python™

זוהר זילברמן

מטה הסייבר הצה"לי

www.cyber.org.il

פייתון 3.8

זהר זילברמן

גרסה 1.21

הודפס בישראל.

http://www.cyber.org.il

תוכן עניינים

חלק 1: מבוא

Python היא שפת Scripting נוחה לשימוש שהפכה למאוד פופולרית בשנים האחרונות. Python נוצרה בשנות ה-90 המוקדמות ע"י חידו ואן רוסם (Guido van Rossum) ב-Stichting Mathematisch Centrum, כשפת המשך לשפה שנקראה ABC. בשנת 1995 גידו המשיך את פיתוח השפה בחברה ששמה Corporation for National Research Initiatives (CNRI), ואף הוציא מספר גירסאות של השפה. במאי 2000 גידו וצוות הפיתוח של Python עברו ל-BeOpen.com כדי לייסד את "BeOpen Python Labs team". באוקטובר 2000 בצוות עבר ל-Digital Creations, וב-2001 נוסד Python Software Foundation (PSF), שהוא ארגון ללא מטרות רווח. Digital Creations היא אחת הממנות של PSF. כל מה שקשור לPython- הוא Open Source, כלומר הקוד פתוח ונגיש לכולם, ולכל אחד מותר לבצע בו שינויים ולהפיצו מחדש. אתר הבית של PSF הוא http://www.python.org/psf/.

כאשר אומרים "שפת Scripting" מתכוונים שאין צורך בהידור (Compilation) וקישור (Linkage) של תוכניות. ב-Python פשוט כותבים ומריצים. כל סביבת עבודה של Python מגיעה עם Interpreter שמאפשר כתיבה ישירה של פקודות אליו, אפילו בלי צורך לכתוב תוכנית או קובץ. בצורה כזו מאוד נוח להתנסות בשפה, לבצע חישובים מהירים (עוד תכונה חזקה של Python) ולבדוק דברים קטנים במהירות, בלי הצורך לכתוב תוכנית שלמה.

ל Python יתרונות רבים על פני שפות Scripting אחרות. ראשית, Python פשוטה בהרבה ללימוד, לקריאה ולכתיבה משפות אחרות (שחלקן מתעללות במתכנת ובמי שצריך לקרוא קוד כתוב). Python גורמת לקוד שכתוב בה להיראות מסודר ופשוט יותר בגלל שהיא מבטלת סיבוכים מיותרים שקיימים בשפות אחרות.

שנית, Python מכילה בתוכה אוסף מכובד מאד של ספריות **סטנדרטיות**, כלומר ספריות שניתן להסתמך על קיומן בכל מימוש של Python. בין הכלים השימושיים ניתן למצוא יכולות התמודדות עם קבצים דחוסים (zip), תכנות לתקשורת TCP/IP, כלים מרובים לתכנות בסביבת האינטרנט, ממשק משתמש גרפי (GUI), ספריות לטיפול בטקסט ומחרוזות (כמו כן עיבוד טקסטואלי למיניו), עבודה עם פונקציות מתקדמות של מערכת ההפעלה, תמיכה בתכנות מבוזר (Distributed Development), יכולת טיפול אינטנסיבית בקבצי XML ונגזרותיהם, ועוד הרבה הרבה הרבה דברים! כמו כן Python מטפלת במידע בינארי (מידע שאינו טקסטואלי) בצורה הרבה יותר טובה משפות מתחרות.

בנוסף, כמו שניתן לכתוב תוכניות קטנות ב-Python, קיימים בה כלים לכתיבת תוכניות גדולות, או מודולים חיצוניים. Python תומכת בתכנות מכוון עצמים (OOP), Exceptions, מבני נתונים (Data Structures) וקישוריות גם ל-C, או כל שפה שניתן לקרוא לה מ-C.

דבר נוסף שכדאי לדעת על Python הוא שיש לה מספר גרסאות. אמנם יש תאימות לאחור, אבל בגרסאות החדשות (2.5 ומעלה) יש הרבה תכונות חדשות ונוחות לשימוש. בחוברת זו נלמד את גרסה 2.7.

על Python-3

שימו לב! הספר נכתב במקור לפייתון 2.7

הספר נערך בחלקים שהתבקשתם ללמוד כדי שיתאים לפייתון 3.8

Python לעומת C

אם אתם מכירים C אז אתם בטח שואלים את עצמכם למה העולם עדיין צריך את C אם ה-Python הזאת היא השפה הכי מגניבה ביקום שמאפשרת לעשות כל דבר בקלות ובמהירות, בלי להסתבך עם קומפיילרים ולינקרים ושמאפשרת אפילו לא לכתוב את התוכניות בקבצים אם רוצים. אז זהו, ש-C עדיין נחוצה. Python היא לא סתם שפה, היא סביבת עבודה שלמה, וסביבת העבודה הזאת די איטית בהשוואה ל-C. יש הרבה מקרים שבהם לא אכפת לנו ש-Python יותר איטית מ-C: נוכל למשל לפתח בה כלים שרצים חלק זניח מהזמן ובשאר הזמן מחכים לקלט מהמשתמש (ואז אין סיכוי שהמשתמש ישים לב שזה Python או C), אבל אם נרצה לכתוב תוכנית שמבצעת הרבה חישובים נגלה ש-Python לעומת C זה כמו חמור זקן ועייף לעומת סוס מירוצים צעיר ושמח.

הפעלת Python

לפני שנמשיך, וודאו שאתם יודעים איך להפעיל את Python:

כיום, Python מופצת יחד עם כל ה-Distribution-ים הפופולריים של Linux (Ubuntu ,Redhat) אז כל מי שמריץ לינוקס בבית לא צריך להוריד כלום. משתמשי Windows צריכים להוריד סביבות פיתוח של Python ל-Windows היישר מ-http://www.python.org/.

כדי להפעיל את Python בלינוקס, פשוט מקלידים את הפקודה python ב-console ומקבלים את ה-prompt של Python:

```
$ python
Python 3.8 (default, Apr 20 2012, 22:39:59)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

במהלך כל החוברת, הדוגמאות שיירשמו יילקחו מסביבת העבודה בלינוקס, אבל אין שום הבדל בין שתי סביבות העבודה.

ה-Interpreter ומאחורי הקלעים

אחרי שהפעלנו את Python, הגיע הזמן להתחיל לשחק איתה קצת... בהנחה שהפעלת את התוכנה הנכונה, על המסך מופיע ה-prompt של Python – סימן של שלושה חיצים:

```
>>>
```

ב-prompt הזה ניתן למעשה לכתוב כל פקודת Python, והיא תורץ כל עוד היא חוקית ותקינה.

אבל הדבר הראשון שננסה הוא לא פקודה, אלא ביטוי. נרשום את הביטוי 1+1:

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

הביטוי נכון, אבל זה לא ממש מרשים. אמרנו ש-Python יש יכולות חישוב מהירות. בואו ננסה משהו קצת יותר מסובך:

```
>>> 2 ** 2 ** 2 ** 2 ** 2
```

את התוצאה כבר לא נרשום כאן... אם הרצת את השורה האחרונה על המחשב שלך, כנראה לקחו לו כמה שניות להגיע לתוצאה, אבל הוא כתב על המסך ממש הרבה מספרים. מה שעשינו בשורה האחרונה היה:

$$2^{2^{2^{2^{2}}}}$$

וכמו שבוודאי הבנת, ** זה חזקה.

Python יודע גם לטפל במחרוזות: מחרוזות נכתבות כרצף של תווים בין שני תווים של גרש בודד ('), או בין שני תווים של גרשיים ("). כמובן, אי-אפשר לשלב (מחרוזת שנפתחת בגרש אחד ונסגרת בגרשיים).

```
>>> 'Hello, World!'
'Hello, World!'
```

עוד על מחרוזות וכל הדברים שניתן לעשות בהן, בפרק הבא.

הדבר האחרון שכדאי להכיר בהקשר של ה-Interpreter הוא _. סימן ה _ (Underscore) ב-Python מציין את התוצאה האחרונה שחושבה ע"י ה-Interpreter. כך למשל:

```
>>> 1+1
2
>>> _+2
4
>>> _
4
>>> _**2
16
```

Underscore לא קיים בתוכניות רגילות והוא משמש אותנו כאשר נבדוק דברים נקודתיים ב-Interpreter.

משתנים והשמות (Assignments)

כמו ברוב שפות-התכנות, גם ב-Python יש משתנים. אבל, שלא כמו בכל שפת תכנות, ב-Python אין צורך להכריז על משתנים או על הסוג שלהם.

ב-Python, משתנה נוצר כאשר מציבים לתוכו ערך, וסוג המשתנה נקבע לפי הסוג של אותו ערך התחלתי. דוגמה:

```
>>> x=5
>>> type(x)
<type 'int'>
```

בדוגמה הזאת נוצר משתנה בשם x, וערכו ההתחלתי הוא 5. כיוון ש-5 הוא מספר שלם (יש לשים לב להבדל בין 5 ל-5.0, כמו ב-C) סוג המשתנה הוא int.

את סוג המשתנה מקבלים באמצעות הפונקצייה המובנית type, המקבלת בסוגריים את המשתנה ומחזירה את הסוג שלו. את נושא ה"סוג" של משתנים ומה בדיוק זה אומר נסקור בהמשך.

כאשר מציבים למשתנה ערך חדש, גם סוג המשתנה מתעדכן בהתאם. אם x היה int עד עכשיו, ונציב לתוכו ערך חדש, סוג המשתנה ישתנה בהתאם לערך החדש:

```
>>> x=5.0
>>> type(x)
<type 'float'>
```

ועכשיו הסוג של x הוא float.

אילוצי שמות המשתנים עובדים כמו ב-C (שם משתנה חייב להתחיל באות או קו-תחתון ויכול להכיל אותיות, מספרים וקווים תחתונים):

```
>>> x = 5
>>> y = x
>>> y
5
>>> third_var = x
```

ואפילו ניתן ליצור השמות מרובות בשורה אחת:

```
>>> x, y = 17, 42
>>> x
17
>>> y
42
```

בנוסף, כדי להדפיס ערך של משתנה, ניתן להשתמש בפקודה print (זה כבר שימושי בתוך תוכניות, לא ב-Interpreter):

```
>>> print(x)
17
```

נקודה נוספת שיש לציין היא ש-Python היא Case-Sensitive, כלומר יש הבדל בין משתנה בשם a לבין משתנה בשם A – אלו הם שני משתנים שונים, משום שהאות A איננה האות a. חוקים אלה הם כמו החוקים של C.

בנוסף לטיפוסים המוכרים ב-C, ב-Python יש מספר טיפוסים מובנים (Built In) נוספים (את כולם נסקור בסעיף הבא). אחד מהם הוא רשימה. יצירה רשימה נעשית באופן הבא:

```
>>> l = [1, 2, 3]
```

כעת יש רשימה בשם l (האות L קטנה) שמכילה את המספרים 1, 2, 3. האינדקס של האיבר הראשון הוא אפס, וניתן להתייחס לרשימה כולה או לאיבר בודד בה:

```
>>> l
[1, 2, 3]
>>> l[0]
1
```

סקירה מלאה של הרשימה ושאר הטיפוסים המובנים נמצאת בפרק הבא.

מה עושה הסימן =

המשתנים ב-Python הם לא יותר מאשר מצביעים – כל משתנה הוא מצביע לאובייקט. לכל אובייקט יש סוג (אובייקט של מספר, אובייקט של מחרוזת, וכד'). משום שמשתנה הוא מצביע, נוח לשנות "ערכים" של משתנים – רק משנים את העצם אליו המשתנה מצביע, וה"ערך" שלו משתנה.

כאשר משתנה מצביע לעצם, ויוצרים השמה למשתנה אחר (ע"י x=y), יוצרים הצבעה חדשה אל העצם, ולא עצם חדש. כאשר משנים את העצם המקורי, כל מי שהצביע אליו מושפע. בדוגמה הבאה ניצור רשימה, וננסה להשים אותה למשתנה אחר:

```
>>> x = [1, 3, 7]
>>> y = x
>>> x
[1, 3, 7]
>>> y
[1, 3, 7]
>>> x[1] = 6
>>> x
[1, 6, 7]
>>> y
[1, 6, 7]
```

כפי שניתן לראות, יצרנו רשימה במשתנה x, והשמנו אותה למשתנה y. כתוצאה מכך, לא נוצרה רשימה חדשה, אלא הצבעה לרשימה הקיימת. כאשר שינינו את הרשימה הקיימת, כל מי שהצביע אליה הושפע מכך.

בהמשך נראה אילו דברים טובים אפשר לעשות בעזרת ההצבעות האלה, וגם אילו בעיות הן יכולות ליצור אם לא משתמשים בהן כמו שצריך.

טיפוסי נתונים ואופרטורים

מספרים שלמים

אז נתחיל ממספרים שלמים, הטיפוס הפשוט ביותר. הטיפוס של מספר שלם הוא int (קיצור של Integer) והם מאוד דומים למספרים שלמים ב-C. פייתון מייצרת מספר שלם כשאנחנו כותבים רק ספרות בלי שום תוספות (שבר עשרוני, או אותיות בעלות משמעות, אותן נראה בהמשך):

```
>>> x = 34
>>> type(x)
<type 'int'>
```

קיבלנו את המשתנה x מסוג int.

מספרים עשרוניים

אחיו החורג של המספר השלם הוא המספר העשרוני. ייצוג המספר העשרוני שונה ממספר שלם, ולכן חלות עליו מגבלות מסוימות. גם למספר עשרוני יש גבולות (הגבולות תלויים במימוש של המספר העשרוני – האם זה מספר עשרוני של 32-ביט או של 64-ביט), אבל אין לו טיפוס מקביל "ארוך" כמו למספר השלם.

יצירת משתנה עשרוני היא כמו ב-C, ע"י כתיבת מספר עם נקודה עשרונית ושבר (אפשר שבר 0, רק כדי להצהיר שזה מספר עשרוני).

```
>>> 21.0
21.0
```

כמו כן, כאשר משלבים בחישוב מסוים מספר עשרוני, התוצאה הופכת אוטומטית להיות עשרונית:

```
>>> 1.0 / 2
0.5
>>> 1 + 1.0
2.0
```

כאשר משלבים בחישוב מספר ארוך ומספר עשרוני יכול להיווצר מקרה בו התוצאה גדולה מדי כדי להיות מיוצגת בצורה עשרונית. אם ניתן לחשב את התוצאה ולאחסן אותה, לא תהיה שום בעיה:

```
>>> 21/7.0
3.0
```

אבל אם לוקחים מספר ממש ממש ארוך ומבצעים חישוב עם מספר עשרוני, תיווצר שגיאה:

```
>>> (2**2**2**2**2)/3.0
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
OverflowError: long int too large to convert to float
```

חשוב לשים לב שלפעמים מספרים עשרוניים מאבדים את הספרות התחתונות של המספר בלי שנוצרת אף שגיאה:

```
>>> 100000000000000000000000000000000000000000000000001.0
1e+50
```

ה-e זאת דרך קצרה להגיד "10 בחזקת" (במקרה הזה 10 בחזקת 50).

הסיבה להתנהגות הזו היא שמספר עשרוני מוחזק בצורה שבה יהיה יעיל לעשות עליו חישובים, אבל הוא לפעמים מאבד קצת ספרות תחתונות, וזה בסדר אם אנחנו יודעים מה אנחנו עושים (כלומר, לא נשתמש במספרים עשרוניים אלא אם נדע שלא אכפת לנו לאבד את החלק התחתון של המספר).

בסיסי ספירה- אם אינכם מכירים, חשוב מאוד ,שתקראו על כך *

בסיסי ספירה ב-Python נכתבים ומתנהגים כמו ב-C: כאשר רוצים לייצג מספר בבסיס 16 (Hexadecimal, או בקיצור Hex) משתמשים בקידומת 0x:

```
>>> 0xBEE
3054
```

Python כמובן תציג את המספר בייצוג עשרוני, וזוהי דרך מצויינת להמיר מבסיס לבסיס ממש מהר. אפשר לעשות גם את הפעולה ההפוכה, בעזרת הפונקצייה המובנית hex() שמקבלת מספר ומחזירה מחרוזת המכילה את המספר בייצוג הקסהדצימלי:

```
>>> 0xBEE
3054
>>> hex(_)
'0xbee'
```

ייצוג מספרים בבסיס 8 (Octal) נעשה ע"י הוספת אפס לפני המספר אותו רוצים לייצג (כמו ב-C) או ע"י הוספת 0o (אפס ואז האות o):

* אם אינכם מכירים בסיסי ספירה, תוכלו לדלג על הסעיף או לקרוא עליהם ב-Wikipedia (חפשו את הערך "בסיסי ספירה" ב-http://he.wikipedia.org/). מומלץ לקרוא ב-Wikipedia, אל תדלגו, זה נורא מעניין!

```
>>> 045
37
>>> 0o45
37
```

כמו הפונקצייה ()hex, קיימת פונקצייה מובנית בשם ()oct שממירה מספר דצימלי למחרוזת בייצוג אוקטלי:

```
>>> oct(45)
'055'
```

ולבסוף, ניתן להציג מספרים בבסיס 2, הלא הוא הבסיס הבינארי:

```
>>> 0b100101
37
>>> bin(_)
'0b100101'
```

פעולות חשבוניות על מספרים

חיבור (+), חיסור (-) וכפל (*) פועלים כמו בשפת C. כלומר, כאשר יש פעולה בין 2 אופרנדים זהים, התוצאה היא מהסוג של האופרנדים. כאשר האופרנדים לא זהים, התוצאה תהיה מהטיפוס ה"יותר מתקדם": בכל פעולה בה מעורב מספר עשרוני התוצאה תהיה עשרונית. אם בפעולה לא מעורב משתנה עשרוני, אבל מעורב משתנה ארוך, התוצאה תהיה ארוכה.

חילוק (/) גם פועל כמו ב-C, וכאשר ישנם טיפוסי נתונים שונים שמעורבים בחישוב, הכלל לגבי חיבור חיסור וכפל קובע גם כאן. כאשר יש חילוק בין שלמים, התוצאה תמיד שלמה. לעומת זאת, אם רוצים תוצאה עשרונית, חייב להיות מעורב מספר עשרוני.

אם רוצים לאלץ את Python לייצר תוצאה עשרונית, אבל משתמשים רק במשתנים שלמים, ניתן להשתמש בפונקצייה ()float כדי להמיר את אחד המשתנים השלמים למספר עשרוני, ועל-ידי כך לגרום לתוצאה להיות עשרונית:

```
>>> x = 8
>>> y = 3
>>> x / y
2
>>> float(x) / y
2.6666666666666665
```

כמובן, כדי לקבל תוצאה עשרונית נכונה בדוגמה האחרונה, לא ניתן לעשות את הדבר הבא:

```
>>> float(x / y)
2.0
```

במקרה כזה קודם x/y מחושב, ורק אז הוא מומר למשתנה עשרוני.

מודולו (%) לא מתנהג כמו בשפת C, הוא קצת יותר משוכלל. אופרטור המודולו יכול לקבל מספרים שלמים רגילים, ארוכים וגם משתנים עשרוניים. התוצאה במקרים בהם המספרים שלמים וארוכים ברורה, היא מחזירה את השארית של החלוקה בין שני המספרים, והטיפוס (כמו בכל האופרטורים האחרים) יהיה לפי הכלל של חיבור וחיסור.

במקרה בו אחד האופרטורים הוא משתנה עשרוני, התוצאה מחושבת כמספר שלם (כלומר, מחושב המודולו של שני הפרמטרים, והתוצאה הנה בצורת מספר עשרוני).

דוגמה להתנהגות זו:

```
>>> 3.0 % 6.0
3.0
>>> 6.0 % 3.0
0.0
>>> 2.5 % 3.5
2.5
```

בדוגמה הראשונה חושב המודולו של המספרים 3 ו-6, כי הם בסך-הכל שלמים שמיוצגים בצורה עשרונית. בדוגמה השניה חושב המודולו ההפוך (והתוצאה בהתאם).

בדוגמה השלישית לעומת זאת, חושבה השארית של 2.5 לחלק ל-3.5.

בוליאנים

הטיפוס הבוליאני הוא עוד אח חורג של המספרים השלמים. טיפוס בוליאני יכול לקבל רק את הערכים True או False, ובהרבה פונקציות של פיתון (וכמובן גם בפונקציות שניצור בעצמנו, כמו שעוד נראה בהמשך) מקובל להחזיר ערכים אלה כאשר הערך יכול להיות "כן" או "לא".

לדוגמה:

```
>>> 5 == 6
False
```

ניסינו לבדוק האם 5 שווה ל-6 וקיבלנו את התשובה "לא". האופרטור == בדוגמה שלנו משווה בין שני מספרים ומחזיר True או False בהתאם לערכי המספרים. את == נפגוש הרבה בהמשך ונשווה באמצעותו גם דברים שאינם מספרים.

הסיבה שבוליאנים דומים למספרים היא שפיתון יודעת לתרגם, במידת הצורך, בין בוליאנים למספרים שלמים רגילים. True מתורגם ל-1, False מתורגם ל-0, והתרגום מתבצע רק כאשר מנסים לבצע פעולות חשבוניות על בוליאנים, למשל:

```
>>> True + 1
2
>>> 0 * True
0
>>> False * True
0
>>> False + True
1
>>> True/False
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

הדוגמה האחרונה מנסה לחלק ב-False, שהוא למעשה חילוק ב-0, וכמובן אסור.

מחרוזות

מחרוזות ב-Python מיוצגות בצורה הפוכה מ-C: ב-C, מחרוזת היא אוסף של תווים. ב-Python, מחרוזת היא טיפוס נתונים מובנה, ותו הוא מחרוזת באורך 1. ייצוג זה מקל מאוד על העבודה עם מחרוזות – מה שמצריך ב-C שימוש בפונקציות של הספריה <string.h> מצריך כאן שורת קוד אחת קצרצרה.

יצירת מחרוזת תיעשה באחת מהצורות הבאות:

```
>>> str = "Look how easy!"
>>> str
'Look how easy!'
>>> rts = 'Look how easy!'
>>> rts
'Look how easy!'
```

כמו שניתן לראות, מחרוזות ניתן ליצור בשתי צורות: המחרוזת נמצאת בין 2 תווים של גרשיים או בין 2 תווים של גרש בודד. כמובן, אם מחרוזת נמצאת בין 2 תווים של גרשיים, ניתן לכלול בה גרש בודד (בלי\' '), ולהפך. כמובן, אפשר להשתמש בתו ה-'\' כדי לכלול את התווים האלה במחרוזות:

```
>>> "It's a rainy day"
"It's a rainy day"
>>> 'It\'s another rainy day'
"It's another rainy day"
>>> "It's a 42\" display"
'It\'s a 42" display'
```

כמו כן, כל הצירופים של `'\'` ותו אחריהם מ-C תקפים ב-Python: `'\n'` לשורה חדשה, `'\r'` לתחילת השורה, `'\b'` לחזור תו אחד אחורה, `'\t'` עבור טאב, וכו'...

כמו כן, ב-Python יש מנגנון מובנה לפירמוט (Formatting) של מחרוזות (מה שעושה ()printf). זה נעשה ע"י כתיבת כל ה-% המוכרים כבר מ-C לתוך המחרוזת וכתיבת כל הפרמטרים אחרי המחרוזת:

```
>>> "I am %d years old, and my cat's name is %s." % (17, 'Shmulik')
"I am 17 years old, and my cat's name is Shmulik."
```

בפועל, אחרי המחרוזת יש את התו '%', ואחריו אנחנו מעבירים Tuple (הטיפוס יוצג בהמשך פרק זה) של כל הפרמטרים, לפי הסדר. אם מעבירים מספר לא מדויק של פרמטרים (למשל, רשמנו רק d% במחרוזת, והעברנו שני פרמטרים), יש שגיאה. כמו כן, אם מעבירים פרמטר שהוא לא מהסוג שהעברנו ב-% (מחרוזת עבור d%, וכו'), גם יש שגיאה.

פירמוט המחרוזות של Python הרבה יותר מוגן ונוח מזה של C, הוא לא מצריך קריאה לפונקציה, וניתן לשלב אותו בכל מקום, אפילו בתוך קריאה לפונקציה (בפרמטר, במקום להכניס סתם מחרוזת רגילה, מכניסים מחרוזת מפורמטת):

אין צורך להבהל מהדוגמה, פונקציות יוצגו בפרק הבא...

```
>>> def func(s):
...     print(s*5)
...
>>> func("boker%s" % ("tov!"))
bokertov!bokertov!bokertov!bokertov!bokertov!
```

כעת נסקור כמה פונקציות שימושיות מאוד. חלק מפונקציות אלה מובנות ב-Python עצמה, וחלקן של טיפוס המחרוזת.

()chr מקבלת כקלט מספר המייצג את ערך ה-ASCII של תו מסוים, ומחזירה מחרוזת ובה תו אחד שערך ה-ASCII שלו הוא הערך שקיבלה הפונקציה כפרמטר:

```
>>> chr(97)
'a'
>>> chr(65)
'A'
```

()str לעומתה מקבלת משתנה ומנסה להמיר אותו למחרוזת. הפונקציה מסוגלת לקבל יותר מטיפוסים בסיסיים, אפילו רשימות ומילונים (נראה בהמשך):

```
>>> str(1234)
'1234'
>>> str([1,2,3])
'[1, 2, 3]'
```

הפונקציה ()len מחזירה את אורך המחרוזת:

```
>>> len('Hello')
5
```

ומיד נראה ש-()len פועלת על טיפוסים אחרים ולא רק על מחרוזות.

הפונקציה ()split מקבלת מחרוזת ותו ומפצלת את המחרוזת לרשימה, כאשר התו שהיא קיבלה מפריד בין האיברים ברשימת הפלט. לדוגמה:

```
>>> 'look at me'.split(' ')
['look', 'at', 'me']
```

הפונקציה ()join, לעומתה, עושה בדיוק ההפך – היא מקבלת רשימה, ומופעלת על מחרוזת. הפלט הוא מחרוזת המורכבת מכל איברי הרשימה, והמחרוזת הנתונה ביניהם:

```
>>> ' '.join(['Hello,', 'world', 'I', 'am', 'a', 'joined', 'string'])
'Hello, world I am a joined string'
```

דבר שלא ניתן לעשות עם מחרוזות הוא לשנות את ערכו של תו בודד. הסיבה לכך נעוצה במימוש של Python, שכמו שכבר ציינו בנוי ממצביעים. מחרוזת היא מצביע לעצם שמכיל את המחרוזת. הביטוי הבא:

```
>>> s = 'ugifletzet'
>>> s[1]
'g'
```

מחזיר תת-מחרוזת שמכילה רק תו אחד במחרוזת, ולא מאפשר השמה לערך זה. אם נרצה לשנות את המחרוזת, ניאלץ להזין את המשתנה בערך חדש לגמרי.

בסעיף על רשימות נראה שבאמצעות Slicing ניתן לקבל תת-מחרוזת (ולא רק תו אחד).

בנוסף, קיימות במחרוזות פונקציות לשינוי ה-Capitalization של מחרוזות, כלומר כל נושא האותיות הקטנות והגדולות באנגלית. הפונקציה ()capitalize מקבלת מחרוזת ומחזירה את אותה המחרוזת, כאשר האות הראשונה במילה הראשונה מתחילה באות גדולה והשאר קטנות.

הפונקציה ()upper מחזירה מחרוזת בה כל האותיות גדולות. הפונקציה ()lower עושה אותו הדבר, רק מחזירה מחרוזת בה כל האותיות קטנות.

strip מסירה מהמחרוזת את תווי ה-whitespace שנמצאים בקצוות המחרוזת. למשל, הנה המחרוזת הבאה אחרי strip:

```
>>> s = '  power rangers   \t  '
>>> s.strip()
'power rangers'
```

הפונקציה index תת-מחרוזת כפרמטר, ומחזירה את המקום הראשון בו תת-המחרוזת מופיעה בתוך המחרוזת עליה הרצנו את index. אם לא נמצא מקום כזה, הפונקציה זורקת שגיאה (יילמד בהמשך בפרק על Exceptions). הפונקציה find עושה בדיוק אותו הדבר, אבל אם היא לא מוצאת את תת-המחרוזת היא מחזירה 1-:

```
>>> 'Hello'.index('H')
0
>>> 'Hello'.find('b')
-1
```

מחרוזות Unicode

דבר נוסף שנתמך ע"י Python הוא מחרוזות Unicode – מחרוזות אלו הן מחרוזות בהן כל תו מיוצג ע"י שני בתים, וכך ניתן לייצג יותר תווים בטיפוס אחד. תמיכה ב-Unicode חשובה לדוגמה כדי לאפשר לנו לעבוד עם שפות כמו עברית, וב-Python 3 (אותה אנחנו לא לומדים כאן) קיימת תמיכה נרחבת בהרבה לטקסט שמקודד ב-Unicode. אנחנו לא נלמד על Unicode, אך תוכלו לקרוא עוד על הנושא ב-help של str.encode ו-str.decode.

מחרוזת Unicode נוצרת בדיוק כמו מחרוזת רגילה, רק שלפני המחרוזת יש את התו `'u'`:

```
>>> u'Unicode String'
u'Unicode String'
>>> type(_)
<type 'unicode'>
```

רשימות

רשימה היא טיפוס שמחזיק טיפוסים אחרים, שומר על הסדר שלהם, מאפשר להוסיף, להסיר ולקבל איברים מכל מקום ברשימה, וניתן לעבור על כל אחד מהאיברים שברשימה. ב-Python, רשימה היא טיפוס מובנה, ובנוסף, רשימה אחת יכולה להחזיק איזה טיפוס שנרצה, ואפילו כמה סוגי טיפוסים בבת-אחת.

יצירת רשימה נעשית ע"י כתיבת האיברים שלה בין סוגריים מרובעות:

```
>>> x = [1, 2, 3, 4, 5]
>>> x
[1, 2, 3, 4, 5]
```

זאת הרשימה שמכילה את המספרים 1 עד 5. עכשיו נוסיף לרשימה את האיבר 6 – את זה ניתן לעשות ב-2 דרכים. הראשונה, לחבר שתי רשימות:

```
>>> x = x + [6]
>>> x
[1, 2, 3, 4, 5, 6]
```

והשניה היא להשתמש בפונקציה ()append הפועלת על רשימה:

```
>>> x.append(6)
>>> x
[1, 2, 3, 4, 5, 6]
```

הדרך הראשונה יכולה לגרום להקצאה של רשימה חדשה – וזה בגלל שיכול להיות שנוצרת רשימה חדשה אליה מוכנסים האיברים של שתי הרשימות המחוברות, ובה משתמשים מאותו הרגע. הדרך השניה מבטיחה שלא תהיה יצירה של רשימה חדשה, אלא שימוש ברשימה הקיימת והוספת איבר אליה.

בנוסף להוספת איברים לרשימה, נוכל גם להחליף איברים מסוימים:

```
>>> x[0] = 7
>>> x
[7, 2, 3, 4, 5, 6]
```

כמו כן, ניתן "לשכפל" רשימה, כלומר להעתיק את אותה הרשימה מספר פעמים:

```
>>> x * 3
[7, 2, 3, 4, 5, 6, 7, 2, 3, 4, 5, 6, 7, 2, 3, 4, 5, 6]
```

וגם לקבל את אורך הרשימה בעזרת len():

```
>>> len(x)
6
```

Slicing – או בעברית "חיתוך", היא הפעולה בה אנחנו לוקחים רשימה קיימת ויוצרים ממנה רשימה חדשה, ע"י חיתוך של חלק מהרשימה המקורית. ה-Slicing נעשה ע"י ציון האינדקסים של תחילת וסיום החיתוך. לדוגמה, הנה רשימה חדשה שמורכבת מאינדקסים 1 עד (לא כולל!) 3:

```
>>> x[1:3]
[2, 3]
```

המספר הראשון (לפני הנקודותיים) הוא האינדקס ממנו מתחיל החיתוך, והמספר השני הוא האינדקס שבו יסתיים החיתוך, אבל הוא בעצמו לא ייכלל ברשימה הנוצרת.

מטעמי נוחות, כאשר רוצים להעתיק קטע רשימה שמתחיל מיד בהתחלה שלה, או מסתיים מיד בסופה, אין צורך לציין את האינדקס הראשון (תמיד 0) או את האינדקס האחרון (תמיד אורך הרשימה), וניתן להשמיט אותו. לדוגמה, הנה הרשימה <u>עד</u> אינדקס 4:

```
>>> x[:4]
[7, 2, 3, 4]
```

והנה הרשימה מאינדקס 4 ועד סופה:

```
>>> x[4:]
[5, 6]
```

כמו כן, נוכל לבצע slicing עם קפיצה. למשל, ניקח רק את האיברים הזוגיים מהרשימה:

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

בדוגמה הזו אינדקס ההתחלה היה 0 (אנחנו יכולים לדלג עליו כמו בדוגמה הקודמת), האינדקס האחרון היה סוף הרשימה (גם עליו נוכל לדלג) והפרמטר האחרון מציין באיזה ערך להגדיל את האינדקס בכל העתקה.

נוכל לשכלל את הקפיצה ולציין גם קפיצה שלילית, מה שיגרום ל-Python ללכת אחורה:

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

וגם, נוכל להשתמש באינדקסים שליליים כדי לקבל לדוגמה את האיבר האחרון ברשימה מבלי לדעת את האורך שלה:

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

ליתר דיוק, ביקשנו את האיבר הראשון מסוף הרשימה (בעצם, נוכל לדמיין את הרשימה כרשימה מעגלית, כלומר אם נלך אחורה מההתחלה נגיע לסוף).

