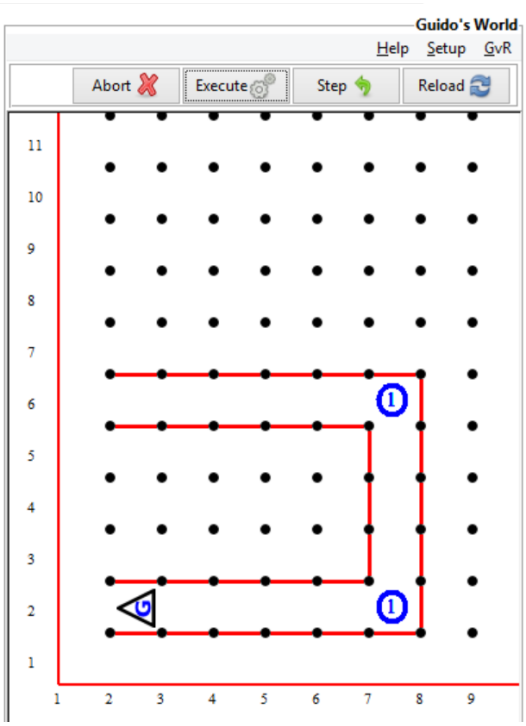
ראשית, נתחיל בחידה:

כיצד ניתן לגרום ל-Guido לפנות ימינה?

ובכן, הדרך לעשות זאת יכולה להיות שימוש שלוש פעמים בפקודה שכבר הכרנו, **turnleft**.

תרגיל 1.8 – התחמקות מחומה בפניות ימינה

כעת, עליכם להשתמש בעולם מתרגיל 1.7, אך הפעם גירמו ל-Guido להתחיל במשבצת בטור השני, בשורה השישית, כשהיא פונה מזרחה. הניחו את הביפרים באותם המקומות בדיוק כמו בתרגיל 1.7, וסיימו את ריצת התכנית כש-Guido נמצאת במשבצת בטור השני, בשורה השניה:



בשלב זה, הקוד שכתבנו יכול היה להיראות, פחות או יותר, כך:

```
move
move
move
move
```

```
move
turnleft
turnleft
turnleft
putbeeper
move
move
move
move
turnleft
turnleft
turnleft
putbeeper
move
move
move
move
move
turnoff
```

לא פחות מ-23 שורות קוד עבור הפעולה הפשוטה שביצענו. זאת ועוד, בוודאי חשתם תחושה מוזרה – מדוע עלינו לכתוב שוב ושוב שלוש פעמים את הפקודה **turnleft**, כשברור לנו שהמשמעות של שלוש השורות האלה היא למעשה פנייה ימינה?

על מנת לשפר את הקוד שלנו, נכיר מושג חדש:

פונקציה היא רצף של פקודות, המאוגדות יחדיו, ומבצעות פעולה מסויימת.

תרגיל מודרך 1.9 – הגדרת פונקציה

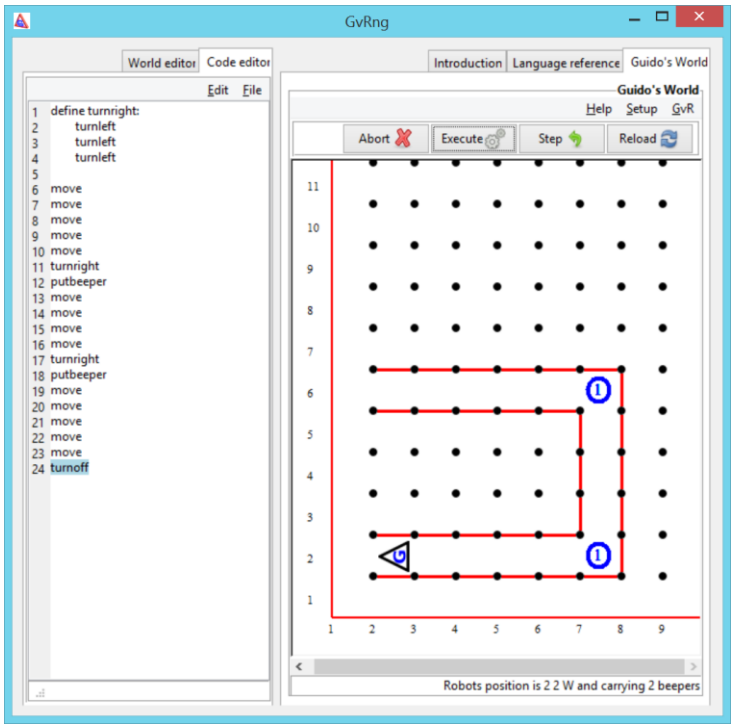
נתחיל מדוגמה – נכתוב בעצמנו את הפונקציה **turnright**, שתגרום ל-Guido לפנות ימינה. נעשה זאת כך:

```
define turnright:
    turnleft
    turnleft
```

turnleft

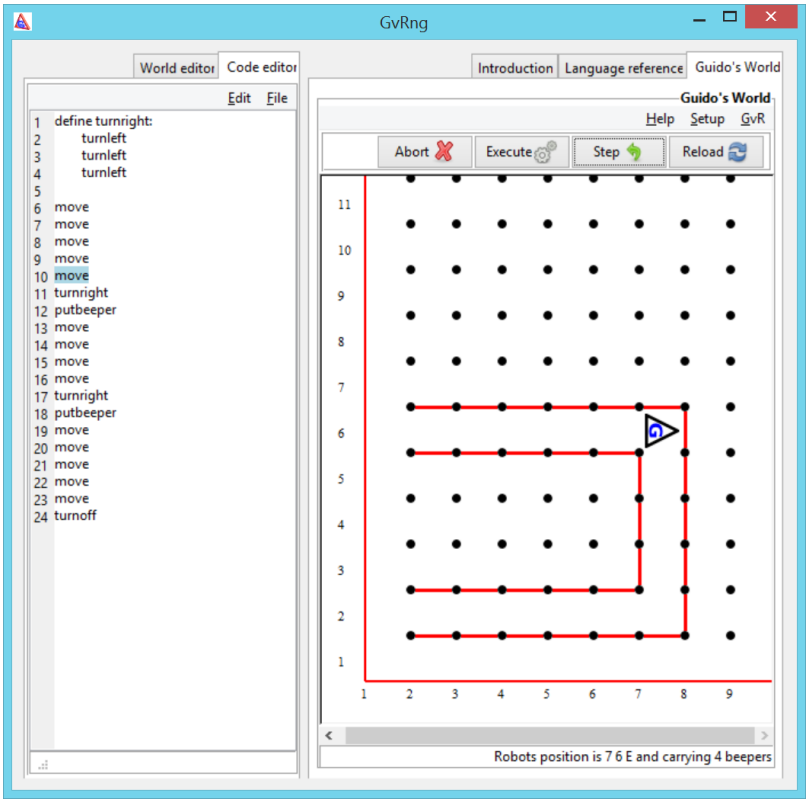
כעת ננתח את הקוד שכתבנו. ראשית, השתמשנו במילה **define**, ולאחריה ציינו את שם הפונקציה שאנו מגדירים - **turnright**. את השורה הזו סיימנו בתו נקודותיים (**:**). כעת, גוף הפונקציה, כולל פקודה אחת או יותר, שיקראו בכל פעם שנקרא לפונקציה שהגדרנו. שימו לב כי גוף הפונקציה נכתב ב**אינדנטציה** מסויימת – מספר של רווחים שמופיע לפני כל שורה שנמצאת בגוף הפונקציה. כך אנו מבדילים את גוף הפונקציה משאר הקוד של התכנית. את קוד זה נכתוב בתחילת התכנית שלנו, בפרט - לפני הקוד שמשתמש בפקודה **turnright** (ראו צילום מסך בהמשך).

כעת, נשתמש בפונקציה שהגדרנו בקוד הכללי של התכנית. במקום הקוד שכתבנו קודם, נוכל להחליף כל שלוש שורות **turnleft** בשורה יחידה של **turnright**:

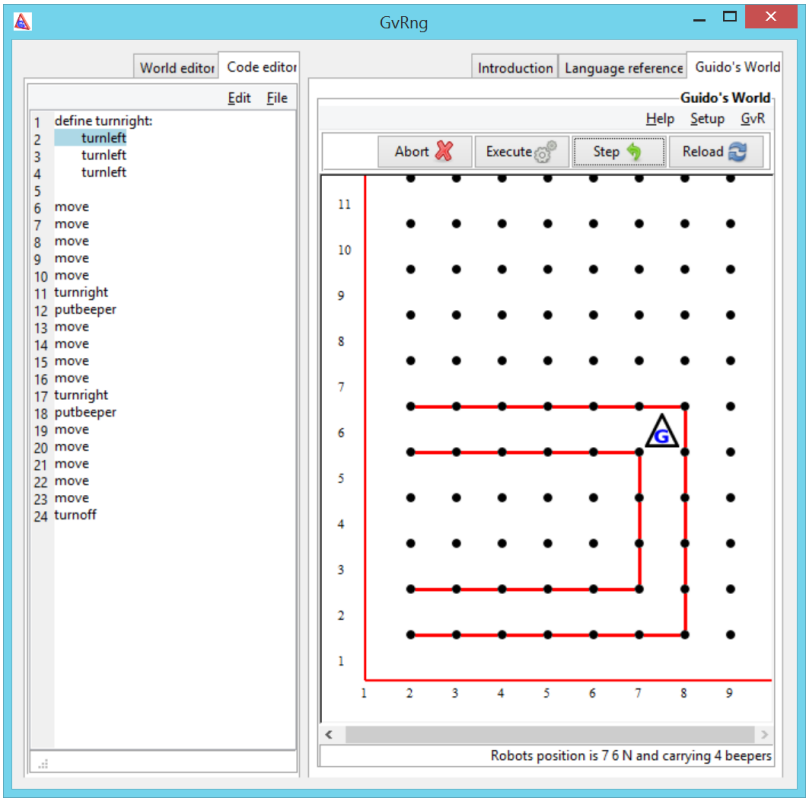


זה כבר נראה הרבה יותר טוב!

בואו ננסה להבין מה קורה. נתחיל מחדש את התכנית, ונלחץ על **Step** חמש פעמים. השורה שאמורה להיות מסומנת כעת היא שורה 10, פקודת ה-**move** האחרונה לפני הקריאה לפונקציה שזה עתה הגדרנו, **turnright**:

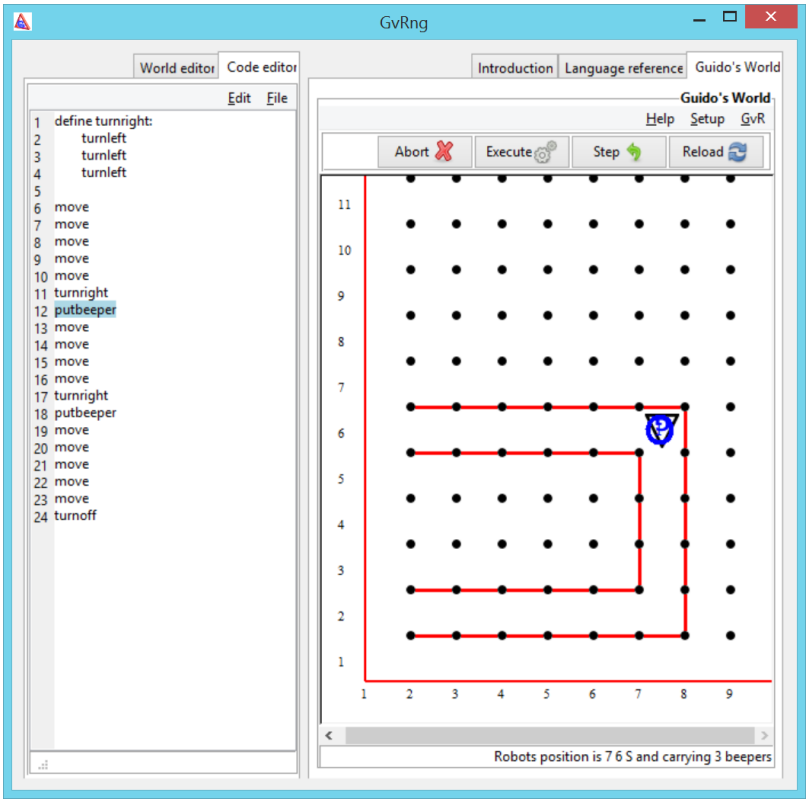


כעת, נלחץ שוב על הכפתור **Step**.



נראה שהקוד "קפץ" אחורה! למעשה, ברגע ש-Guido זיהתה שהיא התבקשה להריץ את הפקודה **turnright**, היא "קופצת" אל ההגדרה של **turnright** ומבצעת אותה.

נלחץ על **Step** עוד שלוש פעמים, ונראה כי Guido המשיכה בריצת התכנית באופן הצפוי:



המשיכו בעצמכם – השתמשו במקש **Step** כדי לרוץ על כל שורות הקוד בתכנית. שימו לב איזו שורה מתבצעת בכל שלב.

שאלת מחשבה – מה הרווחנו כאן?

אז למדנו לכתוב פונקציות משלנו, נהדר! אבל מה בעצם השגנו? למה לכתוב פונקציות זה כל כך חשוב כשמתכנתים?

לשאלה זו יש תשובות רבות מאוד. בשלב זה, נתמקד בעיקריות שבהן:

קודם כל, הקוד הרבה יותר **קריא** – בתור אדם שקורא את הקוד, קל יותר להבין ש-**turnright** מבצעת פנייה ימינה, מאשר שלוש קריאות ל-**turnleft**. בנוסף על כך, חסכנו שורות קוד. דבר זה מקל אף הוא על מידת הקריאות.

בנוסף, הקוד שלנו יותר **מודולרי** – נאמר שהיינו רוצים לשנות את הפונקציה **turnright** מסיבה כלשהי. למשל, גילינו שבכל פניה ימינה, כדאי ש-Guido גם תשליך ביפר אל המשבצת שעליה היא נמצאת. אם לא היינו כותבים את הפונקציה **turnright**, היינו צריכים ללכת לכל מקום בקוד שבו יש שלוש פקודות **turnleft**, ולהוסיף את

הפקודה **putbeeper**. לעומת זאת, בזכות העובדה שהגדרנו את הפונקציה **turnright**, נוכל רק לשנות את המימוש שלה. כך למשל, נוכל להוסיף את הפקודה **putbeeper** רק בגוף הפונקציה **turnright**:

```
define turnright:
    turnleft
    turnleft
    turnleft
    putbeeper
```

בעקבות הוספת שורה אחת בלבד, הצלחנו לגרום לשינוי **בכל** הקריאות לפונקציה **turnright** בתכנית שלנו.

כאמור, ישנן סיבות רבות נוספות לשימוש בפונקציות, אך לא נסקור את כולן בשלב זה. שתי הסיבות שתארנו לעיל בהחלט משכנעות שנשאף להשתמש בפונקציות באופן מירבי בקוד שלנו.

פונקציות הן כלי נהדר. נשתמש בהן!

תרגיל 1.10 – פונקציית תזוזה מרובה

הקוד האחרון שראינו, שמשתמש בפונקציה **turnright**, עדיין ארוך למדי. באופן ספציפי, ישנן הרבה קריאות לפקודה **move**. הגדירו פונקציה בשם **move_4_times**, שבאופן לא מפתיע מבצעת 4 פעמים את הפקודה **move**. החליפו כל ארבע שורות רצופות של הפקודה **move** בקריאה לפונקציה החדשה שהגדרתם. שימו לב עד כמה הקוד שלכם הפך לקצר וקריא יותר.

תרגיל 1.11 – פונקציית תזוזה אחורנית

כעת, הגדירו פונקציה בשם **backup**. הפונקציה תגרום ל-Guido לנוע צעד אחד אחורה. כלומר, בסוף ריצת הפונקציה, Guido תפנה עדיין לאותו כיוון שאליו פנתה לפני ריצת הפונקציה, אך במיקום אחד אחורנית. לדוגמה, אם Guido הייתה במשבצת בטור השלישי, בשורה השניה, ופנתה מזרחה – לאחר ריצת הפונקציה היא תהיה במשבצת בטור השני, בשורה השניה, ותפנה מזרחה.

השתמשו בפונקציה זו, יחד עם הפונקציה **turnright** והפקודות **move** ו-**turnleft** בכדי לגרום ל-Guido לנוע בעולם. כמה חופש פעולה יש לנו פתאום!

פרק 1 - סיכום

בפרק זה למדנו רבות. התחלנו מלהכיר את Guido, ולהבין מדוע אנו משתמשים ברובוטית כדי להתחיל את מסעינו בעולם התכנות. בנינו עולמות על ידי שימוש ב-**World editor**, וכתבנו קוד ל-Guido ב-**Code editor**. הכרנו את העולם של Guido, ולמדנו שהוא מורכב ממשבצות.

הכרנו חמש פקודות ש-Guido מכירה מיום היוולדה[7]:

move

turnleft

pickbeeper

putbeeper

turnoff

ראינו שלפקודות שונות יש **syntax** שונה, עם מספר שונה של **פרמטרים**, שחלקם עשויים להיות **פרמטרים אופציונאליים**. נוכחנו לדעת שחשוב לשמור על חוקי ה-**syntax**.
ראינו כיצד פועל קוד – שורות של פקודות שרצות בסדר מסויים.

הכרנו אובייקטים חשובים בעולמה של Guido – **ביפרים**, שהם מכשירי שמע ש-Guido יכולה להניח או להרים, ו**חומות** – בהן Guido מאוד לא אוהבת להתנגש. למדנו להניח ולהרים ביפרים, וכן התאמנו בהתחמקות מחומות.

בסוף הפרק, למדנו על נושא חשוב במיוחד – **פונקציות**. ראינו כי פונקציה היא רצף של פקודות, המאוגדות יחדיו, ומבצעות פעולה מסויימת. הגדרנו את הפונקציות **turnright**, **move_4_times** וכן **backup**. הפונקציות האלו הקלו עלינו מאוד בכתיבת הקוד שלנו ל-Guido. ראינו שהן משפרות מאוד את **הקריאות** של הקוד, וכן את ה**מודולריות** של הקוד. השתמשנו בכפתור **Step** בכדי לעקוב אחר אופן פעולת הקוד בעת קריאה לפונקציה.

בפרק הבא נלמד עוד על עולמה של Guido. נלמד כיצד ליצור בעצמנו עולמות מורכבים יותר, ונלמד לאפשר ל-Guido לקבל החלטות בזמן אמת: מה לעשות במצבים שונים. כמו כן, נכיר שיטות נוספות כדי להקל על הקריאות של הקוד שלנו ולעשות דברים מורכבים יותר.

אנחנו לא יודעים מה איתכם, אבל Guido כבר לא יכולה לחכות.

[7] Guido מעדיפה את המונח "היוולדה" על "יצורה במפעל". אנחנו משתדלים להתחשב ברגשותיה.

פרק 2 – תנאים, לולאות ועולמות

בפרק הראשון למדנו המון דברים. הכרנו את Guido, הבנו איך בונים עולם פשוט, למדנו מה זה קוד, syntax, פונקציות... ולא שכחנו להנות כל הדרך.
בפרק זה נגיע לעולמות חדשים, ויחד איתנו – גם Guido. נתחיל מללמוד איך ליצור עבור Guido עולמות מעניינים יותר, ונמשיך בלמידה של קונספטים חשובים מאוד בתכנות – תנאים ולולאות.

עולמות עם חומות וביפרים

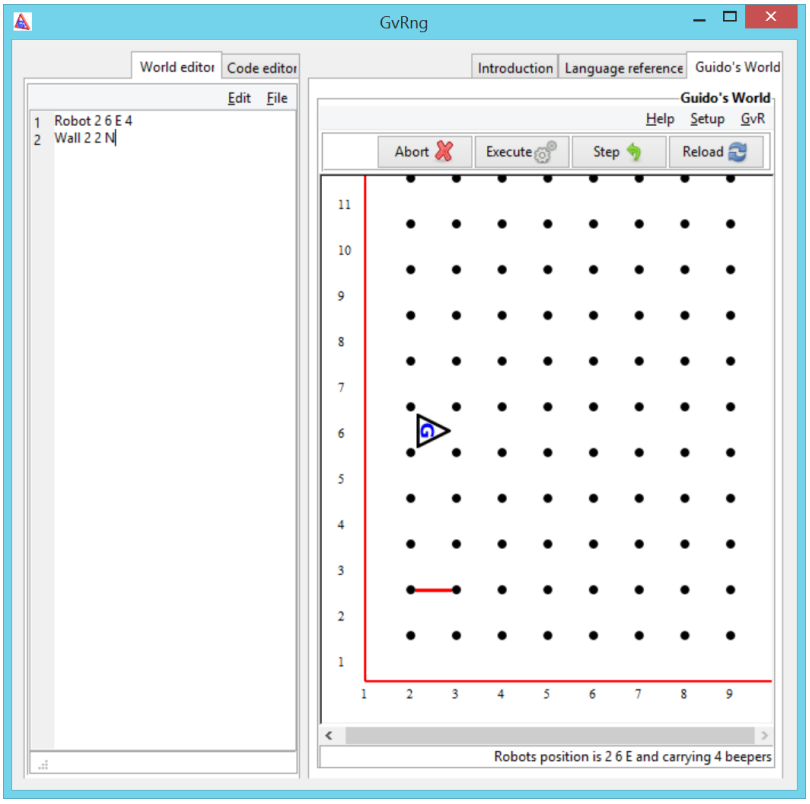
כבר פגשנו בפרק הקודם בעולם עם חומות. נסתכל כעת על ה-syntax של הפקודה **Wall**, שמייצרת חומה כחלק מהקוד של העולם:

Wall <מספר טור> <מספר שורה> <הכיוון אליו המשבצת נחסמת> <אורך החומה>

רוב הפרמטרים כאן מוכרים לנו. מספר הטור ומספר השורה מציינים את המקום ממנו החומה מתחילה. הכיוון אליו המשבצת נחסמת יכול להיות N, S, W או E – בדומה מאוד ל-syntax שפגשנו בפקודה **Robot**. אורך החומה הוא פרמטר אופציונלי. אם לא נגדיר אותו באופן מפורש, תיווצר לנו חומה באורך של משבצת אחת.