דוגמה חשובה ל-slicing היא המקרה שבו משמיטים גם את התחלת הרשימה וגם את סופה. במקרה הזה נקבל את כל הרשימה:

```
>>> x[:]
[7, 2, 3, 4, 5, 6]
```

לכאורה זהו בזבוז מקום, הרי ניתן לרשום סתם x ולקבל את הרשימה כמו שהיא, אבל מיד נראה למה זה טוב*.

כמו שכבר ציינו מקודם, ב-Python משתנים הם בסך-הכל מצביעים, כלומר שם של משתנה הוא מצביע לטיפוס מסוים. כאשר משתנה מצביע לטיפוס כלשהו – בין אם זה טיפוס מובנה או טיפוס שנוצר ע"י המשתמש (בהמשך החוברת נראה כיצד עושים את זה), הוא מצביע לכתובת מסוימת בזיכרון. כאשר מבצעים השמה (עושים =) בין שני משתנים, והמשתנים הם לא מטיפוס "פשוט" (לא מספר, לא מחרוזת ולא Tuple שמיד נפגוש), ההשמה למעשה **מעתיקה** את המצביע ממשתנה אחד למשתנה האחר.

בדוגמה הבאה ניצור רשימה בשם x, נשים אותה במשתנה y, נוסיף איבר לרשימה y ונראה כיצד הרשימה x משתנה גם כן:

```
>>> x = [1, 2, 3]
>>> y = x
>>> y.append(4)
>>> x
[1, 2, 3, 4]
>>> y
[1, 2, 3, 4]
```

הדוגמה האחרונה מראה תכונה חזקה מאוד של Python – אין בה צורך במצביעים, כיוון שאין בה משהו אחר ממצביעים. ב-Python כל משתנה הוא מצביע, והשמה היא העתקה של המצביע, ולא התוכן.

כאשר מעבירים פרמטרים לפונקציה, הדבר עובד באותה הצורה – שינוי של פרמטר שיועבר לפונקציה הוא שינוי המשתנה המקורי שהועבר לפונקציה.

תכונה זו מאוד נוחה, אבל לפעמים נרצה ליצור עותק של משתנה קיים – מה נעשה אז? השיטה הכי טובה היא להשתמש ב-Slicing אותו למדנו מקודם, ובו ראינו שכאשר עושים Slicing שכולל את כל האיברים (בלי ציון התחלה ובלי ציון סוף), נוצר עותק חדש של העצם המקורי. כעת נראה איך תתבצע ההשמה עם Slicing:

* העתקה כזו של רשימה לרשימה חדשה נקראת Shallow-copy.

```
>>> x = [1, 2, 3]
>>> y = x[:]
>>> y.append(4)
>>> x
[1, 2, 3]
>>> y
[1, 2, 3, 4]
```

יש לציין שה-Slicing יעבוד עבור מחרוזות, רשימות, Tuple-ים, וכל טיפוס שיש בו איברים לפי סדר מסוים.

Tuples

Tuple הוא טיפוס של רשימה קבועה. הטיפוס נועד לשימוש במצבים בהם אין אפשרות ליצור רשימה, או כאשר יש צורך ברשימה שאורכה לא משתנה. כמו כן, לא ניתן לשנות את האיברים שה-Tuple מצביע אליהם.

יצירת Tuple נעשית בדיוק כמו יצירת רשימה, רק שבמקום סוגריים מרובעות משתמשים בסוגריים עגולות:

```
>>> t = (1, 2, 3)
>>> t
(1, 2, 3)
```

קבלת איבר מה-Tuple היא כמו איבר מרשימה – ע"י סוגריים מרובעות:

```
>>> t[0]
1
```

אבל אם ננסה לשנות איבר ברשימה, לא יהיה ניתן לעשות זאת:

```
>>> t[0] = 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object doesn't support item assignment
```

כדי ליצור Tuple ריק נכתוב פשוט סוגריים בלי שום דבר ביניהן:

```
>>> empty_tuple = ()
```

כמו כן, ניתן לחבר שני Tuple-ים. התוצאה תהיה Tuple חדש:

```
>>> t1 = (1, 2, 3)
>>> t2 = (4, 5, 6)
>>> t1 + t2
(1, 2, 3, 4, 5, 6)
```

צריך לדעת שלא ניתן לחבר Tuple לרשימה, משום שהם אינם טיפוסים מאותו הסוג. בנוסף לחיבור, ניתן לעשות Slicing על Tuple:

```
>>> t = (1, 2, 3, 4, 5)
>>> t[1:4]
(2, 3, 4)
```

ובדומה לרשימה, השמת Tuple לא תיצור עותק שלו אלא תעביר מצביע ל-Tuple המקורי.

Tuple Assignments – כאשר מבצעים השמה בין שני Tuple-ים, Python מבצעת השמה בין כל שני איברים מתאימים ב-Tuple-ים. לדוגמה:

```
>>> (a, b, c) = (1, 2, 3)
>>> a
1
>>> b
2
>>> c
3
```

בדוגמה האחרונה נוצרו שני Tuple-ים – באחד שלושה משתנים, ובשני שלושה ערכים. Python עשתה השמה בין a ל-1, בין b ל-2 ובין c ל-3. בצורה כזו, נוצרו שלושה משתנים עם שלושה ערכים התחלתיים בשורה אחת. כמובן, לא ניתן לבצע השמה בין שני Tuple-ים שמכילים "סתם" איברים שלא ניתן להשם ביניהם:

```
>>> (1, 2, 3) = (1, 2, 3)
SyntaxError: can't assign to literal
```

אחד הטריקים שניתן לבצע בעזרת Tuple Assignments הוא החלפת ערכי שני משתנים בשורה אחת:

```
>>> x, y = y, x
```

השורה האחרונה פשוט החליפה בין הערכים של x ו-y, ללא צורך במשתנה זמני. שימו לב גם שהורדנו את הסוגריים, כי Python מאפשרת להשמיט סוגריים בכל מקום שבו אין דו-משמעות.

השימוש העיקרי ב-Tuple-ים ב-Python הוא עבור אחסון מידע שאין צורך לשנותו (או שיש צורך לוודא שהוא לא ישתנה). כמו כן, בהמשך החוברת נראה שפונקציה יכולה להחזיר רק איבר אחד. משום ש-Tuple הוא טיפוס, ניתן להחזיר מפונקציה Tuple המכיל מספר איברים, וע"י כך להחזיר יותר מאיבר אחד מפונקציה.

מילונים

מילון (Dictionary) ב-Python הוא טיפוס שמטרתו היא המרה בין מפתחות לבין ערכים (ב-C++ משתמשים ב-map לכך). מילון למעשה מחזיק ערכים, מפתחות ואת המיפוי ביניהם. כאשר המשתמש פונה למילון, הוא מציין מפתח, ומקבל עבורו ערך. היתרון בכך הוא שבמקום מספרי אינדקסים (שלא ממש אומרים משהו למישהו), ניתן להשתמש בכל ערך שרק רוצים, אפילו מחרוזות.

יצירת מילון נעשית כמו רשימה, אבל עם סוגריים מסולסלות:

```
>>> d = {}
```

כעת נוסיף למילון d את המיפוי בין המפתח 1 למחרוזת "Sunday":

```
>>> d[1] = "Sunday"
>>> d
{1: 'Sunday'}
```

שימו לב ש-Python מדפיסה את המילון שלנו בצורה {1: 'Sunday'}, ונוכל לבנות מילון בדיוק באותה צורה בעצמנו מבלי להזין את האיברים אחד אחרי השני:

```
>>> d = {1: 'Sunday', 2: 'Monday'}
>>> d
{1: 'Sunday', 2: 'Monday'}
```

בהמשך נשתמש בשיטה הזו כדי ליצור מילונים כחלק מהקוד שלנו.

עכשיו, כשהמילון d יודע שה"ערך" של 1 הוא "Sunday", נוכל לבקש ערך עבור מפתח מסוים כמו ברשימה:

```
>>> d[1]
'Sunday'
```

כמו כן, ניתן ליצור מפתחות שאינם מספרים. למשל, ניצור מילון שממיר בין מדינה לבירה שלה:

```
>>> cap = {}
>>> cap['Israel'] = 'Jerusalem'
>>> cap['USA'] = 'Washington'
>>> cap['Russia'] = 'Moscow'
>>> cap['France'] = 'Paris'
>>> cap['England'] = 'London'
>>> cap['Italy'] = 'Rome'
```

בעת פנייה למילון נוכל לציין מייד את שם המדינה ולקבל את הבירה שלה:

```
>>> cap['France']
'Paris'
```

אם נפנה למפתח שאינו קיים, תוחזר שגיאה:

```
>>> cap['Japan']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: Japan
```

בנוסף לפעולות ההמרה, ניתן לבדוק גם האם מפתח מסוים קיים במילון, בעזרת האופרטור in:

```
>>> 'Japan' in cap
False
>>> 'Israel' in cap
True
```

keys היא פונקציה שמחזירה את רשימת המפתחות במילון:

```
>>> cap.keys()
['Israel', 'Italy', 'USA', 'England', 'Russia', 'France']
```

values היא פונקציה שמחזירה את רשימת הערכים במילון:

```
>>> cap.values()
['Jerusalem', 'Rome', 'Washington', 'London', 'Moscow', 'Paris']
```

וכמובן נוכל לקבל את גודל ("אורך") המילון בעזרת len():

```
>>> len(cap)
6
```

חשוב לדעת שבמילונים אין סדר לאיברים. כלומר, לא משנה באיזה סדר נכניס את האיברים, המילון ישמור אותם בסדר פנימי ולא ידוע משלו. לכן, אי-אפשר להמיר בין מילון לרשימה או Tuple, בעוד שכן אפשר להמיר בין Tuple לרשימה לדוגמה.

לבסוף, נוכל גם לקבל מהמילון רשימת tuple-ים, כשבכל tuple נקבל זוג של (key, value). בדוגמת ערי הבירה שלנו:

```
>>> cap.items()
[('Israel', 'Jerusalem'), ('Italy', 'Rome'), ('USA', 'Washington'), ('France',
'Paris'), ('England', 'London'), ('Russia', 'Moscow')]
```

set

קבוצה (set) היא כמו רשימה, מלבד העובדה שאי-אפשר לאחסן פעמיים את אותו האיבר. כדי ליצור קבוצה מרשימת איברים משתמשים בסוגריים מסולסלים:

```
>>> x = {1, 2, 3}
>>> x
set([1, 2, 3])
```

לקבוצה נוכל להוסיף איברים, ואם ננסה להוסיף איבר שקיים כבר בקבוצה, לא יקרה כלום:

```
>>> x.add(2)
>>> x.add(3)
>>> x.add(4)
>>> x
set([1, 2, 3, 4]
```

הסרת איברים אפשרית בשתי דרכים. הדרך הראשונה, הכי פשוטה, היא להגיד לקבוצה להוריד את האיבר אם הוא קיים, ואם הוא לא בקבוצה אז לא לעשות כלום:

```
>>> x.discard(5)
>>> x.discard(2)
>>> x
set([1, 3, 4])
```

הדרך השניה להסיר איבר היא לבקש מהקבוצה להסיר אותו בצורה מפורשת. אם האיבר לא קיים, תיווצר שגיאה:

```
>>> x.remove(1)
>>> x.remove(5)

Traceback (most recent call last):
  File "<pyshell#150>", line 1, in <module>
    x.remove(5)
KeyError: 5
>>> x
set([3, 4])
```

אם נרצה, נוכל גם ליצור קבוצה מרשימה או tuple קיים, וכך לקבל את אותה רשימה בלי כפילויות:

```
>>> w = [1, 1, 1, 2, 3, 9, 2, 4, 3, 6]
>>> set(w)
set([1, 2, 3, 4, 6, 9])
```

תנאים: if, elif, else

if היא כמובן פקודת ההתניה (Conditioning), כמו ברוב שפות התכנות. מבנה הפקודה דומה לזה שבשפת C מלבד העובדה שאין חובה בסוגריים. דוגמה די פשוטה ל-if:

```
num = 5
if num > 3:
    print(num*2)
```

בשורה הראשונה יצרנו משתנה בשם num שערכו 5. השורה השניה היא שורת ה-if עצמה, ומבנה ההתניה דומה לזה שב-C (מיד נראה מספר הבדלים). בסוף השורה השניה יש נקודותים. תו ה-':' ב-Python מורה על פתיחת בלוק חדש, בדיוק כמו הסוגריים-המסולסלות ב-C.

בלוקים

ואיך Python יודעת מתי בלוק נגמר? את זה היא עושה באמצעות הרווחים ששמנו בתחילת השורה בבלוק ה-if. בדוגמה למעלה, השורה השלישית מוזחת ימינה ב-4 רווחים, ולא סתם בשביל שיהיה קל יותר לקרוא את הקוד (כמו בשפות אחרות). הוספת הרווחים לפני כל שורה בקוד Python משייכת אותה לבלוק מסוים.

השיטה הזו נקראת אינדנטציה (Indentation), והיא אומרת שבלוק שתלוי בבלוק קודם ייכתב "פנימה" יותר בקוד. ב-Python אינדנטציה היא חלק מהמבנה שלה, ולכן אי-אפשר לכתוב בלוק חדש בלי להוסיף ריווח לפני כל שורה בבלוק. ריווח הוא בסה"כ כמות רווחים מסויימת, וכדי ש-Python לא תתבלבל אנחנו צריכים להקפיד על ריווח אחיד. לא נוכל למשל לכתוב בלוק שבו חלק מהשורות יהיו עם ריווח של 3 רווחים וחלק אחר עם 4 רווחים. אם נעשה כזה דבר Python לא תוכל לדעת איזו שורה שייכת לאיזה בלוק, ולכן עלינו להקפיד שכל שורה תתחיל עם מספר רווחים זהה שמתאים לבלוק בה היא נמצאת.

אפשר גם להשתמש בתו TAB עצמו, אבל אז Python תהפוך כל TAB ל-8 רווחים. שימו לב לא לכתוב בטעות TAB-ים ולעבוד בעורך קוד שמציג TAB-ים כ-4 רווחים, כי אז הקוד שאתם רואים שונה מהקוד שאתם מריצים. באופן כללי, העדיפו לא להשתמש ב-TAB-ים מאחר שהם מייצרים בלבול בין אנשים שונים שנוהגים להציג את TAB בתור כמות רווחים שונה.

מקובל בעולם לעבוד עם אינדנטציה של 4 רווחים רגילים, וכך גם אנחנו נעשה.

and, or, not

בנוסף לתנאי if רגילים, אפשר ליצור גם תנאים לוגיים. ב-Python קיימות מילות המפתח and, or ו-not, בנוסף לאופרטורים הלוגיים שלקוחים מ-C. השימוש הוא די פשוט:

```
num = 17
if (num > 5) or (num == 15):
    print("something")
```

השימוש הוא זהה עבור and. not יכול לבוא לפני תנאי כדי להפוך אותו:

```
if not 4 == 7:
    print ("always true")
```

else, elif

אם נרצה לבצע משהו במקרה והתנאי לא מתקיים, נוכל להשתמש ב-else:

```
>>> if 1 == 2:
...     print ('Something is wrong')
... else:
...     print ('Phew...')
...
Phew...
```

אבל מה יקרה אם נרצה לשים if בתוך בלוק ה-else? נקבל משהו כזה:

```
>>> sky = 'Blue'
>>> if sky == 'Red':
...     print ("Hell's rising!!")
... else:
...     if sky == 'Yellow':
...         print ('Just a sunny day ^_^')
...     else:
...         if sky == 'Green':
...             print ('Go see a doctor')
...         else:
...             print ('It appears the sky is %s' % (sky, ))
...
It appears the sky is Blue
```

פשוט זוועה. בשביל זה הוסיפו ב-Python את elif:

```
>>> if sky == 'Red':
...     print ("Hell's rising!!")
... elif sky == 'Yellow':
...     print ('Just a sunny day ^_^')
... elif sky == 'Green':
...     print ('Go see a doctor')
... else:
...     print ('It appears the sky is %s' % (sky, ))
...
It appears the sky is Blue
```

מתי תנאי "נכון"

כמו שראינו עד עכשיו ועוד נראה בהמשך, אחרי if שמנו כל מיני תנאים. לפעמים אלה היו תנאים עם ==, לפעמים עם >, וכמו שתראו בעוד הזדמנויות נוכל גם לשים משתנים בפני עצמם, למשל:

```
>>> alive = 'Yes'
>>> if alive:
...     print ("I'm alive!")
...
I'm alive!
```

הסיבה שהקוד הזה רץ היא ש-Python מבצעת עבורנו המרה של כל תנאי שהיא פוגשת ל-True או False. כמו שראינו מקודם, נוכל לרשום את התנאים שבאים אחרי if ב-interpreter ולקבל את הערך שלהם. לדוגמה:

```
>>> 5 == 6
False
```

כשאנחנו רושמים משתנה בפני עצמו, Python פשוט ממירה את המשתנה או האובייקט לבוליאני ע"י-כך שהיא בודקת האם אותו אובייקט נחשב מבחינתה ל-False או True. אובייקטים שמומרים אוטומטית ל-False הם 0, 0.0, רשימה ריקה, tuple ריק, מילון ריק, set ריק, None ומחרוזת ריקה. כל אובייקט אחר יומר ל-True, ולכן:

```
>>> if '':
...     print ('I will never run :(')
...
>>> if 7:
...     print ('Seven is an awesome number')
...
Seven is an awesome number
```

לולאות

לולאת while

פקודת הלולאה while מאפשרת חזרה על פקודות עד שתנאי מסוים מתקיים. התנאי יכול להיכתב כמו תנאי ב-if. דוגמה:

```
i = 0
while i < 10:
    print (i)
    i += 1
```

לולאת for

לפני שנסקור את פקודת for, נגדיר מהו **רצף** (Sequence) – רצף הוא עצם שמכיל בתוכו איברים, בסדר מסוים, כאשר העצם מכיל פונקציות לקבלת כל-אחד מהאיברים (או פונקציות לקבלת איבר ראשון ואיבר עוקב לאיבר קיים). דוגמאות לרצפים שאנחנו כבר מכירים ב-Python הן רשימות, Tupleים, מחרוזות ומילונים.

פקודת הלולאה for שונה מזו שב-C. נחזור על זה: פקודת הלולאה for שונה מזו שב-C. במילים אחרות לגמרי – לולאת for של Python זה לא כמו ב-C. לא כמו ב-C. לא כמו ב-C. לא כמו ב-C. גם לא כמו לולאת for של פסקל, וגם לא כמו של בייסיק או שאר השפות המקולקלות האלה. for ב-Python לא מבצעת לולאה על תחום ערכים (כמו בפסקל) וגם לא מאחדת בתוכה משפט-אתחול, תנאי ופקודה לביצוע בכל איטרציה (כמו ב-C). אם אתם לא מכירים אף אחת מהשפות האלה, הכי טוב. ככה תוכלו ללמוד מהי לולאת for כמו שהטבע התכוון אליה.

for ב-Python מקבלת רצף (כלומר אוסף של איברים), ובכל איטרציה מתייחסת רק לאיבר אחד ברצף. ההתייחסות לאיברים נעשית לפי הסדר בו הם מאוחסנים ברצף, כאשר הסדר נקבע לפי טיפוס הנתונים הנתון (אף אחד לא מבטיח סדר מסוים, אלא אם כן זוהי אחת מהגדרותיו של טיפוס הנתונים).

דוגמה פשוטה ל-for:

```
days = ('Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat')
for day in days:
    print (day)
```

בדוגמה, הלולאה יצרה משתנה בשם day. בכל איטרציה, day מקבל את האיבר הבא של days. הפלט של הריצה יהיה:

```
Sun
Mon
Tue
Wed
Thu
Fri
Sat
```

חשוב לציין שבסיום ריצת הלולאה, המשתנה day ימשיך להתקיים, וערכו יהיה הערך האחרון שקיבל במהלך ריצת הלולאה.

אם מריצים לולאת for עם רשימה ריקה, כלום לא יקרה והמשתנה day (או כל משתנה אחר שייכתב שם) לא ייווצר. יש לשים לב שאם מריצים כמה לולאות, אחת אחרי השניה, עם אותו משתנה (נגיד i), ובאחת הלולאות יש רשימה ריקה, אין הדבר אומר שהמשתנה i יושמד. זה רק אומר שהוא לא ישתנה עקב הלולאה הזו. אם הלולאה הזו היא הלולאה הראשונה, i באמת לא ייווצר:

```
>>> for i in [0, 1, 2, 3, 4]:
...     print(i)
0
1
2
3
4
>>> i
4
>>> for i in []:
...     print(i)
>>> i
4
```

כאמור, ניתן לעבור לא רק על רשימות, אלא גם על מילונים ומחרוזות. להלן שתי דוגמאות להמחשת העבודה עם מחרוזות ומילונים:

```
>>> dic = {}
>>> dic[1] = 10
>>> dic[2] = 20
>>> dic[3] = 30
>>> dic[4] = 40
>>> dic[5] = 50
>>> for i in dic:
...     print(i)
...
1
2
3
4
5
>>> str = 'Yet another string'
>>> for char in str:
...     print(char)
...
Y
e
t

a
n
o
t
h
e
r

s
t
r
i
n
g
```

הערה לגבי מילונים – בהגדרת המילון לא נאמר שהסדר בו האיברים מופיעים בו הוא הסדר בו הם יאוחסנו או הסדר בו הם יוחזרו בלולאת for. לכן, אין להניח הנחות על סדר האיברים בטיפוס מסוים, אלא אם נאמר אחרת בהדגרת הטיפוס.

הנחה שכן ניתן להניח (וגם זאת בזהירות רבה מאוד) היא שאם הרצנו לולאת for על רשימה מסוימת, והרשימה לא השתנתה, ריצת for נוספת על אותה הרשימה תתבצע באותו הסדר. לקראת סוף החוברת נראה באילו מקרים ההנחה הזאת לא מתקיימת יותר.

break, continue

משפט break נועד ליציאה מלולאות for או while. break יוצא רק מבלוק ה-for או בלוק ה-while בו הוא נמצא, ולא מסוגל לצאת החוצה יותר מבלוק אחד.

משפט continue מורה להפסיק מיד את האיטרציה הנוכחית של הלולאה ולהתחיל את האיטרציה הבאה. כמו break, גם continue יכול לצאת רק מבלוק הלולאה הקרוב אליו. דוגמה:

```
>>> for monty in (0, 1, 2, 3, 4):
...     for python in (0, 1, 2, 3, 4):
...         continue
...     print (monty * python)
0
4
8
12
16
```

אם לא הבתנם עד הסוף מה קרה כאן, העתיקו את הדוגמה למחשב, הוסיפו print-ים ושנו את המספרים בדוגמה כדי להבין יותר טוב מה הקוד עושה.

בנוסף, אפשר להוסיף בלוק else ללולאות for ו- while. הבלוק של else מורץ במקרה בו סיימנו את ריצת הלולאה כולה ולא עצרנו בעקבות break. דוגמה למקרה בו לולאה מצליחה להסתיים בשלום ולכן בלוק ה-else שלה מורץ:

```
>>> for i in []:
...     pass
... else:
...     print ('Yey! Loop ended gracefully')
...
Yey! Loop ended gracefully
```

והדוגמה ההפוכה:

```
>>> for i in [1]:
...     break
... else:
...     print ('I am a line of code that will never run :(')
...
>>>
```

range

טוב, אז יש לולאת for שיודעת לעבור על כל האיברים ברשימה. אבל מה אם כן רוצים לעשות לולאת for שעוברת על סדרת מספרים, אבל לא רוצים להשתמש ב-while?

פתרון אחד (לא מומלץ) הוא ליצור רשימה שמכילה את כל המספרים שנרצה לעבור עליהם, מ-0 עד X. ברור שזה פתרון לא משהו, כי הרי איך ניצור את הרשימה? בעזרת לולאת while (ואז מה עשינו בדיוק..?) או שניצור אותה ישירות בקוד (שזה מגעיל).

פייתון פותרת לנו את הבעיה עם הפונקציה range

מבנה הפונקציה range:

```
range([start], end, [step])
```

- start – זהו פרמטר אופציונלי, ומציין את ערך ההתחלה (כולל) של הרשימה.
- end – ערך זה תמיד ייכלל ב-range ומציין את הערך האחרון ברשימה (לא כולל). כלומר, הרשימה מגיעה עד הערך הזה, ולא כוללת אותו.

- step – ההפרש בין כל שני איברים סמוכים ברשימה. כמובן, הפרש זה חייב להיות גדול מאפס.

דוגמאות ל-range:

```
>>> for i in range(10):
...     print i
0
1
2
3
4
5
6
7
8
9
>>> for i in range(2, 10, 2):
...     print i
2
4
6
8
```

אז מה עושה range?

range לא יוצרת עבורנו רשימה, אלא יוצרת אובייקט רצף, שניתן לקבל ממנו התחלה, סוף ואת האיבר הבא לאיבר קיים. בצורה כזו, אובייקט range מחזיק את תחום הערכים אותו הוא צריך להחזיר ואת ההפרש בין כל שני איברים. בכל קריאה לקבלת האיבר הבא, האובייקט מחשב את האיבר הבא במקום לאחסן אותו בזיכרון. כאשר מבקשים את האיבר הבא לאיבר האחרון, אובייקט ה- range מודיע על סוף הרשימה.

ככלל אצבע, נשתמש ב- range כשנרצה רק תחום של מספרים (כמו בלולאת for)

pass

תכירו את pass – המשפט שאומר לא לעשות כלום. כן, יש דבר כזה.

הסיבה לכך שיש כזה משפט בכלל היא לא בשביל שאפשר יהיה לכתוב את המשפט הזה בכל מני מקומות בקוד כדי שיהיו יותר שורות. משפט זה עוזר לנו לכתוב קוד: לפעמים אנחנו כותבים לולאה, כי אנחנו יודעים שהיא צריכה להיות במקום מסוים בקוד, אבל אנחנו לא ממש רוצים לכתוב אותה כרגע, אין לנו איך, או 1001 (9) סיבות אחרות.

בגלל ש-Python מחפשת את תו ה-TAB בבלוק חדש, זאת תהיה שגיאה לא לפתוח בלוק במקומות מסוימים. כמו כן, זאת שגיאה לשים סתם שורה ריקה עם TAB בהתחלה.

לכן, ניתן לרשום במקום הבלוק החסר את המשפט pass, ו-Python פשוט תעבור הלאה את הלולאה:

```
>>> for k in xrange(0, 500, 5):
...     pass
```

חשוב להבין שהלולאה תתבצע – היא תרוץ 100 פעמים ובכל אחת מהריצות של הלולאה לא תעשה שום דבר. אם נשים מספר מאוד גדול במקום ה-500 (נגיד 50000000000) נוכל אפילו לראות את המחשב נתקע למעט זמן. לכן, pass אומר לעשות "כלום", ולא לצאת מהלולאה כמו break.

חלק 2: ביטויים ופונקציות

עד כה עסקנו במשחקים קטנים ונחמדים עם קוד קצר ופשוט. אבל, כמו שנאמר במבוא, Python היא שפה מאוד חזקה, עד כדי כך שניתן לכתוב בה אפליקציות שלמות.

כמו בכל שפה, כתיבת כמויות גדולות של קוד, או סתם תוכניות רגילות, תתבצע ע"י חלוקה לפונקציות, מודולים, וכו'. פרק זה עוסק בביטויים ובפונקציות.

הגדרת פונקציות

פונקציה ב-Python מאפשרת לנו לקחת כמה שורות קוד ולשים אותן במקום אחד כדי שנוכל להפעיל אותן בעתיד. כפי שמיד נראה, פונקציה יכולה להחזיר ערך כלשהו (נגיד מספר או רשימה), לא להחזיר כלום, או להחזיר מספר טיפוסים שונים במקרים שונים, וזאת מבלי לשנות את ההגדרה שלה.

כמו כן, חוקי ה-Case Sensitivity בשמות של פונקציות זהים לאלו של משתנים – יש הבדל בין אותיות גדולות וקטנות (לדוגמה, פונקציה בשם f ופונקציה בשם F הן שתי פונקציות שונות).

נתחיל בהגדרת פונקציה פשוטה בלי פרמטרים:

```
>>> def func():
...     print "I am a function"
>>> func()
'I am a function'
```

הפונקציה הזו לא מקבלת שום פרמטר, וגם לא מחזירה כלום (אין בה משפט return). אם ניקח את ערך ההחזרה של הפונקציה ונשים אותו בתוך משתנה, המשתנה יכיל "כלום":

```
>>> h = func()
'I am a function'
>>> h
>>>
```

וכמו שניתן לראות, כלום לא הודפס – המשתנה h מכיל "כלום".

None

ה-"כלום" שהרגע ראינו מכונה ב-Python "None". None הוא עצם יחיד (Singleton), שמצביעים אליו כשרוצים לציין שאין לנו למה להצביע. None לא אומר שאין לנו מה להחזיר (כמו void ב-C), אלא אומר שאנחנו לא מחזירים כלום (או ליתר דיוק, מחזירים "כלום"). כמו כן, None הוא ערך חוקי לכל דבר, וניתן להשים אותו למשתנים באותו אופן שניתן להחזיר אותו מפונקציות.

על-מנת שפונקציה תחזיר None, ניתן לעשות שלושה דברים:

1. לא לרשום שום משפט return.
2. לרשום שורה ובה המילה return בלבד.
3. לרשום את השורה return None.

pass בפונקציות

כמו בלולאות, pass עובד יפה מאוד גם בפונקציות. פשוט רושמים pass במקום בלוק הפונקציה:

```
def doing_nothing():
    pass
```

גם כאן, בדיוק כמו בלולאת ה-for מסוף הפרק הקודם, pass הוא לא return או משהו בסגנון. כדי להבין את pass יותר טוב, נסתכל על הפונקציה:

```
>>> def f():
...     print 1
...     print 2
...     pass
...     print 3
...
>>> f()
1
2
3
```

כאמור, pass לא עושה כלום והיא קיימת רק כדי שנוכל להגדיר פונקציות או בלוקים בלי שנהיה חייבים לשים בהם תוכן.

תיעוד

תיעוד (Documentation) הוא פעולה שאיננה כתיבת קוד. תיעוד הוא כתיבת הערות לתוכניות שלנו כדי שאנחנו, או אנשים אחרים, נוכל לקרוא בעתיד הערות על הקוד ולהבין כיצד להשתמש בו.

יש שני סוגים גדולים של תיעוד ב-Python:

1. סתם תיעוד שזורקים באמצע הקוד, כדי שהקוד עצמו יהיה יותר ברור למי שיקרא אותו.
2. מחרוזת תיעוד קבועה בתחילת הפונקציה.

הסוג הראשון נעשה בצורה פשוטה מאוד – כמו שב-++C יש את רצף התווים "//" שאומר שמעכשיו ועד סוף השורה יש דברים שלא צריך לקמפל, כך גם ב-Python יש את התו "#" שאומר שאין להתייחס למה שכתוב החל מתו זה ואילך:

```
>>> def func():
...    print "We have to write something" # ladod moshe hayta para
```

הסוג השני של תיעוד הוא סוג מיוחד, ואליו Python מתייחסת בצורה שונה – בתחילת פונקציה, שורה אחת אחרי שורת ה-def (בלי שום רווחים מסתוריים), ניתן לשים מחרוזת (כמובן, עם TAB לפניה). המחרוזת הזאת היא מחרוזת התיעוד של הפונקציה. כאשר מישהו אחר (או אנחנו) ירצה יום אחד לדעת מהי הפונקציה הזו, הוא יוכל לקבל את המחרוזת בקלות רבה. העובדה שהתיעוד נמצא בתוך הקוד עצמו הוא דבר מאוד חזק – זה אומר שאם יש לך את הקוד, יש לך את התיעוד. אין צורך לחפש קובץ תיעוד במקום שלא קשור לקוד או ספר מודפס. מושג מחרוזות התיעוד בתחילת פונקציות שאול משפת התכנות LISP, וקרוי DocStrings (או Documentation Strings)

```
>>> def tor():
...     "I am a function that does absolutely nothing"
...     pass
...
>>> tor()
>>> tor.__doc__
'I am a function that does absolutely nothing'
```

אז את מחרוזת התיעוד מקבלים ע"י כתיבת שם הפונקציה, נקודה, ו- __doc__. ניתן גם להשתמש בפונקציה המובנית help() שיודעת להדפיס בצורה יפה יותר מחרוזת תיעוד של פונקציה (או כל אובייקט אחר עם מחרוזת תיעוד):

```
>>> help(tor)
Help on function tor in module __main:__

tor()
    I am a function that does absolutely nothing
```

בהמשך כשנלמד על מודולים, זכרו שאפשר גם להפעיל את help() על מודול ולראות את רשימת כל הפונקציות שהוא מכיל ואת התיעוד של כל אחת מהן.

מחרוזות תיעוד כמו שראינו הן דבר נחמד. אבל, ברוב הפעמים נרצה לכלול יותר משורה אחת בתיעוד, שתכיל את מבנה הפונקציה, מה היא מחזירה, פרמטרים, תיאור מפורט, וכו'. כדי לעשות את זה, כותבים את מחרוזת התיעוד, ומוסיפים '\n'-ים כאשר רוצים לרדת שורה:

```
>>> def func_with_long_doc():
...     "I have a very very very very very\nvery very long documentation"
...
```

כדי להדפיס את מחרוזת התיעוד כמו שצריך (ולא את תוכן המחרוזת עצמה), נשתמש ב-help():

```
>>> help(func_with_long_doc)
Help on function func_with_long_doc in module __main__:

func_with_long_doc()
    I have a very very very very very
    very very long documentation
```

קצת יותר טוב, אבל הדרך הנוחה ביותר שפיתון מציעה היא בעזרת סוג אחר של מחרוזות שנפרשות על פני כמה שורות:

```
>>> def f(a, b, c):
...     """f(a, b, c) --> list
...
...     Takes 3 integers (a, b, c), and returns a list with those
...     three numbers, and another number at the end of that list,
...     which is the largest number of all three.
...     """
...     return [a, b, c, max(a, b, c)]
...
```

למעשה, כדי ליצור מחרוזת תיעוד בלי להסתבך עם \n-ים מגעילים, משתמשים במחרוזת עם 3 גרשיים (כפולים או בודדים, העיקר שהגרשיים הפותחים והסוגרים יהיו סימטריים). המחרוזת מסתיימת כאשר פיתון פוגשת שוב 3 גרשיים, וכל מה שבאמצע, כולל \n-ים, נכנס למחרוזת:

```
>>> f.__doc__
'f(a, b, c) --> list\n\n    Takes 3 integers (a, b, c), and returns a list with
those\n    three numbers, and another number at the end of that list,\n    which is
the largest number of all three.\n    '
```

כמובן שכדי לקרוא את התיעוד של פונקציות נשתמש ב-()help ולא נסתכל ישירות על המחרוזות __doc__ שלהן.

()help יודעת לפרמט את מחרוזת התיעוד בצורה יפה, בלי ה-\t או הרווחים שהתווספו לתחילת כל שורה:

```
>>> help(f)
Help on function f in module __main__:

f(a, b, c)
    f(a, b, c) --> list

    Takes 3 integers (a, b, c), and returns a list with those
    three numbers, and another number at the end of that list,
    which is the largest number of all three.
```

פרמטרים לפונקציות

פונקציות שרק מדפיסות דברים קבועים הן לא ממש שימושיות. בשביל לגרום לפונקציות להיות יותר מועילות נצטרך להעביר להן פרמטרים. העברת פרמטרים תיעשה בסוגריים של הפונקציה:

```
def func_with_args(a, b, c):
    pass
```

לפונקציה זו העברנו שלושה פרמטרים: a, b, ו-c. כמו שניתן לראות, אין צורך בטיפוס, כי ב-Python משתנה יכול להיות מכל טיפוס שהוא. זה לא אומר שהפונקציה מוכנה לקבל כל סוג של משתנה, הרי היא מצפה לסוג משתנה מסוים (מספר, מחרוזת...). אם לא נוהגים בחופש זה בזהירות, ניתן להקריס קוד די פשוט.

הפרמטרים שמועברים לפונקציה מועברים אליה בדיוק כמו השמה רגילה של משתנה. ההבדל הוא שהמשתנה ה"חדש" שמוגדר בתוך הפונקציה לא קשור למשתנה שהעברנו לפונקציה מבחוץ. לדוגמה:

```
>>> def f(num):
...     num = 3
...
>>> x = 1
>>> f(x)
>>> x
1
```

קראנו לפונקציה עם המשתנה x, וכאשר הפונקציה קיבלה אותו הוא נקרא אצלה num. המשתנה num בפונקציה f הוא לא אותו משתנה x. שני המשתנים מצביעים לאותו אובייקט (במקרה הזה המספר 1), אבל אם נשנה את num מתוך הפונקציה, הערך של x לא ישתנה. כמו שאפשר לראות בדוגמה, x ממשיך להצביע ל-1 בעוד ש-num הצביע ל-3 ברגע שהפונקציה יצאה.

אחרי כל הדוגמאות שלא עושות כלום, בואו נראה דוגמה לפונקציה שעושה עבודה כלשהי:

```
>>> def f(a, b):
...     return a + b
...
>>> f(1, 2)
3
```

אוקי, זה מאוד בסיסי. מה אם נרצה לקבל מספר לא קבוע של פרמטרים? כלומר, היינו רוצים לקבל כל כמות פרמטרים, או 3 פרמטרים ויותר? Python מאפשרת לנו להגדיר את הפרמטרים שנהיה חייבים לקבל, וגם להגדיר פרמטר מיוחד שיקבל את כל הפרמטרים ה"מיותרים" שהעברנו לפונקציה:

```
>>> def greet(first_name, last_name, *more_names):
...     return 'Welcome, %s %s %s' % (first_name, last_name, ' '.join(more_names))
...
>>> greet('Galila', 'Ron', 'Feder', 'Amit')
Welcome, Galila Ron Feder Amit
```

הכוכבית לפני השם של הפרמטר האחרון (*more_names) אומרת שהפרמטר הזה הוא לא פרמטר רגיל, אלא צריך לקבל אליו את כל הפרמטרים שבאים אחרי הפרמטרים הרגילים שאנחנו חייבים להעביר לפונקציה כדי לקרוא לה.

ומה ה-type של הפרמטר המיוחד הזה? ניחשתם נכון – tuple. את הפרמטרים העודפים נקבל ב-tuple ופשוט נוכל להשתמש בהם. ה-tuple יכול להיות כמובן גם ריק, אם לא נעביר אף פרמטר מעבר למה שהיינו חייבים.

בד"כ מקובל לקרוא לפרמטר הזה args (ונכתוב *args כשנרצה להיות אפילו יותר מפורשים), אבל השם הזה הוא סתם קונבנציה ולא חייבים להשתמש בו אם יש לנו שם יותר משמעותי, כמו בפונקציה greet שכתבנו מקודם.

דוגמה נוספת:

```
>>> def sum_two_or_more_numbers(num1, num2, *args):
...     result = num1 + num2
...     for other_num in args:
...         result += other_num
...     return result
...
>>> sum_two_or_more_numbers(1, 2)
3
>>> sum_two_or_more_numbers(1, 2, 3, 4, 5)
15
```

אוקי, בואו נסתכל שניה על הדוגמה הזאת. איך היינו יכולים לממש את sum_two_or_more_numbers()? דרך אחת היא המימוש שכאן – קיבלנו שני מספרים או יותר, אז סכמנו את שני המספרים והוספנו לסכום הזה את כל שאר המספרים, אם יש כאלה. זה מימוש טוב ויפה, וככה צריך לממש.

אבל, רק בשביל הדוגמה, נניח שהיינו רוצים לממש את אותה הפונקציה בדרך קצת יותר מתחכמת: בת'כלס, אנחנו תמיד מחברים את num1 ו-num2 ואז מוסיפים את מה שיש ב-args. אבל אפשר להסתכל על זה אחרת – אנחנו תמיד מחברים בין num1 למשהו אחר. המשהו האחר הזה הוא num2 בלבד (אם אין עוד מספרים) או הסכום של num2 ושאר המספרים ב-args. מאחר ש"שהסכום של num2 ושאר המספרים" הוא בעצם מה שהפונקציה עצמה עושה (היא סוכמת שניים או יותר מספרים, ויש לנו שניים או יותר מספרים), נוכל לקרוא ל-sum_two_or_more_numbers במקום לבצע את הלולאה*.

טריק נחמד, רק שבשביל זה אנחנו צריכים להיות מסוגלים לקרוא ל-sum_two_or_more_numbers עם מספר משתנה של פרמטרים. זה בעצם ההפך מהגדרה של פונקציה עם *args, ו-Python מאפשרת לנו לעשות את זה. זה נראה בדיוק

* השיטה הזו שבה פונקציה קוראת לעצמה נקראת "רקורסיה". תוכלו לקרוא עוד על רקורסיות ב-Wikipedia.

אותו הדבר כמו ההגדרה: אם נרצה לקרוא לפונקציה ולהעביר לה מספר משתנה של פרמטרים, פשוט נשים * לפני ה-tuple או הרשימה שנעביר לפונקציה. לדוגמה:

```
>>> sum_two_or_more_numbers(*range(5))
10
```

זה כבר ממש מגניב. עכשיו נממש את הפונקציה כמו שאמרנו:

```
>>> def sum_two_or_more(x1, x2, *args):
...     if args:
...         second = sum_two_or_more(x2, *args)
...     else:
...         second = x2
...     return x1 + second
...
>>> sum_two_or_more(1, 2)
3
>>> sum_two_or_more(1, 2, 3, 4, 5)
15
>>> sum_two_or_more(*range(5))
10
>>> sum_two_or_more('Hello', 'World', '!')
'HelloWorld!'
```

שימו לב שהפונקציה שלנו (גם בגרסה הראשונה וגם בגרסה שהרגע כתבנו) פועלת על כל טיפוס שתומך בחיבור, ולא רק על מספרים. בדוגמה הזאת הפעלנו אותה גם על מחרוזות, והיא תחבר גם רשימות ו-tuple-ים.

פרמטרים אופציונליים

כרגע אנחנו יכולים להעביר כל כמות של פרמטרים לפונקציה, ואנחנו יכולים לציין בהגדרת הפונקציה אילו פרמטרים היא חייבת לקבל. זה אפילו די קל, אבל חסר משהו, וזה האמצע בין "פרמטר שהוא חובה" לבין "כל שאר הפרמטרים". היינו רוצים את היכולת להגיד על פרמטר מסוים שלא חייבים להעביר אותו. Python מאפשרת לעשות את זה ע"י כך שנקבע מראש ערכי Default לפרמטרים. ערך Default-י אומר שאם לא העברנו ערך לאותו פרמטר בקריאה לפונקציה, יהיה לו ערך התחלתי כלשהו. בגלל הצורה שבה מגדירים פרמטרים עם ערך התחלתי, הם יכולים להופיע רק אחרי פרמטרים רגילים, אחרת Python לא תדע איזה ערך להכניס לכל פרמטר.

דוגמה פשוטה לפונקציה עם ערכי Default לפרמטרים:

```
>>> def func_with_defaults(x, y=5, z=17):
...    return x, y, z
>>> func_with_defaults(1)
(1, 5, 17)
>>> func_with_defaults(1, 4)
(1, 4, 17)
>>> func_with_defaults(9, 8, 5)
(9, 8, 5)
```

כמו כן, בעת הקריאה לפונקציה, אי-אפשר להשאיר "פרמטרים ריקים" (כלומר, לשים פסיק כדי לדלג על משתנה), וחובה להעביר את כל הפרמטרים לפונקציה. לכן לא נוכל לרשום מוטציות מוזרות כמו:

```
>>> func_with_defaults(1,,9)
File "<stdin">, line 1
    func(1,,9)
SyntaxError: invalid syntax.
```

עם זאת, נוכל "לדלג" על פרמטרים בעלי ערך מוגדר מראש, ולהעביר רק את הפרמטרים שנרצה בצורה מפורשת:

```
>>> func_with_defaults(1, z=9)
(1, 5, 9)
```

רגע... יש פרמטר לפונקציה שקוראים לו z, והפונקציה מסכימה שנעביר לה את z בצורה "מיוחדת" שבה נגיד לפונקציה "הנה הערך של z". אולי הצורה הזאת בכלל לא מיוחדת והפונקציה תתבלבל ותסכים גם אם נעביר לה ככה את x?

```
>>> func_with_defaults(x=2, z=3)
(2, 5, 3)
```

אולי הסדר בכלל לא משנה?

```
>>> func_with_defaults(z=3, x=2)
(2, 5, 3)
```

אז בעצם אנחנו יכולים להעביר כל פרמטר, אופציונלי או "רגיל", לפי השם שלו. תכף נחזור לנושא של העברת פרמטרים לפי שם, כי לפני זה אנחנו חייבים להבין עוד משהו אחד אחרון.

בסעיף הקודם הצגנו את הנושא של פרמטרים, ואז שכללנו את הפונקציות שלנו בכך שאפשרנו להן לקבל כל מספר של פרמטרים. בצורה דומה, אנחנו יכולים לשכלל את הפונקציות שלנו כדי שיוכלו לקבל כל מספר של פרמטרים עם שם. את זה עושים עם ** וזה נראה ככה:

```
def f(x, y, **kwargs):
    pass
```

ב-kwargs נקבל מילון שיכיל מיפוי בין כל הפרמטרים שהעברנו לפונקציה לפי שם ולא ציינו ערך דיפולטי עבורם. דוגמה:

```
>>> def print_people_age(**ages):
...     for person, age in ages.items():
...         print '%s is %d years old' % (person, age)
...
>>> print_people_age(moshe=8, david=15, haim=9)
moshe is 8 years old
haim is 9 years old
david is 15 years old
```

הפיצ'ר של קבלת ארגומנטים לפי שם נקרא Keyword Arguments (לכן בד"כ השם בקוד יהיה kwargs**) והוא פועל בדומה ל-args*:

```
>>> def book_hotel_room(floor=1, beds=2, **kwargs):
...     print 'You ordered a room on floor #%d, with:' % (floor, )
...     print '  %d beds' % (beds, )
...     for key, value in kwargs.items():
...         print '  %d %s' % (value, key)
...
>>> book_hotel_room(ashtrays=3, toilets=8, windows=2)
You ordered a room on floor #1, with:
  2 beds
  2 windows
  8 toilets
  3 ashtrays
```

ובאותו אופן כמו שקראנו לפונקציה עם args*, נוכל גם לקרוא לה עם kwargs**:

```
>>> choices = {}
>>> choices['ashtrays'] = 7
>>> choices['toilets'] = 4
>>> choices['windows'] = 95
>>> book_hotel_room(**choices)
You ordered a room on floor #1, with:
  2 beds
  95 windows
  4 toilets
  7 ashtrays
```

עכשיו אתם בטח שואלים את עצמכם – למה שלא נאחד את כל מה שראינו עד עכשיו? בואו ניקח גם פרמטרים רגילים, גם פרמטרים לפי שם, גם פרמטרים רגילים מיותרים וגם פרמטרים מיותרים לפי שם:

```
>>> def together(a, b, c=0, *args, **kwargs):
...     print a, b, c, args, kwargs
...
>>> together(1, 2)
1 2 0 () {}
>>> together(1, 2, 3)
1 2 3 () {}
>>> together(1, 2, 3, 4, 5)
1 2 3 (4, 5) {}
>>> together(*range(10))
0 1 2 (3, 4, 5, 6, 7, 8, 9) {}
>>> together(1, 2, c=3, d=4, e=5)
1 2 3 () {'e': 5, 'd': 4}
>>> together(1, 2, 3, 4, 5, f=6, z=7)
1 2 3 (4, 5) {'z': 7, 'f': 6}
```

וזה המקרה הכללי של קריאה לפונקציה. כשפונקציה נקראת, Python אוספת את כל הפרמטרים שהעברנו לפונקציה ואז:

- הפרמטרים שהועברו בלי שם ספציפי (אלה נקראים Positional Arguments) מוכנסים לפרמטרים של הפונקציה לפי הסדר בו הם הוגדרו.
- הפרמטרים ה-Positional שנשארו בלי אף משתנה מוכנסים ל-*args.
- אם הועברו פרמטרים לפי שם (Keyword Arguments), הערכים שלהם מוכנסים למשתנים של הפונקציה בהתאמה לפי השם.
- שאר ה-Keyword arguments שאין פרמטר שמוכן לקבל אותם יוכנסו ל-**kwargs.

אם יש איזושהי התנגשות (למשל, העברנו ערך פעמיים לאותו פרמטר), תיווצר שגיאה:

```
>>> together(1, 2, 3, c=4)
Traceback (most recent call last):
  File "stdin", line 1, in <module>
    together(1, 2, 3, c=4)
TypeError: together() got multiple values for keyword argument 'c'
```

Conditional Expression

לפני שנמשיך עם פרמטרים אופציונליים, נכיר את ה-Conditional Expression, או בעברית "ביטוי מותנה". ביטוי מותנה (מקביל ל-Ternary Operator משפת C) הוא if ו-else בביטוי אחד. לדוגמה, במקום לרשום:

```
if sales > 50000000:
    employee_bonus = 1000
else:
    employee_bonus = 0
```

נוכל פשוט לרשום:

```
employee_bonus = 1000 if sales > 50000000 else 0
```

שזאת צורה הרבה יותר קצרה וקלה לקריאה, ולכן נשתמש בה הרבה בהמשך.

הביטוי בנוי בצורה X if C else Y, כלומר אם C נכון (True, מספר שאיננו 0, רשימה לא ריקה, מחרוזת לא ריקה או כל list/tuple/set שאיננו ריק) אז הביטוי יחזיר את הערך X, אחרת הביטוי יהיה Y.

שימו לב שנוכל לכתוב מה שנרצה בביטויים עצמם, כולל ביטויים וקריאה לפונקציות. הסיבה לכך היא ש-X או Y לא מורצים ע"י Python עד שהוא מסתכלת על התנאי ומחליטה האם להשתמש ב-X או ב-Y כערך לביטוי. לכן, הדוגמה הבאה תרוץ ולא תהיה שום שגיאה (החלק השני בביטוי לא ירוץ כלל):

```
>>> 'All is good' if True else int('Not a Number!')
'All is good'
```

פרמטרים בעייתיים

בפרק המבוא ראינו שכשאנחנו משנים רשימה, כל מי שמצביע אליה מושפע מכך:

```
>>> x = [1, 2, 3]
>>> y = x
>>> x.append(4)
>>> x
[1, 2, 3, 4]
>>> y
[1, 2, 3, 4]
```

אותה ההתנהגות קיימת גם בפונקציות. אם ניצור פונקציה שמקבלת רשימה ונשנה את הרשימה מתוך הפונקציה, הרשימה תתעדכן גם עבור מי שקרא לפונקציה (כי זאת אותה הרשימה):

```
>>> def f(p):
...     p.append(0)
...
>>> x = []
>>> f(x)
>>> x
[0]
```

אין כאן שום דבר מפתיע. בהינתן ההתנהגות עם משתנים רגילים, זה בסדר גמור שפונקציה מתנהגת באותה צורה. אבל, מה יקרה אם ננסה ליצור משתנה עם ערך דיפולטי של רשימה ריקה:

```
>>> def f(p=[]):
...     p.append(0)
...     return p
```

לכאורה זה נראה בסדר גמור: אם נעביר לפונקציה רשימה, היא תוסיף לרשימה שלנו 0. אם לא, היא תיצור עבורנו רשימה חדשה ותחזיר את הרשימה כדי שנוכל להשתמש בה. אז זהו, שלא. משהו אחר לגמרי קורה כאן:

```
>>> f()
[0]
>>> f()
[0, 0]
```

בעצם, כשהגדרנו את הפונקציה לא גרמנו לה "לייצר רשימה ריקה אם לא העברנו פרמטר לפונקציה". הערך הדיפולטי של p הוא אותו הערך בכל פעם. זאת אותה רשימה. לא נוצרת רשימה חדשה כל פעם, ולכן אם לא נעביר פרמטר לפונקציה עם רשימה ספציפית שהתכוונו לשנות, נקבל בחזרה איזושהי רשימה מלאה באיברים שלא ביקשנו בכלל.

כדי לפתור את זה, מקובל ב-Python להעביר את הערך None כפרמטר לרשימה, ואז אם p תהיה None, ניצור רשימה חדשה בכניסה לפונקציה:

```
>>> def f(p=None):
...     p = [] if p is None else p
...     p.append(0)
...     return p
...
>>> f()
[0]
>>> f()
[0]
```

שימו לב שהבעיה הזאת לא קיימת עם מספרים, מכיוון שמספר הוא אובייקט שלא ניתן לשנות את הערך שלו. במקרה של מספרים, מחרוזות ו-tuple-ים, כל פעולה שתתבצע על האובייקט המקורי תיצור אובייקט חדש שייכנס למשתנה המקומי בפונקציה, ולכן הקורא לפונקציה לא יושפע מכך.

כמו כן, שימו לב שכשבדקנו האם p הוא None, לא השתמשנו באופרטור == אלא ב-is. בהמשך נפגוש את is ונבין בדיוק מה הוא עושה, אבל כרגע רק תזכרו שכדי לבדוק האם משתנה הוא None משתמשים ב-is.

משתנים בפונקציות

אחרי שהסתכלנו על יצירת פונקציות, על תיעוד ועל הפרמטרים לפונקציות, נצלול פנימה ונראה מה קורה כאשר מכריזים על משתנים בתוך מקומות שונים בפונקציה.

המקרה הפשוט ביותר, כאשר כותבים פונקציה, ופתאום מכריזים בה על משתנה:

```
def func():
    print "Here we go"
    i = 9
    while i > 0:
        print i * 5
```

בפונקציה הזאת, הוכרז משתנה בשם i. המשתנה i יהיה קיים עד שהפונקציה תגיע לסופה, ואז יושמד. אם נקרא לפונקציות אחרות מתוך הפונקציה func, הן לא יכירו את המשתנה i. יותר מכך, אם פונקציה אחרת תכריז על משתנה באותו השם i, לכ"א מהפונקציות יהיה משתנה i משלה:

```
>>> def alice():
...     k = 3
...
>>> def bob():
...     k = 1
...     print k
...     alice()
...     print k
...
>>> bob()
1
1
```

בנוסף לכך, יכול להיות מקרה בו פונקציה יוצרת משתנה, אבל המשתנה לא נוצר בתוך הבלוק של הפונקציה, אלא באחד מתת-הבלוקים שלה, בתוך תנאי if או לולאת for או while. במצב כזה, המשתנה שנוצר ימשיך להתקיים גם אחרי היציאה מתת-הבלוק, בלי שום קשר לזה שהוא נוצר בתוך הבלוק (בשונה מאוד מ-Scopes ב-C וב-++C):

```
>>> def heavy_function():
...     e = 3
...     while e != 0:
...         e -= 1
...         if e == 2:
...             g = 9.8
...     print e, g
...
>>> heavy_function()
0 9.8
```

החזרת Tuple-ים מפונקציה

ב-Python קיים טיפוס הנתונים Tuple. בפרק על טיפוסי הנתונים סקרנו את הטיפוס, וראינו שניתן לאגד מספר ערכים ב-Tuple אחד. ניתן להשתמש בתכונה זו כדי להחזיר מספר ערכים מפונקציה, ע"י כך שמאגדים אותם ביחד ב-Tuple.

לדוגמה, הנה פונקציה שמחזירה שני ערכים:

```
>>> def f():
...     return ('bibibim', 'bobobom')
```

כאשר נשתמש בפונקציה, נוכל לקבל ממנה מיד את שני הערכים לתוך שני משתנים:

```
>>> str1, str2 = f()
>>> str1
'bibibim'
>>> str2
'bobobom'
```

וגם, ברוב המקרים ברור לפיתון שאם ביקשנו מפונקציה להחזיר שני ערכים, היא צריכה לעשות את זה ב-Tuple. לכן, במקרים הברורים נוכל להשמיט את הסוגריים:

```
>>> def f():
...     return 'bibibim', 'bobobom'
```

והכל יעבוד בדיוק כמו בפונקציה הקודמת.

חלק 3: עיבוד מידע

כמו בכל שפת תכנות, ב-Python יש הרבה מקרים בהם יש אוסף של נתונים (רשימה, תור, מחסנית, מערך, וכד') עליו מבצעים עיבוד מסוים. העיבוד יכול להיות כמעט כל דבר (קריאה לפונקציות, פעולות בין שני איברים עוקבים, וכד'). ברוב המקרים, העיבוד הזה ייעשה בלולאה.

בפרק זה נראה אילו פונקציות וכלים מובנים ב-Python כדי לאפשר לנו לעבד מידע בקלות, לטפל בקבצים או לקבל קלט מהמשתמש.

map, reduce, filter, lambda

כשיש לנו רשימת איברים הגיוני שנעבור עליה בעזרת לולאה. כדי שהקוד שלנו לא יתנפח מלולאות זהות שעושות את אותה עבודה שוב ושוב, נוצרו הפונקציות map, reduce ו-filter.

שלוש הפונקציות האלה הן פונקציות מובנות ב-Python, שכל מטרתן היא לקחת רשימה (או Tuple), להריץ פונקציה על האיברים שלה (כ"א משלוש הפונקציות מריצה את הפונקציה בדרך שונה, עם איבר או איברים שונים) ולהפיק פלט מתאים.

כל זה נעשה ע"י קריאה לפונקציה אחת על הרשימה, דבר שחוסך כתיבה מחדש של לולאה טריוויאלית בכל פעם מחדש (ולפעמים קצת יותר מלולאה טריוויאלית).

map

פונקצית קסם ראשונה.

map מקבלת רשימה ופונקציה. הפונקציה חייבת לקבל פרמטר אחד בלבד. map מריצה את הפונקציה עם כ"א מאיברי הרשימה (כל איבר בתורו, לפי הסדר בו הם מופיעים ברשימה).

map מחזירה רשימה חדשה, ובה כל איבר הוא התוצאה של הפונקציה עם האיבר המתאים לו ברשימה המקורית.

בעברית: map לוקחת את האיבר הראשון, מריצה עליו את הפונקציה, ושמה את התוצאה ברשימה החדשה. אח"כ היא לוקחת את האיבר השני, מריצה עליו את הפונקציה, ושמה את התוצאה באיבר הבא ברשימה החדשה. כך היא עושה לכל האיברים.

```
>>> def func(x):
...     return x * 2
...
>>> map(func, range(1, 11))
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

reduce

פונקצית קסם שניה.

reduce מקבלת רשימה ופונקציה. הפונקציה חייבת לקבל שני פרמטרים. reduce לוקחת את שני האיברים הראשונים ברשימה ומריצה את הפונקציה כאשר האיבר הראשון הוא הפרמטר הראשון והאיבר השני הוא הפרמטר השני של הפונקציה. לאחר מכן, reduce לוקחת את התוצאה של הפונקציה ומריצה את הפונקציה עם התוצאה כפרמטר הראשון והאיבר הבא מהרשימה כפרמטר השני. כך היא ממשיכה עד סוף הרשימה.

אם ברשימה יש רק איבר אחד, reduce תחזיר את האיבר היחיד ברשימה (מבלי להריץ את הפונקציה). אם הרשימה ריקה, reduce תקרוס ותדפיס הודעת שגיאה.

```
>>> def func(x, y):
...     return x + y
...
>>> reduce(func, range(1, 11))
55
```

בדוגמה הזו הודפס הסכום של כל המספרים בין 1 ל-10.

אם נפעיל את reduce עם רשימה שמכילה רק איבר אחד, reduce פשוט תחזיר את האיבר מבלי לקרוא לפונקציה. הסיבה לכך היא שהמטרה של reduce היא לצמצם רשימה לאיבר אחד וכשיש רק איבר אחד ל-reduce אין מה לעשות:

```
>>> reduce(lambda x, y: 0, [7])
7
```

כמו כן, נוכל להעביר ל-reduce פרמטר שלישי שאותו reduce תכניס לפני כל האיברים ברשימה שלנו, ובד"כ משתמשים בו כדי להתמודד עם המקרה של רשימה ריקה:

```
>>> reduce(lambda x, y: x + y, [8], 0)
8
>>> reduce(lambda x, y: x + y, [], 0)
0
>>> reduce(lambda x, y: x + y, [], 'haha')
'haha'
```

filter

פונקצית קסם שלישית.

filter מקבלת רשימה ופונקציה. הפונקציה חייבת לקבל פרמטר אחד. filter תיקח כ"א מאיברי הרשימה ותריץ את הפונקציה עם האיבר הזה. אם התוצאה של ריצת הפונקציה היא True (איננה אפס, איננה מחרוזת ריקה, ולא הופכת לאפס אם ממירים אותה ל-int), האיבר (האיבר מהרשימה שהועברה ל-filter, לא ערך ההחזרה של הפונקציה) יתווסף לרשימת התוצאה.

filter למעשה מאפשרת לנו לסנן ערכים מרשימה נתונה, בשורה אחת בלבד.

```
>>> def func(x):
...     return x % 2
...
>>>
>>> filter(func, xrange(20))
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

בדוגמה הזו הודפסו כל המספרים בין 0 ל-19 ששארית החלוקה שלהם ב-2 איננה אפס. בקיצור, כל המספרים האי-זוגיים.

ניתן כמובן להפוך את התנאי ולהדפיס את כל המספרים הזוגיים:

```
>>> def func(x):
...     return (x % 2) == 0
...
>>> filter(func, xrange(20))
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

lambda

אז מה lambda עושה בכלל ולמה היא קשורה לפרק הזה?

כמו שאפשר לראות בדוגמאות למעלה, עבור כ"א מההרצות של map, reduce או filter היינו צריכים לכתוב פונקציה חדשה כדי שנוכל להעביר פונקציה לכ"א מהפונקציות. אבל כתיבת פונקציה חדשה כל פעם זה סתם העמסה על הקוד, ולפעמים גם יכול להיות מסובך למצוא את הפונקציה (אם שמנו אותה במקום אחר).

lambda מאפשרת לנו לחסוך עוד יותר בכמות הקוד – במקום לכתוב פונקציה חדשה בכל פעם, lambda יוצרת עבורנו פונקציה ללא-שם, כבר בתוך שורת הקוד של map, reduce או filter.

בת'כלס, כאשר יוצרים פונקציה, השם שנותנים לפונקציה הוא מצביע לפונקציה עצמה. זאת גם הסיבה שכאשר כותבים שם של פונקציה, בלי סוגריים, Python כותבת לנו את שם הפונקציה (היא בעצם מדפיסה לנו את האובייקט של הפונקציה). באופן הזה, lambda יוצרת עבורנו פונקציה חדשה, ומחזירה את המצביע לפונקציה החדשה. לפונקציה החדשה אין שם, וכאשר תסתיים הרצת map, reduce או filter, הפונקציה חסרת-השם תיעלם.

דוגמה קטנה כדי להבין יותר טוב מה קורה:

יצרנו פונקציה חדשה

```
>>> def func():
...     return 5
...
>>> func
<function func at 0x00AD57A0>
>>> lambda x: x + 2
<function <lambda> at 0x00AFB3D0>
```

זאת הפונקציה

וזו פונקציית lambda חדשה שזה הרגע נוצרה.

קצת על המבנה של lambda: כתיבת פונקצית lambda כולל רק את הפרמטרים לפונקציה ואת ערך ההחזרה שלה.

הנה דוגמה לשימוש ב-map ו-lambda ביחד. הדוגמה עושה את מה שעושה הדוגמה הראשונה – מקבלת רשימת מספרים ומחזירה רשימה של האיברים המקוריים כפול 2:

```
>>> map(lambda x: x * 2, range(1, 11))
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

בדוגמה הזו כתבנו פונקצית lambda המקבלת פרמטר אחד, x, ומחזירה את x*2. כמובן, אפשר לכתוב כל ביטוי שתלוי ב-x, ואפשר גם לכלול פונקציות בביטוי הזה.

למעשה, כאשר אנו כותבים את פונקציית ה-lambda הבאה:

```
>>> f = lambda x: x * 2
```

יצרנו פונקציה, אותה יכולנו ליצור באופן הבא:

```
>>> def f(x):
...     return x * 2
...
```

התוצאה זהה לחלוטין בשני המקרים – פונקציה f שמכפילה ב-2.

שימוש ב-map, reduce ו-filter ביחד

שלוש הפונקציות לבדן אינן הסוף – אפשר לשלב כמה פונקציות כאלה ביחד: למשל, התוצאה של map תהיה הרשימה של filter, וכד'.

הנה שתי דוגמאות לשימושים כאלה:

פונקציה שמחזירה האם מספר הוא ראשוני:

```
def is_primary(n):
    return reduce(lambda x, y: x and y, map(lambda x: n % x, range(2, n)))
```

פונקציה שמחזירה את סכום הספרות במספר:

```
def sum_of_digits(n):
    return reduce(lambda x, y: x + y, map(int, map(None, str(n))))
```

קלט מהמקלדת

כאשר תוכנית מעוניינת לקבל קלט מהמשתמש (דרך המקלדת בד"כ), ניתן להשתמש בפונקציה המובנית input הפונקציה מקבלת כפרמטר מחרוזת אותה היא מדפיסה למסך, ואז היא ממתינה לשורת קלט מהמשתמש. הקלט יסתיים כאשר המשתמש יקיש על Enter. דוגמה לשימוש ב- input

```
>>> x = input('-->')
--> hello world!
>>> x
'hello world!'
```

ערך ההחזרה של הפונקציה הוא מחרוזת.

List Comprehensions

בפרק הקודם פגשנו את map, reduce ו-filter וראינו שהן פונקציות די שימושיות, כי הן מאפשרות לנו לעשות משהו נפוץ בפחות קוד ממה שהיינו צריכים לכתוב לפני שהכרנו אותן. אבל עדיין חסר משהו... בעצם, כמעט בכל הפעמים שנרצה לקרוא ל-map נצטרך גם ליצור פונקציה עם lambda. זה קצת חבל, כי הרי חיפשנו דרך לקצר ולא לחזור על עצמנו.