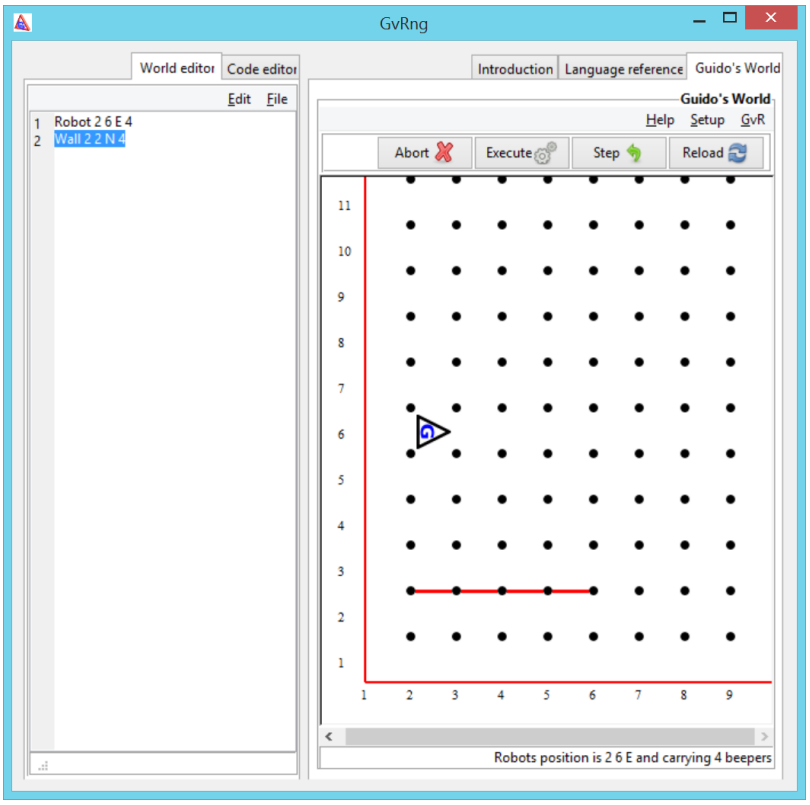
קדימה, בואו נשחק! נבנה עולם עם חומה קצרה מאוד, שמתחילה במשבצת בטור השני, בשורה השניה, וחוסמת אותה מכיוון צפון. היא תהיה באורך 1. לשם כך נכתוב את השורה:

Wall 2 2 N 1



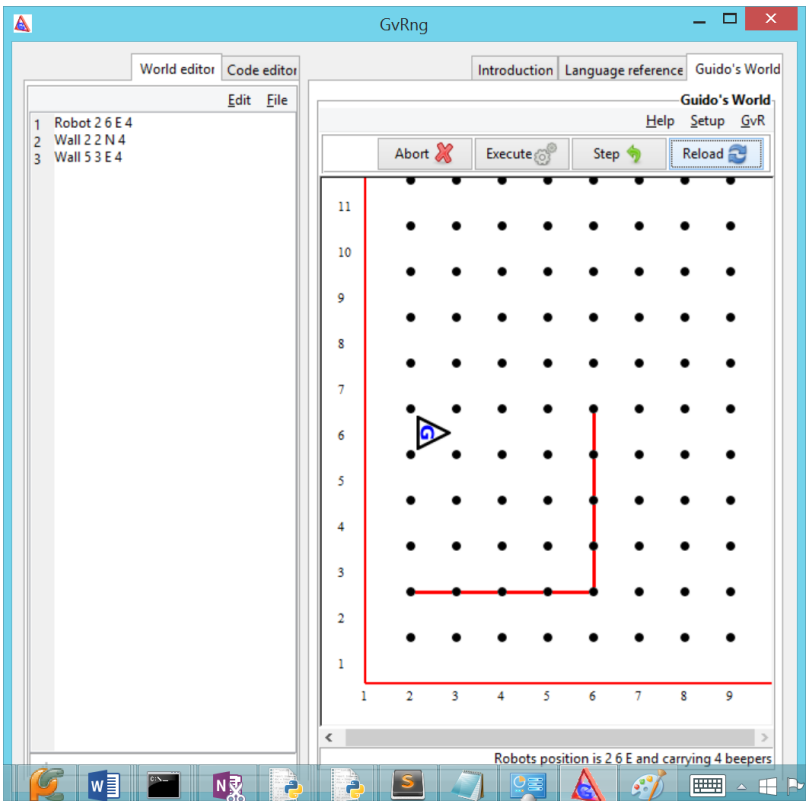
כעת נוכל להאריך קצת את החומה:

Wall 2 2 N 4



שימו לב לדרך המחשבה בה נקטנו: ראינו פקודה, והבנו את הפורמט שלה. כעת, כדי להרגיש שאנו שולטים בפקודה ובאמת מבינים אותה, התחלנו "לשחק" איתה. יצרנו חומה פשוטה, ואז שינינו רק פרמטר אחד שלה – פרמטר האורך.

כעת נבנה חומה נוספת, שתשיק לחומה שבנינו. נרצה שהיא תתחיל מהטור החמישי, מהשורה השלישית, ותחסום את המשבצת הזו מצד מזרח. גם את החומה הזו נגדיר באורך 4.

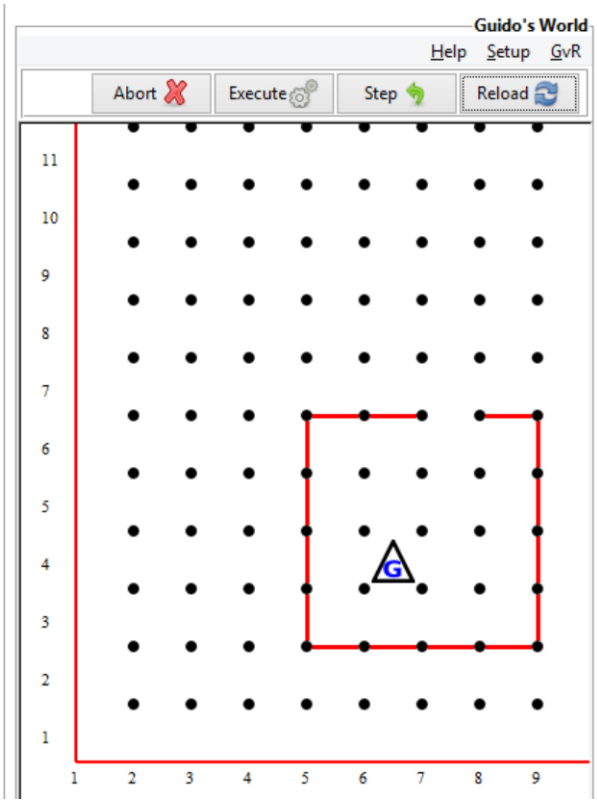


תרגיל 2.1 – כלא בצורת ריבוע

מה זה, כמעט שלושה עמודים שלמים לתוך הפרק הזה – ועדיין לא תרגלנו. זה לא לעניין. קדימה, השלימו את שתי ה"צלעות" שנבנו עד כה לכדי ריבוע ש"יכלא" את Guido בתוכו.

תרגיל 2.2 – עולם עם מסר

Guido לא אוהבת להיות בתוך כלא, חסרה לה אינטראקציה אנושית. כלומר, רובוטית. בנו ל-Guido בית עם דלת קטנה, שנראה כך:

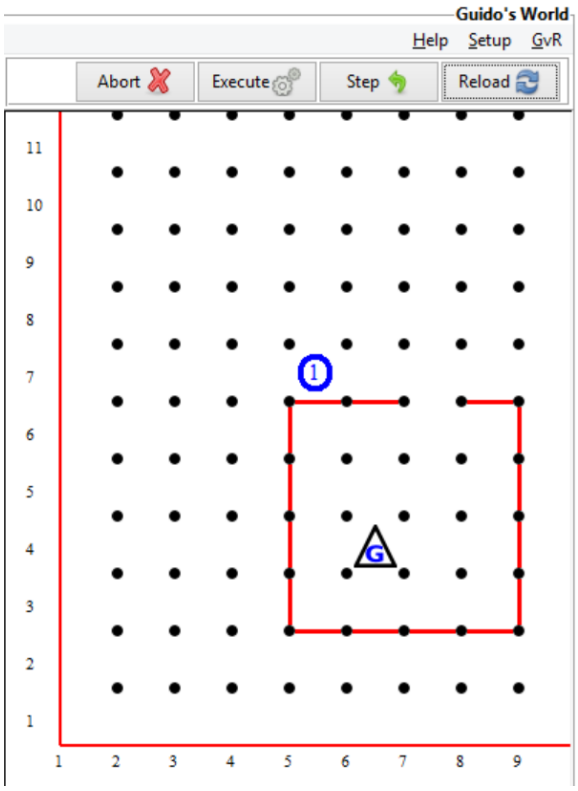


מידי פעם מגיע Syntix, חברה הטוב של Guido, ומשאיר לה מסר מחוץ לבית. המסר הוא כמובן ביפר, שאליו Guido מאזינה.

כדי לשים את הביפר, עלינו להשתמש בפקודה **Beepers** בקובץ העולם. זהו הפורמט של הפקודה:

Beepers <מספר טור> <מספר שורה> <כמות ביפרים>

כעת יש לכם מספיק ניסיון כדי לשחק עם הצהרה זו בעצמכם, ולמקם את הביפר של Syntix במקום שלו, בהתאם לתמונה:



הגדירו פונקציה חדשה בשם **turnaround**, שגורמת ל-Guido להשאר באותה המשבצת, אך להסתובב במקום. כך למשל, אם Guido נמצאת במשבצת בטור 6 ובשורה 4, ופונה לכיוון צפון, הרי שלאחר הרצת הפונקציה **turnaround** היא תהיה עדיין במשבצת בטור 6 ובשורה 4, אך תפנה לכיוון דרום.

באמצעות הפונקציה הזו, הפונקציה **turnright** שהגדרנו בפרק הקודם ושאר הפקודות המוכרות – עזרו ל-Guido לצאת מהבית, לקחת את הביפר, ולחזור חזרה למקומה. שימו לב שבסוף הריצה, Guido צריכה לפנות לכיוון צפון, ממש כמו בתחילה.

תרגיל 2.3 – היום יום הולדת! (תרגיל מתקדם – בונוס)

גם לרובוטים יש יום הולדת, ו-Guido חוגגת את יום הולדתה ה-17. מזל טוב!

Guido מאוד שמחה, ורוצה לציין את יום הולדתה בדרך מיוחדת. מכיוון שהיא לא יכולה לדבר, היא מחליטה לרשום את המספר 17 באמצעות ביפרים. עם זאת, Guido היא רובוטית, והיא סופרת בבסיס בינארי[8]. בבסיס זה, המספר 17 מיוצג בצורה הבאה: 10001.

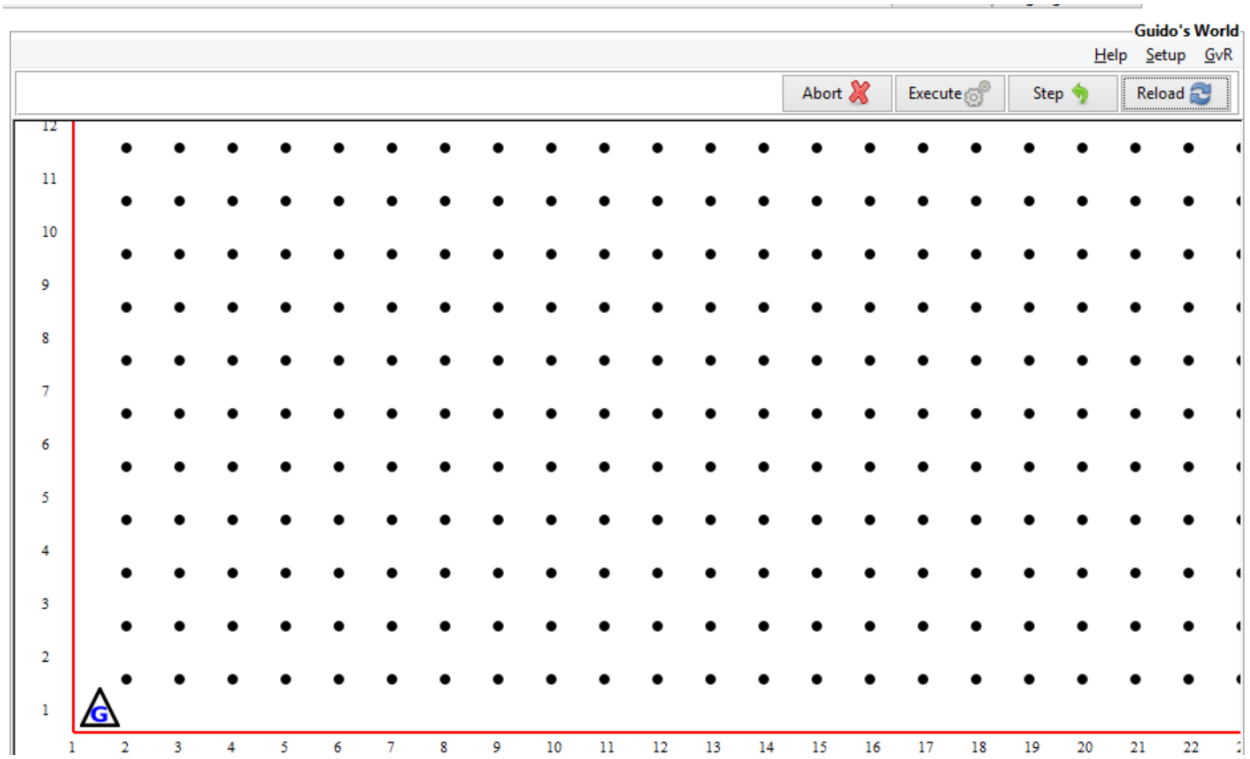
[8] אל דאגה אם אינכם יודעים מה זה בסיס בינארי. כרגע, כל שעליכם לדעת הוא שניתן להציג כל מספר באמצעות הספרות "0" ו-"1" בלבד. מעוניינים לדעת יותר? אתם מוזמנים מאוד לקרוא על כך בספר הלימוד "ארגון המחשב ושפת סף" מאת ברק גונן.

על מנת לפתור את התרגיל, צרו שתי פונקציות חדשות:

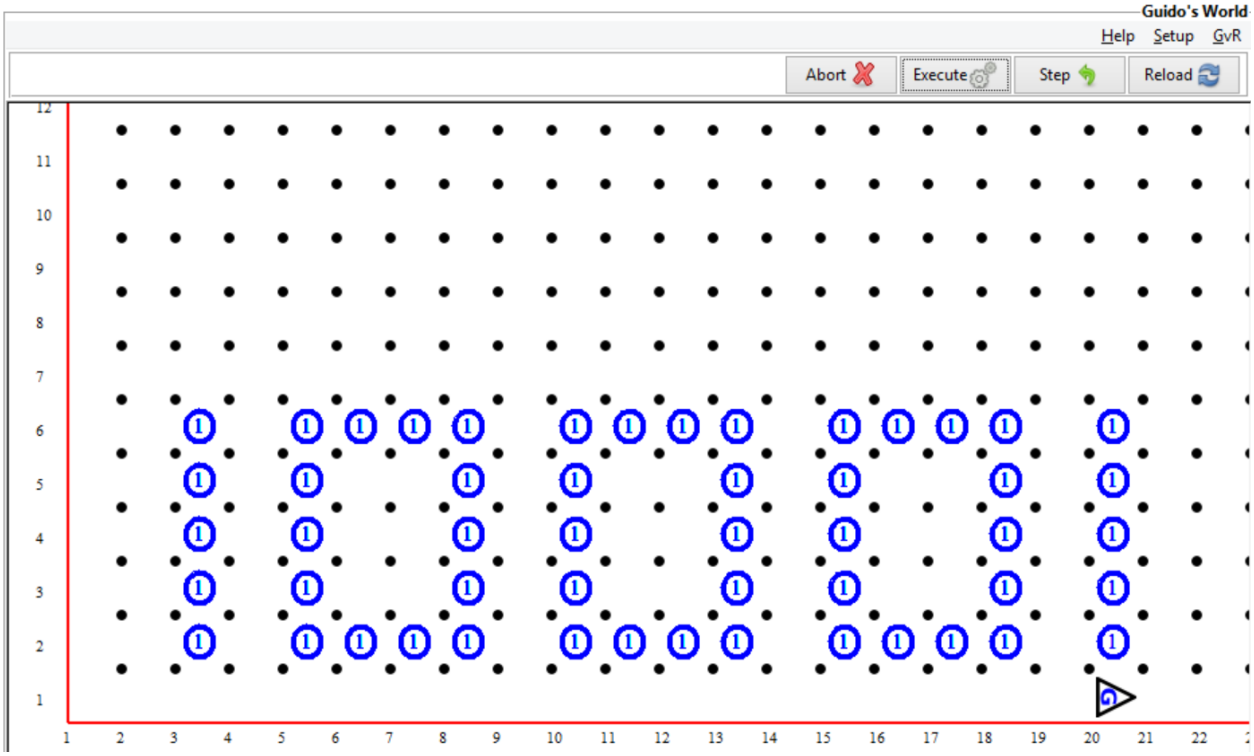
- **drawzero** – הפונקציה תצייר את הספרה "0" באמצעות ביפרים.
- **drawone** – הפונקציה תצייר את הספרה "1" באמצעות ביפרים.

כל פונקציה צריכה לצייר את הספרה המתאימה, וגם להותיר את Guido במצב המתאים כדי לצייר את הספרה הבאה.

בתחילת הריצה של התכנית, על המסך של העולם להיראות בדיוק כך:



לאחר הדפסת המספר, על העולם להראות בדיוק כך:



תנאים ועצמאות

ל-Guido יש המון רעיונות משלה אותם הייתה רוצה לממש, בניגוד למצב בו היא מקבלת פקודות ברורות שאומרות לה בדיוק מה לעשות בכל שלב ושלב.

Guido למדה במהלך שנות חייה לחוש את העולם שסביבה. היא יודעת לגלות האם יש מולה קיר, למשל. היא יודעת להבין אם היא נמצאת ליד ביפר, אם היא פונה לכיוון צפון וכך הלאה. בשלב זה של חייה, הגיעה הזמן שהיא תוכל לנצל את המידע הזה.

ביטוי בוליאני הוא ביטוי בעל הערך אמת (true) או שקר (false).

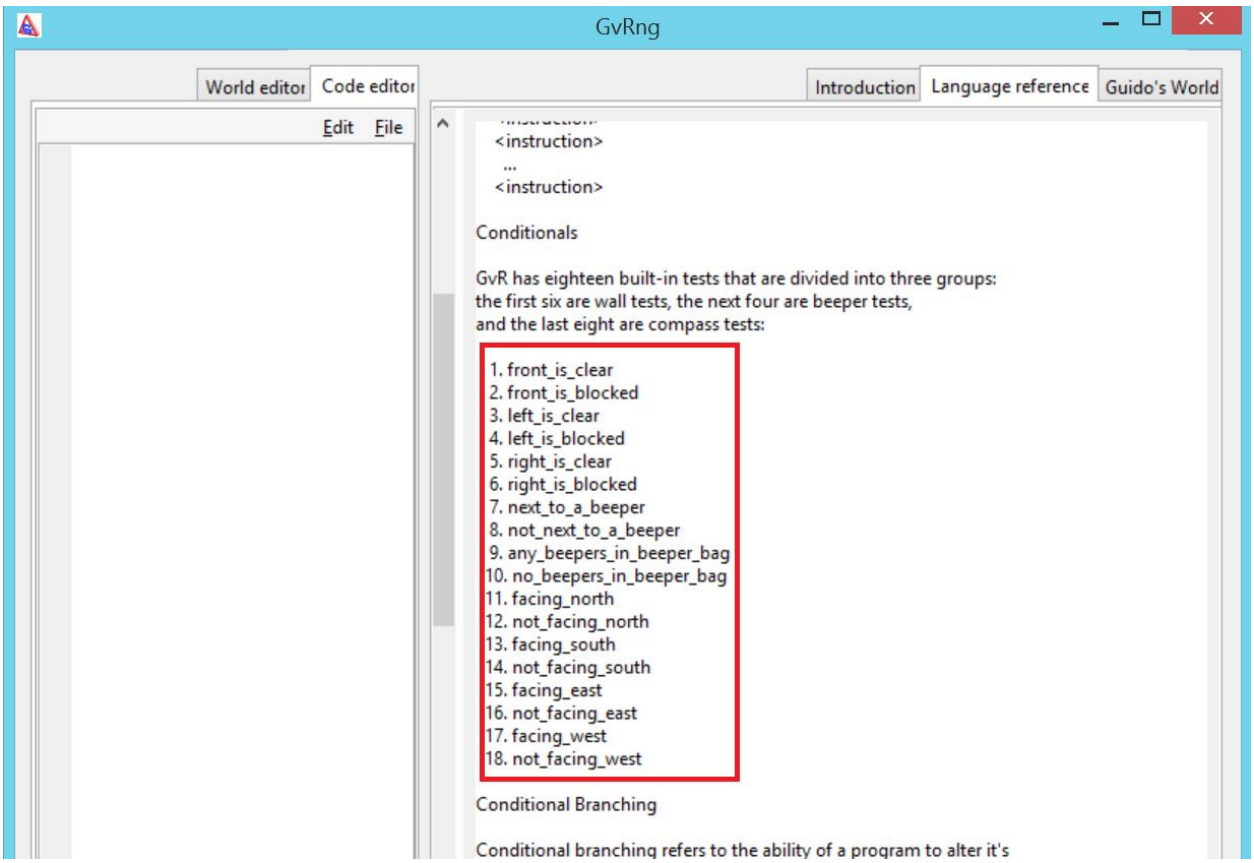
Guido יכולה לבצע כל מיני פעולות, בהינתן הערך של ביטוי בוליאני כלשהו. לדוגמה, Guido יכולה לבדוק אם היא ליד ביפר. אם כן (הערך של הביטוי הוא **true**) – היא תרים את הביפר הזה. לשם כך, נשתמש ב**תנאי** מסוג **if**:

```
if next_to_a_beeper:
    pickbeeper
```

נשים לב לניסוח של התנאי. התחלנו במילה **if**, שמגדירה כי מדובר בתנאי. **next_to_a_beeper** הוא ביטוי בוליאני: הערך שלו יהיה **true** במידה ש-Guido תהיה ליד ביפר, ו-**false** אחרת. לאחר מכן, באמצעות

אינדנטציה, מוגדרות שורות הקוד שיתרחשו אם התנאי החזיר **true**. בדוגמה הזו – רק במידה ש-Guido נמצאת ליד ביפר, כלומר הערך של הביטוי **next_to_a_beeper** הוא **true**, תרוץ הפקודה **pickbeeper**. שימוש זה ב**אינדנטציה** זהה לשימוש שראינו בפונקציות.

בשלב זה, חשוב שנכיר דבר נוסף. בתוכנת GvR, ישנה לשונית בשם **Language reference**. היא כוללת פרטים על שפת התכנות של Guido. בין השאר, יש בה רשימה בשם **Conditionals** – תנאים:



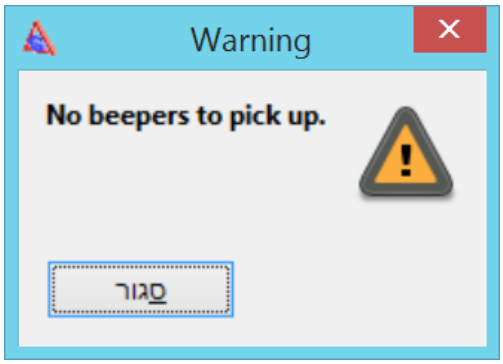
ניתן לראות כי Guido מכירה 18 תנאים שונים – שיכולים לאפשר לה להגיב לסביבה משתנה. לא יאמן כמה מהר הרובוטית הזו גדלה!

תרגיל 2.4 – איסוף בקבוקים

ראשית נעשה ניסיון קצר. נשים את Guido בעולם ללא ביפרים, ונבקש ממנה להרים ביפר באמצעות הפקודה:

pickbeeper

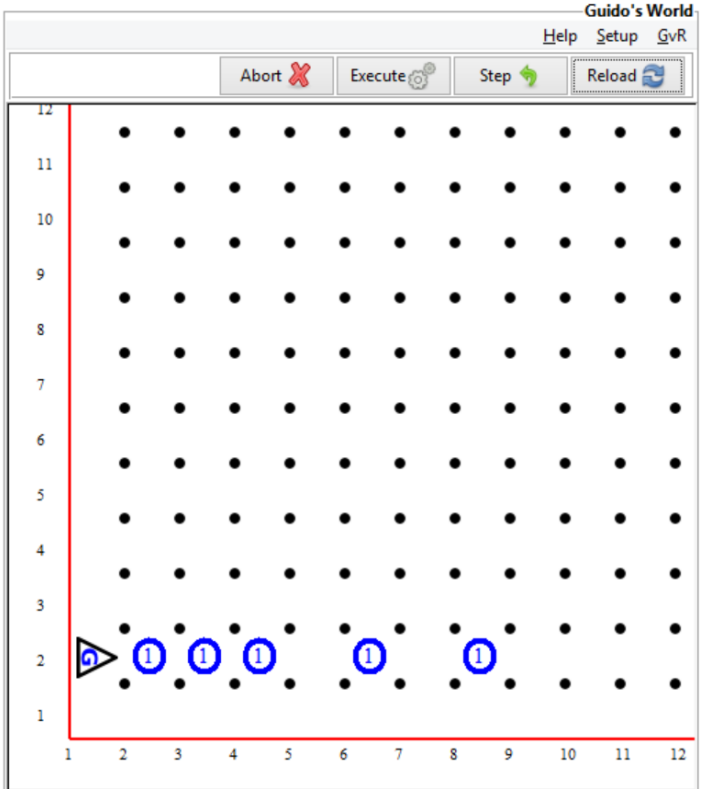
מיד תתקבל הודעת שגיאה:



ושאר הקוד שלנו לא ירוץ כמתוכנן.

Guido מאמינה במחזור. בעולם שלה, היא ממחזרת ביפרים. היום היא החליטה לצאת ולמחזר את **כל** הביפרים שנמצאים בשורה השניה.

צרו עולם שבו Guido נמצאת בתחילת השורה השנייה, ופונה מזרחה. לאורך השורה יהיו ביפרים, אך לא בכל השורה. להלן דוגמה לעולם כזה:



גרמו ל-Guido ללכת עד למשבצת בטור 11, בשורה 2. Guido צריכה לאסוף בהצלחה את כל הביפרים לצרכי מחזור. השתמשו בתנאי **if** על מנת לבדוק האם נמצא ביפר בכל נקודה. אין להסתמך על כך ש-Guido יודעת מראש באיזו משבצת יש ביפר ובאיזו לא.

תנאי ה-**if** גורם ל-Guido לבחון את המצב בעולם ולקבל החלטה לפיו. אם יש כרגע ביפר – הרימי אותו.

לולאות

בכל התרגילים האחרונים, יצא שעשינו משהו מוזר. כתבנו את אותן שורות קוד שוב ושוב. לקחנו שורות קוד, השתמשנו בהעתק, ולאחר מכן בהדבק. זה גרם לקוד שלנו להיות פחות **קריא**, יותר ארוך ומסורבל. בתרגיל האחרון, למשל, השתמשנו מספר פעמים בתנאי **if**, וכן מספר רב של פעמים בפונקציות **move** ו-**pickbeeper**. אולי אף הגדלנו לעשות, והגדרנו פונקציה שנעה קדימה, בודקת האם יש ביפר במשצת הנוכחית, ואם כן אוספת אותו. במקרה כזה, קראנו לאותה הפונקציה שוב ושוב ושוב.

על מנת להריץ את אותן שורות קוד יותר מפעם אחת, נשתמש ב**לולאות**. כדי לכתוב **לולאה**, Guido מכירה את הפקודה **do**. נאמר שאנו מעוניינים לנוע קדימה חמש משבצות בדיוק. במקום לכתוב:

```
move
move
move
move
move
```

נוכל לרשום:

```
do 5:
    move
```

אכן קריא ונוח יותר. שימו לב ששוב השתמשנו באינדנטציה. בדומה לפעמים הקודמות, נוכל לכלול מספר פעולות בתוך האזור הפנימי. כשדיברנו על פונקציות, קראנו לו גוף הפונקציה. כעת נקרא לו **גוף הלולאה**. לדוגמה, נוכל לגרום ל-Guido להניח ביפר ולנוע למשבצת הבאה, חמש פעמים:

```
do 5:
    putbeeper
    move
```

כמובן, רק שורות שנמצאות בגוף הלולאה יקראו שוב ושוב. אם נכתוב את אותן הפקודות בצורה הבאה:

```
do 5:
    putbeeper
move
```

הפקודה **putbeeper** אכן תקרא 5 פעמים, אך הפקודה move תקרא ותפעל פעם אחת בלבד.

חשוב לשים לב לאינדנטציה.

תרגיל 2.5 – איסוף בקבוקים חכם

עכשיו ש-Guido שמעה על הפקודה **do**, היא מאוד מתרגשת ומחכה לפעול באופן חכם יותר. שכתבו את תרגיל 2.3, כך ש-Guido תבצע את אותן הפעולות בדיוק – כלומר תאסוף את כל הביפרים שבשורה השנייה. עם זאת, עשו זאת הפעם ב**לולאה**, והמנעו משכפול קוד מיותר.

באמצעות תרגיל זה, וגם מוקדם יותר בתכנית הלימודים, הבנו למעשה נקודה חשובה:

ישנן דרכים רבות לכתוב קוד שמבצע את אותה הפעולה.

ניתן לגרום ל-Guido לאסוף בקבוקים באמצעות רישום של תנאי **if** ופקודת **move** ו-**pickbeeper** בלבד. ניתן היה לשפר את הקוד, ולהשתמש בפונקציה. כעת למדנו שניתן היה לשפר את הקוד אף יותר, ולכתוב לולאה. בעקבות כתיבת הלולאה, יכולנו לפתור את תרגיל 2.4 באמצעות תכנית תקינה באורך 5 שורות בלבד.

בעתיד, נלמד על תכנות נכון, ונבין שהעובדה שקוד שכתבנו עובד בתנאים מסויימים כמו שרצינו, אינה אומרת בהכרח שהוא קוד טוב. בשלב זה, חשוב שנזכור שאפשר לכתוב את הקוד שלנו ביותר מדרך אחת, וכדאי לשקול מה הדרך הטובה ביותר. לא כל הדרכים טובות במידה שווה.

תרגיל מודרך 2.6 – תנועה במרובע

בתרגיל זה נבחן דרכים שונות לכתוב קוד, ונחדד מספר נקודות. נתחיל בליצור עולם שבו Guido ממוקמת במשבצת שבטור החמישי, בשורה הראשונה, כשהיא פונה מזרחה:

```
Robot 5 1 E
```

מטרתינו היא לגרום ל-Guido לנוע שני צעדים קדימה, להסתובב שמאלה, לנוע שישה צעדים, להסתובב שמאלה, לנוע שני צעדים, להסתובב שמאלה, לנוע שישה צעדים, ושוב להסתובב שמאלה. בסוף פעולת התכנית, Guido תהיה בדיוק באותה משבצת, כשהיא פונה מזרחה. במהלך התכנית, היא תנוע בצורה של מלבן בגודל 2*6.

את הדרך הטריוויאלית ביותר לבצע זאת, על ידי שימוש בפקודות **move** ו-**turnleft** בזו אחר זו, אנו מכירים, ולא נתעכב על פתרון זה. שיפור ראשון לדרך הטריוואלית הזו, יכול להיות לקבץ כל שתי פקודות **move** סמוכות לפונקציה אחת:

```
define two_steps:
    move
    move
```

כעת התכנית המלאה שלנו צפויה להראות כך:

```
define two_steps:
        move
        move

two_steps
turnleft
two_steps
two_steps
two_steps
turnleft
two_steps
turnleft
two_steps
two_steps
two_steps
turnleft

turnoff
```

ברור למדי כי יש כאן שכפול קוד. נוכל לעשות הרבה דברים כדי לשפר זאת. בתור התחלה, ניתן לראות כי אנו חוזרים פעמיים על ארבע שורות הקוד הבאות:

```
two_steps
two_steps
two_steps
turnleft
```

נוכל לאגד אותן תחת פונקציה אחת, בצורה הבאה:

```
define six_steps_and_turn:
        two_steps
        two_steps
        two_steps
        turnleft
```

שימו לב שכאן הגדרנו פונקציה משלנו, שקוראת לפונקציה אחרת שהגדרנו קודם לכן. דבר זה אפשרי ואף רצוי. הקוד שלנו כבר נראה קצר יותר, וגם קריא יותר. אך למעשה, יכולנו להפוך את הפונקציה six_steps_and_turn לקצרה וברורה אף יותר, על ידי שימוש בלולאה:

```
define six_steps_and_turn:
    do 3:
        two_steps
    turnleft
```

כאן הגדרנו פונקציה. בתוך הפונקציה השתמשנו בלולאה, ובלולאה קראנו לפונקציה אחרת שהגדרנו קודם לכן. למעשה, אנחנו כבר "משחקים" בהרבה מרכיבים שונים שלמדנו – פקודות, פונקציות ולואות, בכדי לשפר את הקוד שלנו. ראו כמה קצר וקריא יותר הוא נראה:

```
define two_steps:
    move
    move

define six_steps_and_turn:
    do 3:
        two_steps
    turnleft

two_steps
turnleft
six_steps_and_turn
two_steps
turnleft
six_steps_and_turn

turnoff
```

כעת, ניתן גם לשפר את הקוד הראשי שלנו, שכן הוא חוזר פעמיים על אותן שלוש השורות. נעשה זאת באמצעות **לולאה**, ונקבל את הקוד היפה הבא:

```
define two_steps:
    move
    move
```

```
define six_steps_and_turn:
    do 3:
        two_steps
    turnleft

do 2:
    two_steps
    turnleft
    six_steps_and_turn

turnoff
```

קצר, ברור ועובד.

נשים לב שיכולנו גם להשתמש בלולאה בפונקציה **two_steps**, אך במקרה זה הקוד לא היה הופך קצר יותר, וגם לא קריא יותר, ולכן החלטנו לא לעשות זאת. החלטות כאלו מתקבלות על סמך ניסיון שנצבר, ועם הזמן נלמד לקבל אותן באופן מושכל.

תרגיל 2.7 – כל הרובוטים קופצים רוקדים, כל הרובוטים קופצים רוקדים

גם רובוטים אוהבים לרקוד, ו-Guido חברה בלהקה יחד עם חברה Syntix. בקרוב מתקיימת תחרות ריקודים בינלאומית, ו-Guido רוצה להוביל את הלהקה בתחרות עם ריקוד חדש – ריקוד הסינטקרוניקס. אלו צעדי הריקוד:

1) מנקודת ההתחלה, צועדים שני צעדים קדימה.
2) מסתובבים במקום (כלומר – נשארים באותה המשבצת, אך פונים לכיוון ההפוך), וצועדים שלושה צעדים.
3) שלוש פעמים מבצעים את הפעולה הבאה: פונים ימינה, וצועדים קדימה.

לאחר הצעדים האלו, Guido תחזור אל נקודת ההתחלה, אך תפנה לכיוון אחר מהכיוון אליו פנתה בתחילת התכנית. הריצו את צעדי הריקוד (שלבים 1 עד 3 שתוארו לעיל) ארבע פעמים, כך ש-Guido תחזור אל נקודת ההתחלה, תפנה אל הכיוון אליו פנתה מלכתחילה, ותוכל להיות גאה בעצמה. וגם בכם, כמובן!

על מנת לממש את התרגיל, תוכלו לכתוב פונקציות ללא הגבלה, להשתמש בלולאות, תנאים וכל הדברים שלמדנו עד כה. נסו לכתוב את הקוד הקצר והקריא ביותר.

עצה: אל תכתבו את כל הקוד ורק אז תנסו להריץ אותו. נסו לכתוב בכל פעם רק חלק קטן מהקוד, להריץ ולבדוק שהוא עובד כפי שתכוננתם. כך למשל, התחילו בכך ש-Guido תתקדם שני צעדים ואז תסתובב. אחר כך, תדאגו שהיא גם תצעד שלושה צעדים. רק לאחר מכן, המשיכו הלאה.

סיימתם? נהדר. קראו שוב את הקוד שלכם. האם זו הדרך הקצרה והקריאה ביותר? אם לא, נסו לשפר אותו.

ומה אם לא?

מוקדם יותר בפרק זה, עסקנו ב**תנאים**, ופגשנו את תנאי **if**. השתמשנו בו בכדי לקבל החלטה במהלך הריצה, למשל: אם יש ביפר במשבצת הזו, נרים אותו. בעולם האמיתי, לא כל ההחלטות כל כך פשוטות, ולעתים צריך לחשוב מעט יותר לפני שבוחרים מה לעשות.

מידי שבוע, Guido הולכת לבקר את חברה Syntix, שאוהב לאפות. זה מסביר את החברות המעולה. בדרך כלל, Syntix מכין את עוגת הביפרים המפורסמת שלו[9]. עם זאת, עקב ההכנות לחתונה, Syntix עסוק בזמן האחרון, ולא תמיד מכין את העוגה האהובה. במעט הזמן הפנוי שנותר לו, הוא מכינה RoboCookies.

Guido לא מעוניינת ללחוץ על Syntix, ורוצה לבקש ממנו מנה מעוגת הביפרים המופלאה, אך רק אם Syntix הכין כזו. אם לא, Guido תסתפק ב-RoboCookies. לשם כך, נכיר סוג חדש של תנאי: **if… else…**. תנאי כזה נראה כך:

```
if <condition>:
        <instruction>
else:
        <other instruction>
```

כלומר, אם (**if**) התנאי <condition> מתקיים, Guido תבצע פעולה מסויימת. אחרת (**else**), היא תבצע פעולה אחרת. להלן דוגמה לשימוש בתנאי מסוג זה:

```
if next_to_a_beeper:
        pickbeeper
        move
else:
        turnleft
        move
```

תרגיל 2.8 – הליכה בין החומות

ראשית, ניצור עולם שבו Guido מוקפת חומה מכל צדדיה. לשם כך, נשתמש בפקודות הבאות ליצירת קובץ העולם:

[9] המתכון די דומה לעוגת שזיפים, אבל במקום שזיף – משתמשים בביפר. אל תנסו את זה בבית.

```
Robot 1 1 E
Wall 1 9 N 9
Wall 9 1 E 9
Beepers 1 9 1
Beepers 9 1 1
Beepers 9 9 1
```

Guido לא יודעת מה הגודל של העולם הזה – כלומר, היא לא יודעת עד כמה החומות רחוקות ממנה. אולם, Guido יודעת שבכל "פינה" – יש ביפר, ואז עליה לפנות שמאלה.

עזרו ל-Guido לקחת את שלושת הביפרים ולחזור לנקודת ההתחלה שלה, מבלי לפגוע בחומה. הקוד שלכם צריך לאפשר ל-Guido לעשות זאת ב-32 צעדים (פעולות **move**) בדיוק. השתמשו בלולאת **do** 32 פעמים, ובתוכה השתמשו בתנאים **if… else**.

תרגיל 2.9 – הליכה קצרה בין החומות

טוב, ההליכה בתרגיל זה לא תהיה קצרה יותר, אבל הקוד שלנו כן. פתרו שוב את תרגיל 2.7, אך הפעם על ידי שימוש בתנאי **if** בלבד (ללא **else**).

שאלת מחשבה: איזה קוד נראה קריא וברור יותר? האם זה שנכתב בתשובה לתרגיל 2.7, או זה שנכתב בתשובה לתרגיל 2.8?

לולאות חכמות

עד כה הכרנו לולאות **do**. הן עזרו לנו מאוד – בתרגיל הקודם הצלחנו להריץ את אותן שורות קוד 32 פעמים, וכתבנו אותן רק פעם אחת. הן הפכו את הקוד שלנו לברור יותר וקצר יותר. אבל לולאות **do** הן מאוד מוגבלות: עלינו לדעת **מראש** כמה פעמים הן צריכות לרוץ. בתרגיל הקודם, למשל, ידענו שהלולאה צריכה לרוץ 32 פעמים. מה היה קורה לו היינו משנים מעט את העולם, ושמים מספר שונה של ביפרים בכל פינה?
לצורך הדוגמה, נבחר את קובץ העולם הבא:

```
Robot 1 1 E
Wall 1 9 N 9
Wall 9 1 E 9
Beepers 1 9 1
Beepers 9 1 3
Beepers 9 9 4
```

בפינה אחת של העולם יש שלושה ביפרים, בשניה יש ארבעה ובשלישית יש אחד בלבד, אך Guido לא יודעת זאת מראש. כיצד נכתוב בצורה נוחה לולאה שתאפשר לנו לאסוף את כל הביפרים האלו?

לשם כך, נצטרך לולאות חכמות – לולאות ש"יודעות" להתאים את עצמן למצב – ולרוץ רק במידה שתנאי מסויים מתקיים. סוג אחד של לולאות כאלו נקרא לולאות **while**. באופן כללי, לולאה כזו נראית כך:

```
while <condition>:
    <instruction>
```

כלומר, כל עוד התנאי **<condition>** מתקיים, הפעולה **<instruction>** תרוץ. ברגע שהתנאי **<condition>** לא יתקיים, הלולאה תפסיק את הריצה שלה לאלתר. כמובן שניתן לרשום יותר מפקודה אחת בגוף הלולאה. למעשה, לולאת **while** דומה מאוד ללולאת **do**. ההבדל היחיד ביניהן הוא שלולאת **do** מקבלת מספר קבוע של פעמים שעליה לרוץ, ולולאת **while** מקבלת תנאי ורצה כל עוד התנאי הזה מתקיים.

לדוגמה, הלולאה הבאה אוספת את כל הביפרים שנמצאים במשבצת מסויימת:

```
while next_to_a_beeper:
    pickbeeper
```

נשים לב שברגע שלא יהיו עוד ביפרים במשבצת, מכיוון ש-Guido אספה את כולם, התנאי **next_to_a_beeper** לא יתקיים (כלומר יהיה בעל הערך **false**). בשלב זה הלולאה תפסיק לרוץ, והתכנית תתקדם לפקודה הבאה.

לולאות **while** משלבות למעשה שני קונספטים תכנותיים שהכרנו – תנאים ולולאות. זהו כלי חזק מאוד, שמאפשר לנו לכתוב קוד מעניין יותר.

תרגיל מודרך 2.10 – איסוף ביפרים בין החומות

נשתמש בקובץ העולם שהצגנו בדוגמה האחרונה, ונכתוב קוד שיאפשר ל-Guido לאסוף את כל הביפרים.
ראשית, נחשוב על הקוד הראשי. באופן כללי, נרצה ש-Guido תתקדם במרחב כל עוד אין ביפר. אם יש ביפר, עליה לאסוף אותו ואת כל שאר הביפרים במשבצת הזו, ולפנות שמאלה. זה נשמע כמו תנאי **if** פשוט. כמו כן, אנו יודעים שתנאי זה צריך לרוץ 32 פעמים בדיוק:

```
do 32:
    if next_to_a_beeper:
        pick_all_beepers
        turnleft
    move

turnoff
```

מצויין, כל שנותר לנו הוא לכתוב את הפונקציה **pick_all_beepers**.

שימו לב לדרך המחשבה ותהליך העבודה שלנו.

אנו מתחילים מתאור המקרה הכללי – מה Guido צריכה לעשות. כשאנו רואים פעולה מעט מסובכת, כגון **pick_all_beepers**, אנחנו כותבים אותה בתור פונקציה. בהמשך – נממש את הפונקציה הזו. בינתיים, רק נזכור שהיא שם.

גם כאן, ניתן לראות עד כמה השימוש בפונקציות עוזר לנו להפוך את הקוד לקריא יותר. הקוד לעיל מאוד ברור, וקל להבין מה הוא עושה.

כעת נותר לממש את **pick_all_beepers**. נעשה זאת:

```
define pick_all_beepers:
    while next_to_a_beeper:
        pickbeeper
```

הקוד המלא שלנו כולל הרבה קונספטים שונים שפגשנו לאורך פרק זה והפרק הקודם – פונקציה, תנאי **if**, לולאת **do** ולולאת **while**.

כמו שאמרנו, לולאת **while** היא כלי חזק. אבל, כמו שאמר בנג'מין פארקר[10], "עם כוח גדול באה אחריות גדולה". חשוב לשים לב לכמה פרטים כאשר אנו כותבים לולאות **while**.

שאלת מחשבה: מה עושה הלולאה הבאה? מה יקרה?

```
while not_next_to_a_beeper:
        move
        turnleft
        turnleft
```

ובכן, במידה שאין ביפר במשבצת הנוכחית עליה Guido עומדת או במשבצת הסמוכה אליה (בצד שאליו Guido פונה), הלולאה הזו היא **לולאה אינסופית**, כלומר – היא תמשיך לרוץ לנצח. נלחץ על **Execute**, ונצפה בריצת הלולאה.

למזלנו, כשמתכנתים ל-Guido יש גם את כפתור **Abort** שמאפשר לנו להפסיק את ריצת התכנית. נלחץ עליו בטרם הרגליים של Guido יתחילו לכאוב. מן הסתם, חשוב לשים לב שלא לגרום ללולאה אינסופית בקוד שלנו. בדוגמה זו, קל יחסית לראות שמקרה זה אפשרי, אך בלולאות מורכבות וארוכות יותר – העניינים עלולים להסתבך.

תרגיל 2.11 – הליכה בין החומות, ובלי שום ביפר!

בתרגיל זה Guido נמצאת בעולם זהה לעולם בו הייתה בתרגיל 2.9, למעט פרט אחד חשוב: בעולם זה, אין ביפרים. נשתמש בקובץ הבא כדי לתאר את העולם:

```
Robot 1 1 E
Wall 1 9 N 9
Wall 9 1 E 9
```

[10] הידוע יותר בתור "דוד בן", או פשוט – "הדוד של ספיידרמן".

כעת, גרמו ל-Guido לבצע את אותו מסלול ההליכה שעשתה בתרגיל 2.9, כלומר "טיול" לאורך החומות, עד שתחזור אל נקודת ההתחלה. עשו זאת באמצעות לולאת **while**.

רמז1: השתמשו בלשונית **Language reference**.

רמז2: הפתרון שלכם יכול להיות באורך 5 שורות בלבד (השורה החמישית היא פקודת **turnoff**).

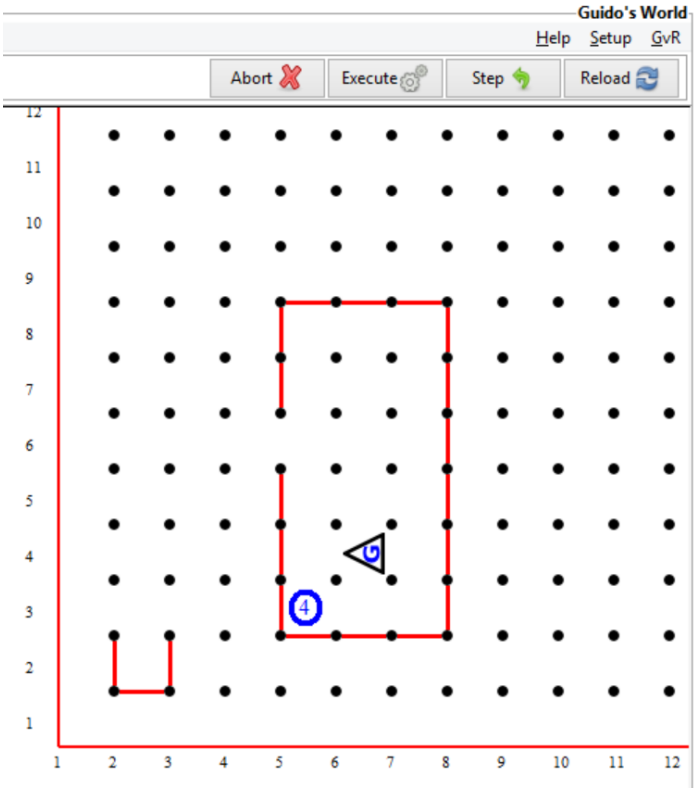
תרגיל 2.12 – מי זאת רומבה?

Guido שמעה כמה בני אדם מדברים על רומבה, ומאוד נעלבה. מי צריך את הדבר הזה? היא יודעת לנקות מצויין. היא החליטה להוכיח את יכולותיה, ולהוציא את הזבל.