לשמחתנו, Python הלכה צעד אחד קדימה והכניסה את map כתחביר של ממש בשפה. כדוגמה, במקום לכתוב את הקוד הבא:

```
map(lambda x: x * 2, range(10)]
```

נוכל לכתוב:

```
[x * 2 for x in range(10)]
```

התחביר הזה נקרא List Comprehensions, וכמו שהשם מרמז הביטוי הזה מייצר רשימה:

```
>>> [x * 2 for x in range(10)]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

ב-List Comprehensions אנחנו בעצם קובעים מה התבנית שהביטוי יחזיר (בדוגמה זה x * 2), ואנחנו קובעים את השם של האיברים בעזרת תחביר דמוי לולאת-for (במקרה הזה, ... for x in).

עד כאן זה לא נראה משהו, כי בעצם החלפנו את העונש של כתיבת "lambda" בעונש אחר שבו ניאלץ לכתוב "for". זה נכון חלקית, כי קיבלנו קוד יותר קל לקריאה. אבל הכוח האמיתי הוא אם נרצה להכניס map אחד בתוך השני, וכאן זה כבר מתחיל להיות ממש קל:

```
['%02d:%02d' % (hour, minute) for hour in range(24) for minute in range(60)]
```

לא נכתוב את כל התוצאה כי היא די ארוכה... קיבלנו כאן רשימה שמכילה מחרוזות, שבכל מחרוזת יש את השעה בכל דקה של היממה:

```
['00:00', '00:01', '00:02', '00:03', '00:04', ...
```

בעצם, התחביר הזה הרבה יותר טוב מ-map כי הוא מאפשר לנו לקנן כמה לולאות בביטוי אחד ולהתייחס לכל המשתנים של כל הלולאות בבת-אחת בביטוי של התוצאה.

אבל זה לא הכל: List Comprehensions מאפשרים לנו לבצע גם את העבודה של filter כחלק מהתחביר. בואו ניקח את כל המספרים בין 0 ל-9 שמתחלקים ב-3 ונכפיל אותם ב-2:

```
>>> [x * 2 for x in range(10) if x % 3 == 0]
[0, 6, 12, 18]
```

ולדוגמה אחרונה, בואו נמצא את כל הזמנים ביממה שבהם סכום השעות, הדקות והשניות הוא 137:

```
>>> ['%02d:%02d:%02d' % (hour, minute, second)
     for hour in range(24)
     for minute in range(60)
     for second in range(60)
     if hour + minute + second == 137]
['19:59:59', '20:58:59', '20:59:58', '21:57:59', '21:58:58', '21:59:57', '22:56:59',
'22:57:58', '22:58:57', '22:59:56', '23:55:59', '23:56:58', '23:57:57', '23:58:56',
'23:59:55']
```

קבצים

טיפול בקבצים ב-Python הוא די פשוט. כדי לפתוח קובץ עלינו לקרוא לפונקציה open:

```
>>> opened_file = open('x.txt', 'w')
```

כלומר, הפונקציה open מחזירה אובייקט מסוג קובץ, כאשר הפרמטרים לפונקציה הם שם הקובץ וסוג הגישה ('w' לכתיבה, 'r' לקריאה, וכו' כמו ב-C). לאחר מכן ניתן לכתוב לקובץ באמצעות הפונקציה write:

```
>>> opened_file.write('my first file!\n')
>>> opened_file.write('my second line!')
```

בסיום העבודה עם הקובץ יש לסגור אותו באמצעות הפונקציה close:

```
>>> opened_file.close()
```

ניתן גם לפתוח קובץ לקריאה ולקרוא ממנו את כל התוכן שלו למשתנה אחד:

```
>>> file = open('x.txt', 'r')
>>> content = file.read()
>>> file.close()
```

כעת המשתנה content מכיל את תוכן הקובץ שיצרנו בדוגמה הקודמת:

```
>>> print(content)
my first file!
my second line!
```

בהמשך נראה כיצד מטפלים ב-Python בייצור של שמות הקבצים (הרי לא נפתח סתם קובץ בשם x.txt...) ואיך לייצר קבצים זמניים.

כמו כן, בהמשך נראה שניתן לוותר על הקריאה ל-()close ע"י-כך שניתן ל-Python לטפל עבורנו בסגירת הקובץ.

print

את print פגשנו מקודם – היא מדפיסה למסך. אבל עכשיו כשאנחנו כבר מכירים את Python קצת יותר טוב, בואו נבין מה אנחנו עושים כשאנחנו קוראים ל-print. השימוש הכי פשוט הוא להעביר ל-print מחרוזת אחת:

```
>>> print('Hello')
Hello
```

הפקודה יכולה לקבל פרמטר אחד (כמו בדוגמה האחרונה) או מספר פרמטרים מופרדים בפסיקים:

```
>>> print('Hello World!', 'My age is', 17)
Hello World! My age is 17
```

כפי שניתן לראות, print מוסיפה אוטומטית רווחים בין הפרמטרים שנשלחים להדפסה. כמו כן, ניתן לתת כפרמטר כל סוג משתנה, כל עוד ניתן להמיר אותו למחרוזת כדי להדפיס אותו.

בנוסף, הפקודה תמיד מדפיסה תו סיום שורה בסופה,

כמו כן, ניתן לתת ל-print מחרוזת מפורמטת:

```
>>> print ('His age is %d.' % 17)
His age is 17.
```

אבל זה בעצם לא מעניין בהקשר של print, כי פירמוט מחרוזות קשור למחרוזות עצמן, ואפשר ליצור מחרוזות לפי פורמט ולאחסן אותן במשתנה או להעביר לפונקציות.

פירמוט מחרוזות

אופרטור %

הדרך הראשונה והישנה יותר לפירמוט מחרוזות היא השימוש ב-%. כבר ראינו כיצד הפירמוט מתבצע – כותבים % אחרי המחרוזת, ואז tuple שמכיל את הפרמטרים שנרצה לפרמט:

```
>>> 'My age is %d' % (20, )
'My age is 20'
```

הדומגה הזאת קצת שונה מהדוגמאות שראינו עד עכשיו – במקום לכתוב סתם 20 כתבנו (,20). התחביר המוזר הזה הוא הדרך ב-Python ליצור tuple עם איבר אחד בלבד. בעצם, אם נכתוב (20) לא נייצר tuple אלא את המספר 20, כי Python מורידה כל סוג סוגריים סימטריים (וזה גם הרבה יותר הגיוני ש-(20) אומר 20 ולא tuple שמכיל 20).

באופן כללי, בכל מקום ב-Python שמצפה לאיברים מופרדים בפסיקים, נוכל תמיד לשים פסיק אחרי האיבר האחרון.

מאחר ש-Python מצפה לקבל tuple אחרי ה-% במחרוזות, תמיד ניצור עבורה tuple. הסיבה לכך היא שלא תמיד נדע מה הטיפוס שאנחנו מנסים לפרמט. לדוגמה, אחד הפורמטים שניתן לציין במחרוזת הוא r% (שלא קיים ב-C), והוא אומר לפרמט את הפרמטר שקיבלנו כאילו הוא היה מודפס ב-Interpreter. נסתכל על הדוגמה הבאה:

```
>>> x = 8
>>> 'Look: %r' % x
'Look: 8'
```

עד כאן הכל מצוין. ביקשנו לפרמט 8 וקיבלנו 8. אבל מה יקרה אם נעשה את זה:

```
>>> x = (1, 2, 3)
>>> 'Look: %r' % x
Traceback (most recent call last):
  File "<pyshell#251>", line 1, in <module>
    'Look: %r' % x
TypeError: not all arguments converted during string formatting
```

קיבלנו שגיאה. הסיבה לכך היא ש-Python מאפשרת לנו להעביר פרמטר אחד ל-% של המחרוזת, אבל זה מתוך איזושהי נחמדות לא מוסברת כלפי המשתמש. היא תשתמש בפרמטר האחד הזה כאילו הוא פרמטר אחד רק אם הוא לא tuple. אם הפרמטר הוא כן tuple, Python מניחה שהעברנו לה רשימת ארגומנטים ולכן היא אוטומטית מפרקת את ה-tuple. בעצם יכלנו לקבל תוצאה יותר גרועה משגיאה, והיא ש-Python תפרמט את המחרוזת, אך תעשה את זה לא נכון:

```
>>> x = (1, )
>>> 'Look: %r' % x
'Look: 1'
```

שזה בכלל לא מה שביקשנו...

לכן, מעכשיו תמיד נעביר tuple כשנפרמט מחרוזות עם %:

```
>>> x = (1, 2, 3)
>>> 'Look: %r' % (x, )
'Look: (1, 2, 3)'
```

str.format()

פירמוט מחרוזות בעזרת % היא ה"דרך הקלאסית" לפרמט מחרוזות ב-Python. הסיבה שהיא נכנסה לשימוש היא שהמבנה של המחרוזות מאוד דומה לזה שב-C (השימוש ב-%d, %s, וכו') ולכן היה קל להטמיע אותו בקרב משתמשי Python. עם זאת, המבנה הישן הוא... טוב נו, ישן... ובגלל זה הוא קצת לא אינטואיטיבי וגם לא מאפשר להרחיב אותו כדי שיתמוך בפיצ'רים של Python.

לכן, אימצו ב-Python מבנה חדש ל-Format Strings שקיים במקביל למבנה הישן. במבנה הישן תוכלו להמשיך להשתמש וכנראה שגם נמשיך לראות אותו בהרבה קוד קיים. השימוש במבנה החדש הוא לא בעזרת אופרטור %, בעיקר בגלל החיסרון שראינו בסוף הסעיף הקודם בנוגע להעברת tuple.

נסתכל על דוגמה פשוטה לפירמוט מחרוזת:

```
>>> 'Hello {0}!'.format('Moshe')
'Hello Moshe!'
```

במקרה הזה אמרנו ל-Python שאנחנו רוצים שהיא תחליף את {0} בארגומנט הראשון שנעביר ל-format. אם היינו רוצים לתמוך בשני ארגומנטים, היינו כותבים:

```
>>> 'Hello {0} {1}!'.format('Mr.', 'Moshe')
'Hello Mr. Moshe!'
```

מעולה. אז את בעיית ה-tuple פתרנו – לא צריך לחשוב יותר על מה מעבירים, כי format היא פונקציה רגילה שלא עושה בעיות כמו %.

אבל אם format היא פונקציה רגילה, אז כבר למדנו הרי איך עובדות הפונקציות המשוכללות האלה ב-Python: הן מקבלות *args ו-**kwargs וככה הן יכולות להיות גמישות בכמות הפרמטרים שלהן. אז בעצם, Python גם מאפשרת לנו לחזור פעמיים על הפרמטר הראשון:

```
>>> 'Hello {0} {0}!'.format('Moshe')
'Hello Moshe Moshe!'
```

וגם, במידה ואנחנו רוצים להעביר פרמטרים אחד אחרי השני, אין צורך בכלל לציין את האינדקס שלהם:

```
>>> 'Hello {}!'.format('Moshe')
'Hello Moshe!'
```

שזה כבר משמעותית יותר נוח מ-%.

אם נרצה, נוכל להשתמש בהעברת Keyword Arguments:

```
>>> 'Hello {name}!'.format(name='Moshe')
'Hello Moshe!'
```

וגם לבקש להדפיס דברים כמו ב-Interpreter (מקביל ל-%r):

```
>>> 'Hello {name!r}'.format(name='Moshe')
"Hello 'Moshe'"
>>> 'Hello {!r}'.format('Moshe')
"Hello 'Moshe'"
>>> 'Hello {0!r}'.format('Moshe')
"Hello 'Moshe'"
```

כמובן, format יתלונן אם לא נעביר לו את הפרמטרים שהוא ציפה להם:

```
>>> 'Hello {}'.format()

Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
    'Hello {}'.format()
IndexError: tuple index out of range
```

אבל כן חשוב לדעת ש-format מתעלם מפרמטרים מיותרים:

```
>>> 'Hello'.format('Moshe')
'Hello'
>>> 'Hello {}'.format('Moshe', 'David')
'Hello Moshe'
```

הסיבה לכך נעוצה בעיקר בשימוש ב-Keyword Arguments, והיא שבשיטה הזו אנו יכולים לתת יותר פרמטרים ממה שהיינו צריכים, ו-format ייקח רק את אלה שהוא חייב בשביל לפרמט את המחרוזת. כך לא נצטרך ליצור אובייקט מיוחד רק עבור פירמוט מחרוזות.

כשנראה בהמשך איך יוצרים אובייקטים משלנו נבין יותר טוב למה format הרבה יותר נוחה מ-%.

חלק 4: אובייקטים

ב-Python כל דבר הוא אובייקט. התכונה הזאת נקראת ברוב שפות התכנות Object-Oriented, אך בניגוד להרבה שפות אחרות שרק תומכות באובייקטים, Python באמת מייצגת כל דבר בה בעזרת אובייקטים. מספרים, מחרוזות, רשימות, מילונים, tuple-ים, None, ואפילו פונקציות – כולם אובייקטים.

כל מה שראינו עד עכשיו היה מבוא מאוד נחמד שבו שיחקנו עם פיצ'רים של השפה אבל לא באמת הבנו מה אנחנו עושים. בפרק זה נראה בדיוק מה האובייקטים בשפה עושים, מה משותף ביניהם ונבין גם למה חלק מהם מתנהגים כמו שהם מתנהגים.

id() ו-is

בפרקים הקודמים אמרנו שאנחנו לא משווים ל-None עם האופרטור == אלא בעזרת האופרטור is. אפילו ציינו את הסיבה לכך, והיא ש-None הוא Singleton, כלומר יש רק אחד כזה. העובדה שיש רק None אחד אומרת שלא מעניין אותנו להשוות משתנה מסוים ל-None. לצורך העניין, ההשוואה לא רלוונטית כאן, כי השוואה בודקת את התוכן של האובייקט. אנחנו רוצים לבדוק האם המשתנה שלנו מצביע לאובייקט, ולכן התוכן בכלל לא משנה.

כדי שנוכל לדעת האם שני אובייקטים הם אותו אובייקט בדיוק או שני אובייקטים עם תוכן זהה, קיימת ב-Python פונקציה מובנית בשם id(). הפונקציה id() מחזירה לנו מספר כלשהו, כאשר המספר הזה שונה עבור כל אובייקט, אבל אם נקבל את אותו מספר עבור שני משתנים נוכל לדעת בוודאות שהם מצביעים לאותו אובייקט. בפועל, המספר הזה הוא הכתובת של האובייקט בזיכרון (ולמרות שאין לנו מה לעשות עם כתובות ב-Python, נחמד לדעת את זה).

דוגמה פשוטה לשימוש ב-id():

```
>>> id(None)
4296516488
>>> id(0)
4298191184
>>> id([])
4299637824
```

כך נוכל להשוות בין ה-id()-ים של שני משתנים ולדעת האם הם מצביעים לאותו אובייקט:

```
>>> x = None
>>> y = None
>>> id(x) == id(y)
True
```

מאחר שהשוואת id()-ים היא פעולה נפוצה, ב-Python קיים אופרטור עבור ההשוואה הזו, והוא האופרטור is:

```
>>> x = None
>>> y = None
>>> z = 3
>>> x is y
True
>>> x is z
False
```

במקביל, קיים האופרטור is not שבודק את ההפך (כלומר ששני id()-ים שונים):

```
>>> x is not y
False
>>> y is not z
True
```

אמנם ניתן לממש את "X is not Y" בעזרת "not X is Y", אך הגרסה "X is not Y" הרבה יותר קלה לקריאה ולכן הוכנסה לשפה.

כדי להבין את אופרטור is ואת ההתנהגות של משתנים, ניצור שני משתנים מספריים:

```
>>> x = 1000
>>> y = 1000
```

כעת, אם ננסה להשוות ביניהם נגלה שהם שווים, בדיוק כמו שהיינו מצפים:

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

אבל, הם לא אותו האובייקט:

```
>>> x is y
False
```

כשכתבנו x=1000 גרמנו ליצירת אובייקט מסוג int שהתוכן שלו הוא 1000. כשכתבנו y=1000 יצרנו אובייקט נוסף מאותו סוג ועם אותו תוכן, אבל אלה שני אובייקטים שונים. אם נרצה לייצר שני משתנים שמצביעים לאותו אובייקט, נכתוב:

```
>>> x = 1000
>>> y = x
```

ואז נקבל את התוצאה הרצויה:

```
>>> x == y
True
>>> x is y
True
```

נקודה אחרונה שחשוב להבהיר לפני שנמשיך היא שאם ננסה להריץ את הדוגמאות האלה עבור מספרים קטנים יותר, ניתקל בתופעה משונה:

```
>>> x = 1
>>> y = 1
>>> x is y
True
```

לכאורה מה שאמרנו כאן לא נכון.

הסיבה לכך היא ש-Python שומרת עותקים מוכנים של המספרים השלמים שבין 5- ל-256. הסיבה לכך היא שאלה מספרים די נפוצים ו-Python מעדיפה לשמור עותקים שלהם במקום ליצור אותם כל פעם מחדש.

type

עכשיו כשיש בידינו את דרך להבחין בין אובייקטים שונים, נמשיך ונסתכל על אחת התכונות של אובייקטים שכבר פגשנו מוקדם יותר: לכל אובייקט (מספר, מחרוזת, פונקציה, ...) יש סוג (type). את ה-type של אובייקט נוכל לקבל ע"י קריאה לפונקציה המובנית type:

```
>>> type(0)
<type 'int'>
>>> type('abc')
<type 'str'>
>>> type([])
<type 'list'>
```

עד כאן אין הרבה הפתעה, כי מספר הוא int, מחרוזת היא str ורשימה היא אכן list. אבל הרי אמרנו שכל דבר ב-Python הוא אובייקט, ולכן גם הטיפוס של אובייקט הוא אובייקט. בעצם, type זאת לא פונקציה שמדפיסה על המסך טקסט שאומר מה הטיפוס של משהו, היא מחזירה את ה-type object שלו. אז אם פונקציה מחזירה ערך, אנחנו יכולים לשמור את הערך הזה במשתנה:

```
>>> x = type(0)
>>> x
<type 'int'>
```

או לדוגמה להשוות בין שני טיפוסים של אובייקטים כדי לראות האם הם מאותו סוג:

```
>>> type(1) is type(2)
True
>>> type([]) is type(())
False
```

שימו לב לשימוש ב-is: אנחנו הרי רוצים לדעת שהטיפוס הוא אותו טיפוס (אין משמעות לתוכן של ה-type object).

Type Objects

יש הרבה מספרים בעולם. יש גם הרבה רשימות, והרבה tuple-ים והרבה מילונים. אבל לכל אחד מאלה יש רק type object אחד. במקרה של המספרים, ה-type object שלהם הוא int:

```
>>> type(5) is int
True
```

כן כן, זה אותו int שפגשנו במבוא (שם קראנו לו פונקציה, וזה היה שקר לבן). אותו int משמש גם כדי להמיר אובייקטים אחרים ל-int:

```
>>> int(7.0)
7
>>> int('4')
4
>>> int('   50')
50
```

במחרוזות הסידור די דומה – ה-type object נקרא str:

```
>>> type('Moshe') is str
True
```

וגם הוא יכול לשמש כדי להפוך דברים למחרוזות (שימו לב שבדומה הזו מודפסות מחרוזות שמכילות את הייצוג הטקסטואלי של האובייקטים שהמרנו למחרוזת, ולמרות שהן נראות כמו אובייקטים, הן מחרוזות רגילות לחלוטין):

```
>>> str(5)
'5'
>>> str(7.0)
'7.0'
>>> str(range(10))
'[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]'
```

אז בעצם, type objects הם גם מעין פונקציות שמייצרות אובייקטים חדשים מהסוג של עצמן, וגם משמשים אותנו כדי שנוכל לזהות את האובייקטים אחר-כך ולהגיד מה הסוג של האובייקט.

בפרק הבא נראה גם ש-type objects הם אלה שקובעים את ההתנהגות של האובייקטים, ונראה מהי בדיוק התנהגות של אובייקט ואיך אפשר לשנות את ההתנהגויות האלה ע"י-כך שניצור type objects משלנו.

Immutable ו-Mutable

בינתיים נסתפק בכך ש-type objects קיימים ובכך שאנחנו יודעים להבדיל בין סוגים שונים של אובייקטים. אבל לא נסתפק בכך שעדיין יש הבדלים מהותיים בין האובייקטים שפגשנו עד עכשיו. יש את המספרים, המחרוזות, ה-tuple-ים ואת None. ה-type-ים האלה הם סבבה, כי הם לא משתנים. מהרגע שיצרנו אותם הערך שלהם קבוע. נבהיר את זה:

```
>>> x = y = 1
>>> id(x), id(y)
(13637560, 13637560)
>>> x += 1
>>> x, y
(2, 1)
>>> id(x), id(y)
(13637536, 13637560)
```

מה שקרה כאן הוא שכאשר ניסינו לשנות את הערך של האובייקט 1 מסוג int, הערך שלו לא השתנה, אלא נוצר אובייקט חדש ו-x עבר להצביע למשתנה החדש.

ההתנהגות הזאת היא ההפך הגמור מההתנהגות של שאר האובייקטים. לדוגמה, נסתכל על רשימות:

```
>>> x = y = []
>>> id(x), id(y)
(139838462741248, 139838462741248)
>>> x.append(0)
>>> x, y
([0], [0])
>>> id(x), id(y)
(139838462741248, 139838462741248)
```

כאן x ו-y מצביעים לאותה הרשימה, ושני המשתנים מושפעים כשאנחנו משנים את התוכן של הרשימה. אז מה בעצם ההבדל בין מספרים לרשימות? ההבדל הוא שאובייקטים מתחלקים לשני סוגים:

- Immutable objects: אובייקטים שהם Immutable לא ניתנים לשינוי מהרגע שיצרנו אותם. אלה הם המספרים, ה-tuple-ים, None (שכבר נוצר עבורנו) ומחרוזות.

- Mutable objects: אלה כל שאר האובייקטים, ואפשר לשנות אותם אחרי שנוצרו.

החלוקה הזאת קיימת בעיקר עבור מילונים. כשפגשנו את המילונים בפעם הראשונה לא טרחנו לציין עובדה די חשובה לגביהם – ה-keys במילון חייבים להיות immutable. אם ננסה להכניס למילון key שהוא mutable, נקבל שגיאה:

```
>>> {[]:[]}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

(לפני שנמשיך, וודאו שהבנתם שניסינו ליצור מילון שבו יש מפתח [] שמצביע לערך []).

הסיבה שמילון לא אוהב שה-keys הם mutable היא שהמילון לא יכול להרשות לעצמו שנכניס לתוכו זוג (key, value) ואז נשנה את ה-key מתחת לרגליים שלו. אם נשנה את ה-key אז נוכל ליצור התנגשות (שני keys שונים בתוך אותו מילון), וגם נפריע לסידור הפנימי של המילון (איך הוא ימצא את ה-key אם נשנה את הערך שלו?)

לכן, יש לנו טיפוסים שהם immutable שבהם בטוח להשתמש כמפתחות במילון.

בפרק הבא נראה כיצד type objects משפיעים על ההתנהגות של אובייקט בהקשר ה-mutability ואפילו נראה כיצד אפשר ליצור אובייקטים משלנו שיכולים לשמש כ-keys במילון.

Attributes ו-()dir

בפרק המבוא ראינו שכדי להוסיף איבר לרשימה משתמשים ב-()append:

```
>>> l = []
>>> l.append(0)
>>> l
[0]
```

append הוא Attribute של list-ים. Attribute-ים למיניהם גם הם אובייקטים (כי הרי כל דבר ב-Python הוא אובייקט):

```
>>> [].append
<built-in method append of list object at 0x103bb4440>
```

וכמובן יש type ל-append, כי לכל אובייקט יש type:

```
>>> type([].append)
<type 'builtin_function_or_method'>
```

אז Attribute (לעיתים נשתמש בקיצור attr) הוא בסה"כ אובייקט שמוכל בתוך אובייקט אחר. ההבדל בין attr לבין כל אובייקט ש"סתם" מוכל בתוך אובייקט אחר (למשל, רשימה מכילה אובייקטים שהכנסנו לתוכה) הוא של-attr-ים יש שם. כשאנחנו רוצים לפנות ל-attr מסוים אנחנו משתמשים בשם שלו.

כדי לראות את רשימת כל ה-attributes שיש לאובייקט, משתמשים בפונקציה המובנית ()dir:

```
>>> dir([])
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__delslice__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
'__getslice__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__iter__',
'__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__',
'__setslice__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'count',
'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

ה-attributes שמתחילים ונגמרים בשני קווים תחתונים נקראים slots, ועל רובם נדבר בפרק הבא.

אחרי ה-slots למיניהם אפשר לראות את כל ה-attributes ש-list מציעה לנו: append, count, extend, index, insert, pop, remove, reverse ו-sort. כדי לגלות מה עושה כל אחד מה-attributes האלה, נוכל לקרוא ל-help() עליו. לדוגמה:

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

sort(...)
    L.sort(cmp=None, key=None, reverse=False) -- stable sort *IN PLACE*;
    cmp(x, y) -> -1, 0, 1
```

וכך גילינו שאפשר למיין רשימה ע"י קריאה ל-sort() ואילו פרמטרים sort מקבלת במידה ונרצה לשלוט על הצורה שבה המיון מתבצע. באופן כללי נוכל לקרוא ל-help([]) ולקבל את התיעוד של ה-attribues של הרשימה, ושל כל אובייקט שנפגוש.

Reference Counting

כשאנחנו כותבים שורת קוד כמו זו:

```
x = 1000
```

Python טורחת ויוצרת עבורנו אובייקט חדש מסוג int ומכניסה לתוכו את הערך 1000. נוכל גם להמשיך ולהשתמש במשתנה x:

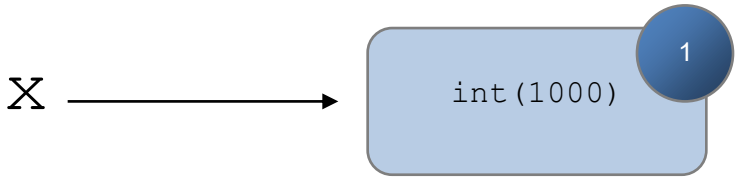
```
y = x
```

ו-Python יודעת ש-x הוא אובייקט כלשהו וש-y מצביע לאותו אובייקט ש-x מצביע אליו. אבל מה יקרה אם נכתוב פתאום את השורה הבאה:

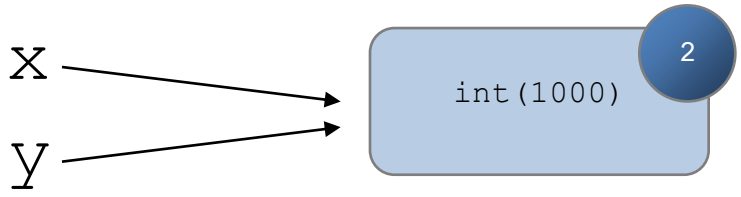
```
x = y = 0
```

אף אחד לא צריך יותר את ה-1000 שיצרנו מקודם. יותר גרוע מזה, אין לנו איך לגשת ל-1000 הזה אפילו אם נרצה. האם Python יודעת שה-1000 הזה לא נחוץ יותר או שאולי היינו צריכים לעשות משהו כדי לסיים את השימוש באותו מספר?

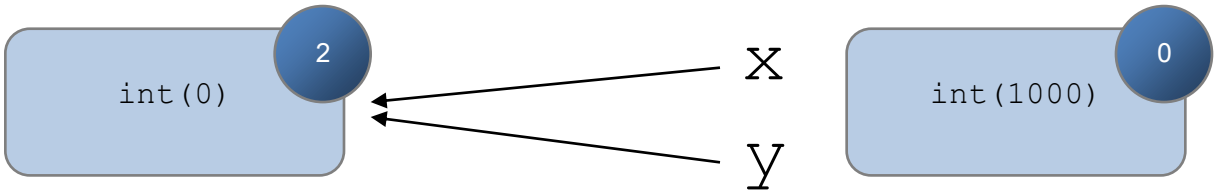
ברור ש-Python יודעת וברור שלא נצטרך לעשות כלום. הדרך שבה Python יודעת שסיימנו להשתמש באובייקט מסוים היא ע"י טכניקה שנקראת Reference Counting: לכל אובייקט מוצמד מונה שסופר כמה משתנים מצביעים אליו. כאשר יצרנו את ה-1000 שלנו והצבענו אליו עם x, יצרנו את המצב הבא:



לאחר מכן, הצבענו ל-1000 עם y, ו-Python זכרה את זה ע"י-כך שהיא הוסיפה 1 ל-Reference count של 1000:



לבסוף, כשסיינו את ההצבעה של x ו-y לאובייקט אחר, ה-Reference count של 1000 ירד ל-0:



בנקודה הזו, Python יודעת שה-int שלנו לא בשימוש יותר והיא מסמנת אותו. באיזושהי נקודה שבה יהיה ל-Python נוח, היא תנקה את האובייקטים המיותרים. תהליך הניקוי הזה נקרא Garbage collection ולא נכסה את התהליך הזה במסגרת הלימוד שלנו.

עלינו לזכור רק שב-Python אין לנו דרך לדעת מתי יתבצע Garbage collection, מאחר שאותו תהליך ניקוי תלוי במנגנונים נוספים שעוד לא ראינו (כמו Exceptions) וכן Python שומרת לעצמה את הזכות לשנות את השיטה לפיה היא מבצעת Garbage collection כדי שאפשר יהיה לשנות אותה בין גרסאות מבלי לפגוע בקוד קיים.

קבצים

כדוגמה לנושא של Garbage collection, נסתכל על אובייקטים שפגשנו בפרק הקודם – הקבצים.

קובץ הוא אובייקט די פשוט. אחרי שיצרנו אובייקט file, אנחנו יכולים לבצע מולו מספר די מצומצם של פעולות (read/write/seek) ובסוף עלינו לסגור אותו.

אבל אם Python מבצעת Garbage collection, אולי אנחנו לא צריכים לסגור את הקובץ בכלל? תלוי. אם לא אכפת לנו מתי הקובץ ייסגר, אז בהחלט אפשר לנטוש את אובייקט הקובץ (לדוגמה ע"י כך שפשוט נצא מפונקציה שבה הקובץ הוא משתנה לוקאלי) ואז Python תשמיד את האובייקט מתישהו ועל הדרך גם תסגור את הקובץ.

במקרים בהם כן נרצה לדעת בדיוק מתי הקובץ נסגר – למשל אם נרצה לקרוא קובץ שהרגע סיימנו לכתוב, או אם נרצה למחוק קובץ כשהוא עדיין פתוח – נהיה חייבים לסגור את הקובץ בצורה מפורשת ע"י קריאה ל-close().

בד"כ נפתח קבצים לקריאה או לכתיבה בלבד, ולכן לא יעניין אותנו לוודא שהקובץ נסגר. כדוגמה, במקרה הפשוט שבו נרצה לקרוא את התוכן של קובץ מסוים, נוכל לעשות את זה בשורה אחת בלבד:

```
>>> open('some_file.txt', 'r').read()
'Some file contents...'
```

למתעניינים, ניתן לקרוא על Context Managers ולראות כיצד אפשר להשתמש בקבצים תוך כדי שניהנה מכל העולמות – גם נדע בדיוק מתי הם נסגרים, אך מבלי לשמור אותם במשתנה מיוחד או לקרוא ל-close().

לבינתיים נסתפק בכך שאין צורך לקרוא ל-close() מפורשות, מאחר שזהו המקרה הנפוץ.

חלק 5: מחלקות

הטיפוסים המובנים ש-Python מספקת לנו הם די טובים, אבל הם לא תמיד יספקו אותנו. מה אם נרצה ליצור רשימה שלא מסכימה לאחסן יותר מ-5 איברים? או אולי נרצה ליצור אובייקט שנראה כמו tuple אבל מכיל attributes שמאפשרים לנו לקבל את המקומות 0 ו-1 כדי ליצור טיפוס של נקודה (x, y)?

class

כדי להתגבר על הדרישות שיכולות להיות לכל מפתח שהוא, Python מאפשרת לנו ליצור סוגי אובייקטים משלנו. כדי ליצור סוג חדש של אובייקט, נשתמש במילה השמורה class:

```
>>> class Greeter(object):
...     def greet(self):
...         print 'Hello there!'
```

class היא הגדרה של מחלקה חדשה. למחלקה החדשה קוראים Greeter במקרה הזה, ויש בתוכה מתודה אחת בשם greet שמדפיסה למסך הודעה. מתודה (method) היא פונקציה שפועלת על אובייקט מסוים (להבדיל מ"סתם" פונקציה). הפרמטר self ש-greet מקבלת הוא פרמטר שחייב להיות הפרמטר הראשון בכל מתודה במחלקה. self הוא למעשה משתנה המכיל את האובייקט עליו תורץ הפונקציה, כי אם נריץ "סתם פונקציה", היא לא יכולה לדעת על איזה מופע של Greeter עליה לפעול.

את המשמעות של (object) נבהיר בהמשך.

השימוש במחלקה שיצרנו מאוד פשוט:

```
>>> x = Greeter()
>>> x.greet()
Hello there!
```

x נקרא מופע (Instance) של Greeter. אם נרצה, נוכל ליצור כמה instance-ים של Greeter:

```
>>> y = Greeter()
>>> x is y
False
```

וקיבלנו Greeter אחר. x ו-y הם שני אובייקטים שונים, אבל יש להם משהו אחד משותף, הרי שניהם instance-ים של Greeter. נוכל לדעת את זה ע"י קריאה לפונקציה type():

```
>>> type(x)
<class '__main__.Greeter'>
>>> type(x) is Greeter
True
```

בעצם, Greeter הוא type-object חדש שאנחנו יצרנו.

נוכל גם לראות שמתודה היא בסה"כ מעטפת יפה לקריאה נוחה לפונקציות על אובייקט:

```
>>> Greeter.greet(x)
Hello there!
>>> x.greet()
Hello there!
```

בדוגמה הזו רואים בצורה מאוד ברורה ש-x מועבר כפרמטר self כשהמתודה נקראת, ו-self הוא ה-instance עליו פועלת המתודה כשהיא נקראת. במקרה של המחלקה שלנו לא עשינו שום דבר עם self, אך מיד נראה מדוע אנחנו צריכים את ה-instance.