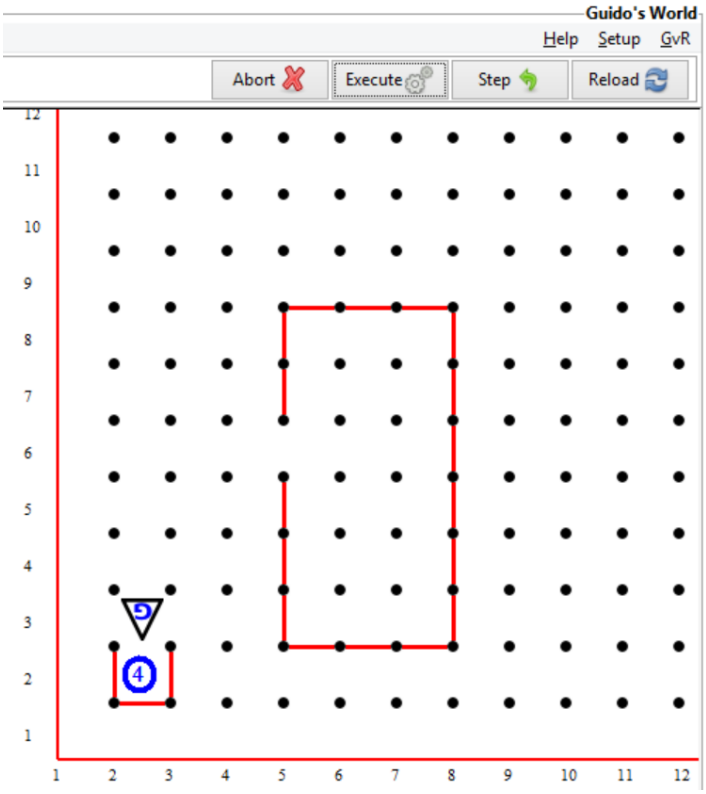
Guido נמצאת בבית, בתוך עולם המתואר כך:

```
Robot 6 4 W
Beepers 5 3 4
Wall 2 2 E
Wall 2 2 S
Wall 2 2 W
Wall 5 2 N 3
Wall 8 3 W 6
Wall 5 8 N 3
Wall 5 3 W 3
Wall 5 7 W 2
```

Guido יודעת שהזבל (ערימה של ביפרים) נמצא בפינה השמאלית התחתונה של הבית, אבל היא לא יודעת עד כמה המצב גרוע – כלומר כמה ביפרים יש בערימה.



כתבו קוד שיגרום ל-Guido לאסוף את כל הביפרים שבערימה, לצאת מן הבית ולשים אותם בפח הזבל. לאחר ש-Guido תסיים, עליה לצעוד צעד אחד לאחור, להביט בפח ולוודא שהזבל אכן שם. כך אמור להיראות העולם בסוף ריצת התכנית:



השתמשו בכל החומר שלמדנו עד כה – תנאים, לולאות ופונקציות – בכדי לכתוב את התכנית שלכם.

פרק 2 – סיכום

בפרק זה למדנו המון דברים חדשים. למדנו איך ליצור עולמות מורכבים יותר, כדי ש-Guido לא תשתעמם, על ידי שילוב של חומות וביפרים. בנוסף, למדנו שני קונספטים חשובים ביותר בעולם התכנות: **תנאים** ו**לולאות**. הכרנו תנאי **if** כאשר לימדנו את Guido למחזר בקבוקים. היינו צריכים להגיב למציאות משתנה – רק אם היה ביפר במשבצת שעליו Guido עמדה, היא ניסתה לאסוף את הביפר. לאחר מכן, אספנו את הבקבוקים באופן חכם על ידי שימוש בלולאת **do**, ובכך חסכנו הרבה שורות קוד מיותרות.

ראינו גם שיש דרכים שונות לכתוב קוד שעושה בדיוק אותו הדבר. נוכחנו לדעת שפעמים רבות דרך אחת תהיה קריאה, ברורה וקצרה יותר מדרך אחרת, ולכן נעדיף להשתמש בה.

הודות לנימוסים של Guido, שלא רצתה ללחוץ על חברה Syntix, הכרנו גם תנאי **if… else**, המאפשר לנו לבצע פעולה אחת אם תנאי מסויים מתקיים, ופעולה שונה במידה שהתנאי לא מתקיים. לאחר מכן, למדנו על **לולאות חכמות** – כאלו שפועלות רק כאשר תנאי מסויים מתקיים, לולאות **while**. ראינו גם **לולאות אינסופיות**, והבנו שצריך להמנע מהן.

בפרק זה התחלנו לכתוב קוד מעט מורכב יותר. עובדה זו גרמה לנו לחשוב על תהליך העבודה שלנו. אולי כדאי לתכנן קודם את דרך הריצה הכללית של הקוד, ורק לאחר מכן לממש את הפונקציות הפנימיות? אולי עדיף לבדוק חלק קטן, ורק אז להתקדם לתמונה הכללית? אלו דרכי חשיבה שנסגל לעצמנו להמשך הדרך. כמו כן, ראינו שנוח לעתים לעבור שורה-שורה על הקוד במהלך הריצה שלו (באמצעות **Step**), כדי להבין איך הקוד פועל.

בפרק הבא נשתמש בידע שרכשנו בפרק זה על מנת לכתוב תכניות מורכבות ומלאות יותר. נשחק עם כל האלמנטים שהכרנו עד כה – פונקציות, תנאים ולולאות, ונשלב אותם בכדי ליצור דברים מעניינים. לאחר שנתרגל קצת, נהיה מוכנים לפגוש בשפת התכנות המלאה הראשונה שלנו – שפת Python.

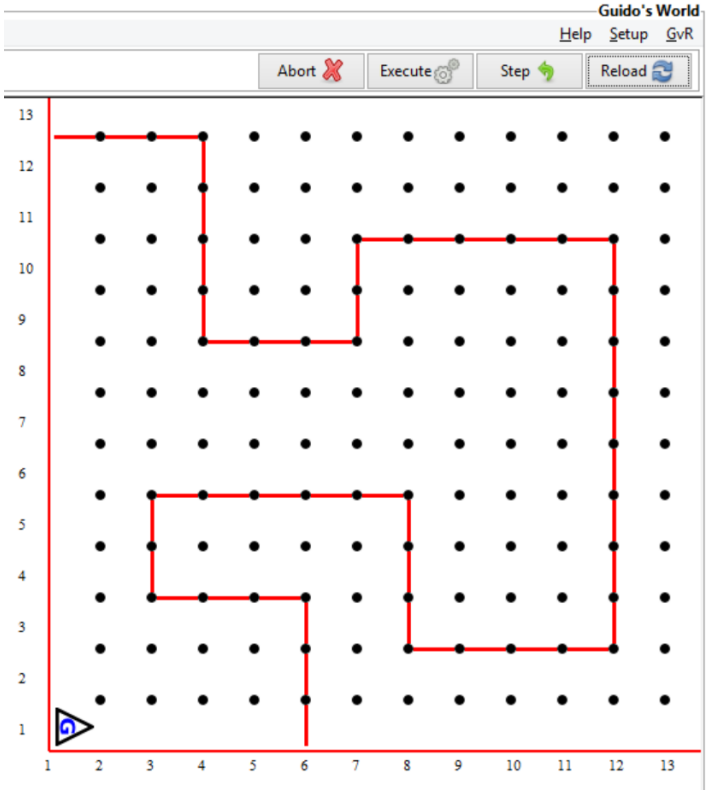
פרק 3 – רובוטים מורכבים

בפרק הקודם הכרנו קונספטים שונים, ודנו בעיקר בלולאות ותנאים. בפרק זה נתרגל את כל הנושאים שהכרנו עד כה – בניית עולמות מורכבים, פונקציות, לולאות, תנאים, ביפרים וחומות – ונכתוב תכניות מעניינות ומורכבות יותר מאשר בפרקים הקודמים. למעשה, בפרק זה לא נלמד נושאים חדשים לגמרי, אלא רק נתרגל נושאים ישנים.

מכיוון שבפרק זה נכתוב תכניות מורכבות יותר, כדאי מאוד שנתכנן את הקוד שלנו בטרם ניגש ישר לכתוב אותו. אפשר לתכנן עם דף ונייר, באמצעות תוכנה לכתיבת טקסט, בציור או בכל דרך שנוחה לכם. תכנון נכון יאפשר לנו לכתוב את הקוד שלנו בקלות.

תרגיל 3.1 - מגלה עולמות

Guido משמת בעולם חדש ורוצה להסתובב בו קצת, כדי לגלות מה יש בו. העולם הזה מורכב יותר מעולמות שראינו עד כה, והוא בצורה של פוליגון – מצולע סגור. הוא נראה כך:



נשתמש בשורות הבאות כדי לתאר את העולם הזה:

```
Robot 1 1 E 1
Wall 5 1 E 3
```

Wall 3 3 N 3
Wall 2 4 E 2
Wall 3 5 N 5
Wall 7 3 E 3
Wall 8 2 N 4
Wall 11 3 E 8
Wall 7 10 N 5
Wall 6 9 E 2
Wall 4 8 N 3
Wall 3 9 E 4
Wall 1 12 N 3

Guido מתחילה בפינה הימנית התחתונה של העולם, ומודעת לכך שהיא נמצעת בתוך מצולע סגור. אך מעבר לכך – Guido לא יודעת דבר על צורת העולם: היא לא יודעת מה גודל העולם, כמה צלעות יש בו או כמה פניות היא צריכה לעשות. היא כן יודעת שבעולם לא מונח כרגע אף ביפר.

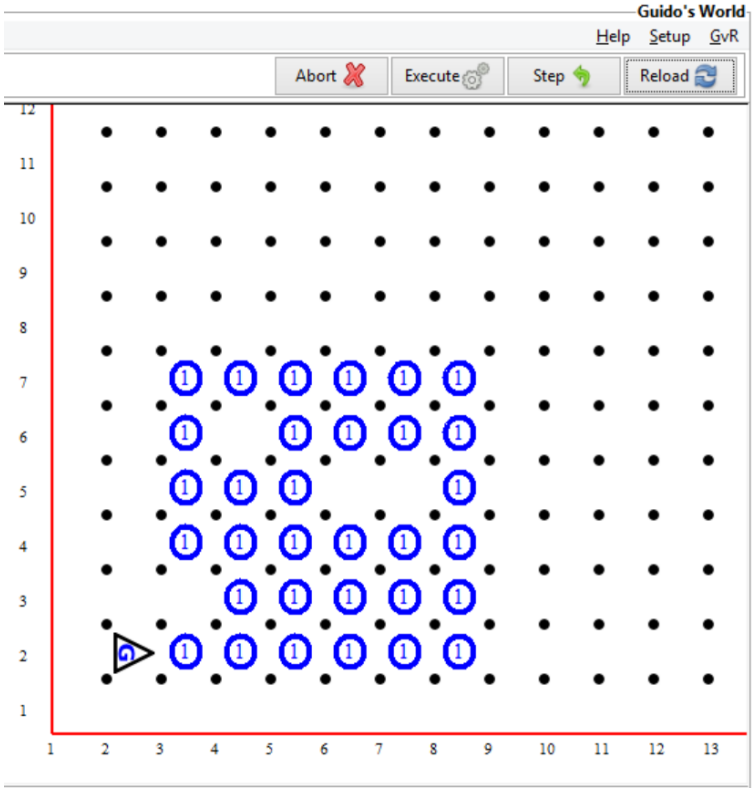
בתרגיל זה עלינו לכתוב קוד שיגרום ל-Guido להקיף את העולם (כלומר ללכת לאורך החומה) פעם אחת בדיוק. מכיוון ש-Guido לא יודעת מה גודל העולם, נצטרך למצוא דרך אחרת בה היא תגלה שהיא חזרה לנקודה ההתחלה. לשם כך, היא תשתמש בביפר – היא תניח ביפר על הרצפה בתחילת התכנית. לאחר מכן, בפעם הראשונה שתיתקל בביפר – היא תדע שהיא חזרה לנקודת ההתחלה, היות שהיא יודעת שאין ביפרים אחרים בעולם.

נשים לב כי היות ש-Guido לא יודעת את גודל העולם, היא לא תוכל להשתמש בלולאת **do** בלבד. עם זאת, לולאת **while** תוכל לעזור, וכך גם תנאי **if… else**. בנוסף, נשים לב כי בעולם שכזה, Guido צריכה להתאים את כיוון הפניה שלה – לעתים היא תפנה שמאלה, ולעתים ימינה.

לאחר שקוד התכנית ירוץ בהצלחה – נסו לשנות את העולם, ולראות שהקוד גורם ל-Guido לסייר בהצלחה גם בעולם החדש שנוצר.

תרגיל 3.2 – עת לקצור

ל Guido-תחביבים רבים, ואחד מהם הוא לעסוק בחקלאות. מחוץ לביתה ישנו שדה קטן, ובעונה המתאימה היא קוצרת את היבול בו. השדה הוא בגודל של 6*6 משבצות, אך לא בכל משבצת בהכרח גדל יבול (ביפר). שדה כזה יכול, למשל, להראות כך:



צרו עבור Guido עולם כזה – עם שדה בגודל 6*6, עם יבול ברוב המשבצות, אך לא בכולן. העולם לא חייב להראות בדיוק כמו בתמונה. כעת, כתבו קוד שיאפשר ל Guido-לקצור את כל היבול. שימו לב כי Guido יודעת מה גודל השדה, אך היא אינה יודעת מראש היכן יש יבול והיכן לא.

הכוונה: על מנת לפתור את התרגיל, מתאים להשתמש בלולאות **מקוננות** – לולאה בתוך לולאה. אחת הלולאות תציין כי יש 6 שורות שצריך לפעול על פיהן. כלומר, לבצע 6 פעמים את אותה פעולה – פעולה שמבצעים על כל שורה. בתוך הלולאה הזו, תתאים לולאה נוספת – שתציין כי יש 6 משבצות בכל שורה. כלומר, היא תגרום לכך ש-6 פעמים תתבצע הפעולה הרלבנטית לכל משבצת בנפרד.

תרגיל 3.3 – לתפוס את כולם!

אחד מתחביביה הרבים של Guido הוא לאסוף ביפרים, ובאוסף המרשים שלה יש ביפרים בשלל צורות וצבעים. למעשה, האוסף כה גדול עד שהוא לא נכנס בביתה של ,Guido והיא מחזיקה אותו במתחם המיועד לו בלבד. ,Guidoחובבת המתמטיקה, שמחה לגלות שדה שלם של ביפרים הכוללים שיעורים על נושאים מתמטיים שונים. השדה בצורת מלבן, והוא מוקף בחומות מכל צידיו. ממש גן עדן! אך בכל זאת, מסתבר שיש מספר בעיות – ראשית, Guido תגיע אל השדה בהסעה, שתניח אותה במקום רנדומלי בשדה. במילים אחרות, Guido לא יכולה לדעת מראש איפה בשדה היא תושם בתחילת פעולת התכנית שלה (באיזו שורה ובאיזה טור), וגם לא לאיזה כיוון היא תפנה. שנית, Guido לא יודעת מראש כמה ביפרים יהיו בשדה, וגם לא היכן הם נמצאים. במשבצת מסויימת יתכן שלא יהיה ביפר כלל, שיהיה ביפר אחד, או שיהיו כמה ביפרים .

על מנת להצליח במשימה הזו, עלינו לכתוב קוד שיגרום ל-Guido לאסוף את **כל** הביפרים, ולהניח אותם באופן מסודר בפינה הצפון-מזרחית של השדה. מאוחר יותר, ההסעה תגיע ותסיע את הביפרים האלו היישר אל האוסף. לאחר מכן, Guido צריכה להגיע אל הפינה הדרום-מערבית של הפארק, אליה תגיע הסעה אחרת שתיקח אותה לביתה. במשבצת זו, Guido צריכה לפנות לכיוון צפון, ואז לכבות.

על מנת לבדוק את הקוד, השתמשו בעולם שמתואר על ידי הקובץ הבא:

```
Robot 7 9 S
Wall 1 15 N 22
Wall 22 1 E 15
Beepers 1 3 2
Beepers 1 12 1
Beepers 2 5 4
Beepers 3 6 1
Beepers 3 7 2
Beepers 3 8 3
Beepers 4 1 1
Beepers 4 14 1
Beepers 5 10 5
Beepers 7 3 1
Beepers 7 8 1
Beepers 7 9 1
Beepers 7 10 1
Beepers 8 4 1
```

Beepers 9 13 2
Beepers 10 3 3
Beepers 12 11 1
Beepers 12 12 1
Beepers 13 12 1
Beepers 13 13 1
Beepers 14 3 1
Beepers 15 2 1
Beepers 15 3 2
Beepers 15 4 1
Beepers 16 3 1
Beepers 17 15 5
Beepers 18 8 10
Beepers 20 11 2
Beepers 21 1 3
Beepers 22 11 2

לאחר מכן, שנו את קובץ העולם. ביחרו מספר שונה ומיקום שונה של ביפרים, וכן גודל שונה של השדה. האם הקוד עדיין עבד בהצלחה?

פרק 3 – סיכום

בפרק זה כמעט שלא למדנו תוכן חדש. בעיקר, השתמשנו בידע שרכשנו בשני הפרקים הקודמים על מנת לכתוב תכניות מורכבות ומלאות יותר. שיחקנו עם כל האלמנטים שהכרנו עד כה – **פונקציות, תנאים ולולאות**, ושילבנו אותם בכדי ליצור דברים מעניינים. בהתחלה, עזרנו ל-Guido לגלות את העולם שבו היא נמצאת – וללכת בו גם כשהוא לא בנוי בצורת מלבן סטנדרטי. לאחר מכן, השתמשנו ב**לולאות מקוננות** כדי לקצור שדה של חיטה, ולבסוף הצלחנו להרחיב את אוסף הביפרים של Guido על ידי כתיבת תוכנית שסורקת מפת עולם לא ידוע, בו Guido מתחילה ממקום לא ידוע.

למדנו הרבה בשלושת הפרקים האחרונים. אנחנו יכולים להיות גאים בעצמנו! התחלנו ממצב בו איננו מכירים אף שפת תכנות, ולא בטוחים כיצד תכניות מחשב פועלות. מהר מאוד הגענו למצב שבו אנו בונים תכניות מעניינות – שכוללות תנאים, פונקציות ולולאות. הכרנו למעשה את יסודות התכנות, ודיברנו גם על **תכנות נכון**.

בנקודה זו, הגיע הזמן שנבחין שמשהו חסר. בתכניות האחרות שלנו נתקלנו מספר פעמים באתגרים – איך לדעת היכן Guido התחילה? האם להניח ביפר זו באמת הדרך היעילה ביותר? כיצד Guido תדע כמה משבצות היא זזה לאורך המסך? הבעיות האלו נובעות מעניין מהותי – ל-Guido אין זיכרון. כלומר, Guido יכולה לבצע פקודות שהגדרנו לה, אך היא לא יכולה באמת לזכור דברים לאורך זמן. כתוצאה מכך, היא לא יכולה לספור. היא כבר התרגלה למצב הזה, ולמדה לחיות איתו בשלום, אך היא יודעת שלו היה לה זיכרון – ניתן היה לכתוב לה תכניות הרבה יותר מורכבות.

Guido לא עצובה. היא יודעת שהיא כאן בכדי לעזור לאנשים ללמוד – להבין איך לתכנת, להכיר את הקונספטים של פונקציות, לולאות ותנאים. זו משימה חשובה, והיא גאה בהצלחה שלה. עם זאת, הגיע הזמן שנעבור הלאה. בפרקים הבאים נכיר את שפת התכנות Python – שפה חזקה שאפשר לעשות באמצעותה דברים רבים, כמו לשמור נתונים בזיכרון, לכתוב לקבצים, לקרוא הודעה מהמשתמשים במחשב, לשלוח הודעות ברחבי האינטרנט, ו... עוד ממש המון! בשלב זה נגיד ל-Guido תודה על הסיוע, ונמשיך במסע שלנו בו נלמד עוד רבות, ונהפוך להיות מתכנתים טובים הרבה יותר.

פרק 4 – להתראות Guido

בפרק הזה נלמד את יסודות שפת Python, ואיך לכתוב בה את כל הפקודות והמבנים שכבר למדנו בעזרת Guido.

למה אנחנו נפרדים מGuidon?

עד כה השתמשנו ב-Guido כדי לכתוב קוד, אבל הגיע הזמן להיפרד ממנה ולעבור הלאה לפייתון. הסיבה לכך היא, כפי שאנחנו יודעים, של-Guido אין זיכרון. בנוסף, Guido פועלת רק בעולם שלה, עם ביפרים וחומות – ואנחנו נרצה לתכנת גם מחוץ לעולם הזה, ולעשות דברים אחרים.

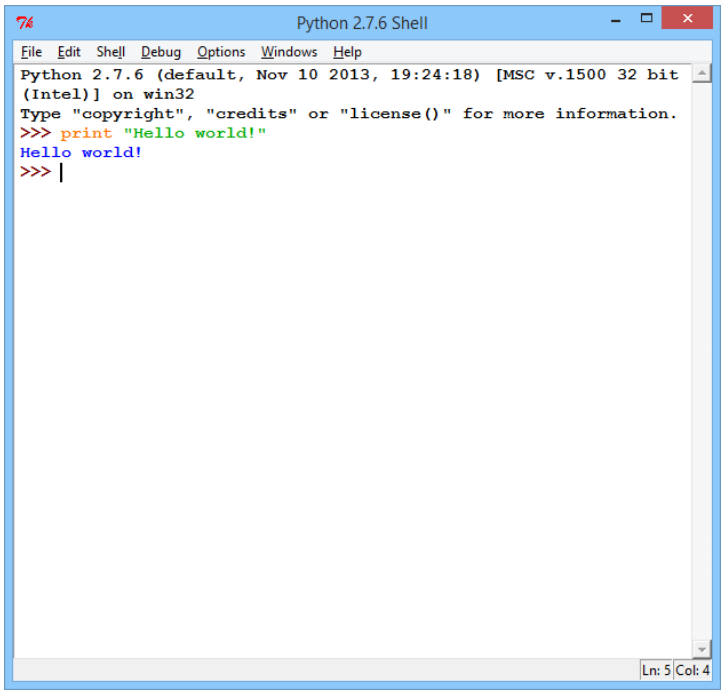
כפי שנראה בהמשך, לשפת Python ולשפה של Guido יש הרבה במשותף. בפרק זה נלמד איך "לתרגם" את כל מה שלמדנו מ-Guido לשפת Python, ובפרק הבא נלמד איך להשתמש בזיכרון של Python ובכלים הנוספים שהיא מאפשרת.

מתחילים – Python IDLE

כפי של-Guido הייתה תוכנית משלה, שדרכה מריצים את הפקודות שלה, כך גם לפייתון. בהמשך, נלמד לעומק על סביבת העבודה שלנו ב-Python, אבל בינתיים תספיק לנו התכנית IDLE. התכנית מגיעה עם התקנת Python - https://www.python.org/ftp/python/2.7.10/python-2.7.10.msi. לאחר שתתקינו את Python, נוכל להתחיל לתכנת.

חפשו בכפתור ה-Start את התוכנה IDLE .אם אתם משתמשים ב-Windows 8 ומעלה, החיפוש נמצא בכפתור במקלדת ששמו WinKey: .

כעת, נפתח לנו חלון הפקודה של IDLE. חלון זה נקרא ה-**interpreter**, וכל פקודה שנריץ בו תקרה באופן מיידי:



כבר עכשיו אנחנו יודעים פקודה אחת בפייתון – **print**. הפקודה מדפיסה כל דבר שניתן לה, במקרה זה היא קיבלה **מחרוזת**.