__init__

אם אתם זוכרים, כשפגשנו את dir() ראינו שלאובייקטים יש הרבה attributes שמתחילים ונגמרים בשני קווים תחתונים (מעכשיו פשוט נכתוב __). ה-attributes האלה נקראים slots, ומיד נראה שרוב ה-attributes האלה הם מתודות.

המתודה הראשונה שנכיר היא __init__, והיא המתודה שנקראת כשנוצר אובייקט חדש מה-class שלנו:

```
>>> class Greeter(object):
...     def __init__(self):
...         print "I'm alive!"
...     def greet(self):
...          print 'Hello there!'
...
>>> g = Greeter()
I'm alive!
>>> g.greet()
Hello there!
```

בנוסף לדוגמה הזו, __init__ יכולה לקבל פרמטרים: כל פרמטר שנעביר בסוגריים ל-Greeter בעת יצירת האובייקט יועבר כפרמטר ל-__init__ בנוסף לפרמטר self (אותו Python מעבירה עבורנו בצורה אוטומטית).

בדוגמה הזו נראה העברת פרמטר ל-__init__ וכיצד המחלקה שלנו שומרת את הפרמטר הזה כ-attribute חדש:

```
>>> class Greeter(object):
...     def __init__(self, greeting):
...         self.greeting = greeting
...     def greet(self):
...         print self.greeting
...
>>> g = Greeter('Yeah hello...')
>>> g.greet()
Yeah hello...
```

מחלקה יכולה לשמור attributes פשוט ע"י הצבה שלהם ב-self.

דוגמה אחרת שממחישה את אותה נקודה בצורה אחרת:

```
>>> dir(g)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__', '__getattribute__',
'__hash__', '__init__', '__module__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__',
'greet', 'greeting']
>>> g.xxxxxxxxx = 1
>>> dir(g)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__', '__getattribute__',
'__hash__', '__init__', '__module__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__',
'greet', 'greeting', 'xxxxxxxxx']
```

למעשה, ל-Python לא אכפת מי קורא או כותב ל-attributes באובייקטים, וכמו שראינו בשתי הדוגמאות האחרונות, אפשר לקרוא ולכתוב כל attribute מתוך מתודה של המחלקה או מחוץ למחלקה. ב-Python אין שום הבדל בין שתי הגישות.

כעת נשכלל מעט את הדוגמה של המחלקה Greeter:

```
>>> class Greeter(object):
...     def __init__(self, greeting):
...         self.set_greeting(greeting)
...     def set_greeting(self, greeting):
...         self._greeting = greeting.strip().title()
...     def greet(self):
...         print self._greeting
...
>>> greeter = Greeter('Hello')
>>> greeter.greet()
Hello
>>> greeter.set_greeting('Hola')
>>> greeter.greet()
Hola
>>> greeter._greeting
'Hola'
```

כאן אנחנו רואים כמה דברים מעניינים. קודם כל, אפשר לקרוא למתודות מתוך המחלקה ע"י שימוש ב-self. מאחר שאפשר לקרוא למתודות של המחלקה מתוך עצמה, אנחנו משתמשים במתודה set_greeting() כדי לאתחל את self._greeting, ובצורה הזו אנחנו גם מאפשרים למשתמש לשנות את הברכה שתודפס כשהוא יקרא ל-greet().

בנוסף, את הברכה שהמחלקה מדפיסה אחסנו ב-attribute שנקרא greeting_ ולא greeting כמו בדוגמה הקודמת. הסיבה לכך היא שב-Python אין באמת דרך להחביא attributes כך שלא ניתן יהיה לגשת אליהם מחוץ למחלקה. אמנם קיימת שיטה להקשות על המשתמש לגשת ל-attributes פנימיים (ומיד נראה איך), אבל מאחר שהשיטה רק מקשה ולא באמת פותרת את בעיית הגישה מבחוץ, לא מקובל להשתמש בה. לכן, כשנרצה לסמן ש-attribute מסוים הוא לשימוש פנימי של המחלקה פשוט נשים _ לפני השם שלו.

Attributes פנימיים

כמו שראינו לפני שתי שורות, לפעמים נרצה שמישהו שמשתמש במחלקה שלנו בצורה חיצונית לא יוכל לגעת ב-attributes פנימיים למחלקה. דוגמה ל-attribute פנימי הוא הברכה ש-Greeter מדפיסה, כי יכול להיות שבנוסף לברכה אנחנו גם שומרים במחלקה את השפה שבה עלינו להדפיס ואם המשתמש ישנה את הברכה בלי לשנות את השפה נדפיס ברכה לא נכונה (למשל בכיוון ההדפסה, ימין לשמאל או שמאל לימין).

כדי לפתור את הבעיה הזו, Python מאפשרת לנו להגדיר attributes בצורה הבאה:

```
>>> class Greeter(object):
...     def __init__(self, greeting):
...         self.set_greeting(greeting)
...     def set_greeting(self, greeting):
...         self.__greeting = greeting
...         self.__politeness = 0
...     def greet(self):
...         print self.__greeting
...         self.__politeness += 1
...     def politeness(self):
...         return self.__politeness
...
>>> greeter = Greeter('Hello')
>>> greeter.greet()
Hello
>>> greeter.greet()
Hello
>>> greeter.politeness()
2
```

כאן יש לנו מחלקה שסופרת כמה פעמים היא הדפיסה את הברכה שאמרו לה להדפיס. בכל פעם שהברכה מודפסת, מדד הנימוס ("politeness") עולה ב-1, כי כמה שנברך יותר פעמים ניחשב יותר מנומסים. אבל, אם נשנה את הברכה ניאלץ לאפס את מדד הנימוס שלנו, כי הוא כבר לא יהיה נכון מאחר שהברכה התחלפה.

לכן, הגדרנו את greeting__ ו-politeness__ עם שני קווים תחתונים לפניהם. נסתכל על dir(greeter):

```
>>> dir(greeter)
['_Greeter__greeting', '_Greeter__politeness', '__class__', '__delattr__', '__dict__',
'__doc__', '__format__', '__getattribute__', '__hash__', '__init__', '__module__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', '__weakref__', 'greet', 'politeness', 'set_greeting']
```

ה-attributes שהגדשנו הם בעצם politeness__ ו-greeting__. כשאנחנו ניגשים ל-attribute שמתחיל בשני קווים תחתונים אבל לא מסתיים בשני קווים תחתונים, Python תמיד מוסיפה לפני קו תחתון ואת שם המחלקה שבה אנחנו רצים כרגע. אם אין מחלקה, נקבל שגיאה וכך Python מגינה מפני גישה חיצונית:

```
>>> greeter.__greeting
Traceback (most recent call last):
  File "<pyshell#51>", line 1, in <module>
    greeter.__greeting
AttributeError: 'Greeter' object has no attribute '__greeting'
```

אבל אם נרצה כמובן נוכל לגשת לשם המפורש:

```
>>> greeter._Greeter__greeting
'Hello'
```

כך שהמנגנון הזה הוא מנגנון הגנה מפני טעויות, ולא מפני משתמש שיודע מה הוא עושה. לכן, מקובל להשתמש בשיטה שראינו בסוף הסעיף הקודם (הוספת קו תחתון בודד לפני שמות של attributes פנימיים) ולא במנגנון ש-Python מספקת.

__repr__, __str__

אם ננסה להדפיס אובייקטים של המחלקה Greeter ב-interpreter נגלה שה-interpreter לא יודע להגיד עליה יותר מדי:

```
>>> greeter
<__main__.Greeter object at 0x00000000028CE828>
```

וגם נקבל את אותה התוצאה אם ננסה להמיר את greeter למחרוזת:

```
>>> str(greeter)
'<__main__.Greeter object at 0x00000000028CE828>'
```

כדי שנוכל להמיר מחלקה למחרוזת וכדי שהיא תודפס כמו שצריך ב-interpreter נצטרך לממש את ה-slot-ים __str__ ו-__repr__. המתודה __repr__ תיקרא לצורך הדפסה ב-interpreter או כשנקרא מפורשות לפונקציה ()repr. המתודה __str__ תיקרא כשנקרא ל-()str עם האובייקט שלנו. אם לא נממש את __str__, Python תקרא עבורנו למימוש של __repr__, כך שאם אין לנו רעיון טוב למימוש עבור __str__ מספיק לממש רק את __repr__. לצורך הדוגמה נממש מחלקה שמייצגת נחש:

```
>>> class Snake(object):
...     def __init__(self, length):
...         self.length = length
...     def __repr__(self):
...         return 'Snake({})'.format(self.length)
...     def __str__(self):
...         return '-' * self.length + '>'
...
>>> snake = Snake(5)
>>> snake
Snake(5)
>>> str(snake)
'----->'
>>> repr(snake)
'Snake(5)'
```

אז מה ההבדל בין __str__ ל-__repr__? למה לא היה מספיק שיהיה רק אחד מהם? ההבדל הוא ש-__str__ הוא slot למתודה שתחזיר מחרוזת שתייצג את האובייקט שלנו עבור המשתמש, כלומר עבור בן-אדם. שימוש אפשרי ל-__str__ הוא להדפיס את התוצאה שלו למסך כחלק מאיזושהי אינטראקציה עם המשתמש.

__repr__ לעומת זאת הוא slot למתודה שתחזיר, בתקווה, מחרוזת שמייצגת את האובייקט כאובייקט. המטרה כאן היא שאם ניקח את התוצאה של __repr__ ונעתיק אותה ל-interpreter נקבל בחזרה את האובייקט שהיה לנו במקור. במקרה של המחלקה Snake עשינו את זה די בקלות, אבל יש הרבה מחלקות אחרות שעבורן לא נוכל לממש מתודה __repr__ שתאפשר לנו לשחזר את האובייקט המקורי שהיה לנו ולכן רק נשתדל להחזיר מחרוזת שתתאר את האובייקט בצורה מספיק אינפורמטיבית לצורך הדפסה ב-interpreter.

__getitem__, __setitem__, __delitem__

שלושה slot-ים שימושיים הם __getitem__, __setitem__ ו-__delitem__,והם ה-slot-ים שנקראים כשאנחנו פונים לאובייקט עם סוגריים מרובעים. כדוגמה, נממש אובייקט שנראה כמו רשימה, אבל לא מאחסן יותר מ-5 איברים:

```
>>> class Only5Items(object):
...     def __init__(self):
...         self._items = []
...     def __getitem__(self, item):
...         return self._items[item]
...     def __setitem__(self, item, value):
...         self._items[item] = value
...     def __delitem__(self, item):
...         del self._items[item]
...     def append(self, item):
...         self._items.append(item)
...         if len(self._items) > 5:
...             self._items.pop(0)
...     def __repr__(self):
...         return repr(self._items)
...
```

נבדוק קודם שהמתודה append עובדת כמו שרצינו:

```
>>> x = Only5Items()
>>> x
[]
>>> for i in range(1, 6):
...     x.append(i)
>>> x
[1, 2, 3, 4, 5]
>>> x.append(6)
>>> x
[2, 3, 4, 5, 6]
```

מעולה. כעת נראה איך __getitem__ עובדת:

```
>>> x[0]
2
>>> x[-1]
6
```

שימו לב ש-__getitem__ עובדת בדיוק כמו רשימה רגילה (תוכלו לבדוק את אותו הקוד על רשימה רגילה ב-interpreter בעצמכם). כעת נבדוק את __setitem__:

```
>>> x[0] = 9
>>> x
[9, 3, 4, 5, 6]
```

ולבסוף נבדוק את __delitem__. שימו לב ש-__delitem__ מדגימה שימוש ב-statement חדש בשם del שעוד לא פגשנו. del יכולה לשמש אותנו כדי למחוק משתנים, אבל גם כדי למחוק איבר מ-container כמו מילון או רשימה:

```
>>> del x[-1]
>>> x
[9, 3, 4, 5]
>>> del x[0]
>>> x
[3, 4, 5]
>>> del x[1]
>>> x
[3, 5]
```

בדוגמה הזו מימשנו אובייקט שמגיב ל-items מספריים, אבל באותה קלות נוכל לממש אובייקט שמקבל items כלשהם:

```
>>> class ScumbagDict(object):
...     def __init__(self):
...         self._items = {}
...     def __repr__(self):
...         return repr(self._items)
...     def __getitem__(self, item):
...         return self._items[item]
...     def __setitem__(self, item, value):
...         self._items[item[::-1]] = value
...
>>> d = ScumbagDict()
>>> d['sunday'] = 1
>>> d['monday'] = 2
>>> d
{'yadnus': 1, 'yadnom': 2}
```

האובייקט במקרה הזה הוא מילון מרושע שהופך את ה-key בכל פעם שאנחנו מכניסים לתוכו איבר, כך שנצטרך להפוך את ה-key כשנרצה לקבל את האיבר מהמילון. לא שימושי במיוחד בתוכנית אמיתית אבל אחלה דוגמה.

שימו לב שלא מימשנו את __delitem__ ולכן האובייקט שלנו לא תומך ב-del. בעצם נוכל לממש רק את ה-slot-ים שנראה לנכון, וכל slot שלא נממש הוא פונקציונליות שהאובייקט שלנו לא יתמוך בה. כמובן שאם ננסה לבצע del במקרה הזה נקבל שגיאה, ונקבל שגיאה אם ננסה להשתמש ב-get/set ל-item כשלא נממש את __setitem__/__getitem__.

ירושה

בדוגמה האחרונה פגשנו את ScumbagDict. בסה"כ הוא די בסדר, חוץ מ-slot אחד קצת דפוק. נגיד שנרצה לתקן אותו, האם נצטרך לממש את כל המילון מחדש? הרי רק מתודה אחד לא טובה בו וכל השאר בסדר.

ב-Python יש מנגנון בשם ירושה (Inheritance). ירושה מאפשרת לנו לקחת אובייקטים קיימים ולהחליף בהם מתודות:

```
>>> class NiceDict(ScumbagDict):
...     def __setitem__(self, item, value):
...         self._items[item] = value
...
>>> d = NiceDict()
>>> d['sunday'] = 1
>>> d['monday'] = 2
>>> d
{'sunday': 1, 'monday': 2}
```

במקרה הזה, NiceDict הוא השם של המחלקה והיא יורשת מ-ScumbagDict. במושגים, NiceDict היא subclass של ScumbagDict ו-ScumbagDict היא superclass של NiceDict. הסוגריים אחרי שם המחלקה אומרים מהי המחלקה

שאנחנו יורשים ממנה, והרגע גילינו שכל המחלקות שראינו עד עכשיו ירשו מ-object. מיד נסביר מיהו אותו object ומה משמעות הירושה ממנו.

הפעולה שביצענו עבור __setitem__ נקראת "דריסה" (overriding), כלומר דרסנו את __setitem__ של ScumbagDict והחלפנו אותו בגרסה אחרת משלנו.

כעת, נשתכלל קצת. נתחיל ממחלקה חדשה בשם Animal:

```
>>> class Animal(object):
...     def __init__(self):
...         self.age = 0
...         self.hunger = 10
...         self.fun = 0
...     def grow(self):
...         self.age += 1
...     def eat(self):
...         if self.hunger > 0:
...             self.hunger -= 1
...     def play(self):
...         self.fun += 1
...     def go_to_toilet(self):
...         self.hunger += 1
...     def sleep(self):
...         self.fun -= 1
```

המחלקה מייצגת חיה, כאשר החיה יכולה לגדול, לאכול, לשחק, ללכת לשירותים ולישון. שימו לב שמלבד העובדה שהחיה שלנו לא עושה הרבה חוץ מלעדכן כמה מונים, לא היינו רוצים להחליף אף מתודה בה.

כעת נגדיר מחלקה נוספת בשם Dog שיורשת מ-Animal:

```
>>> class Dog(Animal):
...     def __init__(self):
...         Animal.__init__(self)
...     def bark(self):
...         print 'Waff Waff'
...     def wag_tail(self):
...         self.fun += 2
...         print 'Wagging'
```

בנוסף לכל המתודות ש-Animal מגדירה, Dog מכילה 2 מתודות חדשות – מתודה לנביחה ומתודה לכשכוש בזנב.

יש לשים לב שבמתודת ה-__init__ של Dog יש קריאה למתודת ה-__init__ של Animal. דבר זה נעשה כדי לאפשר ל-Animal לאתחל את כל ה-attributes שרלוונטיים אליה. קריאה זו לא נעשית אוטומטית מאחר שכשאנחנו דורסים מתודה אנחנו מחליפים אותה ולכן אם נרצה לקרוא למתודה של ה-superclass ניאלץ לעשות את זה ידנית.

שימוש במחלקה החדשה Dog הוא בדיוק כמו שימוש במחלקה רגילה:

```
>>> dog = Dog()
>>> dog.play()
>>> dog.bark()
Waff Waff
>>> dog.wag_tail()
Wagging
>>> dog.eat()
>>> dog.sleep()
>>> dog.fun, dog.hunger, dog.age
(2, 9, 0)
```

MRO

כידוע, יש בעולם שלנו כלבים שמזדקנים הרבה יותר מהר מכלבים רגילים:

```
>>> class AgingDog(Dog):
...     def grow(self):
...         self.age += 10
...
>>> aging_dog = AgingDog()
>>> aging_dog.grow()
>>> aging_dog.age
10
```

כשאנחנו פונים ל-attribute של אובייקט מסוים, למשל מתודה של aging_dog, Python צריכה לחפש את ה-attribute שרצינו לקבל. למשל, אם נקרא למתודה ()grow, נוכל למצוא אותה מיד במחלקה AgingDog. אבל מה יקרה אם נקרא ל-()bark? הרי ל-AgingDog אין מתודה בשם ()bark. אנחנו כבר יודעים שהכל יפעל כמצופה והמתודה ()bark תיקרא מהמחלקה Dog:

```
>>> aging_dog.bark()
Waff Waff
```

Python יודעת למצוא את ה-attributes שלנו לאורך עץ הירושה בעזרת שיטה מאוד פשוטה: Python שומרת בכל אובייקט attribute מיוחד בשם __class__. ה-__class__ של אובייקט הוא ה-type object שלו:

```
>>> aging_dog.__class__
<class '__main__.AgingDog'>
>>> type(aging_dog)
<class '__main__.AgingDog'>
```

כעת, לכל type object יש גם attribute מיוחד בשם __bases__:

```
>>> AgingDog.__bases__
(<class '__main__.Dog'>,)
```

__bases__ הוא בסה"כ tuple שמכיל את כל המחלקות שירשנו מהן. במקרה שלנו זוהי מחלקה אחת, ובהמשך נראה מה יקרה כשנירש מכמה מחלקות.

אז מכאן זה כבר די פשוט – בכל פעם שנחפש attribute מסוים נסתכל קודם כל על האובייקט עצמו (self). אם ה-attribute לא שם, נלך ל-__class__ שלו ונחפש שם. אם לא מצאנו גם ב-__class__ נעבור על כל אחד מה-__bases__ של ה-type object ונחפש בו באותו אופן (על ה-__bases__ שלו ועל ה-__bases__ של ה-__bases__, עד שלא יהיו יותר __bases__).

ולמי אין יותר __bases__? ניחשתם נכון – ל-object:

```
>>> object.__bases__
()
```

הטכניקה הזאת נקראת MRO (Method Resolution Order), והיא דואגת לשני דברים. קודם כל, היא מוודאת שנמצא את מה שחיפשנו. בנוסף, היא מוודאת שנמצא את המתודה הנכונה. אם בכל אחד מה-type objects בעץ הירושה שלנו (AgingDog → Dog → Animal → object) הייתה המתודה age(), היינו צריכים לדעת שכשאנחנו קוראים ל-age() מ-AgingDog לא נקבל בטעות את Animal.age(), ומאחר שהחיפוש מתבצע לפי סדר הירושה אנחנו יודעים שתמיד נקבל את המתודה הנכונה.

isinstance, issubclass

שתי פונקציות שימושיות שקיימות ב-Python נקראות isinstance ו-issubclass. שתיהן יודעות לטייל על המסלול של ה-__bases__ שנוצרים בירושות שלנו, וכך אנחנו יכולים לקבל בזמן ריצת התוכנית מידע על האובייקטים שלנו.

isinstance היא פונקציה שמקבלת אובייקט ו-type object ומחזירה True אם הטיפוס של האובייקט יורש מה- type object:

```
>>> isinstance(aging_dog, AgingDog)
True
>>> isinstance(aging_dog, Dog)
True
>>> isinstance(aging_dog, Animal)
True
>>> isinstance(aging_dog, int)
False
```

ידענו ש-aging_dog הוא instance של AgingDog, אבל הוא גם מוגדר כ-instance של Dog. וזה הרי די הגיוני, כי אם AgingDog יורש מ-Dog אז כל מי שיקבל אובייקט מסוג AgingDog יוכל לעשות איתו כל דבר ש-Dog יודע לעשות (ולא אמור להיות לו אכפת שה-Dog שהוא קיבל הוא Dog קצת יותר משוכלל).

באופן דומה, issubclass היא פונקציה שמקבלת שני type-objects ומחזירה האם האחד הוא subclass של השני:

```
>>> issubclass(Dog, Animal)
True
>>> issubclass(Dog, AgingDog)
False
>>> issubclass(AgingDog, Dog)
True
```

שימו לב שהסדר חשוב בשתי הפונקציות – האובייקט הראשון הוא תמיד האובייקט שעליו אנחנו שואלים האם הוא instance או subclass, והאובייקט השני הוא הסוג שאליו אנחנו משווים. אם נתבלבל נקבל תשובה לא נכונה או שגיאה.

אז מה אפשר לעשות עם שתי הפונקציות האלה? נראה דוגמה: במחלקה Animal מימשנו מתודת eat, אבל ה-eat של החיות שלנו מאוד מוזר כי הוא לא מקבל אוכל. חיות בדרך כלל אוכלות אוכל, ומאחר שאוכל הוא אובייקט הוא צריך מחלקה משלו:

```
>>> class Food(object):
...     def __init__(self, mana):
...         self.mana = mana
```

יופי. עכשיו נתקן את Animal:

```
>>> class Animal(object):
...     def __init__(self):
...         self.age = 0
...         self.hunger = 10
...         self.fun = 0
...     def grow(self):
...         self.age += 1
...     def eat(self, food):
...         if not isinstance(food, Food):
...             return
...         self.hunger -= food.mana
...         if self.hunger < 0:
...             self.hunger = 0
...     def play(self):
...         self.fun += 1
...     def go_to_toilet(self):
...         self.hunger += 1
...     def sleep(self):
...         self.fun -= 1
```

עכשיו החיה שלנו הרבה יותר הגיונית. כשהיא מקבלת לאכול משהו שאיננו אוכל היא לא עושה איתו כלום ופשוט חוזרת. רק אם ניתן לה אוכל, היא תאכל אותו עד שהיא תשבע. אם האוכל לא יספיק לה בשביל לשבוע היא תישאר רעבה במידה מסוימת. נראה דוגמה ישירות על Animal:

```
>>> animal = Animal()
>>> animal.hunger
10
>>> animal.eat(Food(7))
>>> animal.hunger
3
>>> animal.eat(Food(7))
>>> animal.hunger
0
```

super

עכשיו יש לנו חיות שיודעות לאכול רק אוכל ולא דברים אחרים. אבל נניח שגילינו שהכלבים אוכלים לתרנגולות את כל האוכל (הרי כלבים הם חזקים ותרנגולות הן חלשות ומסכנות). עלינו לשנות את המימוש של הכלבים כך שיאכלו רק אוכל של כלבים.

מה בעצם נרצה לעשות? נרצה שהמחלקה Dog תירש מ-Animal, תדרוס את eat ותוודא שהאוכל הוא אוכל של כלבים. ומה אז? נרצה לקרוא ל-eat של Animal? כן ולא. בפועל אנחנו רוצים לקרוא ל-Animal.eat, אבל אנחנו לא רוצים לקרוא מפורשות ל-Animal.eat אלא למתודה "ה-eat של מי שירשנו ממנו". כלומר, לא מעניין אותנו ש-Animal הוא ה-superclass של Dog. מעניין אותנו לקרוא ל-eat הנכון.

בשביל זה המציאו ב-Python את super:

```
>>> class DogFood(Food):
...     pass
...
>>> class Dog(Animal):
...     def __init__(self):
...         super(Dog, self).__init__()
...     def bark(self):
...         print 'Waff Waff'
...     def wag_tail(self):
...         self.fun += 2
...         print 'Wagging'
...     def eat(self, food):
...         if isinstance(food, DogFood):
...             super(Dog, self).eat(food)
```

במקום להשתמש ישירות ב-Animal, אנחנו משתמשים ב-super(Dog, self) (שימו לב שהחלפנו גם את המימוש של __init__). super יודע להחזיר אובייקט שמשמש כמתווך שיודע להחזיר לנו את המתודה הנכונה עבור ה-superclass שלנו. ה-superclass נקבע בצורה דינמית לפי ה-__bases__ של ה-type object שהעברנו ל-super, במקרה הזה זהו Dog.

super חוסך לנו הרבה חשיבה, כי כל מה שעלינו לעשות הוא לציין מיהו ה-class הנוכחי ו-Python תעשה בשבילנו את כל העבודה. אבל למה בעצם צריך לציין את ה-class? הרי אנחנו יודעים באיזה מחלקה אנחנו. למה Python לא יכולה לקרוא לבד ל-type(self) ולחסוך לנו את הטרחה? נסתכל על הדוגמה הבאה:

```
>>> class AgingDogFood(DogFood):
...     pass
...
>>> class AgingDog(Dog):
...     def eat(self, food):
...         if isinstance(food, AgingDogFood):
...             super(AgingDog, self).eat(food)
...
```

כאשר נקרא ל-AgingDog.eat עם האוכל המתאים, המתודה נקרא ל-super. במקרה הספציפי הזה type(self) הוא אכן AgingDog. אבל כשנגיע ל-Dog.eat מה יהיה type(self)? הוא גם יהיה AgingDog, הרי ה-type של אובייקט לא משתנה. אם לא נגיד ל-Python איפה אנחנו בשרשרת הקריאות של הפונקציות, היא לא תוכל לדעת מאיזה type object לקחת את __bases__ כדי שתוכל להמשיך ב-MRO כמו שצריך.

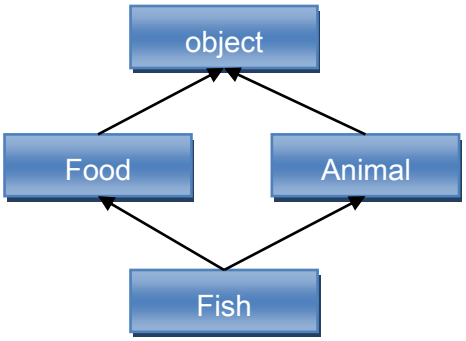
מעכשיו, תמיד נשתמש ב-super ולעולם לא נציין מפורשות את ה-superclass. כמו כן, תמיד נקרא ל-super של __init__. בסעיף הבא נראה סיבה נוספת לשימוש ב-super והיא שאנחנו לא תמיד יודעים מיהי ה-superclass שלנו.

ירושה מרובה

חיות שאוכלות אוכל... כמה תמים. שכחנו דבר אחד קטן: יש חיות שהן גם אוכל.

בסעיף הזה ניאלץ לפתוח מחדש את כל המחלקות שכתבנו, ונראה כמה חמור יכול להיות מצב בו לא קראנו ל-super מ-__init__. כמו כן, בסעיף הזה נראה ש**ירושה מרובה היא דבר רע**. היא מייצרת קוד מסובך ולכן רצוי להמנע ממנה.

אז נתחיל מהסוף – מה המטרה שלנו? להגיע למחלקה שיורשת מ-Food ומ-Animal:



נגדיר את המחלקה הזאת:

```
>>> class Fish(Food, Animal):
...     def __init__(self, mana):
...         super(Fish, self).__init__(mana)
```

כעת עלינו לחשוב למי super(Fish, self) הולך – ל-Food או ל-Animal. לפי ה-MRO, Python לוקחת את __bases__ ועוברת עליהם אחד אחרי השני. נבדוק מהו Fish.__bases__:

```
>>> Fish.__bases__
(<class '__main__.Food'>, <class '__main__.Animal'>)
```

כלומר, super(Fish, self) יקרא ל-Food. אבל מישהו צריך לקרוא ל-Animal.__init__, וכאן זה מתחיל להסתבך. המשמעות של super(Fish, self) היא "תקרא להבא בתור בשרשרת הירושה". אז אם השרשרת שלנו היא ש-Fish יורש מ-Food ומ-Animal, כאשר __init__ של Fish קורא ל-__init__ של Food, ההגדרה אומרת שה"הבא בתור" של Food.__init__ הוא Animal.__init__. כלומר, Animal.__init__ ייקרא ע"י Food.__init__.

נחזור על זה: Animal.__init__ ייקרא ע"י Food.__init__. כן. למרות ש-Animal יורש מ-object ובחיים לא שמע על Food (לפחות לא בשרשרת הירושה), הולכים לקרוא ל-__init__ שלו מתוך Food.__init__.

לכן, הדבר הראשון שנעשה הוא לתקן את Food, כי הרי לא קראנו ל-super של __init__ מ-Food:

```
>>> class Food(object):
...     def __init__(self, mana):
...         super(Food, self).__init__()
...         self.mana = mana
```

הדבר השני שנעשה הוא לתקן את Animal, כי אנחנו צריכים שגם super של Animal ייקרא:

```
>>> class Animal(object):
...     def __init__(self):
...         super(Animal, self).__init__()
...         self.age = 0
...         self.hunger = 10
...         self.fun = 0
...     def grow(self):
...         self.age += 1
...     def eat(self, food):
...         if not isinstance(food, Food):
...             return
...         self.hunger -= food.mana
...         if self.hunger < 0:
...             self.hunger = 0
...     def play(self):
...         self.fun += 1
...     def go_to_toilet(self):
...         self.hunger += 1
...     def sleep(self):
...         self.fun -= 1
```

ועכשיו, בשעה טובה, נוכל ליצור instance של דג, ונוכל גם לראות שיש לו את כל ה-attributes של Food וגם של Animal:

```
>>> fish = Fish(3)
>>> fish.mana
3
>>> fish.age
0
>>> fish.hunger
10
>>> fish.fun
0
```

ובניגוד לשאר הדגים בטבע, קיבלנו דג שיכול לאכול את עצמו:

```
>>> fish.eat(fish)
>>> fish.hunger
7
```

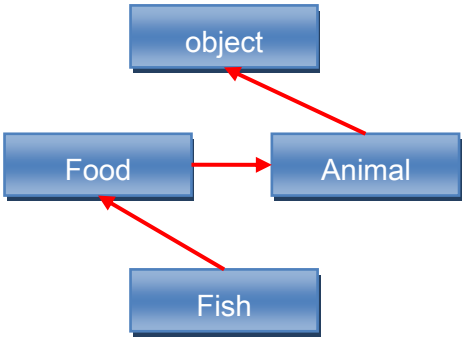
נסביר שוב מה קרה כאן במילים אחרות, מאחר שקצת קשה להבין את ההתנהגות הזאת בפעם הראשונה:

ירושה רגילה שבה מחלקה יורשת ממחלקה אחת בלבד היא מקרה פשוט. יש __init__ אחד, קוראים לו וממשיכים הלאה להשתמש באובייקט כמו שצריך. לרוב __init__ יקבל פרמטרים, ולכן נעביר לו את הפרמטרים שהוא יצפה להם.

בירושה מרובה ב-Python אנחנו מייצרים לעצמנו בעיה בכך שאנחנו יורשים ממחלקה ודורסים את ה-__init__ שלה, כי דרסנו יותר מ-__init__ אחד. פתרון אחד לבעיה (שהוא הפתרון הקל, אבל לא בהכרח הכי נכון) הוא לדרוש מהמתכנת לקרוא לכ"א מפונקציות ה-__init__ של ה-superclass-ים ישירות. ככה לא נשכח אף __init__ ולא נוכל לטעות בפרמטרים שנעביר ל-__init__ אחד או אחר. אבל זה פתרון שטוב רק ל-__init__, כי מה יקרה אם נרצה את אותו הפתרון גם על מתודה רגילה (למשל __getattr__)? נצטרך תמיד לקרוא מפורשות למחלקת ה-super שלנו, וזו לא תמיד אותה מחלקה. הבעיה מחמירה אם נירש בצורה מרובה מכמה מחלקות שלכולן מחלקת super משותפת, כאשר רק חלק מהמחלקות דורסות מתודה מסוימת*.

* הנכם מוזמנים לקרוא על כך ב-http://en.wikipedia.org/wiki/Multiple_inheritance#The_diamond_problem

לכן ב-Python החליטו ללכת על הפתרון הכללי יותר. בפועל נצטרך לכתוב פחות קוד, אבל ניאלץ להבין טוב טוב מה אנחנו עושים. הפתרון של Python הוא לקחת את הירושה המרובה ולנסות להפוך אותה לירושה רגילה, ע"י-כך שבמקום לדמיין עץ ירושה מסועף, נסדר את כל המחלקות שירשנו מהן (בין אם אלה היו ירושות מרובות או בודדות) ולפי הסדר החדש הזה נקרא לפונקציות ב-superclass. במקרה של המחלקה Fish קיבלנו את סדר הקריאות הבא:



היתרון של השיטה הזו הוא שכאשר אנחנו קוראים למתודה ב-superclass, נחפש את המתודה קודם כל במחלקות שירשנו מהן, אחר כך במחלקות שהן ירשו מהן, כלומר נטייל על הרמות בעץ ולא נבצע סתם רקורסיה כלפי מעלה. אפשר גם לדמיין שאם המחלקה שלנו היא ילד אז Python מחפשת מתודה אצל ההורים, אחר כך אצל הסבים והסבתות, אחר כך אצל סבא וסבתא רבא, וכך הלאה.

מבחינתנו, נעדיף להמנע מירושה מרובה, ותמיד נזכור להשתמש ב-()super כשנקרא למתודות של ה-superclass שלנו ולא נשתמש ישירות בשם המחלקה שירשנו ממנה.

__getattr__, __setattr__, __delattr__

לפני כמה סעיפים פגשנו את __get/set/delitem__ שמאפשרים לנו ליצור אובייקטים שמתנהגים כמו מילון, רשימה, או כל דבר אחר שנרצה שיתמוך ב-syntax של סוגריים מרובעים אחרי האובייקט. אבל יש דבר הרבה יותר בסיסי מסוגריים מרובעים והוא attributes. מה אם נרצה ליצור אובייקט שיש לו את אותה גמישות שקיבלנו ב-__get/set/delitem__, רק במקום items עם attributes?

Python מאפשרת לנו לעשות את זה בעזרת __getattr__, __setattr__ ו-__delattr__. נתחיל בדוגמה:

```
>>> class Attrable(object):
...     def __init__(self):
...         super(Attrable, self).__init__()
...         self._items = {}
...     def __getattr__(self, attr):
...         if attr.startswith('_'):
...             return super(Attrable, self).__getattr__(attr)
...         return self._items[attr]
...     def __setattr__(self, attr, value):
...         if attr.startswith('_'):
...             return super(Attrable, self).__setattr__(attr, value)
...         self._items[attr] = value
...
>>> a = Attrable()
>>> a.moshe = 1
>>> a.haim = 2
>>> dir(a)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__', '__getattr__',
'__getattribute__', '__hash__', '__init__', '__module__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__', '_items']
>>> a._items
{'moshe': 1, 'haim': 2}
```

מתודת ה-__init__ אמורה להיות די ברורה. המתודות __setattr__ ו-__getattr__ עובדות בדומה ל-__getitem__ ו-__setitem__, מלבד הבדל אחד קטן. גם Python משתמשת בהן כדי לגשת ל-attributes. הרי זה כל הקטע של attributes. לכן, אנחנו בוררים את ה-attributes שנכניס ל-_items (אלה ה-attributes שמתחילים בקו תחתון). כל מי שלא מתחיל בקו תחתון מופנה ל-_items. כל מי שמתחיל בקו תחתון מופנה ל-super, כי אנחנו רוצים לתת למימוש המקורי לטפל בו.

וכמו שאפשר לראות, הדוגמה הזאת עובדת בדיוק כמו שרצינו.

__dict__

כדי לממש את מנגנון ה-attributes, Python צריכה להחזיק מיפוי בין שם של attribute לבין האובייקט שהוא מייצג. למזלנו כבר יש לנו סוג כזה של אובייקט – מילון. מילון הרי יודע לשמור מיפוי בין keys ל-values.

במקרה הספציפי של attributes אנחנו רוצים להחזיק מיפוי בין keys מסוג str ל-values כלשהם. וזה בדיוק מה ש-Python עושה – לכל אובייקט עם attributes מוחזק attribute מיוחד בשם __dict__ ששומר את המיפוי בין שמות ה-attributes ל-attributes עצמם. אם נסתכל על dir() שהרצנו מוקדם יותר נוכל לראות בין כל ה-attributes גם את __dict__:

```
>>> x = Food(2)
>>> dir(x)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__', '__getattribute__',
'__hash__', '__init__', '__module__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__',
'mana']
```

ואם נסתכל בתוך __dict__ נוכל למצוא שם את:

```
>>> x.__dict__
{'mana': 2}
```

כלומר, Python שומרת ב-__dict__ רק את ה-attributes שהם ספציפיים לאובייקט הספציפי שלנו. הרי את כל שאר הדברים היא יכולה להשלים מה-type object, וזה בדיוק מה שהיא עושה. כשאנחנו פונים לאיזשהו attribute, Python מחפשת אותו ב-__dict__, ואם הוא לא נמצא היא פונה ל-MRO.

כך בדיוק __getattr__ ו-__setattr__ עובדים בדוגמה מהסעיף הקודם. כל פניה ל-attribute שלא שמרנו לעצמנו (אלה היו הפניות ל-attributes שהכילו קו תחתון בתחילת השם שלהם) הופנתה למימוש הדיפולטי של Python.

בנוסף, קיימות שלוש פונקציות מובנות שנקראות getattr, setattr ו-delattr. פונקציות אלה מבצעות את הפעולה של קבלת, שינוי ומחיקת attribute בצורה דינמית. נראה דוגמה:

```
>>> getattr(x, 'mana')
2
>>> attr = 'ma' + 'na'
>>> getattr(x, attr)
2
>>> setattr(x, 'mana', 3)
>>> x.mana
3
```

שלושת הפונקציות האלה מאפשרות לנו לגשת ל-attributes בצורה דינמית, כלומר מבלי שנדע בזמן כתיבת הקוד את שם ה-attribute. למשל, נוכל לעבור על כל ה-attributes באובייקט מסוים ולבדוק האם כולם מספרים שלמים:

```
>>> def all_ints(obj):
...     for attr in dir(obj):
...         if not isinstance(getattr(obj, attr), int):
...             return False
...     return True
...
>>> all_ints(x)
False
```

וכאן תזכורת – dir() היא פונקציה שמחזירה רשימה של מחרוזות, כשכל מחרוזת מכילה שם של attribute באובייקט.

נסביר אגב את מה שכבר כנראה הבנו בצורה כזאת או אחרת – dir(), הפונקציה הפשוטה לכאורה, בעצם עושה די הרבה. היא עוברת על כל ה-keys של __dict__, על כל ה-attributes של ה-type object, על כל אחד מה-__bases__ של ה-type object ועל ה-attributes שלו בצורה רקורסיבית, ולבסוף מחזירה רשימה אחת יפה וממויינת עם כל ה-attributes שנוכל למצוא באובייקט שהעברנו ל-dir().

לסיום, נציין שיש אובייקטים שאין להם __dict__. דוגמה אחת היא int. בהקשר הזה אנחנו יכולים להגיד של-int-ים אין attributes. זה נכון בהקשר בו ל-int אין __dict__, כל עוד ברור לנו שהכוונה ל-attributes דינמיים ולא כאלה שמגיעים מה-type object. נדגים למה הכוונה:

```
>>> x = 5
>>> '__dict__' in dir(x)
False
```

כלומר, למספרים שלמים אין __dict__. התוצאה היא שאיננו יכולים לאחסן בהם attributes אחרת נקבל שגיאה:

```
>>> x.moshe = 1
Traceback (most recent call last):
  File "<pyshell#236>", line 1, in <module>
    x.moshe = 1
AttributeError: 'int' object has no attribute 'moshe'
```

object

עכשיו כשהבנו איך אובייקטים אמורים להתנהג ואנחנו מבינים מהם type-objects ומה העבודה שלהם, נוכל סוף סוף להבין מיהו אותו object מסתורי שנאלצנו לרשת ממנו בתחילת הפרק כשאפילו לא ידענו מהי ירושה.

ב-Python יש שני סוגי מחלקות – Old-style ו-New-style. כשאנחנו יורשים מ-object אנחנו מייצרים אובייקט מסוג new-style-class. כמו שאפשר לנחש, אם לא נירש מ-object נקבל אובייקט מסוג old-style-class.

כמו שהשם מרמז, old-style זה לא דבר חיובי אחרת השם שלו היה יותר מושך. ניצור מחלקת old-style וננסה להבין למה:

```
>>> class Older:
...     def __init__(self):
...         self.attr = 'value'
...
>>> older = Older()
>>> older.attr
'value'
```

עד כאן הכל בסדר. יש אובייקט ויש לו attributes. נסתכל על dir(older):

```
>>> dir(older)
['__doc__', '__init__', '__module__', 'attr']
```

משהו כאן לא בסדר... עד עכשיו היינו מקבלים הרבה יותר attributes. הסיבה העיקרית לכך (שלא נתעמק בה) היא שב-new-style-objects המתודות של האופרטורים (לדוגמה __add__, __getitem__ ואפילו __init__) ממומשות במחלקת האב (object) ונקראות משם. בעולם ה-old-style לאובייקט יש רק את המתודות שהוא הגדיר לעצמו וניתן אפילו לדרוס אותן בזמן ריצה.

נקרא כעת ל-type() עם האובייקט שלנו:

```
>>> type(older)
<type 'instance'>
```

עכשיו ברור שמעבר להתנהגות המשונה של dir(), אין סימטריה בין ה-type object ל-type(), כלומר ב--Old-style classes ערך ההחזרה של type() הוא בכלל לא ה-type-object. ההבדל הזה אמנם לא יפה לעין, אבל הוא בעייתי מהסיבה שבעולם ה-old-style המחלקות שלנו אינן type objects ולכן ה-instance-ים של המחלקות שלנו הם סתם מקום אחסון ל-attributes.

הסימטריה בין type object ו-type() מאפשרת לנו לקבל פיצ'רים מתקדמים בשפה (כמו __getattr__) וכן היא מאפשרת לנו לרשת מהטיפוסים המובנים של Python (אפשר לדוגמה לרשת מ-dict וליצור לעצמנו סוג חדש של מילון).

ב-Python 3.0 לא תהיה תמיכה ב-old-style-classes ולכן גם object ייעלם ויהיו רק new-style-classes. כדי להיות מוכנים לעתיד הקרב ובא, ניצור רק מחלקות new-style כדי שהקוד שלנו ירוץ בסביבת Python 3.0 גם אם לא נשתמש בפיצ'רים של new-style-classes.

Slot-ים נוספים

Python מאפשרת לנו לדרוס הרבה slot-ים נוספים כדי שנוכל לגרום לאובייקטים שלנו להשתלב יפה בסביבה פייתונית. בסעיף זה נציין את ה-slot-ים ומה תפקידם, אך לא נפרט עליהם כי עוד לא למדנו על Exception-ים (אותם נכסה בפרק הבא). בעזרת החומר שלמדנו עד כה ואחרי שנלמד על Exception-ים תוכלו בקלות להבין כיצד להשתמש ב-slot-ים שנציין כאן.

- __cmp__, __ge__, __gt__, __ne__, __nq__, __lt__, __le__ הם slot-ים להשוואה בין שני אובייקטים. שניהם מקבלים תמיד שני פרמטרים (self, other) וצריכים להחזיר True או False, מלבד __cmp__ שמחזיר 0, 1 או 1- ועל כולם תוכלו לקרוא בתיעוד של Python.
- __call__ הוא slot שנקרא כאשר אנחנו "קוראים" למחלקה שלנו כאילו היא פונקציה.
- __mul__, __sub__, __add__ ועוד רבים אחרים משמשים כדי לתמוך בפעולות חשבוניות. כך נוכל לדוגמה לחבר בין שני instance-ים של המחלקה שלנו.
- __contains__ מאפשר לנו לממש את האופרטור in עבור המחלקה שלנו.
- __len__ יחזיר את אורך המחלקה.
- __float__, __long__, __int__ עושים בדיוק מה ש-__str__ עושה עבור מחרוזות, ומאפשרים לספר מתודה שאומרת איך להמיר את האובייקט שלנו למספר, מספר ארוך או float.
- __enter__, __exit__ מאפשרים לנו לממש Context managers, עליהם ניתן לקרוא בתיעוד של Python.
- __iter__ מאפשרת לנו לקבל iterator עבור האובייקט שלנו. על איטרטורים נלמד בהמשך.
- __hash__ משמש לחישוב hash של אובייקט. Hash הוא מספר "קסם" שמשמש מילונים כדי לדעת איפה לאחסן ואיך למצוא אובייקטים בתור keys במילון.

חלק 6: Exceptions

בוודאי שמתם לב שעד עכשיו אמרנו הרבה פעמים "שגיאה" על כל מיני מקרים שלא התעמקנו בהם, אבל לא הגדרנו בצורה מדויקת מהי שגיאה. יותר מכך, לא אמרנו מה אפשר לעשות עם השגיאות האלה. בפרק הזה נכיר מנגנון שימושי מאוד ב-Python שנקרא Exceptions. למזלנו היינו סבלניים ולמדנו היטב על אובייקטים ומחלקות, כך שאנחנו כבר מכירים טריק או שניים של Python ולכן Exception-ים כבר לא ייראו כמו משהו חדש במיוחד.

Exception-ים מנסים לעזור לנו בבעיה ששפות אחרות לא תמיד מטפלות בה, והיא נושא הטיפול בשגיאות. שגיאה היא כל מקרה שבו התוכנית שלנו לא יכולה לעבוד כמו שביקשו ממנה. דוגמה פשוטה היא אם ננסה להמיר את המחרוזת 'a' ל-int. אי-אפשר להמיר את 'a' ל-int, והפונקציה int צריכה להגיד לנו את זה איכשהו.

דרך אחת בה int יכלה להגיד לנו שהיא לא מוכנה לקבל את הקלט 'a' היא ש-int תמיד הייתה מחזירה tuple. באיבר הראשון היא הייתה מחזירה True או False כדי להגיד האם היא הצליחה או נכשלה, ובאיבר השני היא הייתה מחזירה את התוצאה אם היא הצליחה או None אם לא. אבל אז **תמיד** היינו צריכים לבדוק, בכל קריאה ל-int, האם היא הצליחה או לא. כל פעם. לא משנה ש-int כמעט אף פעם לא נכשלת, תמיד נצטרך לבדוק מה היא החזירה.

יותר גרוע מזה, לפעמים אנחנו עשויים לשכוח לבדוק את ערך ההחזרה ואז נהיה חשופים לבאגים מוזרים שבהם נשתמש בערכים לא נכונים בכל מיני מקומות בלי שנדע מאיפה הם הגיעו באמת (הרי יש עוד פונקציות חוץ מ-int...).

מה שאנחנו רוצים הוא פתרון שפשוט יודיע לנו כשמשהו לא היה בסדר, וככה גם לא נצטרך לבדוק האם כל שורת קוד הצליחה וגם לא נוכל לשכוח שום שגיאה.

אובייקט Exception

נמשיך בדוגמה שממנה התחלנו – ננסה להמיר את 'a' ל-int:

```
>>> int('a')
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    int('a')
ValueError: invalid literal for int() with base 10: 'a'
```

Python הדפיסה דבר שנקרא Traceback, והוא קורה כשמתרחשת שגיאה ואף אחד לא מטפל בה. השורה הראשונה היא כותרת קבועה שתמיד אומרת שזה Traceback. השורות האמצעיות אומרות לנו מאיפה בקוד קרתה השגיאה, כלומר אילו פונקציות נקראו עד שהגענו למצב שאליו הגענו. השורה האחרונה והכי מעניינת היא השורה שאומרת לנו מה קרה, ובמקרה שלנו היא מתחילת ב-ValueError.

נקליד ValueError ב-interpreter ונגלה ש:

```
>>> ValueError
<type 'exceptions.ValueError'>
```

ValueError הוא type object. ספציפית, נוכל גם לגלות שהוא יורש ממחלקה קדומה יותר שנקראת Exception:

```
>>> issubclass(ValueError, Exception)
True
```

וגם נוכל לגלות ש-Exception (ולכן גם ValueError) יורש ממחלקה נוספת שנקראת BaseException:

```
>>> issubclass(Exception, BaseException)
True
```

השגיאה שלנו היא בסה"כ instance של ValueError, ו-ValueError יורש מ-BaseException.

בגדול, exception-ים הם בסך הכל אובייקטים. אובייקט exception חייב לרשת מ-superclass שנקרא BaseException. יכול להיות עץ ירושה שלם עד שנגיע לאובייקט שלנו, אבל Python דורשת רק דבר אחד והוא שאי-שם בראש עץ הירושה של ה-exception יהיה BaseException.

אם נרצה נוכל ליצור אובייקטי exception סתם ככה:

```
>>> ValueError('Our first error!')
ValueError('Our first error!',)
```

וכמו כל אובייקט רגיל נוכל לשמור אותם במשתנים, לקבל את ה-type שלהם וכמובן ליצור exception-ים משלנו.

כדי ליצור exception-ים משלנו, כל מה שעלינו לעשות הוא להגדיר מחלקה שיורשת מ-Exception. מאחר ש-Exception יורשת מ-BaseException אנחנו מקיימים את הדרישה של ירושה מ-BaseException ולכן אין בעיה לרשת מ-Exception:

```
>>> class SomeError(Exception):
...     pass
```

וכמובן נוכל ליצור instance-ים של ה-exception החדש שלנו:

```
>>> error = SomeError('OK, now what?')
>>> error
SomeError('OK, now what?',)
```

זריקת Exception-ים: raise

עכשיו כשראינו איך ליצור אובייקט עבור ה-expcetion שלנו, ננסה לחקות את ההתנהגות של int. אנחנו נעשה משהו קצת יותר מעניין והוא לכתוב פונקציה שמוכנה לקבל רק מספרים שמתחלקים ב-3. אם המשתמש נותן לפונקציה מספר שמתחלק ב-3, היא מחזירה את המספר מחולק ב-3. אם לא הוא מתחלק ב-3, ניצור שגיאה בדיוק כמו ש-int עשתה.

יצירת השגיאה תיעשה בעזרת הפקודה raise שמקבלת אובייקט exception, כמו שניתן לראות בדוגמה:

```
>>> def mysterious_func(num):
...     '''Takes a number 'num' and returns some other number.
...     at some cases this function raises ValueError.
...     '''
...     if num % 3 != 0:
...         raise ValueError('{} must divide by 3'.format(num))
...     return num / 3
```

שימו לב שבמקרה הזה לא הגדרנו exception משלנו, אלא השתמשנו ב-ValueError שקיים ב-Python. מקובל להשתמש ב-exception-ים המובנים של Python כי גם מישהו שלא מכיר את הקוד שלנו יוכל להבין מה השגיאות הנפוצות אומרות.

כעת ננסה את הפונקציה שלנו במקרים שבהם היא אמורה לפעול:

```
>>> mysterious_func(3)
1
>>> mysterious_func(0)
0
>>> mysterious_func(-3)
-1
```

וננסה להפעיל אותה עם קלט שהיא לא מעוניינת בו:

```
>>> mysterious_func(2)

Traceback (most recent call last):
  File "<pyshell#45>", line 1, in <module>
    mysterious_func(2)
  File "<pyshell#40>", line 3, in mysterious_func
    raise ValueError('{} must divide by 3'.format(num))
ValueError: 2 must divide by 3
```

הפונקציה mysterious_func שלנו עשתה דבר שנקרא "לזרוק exception". הפקודה raise אומרת ל-Python לקחת את אובייקט ה-exception שהעברנו לה ולזרוק אותו.

בדוגמה שלנו, זריקת exception היא בסה"כ הדפסה מאוד לא מוצלחת של השגיאה שרצינו להעביר למשתמש, ולכן Python כוללת בנוסף למנגנון הזריקה גם מנגנון לתפיסת exception-ים.

תפיסת Exception-ים: try…except

כרגע יש לנו קוד שבמקרים מסוימים זורק exception. ברור שהדפסה של השגיאה למסך היא לא הדבר הכי טוב שנרצה לעשות עם השגיאה, לפחות לא ברוב המקרים. בדרך-כלל כשניתקל באיזושהי שגיאה נרצה לטפל בה.

אם נשאר עם הדוגמה מהסעיף הקודם, נניח שיש לנו את mysterious_func ונניח שלא ראינו את הקוד של הפונקציה אלא רק קראנו את ה-doc-string שלה, נרצה לתפוס את ה-exception שנזרק ולטפל בו. את זה נעשה בעזרת ההוראות try ו-except:

```
>>> help(mysterious_func)
Help on function mysterious_func in module __main__:

mysterious_func(num)
    Takes a number 'num' and returns some other number
    at some cases this function raises ValueError.
```

אז אנחנו יודעים ש-mysterious_func צריכה לקבל מספר ושהיא עלולה לזרוק ValueError. בואו נכתוב קטע קוד שמקבל קלט מהמשתמש, ממיר אותו ל-int (כי קלט מ-raw_input מגיע כ-str) ואם הצלחנו להמיר אותו ל-int נקרא ל-mysterious_func. אם mysterious_func תזרוק ValueError, נוריד 1 מהמספר וננסה שוב עד שנצליח:

```
>>> def get_int_from_user():
...     while True:
...         try:
...             raw_num = raw_input('Enter a number: ')
...             return int(raw_num)
...         except ValueError:
...             print '{} is not a number'.format(raw_num)
...
>>> def less_mysterious_func():
...     num = get_int_from_user()
...     while True:
...         try:
...             return mysterious_func(num)
...         except ValueError:
...             num -= 1
...
>>> less_mysterious_func()
Enter a number: moshe
moshe is not a number
Enter a number: 4
1
```

כמו שאפשר לראות, את הבעיה שלנו חילקנו לשניים – פונקציה שקולטת מספר מהמשתמש, ופונקציה שקוראת ל-mysterious_func עד שהיא מצליחה לקבל ערך בחזרה, ובשתי הפונקציות אפשר לראות שהשתמשנו בשני statements חדשים: try ו-except.

כאשר אנחנו כותבים try בקוד שלנו, Python מצפה למצוא בלוק של קוד, בדיוק כמו פונקציה או קטע if. מה שמיוחד ב-try הוא שאם ייזרק exception מתוך הקוד בבלוק ה-try, הוא ייחסם כך שנוכל להגדיר בדיוק איך לטפל בו. זה לא משנה אם הקוד בבלוק ה-try יעשה raise ישירות, או שאולי ייזרק exception מתוך קריאה לפונקציה. אם exception כלשהו ייזרק מתוך הקוד ב-try, הוא ייחסם ברמה הזאת כדי שנוכל להגיד איך לטפל בו.

שיטת הטיפול הראשונה שנכיר היא except. כשאנחנו כותבים בלוק except בקוד שלנו, אנחנו צריכים לציין את ה-type של ה-exception שנרצה לטפל בו. אם נסתכל שוב על get_int_from_user, נוכל לראות שכתבנו except ValueError, כלומר בלוק ה-except יטפל רק ב-exception-ים שיורשים מ-ValueError או שהם ValueError בעצמם. Python משתמשת ב-isinstance כדי לוודא את סוג ה-exception.

נסתכל לדוגמה על קטע הקוד הבא שמתבצע עד שפקודת ה-raise מתרחשת. ברגע שמגיעים ל-raise, השליטה עוברת לבלוק ה-except המתאים:

```
>>> class Error1(Exception):
...     pass
...
>>> class Error2(Exception):
...     pass
...
>>> class Error3(Exception):
...     Pass
...
>>> try:
...     print 'Hello'
...     print 'World'
...     print 'My'
...     print 'Name'
...     raise Error2()
...     print 'Is'
...     print 'Luca'
... except Error1:
...     print 'Error #1 has occured'
... except Error2:
...     print 'Error #2 has occured'
... except Error3:
...     print 'Error #3 has occured'
...
Hello
World
My
Name
Error #2 has occured
```

מקטע ה-try מתחיל לרוץ...

נזרקת שגיאה והשליטה עוברת מיד לקטע הקוד המתאים.

ואכן, השגיאה שנזרקה הייתה מסוג Error2, וקטע ה-except המתאים התבצע.

ה-Exception-ים של Python

ל-Python יש הרבה exception-ים מובנים שמגיעים עם השפה, וברוב המקרים בכלל לא נצטרך להגדיר exception-ים משלנו. כעת נפגוש את ה-exception-ים השימושיים יותר, ובתיעוד של Python תוכלו למצוא את BaseException ואת כל ה-exception-ים שיורשים ממנה ומובנים בשפה.

OSError

OSError הוא exception שנזרק כשמשהו רע קורה בקריאה לפונקציה של מערכת ההפעלה. בפרק הבא נלמד על מודולים ונפגוש שם את המודול os אך לבינתיים נסתפק בכך שאם נכתוב "import os" נוכל להשתמש במודול os ולקרוא לפונקציות של מערכת ההפעלה ממנו.

נתחיל בכך שניצור קובץ (הקובץ יווצר בספריה הנוכחית, אבל זה לא ממש מעניין כי עוד רגע נמחק אותו):

```
>>> file('david.txt', 'w').write('Some file...')
```

מאחר שלא נזרק שום exception אנחנו יכולים לדעת שהקובץ נכתב בהצלחה. כעת נמחק את הקובץ:

```
>>> import os
>>> os.remove('david.txt')
```

ושוב, מאחר שלא נזרק שום exception אנחנו יודעים שהקובץ נמחק. כעת ננסה למחוק את הקובץ שוב, מה שאמור לגרום ל-exception להיזרק:

```
>>> os.remove('david.txt')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OSError: [Errno 2] No such file or directory: 'david.txt'
```

כאן כדאי להתחיל להבין מה קורה בתוך האובייקט OSError, כי OSError נשמע כמו שם מאוד כללי שיכול להעיד על איזושהי שגיאת מערכת-הפעלה, ושגיאת מערכת הפעלה לא בהכרח אומרת שהקובץ לא נמצא, היא גם יכולה להגיד שאין לנו גישה אליו או שהקובץ עדיין פתוח ע"י מישהו אחר.

מהסיבה הזו אובייקטי OSError מכילים attribute בשם errno שמכיל קוד שאומר מה הייתה השגיאה. במקרה שלנו ה"קוד" הזה הוא 2, אך ממש לא צריך לזכור את הקודים האלה מכיוון שיש מודול נוסף בשם errno שמכיל קבועים עבור השגיאות שמערכת ההפעלה יכולה להחזיר.

ועכשיו, כדי שנוכל להשתמש ב-errno של OSError אנחנו צריכים להיות מסוגלים לגשת לאובייקט של ה-exception, ואת זה נעשה כך:

```
>>> import errno
>>> def delete_file(path):
...     try:
...         os.remove(path)
...     except OSError, os_error:
...         if os_error.errno != errno.ENOENT:
...             raise
...
>>> delete_file('blah')
>>> delete_file('/etc/passwd')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in delete_file
OSError: [Errno 13] Permission denied: '/etc/passwd'
```

כאן נוכל לראות כמה דברים חדשים: קודם כל, ב-except אנחנו לא מציינים רק את ה-type של ה-exception אלא גם את שם המשתנה שיצביע לאובייקט ה-exception שקיבלנו. בנוסף, השתמשנו ב-raise בלי פרמטרים. raise בלי פרמטרים מותר לעשות רק בתוך בלוק except והוא גורם ל-exception הנוכחי להמשיך לפעפע לבלוקי ה-try שמעלינו, ומשתמשים בו במקרה שבו החלטנו שאנחנו לא יודעים איך לטפל ב-exception ולכן נמשיך לזרוק אותה למעלה.

במקרה שלנו בדקנו האם errno הוא ENOENT, שהוא קבוע שאומר שלא מצאנו את הקובץ שחיפשנו (קיצור של Error NO ENTry). אם ה-errno שקיבלנו הוא לא ENOENT אנחנו עושים raise ולא מטפלים ב-exception הזה, כי הרי העבודה של הפונקציה שלנו הייתה לוודא שמוחקים קובץ ומתעלמים ממנו אם הוא לא קיים, ואם לא הצלחנו למחוק את הקובץ בטוח לא עשינו את העבודה שלנו ולכן אנחנו חייבים להמשיך לזרוק את ה-exception הלאה.

IOError

IOError מאוד דומה ל-OSError, עם הבדל קל – את IOError נקבל עבור שגיאות שקשורות ספציפית לקלט ופלט, בד"כ בזמן עבודה עם קבצים אבל גם במקרים אחרים.

לדוגמה, נוכל לקבל IOError אם ננסה לפתוח קובץ שלא קיים:

```
>>> file('/etc/passwd2')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: '/etc/passwd2'
```

שימו לב שכביכול היינו אמורים לקבל OSError עם errno של ENOENT, כי הרי זאת בדיוק אותה שגיאה כמו בחלק הקודם – ניסינו לפתוח קובץ שלא קיים. אבל פה ניסינו לעבוד עם הקובץ דרך file ולא דרך קריאות לפונקציות במודול os, ולכן קיבלנו IOError. אם היינו מנסים לעשות את אותה הפעולה דרך os היינו מקבלים OSError כצפוי:

```
>>> os.open('/etc/passwd2', 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OSError: [Errno 2] No such file or directory: '/etc/passwd2'
```

שימו לב ש-os.open בכלל לא מחזירה file-object אלא מספר, וכאמור די מסובך להשתמש בה בהשוואה ל-file ולכן נשתמש ב-file-objects כמו שלמדנו עד עכשיו.

גם ל-IOError יש errno שבו נוכל לבדוק אילו שגיאות קיבלנו, ובכלל IOError מאוד דומה ל-OSError וכל עוד מבינים את ההבדל בהקשר שבו נקבל כל אחד מה-exception-ים כאלה, אין הרבה מה להרחיב עליו.

NameError

NameError קורה כשאנחנו מנסים לפנות למשתנה שלא קיים. להבדיל מ-exception-ים כמו OSError או IOError, NameError קורה כתוצאה מכך שכתבנו קוד שלא עובד כמו שצריך (לפעמים קוראים לזה באג...).

למשל (שימו לב לשגיאת הכתיב):

```
>>> flei('/etc/passwd').read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'flei' is not defined
```

ואת אותו exception נקבל גם אם נפנה למשתנה שלא קיים בפונקציה או מתודה.

הדבר שחשוב להבין כאן הוא ש-Python מחפשת משתנים בזמן הריצה של הקוד, ומאחר שאין כאן תהליך של קומפילציה כמו ב-C, Python מוצאת שמשתנה לא קיים רק כשהיא מגיעה להריץ את הקוד שכתבנו. לכן, שגיאות שהן לכאורה חלק מהקוד הן גם exception-ים רגילים ומבחינת Python אין הבדל בין הדרך שבה נגיד למשתמש שקובץ לא קיים ובין הדרך שנגיד לו שחסר משתנה בתוכנית שלנו.