מחרוזת היא רצף של תווים, המסומנים ב**מרכאות**. ניתן לסמן מחרוזת במרכאות בודדות או במרכאות כפולות.

מרכאות כפולות: **"Hello world!"**

מרכאות בודדות: **'Hi! 123 &%6'**

נרחיב על נושא המחרוזות בהמשך.

בינתיים, נחזור לפקודה שהרצנו ב-interpreter ונתבונן בה ובחלון ה-interpreter בכלל:

```
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.1500 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> print "Hello world!"
Hello world!
>>>
```

בהתחלה, מודפסת עבורנו הודעת כניסה, שאומרת לנו באיזו גירסת Python אנחנו משתמשים. שימו לב שאנו משתמשים בגירסה 2.7.

לאחר מכן, מופיעים לנו שלושה סימני **>** שמראים על שורת הפקודה. בשורה זו נעמיד את הסמן ונכתוב את הפקודה שלנו:

>>> print "Hello world!"

לאחר מכך, יופיע הפלט של הפקודה למסך, בצבע כחול.

Hello world!

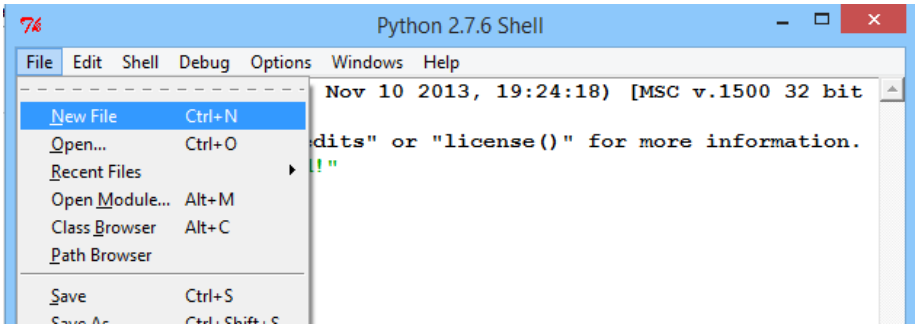
ניתן להדפיס כמה פקודות שנרצה, אחת בכל שורה המסומנת ב- **>>>**.

התכנית הראשונה שלי

תרגיל מודרך 4.1 – התכנית הראשונה שלי

כעת, נלמד איך לכתוב תכנית. למרות שיש יתרון גדול בהרצה של פקודה יחידה ב-interpreter, ברוב המקרים נרצה לכתוב תכנית שבה רצות כמה פקודות בזו אחר זו. בתרגיל זה נכתוב תכנית המדפיסה את המחרוזת "Hello world!" 5 פעמים.

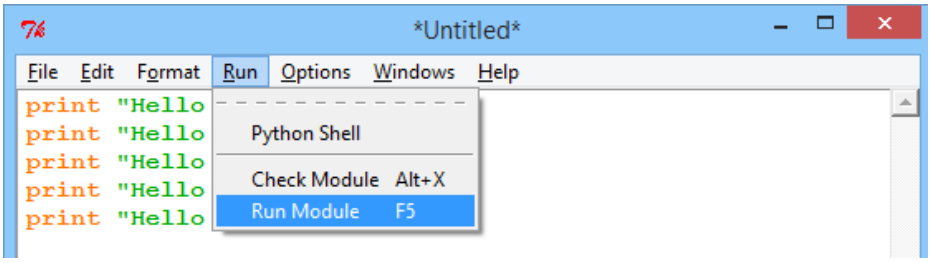
בתוך IDLE, ניכנס לתוך תפריט File, ונלחץ על New File. פעולה זו תפתח לנו חלון חדש בו נוכל לערוך את הקוד שלנו. נשים לב שאפשר גם להשתמש בקיצור Ctrl+N כדי ליצור קובץ חדש של תכנית.



עכשיו, נכתוב את התכנית שלנו. כמו ב-Guido, כל פקודה צריכה להיות רשומה בשורה נפרדת:

```
print "Hello world!"
print "Hello world!"
print "Hello world!"
print "Hello world!"
print "Hello world!"
```

כעת, נשמור ונריץ את התכנית שלנו, בעזרת כפתור F5 במקלדת, או בעזרת לחיצה על הכפתור בתפריט **Run** שנמצא בעמוד של התכנית.



כשנריץ את התכנית בפעם הראשונה, נצטרך לשמור אותה. נשמור את הקובץ בתיקיית התרגילים שלנו, בשם hello.py.

חשוב מאוד לשמור את התכנית שלנו **בסיומת py.** ללא סיומת זו, התכנית לא תרוץ!

כעת, התכנית רצה. נראה את הפלט של התכנית במסך הראשי של IDLE, ב-interpreter:

Hello world!
Hello world!
Hello world!
Hello world!
Hello world!

מזל טוב! כתבנו את התכנית הראשונה שלנו בשפת Python!

כתיבת פונקציה

ל-Guido היו פונקציות, שעזרו לה להגדיר פעולות שיחזרו על עצמן. גם בפייתון יש פונקציות, והשימוש בהן דומה מאוד. נגדיר את הפונקציה שמדפיסה את המספרים 1 עד 5:

```
def print_5():
    print 1
    print 2
    print 3
    print 4
    print 5
```

מה הם ההבדלים בין פונקציה ב-Python לפונקציה ב-Guido?

- במקום המילה define נשתמש בקיצור במילה **def**.
- בפייתון, לאחר שם הפונקציה נרשום סוגריים (). נלמד איך הסוגריים האלו משמשות אותנו בהמשך.

מה הן נקודות הדמיון?

- גם ב-Python צריך לשים **נקודותיים** אחרי הגדרת הפונקציה.
- גם בפייתון ישנה **אינדנטציה**.

כלומר, לשפת Python יש **syntax** משלה, שחלקו דומה ל-Guido וחלקו שונה.

כעת נקרא לפונקציה:

```
print_5()
```

יודפסו למסך המספרים 1 עד 5.

נשים לב כי ניתן להדפיס מספרים ללא מרכאות! לעומת זאת, אם ננסה להדפיס אות ללא מרכאות (לדוג', **print a**), פייתון צפוי "לצעוק" עלינו בשגיאה אדומה. אפשר לנסות.

תרגיל 4.2 – שלום ל...

נכתוב תכנית אשר מדפיסה 3 פעמים את רצף המילים:

```
Hello!
Good morning!
How are you?
Nice to meet you,
someones_name.
```

כאשר במקום **someones_name** יופיע בכל פעם שם אחר (לדוגמא, השמות Omer, Shay ו-Barak).
רמז – ניתן לכתוב את הפונקציה greetings().

כתיבת לולאות while ו-for

כעת, נלמד איך לתרגם את לולאות **while** ו-**do** משפת Guido ל-Python. מכיוון שההקבלה בין השפות אינה מלאה לחלוטין, נלמד את שתי הלולאות האלו באופן חלקי בלבד בשלב זה, ונרחיב עליהן בפרק הבא.

כתיבת לולאת while

ממש כמו ב-Guido, גם בשפת Python לולאת **while** מקבלת ערך בוליאני (ערך אמת או שקר), ופועלת רק אם יש בתנאי הלולאה ערך אמת. ב-Python אנו נשתמש בינתיים בערכי האמת השמורים בשפה - **True** ו-**False**.
בפרק הבא נלמד איך רושמים **תנאים** ללולאת **while**.
נתחיל בכתיבת הלולאה. מבנה הלולאה הוא כזה:

```
while True:
    …
    …
    …
```

לאחר השורה עם הפקודה while ישנה שוב **אינדנטציה** וגם נקודותיים, בדומה לכתיבת לולאת **while** ב-Guido.
במקום **True** נוכל לרשום **False**, ואז הפקודות שבתוך הלולאה לא יתבצעו. אחרי תנאי הכניסה ללולאה (**True** או **False**), נרשום את הפקודות שיתבצעו בתוך הלולאה.

תרגיל מודרך 4.3 – הדפסה בלולאת while

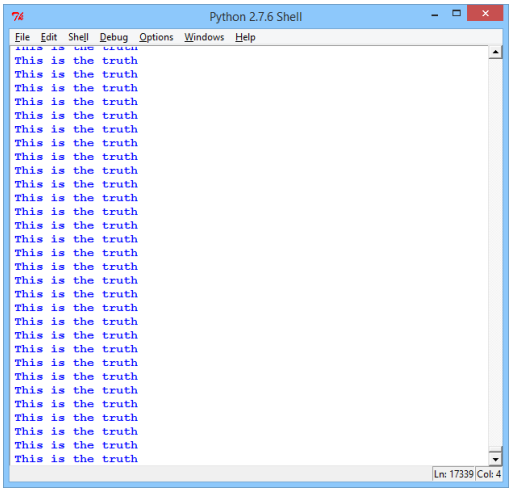
נפתח תכנית חדשה בשם truefalse.py, ונרשום בה את שתי הלולאות הבאות:

```
while False:
    print "This is a lie"

while True:
    print "This is the truth"
```

לפני שנריץ את התכנית שלנו, ענו על השאלה – **מה יודפס למסך?**

התשובה כמובן, היא שהמחרוזת הראשונה, "This is a lie" לא תודפס בכלל, כי התנאי של הלולאה הוא **False** תמיד. לעומתה, המחרוזת "This is the truth" תודפס למסך ללא הפסקה, כי השורה **while True** תמיד תקבל ערך אמת **True**.



כדי לעצור את הריצה האינסופית של הלולאה, לחצו על ctrl + C (או על ctrl+F6).

תרגיל 4.4 – פייתון לנצח

הדפיסו למסך את השורות הבאות בלולאה אינסופית:

I

Love

Python

Forever

עצרו את התכנית כפי שלמדנו[11].

[11] הבהרה: התכנית אמנם נעצרה, אך אתם מצופים להמשיך ולאהוב את פייתון לנצח בכל זאת.

כתיבת לולאת for

ב-Guido הכרנו פקודה בשם do, שקיבלה מספר שהורה על כמות הפעמים שיש לבצע את הפעולה בתוך הלולאה. בפייתון, קיימת פקודה מקבילה ששמה for. הפקודה הזו מורכבת בהרבה מפקודת do, ויכולה לעשות דברים מעניינים הרבה יותר. נלמד עליה עוד בהמשך. בינתיים, נלמד איך לבצע פעולה פשוטה מספר פעמים.

בכתיבת הלולאה, נשתמש בכלי של פייתון ששמו **xrange()**. הוא סופר עבורנו את מספר הפעמים שהפעולה תתבצע[12].

תרגיל מודרך 4.5 – שלום עולם!

נדפיס יחד את המחרוזת "Hello world!" 5 פעמים למסך, כפי שעשינו מקודם.

```
for iteration in xrange(5):
    print "Hello world!"
```

זוהי לולאת for – הריצו אותה וראו מה קורה.

נשים לב שגם בלולאה זו יש נקודותיים **ואינדנטציה** אחרי השורה עם הפקודה for, המסמלת כניסה ללולאה. המספר 5 בתוך ()xrange מסמל את מספר הפעמים שהפעולה תתבצע. המילה iteration מסמלת את הריצה היחידה של הלולאה. בתרגום חופשי לעברית, ניתן לקרוא את הלולאה כך:

```
לכל ריצה ב טווח 5:
    הדפס "שלום עולם!"
```

בלולאה הזו, המילה iteration היא "סתם" שם, ונוכל להחליף אותה בשם יותר קצר – לדוגמה בשם i:

```
for i in xrange(5):
    print "Hello world!"
```

נשים לב שב-IDLE כל מילה שהיא מילה שמורה בשפת Python נצבעת **בצבע כלשהו**. מילים שהן "סתם" שמות (כמו iteration), או דברים שניתנים להחלפה (כמו המספר בתוך ()xrange) נשארות בצבע **שחור**.

[12] בהמשך, נבין בצורה טובה יותר מהו xrange. בשלב זה נתמקד בהבנת לולאת for.

תרגיל 4.5 – נופל וקם

הדפיסו, רק בעזרת לולאות **for** (ללא לולאת **while**) את מילות השיר "נופל וקם". הקפידו לחסוך בשורות קוד, ולא לרשום אף שורה פעמיים! בעקבות הרצה מוצלחת של התכנית, יודפס למסך הפלט הבא:

The road is long and winding, I

Fall and get up

Fall and get up

Fall and get up

The road is long and winding, I

Fall and get up

Fall and get up

Fall and get up

The road is long and winding, I

Fall and get up

Fall and get up

Fall and get up

פרק 4 – סיכום

כפי שניתן לראות, כל מה שלמדנו עד עכשיו עם Guido ימשיך לשמש אותנו. למדנו איך לכתוב **פונקציות**, וגם איך משתמשים ב**לולאות** **while** **ו-**for. למדנו איך להשתמש ב-**print** וב-**()xrange**, ועוד נשתמש בהם בהמשך. ראינו של-Python ו-Guido יש **syntax** דומה מאוד, הכולל שימוש בנקודותיים והזחה. דבר זה אינו מקרי, הוא נובע מכך ש-Guido נבנתה על בסיס פייתון!

הידעת? Guido נקראת על שם Guido Van Rossum, הממציא של שפת Python. בנוסף, שפת Python קרויה על שם חבורת הקומיקאים Monty Python!

בפרק הבא נלמד עוד על לולאות **for** ו-**while**, ואיך להשתמש בהן עם **תנאים**. כך נוכל לכתוב תוכניות גדולות ומעניינות, שעושות דברים יותר מורכבים. בנוסף, נלמד עוד קצת על **()xrange** ותפקידו.

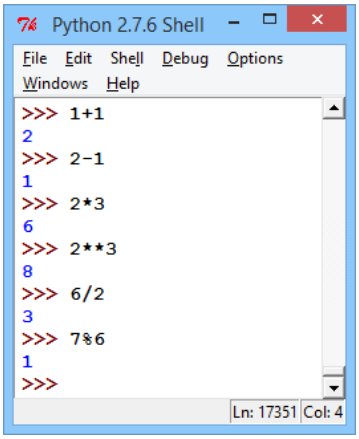
פרק 5 – מתחילים עם פייתון

בפרק זה נלמד כיצד להשתמש בשפת Python כמחשבון, איך פייתון זוכר ואיך כותבים תכנית עם לולאות ותנאים. בסוף הפרק הנוכחי נכתוב את המשחק הראשון שלנו – **חידון טריוויה**. במהלך הפרק נלמד בכל פעם כלי חדש שיעזור לנו להתקדם אל עבר המשחק הסופי. בנוסף, במהלך הפרק משובצים תרגילים קטנים, שיעזרו לנו בתרגול החומר שנלמד בהקשרים שונים.

חשוב מאוד שגם בחלק זה נמשיך לתרגל ב-interpreter של IDLE את כל הפקודות הרשומות בספר. רוב התרגילים הם תרגילים מודרכים, ובסוף הפרק יש תרגיל **מחשבון דלק** ותרגיל **חידון טריוויה**, שאותם תפתרו לבד.

מחשבון חכם מאוד

בבסיס של כל שפת תכנות קיימים חישובים שהשפה מבצעת. בחישובים אלו אפשר ורצוי להשתמש בתכניות שנכתוב, ולכן נכיר אותם כבר עכשיו:



חיבור – מסומן על ידי הסימן +

חיסור – מסומן על ידי הסימן -

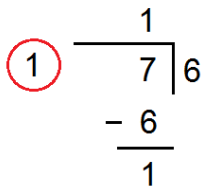
כפל – מסומן על ידי הסימן *

חזקה – מסומן על ידי הסימן ** (שתי כוכביות)

חילוק – מסומן על ידי הסימן /

שארית – מסומן על ידי הסימן %, ונקראת גם מודולו (modulo).

זוהי שארית החילוק במספר, כמו שאנחנו רגילים לקבל בחילוק ארוך:



בנוסף, ניתן להשתמש ב**סוגריים** בפייתון כדי לוודא סדר פעולות חשבון.

תרגיל 5.1 – חישובים

מהי התשובה לתרגילי החשבון הבאים? חשבו בעזרת פייתון:

$$100 \cdot 77 \cdot \left(40 + (36 \div 6)\right) = ?$$

$$(50 mod (7) + 89 + 100{,}563{,}927)^2 = ?$$

תרגיל 5.2 – חישובים נוספים

מהי התשובה שפייתון מחזיר לתרגיל הבא?

$$5 \div 2 = ?$$

האם זו התשובה שציפיתם לה?

סוגי מספרים

למעשה, לפייתון יש שני סוגים של מספרים – מספרים עגולים ומספרים עשרוניים.

המספרים העגולים נקראים בפייתון **int**, או integers. המספרים האלו כוללים את 0, 1, 2, 3, 4, 5 וכולי. מספרים מסוג int יכולים להיות גם שליליים (1-, 2-, 3-).

המספרים העשרוניים נקראים בפייתון **float**, והם מספרים שיש להם איברים אחרי הנקודה (0.77, 1.25, 107.64). גם מספרים "שלמים" יכולים להיות מסוג float, על ידי כך שנוסיף להם נקודה עשרונית – 1.0, 2.0, 3.0 וכולי. כמובן שלא כל המספרים יכולים להיות מיוצגים באופן מדוייק על ידי פייתון – מספרים שהשבר העשרוני שהם הוא אינסופי, כגון $\frac{1}{3}$, לא יכולים להתקיים בפייתון, ובמקום זה נקבל **קירוב** שלהם, לדוגמה 0.333333333 .

אז איך מספרי **float** עוזרים לנו? הם נחוצים בעיקר בפעולות חילוק, ולכן כשנרצה לחלק מספר באופן מדוייק, נשתמש ב-float: 5.0/2.0 .

הערה: לא כדאי להשתמש במספרים מסוג float יותר מידי, כי הם לא מדוייקים ועלולים ליצור בעיות.

תרגיל מודרך 5.3 – שגיאה!

נדפיס ב-interpreter את הפקודה הבאה:

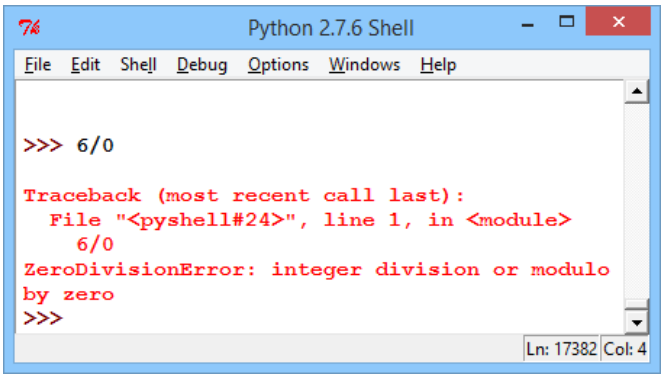
```
>>> 6/0
```

מה שה-Interpreter הדפיס נקרא **שגיאה**. שגיאות קורות כשאנחנו מנסים לעשות משהו בפייתון שהוא "נגד החוקים". למזלינו, פייתון מנסה "לצעוק" עלינו בבירור, והשגיאה כתובה באנגלית. בדרך כלל, כל שנצטרך לעשות יהיה לקרוא את השגיאה על מנת לגלות על איזה "חוק" עברנו.

בשגיאה שקיבלנו עכשיו רשום בשורה האחרונה:

```
ZeroDivisionError: integer division or modulo by zero
```

כלומר, חילקנו ב-0 ועל כן עברנו על ה'חוקים' של פייתון.



שגיאות הן חשובות ומסייעות לנו להבין היכן טעינו. חשוב לקרוא אותן ולהיעזר בהן.

זיכרון ומשתנים

בינתיים, ראינו שפייתון הוא מחשבון מאוד חזק. נוסף על כך, פייתון היא שפה שמאפשרת לנו לשמור כל מיני ערכים, במה שנקרא **משתנים.** משתנה הוא בעצם שם, שבתוכו נשמר מה שאנחנו רוצים לשמור.

ננסה זאת ב-interpreter. נכניס ערך לתוך משתנה בעזרת שימוש בסימן שווה[13] =. פעולה של הכנסת ערך למשתנה נקראת **השמה**. נסתכל בערך שבמשתנה על ידי כתיבתו. חשוב לשים לב – זו איננה הדפסה, וניתן לראות את ערך המשתנה באופן זה רק ב-interpreter.

```
>>> x = 1
>>> x
1
```

אם המשתנה מכיל מספר, ניתן לבצע עליו פעולות חישוב עם מספר אחר, או אפילו עם משתנה אחר שמכיל מספר:

```
>>> x = 1
>>> x + 5
6
>>> y = 3
>>> x+y
4
```

למעשה, נוכל להחליף את הערך של אחד המשתנים בקלות:

```
>>> x = 1
>>> x = 2
>>> x+y
5
```

המשתנה שלא החלפנו, y, נשאר כפי שהיה, ולכן תוצאת החיבור ביניהם תהיה כעת 5.

ניתן גם להוסיף למשתנה את עצמו!

```
>>> x = 1
>>> x = x + 1
>>> x
2
```

[13] על אף הרישום הדומה, אין מדובר במשוואה שאנו מכירים משיעורי מתמטיקה. לכן, x = 1 היא שורה שונה בעליל בפייתון מאשר x = 1, אשר צפוייה לזרוק שגיאה.

דרך נוספת לכתוב את השורה **x = x + 1** היא **x += 1**.

נשים לב: אמנם השתמשנו בדוגמאות בפעולת החיבור, אבל ניתן להשתמש בכל פעולת חשבון שנרצה.

הקפידו על הריווח בזמן ההשמה, כדי שהתכנית תהיה יפה וקריאה:

- **x = 1** **יפה וקריא**
- **x=1** **לא יפה ופחות קריא**

ניתן לעשות השמה לא רק למספרים (int ו-float), אלא גם למחרוזות (עליהן למדנו בפרק שעבר):

```
>>> x = 17
>>> y = 1.68
>>> m = "Hello"
```

בדרך כלל כשנעשה השמה, נרצה לתת למשתנים שלנו **שמות אינדיקטיביים,** שיבהירו מה הם מכילים:

```
>>> age = 17
>>> height = 1.68
>>> greeting = "Hello"
```

שימו לב עד כמה הקוד הופך לקריא וברור יותר, רק בעזרת מתן שמות אינדיקטיביים למשתנים!

ניתן להדפיס את המספרים והמחרוזות הללו בעזרת שם המשתנה שלהם:

```
>>> greeting = "Hello"
>>> print greeting
"Hello"
```

תרגיל 5.4 – מי אני

נכתוב תכנית בשם me.py, המדפיסה את השורות הבאות:

```
my_greeting
My name is:
my_name
My age is:
my_age
And my height is:
```

my_height

במקום ההדפסות שמתחילות ב-my_ יודפסו: ברכה מכם, שמכם, הגיל והגובה שלכם. הקפידו לקחת את המידע הזה מ**משתנים** בתחילת קובץ התכנית.

תנאי if ואופרטורים

אופרטורים

לפעמים נרצה לדעת אם שני דברים הם שונים או דומים, או אם אחד קטן מהשני – כלומר נרצה לערוך **השוואה**. בשביל זה יש **אופרטורים**, שעוזרים לנו להשוות. כרגע נעסוק באופרטורים על מספרים, ואחרי זה נראה השוואה בין אובייקטים שונים.

בשביל להבין מה מחזיר אופרטור, צריך קודם כל להבין מה הוא ערך בוליאני (boolean). ערך בוליאני אומר האם דבר מה הוא אמת (True) או שקר (False). כפי שהזכרנו בפרק הקודם, שתי המילים **True** ו-**False** הן מילים שמורות בפייתון, המסמלות את הערכים הבוליאנים של אמת ושקר. בעזרת ערכים אלו, ניתן לדעת האם התוצאה של ה**השוואה** היא אמת או שקר.