KeyError, IndexError

שני ה-exception-ים האלה קופצים כשננסה לגשת ל-key שלא קיים ב-dict או לאינדקס שלא קיים ב-list. לדוגמה, במקרה של מילון:

```
>>> {1: 1}[2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 2
```

ועבור רשימה:

```
>>> [1, 2, 3][4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

SyntaxError

בדומה ל-NameError, כשנכתוב איזושהי פיסת קוד ש-Python לא יכולה להבין, ייזרק SyntaxError. ה-exception הזה אומר שקיימת שגיאה כלשהי בתחביר, כלומר שרשמנו טקסט שאיננו קוד Python. לדוגמה:

```
>>> if 1
  File "<stdin>", line 1
    if 1
       ^
SyntaxError: invalid syntax
```

בדוגמה הזו רשמנו "if 1". אין עם זה שום בעיה, חוץ מהעובדה ששכחנו לשים נקודותיים בסוף השורה. מבחינת Python זה לא יכול להיות ולכן קיבלנו SyntaxError. דוגמה קצת פחות טריוויאלית היא:

```
>>> def func():
...     if 1 == 1:
...     print "OK, the sky is Blue"
  File "<stdin>", line 3
    print "OK, the sky is Blue"
        ^
IndentationError: expected an indented block
```

במקרה הזה שמנו אינדנטציה לא נכונה, ולכן Python התלוננה של שורת ה-print. ה-error הזה הוא סוג של SyntaxError כי:

```
>>> issubclass(IndentationError, SyntaxError)
True
```

דוגמה נוספת כדי להמחיש את הנקודה ולסיים:

```
>>> if x < 1:
...  print 'Blah'
... else:
...  print 'Bleh'
... else:
  File "<stdin>", line 5
    else:
       ^
SyntaxError: invalid syntax
```

כי ברור שאי-אפשר לשים שני else-ים באותו בלוק if.

KeyboardInterrupt

את KeyboardInterrupt נקבל כשהמשתמש יקיש CTRL+C. CTRL+C הוא צירוף מקשים שמותר למשתמש להקיש כשהוא רוצה להגיד לתוכנית להפסיק, ובד"כ תוכניות יוצאות כשהמשתמש מקיש CTRL+C. Python ממירה את ההקשה הזאת ל-exception שפשוט נזרק במקום שבו הקוד שלנו רץ כרגע.

זאת דוגמה מצוינת לכך שגם אם נכתוב קוד מושלם עדיין יהיו exception-ים, כי זה מנגנון שמשמש את Python כדי להודיע לנו שקרה משהו, ולא בהכרח שהייתה איזושהי שגיאה.

כדוגמה, נקרא ל-()get_int_from_user ונקליד CTRL+C בזמן שהפונקציה תצפה לקלט:

```
>>> get_int_from_user()
Enter a number: Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in get_int_from_user
KeyboardInterrupt
```

כמובן, מאחר שזהו exception נוכל לתפוס אותו ולטפל בו:

```
>>> def get_int_from_user():
...     while True:
...         try:
...             raw_num = raw_input('Enter a number: ')
...             return int(raw_num)
...         except ValueError:
...             print '{} is not a number'.format(raw_num)
...         except KeyboardInterrupt:
...             print 'Nice try... now enter a number already'
```

זוהי דוגמה נוספת למקרה בו אפשר לשים כמה בלוקי except כדי לתפוס שגיאות מ-type-ים שונים. ננסה את הפונקציה החדשה שלנו:

```
>>> get_int_from_user()
Enter a number: seven
seven is not a number
Enter a number: Nice try... now enter a number already
Enter a number: 3
3
```

חשוב לזכור ש-Python היא אמנם שפה מגניבה אבל היא לא קוסם. נסתכל לדוגמה על הקוד הבא:

```
>>> def get_int_from_user():
...     try:
...         return int(raw_input('Enter a number: '))
...     except Exception:
...         print 'Some unknown error has occured'
...     except ValueError:
...         print "You didn't enter a number"
...     except KeyboardInterrupt:
...         print 'You gave up'
...
>>> get_int_from_user()
Enter a number: aaa
Some unknown error has occurred
```

למה בלוק ה-except הראשון רץ אם אמרנו בפירוש שאם ValueError קורה אז שירוץ הבלוק השני? הסיבה לכך היא שרשמנו את except Exception לפני ValueError. כש-Python מחפשת את בלוק ה-except שהיא צריכה כדי לטפל בשגיאה, היא עוברת בלוק בלוק עד שהיא מוצאת בלוק except שב-type שלו יש type שעבורו מתקיים isintance עם האובייקט של ה-exception. מאחר ש-ValueError יורש מ-Exception, לעולם לא נריץ את הבלוק של ValueError. לעומת זאת, KeyboardInterrupt יורש מ-BaseException ולא מ-Exception ולכן כשנקרא לפונקציה ונקליד CTRL+C נקבל:

```
>>> get_int_from_user()
Enter a number: You gave up
```

חשוב לזכור שאפשר לשים כמה בלוקי except אבל צריך לשים אותם בסדר הגיוני, כלומר בסדר "עולה" של ירושה בין ה-exception-ים.

TypeError

TypeError הוא exception די נפוץ, ונקבל אותו בכל מקרה שבו העברנו type לא נכון לפונקציה או כשננסה לקרוא לפונקציה עם שילוב פרמטרים לא נכון. למשל:

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

או:

```
>>> def f(x):
...     pass
...
>>> f(1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() takes exactly 1 argument (2 given)
```

TypeError בעצם מאוד דומה ל-ValueError, רק ש-ValueError היא עבור ערכים ו-TypeError היא עבור type-ים.

בנוסף, חשוב לציין ש-TypeError לא מיועדת לשימוש של Python בלבד, ומומלץ מאוד לזרוק TypeError אם ציפינו למשתנה מסוג מסוים וקיבלנו סוג אחר. מאוד מקובל ב-Python לזרוק TypeError במקרים כאלה ומתכנתים אחרים מצפים לקבל TypeError ולא exception ספציפי שנוצר ע"י המתכנת. אם ניזכר בדוגמת החיות שאוכלות מהפרק הקודם:

```
>>> class Animal(object):
...     def eat(self, food):
...         if not isinstance(food, Food):
...             raise TypeError('Animals can only eat food')
...         self.hunger -= food.mana
...         if self.hunger < 0:
...             self.hunger = 0
```

בלוק finally

אחרי שהכרנו את except הגיע הזמן להכיר בלוק נוסף שיכול להגיע אחרי try והוא בלוק finally. אחרי בלוק try אנחנו יכולים לשים כמה בלוקי except שנרצה, או לא לשים except בכלל, ואחריהם בלוק finally. מה שבלוק finally עושה הוא לתת לנו הזדמנות אחרונה להריץ קוד לפני שנצא מבלוק ה-try. נתחיל בדוגמה:

```
>>> try:
...     print 'Before evil error'
...     raise Exception()
...     print 'This cannot happen'
... finally:
...     print 'Finally code'
...
Before evil error
Finally code
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
Exception
```

המטרה של בלוק finally היא לאפשר לנו להריץ קוד ביציאה מבלוק ה-try בלי קשר לשאלה האם נזרק exception. המקרה הפשוט הוא המקרה בו בלוק ה-try הסתיים בהצלחה: במקרה הזה הקוד ממשיך לרוץ מסוף בלוק ה-try ישירות לבלוק ה-finally ומשם מסתיים בלוק ה-try והקוד הנלווה אליו.

אם נזרק exception, Python מוצאת את בלוק ה-except המתאים (אם יש אחד כזה) ומריצה את הקוד שבו. אחרי שבלוק ה-except מסיים את הרצת הקוד שלו, Python מחפשת בלוק finally ואם יש אחד כזה היא מריצה את הקוד שבו אחרי בלוק ה-except.

בנוסף, בלוק ה-finally יתבצע אפילו אם בלוק ה-except זורק exception (לדוגמה ע"י raise). לפני זריקת ה-exception החדש Python תריץ את הקוד של finally ורק אז תמשיך לזרוק את ה-exception.

ההגיון הוא שבלוק finally מאפשר לנו לכתוב קוד שמבצע סיום או ניקוי של מה שהתחלנו בבלוק ה-try. לדוגמה, יכול להיות שנרצה לוודא שסגרנו קובץ לפני שנצא מקטע הקוד שלנו:

```
>>> f = file('/etc/passwd')
>>> try:
...     content = f.read()
... finally:
...     f.close()
```

Python מאלצת אותנו לשים את בלוק ה-finally אחרי כל בלוקי ה-except, וזאת כדי להדגיש שהוא תמיד רץ אחרון – אחרי בלוק ה-try ואחרי בלוק ה-except שאולי ירוץ.

בלוק else

בלוק try מאפשר לנו לעטוף קוד ולהגיד מה נעשה איתו בכמה מקרים: נשים בלוקי except כדי להריץ קוד ספציפי אם היה exception. נשים בלוק finally כדי להריץ חלק מהקוד בכל מקרה (אם היה exception ואם לא היה exception). חסר רק עוד משהו אחד – בלוק שנשים אם לא היה exception.

אם שמנו לפחות בלוק except אחד, נוכל להוסיף אחרי בלוקי ה-except (אבל לפני finally) גם בלוק else של קוד שירוץ רק אם בלוק ה-try הסתיים בהצלחה, כלומר רק אם לא נאלצנו לבדוק אף אחד מבלוקי ה-except שלנו:

```
>>> def check_password(password):
...     if password != 'Pyth0nRul3z':
...         raise ValueError('Wrong password')
...
>>> def authenticate():
...     while True:
...         try:
...             password = raw_input('Password: ')
...             check_password(password)
...         except ValueError, value_error:
...             print str(value_error)
...         else:
...             print 'Welcome!'
...             return
...
>>> authenticate()
Password: 000
Wrong password
Password:
Wrong password
Password: PythonRul3z
Wrong password
Password: Pyth0nRul3z
Welcome!
```

אבל רגע? למה צריך בכלל בלוק else? הרי הוא קורה אם לא היה exception... יכלנו פשוט לכתוב את הקוד ב-authenticate בלי else, פשוט אחרי בלוק ה-except…try.

זה נכון, ובמקרה הספציפי הזה באמת היינו יכולים לממש את בלוק ה-try בלי ה-else, ע"י-כך שהיינו משתמשים ב-break בבלוק ה-except. אבל אם היינו מוסיפים גם בלוק finally, זאת הייתה הדרך היחידה להכניס קוד שיתבצע בין סיום בלוק ה-try לתחילת בלוק ה-finally, שיורץ במקרה שלא היה אף exception ושלא יטופל ע"י בלוקי ה-except של בלוק ה-try שלנו. לדוגמה:

```
>>> def authenticate():
...     history = file('password_history.txt', 'w')
...     while True:
...         password = raw_input('Password: ')
...         try:
...             check_password(password)
...         except ValueError, value_error:
...             print str(value_error)
...             passed = False
...         else:
...             print 'Welcome'
...             passed = True
...             return
...         finally:
...             print >>history, '{} (passed={})'.format(password, passed)
...
>>> authenticate()
Password: 123
Wrong password
Password: blah
Wrong password
Password: Pyth0nRul3z
Welcome
```

ואם נסתכל על הקובץ שנוצר:

```
password_history.txt
```

```
123 (passed=False)
blah (passed=False)
Pyth0nRul3z (passed=True)
```

sys.exc_info()

כמו שציינו מקודם, בפרק הבא נלמד על מודולים, אך כדי לסיים את הפרק על exception-ים עלינו להכיר את exc_info(). במודול sys יש פונקציה חשובה שנקראת exc_info() שכשנקרא לה נקבל:

```
>>> import sys
>>> sys.exc_info()
(None, None, None)
```

במקרה שלנו, קראנו ל-sys.exc_info() כשהיינו "סתם בקוד רגיל". אבל, אם נהיה במהלך טיפול ב-exception נקבל משהו אחר:

```
>>> try:
...     raise ValueError(7)
... except Exception:
...     print sys.exc_info()
...
(<type 'exceptions.ValueError'>, ValueError(7,), <traceback object at 0x7f40d96ad1b8>)
```

כשאנחנו במהלך טיפול ב-exception, Python זוכרת את המידע על ה-exception הנוכחי במקום גלובלי בזיכרון ומאפשרת לנו לגשת לאותו מקום ע"י קריאה ל-sys.exc_info() כדי שנוכל להשתמש באובייקטי ה-exception אם נרצה.

מה שאנחנו מקבלים הוא tuple עם שלושה איברים. הראשון הוא ה-type של ה-exception, השני הוא ה-instance והשלישי הוא אובייקט ה-traceback. כן, אפילו ה-traceback שמודפס למסך הוא בסה"כ אובייקט.

בגלל שיש לנו גישה ל-exception הנוכחי, אנחנו יכולים לדעת כמה דברים. קודם כל, אנחנו יכולים לדעת האם יש כרגע exception בתהליך טיפול מכל מקום שנהיה בו בקוד. בנוסף, אנחנו יכולים לדעת בדיוק איפה בקוד היה ה-exception ע"י-כך שנבחן את אובייקט ה-traceback וגם נוכל לכתוב פונקציות כלליות שיטפלו ב-exception בלי שנעביר להן פרמטרים ספציפיים.

```
>>> def check_password(password):
...     if sys.exc_info()[0] is not None:
...         if issubclass(sys.exc_info()[0], BadPassword):
...             raise RuntimeError("We can't verify a password now")
...     if password != 'Pyth0nRul3z':
...         raise BadPassword(password)
...
>>> def verify_password():
...     while True:
...         try:
...             password = raw_input('Password: ')
...             check_password(password)
...         except BadPassword, bad_password:
...             # Let's check the password again, maybe it'll work
...             check_password(password)
...         else:
...             return
...
>>> verify_password()
Password: blah
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 8, in verify_password
  File "<stdin>", line 4, in check_password
RuntimeError: We can't verify a password now
```

וכך הצלחנו לכתוב פונקציה שמגינה על עצמה מפני משתמשים רעים, בלי שהמשתמש בכלל יודע שהסתכלנו האם יש exception שמטופל כרגע.

דבר נוסף שהיינו יכולים לעשות הוא:

```
>>> def check_password(password):
...     if sys.exc_info()[0] is not None:
...         if issubclass(sys.exc_info()[0], BadPassword):
...             raise
...     if password != 'Pyth0nRul3z':
...         raise BadPassword()
...
>>> verify_password()
Password: blah
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 8, in verify_password
  File "<stdin>", line 5, in verify_password
  File "<stdin>", line 6, in check_password
__main__.BadPassword
```

כלומר, מה שאמרנו מקודם לגבי היכולת לעשות raise מתוך בלוק except תקף גם מתוך פונקציות. Python לא צריכה לראות שאנחנו עושים raise ספציפית מתוך בלוק except, היא פשוט צריכה להסתכל האם יש exception בטיפול כרגע ואם כן אז מותר לעשות raise בלי פרמטרים.

דבר אחרון שאפשר להבין מ-sys.exc_info() הוא שתמיד יש רק exception אחד שמטופל בכל רגע נתון. זה אומר שאם נגרום לזריקת exception מתוך קוד של טיפול ב-exception, ה-exception הראשון ייעלם ויתחיל טיפול ב-exception החדש:

```
>>>
>>> try:
...     raise ValueError('First error')
... except ValueError:
...     print some_bad_variable
...
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
NameError: name 'some_bad_variable' is not defined
```

חלק 7: מודולים

מודול הוא בסה"כ קובץ שמכיל קוד Python. לרוב, לקובץ של מודול Python תהיה סיומת py, אך קיימות גם סיומות אחרות, לדוגמה dll או so (ב-Windows או UNIX), שמציינות קובץ שמכיל פונקציות Python שכתובות ב-C (או באיזושהי שפה מקומפלת אחרת). לדוגמה, מרבית הפונקציות במודול os כתובות בשפת C.

כאשר קובץ הוא בעל סיומת py, Python יודעת שהקובץ מכיל טקסט רגיל, ושצריך להתייחס אליו כאל קוד Python. קוד Python הוא כל דבר שהכרנו עד עכשיו – משתנים, פונקציות, שימושים במודולים אחרים, if, לולאות, ו-class-ים. אם ניצור פונקציה בקובץ כלשהו, ונשמור אותו עם סיומת py, נוכל לטעון את הקובץ ל-Interpreter או למודול אחר ולהשתמש בפונקציה שכתבנו.

לדוגמה, נניח שיש לנו את הקובץ הבא, שנקרא example.py:

example.py

```
def func1(x):
    return [i * 2 for i in xrange(0, x)]

def func2(x):
    return 2 ** x

def func3(x):
    return "The number is {}".format(x)

def func4(x):
    return [x] * x
```

בקובץ יש 4 פונקציות, כאשר שם הקובץ בו הן מאוחסנות הוא example.py.

import

כדי שנוכל להשתמש בפונקציות שכתבנו נשתמש בפקודה import:

```
>>> import example
```

ואחרי שעשינו import למודול נוכל להשתמש בפונקציות שלו:

```
>>> example.func1(2)
[0, 2]
>>> example.func3(65)
'The number is 65'
>>> example.func4(17)
[17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17]
```

כמו שאפשר לראות, כשעשינו import לא כתבנו את השם המלא של הקובץ (example.py) אלא רק example. הסיבה לכך היא ש-Python יודעת לחפש את המודול example ולמצוא שהוא ממומש בקובץ example.py. לנו בתור משתמשים הרי לא אכפת אם המודול הוא עם סיומת py או so או dll או משהו אחר, לנו אכפת שנצליח לעשות import למודול.

Python מבחינתה יודעת למצוא את המודול הנכון ולעשות לו import.

דרך נוספת לעשות import היא לציין בדיוק את הפונקציות שנרצה ש-Python תביא מהמודול:

```
>>> from example import func1
>>> func1(7)
[0, 2, 4, 6, 8, 10, 12]
>>> from example import func2, func3
>>> func2(4)
16
>>> func3(0)
'The number is 0'
```

בדרך הזו Python מקבלת בדיוק את השמות שהיא צריכה לייבא, ואחרי הייבוא אפשר להשתמש באובייקטים שייבאנו בלי לציין את שם המודול.

Doc-string

במודולים כמו בפונקציות אנחנו יכולים לכתוב doc-string בתחילת הקובץ, ואת התיעוד נקבל ב-help של המודול.

נעדכן את example.py:

example.py

```
'''example.py: Contains several useless functions.
'''

def func1(x):
    '''func1(x) -> [ints...]
    Returns a list of numbers from 0 to x where each
    number is doubled.
    '''
    return [i * 2 for i in xrange(0, x)]

def func2(x):
    '''func2(x) -> int
    Returns 2 to the power of x
    '''
    return 2 ** x

def func3(x):
    '''func3(x) -> str
    Returns a string containing x.
    '''
    return "The number is {}".format(x)

def func4(x):
    '''func4(x) -> [ints...]
    Returns a list with x elements of x. It's assumed
    that x is a number.
    '''
    return [x] * x
```

נריץ ונקבל:

```
>>> import example
>>> help(example)
Help on module example:

NAME
    example - example.py: Contains several useless functions.

FILE
    /home/demo/example.py

FUNCTIONS
    func1(x)
        func1(x) -> [ints...]
        Returns a list of numbers from 0 to x where each
        number is doubled.

    func2(x)
        func2(x) -> int
        Returns 2 to the power of x

    func3(x)
        func3(x) -> str
        Returns a string containing x.

    func4(x)
        func4(x) -> [ints...]
        Returns a list with x elements of x. It's assumed
        that x is a number.
```

Namespace-ים

לפני שנבין בדיוק מה ההבדל בין שתי שיטות ה-import שראינו מקודם נצטרך להכיר מושג חדש-ישן שנקרא namespace. בעצם כבר פגשנו namespace בעבר אבל קראנו לו __dict__ – אם אתם זוכרים, בכל פעם שאנחנו יוצרים אובייקט שיכול להכיל attributes הוא מכיל __dict__ שאחראי לאחסן עבורנו את המיפוי בין שם ה-attribute לאובייקט שהוא מצביע אליו.

אותה השיטה קיימת גם עבור משתנים רגילים במודולים, משתנים מקומיים בפונקציות והמשתנים שאנחנו מייצרים ב-interpreter. אם נבדוק, נוכל לראות של-example גם יש __dict__ ונוכל לראות שהפונקציות שלנו שם:

```
>>> example.__dict__.keys()
['func3', 'func2', 'func1', 'func4', '__builtins__', '__file__', '__package__',
'__name__', '__doc__']
```

בדומה לאובייקטים רגילים שיצרנו מקודם, גם במודול נוכל לאחסן attributes כרצוננו:

```
>>> example.func5 = lambda x: x + 1
>>> example.__dict__.keys()
['func3', 'func2', 'func1', 'func5', 'func4', '__builtins__', '__file__',
'__package__', '__name__', '__doc__']
```

ומאחר שיש __dict__, אז גם יש dir():

```
>>> dir(example)
['__builtins__', '__doc__', '__file__', '__name__', '__package__', 'func1', 'func2',
'func3', 'func4', 'func5']
```

ובאופן כללי, namespace הוא כל מיפוי של שמות משתנים לאובייקטים, וב-Python הוא ממומש ע"י מילון.

אם נחשוב על זה קצת, ב-Python יש שני סוגים של namespace-ים: ה-namespace ה"נוכחי" שבו אנחנו רצים בכל רגע ו-namespace-ים אחרים שאנחנו צריכים לפנות אליהם בצורה מפורשת (לדוגמה המודול example).

כאשר אנחנו רצים בתוך פונקציה ופונים למשתנה, Python יודעת בדיוק איפה לחפש את המשתנה. Python גם יודעת לשמור את המשתנים המקומיים של הפונקציה כך שכשהיא תסיים לרוץ המשתנים האלה יושמדו ולא נראה אותם יותר. לכן, יש עוד namespace שמוחבא מאיתנו ושעד היום לא שמנו לב אליו. ה-namespace הזה נקרא locals.

נתחיל בפונקציה פשוטה שמחזירה רשימה:

```
>>> def f():
...     x = []
...     return x
```

בפונקציה הזו נוצרת רשימה חדשה שמוכנסת למשתנה x, ולאחר מכן x מוחזר למי שקרא לפונקציה. בפועל, האובייקט היחיד שמוגן בכל הסיפור הזה הוא הרשימה, כי רק על הרשימה יש reference-count שסופר כמה משתנים מצביעים אליה.

אם ננסה לבדוק ב-interpreter, נוכל לראות שבכמה קריאות עוקבות ל-()f נקבל את "אותה הרשימה":

```
>>> id(f())
139935593701816
>>> id(f())
139935593701816
>>> id(f())
139935593701816
```

הסיבה לכך היא מאוד פשוטה: מיד כשהרשימה מוחזרת מ-()f אנחנו בודקים את ה-()id שלה, ולאחר מכן הרשימה מושמדת כי אף אחד לא מצביע אליה יותר. בקריאה הבאה נוצרת שוב רשימה חדשה בכתובה שבה הייתה הרשימה הישנה שהושמדה.

כעת נשנה קצת את ()f:

```
>>> def f():
...     x = []
...     return locals()
```

()locals היא פונקציה שמחזירה את ה-namespace המקומי, כלומר את ה-namespace שמחזיק את המשתנים המקומיים. נראה מה ()f מחזירה:

```
>>> f()
{'x': []}
```

כלומר השמה וקריאת משתנה ב-Python הן בסה"כ פניות למילון ש-Python יצרה עבורנו מראש. כאשר יוצאים מפונקציה, כל מה ש-Python צריכה לעשות הוא להוריד את ה-reference-count למילון, וכאשר המילון יושמד הוא יוריד את ה-reference-count לכל האובייקטים שהוא מצביע אליהם, וכך הלאה.

משתנים גלובליים

בנוסף למשתנים המקומיים (locals) קיים ב-Python גם המושג של משתנים גלובליים. משתנים גלובליים, בניגוד למשתנים לוקליים, מוגדרים ברמת המודול כולו. לדוגמה, נסתכל על המודול mod1:

```
mod1.py
CONST = 17

def multiply(num):
    return num * CONST
```

כשאנחנו קוראים לפונקציה multiply, המשתנה num הוא משתנה מקומי שנוצר בעת הקריאה לפונקציה. אבל איך multiply יודעת בזמן הריצה למצוא את המשתנה CONST? בזמן שהפונקציה רצה היא מחפשת את המשתנה CONST ב-()locals ולא מוצאת. בנקודה הזו היא עוברת לחפש את CONST במשתנים הגלובליים, כלומר ב-namespace של המודול כולו. באותו namespace מוגדרים גם לדוגמה multiply עצמה וה-__doc__ של המודול כמו שראינו מקודם.

כדי לגשת ל-dict של המשתנים הגלובליים נוכל לקרוא לפונקציה ()globals. כדי להדגים את ()globals, נוסיף את הפונקציה get_globals ל-mod1:

```
mod1.py
CONST = 17

def multiply(num):
    return num * CONST

def get_globals():
    return globals()
```

ונקרא ל-()get_globals כדי לקבל את ה-globals של mod1:

```
>>> import mod1
>>> mod1.get_globals().keys()
['CONST', '__builtins__', '__file__', '__package__', 'get_globals', 'multiply',
'__name__', '__doc__']
```

וכאן אפשר לראות בבירור שבמשתנים הגלובליים של mod1 מוגדרים CONST, get_globals ו-multiply, בנוסף לכמה attributes נוספים שמוגדרים בכל מודול:

__name__

__name__ הוא attribute מסוג str שמכיל את השם של המודול. בד"כ נקבל מחרוזות פשוטות כמו 'mod1', אך בהמשך נראה דוגמאות פחות טריוויאליות בהן קראנו למודול mod1 אך Python מפיקה שם קצת יותר מסובך.

__file__

מחרוזת המכילה את השם של הקובץ ממנו המודול יובא. שם הקובץ שנקבל במחרוזת הזו הוא לא תמיד path אבסולוטי, למשל במקרה של mod1:

```
>>> mod1.__file__
'mod1.pyc'
```

כלומר שם הקובץ כאן הוא ביחס לספריית העבודה הנוכחית, ולכן __file__ שימושי רק בהנחה שלא שינינו את ספריית העבודה הנוכחית בעזרת os.chdir().

__package__

מציין את שם ה-package שבו המודול נמצא. בהמשך הפרק נלמד על package-ים.

__builtins__

עד עכשיו פגשנו הרבה פונקציות ואובייקטים שהיו "מובנים", כלומר פשוט השתמשנו בהם והם הגיעו מאיפושהו. משתנים מקומיים הגיעו מ-dict שנקרא locals, משתנים גלובליים הגיעו מ-dict שנקרא globals, אז כנראה שכל האובייקטים האלה גם צריכים להיות מוגדרים באיזשהו dict. ה-dict הזה נקרא __builtins__ והוא מוכנס אוטומטית לכל global-namespace בכל פעם שמודול מיובא.

כשאנחנו כותבים שם כלשהו, Python מחפשת אותו ב-locals. אם היא לא מוצאת היא עוברת לחפש ב-globals. אם היא לא מוצאת את השם שחיפשנו גם שם, היא עוברת לחפש ב-__builtins__, ואם גם שם לא נמצא אז ייזרק NameError.

__builtins__ הוא מודול לכל דבר, ונוכל להתסכל על כל השמות שמוגדרים בו:

```
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
'BufferError', 'BytesWarning', 'DeprecationWarning', 'EOFError', 'Ellipsis',
'EnvironmentError', 'Exception', 'False', 'FloatingPointError', 'FutureWarning',
'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError',
'IndexError', 'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError',
'NameError', 'None', 'NotImplemented', 'NotImplementedError', 'OSError',
'OverflowError', 'PendingDeprecationWarning', 'ReferenceError', 'RuntimeError',
'RuntimeWarning', 'StandardError', 'StopIteration', 'SyntaxError', 'SyntaxWarning',
'SystemError', 'SystemExit', 'TabError', 'True', 'TypeError', 'UnboundLocalError',
'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError',
'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning', 'ZeroDivisionError', '_',
'__debug__', '__doc__', '__import__', '__name__', '__package__', 'abs', 'all', 'any',
'apply', 'basestring', 'bin', 'bool', 'buffer', 'bytearray', 'bytes', 'callable',
'chr', 'classmethod', 'cmp', 'coerce', 'compile', 'complex', 'copyright', 'credits',
'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'execfile', 'exit', 'file',
'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr', 'hash',
'help', 'hex', 'id', 'input', 'int', 'intern', 'isinstance', 'issubclass', 'iter',
'len', 'license', 'list', 'locals', 'long', 'map', 'max', 'memoryview', 'min', 'next',
'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit', 'range',
'raw_input', 'reduce', 'reload', 'repr', 'reversed', 'round', 'set', 'setattr',
'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'unichr',
'unicode', 'vars', 'xrange', 'zip']
```

שימו לב שאתם כבר צריכים להכיר חלק לא קטן מהאובייקטים שיש כאן, ועכשיו אנחנו גם יכולים לראות איפה האובייקטים האלה חיים.

locals is globals

נקודה נוספת שחשוב לשים לב אליה היא שה-namespace-ים מתחלפים במהלך ריצת התוכנית. הכוונה היא שחלק עיקרי מהעבודה של Python הוא לוודא שבכל רגע אנחנו רצים עם locals ו-globals שמתאימים לקטע הקוד שמורץ כרגע. הרי לא נרצה להיות בתוך הפונקציה x עם משתנים לוקליים של הפונקציה y...

בחלק מהזמן אין משמעות למשתנים לוקליים, למשל כשאנחנו מריצים קוד ב-global-namespace. דוגמה להרצת קוד ב-global-namespace היא הרצת השורה CONST=17 בזמן ה-import של mod1.

בנקודות כאלה, locals() מחזיר את ה-global-namespace, כלומר אם נריץ ב-interpreter:

```
>>> locals() is globals()
True
```

נוכל לראות שלפעמים המשתנים הלוקליים והגלובליים הם בדיוק אותו מילון.

global

אם נרצה נוכל נסכם את כל מה שלמדנו עד עכשיו בהקשר של משתנים מקומיים במשפט אחד: "כל מה שנוצר בתוך פונקציה נשאר בתוך הפונקציה". נשמע טריוויאלי, אך בואו נסתכל על הדוגמה הבאה:

```
>>> CONST = 17
>>> def f():
...     CONST = 18
...
>>> f()
>>> CONST
17
```

CONST הוא אמנם משתנה גלובלי, אבל כשאנחנו מריצים את השורה "CONST = 18" נוצר משתנה מקומי ב-f() עם הערך 18, וברגע שאנחנו יוצאים מהפונקציה המשתנה מושמד ו-CONST הגלובלי נשאר כמו שהיה.

אם נרצה לשנות את CONST, נצטרך להגיד ל-Python ש-CONST קיים, ואת זה עושים בעזרת הצהרת global:

```
>>> def f():
...     global CONST
...     CONST = 18
...
>>> f()
>>> CONST
18
```

שימו לב ש-global היא statement ולא נוכל להציב בה ערך ל-CONST. global בסה"כ אומרת ל-Python "תשמעי, את המשתנה הזה תקחי מ-globals() ולא מ-locals()".

כמו כן, נוכל ליצור הצהרת global למשתנים שעוד לא קיימים וכך לאתחל אותם (הרי Python לא ידעה אם CONST קיים כשכתבנו global CONST, היא רק ידעה להציב אותו ב-globals()):

```
>>> def g():
...     global BLAH
...     BLAH = 123
...
>>> BLAH
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'BLAH' is not defined
>>> g()
>>> BLAH
123
```

איך import עובד

כשאנחנו עושים import למודול, Python מוצאת את המודול עבורנו, ובהתאם לסוג ה-import שעשינו היא מוסיפה את המודול או את ה-attributes שעשינו להם import ל-local-namespace.

כלומר, אם נעשה import מתוך פונקציה, המודול שעשינו לו import לא יהיה זמין עבורנו מחוץ לפונקציה:

```
>>> def func():
...     import xml
...
>>> func()
>>> 'xml' in globals()
False
```

עד כאן זה נראה בסדר. אבל מה יקרה אם נכתוב שני מודולים באופן הבא:

m1.py	m2.py

`import m2`
 	
`def f1(a, b):`
`  return m2.f2(a) + m2.f2(b)`
	`import m1`
	
	`def f2(x):`
	`  if x < 0:`
	`  return 0`
	`  return f1(x - 1, x - 1) + 7`

האם נוכל לעשות import ל-m1 או ל-m2 בלי שנקבל exception? כן. הסיבה לכך היא ש-Python זוכרת את המודולים שהיא עשתה להם import במקום מרכזי שנקרא sys.modules.

את sys כבר פגשנו מקודם, וראינו שהוא סיפק לנו מידע על ה-exception הנוכחי שבטיפול. למעשה, sys מספק לנו הרבה attributes אחרים שכולם משפיעים על ההתנהגות של Python עצמה. sys הוא המודול שמייצג את "מערכת ה-Python" שאנחנו רצים עליה. במקרה שלנו, sys.modules הוא מילון שמכיל מיפוי בין שם מודול לאובייקט המודול שממנו המודול יובא.

sys.modules

כשאנחנו עושים import, Python לא ממהרת לחפש קובץ שמממש את המודול שלנו. קודם כל היא מסתכלת ב-sys.modules כדי לראות אם המודול כבר יובא בעבר. אם כן, היא פשוט מחזירה לנו את המודול. אם לא, היא מחפשת את הקובץ המתאים ומתחילה את תהליך ה-import. במקרה שלנו:

```
>>> import m1
>>> import m2
>>> import sys
>>> sys.modules.keys()
['copy_reg', 'sre_compile', '_sre', 'encodings', 'site', '__builtin__', 'sysconfig',
'__main__', 'encodings.encodings', 'abc', 'posixpath', '_weakrefset', 'errno',
'encodings.codecs', 'sre_constants', 're', '_abcoll', 'm1', 'types', '_codecs',
'_warnings', 'genericpath', 'stat', 'zipimport', 'encodings.__builtin__', 'warnings',
'UserDict', 'm2', 'encodings.utf_8', 'sys', 'codecs', 'readline', 'os.path',
'sitecustomize', 'signal', 'traceback', 'apport_python_hook', 'linecache', 'posix',
'encodings.aliases', 'exceptions', 'sre_parse', 'os', '_weakref']
```

כאשר Python מוצאת את הקובץ שרצינו לעשות לו import, היא מייצרת אובייקט מודול ריק ב-sys.modules וממשיכה בתהליך ה-import. בשיטה הזו, אם פקודות שנריץ בזמן ה-import יגרמו לפניה מעגלית למודול שלנו (כמו במקרה של m1 שמייבא את m2 שמייבא את m1 בחזרה), ה-import השני יסתיים מאוד מהר כי פשוט יוחזר אובייקט המודול שכבר ייצרנו ב-sys.modules ו-Python לא תצטרך לעשות דבר מעבר לכך.

למעשה, בזמן ש-Python מבצעת את ה-import הפנימי, אין לה מושג שהיא במהלך import אחר, כי היא מצאה את המודול ב-sys.modules וזה כל מה שחשוב במהלך import מבחינת השפה.

sys.path

נקודה אחרונה שלא כיסינו היא איפה Python מוצאת את המודולים שביקשנו לעשות להם import. מודולים כמו mod1 ו-example שכתבנו במהלך הפרק הם מקרה יחסית פשוט, כי שמנו אותם בספריה מסויימת והרצנו את Python מהספריה הזו. אבל מה עם sys או os, או מודולים אחרים שנראה בהמשך כמו itertools?

ב-sys יש attribute חשוב נוסף בשם sys.path שמכיל רשימה עם כל ה-path-ים שבהם Python צריכה לחפש מודולים כשהיא נתקלת ב-import. דוגמה אחת לרשימה כזו*:

```
>>> sys.path
['', '/usr/lib/python2.7', '/usr/lib/python2.7/plat-linux2', '/usr/lib/python2.7/lib-
tk', '/usr/lib/python2.7/lib-old', '/usr/lib/python2.7/lib-dynload',
'/usr/local/lib/python2.7/dist-packages', '/usr/lib/python2.7/dist-packages',
'/usr/lib/python2.7/dist-packages/PIL', '/usr/lib/python2.7/dist-packages/gst-0.10',
'/usr/lib/python2.7/dist-packages/gtk-2.0', '/usr/lib/pymodules/python2.7',
'/usr/lib/python2.7/dist-packages/ubuntu-sso-client', '/usr/lib/python2.7/dist-
packages/ubuntuone-client', '/usr/lib/python2.7/dist-packages/ubuntuone-control-
panel', '/usr/lib/python2.7/dist-packages/ubuntuone-couch', '/usr/lib/python2.7/dist-
packages/ubuntuone-installer', '/usr/lib/python2.7/dist-packages/ubuntuone-storage-
protocol']
```

במקרה שלנו, האיבר הראשון של sys.path הוא מחרוזת ריקה, והיא מסמנת את הספריה הנוכחית. אם נרצה, נוכל להוריד את האיבר הזה מ-sys.path ואז לא יתבצע import מהספריה הנוכחית.

למעשה, נוכל לערוך את sys.path בזמן הריצה של Python כדי לאפשר לה למצוא ספריות במקומות נוספים שבהם היא לא ידעה שעליה לחפש. לדוגמה ניצור ספריה בשם lib בספריה הנוכחית שבה אנחנו רצים ונעביר אליה את m1.py ואת m2.py. כעת ננסה לייבא את m1 ו-m2:

* שימו לב ש-sys.path משתנה בין התקנות שונות של מחשבים שונים כתלות בדרך שבה Python נבנתה, ולכן יכול להיות ש-sys.path יהיה שונה על המחשב שלכם (למשל אם אתם מריצים Windows).