לדוגמה, נרשום שני משתנים[14], באחד נשים את הערך 5 ובשני את הערך 10:

```
>>> x = 5
>>> y = 10
```

האם הערכים שווים? נבצע את ההשואה הזו בעזרת ה**אופרטור** ==:

```
>>> x == y
False
```

קיבלנו את הערך False, כלומר הטענה לא מתקיימת – הערכים במשתנים אינם שווים. חשוב לזכור שההשוואה היא לא בין המשתנים אלא בין הערכים שלהם, כלומר ההשוואה למעלה זהה להשוואה הבאה:

```
>>> 5 == 10
False
```

אם נרשום טענה שמתקיימת, נקבל חזרה את הערך True:

```
>>> 2*x == y
True
```

כמובן שיש עוד דרכים להשוות. ניתן לשאול האם שני ערכים **שונים** זה מזה על ידי ה**אופרטור** =! :

```
>>> x != y
True
```

וניתן גם לשאול האם אחד גדול מהשני (> או <) או גדול-שווה מהשני (=> או =<).

```
>>> x < y
True
```

[14] באופן מיוחד, אנו חורגים ממנהגנו וקוראים למשתנים בשמות לא אינדיקטיביים. אנו עושים זאת רק מכיוון שלמשתנים האלו אין באמת משמעות.

נשים לב שבין משתנים שהערכים שלהם הם לא מספריים ניתן רק להשוות:

```
>>> x = "a"
>>> y = "b"
>>> x == y
False
>>> x != y
True
```

תנאים

ניזכר שאנחנו רוצים לכתוב טריוויה. במשחק טריוויה, מציגים למשתמשים שאלות, ומקבלים תשובות. איך נדע האם המשתמשים צדקו? לשם כך נצטרך להשתמש ב**תנאים**.

בעזרת כל האופרטורים עליהם למדנו, נוכל לכתוב תנאים. כל תנאי בפייתון מתחיל במילה if, ולאחריה התנאי עצמו. נכתוב זאת בתכנית בקובץ בשם user.py:

```
user = "me"
if user == "me":
        print "Welcome to your computer"
```

כעת, אם נריץ את התכנית, המחרוזת תודפס למסך.

חשוב לזכור שההשואה בתנאי if בעצם מסמלת את הערך שהיא מחזירה. לו היינו רוצים שתמיד יתקיים התנאי, היינו יכולים לרשום כך:

```
if True:
        print "Welcome to your computer"
```

שאלת מחשבה: מה יקרה אם נשנה את השורה הראשונה ל – user = "omer"? בדקו וודאו שתשובתכם נכונה.

אפשר לחייב שכמה תנאים יתקיימו יחדיו בעזרת המילה and, שמחייבת ש**כל** ההשוואות יהיו עם הערך True, לדוגמה:

```
user = "me"
pass_num = 7935
if user == "me" and pass_num == 7935:
        print "Welcome to your computer"
```

או לחלופין, אפשר לוודא ש**לפחות אחת** מההשוואות תחזיר ערך True, בעזרת המילה השמורה or.

```
user = "omer"
if user == "me" or user == "omer" or user == "shay":
        print "Welcome to your computer"
```

תרגיל מודרך 5.5 – בדיקת תשובה

אנחנו כבר בדרך לכתיבת הטריוויה! נתרגל כתיבת תנאי אשר בו נבדוק האם התקבלה התשובה הנכונה לשאלה. ראשית, נרשום בתחילת התכנית את השאלה, ואת התשובה. מכיוון שאנחנו לא מתכוונים לשנות את השאלה או את התשובה, נתייחס אל שני המשתנים כאל **קבועים**. נרצה להבדיל אותם בהמשך מהתשובות של השחקן, שבאמת הולכות להשתנות בזמן ריצת התכנית. לכן, נסמן את הקבועים בתחילת התכנית באותיות גדולות. בינתיים נרשום את תשובת השחקן במשתנה כלשהו, ובהמשך נלמד איך לקבל ממנו את התשובה בזמן ריצת התכנית:

```
QUESTION_1 = "What is your favorite programming language?"
ANSWER_1 = "Python"

player_answer = "py thone"
```

כעת נוסיף את הקוד אשר מדפיס את השאלה ובודק את התשובה:

```
print QUESTION_1
if player_answer == ANSWER_1:
        print "Good! You are correct"
```

תרגיל 5.6 – מספר חוסר מזל

כידוע, בכמה מדינות ברחבי העולם, המספר 13 נחשב מספר חסר מזל[15], וכל מי שמשתמש בו בתכנית, סופו לחטוף הרבה **שגיאות**. מה שרוב האנשים לא יודעים הוא שגם הכפולות של 13 הן חסרות מזל.

כיתבו תכנית שבודקת האם מספר הוא כפולה של 13, ואם כן תודפס למסך אזהרה (בסגנון "THIS IS A TERRIBLE NUMBER"). על המספר הנבדק להיות מושם לתוך משתנה, ועל הבדיקה להתרחש על המשתנה.

[15] יש אפילו מונח לפחד ספציפי מהמספר 13 – Triskaidekaphobia.

תנאי elif ו-else

למדנו כיצד לבדוק האם תנאי מתקיים ולבצע פקודות כאשר הוא מתקיים. אבל מה אם נרצה לבצע פקודות אחרות אם מתקיים תנאי אחר? ומה אם נרצה לבצע פקודה כלשהי דווקא במקרה שהתנאי לא מתקיים?

לצורך מענה על הבעיות האלו, יש לפקודת **if** המשך. בעברית, נקרא להמשך זה אם-אם אחר-אחרת, ובאנגלית, if-elif-else. כיצד עובד התנאי המורכב הזה? נראה את פעולתו בעזרת התכנית user.py שרשמנו בעמוד הקודם.

בתחילה, נרשום פסוקית if:

```
user = "omer"
if user == "me":
        print "Welcome to your computer"
```

נוסיף תנאי נוסף, elif, שידפיס הודעה במקרה שהמשתמש מוכר אבל איננו אני:

```
elif user == "omer" or user == "shay":
        print "Welcome, guest"
```

ניתן לרשום כמה תנאי elif שנרצה, כשבכל אחד נרשום תנאי אחר. לדוגמה, נוכל להחליף את תנאי ה-elif למעלה בשני תנאים:

```
elif user == "shay":
        print "Welcome, Shay"
elif user == "omer":
        print "Welcome, Omer"
```

התנאים נבדקים לפי סדר כתיבתם. אם תנאי ה-if התקיים, אז כל תנאי ה-elif שאחריו לא ייבדקו, ואם אחד מתנאי ה-elif מתקיים אז תנאי ה-elif שאחריו לא ייבדקו.

ולבסוף, נרשום תנאי else, שידפיס הודעה עבור משתמש זר:

```
else:
        print "Go away!"
```

נשים לב שתנאי else לא מקבל אף השוואה, שכן הוא פועל רק במידה שכל שאר התנאים לא התקיימו.

לא חייבים לרשום את כל חלקי התנאי. מלבד הצורה הרגילה, תנאי יכול להיות מצורת if-else וגם מצורת if-elif.

תרגיל מודרך 5.7 – בדיקת תשובה

נמשיך בדרכינו לכתיבת משחק טריוויה. נערוך את הקוד שלנו לבדיקת התשובה כך שיודיע למשתמש אם טעה. ראשית ניזכר בקוד שכתבנו:

```
QUESTION_1 = "What is your favorite programming language?"
```

```
ANSWER_1 = "Python"

player_answer = "py thone"

print QUESTION_1
if player_answer == ANSWER_1:
        print "Good! You are correct"
```

ונוסיף את תנאי ה-else:

```
else:
        print "Wrong answer"
```

תרגיל 5.8 – מספר חוסר מזל (כמעט)

ערכו את הקוד שכתבתם בתרגיל 5.6, כך שכעת, אם המספר מתחלק ב-12 **או** ב-14 תודפס למסך הודעת הקלה (נניח, "Whew, that was close. This number is OK"), ובכל מקרה אחר תודפס הודעה שמודיעה שהמספר הוא טוב ("This is a good number").

הבהרה: אם המספר מתחלק גם ב-12 וגם ב-13, עדיין תודפס למסך הודעת אזהרה – שכן המספר עדיין מתחלק ב-13.

לולאת while עם תנאים

לפעמים, נרצה לעשות משהו כל עוד תנאי מתקיים. נוכל, למשל, לעשות זאת כך:

```
x = 1
while x == 1:
        print x
```

הלולאה הזו תדפיס את הערך של x (הספרה 1) למסך עד שלא נעצור את התכנית "בכח", היות שהתנאי יתקיים תמיד. לעומת זאת, אם נוסיף שורה יחידה בקוד שתשנה את x, הספרה 1 תודפס פעם אחת בלבד:

```
x = 1
while x == 1:
        print x
        x += 1
```

מה קרה?

בהתחלה, התנאי התקיים, ואז התחילה הריצה בלולאה, והספרה 1 הודפסה. שורה לאחר ההדפסה, שינינו את x והגדלנו אותו ב-1. בריצה השניה של הלולאה כבר לא התקיים התנאי x == 1 (כי בשלב זה, x == 2), ולכן יצאנו מהלולאה והופסקה הריצה.

תרגיל 5.9 – כל המספרים ששלחתי לדרכם

הדפיסו למסך את כל המספרים מ-1 עד 10, בעזרת לולאת while. היעזרו בדוגמת הקוד לעיל.

רמז – נסו להשתמש בתנאי > ('קטן מ-').

תרגיל מודרך 5.10 – שלמה הגמל

בתוך לולאת while ניתן לרשום תנאים שיאמרו מה צריך לקרות בריצה של הלולאה. בתרגיל הזה נכתוב לולאה שמנסה לחקות את צעדיו של שלמה הגמל, מנקודה 0 עד נקודה 10. שלמה הגמל כבר מזמן פרש מההליכה הישנה שלו (קדימה, קדימה, אחורה, סיבוב) וכעת הוא צועד בצורה מאוד מיוחדת – אם הוא נמצא על נקודה שמספרה מתחלק ב-3 ללא שארית (נניח - 9) אז הוא הולך צעד אחורה. אחרת, הוא מדלג שני צעדים קדימה. נדפיס את כל הנקודות בהן שלמה דורך, בסדר בו הוא דורך עליהן.

ראשית, נרשום את תנאי ההתחלה (הנקודה בה שלמה נמצא – נקודה 0):

```
shlomo_point = 0
```

לאחר מכן, נרשום את התנאי. ידוע לנו כי שלמה לא יעצור עד שלא יגיע ל-10, וידועה לנו צורת ההליכה המשונה שלו:

```
while shlomo_point != 10:
    if shlomo_point % 3 == 0:
        shlomo_point -= 1
    else:
        shlomo_point += 2
```

תזכורת: השורה **shlomo_point -= 1** שקולה ל-**shlomo_point = shlomo_point - 1**, והשורה **shlomo_point += 2** שקולה ל-**shlomo_point = shlomo_point + 2**.

עדיין יש כמה בעיות בתכנית שלנו – היא לא מדפיסה את המיקום של שלמה. בנוסף, נרצה שכל מי שיקרא את התכנית שלנו ידע מה עושים בה, ולכן נוסיף לה **הערות**!

הערות מסומנות בעזרת סולמית # והן לא רצות כחלק מהקוד.

```
# Calculate Shlomo the camel's walking pattern
```

```
shlomo_point = 0
while shlomo_point != 10:
        print shlomo_point
        if shlomo_point % 3 == 0:    # Shlomo is on a point that's devisable by 3
                shlomo_point -= 1    # Walks backwards
        else:
                shlomo_point += 2    # Jumps 2 forward
```

תרגיל 5.11 – 7 בום.

מכירים את המשחק 7 בום? הגיע הזמן לכתוב דף עזר שבעזרתו תמיד תדעו מה לומר.

הדפיסו למסך את כל המספרים בין 0 ל-100 שמתחלקים ב-7 ללא שארית או שמכילים את הספרה 7, לפי הסדר (כלומר מ-0 עד 100). אם מספר כלשהו מכיל 7 או מתחלק ב-7, רשמו במקומו boom.

רמז – תוכלו לעשות זאת בעזרת לולאת while ו-3 פעולות חשבון!

קלט מהמשתמש - raw_input()

פעמים רבות, נרצה לקבל קלט מהמשתמש. בין אם המשתמש רוצה להשתמש בתוכנת המחשבון שנכתוב, ובין אם זו שחקנית במשחק הטריוויה שלנו – נרצה לדעת מה למשתמש יש להגיד.

תרגיל מודרך 5.12 – הדפסת מחרוזת לבקשת המשתמש

בפייתון, על מנת לקבל מידע מהמשתמש, נוכל לקרוא לפונקציה **raw_input**. כך, נוכל לקבל **קלט** מהמשתמש. כשקוראים לפונקציה זו, כותבים בתוך הסוגריים מחרוזת שאנו רוצים שהמשתמש יראה. עדיין לא למדנו מה באמת המשמעות של כתיבה בתוך הסוגריים בקריאה לפונקציה, ונכיר זאת לעומק בהמשך. בשלב זה נסתפק בדוגמה:

```
>>> raw_input("What is your string? --> ")
```

בשלב זה, המחרוזת "What is your string? --> " תוצג למשתמש, שיוכל לכתוב מחרוזת כאוות נפשו, לדוגמה:

```
>>> raw_input("What is your string? --> ")
What is your string? --> I love Python!
'I love Python!'
>>>
```

כפי שניתן לראות, המחרוזת "I love Python!" הוחזרה על ידי ה-Interpreter. זה לא עוזר לנו יותר מידי. עם זאת, על מנת להשתמש בקלט של המשתמש, נוכל לשמור אותו למשתנה. כך למשל:

```
>>> user_string = raw_input("What is your string? --> ")
What is your string? --> I love Python a lot!
>>> print user_string
I love Python a lot!
>>>
```

למעשה, בשלב זה נוכל לבצע כל מיני פעולות עם user_string – הקלט של המשתמש. בעתיד נלמד על פעולות מעניינות שנוכל לעשות על מחרוזות. למשל, אפשר לבדוק האם המשתמש כתב את התשובה הנכונה לשאלה באמצעות **תנאי**. את פעולה זו תתבקשו לבצע כשתכתבו את משחק הטריוויה.

תרגיל 5.13 – שלום לך!

בתרגיל הבא, תבקשו מהמשתמשת את השם שלה. לאחר מכן, תאמרו לה שלום באופן מנומס. ריצה לדוגמה של התכנית שלכם, צריכה להראות כך:

```
>>>
What is your name? --> Shay
Hello
Shay
Nice to meet you!
>>>
```

בונוס: נסו להדפיס את השורה למשתמשת בשורה אחת, למשל:

```
>>>
What is your name? --> Shay
Hello Shay. Nice to meet you!
>>>
```

לא למדנו איך לעשות זאת? נכון. אבל Google כנראה יודע, והיום הוא הבטיח לענות על כל שאלה.

תרגיל מודרך 5.14 – קבלת מספר מהמשתמש

בתרגיל זה נבקש מהמשתמשת מספר, ונדפיס לה את המספר ועוד 2. לכאורה, אין בכך כל בעיה. נשתמש ב-**raw_input** שהכרנו, נשמור את המספר למשתנה, נוסיף לו שתיים, ואז נדפיס את התוצאה למשתמשת. ננסה זאת בקובץ בשם get_number.py:

```
user_number = raw_input("Please choose a number --> ")
print user_number + 2
```

אך כשאנו מריצים את התוכנה, קורה דבר מוזר:

```
>>>
Please choose a number --> 4

Traceback (most recent call last):
  File "C:/Windows/Temp/temp.py", line 2, in <module>
    print user_number + 2
TypeError: cannot concatenate 'str' and 'int' objects
>>>
```

מה קרה כאן?

נחזור לכך בעוד רגע. בינתיים, נזכר כי הכרנו כמה טיפוסים (סוגים) של משתנים שונים בפייתון. הכרנו למשל את המספרים השלמים (**int**), המספרים העשרוניים (**float**), והמחרוזות (**str**). מה יקרה אם ננסה לשלב אותם יחד? מה יקרה, למשל, אם ננסה לחבר מספר שלם ומחרוזת?

בואו ננסה:

```
>>> 5 + "a"

Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    5 + "a"
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

פייתון שוב "צועק" עלינו בהודעת **שגיאה**. גם הפעם, ההודעה יכולה לעזור לנו להבין את הבעיה: האופרטור + לא נתמך עבור טיפוס מסוג מספר שלם (**int**) וטיפוס מסוג מחרוזת (**str**).

גם בהודעת השגיאה מהקוד שלנו, נראה שפייתון לא יודע כיצד לשלב אובייקט מסוג מחרוזת עם אובייקט מסוג מספר שלם. אמנם דבר זה מוזר, הרי הפעולה שלנו רצה על המספר שהמשתמשת בחרה, 4, והמספר שנתנו בקוד, 2. כלומר, אנו פועלים על שני מספרים.

מסתבר שהמצב לא פשוט עד כדי כך, והפונקציה **raw_input** **מחזירה** משתנה מסוג מחרוזת. כלומר, מחרוזת "4" ולא את המספר 4. נשים לב כי לא מדובר באותו משתנה:

```
>>> 4 == 4
True
>>> "4" == "4"
True
>>> "4" == 4
False
>>>
```

למעשה, בשורה שבה רשמנו:

```
print user_number + 2
```

מה שירוץ בעקבות בחירת המשתמשת בספרה 4, זו השורה:

```
print "4" + 2
```

שורה זו, כפי שראינו, צפויה להחזיר שגיאה.

על מנת להתמודד עם בעיה זו, נוכל לבצע **המרה (conversion)** – החלפה של סוג המשתנה בסוג אחר. בדוגמה זו, למשל, נוכל להמיר את המחרוזת "4" להיות מספר שלם – 4. לשם כך, נבצע את הפעולה הבאה:

```
int("4")
```

כלומר, כתבנו את סוג הטיפוס שאליו ההמרה תתבצע, ובתוך הסוגריים – את המשתנה הנוכחי. התוצאה של השורה שכתבנו לעיל היא 4, מסוג **int**, כפי שניתן לראות:

```
>>>
>>> 4 == 4
True
>>> "4" == "4"
True
>>> "4" == 4
False
>>> int("4") == 4
True
>>>
```

נתקן בהתאם את הקוד שלנו של הקובץ get_number.py:

```
user_number = raw_input("Please choose a number --> ")
print int(user_number) + 2
```

לחלופין, יכולנו לבצע את ההמרה באופן מפורש, כך:

```
user_number = raw_input("Please choose a number --> ")
user_number = int(user_number)
print user_number + 2
```

ואף יכולנו ליצור משתנה אחר, במיוחד לשם כך:

```
user_number = raw_input("Please choose a number --> ")
user_number_int = int(user_number)
print user_number_int + 2
```

כעת, התכנית צפויה לרוץ בהצלחה:

```
>>>
Please choose a number --> 4
6
>>>
```

בפרק הבא נלמד עוד על טיפוסים, וגם איך לגלות באמת מהו הטיפוס של משתנה מסויים, מבלי להסתמך על הודעות שגיאה.

תרגיל 5.15 – מחשבון דלק

בתרגיל זה תכתבו מחשבון שיסייע לנהגים לדעת כמה לשלם עבור הדלק שלהם.

עבור התרגיל, נניח כי על כל 10 קילומטרים של נסיעה, נשרף בדיוק ליטר דלק אחד. בנוסף, ידוע כי כל ליטר דלק עולה 5.5 ש"ח. על התכנית שתכתבו לשאול את המשתמש כמה קילומטרים הוא נסע, ולאחר מכן להדפיס את מחיר הנסיעה המלא. כך למשל תראה הרצה של התכנית:

```
>>>
How many kilometers did you drive? --> 12.4
The cost is: 0.225454545455 new Shekels.
>>>
```

הוראות:

א) יתכן שהמשתמש נסע מספר לא שלם של קילומטרים (כמו בדוגמה).

ב) זכרו להשתמש בשמות משתנים אינדיקטיביים, ובשמות משתנים מתאימים לקבועים, כמו שלמדנו בפרק זה.

תרגיל 5.16 – טריוויה

איזו התרגשות! אנו עומדים לכתוב את המשחק הראשון שלנו. נתחיל מלראות כיצד יכולה להראות ריצה של המשחק כשהוא כבר כתוב ועובד:

```
>>>
Welcome to the Trivia game! Good luck!

What is your favorite programming language?
--> Python
Good! You are correct

In what year was Python 2.0 released?
--> 2000
Good! You are correct

Who is the creator of Python?
A) Bill Gates.
B) Steve Jobs.
C) David Ben Gurion.
D) Guido van Rossum.
--> D
Good! You are correct

Who averaged one patent for every three weeks of his life?
A) Thomas Edison.
B) Guido Van Rossum.
C) Albert Einstein.
D) Spiderman.
--> C
Wrong answer :(

What is the world's longest river?
--> HaYarkon
Wrong answer :(

Game over, thanks for trying!
>>>
```

מוקדם יותר בפרק זה הכרנו מחרוזות, ולמדנו להכריז עליהן ולהשתמש בהן. על מנת לכתוב את המשחק, יהיה לנו נוח להגדיר מחרוזת ארוכה, כזו שכוללת בתוכה מספר שורות. ניתן לעשות זאת על ידי שימוש בתו גרש שלוש פעמים (''') או שימוש בתו גרשיים שלוש פעמים ("""). כך למשל:

```
>>> love_song = """Roses are #FF0000
Violets are #0000FF
All my base
Are belong to you"""
>>> print love_song
Roses are #FF0000
Violets are #0000FF
All my base
Are belong to you
>>>
```

הנחיות לכתיבת המשחק

במשחק זה תכתבו טריוויה בת 5 שאלות, שאותן תמציאו. השאלות יכולות לכלול תשובה מספרית (כמו השנה 2000, בדוגמה לעיל), תשובה של טקסט (כמו התשובה "Python", בדוגמה לעיל) או שאלת בחירה, שהתשובה עליה תהיה למשל "A". על כל תשובה, יש להדפיס למשתמשת האם היא צדקה או לא.

שימו לב להשתמש בקונספטים שונים שלמדנו לאורך הפרק – בדגש על תנאים, ומתן שמות אינדיקטיביים למשתנים. על מנת להדפיס שורה ריקה, כך שהפלט למשתמשת יהיה קריא ויפה יותר, תוכלו לרשום שורה שבה נכתב אך ורק **print**.

נשים לב – בעיות במשחק הטריוויה

אנחנו צריכים להיות גאים בעצמנו על כך שהצלחנו לכתוב משחק טריוויה אחרי כל כך מעט זמן של לימוד תכנות. עם זאת, יש הרבה בעיות במשחק שזה עתה כתבנו, בעיות איתן נלמד להתמודד בהמשך.

ראשית, סביר מאוד שיש לנו "שכפול קוד" – כתיבה של אותן שורות קוד, או שורות דומות מאוד, מספר פעמים. בקרוב מאוד נדע איך להתמודד עם בעיה זו.

שנית, הקוד שלנו מתמודד רק עם מקרים מאוד מסויימים. מה יקרה, למשל, אם במקום התשובה "Python" לשאלה הראשונה, המשתמשת תכתוב "python"?

```
>>>
Welcome to the Trivia game! Good luck!

What is your favorite programming language?
--> python
Wrong answer :(

In what year was Python 2.0 released?
-->
```

האם הגיוני ש-"Python" תהיה תשובה נכונה, אך "python" לא? כמובן שמצב זה אינו סביר. הוא נובע מכך שהמחרוזת **"Python"** והמחרוזת **"python"** הן שונות זו מזו:

```
>>> "Python" == "python"
False
```

בהמשך נלמד יותר על מחרוזות, ונגלה איך נכון לטפל במצבים כאלו.

תרגיל מודרך 5.17 – ספירה

בקרוב מאוד נרצה לשפר את משחק הטריוויה שכתבנו, ולספר למשתמשת על כמה תשובות היא ענתה בצורה נכונה. לשם כך, נבין כעת איך מבצעים ספירה על פי תשובות המשתמשת.

בתרגיל זה נכתוב תכנית פשוטה, שסופרת כמה פעמים המשתמש בחר באופציה "y" עבור "yes" מתוך 5 ריצות של לולאה. כך תראה ריצה של התכנית שלנו:

```
>>>
Should I count this time? (y / n) --> y
Should I count this time? (y / n) --> n
Should I count this time? (y / n) --> n
Should I count this time? (y / n) --> y
Should I count this time? (y / n) --> n
Counted 2 times.
>>>
```

על מנת לבצע ספירה, נשתמש ב**משתנה**. בתחילת הריצה, הערך שלו יהיה 0, שהרי המשתמש לא בחר עדיין לספור אף ריצה:

```
counter = 0
```

כעת, נוכל ליצור לולאה, ולבדוק מתי המשתמש בחר לבצע ספירה:

```
for i in xrange(5):
    user_choice = raw_input("Should I count this time? (y / n) --> ")
```

במידה שהמשתמש בחר לספור את הריצה, נגדיל את המשתנה **counter**:

```
for i in xrange(5):
    user_choice = raw_input("Should I count this time? (y / n) --> ")
    if user_choice == "y":
        counter += 1
```

לבסוף, נוכל להדפיס את הערך של **counter**. יהיה לנו נוח להדפיס זאת בשורה אחת, יחד עם מעט טקסט שיסביר את ערך המשתנה. נוכל לעשות זאת כך:

```
print "Counted", counter, "times."
```

כעת יש לנו קוד עובד. זו התחלה טובה, אך כמו שכבר למדנו – זה לא מספיק. עבור אדם אחר שיקרא את הקוד שלנו, יש כאן כמה דברים לא ברורים. מה זה "y"? למה המספר 5 מופיע בקוד? על מנת להפוך את הקוד לברור וקריא יותר, נשתמש במשתנים שיהיו למעשה קבועים. נוכל לשכתב את הקוד שלנו כך שיראה בצורה הבאה, שהיא קריאה הרבה יותר:

```
#############
# Constants #
#############
TIMES_TO_RUN = 5
QUESTION = "Should I count this time? (y / n) --> "
COUNT_THIS_TIME = "y"

#############
# Main Code #
#############
counter = 0

for i in xrange(TIMES_TO_RUN):
    user_choice = raw_input(QUESTION)
    if user_choice == COUNT_THIS_TIME:
        counter += 1

print "Counted", counter, "times."
```

תרגיל 5.18 – טריוויה משופרת

אנו מצויידים כעת בידע רב, שיסייע לנו לשפר את הטריוויה שכתבנו. בתרגיל זה, נגרום לתכנית שלנו להדפיס שני נתונים מעניינים בסוף הריצה שלה:

1) על כמה שאלות המשתמשת ענתה נכון.
2) מה ה"ציון" של המשתמשת (מתוך 100).

להלן ריצה לדוגמה של הטריוויה המשופרת:

```
>>>
Welcome to the Trivia game! Good luck!

What is your favorite programming language?
--> Python
Good! You are correct

In what year was Python 2.0 released?
--> 2000
Good! You are correct

Who is the creator of Python?
A) Bill Gates.
B) Steve Jobs.
C) David Ben Gurion.
D) Guido van Rossum.
--> D
Good! You are correct

Who averaged one patent for every three weeks of his life?
A) Thomas Edison.
B) Guido Van Rossum.
C) Albert Einstein.
D) Spiderman.
--> D
Wrong answer :(

What is the world's longest river?
--> HaYarden
Wrong answer :(

You got 3 right answers.
Your score is 60 out of 100.
Game over, thanks for trying!
```

נשים לב כי עדיין לא התגברנו על הבעיות שהצגנו קודם, כשניתחנו את משחק הטריוויה בגירסה הראשונה שלו. להיפך, ככל הנראה ששכפול הקוד שלנו רק החמיר. עם בעיות אלו נלמד להתמודד בהמשך.

פרק 5 – סיכום

בפרק זה למדנו הרבה דברים חדשים. בתחילה, למדנו על **פעולות חשבון** בפייתון., וכן על **סוגי מספרים**. הכרנו את המספרים השלמים (**int**), והמספרים העשרוניים (**float**). הכרנו **שגיאות**, והבנו שהן חשובות מאוד בעבודה שלנו עם פייתון, ושהן יכולות לעזור לנו.

בהמשך הפרק למדנו על זיכרון ו**משתנים**. הבנו ש**משתנה** הוא בעצם שם, שבתוכו נשמר מה שאנחנו רוצים לשמור. לאחר מכן למדנו על **אופרטורים**, כגון = (השמה), == (בדיקת השוואה), =! (בדיקת שונות), < (קטן מ-), > (גדול מ-) ועוד. באמצעותם למדנו על **תנאים**, ופגשנו ב-**if**, **elif** וגם **else**.

לאחר מכן, למדנו איך להשתמש בתנאים כדי לכתוב לולאות **while** מעניינות יותר – כאלו שתלויות בערך שמשתנה. כך למשל, כתבנו את 7 בום. בהמשך, למדנו כיצד לקבל **קלט** מהמשתמש באמצעות הפונקציה **raw_input**. שילבנו זאת עם הידע שלנו על תנאים ולולאות, כדי לכתוב תכניות שמתאימות את עצמן לקלט מהמשתמש. זה כבר נהיה מעניין!

בהמשך, כתבנו מחשבון דלק. לשם כך למדנו על **המרה** בין טיפוסים שונים של משתנים, באופן ספציפי – המרנו מחרוזת למספר. לקראת סוף הפרק, כתבנו את המשחק הראשון שלנו – משחק טריוויה. לאחר גירסה ראשונית של המשחק, שיפרנו אותו בכך שהוא יודיע למשתמש על כמה שאלות הוא ענה נכון, וכן מה הציון שלו.

כבר במהלך כתיבת תרגיל הטריוויה, נתקלנו במספר קשיים. ראינו שאנו נאלצים לשכפל את הקוד שלנו – לכתוב שורות קוד זהות או דומות מאוד מספר פעמים. כמו כן, ראינו שנדרש טיפול מעמיק וטוב יותר בקלט שמגיע מהמשתמש, כך למשל שגם התשובה "python" תתקבל עבור השאלה " What is your favorite programming language?", ולא רק "Python". עם קשיים אלו נדע להתמודד בהצלחה כבר בפרקים הבאים, בהם אנו צפויים ללמוד עוד על **טיפוסים** בסיסיים בפייתון, נבין בצורה מעמיקה יותר כיצד לכתוב **פונקציות** בפייתון, נדבר יותר על **מחרוזות**, נכיר גם **רשימות** ונבצע דיון מעמיק יותר ב**תכנות נכון**. כמובן שנכתוב עוד הרבה משחקים, ונעבור תהליך מלמד ומהנה.

פרק 6 – סביבת עבודה PyCharm

עד עכשיו השתמשנו ב-IDLE כדי לכתוב את הקוד שלנו. למרות שמדובר בכלי מאוד חזק, לעתים נרצה להשתמש בכלים מתוחכמים יותר. לעיתים קרובות לכתוב קטעי קוד ארוכים, להסתכל על מצב הריצה של הקוד בזמן פועלה, או אפילו פשוט לערוך את הטקסט שלנו בצורה קצת יותר מתוחכמת. לצורך כך, נכיר בפרק הזה את PyCharm, שהוא עורך קוד פייתון. נעבוד איתו במקביל לעבודה עם IDLE, וכשתכתבו את התרגילים תוכלו לעשות בו שימוש.

למידת התוכנה

את הלימוד הראשוני של התוכנה תוכלו לעשות לבד, בעזרת המצגת שנמצאת במיקום הבא:

http://cyber.org.il/networks/1450-2-03.pdf .

עברו עליה שקף אחר שקף עד לשקף 10 (כולל). את התוכן שבשקפים הנוספים נכיר מאוחר יותר. תוכלו להיעזר במצגת תוך כדי פתירת התרגילים הבאים.

תרגיל 6.1 – קסם של פייתון

השתמשו ב-PyCharm על מנת לפתוח תכנית חדשה בשם good_morning, ובה המשתמש נשאל מה שמו ונענה בברכת בוקר טוב. הריצו את התכנית שכתבתם. דוגמה להרצת התכנית:

Hello, what is your name? ***Shay***

Good morning Shay

איזה יופי! הגיע הזמן לגלות עוד סודות וקסמים של PyCharm.

תרגיל 6.2 – לובונגולו מלך זולו

העתיקו את הקוד הבא למסך הקוד שלכם:

lubungulu = 123456

king_of = 84848484

zulu = 2973459432738427 61

כעת, הדפיסו את סכום שלושת המשתנים האלו – עשו זאת בלי להקליד את כל שם המשתנה. קצר, נכון?

תרגיל 6.3 – x מלך זולו

שמרו את הקוד שכתבתם בתרגיל הקודם בתוך פרוייקט. כעת שנו את כל המקומות בהם כתוב lubungulu לשם x. **אל תעשו זאת ידנית!** חפשו את הכפתור ב-PyCharm שמשנה לכם את כל המשתנים בשם הזה.
(רמז: שם הפקודה מתחילה ב-Ref).

תרגיל 6.4 – לעולם בעקבות השמש

הריצו את הקוד הבא באמצעות PyCharm:

```
while True:
    print 'After the sun'
```

מצאו איך לעצור את ריצת הקוד מבלי לסגור את PyCharm.

תרגיל 6.5 – קסם קודר

שנו את סכמת הצבעים של PyCharm ל-Darcula. כעת, שנו לסכמה האהובה עליכם. ואו, צבעוני!

פרק 7 רשימות, לולאות וטיפוסים

בפרק 5 דיברנו על לולאת **while** ועל תנאי **if**. בפרק זה נעסוק בלולאה חדשה, לולאת **for**, ובסוג הבסיסי של מידע שלולאה כזו עוברת עליו - **רשימות**. בנוסף, נלמד איך להבדיל בין סוגים (טיפוסים) שונים של מידע בפייתון.

רשימות

עד כה, למדנו איך לשמור מידע במשתנים, מידע כדוגמת מספר או שם. אבל מה נעשה אם נרצה לשמור כמה סוגים שונים של מידע באותו המקום? לשם כך נצטרך **רשימה**.

רשימה, הקרויה בפייתון **list**, היא אוסף של איברים השמורים. רשימה שומרת על הסדר של האיברים, והיא דינאמית – ניתן להוסיף לה איברים או להוציא ממנה איברים. נגדיר את הרשימה כאוסף של איברים בתוך סוגריים מרובעים:

```
>>> my_list = [1, 2, 3]
>>> another_list = ['a', 'b', 'hello', 99, True, False]
```

בשתי השורות הקודמות הגדרנו שתי רשימות. כפי שניתן לראות, רשימה יכולה להכיל מחרוזות (מילים), מספרים ואפילו ערכים בוליאניים (אמת או שקר). היא יכולה להיות ארוכה ככל שנרצה.

פעולות (מתודות) על רשימות

ניתן לבצע מגוון פעולות על רשימות:

- חיבור רשימות: ניתן לחבר שתי רשימות בעזרת הסימון + או בעזרת הפקודה:extend

```
>>> my_list = [1, 2, 3]
>>> my_list + ['a', 'b']
[1, 2, 3, 'a', 'b']
>>> my_list.extend(['a', 'b'])
```

שאלה למחשבה: מה ההבדל בין שתי הפקודות – **extend** ושימוש ב-**+**?

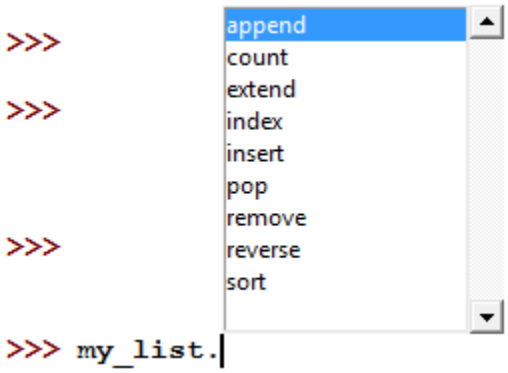
רמז: התבוננו במשתנה my_list לאחר הרצה של כל אחת מהפקודות לעיל. האם המשתנה my_list השתנה אחרי חיבור רשימה נוספת באמצעות **extend**? האם הוא השתנה בעקבות הוספת רשימה באמצעות **+**?

- אפשר להוסיף איבר יחיד על ידי שימוש במתודה **append**:

```
>>> my_list = [1, 2, 3]
>>> my_list.append('a')
>>> my_list
```

[1, 2, 3, 'a']

יש עוד הרבה פעולות על רשימות – תוכלו לגלות אותן, למשל, על ידי כתיבת my_list ואחריה נקודה. יצוץ מסך ובו כל הפעולות האפשריות:



בפרק 9, נכיר דרך נוספת לגלות מה כל המתודות שיש לאובייקט מסויים כגון רשימה, מספר או כל אובייקט שהוא.

תרגיל 7.1 – הרשימה הראשונה שלי

צרו שתי רשימות. רשימה אחת תכיל את המספרים 0 עד 5, והרשימה השניה את האותיות a עד f. חברו אותן לרשימה אחרת, כך שהמספרים יהיו קודם. כלומר, בסוף התרגיל תהיה לכם רשימה שתראה כך:

[0, 1, 2, 3, 4, 5, 'a', 'b', 'c', 'd', 'e', 'f']

תרגיל 7.2 – מקריאים שמות

צרו רשימה עם שמותיהם של חמישה חברים שלכם. סדרו את הרשימה לפי סדר האלף-בית. לשם כך, עליכם להיעזר במתודה של רשימות. לאחר מכן, הוסיפו את שם המורה שלכם בתחילת הרשימה. אם לחבריכם קוראים עומר, שי, ברק, גד ורונן, ואילו המורה שלכם הוא גידי, תיווצר בסוף הרשימה הבאה:

['gidi', 'barak', 'gad', 'omer', 'ronen', 'shay']

לולאת for

יש לנו רשימות, ולכן עכשיו נוכל להכיר את לולאת **for**. לולאה זו מאפשר לנו לעבור על קבוצה של איברים, ולפעול עליהם בזה אחר זה. לעת עתה, נריץ את הלולאה שלנו רק על רשימות, ובהמשך נלמד איך להריץ אותה על מבנים אחרים.

נשים לב שלא חייבים לקרוא לרשימה בשם, וניתן לרוץ ישר על האיברים שלה. כך למשל:

```
>>> numbers = [1, 34, 66, 'three']
>>> for number in numbers:
        print number
```

מה ביצענו כאן בעצם?

הגדרנו רשימה בשם numbers שהכילה ארבעה איברים. לאחר מכן, עברנו על כל איבר ברשימה numbers. לכל איבר כזה קראנו בשם number.

בגוף הלולאה, הדפסנו את המספר הנוכחי.

נשים לב: בגוף הלולאה יש צורך באידנטציה.

בנוסף, השם שבו בחרנו עבור איבר בלולאה, number, הוא שם עבור נוחות בלבד. יכולנו לקרוא לו גם בשם אחר, והיינו מקבלים ממש את אותה התוצאה. כך למשל, השורות הבאות:

```
>>> numbers = [1, 34, 66, 'three']
>>> for shmulik in numbers:
        print shmulik
```

מבצעות את אותה הפעולה בדיוק כמו הלולאה הקודמת בה כל איבר ברשימה נקרא number.

בגוף לולאת **for** ניתן להכניס כל קטע קוד שנרצה: לולאה נוספת, תנאי או סתם הדפסה.

תרגיל 7.3 – מקריאים שמות שוב

הקריאו את רשימת השמות שיצרתם בתרגיל 7.2, כלומר, הדפיסו אותם באמצעות לולאת **for**. הדפיסו את שם המורה עם המחרוזת ': teacher' לפני השם, ולאחר כל השמות הדפיסו את המחרוזת: "all here", כלומר, כולם בכיתה.

range, xrange

לולאת for יכולה לעזור גם בביצוע פעולה כלשהי מספר רב של פעמים. על מנת לעשות זאת, נדאג שבמקום לעבור על רשימה שניתן לה, for תעבור על רשימת מספרים, ובכך "תספור" את כמות הפעולות. כעת נלמד על שתי דרכים לספור, ונשתמש בהן בלולאת for.

הדרך הראשונה היא בעזרת הפונקציה **()range**. זוהי פונקציה שמקבלת מספר, ומחזירה רשימה ובה כל המספרים עד המספר הזה (לא כולל המספר עצמו). כך למשל:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

range יכולה לייצר סוגים נוספים של רשימות מספרים, בקפיצות קבועות ובנקודות התחלה שונות. ראו את הדוגמאות הבאות:

```
>>> range(3, 10)
[3, 4, 5, 6, 7, 8, 9]
>>> range(0, 10, 2)
[0, 2, 4, 6, 8]
>>> range(3, 10, 3)
[3, 6, 9]
```

האם הצלחתם להבין מה התפקיד של כל פרמטר בפונקציה **range**?

חידה: מה תדפיס הפקודה range(10, -10, 4)? חשבו על כך, והריצו את הפקודה בפייתון בכדי לבדוק אם צדקתם.

בדומה לפונקציה **range**, קיימת הפונקציה **xrange**. **xrange** עובדת בדיוק כמו **range**, רק שהיא לא מחזירה רשימה. xrange מחזירה רק **כמות אבסטרקטית** (מדומיינת), שמתפקדת כמו רשימה.

נסתכל על דוגמה בעזרת xrange:

```
for num in xrange(3):
    print num
```

```
0
1
2
```

כמובן שהדוגמה הזו תעבוד גם אם נשתמש ב-**range**, אך מעתה והלאה נשתמש רק ב-**xrange** מכיוון שבדרך כלל, "מאחורי הקלעים" של פייתון xrange תעבוד מהר יותר מ-**range**.

תרגיל 7.4 – המלך אמר לספור עד 40

הדפיסו את כל המספרים מ-1 עד 40 (כולל 40) פעמיים, בעזרת לולאות for. השתמשו פעם אחת ב-range ופעם אחת ב-xrange.

תרגיל 7.5 – זוגות זוגות

הדפיסו את כל המספרים הזוגיים מ-0 עד 50 (כולל 50). במקום המספרים האי-זוגיים, הדפיסו כוכבית.

הרצה של התכנית שלכם תראה כך:

```
0
*
2
*
4
...
```

תרגיל 7.6 – שבע בום!

כולנו מכירים את המשחק שבע בום, ויודעים שקל מאוד להתבלבל בו. כתבו את הפתרון האולטימטיבי – לולאה שמדפיסה את כל המספרים מ-1 עד 100 שמתחלקים ב-7 ללא שארית או שמכילים את הספרה 7, וכך תדעו מה לא להגיד. השתמשו **רק** בפעולות מתמטיות (חישובים) ובלולאות ותנאים שלמדתם עד כה.

(**רמז**: קראו על הפעולה המתמטית מודולו, שמסומנת בפייתון כך - %).

טיפוסים

טיפוסים, כשמם הם, סוגים שונים של מידע. למעשה, כבר הכרנו שלושה טיפוסים שונים מבלי לציין זאת מפורשות: רשימות, מספרים שלמים ומספרים עשרוניים. כעת נכיר את הטיפוסים האלו יותר מקרוב, ואפילו נלמד איך לשנות אותם!

לכל טיפוס יש שם

נשתמש בפקודה **type()** כדי לגלות את סוג הטיפוס של אובייקט מסויים. נשתמש כעת בפקודה זו כדי לגלות את השמות של שלושת הטיפוסים שהכרנו:

```
>>> type(1)
```

```
<class 'int'>
>>> type(1.0)
<class 'float'>
>>> type([1, 2, 3])
<class 'list'>
```

כמו שלמדנו בפרק זה, הטיפוס של רשימות הוא list, וכמו שלמדנו בפרק 5, הטיפוס של מספרים שלמים הוא int והטיפוס של מספרים עשרוניים הוא float. באמצעות הפקודה **type** ביקשנו מפייתון להראות לנו זאת.

מעבר בין טיפוסים

למעשה, כל שם של טיפוס הוא גם פקודה. הפקודה מעבירה מידע מסוג אחד לסוג אחר, כלומר משנה את הטיפוס שלו. נדגים כעת מעבר בין מספר עגול למספר עשרוני:

```
>>> float(1)
1.0
```

השתמשנו בשם הטיפוס של מספר עשרוני (**float**), כדי להפוך את המספר העגול 1 למספר העשרוני 1.0. באותו אופן, הפקודה **int()** תהפוך מספר עשרוני למספר עגול. בתרגיל מודרך 5.14, עשינו זאת כדי להמיר קלט של המשתמש ממחרוזת למספר. כעת אנו מבינים שניתן להשתמש ב**שם הטיפוס** בכדי לבצע המרה בין סוגים שונים של משתנים.

נשים לב כי פעולה שכזו לא תמיד תעבוד. כך למשל:

```
>>> int("Hello")
```

פקודה זו צפויה לגרום לפייתון לצעוק. אחרי הכל, פייתון לא יודע מה המשמעות של להמיר את המחרוזת "Hello" למספר.

אזהרה – לא כדאי להשתמש בשם שמור בתור שם משתנה!

נאמר שברצונינו לשמור מספר אל תוך משתנה. ניתן, עקרונית, לקרוא למשתנה הזה בשם int:

```
>>> int = 6
```

פייתון יאפשר לנו לבצע את הפעולה הזו ללא קושי.

חשבו: מה הבעיה בשורה שכתבנו לעיל?

הבעיה היא שכעת השם "int" אינו עוד הטיפוס int, אלא הוא שם המשתנה שמכיל את המספר 6. כעת, אם ננסה לעשות המרה שכזו:

```
>>> my_number = int(6.0)
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    my_number = int(6.0)
TypeError: 'int' object is not callable
```

פייתון זעק ובצדק – השם "int" אינו עוד משמש בתור פקודה שממירה בין טיפוס למספר שלם, אלא שייך למשתנה מסויים שמכיל את הערך 6. בתור לקח – נמנע מלהשתמש בשמות הטיפוסים בתור שמות משתנים.

תרגיל 7.7 – הלוך ושוב

המירו את המספר 7.7 (המספר של התרגיל הזה) ממספר עשרוני למספר עגול, ואז בחזרה למספר עשרוני. האם קיבלתם את המספר המקורי?

פרק 7 – סיכום

בפרק זה הכרנו **רשימות**. למדנו כי רשימה היא אוסף של איברים, ולמדנו כיצד ליצור אותה, להוסיף לה איברים ולהשתמש בה. בהמשך, למדנו להשתמש בלולאת **for**, הן על מנת לעבור על איברים בתוך רשימה, והן כדי לבצע פעולה מסויימת מספר פעמים. לשם כך, הכרנו את הפונקציות המובנות **()range** ו-**()xrange**.

לאחר מכן, הכרנו את המושג **טיפוס**. למעשה, עבדנו כל הזמן עם טיפוסים – מספרים שלמים, עשרוניים, מחרוזות וגם רשימות. כעת, למדנו שלסוג המשתנה קוראים טיפוס, וכן למדנו שכדי לבצע **המרה** מטיפוס אחד לאחר, נוכל להשתמש בשם הטיפוס.

טיפוסים ורשימות הם חלקים בסיסיים בהכרות שלנו עם פייתון, והם ילוו אותנו במשך כל תהליך הלמידה.

פרק 8 – קונבנציות

בשיעור בכיתה תלמדו על החשיבות העצומה שבתכנות נכון בכלל, ושמירה על **קונבנציות** בפרט. תוכלו למצוא את תקציר הקונבנציות של תכנית "גבהים" בכתובת:

http://www.cyber.org.il/python/conventions.pdf

תקציר הקונבנציות מבוסס על PEP8 [16]. מעתה ואילך, כל קוד שנכתוב יעמוד בקונבנציות אלו ובכללי PEP8. אנו ממליצים לבדוק את הקוד שלכם באמצעות שרותים המיועדים לכך, כגון אתר pep8online.com. כמו כן, נשים לב ש-PyCharm עוזר לנו בכך שהוא "מסמן" מקומות בהם אנו חורגים מהקונבנציות. לדוגמה, ראו את קטע הקוד הבא:

```
if __name__ == '__main__':
    x=1
```

ישנו סימון צהוב מתחת לסימן השווה (=). אם נעמיד עליו את סמן העכבר, PyCharm יסביר לנו שחרגנו מקונבנציות PEP8:

```
if __name__ == '__main__':
    x=1
PEP 8: missing whitespace around operator
```

תרגיל 8.1 – טריוויה איכותית

אתם וודאי זוכרים את תרגיל 5.18, בו כתבנו משחק טריוויה, ואף שיפרנו אותו בכך שהדפסנו את כמות השאלות עליה המשתמשת ענתה נכון, וכן הציון של המשתמשת. הפעם, נשפר את משחק הטריוויה עוד יותר. קחו את הקוד שכתבתם, ושנו אותו כך שיעמוד בכלל הקונבנציות של התכנית.

זכרו לבדוק את הקוד שלכם באמצעות שרותים המיועדים לכך, כגון אתר pep8online.com, וכך תדעו שבאמת עמדתם בכל הקונבנציות הנדרשות.

כתיבה נכונה של קוד תלווה אותנו לאורך תכנית הלימודים, ונחזור לסוגיות נוספות בנוגע לתכנות נכון בהמשך.

[16] https://www.python.org/dev/peps/pep-0008

פרק 9 – מחרוזות

מהי מחרוזת?

בפרקים הקודמים, כבר פגשנו ב**מחרוזות** (באנגלית – **Strings**), סוג של משתנים שמכילים טקסט. למשל, בדוגמה הבאה:

```
>>> my_string = "Hello!"
```

המשתנה my_string הינו מחרוזת. מה הסוג של מחרוזת? איך נוכל לדעת זאת? ובכן, נוכל להשתמש בפונקציה **type** אותה הכרנו קודם לכן. נוכל לעשות כך:

```
>>> type(my_string)
<type 'str'>
```

אם כך, הטיפוס של מחרוזת הוא str (קיצור של string).

אפשר להגדיר מחרוזת באמצעות מרכאות כפולות, כמו בדוגמה למעלה, או באמצעות גרש יחיד, למשל כך:

```
>>> my_string = 'Hello!'
```

למעשה, מחרוזת היא אוסף של תווים. כל תו הוא אות או סימן. כך למשל, במחרוזת my_string, "H" הוא התו הראשון, ו-"e" הוא התו השני. הסימן "!" הוא התו האחרון במחרוזת. אפשר לחשוב על מחרוזת בתור רשימה של תווים, על אף שמחרוזת אינה רשימה מהסוג שאנו מכירים (**list**). אם כך, נוכל למשל לעבור על כל התווים במחרוזת באמצעות לולאה:

```
>>> for character in my_string:
        print character

H
e
l
l
o
!
```

שימו לב – ביקשנו מפייתון לעבור על כל תו במחרוזת, ובגוף הלולאה – הדפסנו את התו. יש כל מיני דברים מעניינים שאפשר לעשות עם מחרוזות. אפשר, למשל, לחבר בין מחרוזות שונות:

```
>>> first_string = "Hello, "
```

```
>>> second_string = "world!"
>>> first_string + second_string
'Hello, world!'
```

באופן דומה, אפשר לבצע השמה עם חיבור:

```
>>> first_string
'Hello, '
>>> first_string += second_string
>>> first_string
'Hello, world!'
```

אפשר גם לבצע "הכפלה" של מחרוזת במספר. מה המשמעות של כך? בואו ננסה:

```
>>> first_string * 3
'Hello, Hello, Hello, '
```

תרגיל 9.1 – שם מרובע

בתרגיל זה, בקשו מהמשתמש לרשום את שמו. לאחר מכן, הדפיסו את שמו ארבע פעמיים, באמצעות קריאה יחידה ל-**print**. שימו לב שעל מנת שהפלט יהיה קריא, עליכם להפריד איכשהו בין כל הדפסה. כך למשל יכולה להיראות הרצה של התכנית שלכם:

```
Please, insert your name --> Gvahim
Gvahim Gvahim Gvahim Gvahim
```

תרגיל 9.2 – שם משוכלל

התרגיל הקודם עזר למשתמש מכיוון שהוא חסך כתיבה חוזרת ונשנית של השם שלו. עם זאת, לא אפשרנו למשתמש לבחור כמה פעמים נדפיס את שמו. בתרגיל זה, עליכם לבקש מהמשתמש גם את שמו, וגם את מספר הפעמים שברצונו שהשם יודפס. לאחר מכן, הדפיסו את השם. **רמז:** הזכרו בפעולת ההמרה (conversion) אותה הזכרנו בפרקים הקודמים.

להלן ריצה לדוגמה של התכנית:

```
Please, insert your name --> Cyber
How many times should I print your name? --> 3
Cyber Cyber Cyber
```

תרגיל 9.3 – שם מטורלל

בתרגיל זה, בקשו את שמו של המשתמש, ולאחר מכן הדפיסו אותו כך שכל אות מופיעה בו פעמיים. כך למשל:

```
Please, insert your name --> Omer
OOmmeerr
```

מחרוזות ארוכות

לעתים, נרצה לכתוב מחרוזות ארוכות – כאלו שמורכבות ממספר שורות. הדרך הנוחה ביותר לעשות זאת, היא להשתמש בשלושה סמני מרכאות, כך:

```
>>> long_string = """Hello,
This is a very long string.
It has three different lines!"""
```

Slicing

לעתים, לא נרצה לעבוד עם כל המחרוזת, אלא רק חלקים ממנה. לשם כך, נוכל להשתמש ב-slicing. הפעולה דומה מאוד לזו שראינו בפרק 7 כשלמדנו על רשימות, ומקבלת שלושה ערכים: מיקום התחלתי, מיקום סופי, וקפיצות.

למיקום של תו בתוך מחרוזת, כמו גם למיקום של איבר בתוך רשימה, נהוג לקרוא **אינדקס (index)**. האינדקסים נספרים החל מ-0. כך למשל, האינדקס של התו "H" במחרוזת לעיל הינו 0, שכן הוא התו הראשון במחרוזת. האינדקס התו השני, "e", הוא 1. מהו התו שנמצא באינדקס התשיעי? נבדוק:

```
>>> my_string[9]
'r'
```

כך ניתן לקבל את התו שנמצא באינדקס מסויים. בדרך כלל, נרצה לעשות דברים מעניינים יותר. נבחן מספר דוגמאות על המחרוזת הבאה:

```
>>> my_string = "Hello, world!"
>>> my_string[0:5]
```

בשורה זו ביקשנו לקבל את המחרוזת מהתו הראשון שלה (התו הראשון הינו במקום ה-0), עד לתו השישי (כלומר התו במיקום החמישי), לא כולל. במילים אחרות, ביקשנו לקבל את חמשת התווים הראשונים במחרוזת. כצפוי, נקבל את הערך הבא:

```
'Hello'
```

פייתון גם מאפשר לנו לא לכתוב את הערך "0" כשאנו מתכוונים אליו, ולכן נוכל לכתוב גם כך:

```
>>> my_string[:5]
```

```
'Hello'
```

כך אפשר לקבל בקלות מספר מסויים של תווים ראשונים מהמחרוזת.

מה יקרה אם נרצה לקבל את שלושת התווים האחרונים במחרוזת? לשם כך, נוח להשתמש ב**אינדקסים שליליים**.

על מנת להקל עלינו, פייתון דואג תמיד שהתו האחרון במחרוזת יהיה בעל האינדקס **-1**. נראה זאת:

```
>>> my_string[-1]
'!'
```

התו השני מהסוף יהיה, כצפוי, בעל האינדקס **-2**:

```
>>> my_string[-2]
'd'
```

נוח מאוד. לכן, אם נרצה לקבל את שלושת התווים הראשונים, נוכל לבקש מפייתון לתת לנו את המחרוזת החל מהתו השלישי מהסוף, ועד לסוף המחרוזת:

```
>>> my_string[-3:]
'ld!'
```

נוכל גם לבקש מפייתון לתת לנו חלק מהמחרוזת שאינו נמצא בתחילת המחרוזת או בסופה. למשל, נבקש מפייתון לתת לנו את המחרוזת החל מהתו השני ועד לתו השישי (כולל):

```
>>> my_string[3:8]
'lo, w'
```

נשים לב כי ציינו בפני פייתון שאנו רוצים להתחיל מהתו באינדקס 3 (זהו התו השני), ולסיים בתו באינדקס 8 (זהו התו השביעי), לא כולל.

כיצד היינו יכולים להשיג את אותו חלק של המחרוזת, על ידי שימוש באינדקס שלילי במקום באינדקס 8? נוכל לעשות זאת כך:

```
>>> my_string[3:-5]
'lo, w'
```

זאת מכיוון שהתו השביעי במחרוזת זו הינו גם התו החמישי מהסוף.

לבסוף, נוכל גם לבקש מפייתון לתת לנו את המחרוזת ב"קפיצות" מסויימות. למשל, נאמר שנרצה לקבל כל תו שני במחרוזת – תו כן, תו לא. נוכל לבקש מפייתון לתת לנו את המחרוזת ב"קפיצות" של 2:

```
>>> my_string[::2]
'Hlo ol!'
```

פייתון נתן לנו למעשה את התו באינדקס 0 ("H"), את התו באינדקס 2 ("l"), וכך הלאה – כלומר התווים באינדקסים הזוגיים בלבד. אם היינו רוצים לקבל את התווים באינדקסים האי-זוגיים, היינו יכולים לבקש מפייתון לתת לנו את המחרוזת ב"קפיצות" של 2, אבל להתחיל מאינדקס 1 במקום 0:

```
>>> my_string[1::2]
'el,wrd'
```

תרגיל 9.4 – Slicer

שימו לב: בתרגיל זה יקל עליכם מאוד לעבוד עם ה-Interperter של פייתון, ולעשות "ניסויים".

כתבו תוכנית שמכילה משתנה מסוג string, בעל הערך: "Hello, my name is Inigo Montoya". השתמשו **רק** ב-slicesים על המחרוזת הזו, והדפיסו את המחרוזות הבאות:

1. "Hello"
2. "my name"
3. "Hlo ynm sIioMnoa"
4. "lo ynm sI"

רגע, אז מה עוד אפשר לעשות עם מחרוזות?

השאלה הזו שאלה מצויינת, והיא פתח לשאלה כללית יותר – יש לנו אובייקט פייתוני כלשהו – רשימה, מספר, מחרוזת או כל דבר אחר – ואנחנו רוצים לדעת מה אפשר לעשות איתו. איך נוכל לענות על השאלה הזו?

אז נוכל להיעזר בידע של כותבי הספר שיכתבו לנו את כל הדברים שאפשר לעשות עם מחרוזות. זה יהיה קל, מהיר ויעיל. מצד שני, איך זה יעזור לנו בפעם הבאה? מה יקרה כשנרצה לדעת מה אפשר לעשות עם רשימה? ואז עם אובייקט שאנחנו עדיין לא מכירים בכלל?

לשם כך, קיימת בפייתון הפונקציה המובנית (והקסומה) **dir**. כשנקרא ל-**dir** על אובייקט כלשהו, נקבל רשימה של כל הפעולות שניתן לבצע עליו. לדוגמא, נוכל לעשות כך:

```
>>> my_string = "Hello, world!"
>>> dir(my_string)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__getslice__', '__gt__',
'__hash__', '__init__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '_formatter_field_name_split',
'_formatter_parser', 'capitalize', 'center', 'count', 'decode', 'encode', 'endswith',
'expandtabs', 'find', 'format', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace',
```

'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']

וואו, קיבלנו ממש רשימה ארוכה. אין צורך כרגע שנבין את המשמעות של כל הפעולות שכתובות כאן, או את ההבדל בין שמות של מתודות שמתחילות ב-"__" לכאלו שלא. כרגע, נתרכז במתודות האחרונות – אלו ששמן לא מתחיל ב-"__".

תרגיל 9.4 – שלום גדול

השתמשו במחרוזת my_string שהגדרנו למעלה, וגרמו לכך שהערך שלה יהיה: "HELLO, WORLD!" – כלומר אותו הערך בדיוק, אך עם אותיות גדולות. לשם כך, עליכם להשתמש במתודה אחת בלבד של מחרוזות. היעזרו ברשימה שמצאתם באמצעות הפקודה **dir** כדי למצוא מתודה מתיאמה.

לעתים מעבר על **dir** אינו מספיק. מה בדיוק כל מתודה שכתובה שם עושה? אילו פרמטרים ניתן להעביר לה? לשם כך, פייתון ציידה אותנו בפונקציה נהדרת נוספת – **help**. באמצעות מתודה זו, ניתן לקבל מידע על כל מתודה. למשל, ברשימה לעיל נמצאת המתודה isupper. מה היא עושה? נוכל לבדוק זאת באמצעות הרצת המתודה **help** על שם המתודה:

```
>>> help(my_string.isupper)
Help on method_descriptor:

isupper(...)
    S.isupper() -> bool

    Return True if all cased characters in S are uppercase and there is
    at least one cased character in S, False otherwise.
```

איזה יופי! פייתון מראה לנו כיצד להשתמש במתודה הזו ומסביר בדיוק מה היא עושה. **isupper** תחזיר ערך בוליאני: **True** במידה שכל האותיות במחרוזת הן upper case (כלומר אותיות גדולות כמו "A" ו-"B"), והמחרוזת מכילה לפחות אות אחת, ואחרת – תחזיר **False**.

שימו לב שהרצנו את **help** על **my_string.isupper**, ולא "סתם" על **isupper**, זאת מכיוון שהשם isupper אינו מוכר לפייתון כשלא מדובר במתודה שרצה על מחרוז מסויימת. לחלופין, ניתן היה גם להריץ את הפקודה כך:

```
>>> help(str.isupper)
```

כלומר, על ידי שימור בשם הטיפוס str.

תרגיל 9.5 – משחק ניחוש מספר

בתרגיל זה תכתבו משחק שבו על השחקן לנחש מספר שבו בחרתם.

ראשית, בחרו מספר סודי כלשהו בין 0 ל-100, והשימו אותו במשתנה:

```
secret_number = 42
```

לשחקן יש עשרה תורות, כאשר בכל תור של השחקן (המשתמש), עליו לנסות ולנחש את המספר הזה. לשם כך, עליכם להציג לו את ההודעה: "Please make a guess". בכל תור, השחקן יציע את הפתרון שלו, ועליכם להציג לו:

- כמה ניחושים הוא ביצע.
- האם הניחוש שלו גבוה מידי, נמוך מידי או פגע בדיוק.

להלן דוגמה לריצה שבה השחקן הצליח לנחש:

```
>>>
Try to guess my number :)
Turn #1
Please make a guess --> 60
Too high...
Turn #2
Please make a guess --> 20
Too low...
Turn #3
Please make a guess --> 50
Too high...
Turn #4
Please make a guess --> 30
Too low...
Turn #5
Please make a guess --> 40
Too low...
Turn #6
Please make a guess --> 42
Great! You won!
```

להלן דוגמה שבה השחקן לא הצליח:

```
>>>
Try to guess my number :)
Turn #1
Please make a guess --> 90
Too high...
Turn #2
Please make a guess --> 10
Too low...
Turn #3
Please make a guess --> 80
Too high...
Turn #4
Please make a guess --> 20
Too low...
Turn #5
Please make a guess --> 70
Too high...
Turn #6
Please make a guess --> 30
Too low...
Turn #7
Please make a guess --> 65
Too high...
Turn #8
Please make a guess --> 60
Too high...
Turn #9
Please make a guess --> 45
Too high...
Turn #10
Please make a guess --> 41
Too low...
No more guesses. Maybe next time!
```

בהמשך, נלמד כיצד להגריל מספר חדש בכל ריצה – וכך נהפוך את המשחק למעניין יותר.

בהצלחה!

תרגיל 9.5 – משחק הגמל

משחק הגמל התפרסם בספר More BASIC Computer Games בשנת 1979. תמונה מהגירסה המקורית ניתן לראות כאן:

http://www.atariarchives.org/morebasicgames/showpage.php?page=24

במשחק זה על השחקנית לרכב על גמל לאורך המדבר, בזמן ששודדים רודפים אחריה. על מנת לנצח, על השחקנית להתחשב במספר דברים:

1) הגמל צמא, ואם לא ניתן לו מים – הוא עלול למות.
2) הגמל עייף, ואם לא נאפשר לו לישון – הוא עלול למות.
3) רודפים אחרינו, וצריך להספיק לצאת מהמדבר לפני שהשודדים יגיעו.

בכל תור, השחקנית יכולה לבחור אחת משש פעולות:

א. להשקות את הגמל במים.
ב. להתקדם במהירות "רגילה".
ג. להתקדם במהירות "מוגברת".
ד. לעצור ולנוח.
ה. לבדוק את מצב הגמל.
ו. לצאת מהמשחק.

להלן דוגמה להרצה של המשחק:

```
Welcome to the camel game!
Your goal is to get out of the desert before the bad guys get you.
Good luck!

        A. Drink from your canteen.
        B. Ahead moderate speed.
        C. Ahead full speed.
        D. Stop for the night.
        E. Status check.
        Q. Quit.
What's your choice? --> c
You walked 12 kilometers

        A. Drink from your canteen.
        B. Ahead moderate speed.
        C. Ahead full speed.
        D. Stop for the night.
        E. Status check.
        Q. Quit.
What's your choice? --> c
You walked 12 kilometers
Warning! Your camel is getting tired!

        A. Drink from your canteen.
        B. Ahead moderate speed.
        C. Ahead full speed.
        D. Stop for the night.
        E. Status check.
        Q. Quit.
What's your choice? --> e
Kilometers traveled: 24
Drinks in canteen: 5
The bad guys are 24 kilometers behind you.
Warning! Your camel is getting tired!

        A. Drink from your canteen.
        B. Ahead moderate speed.
        C. Ahead full speed.
        D. Stop for the night.
        E. Status check.
        Q. Quit.
What's your choice? --> gvahim
Invalid choice! Please choose again.
Warning! Your camel is getting tired!

        A. Drink from your canteen.
        B. Ahead moderate speed.
        C. Ahead full speed.
        D. Stop for the night.
        E. Status check.
        Q. Quit.
What's your choice? --> b
You walked 8 kilometers
Warning! Your camel is getting tired!
Your camel died out of exhaustion. Game over :(
```

נקודות חשובות להתחשב בהן:

- יתכן שהמשתמשת תקיש אות קטנה במקום אות גדולה, למשל "a" ולא "A". במקרה זה, עליכם לקבל את התשובה בתור "A".
- יתכן שהמשתמשת תקיש אפשרות לא נכונה. במקרה זה, עליכם להדפיס הודעת שגיאה ולבקש ממנה שוב לבחור.

הנחיות

חשבו על התהליך במהלך כתיבת הקוד. אילו חלקים יהיו בו? נסו לכתוב את הקוד לכל חלק בנפרד, ואז לבדוק שהקוד עובד בטרם תמשיכו. למשל, תוכלו לעבוד כך:

- הדפיסו למסך את ההודעה שמכילה את ההנחיות למשחק.
- על מנת להמשיך לבקש מהמשתמשת לבחור פעולה, השתמשו בלולאת **while**.
    - חשבו – כיצד תצאו מהלולאה בשלב הנכון?
- בתוך הלולאה, הדפיסו את הודעת הבחירה.
- בקשו מהמשתמשת להכניס את הבחירה שלה.
    - בשלב זה, תוכלו גם לנסות להתמודד עם קלט שונה מהצפוי ("a" במקום "A") או שגוי (למשל "hello").
- וודאו שאתם מצליחים לצאת מהלולאה אם המשתמשת בחרה ב-"Q".
- ביחרו קבועים שמתקיימים במהלך המשחק, ושמרו אותם במשתנים (קבועים):
    - כמה "מנות" מים יש בהתחלה?
    - כמה קילומטרים הגמל מתקדם בקצב רגיל?
    - כמה קילומטרים הגמל מתקדם בקצב מהיר?
    - כמה הגמל מתעייף לאחר הליכה בקצב רגיל או מהיר?
    - מה המרחק הראשוני של השודדים מהשחקנים?
    - כמה השודדים מתרחקים מידי יום?
    - מהי רמת הצמא שבה הדמות מתה מצמא?
    - מהי רמת העייפות שבה הגמל מת מעייפות?
    - מה האורך הכולל (בקילומטרים) של המדבר?
    - ועוד...

- אתחלו כמה משתנים שישמרו על המצב ברגע מסויים:
    - כמה מנות מים נותרו?
    - עד כמה הדמות עייפה?
    - עד כמה הגמל צמא?
    - כמה קילומטרים הדמות התקדמה עד כה?
    - עד כמה השודדים התקדמו עד כה?

- כעת, תוכלו לממש את הפעולה הבסיסית ביותר – בקשת סטאטוס ("E").
    - אם המשתמשת מבקשת סטאטוס, הדפיסו לה הודעה מתאימה כמו בהודעה לעיל.
    - אם המשתמשת בוחרת ב-"Q", צאו מהמשחק.
    - אחרת, המשיכו לבקש קלט תקין והדפיסו הודעת שגיאה.

- באופן הדרגתי, הוסיפו טיפול בבחירות נוספות של המשתמשת.

בהצלחה!

פרק 9 – סיכום

בפרק זה הכרנו אבן יסוד נוספת בפייתון – **מחרוזות**. למדנו לעבור על מחרוזת כמו על רשימה, לחבר מחרוזות ו"להכפיל" מחרוזות. לאחר מכן, למדנו על מושג ה**אינדקס**, וכן כיצד לבצע **slicing** ולקחת חלק מסויים מתוך מחרוזת שנתונה לנו.

לאחר מכן, הכרנו את הפונקציה הקסומה **dir**, וכן את הפונקציה **help**, שיעזרו לנו להבין איך להתמודד עם כל אובייקט פייתון, מוכר או חדש.

לקראת סוף הפרק, שילבנו את הכלים החדשים שרכשנו בפרקים האחרונים לכתיבת שני משחקים משמעותיים – ניחוש מספר, ומשחק הגמל.

בשלב זה אנחנו כבר יודעים לכתוב לא מעט קוד פייתון! עם זאת, יש הרבה לאן להתקדם. עדיין לא למדנו להגדיר פונקציות משלנו, כמו שלמדנו לעשות עם Guido. ישנם מבנים נוספים בשפה, בנוסף לרשימות ומחרוזות, שיוכלו לעזור לנו מאוד בכתיבת קוד מורכב. כמו כן, עדיין לא למדנו כיצד לבדוק האם הקוד שלנו עובד.

כל זאת ועוד נלמד בפרקים הבאים.