```
>>> import m1, m2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named m1
```

ברור שלא הצלחנו. כעת נוסיף את lib ל-sys.path:

```
>>> import sys
>>> sys.path.append('lib')
>>> import m1, m2
```

כדאי לזכור ששינוי של sys.path הוא דבר שכדאי לבצע בזהירות, כי לפעמים אנחנו עלולים להוסיף ספריות שיגרמו יותר נזק מתועלת, למשל במקרה של מודולים עם שמות כפולים.

כאשר משנים את sys.path חשוב לזכור להשתמש בתת-מודול של os שנקרא os.path (עליו נלמד בפרק הבא), ובאופן כללי כדאי להימנע מכך ולהשתמש ב-package-ים, עליהם נלמד בהמשך הפרק הזה.

import *

עכשיו אנחנו הולכים ללמוד משהו רע. הסיבה שאנחנו הולכים ללמוד את זה היא כדי שנדע שזה קיים, כי יש קוד שמשתמש ביכולת הזו של Python ובטוח תיתקלו בו, אבל תזכרו שעדיף לא להשתמש ב-import כמו שנראה מיד.

ראינו מקודם שאפשר לעשות import לשם של מודול או ל-attributes מתוך המודול. אם נסתכל על דוגמה לדרך השניה מביניהן:

```
>>> from sys import path
>>> from sys import modules
```

נראה שעשינו import ל-path ול-modules מ-sys ל-local-namespace שלנו. עכשיו כשנרצה לפנות ל-sys.path לא נצטרך לרשום "sys.path" אלא נוכל להסתפק ב-path. כמו כן, בשיטה הזו לא יתבצע כלל import ל-sys והוא בכלל לא יהיה מוגדר ב-namespace שלנו:

```
>>> len(path)
18
>>> len(modules)
42
>>> sys
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sys' is not defined
```

אך חשוב לזכור שבשתי שיטות ה-import השימוש ב-sys.modules תמיד זהה, כלומר ב-sys.modules יווצר בסוף המקרה שלנו מופע של sys, ושיטת ה-import השניה היא בסה"כ שיטה קצת יותר נוחה לייבא attributes שנשתמש בהם הרבה ולכן לא נרצה לציין בכל שימוש שלהם את שם המודול ממנו הם באו.

אבל, במקרים מסוימים יש מודולים שמכילים ממש הרבה attributes שנרצה להשתמש בהם, ולכן הומצא הפיצ'ר שנקרא import *, והוא נראה ככה:

```
>>> from sys import *
```

מה שעשינו הרגע הוא להגיד ל-Python "קחי כל מה שיש ב-sys ותשימי אותו ב-namespace שלי". זה לא סבבה, כי אין לנו מושג מה יש ב-sys. אנחנו מכירים את path, modules ו-exc_info, אבל יש שם עוד הרבה דברים שאנחנו לא צריכים, והרגע זיהמנו את ה-namespace שלנו.

יותר מכך, אנחנו יכולים לחשוב שהכל בסדר ולעשות import * לעוד מודול:

```
>>> from sys import *
>>> from os import *
```

כשגם ב-os וגם ב-sys יש attribute בשם path, ועכשיו os.path דרס את sys.path ואין לנו מושג שזה קרה.

לכן, נעדיף להימנע מקוד שעושה import *.

אבל, אם בכל זאת נרצה לאפשר למי שמשתמש במודול שאנחנו כותבים לעשות import *, נרצה לציין את השמות שהמשתמש יעשה להם import, ואת זה נעשה בעזרת __all__. אם ניקח לדוגמה את m1 ו-m2 שראינו מקודם, נוכל להוסיף להם את __all__ כדי שאם מישהו יעשה import * לאחד מהם הוא יקבל רק את הפונקציות שכתבנו ולא את המודולים שיש ב-namespace-ים שלהם:

m1.py

```
__all__ = ['f1']

import m2

def f1(a, b):
    return m2.f2(a) + m2.f2(b)
```

m2.py

```
__all__ = ['f2']

import m1

def f2(x):
    if x < 0:
        return 0
    return f1(x - 1, x - 1) + 7
```

וכעת נריץ ונשווה את locals() לפני ואחרי ה-import:

```
>>> locals().keys()
['__builtins__', '__name__', '__doc__', '__package__']
>>> from m1 import *
>>> from m2 import *
>>> locals().keys()
['f1', 'f2', '__builtins__', '__package__', '__name__', '__doc__']
```

כלומר לא קיבלנו ב-import את m2 (שהיה מגיע מ-m1) ולא את m1 (שהיה מגיע מ-m2) אלא רק את ה-attributes שכותב המודול החליט שכדאי שנייבא.

רק לצורך הדוגמה, כך זה היה נראה בלי ה-__all__ במודולים:

```
>>> locals().keys()
['__builtins__', '__name__', '__doc__', '__package__']
>>> from m1 import *
>>> from m2 import *
>>> locals().keys()
['f1', 'f2', '__builtins__', '__package__', 'm1', 'm2', '__name__', '__doc__']
```

reload

לפעמים אנחנו עורכים מודול בקובץ אבל בודקים אותו תוך-כדי ב-interpreter. אחרי שהבנו שאנחנו לא יכולים לעשות import פעמיים (כי Python תשתמש בעותק שב-sys.modules ולא תסתכל על הקובץ המעודכן שלנו), צריכה להיות דרך אחרת להכריח את Python לעשות import. הדרך הזאת נקראת reload.

reload היא פונקציה שמקבלת מודול ומבצעת לו טעינה מחדש מהקובץ שממנו המודול יובא במקור:

```
>>> reload(m1)
<module 'm1' from 'm1.pyc'>
```

reload גם מחזירה את האובייקט של המודול, אך אין בו צורך מאחר שה-import קורה על האובייקט המקורי של המודול.

הסיבה שזה עובד ככה היא שב-Python הרבה יותר חשוב שיהיה עותק אחד מכל מודול מאשר שנקבל עותק עדכני אחרי ה-import. אם לדוגמה עשינו reload ל-m1, אז גם העותק של m1 ב-m2 צריך להתעדכן, וזה לא יקרה אם ניצור אובייקט חדש (איך נדע מהם כל המקומות שצריך לעדכן בהם את המופע של m1?).

אבל, השיטה הזו מייצרת גם בעיה לא קטנה – מה יקרה אם מחקנו attribute מ-m1? Python לא תדע למחוק את ה-attribute הזה מהמודול ה"חדש" כי כל מה שהיא עושה הוא לקרוא את הקובץ ולהריץ אותו על גבי אובייקט המודול שכבר היה לה. לדוגמה, אם נכתוב את m1 מחדש לגמרי (אבל נשאיר את m2 כמו שהוא בזיכרון אחרי import):

m1.py	m2.py

`import m2`
 	
`def totally_new_func(x):`
`  return x + 1`
	`import m1`
	
	`def f2(x):`
	`  if x < 0:`
	`  return 0`
	`  return f1(x - 1, x - 1) + 7`

אז אחרי ה-reload, m2.f2 תרוץ בלי שום בעיה כי המודול m1 יכיל גם את f1 וגם את totally_new_func. לכן, מומלץ מאוד לא להשתמש ב-reload אלא אם אנחנו מודעים להתנהגות הזו ויודעים שרק עדכנו attributes ולא מחקנו attributes בצורה שתשפיע על ההתנהגות של הקוד שלנו.

סקריפטים

Python מאפשרת לנו לא רק ליצור מודולים ולייבא אותם, אלא גם להריץ קבצי Python בתור תוכניות. הדרך הפשוטה ביותר ליצור תוכנית Python היא ליצור קובץ עם סיומת py ולהריץ אותו בעזרת ה-executable של Python (ב-Linux פשוט מקישים python וב-Windows נצטרך לציין את ה-path, שהוא בד"כ C:\Python27\Python.exe).

כשאנחנו כותבים קובץ Python שבכוונתנו להריץ אנחנו קוראים לו סקריפט (script).

נכתוב לדוגמה את הסקריפט הבא:

```
yo_dog.py
print "What's your cat name?"
cat_name = raw_input('')

print "What's your dog name?"
dog_name = raw_input('')

print 'You have a dog named {} and a cat named {}'.format(dog_name, cat_name)
```

ונריץ אותו בלינוקס:

```
$ python yo_dog.py
What's your cat name?
Cat
What's your dog name?
Dog
You have a dog named Dog and a cat named Cat
```

השורה הראשונה (שמתחילה ב-$) היא שורת הפקודה שהקלדנו בלינוקס כדי להריץ את הסקריפט שלנו.

במקרה הזה, הסקריפט היה אוסף פקודות ש-Python הריצה אחת אחרי השניה. אבל אם נרצה, נוכל לכתוב קוד קצת יותר שימושי:

```
yo_dog.py
def main():
    print "What's your cat name?"
    cat_name = raw_input('')

    print "What's your dog name?"
    dog_name = raw_input('')

    print 'You have a dog named {} and a cat named {}'.format(dog_name, cat_name)

main()
```

ההבדל בין הגרסאות של yo_dog.py הוא שעכשיו yo_dog הוא קובץ Python שמכיל פונקציה אחת בשם main() שעושה את העבודה של yo_dog. ברגע שאנחנו מריצים את yo_dog.py, Python מגדירה את הפונקציה main() ורק בשורה האחרונה מריצה אותה.

בשיטה הזו היינו יכולים לעשות import ל-yo_dog ולקרוא ל-main כמה פעמים שהיינו רוצים. יש רק בעיה אחת, והיא שאם נעשה import ל-yo_dog נגרום להרצה של main. נראה את זה קורה:

```
>>> import yo_dog
What's your cat name?
X
What's your dog name?
Y
You have a dog named Y and a cat named X
```

בעצם, היינו רוצים גם להגדיר את הקוד בפונקציה בשם main שנריץ כשהמודול יורץ בתור סקריפט, אבל גם לדאוג שהמודול שלנו לא יריץ את main אם עשו לו import רגיל. ככה מי שיעשה import למודול יוכל לקרוא ל-main כמה

פעמים שהוא ירצה ואיפה בקוד שהוא ירצה (ולא להכריח אותו להכניס את השמות של הכלבים והחתולים שלו בזמן ה-import).

בשביל לעשות את זה, Python עושה דבר מאוד נחמד: כשמריצים את המודול שלנו בתור סקריפט, השם שלו (__name__) הוא לא השם של המודול, אלא '__main__'. נתקן את yo_dog.py:

yo_dog.py

```
def main():
    print "What's your cat name?"
    cat_name = raw_input('')

    print "What's your dog name?"
    dog_name = raw_input('')

    print 'You have a dog named {} and a cat named {}'.format(dog_name, cat_name)

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

וכעת נריץ את הסקריפט:

```
$ python yo_dog.py
What's your cat name?
Cat
What's your dog name?
Dog
You have a dog named Dog and a cat named Cat
```

וגם נוכל לעשות לו import בלי לחשוש שאיזושהי פונקציה תורץ:

```
>>> import yo_dog
>>>
```

הרצת קבצי Python ב-Windows

ב-Windows קיימים כמה דברים נוספים שכדאי לדעת כשמריצים קבצי py.

דבר ראשון הוא שאם נרצה להריץ תוכניות Python מ-cmd.exe, תמיד נצטרך לכתוב את ה-path ל-Python.exe לפני שם הסקריפט, לדוגמה:

```
C:\> C:\Python27\Python.exe yo_dog.py
```

אם אנחנו משתמשים ב-Python הרבה, נוכל להוסיף את C:\Python27 ל-$PATH$ שלנו דרך System Properties ב-Control Panel, ואז נצטרך להקליד רק Python.exe לפני הסקריפט.

בנוסף, כשאנחנו מתקינים את Python על Windows, היא דואגת לרשום את עצמה עבור קבצי py כך שאם נקליד פעמיים על קובץ py ייפתח חלון cmd ובו התוכנית שלנו תרוץ. החסרון של השיטה הזו הוא שאם יקפוץ exception החלון ייסגר מייד ולא נוכל לראות אותו.

הרצת קבצי Python ב-UNIX

הרצת סקריפטי פיתון ב-UNIX דומה לרצה שלהם ב-Windows, הבדל אחד קטן – אנחנו יכולים לציין את ה-path ל-executable של Python בסקריפט עצמו, ואחרי שנסמן אותו כ-executable נוכל להריץ אותו כמו תוכנית רגילה.

למעשה, אחרי שהקובץ הוא executable אין גם צורך בסיומת py ונצטרך להשאיר אותה רק אם נרצה שהמודול יוכל לעשות import בנוסף להיותו סקריפט. נעדכן את yo_dog.py כך שירוץ מ-Linux:

yo_dog.py

```
#!/usr/bin/python
def main():
    print "What's your cat name?"
    cat_name = raw_input('')

    print "What's your dog name?"
    dog_name = raw_input('')

    print 'You have a dog named {} and a cat named {}'.format(dog_name, cat_name)

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

נהפוך את yo_dog.py ל-executable (צריך לעשות את זה רק פעם אחת לכל סקריפט):

```
$ chmod +x yo_dog.py
```

ועכשיו נוכל להריץ אותו כמו תוכנית רגילה:

```
$ ./yo_dog.py
What's your cat name?
...
```

פרמטרים לתוכנית פיתון

דבר נוסף שאנחנו יכולים לעשות בפיתון הוא להעביר לסקריפט פרמטרים בשורת הפקודה, בדומה לתוכנית בהרבה שפות אחרות. כל פרמטר שנעביר ייכנס ל-sys.argv, ונוכל לגשת ל-sys.argv כדי לראות את הפרמטרים האלה. לדוגמה, נכתוב תוכנית קצרה שמדפיסה את הפרמטרים שהיא מקבלת:

prints_args.py

```
#!/usr/bin/python
import sys
print sys.argv
```

ונריץ:

```
$ ./prints_args.py
['./prints_args.py']
```

אפשר לראות ש-sys.argv[0] הוא שם הסקריפט שלנו. עכשיו נבדוק את התוכנית עם פרמטרים נוספים:

```
$ ./prints_args.py 1 2 3
['./prints_args.py', '1', '2', '3']
```

כלומר כל פרמטר נוסף שנעביר לתוכנית יוכנס כאיבר ל-sys.argv. אפשר להשתמש בזה כדי לקבל פרמטרים לתוכנית, וגם לוודא שקיבלנו את הפרמטרים שציפינו להם.

לדוגמה, נממש תוכנית שמעתיקה קובץ ממקום למקום. התוכנית שלנו תקבל כפרמטר הראשון את הקובץ שצריך להעתיק וכפמרטר שני את שם הקובץ החדש שצריך ליצור:

copy.py

```
#!/usr/bin/python
import os
import sys

def main():
    if len(sys.argv) != 3:
        print 'Synopsis: {} [src-file] [dest-file]'.format(sys.argv[0])
        raise SystemExit(1)

    source_file = sys.argv[1]
    dest_file = sys.argv[2]

    if not os.path.exists(source_file):
        print '{} does not exist'.format(source_file)
        raise SystemExit(2)

    file(dest_file, 'wb').write(file(source_file, 'rb').read())

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

בסקריפט הזה יש 3 דברים חדשים:

- פונקציה שימושית בשם os.path.exists: מקבלת path ומחזירה True רק אם קיים קובץ כזה.
- כאשר אנחנו זורקים SystemExit התוכנית שלנו יוצאת. את SystemExit יוצרים עם ערך ההחזרה של התוכנית שלנו (ערך ההחזרה הוא הערך שמערכת ההפעלה מקבלת מהתוכנית שלנו. הערך 0 אומר הצלחה וכל ערך אחד הוא כשלון).
- כדי לבדוק את sys.argv בדקנו את האורך שלו.

לגבי הנקודה האחרונה, בדיקת האורך של sys.argv היא טכניקה שמקובלת בשפות עתיקות כמו C, בהן אין רשימות ולכן מקבלים את argv ואת כמות הפרמטרים שהתוכנית שלנו קיבלה.

מאחר ש-Python קצת יותר משוכללת נוכל לכתוב את הקוד שלנו בצורה קצת יותר קריאה בעזרת exception-ים:

```
copy.py
#!/usr/bin/python
import os
import sys

def main():
    try:
        exe, source_file, dest_file = sys.argv
    except ValueError:
        print 'Synopsis: {} [src-file] [dest-file]'.format(sys.argv[0])
        raise SystemExit(1)

    if not os.path.exists(source_file):
        print '{} does not exist'.format(source_file)
        raise SystemExit(2)

    file(dest_file, 'wb').write(file(source_file, 'rb').read())

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

תזכורת: אפשר לעשות השמה של כמה משתנים באותה השורה, וכן השמה של משתנים מ-list/tuple למשתנים ספציפיים.

במקרה שלנו פרקנו את sys.argv לשלושה משתנים בשם exe, source_file ו-dest_file. אם ב-sys.argv היו יותר או פחות מ-3 איברים, היינו מקבלים ValueError, ולכן אנחנו תופסים את ה-exception הזה ומניחים שאם הוא קפץ אז ה-command line לא היה תקין.

בפרק הבא נכיר את המודול argparse שעוטף עבורנו את sys.argv ומאפשר לתוכניות שלנו להיות הרבה יותר נוחות לשימוש.

Packages

במהלך הפרק ראינו שאנחנו יכולים להפריד את הקוד שלנו לקבצים, שזה בפני עצמו דבר חשוב מאוד. בנוסף, ראינו שאנחנו יכולים לאחסן את הקבצים האלה בספריות שונות ולהוסיף path-ים ל-sys.path כדי ש-Python תמצא את המודולים שלנו.

אבל כאן נוצרות כמה בעיות:

- אנחנו לא רוצים לערוך את sys.path כי לא תמיד נדע איפה בדיוק נמצא הקוד שלנו. גם אם ננסה למצוא את המיקום של הקוד שלנו בצורה דינמית בעזרת __file__, זה לא מבטיח לנו שהתוצאה תהיה נכונה כי תמיד קיים סיכוי שמישהו ישנה את ספריית העבודה הנוכחית.
- סיבה נוספת לא לשנות את sys.path היא שאחרי לא הרבה import-ים של כמה ספריות נסיים עם sys.path שמכיל הרבה "זבל" שאין לנו צורך בו יותר.
- שינוי של sys.path לא נותן פתרון למקרה שבו שני מתכנתים שונים כותבים שתי ספריות, ובשתיהן קיים מודול בעל אותו שם.

בגלל הבעיות האלה הרחיבו ב-Python את קונספט המודולים ויצרו Package-ים. בגדול, Package זאת ספריה שמכילה קבצי Python. מה שמיוחד בספריה הזו הוא שהיא מכילה קובץ בשם __init__.py.

כדוגמה, ניצור ספריה בשם pkg ובתוך pkg ניצור קובץ ריק בשם __init__.py:

```
$ find pkg/
pkg/
pkg/__init__.py
```

כעת ניצור את המודול הבא בתוך pkg:

```
pkg/mod.py
def func1(x):
    return x + 1

def func2(y):
    return func1(y) * 2
```

עכשיו, כשאנחנו בספריה שמכילה את pkg, נריץ את Python ונראה מה יש בתוך pkg:

```
>>> import pkg
>>> dir(pkg)
['__builtins__', '__doc__', '__file__', '__name__', '__package__', '__path__']
```

מוזר... ב-pkg אין כלום חוץ מהדברים הרגילים שיש במודול. כלומר pkg הוא בסה"כ מודול ריק. כעת נעשה import ל-pkg.mod:

```
>>> import pkg.mod
>>> dir(pkg.mod)
['__builtins__', '__doc__', '__file__', '__name__', '__package__', 'func1', 'func2']
```

כלומר, כשאנחנו מייצרים מודולים בתוך package, אנחנו צריכים לעשות להם import בצורה מפורשת. הסיבה לכך היא שיש ב-Python כמה package-ים מאוד גדולים שבד"כ נרצה לעשות import רק לחלק מאוד קטן מהם ולכן Python מאפשרת לנו לבחור ידנית למה נרצה לעשות import.

בנוסף, אנחנו רואים ש-package מאפשר לנו ליצור היררכיה של מודולים. אפשר גם לשים package-ים בתוך package-ים וכך נוכל את הקוד שלנו בצורה שיהיה נוח לשימוש.

כאשר אנחנו עושים import רק ל-pkg, אנחנו בעצם עושים import ל-__init__.py, ולכן נוכל לשים ב-__init__.py של ה-package קוד שירוץ תמיד כשה-package יעבור import. לדוגמה, נשנה את pkg/__init__.py כך שיכיל import ל-mod:

```
pkg/__init__.py
import mod
```

כעת נריץ מחדש את ה-interpreter:

```
>>> import pkg
>>> dir(pkg)
['__builtins__', '__doc__', '__file__', '__name__', '__package__', '__path__', 'mod']
```

זו דרך שימושית לחסוך למשתמשים ב-package שלנו import-ים שחוזרים על עצמם (במקרה שלנו mod הוא מאוד שימושי ולכן נרצה לחסוך מהמשתמש לעשות import ספציפי בכל פעם ל-pkg.mod).

כמו כן, ה-import ב-__init__.py או ב-mod.py הוא יחסי לספריה בה הקבצים האלה נמצאים, ולכן המודולים ב-pkg לא צריכים לדעת איפה הם נמצאים פיסית, הם יכולים לעשות import אחד לשני ע"י שימוש ישירות בשם המודול.

from pkg import X

בדומה ל-import-ים הראשונים שראינו, גם כאן נוכל לעשות import מתוך package:

```
>>> from pkg import mod
```

ואם נרצה נוכל גם לייבא ישירות attribute של מודול מתוך ה-package:

```
>>> from pkg.mod import func2
```

Package-ים ב-sys.modules

לפני כמה סעיפים אמרנו שב-sys.modules נוכל למצוא מיפוי בין שמות מודולים למודולים. אז עכשיו כשהוספנו package-ים לכל החגיגה הזאת, איך Python יודעת להבדיל בין package למודול (נגיד במקרה של pkg שהוא מודול לבין pkg שהוא package)?

התשובה היא שאין הבדל. אחרי ש-import קורה, Package הוא בסה"כ מודול, והוא מופיע ב-sys.modules כמודול רגיל לחלוטין. ההבדל הוא שמודולים שנמצאים בתוך package-ים מקבלים את השם המלא שלהם ב-package ולא את שם המודול בלבד:

```
>>> import sys
>>> import pkg
>>> sys.modules['pkg']
<module 'pkg' from 'pkg/__init__.py'>
>>> from pkg import mod
>>> sys.modules['pkg.mod']
<module 'pkg.mod' from 'pkg/mod.pyc'>
```

אם נחפש את 'mod' ב-sys.modules נוכל לראות שאין כזה key:

```
>>> sys.modules['mod']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'mod'
```

Relative Imports

יכולת נוספת ש-package-ים מאפשרים לנו היא לבצע import בצורה יחסית ל-package שבו אנחנו נמצאים. עד עכשיו עשינו import רק לשמות מפורשים, כלומר:

```
>>> import pkg.mod
>>> from pkg import mod
```

לא משנה מה עשינו, תמיד היינו צריכים להגיד ל-Python את כל המסלול עד למה שרצינו לייבא.

כל עוד אנחנו משתמשים ב-package, אין לנו הרבה ברירה, כי אם לא נבדיל בין package-ים ומודלים אז הפסדנו את אחד הדברים שרצינו כשהתחלנו להשתמש ב-package-ים.

אבל נניח שדברים קצת מסתבכים וקיבלנו את המבנה הבא:

```
$ find pkg/
pkg/
pkg/__init__.py
pkg/sub1
pkg/sub1/__init__.py
pkg/sub1/x.py
pkg/sub1/y.py
pkg/sub2
pkg/sub2/__init__.py
pkg/sub2/y.py
pkg/sub2/x.py
```

בעצם, יש לנו כאן package שנקרא pkg ובתוכו שני תתי-package-ים שנקראים sub1 ו-sub2. בכל אחד מהם יש שני מודולים x ו-y.

אנחנו יכולים לדוגמה לעשות import כזה:

```
>>> import pkg.sub1.x
```

ונוכל להשתמש בפונקציות בתוך x של sub1. אבל מה יקרה אם נרצה לעשות import מתוך sub2.x לפונקציות מתוך sub1.x? נצטרך בלית ברירה לכתוב את זה:

```
pkg/sub2/x.py
import pkg.sub1.x
```

אבל לא נרצה ש-pkg.sub2 יידע שהוא חי בתוך pkg (אולי נשנה את pkg לשם אחר בעתיד?). לכן, נעדיף לבצע import מסוג אחר שנקרא relative-import:

```
pkg/sub2/x.py
from ..sub1 import x
```

בנוסף, נוכל לעשות relative-imports גם בין מודולים באותו package:

```
pkg/sub1/x.py
from .y import func
```

הצורה הזו עדיפה בהרבה מאשר "from y import func" כי כאשר אנחנו עושים relative-import אנחנו אומרים בדיוק מאיפה ה-import צריך להתבצע. אם לדוגמה מחקנו את המודול y אבל y קיים במקום אחר שאפשר למצוא ב-sys.path, נקבל ImportError ולא נבצע import למודול לא נכון.

בנוסף, relative-imports מגינים עלינו מפני import מחוץ ל-package בו אנחנו נמצאים (ע"י exception כמובן). ננסה לעשות import כזה:

```
pkg/sub2/y.py
from .....non_existent import non_existent
```

ונריץ:

```
>>> import pkg.sub2.y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pkg/sub2/y.py", line 1, in <module>
    from .....non_existent import non_existent
ValueError: Attempted relative import beyond toplevel package
```

חלק 8: מודולים נפוצים

הגרסאות העדכניות של Python מסופקות עם המון (ממש המון) מודולים לשימוש של המתכנתים בה. ידע והכרה של המודולים חשובים לא פחות מהיכרות עם השפה, בגלל שכמו ש-Python חוסכת זמן בתכנות ומאפשרת כתיבה של תוכניות חכמות וקצרות בהרבה, כך גם הספריות שמסופקות איתה – הן כוללות הרבה מחלקות ופונקציות סטנדרטיות, וחוסכות זמן בבניה של כלים שכבר קיימים ומוכנים לשימוש. בנוסף, שימוש בספריות קיימות הוא יתרון כי סביר להניח שעוד אנשים כבר מכירים את הספריות האלה ולכן יהיה קל יותר לתחזק קוד שתכתבו איתן, וגם די סביר שיהיו בהן פחות באגים מקוד שתכתבו בעצמכם.

פרק זה לא מכסה את כל המודולים שקיימים בשפה, אלא כמה מודולים אותם נוח ללמוד בתחילת העבודה עם מודולים חיצוניים, כי הם שימושיים ולא דורשים הרבה ידע קודם ב-Python.

מומלץ להשתמש בתיעוד באינטרנט שנמצא ב-http://docs.python.org/library/ כדי להכיר את המודולים שמוצגים כאן לעומק, וכדי להכיר מודולים שלא מתועדים בפרק הזה.

os

את os פגשנו כבר וראינו שהוא מכיל פונקציות לתקשורת עם מערכת ההפעלה. המודול לא תלוי במערכת ההפעלה עליה אנחנו רצים, ולכן אפשר לסמוך על כך שאם נכתוב את הקוד שלנו במערכת הפעלה אחת וניקח אותו למערכת הפעלה אחרת הוא ימשיך לפעול כמו שצריך.

בנוסף ל-os קיימים המודולים nt ו-posix לתקשורת עם מערכות הפעלה Windows-NT ו-UNIX-ים למיניהם ועל המודולים האלה לא נרחיב כאן.

כדי להשתמש במודול נכתוב:

```
>>> import os
```

ב-os נוכל למצוא את ה-attribute שנקרא environ. environ הוא מילון המכיל את משתני הסביבה של המערכת (כמו המיקום של ספריית ה-temp). משתני הסביבה משתנים ממערכת למערכת, אך ניתן לראות את כל המשתנים הללו ע"י קריאה ל-()keys של המילון:

```
>>> os.environ.keys()
['LANG', 'TERM', 'SHELL', 'LESSCLOSE', 'XDG_SESSION_COOKIE', 'SHLVL', 'SSH_TTY',
'PWD', 'LESSOPEN', 'SSH_CLIENT', 'LOGNAME', 'USER', 'PATH', 'MAIL', 'LS_COLORS',
'HOME', '_', 'SSH_CONNECTION']
```

בעיקר נוכל למצוא ב-os פונקציות לטיפול בתהליכים וקבצים, והן מאוד בסיסיות ולכן קשה להשתמש בהן. יש ב-Python הרבה ספריות שעוטפות את הפונקציות ב-os כדי שלא נצטרך להתעסק עם הפונקציות האלה, אך כן כדאי להכיר כמה מהן:

- os.mkdir מאפשרת לנו ליצור ספריה.
- os.rmdir מוחקת ספריה ריקה (נקבל OSError אם בספריה יש עוד קבצים).
- os.unlink היא אותה הפונקציה כמו os.remove.
- os.getcwd מחזירה את הספריה הנוכחית בה אנחנו רצים (שם לדוגמה נוצרים קבצים כשאנחנו פותחים קובץ עם file ולא מציינים ספריה ספציפית).

- os.chdir משנה את הספריה הנוכחית.

sys

המודול sys מכיל את כל הפונקציות שקשורות למערכת ה-Python ולהרצת תוכנית ה-Python.

- אובייקטי file שמייצגים את הקלט לתוכנית:
  - sys.stdin הוא קובץ שמייצג את הקלט לתוכנית. אם המשתמש עובד על terminal, הקובץ הזה יכיל את ההקלדות של המשתמש, ואם הפעילו את התוכנית שלנו עם קלט מקובץ, sys.stdin יקרא מהקובץ הזה. לדוגמה, raw_input() משתמשת ב-sys.stdin כדי לקרוא קלט.
  - sys.stdout הוא קובץ שמייצג את הפלט לתוכנית. בדומה ל-sys.stdin, זה יכול להיות טרמינל או קובץ. print מדפיסה ל-sys.stdout אם לא אמרנו לה לכתוב לאף קובץ אחר.
  - sys.stderr הוא קובץ שמאוד דומה ל-sys.stdout, אבל אליו מדפיסים שגיאות ומידע על ריצת התוכנית שהוא בד"כ יותר טכני ומסובך, כזה שלא היינו רוצים שמשתמש שאינו מתכנת יראה.
- מידע על גרסת ה-Python עליה אנחנו רצים כרגע:
  - sys.version מכיל את גרסת ה-Python בצורת מחרוזת (לא כזה שימושי, אבל יפה להדפסה).
  - sys.version_info הוא אובייקט שאפשר לקבל ממנו מידע על הגרסה שלנו.
- sys.platform מכיל מחרוזת עם שם מערכת-ההפעלה עליה אנחנו רצים.
- Attributes שכיסינו בפרקים קודמים:
  - sys.path
  - sys.modules
  - sys.argv
  - sys.exc_info()

כמו כן, sys מכיל attributes לשימושים יותר מתקדמים, כמו:

- sys.setprofile() ו-sys.getprofile() שמאפשרות להגיד ל-Python להריץ פונקציה משלנו בכל פעם שפונקציה פייתונית נקראת (זה משמש ליצירת Profiler שהוא כלי למדידת ביצועים).
- sys.getrefcount() שמאפשר לקבל את ה-reference-count לאובייקט מסוים.

os.path

במהלך החוברת התעסקנו עם קבצים. לדוגמה, פתחנו קבצים עם file() וגם עשינו import אחרי שהוספנו איברים ל-sys.path. בכולם היינו צריכים לייצר באיזושהי צורה path לקובץ, ועד עכשיו נמנענו לעסוק בנושא בצורה יסודית.

המודול os.path (שמגיע עם os, אין צורך לעשות לו import בנפרד) מכיל פונקציות לטיפול ב-path-ים לקבצים:

- os.path.sep מכיל את התו שמפריד בין שמות ספריות ב-path (ב-Unix זה "/" וב-Windows זה "\\"),
- os.path.pardir מכיל את שם הספריה שמצביעה לספריה שמעל הספריה הנוכחית,
- os.path.curdir מכיל את שם הספריה של הספריה הנוכחית:

```
>>> os.path.sep
'/'
>>> os.path.pardir
'..'
>>> os.path.curdir
'.'
```

- os.path.dirname מחזיר את שם הספריה בה קובץ מסוים נמצא, ו-os.path.basename מחזירה את שם הקובץ בלי הספריה:

```
>>> os.path.dirname('/usr/local/lib/python2.7/re.py')
'/usr/local/lib/python2.7'
>>> os.path.basename('/etc/passwd')
'passwd'
```

- os.path.exists מחזיר האם קובץ מסוים קיים או לא:

```
>>> os.path.exists('/etc/passwd')
True
>>> os.path.exists('krl2lk3rweqklwqklwqkl')
False
```

- os.path.expanduser מחליפה את התו ~ ב-homedir של המשתמש הנוכחי:

```
>>> os.path.expanduser('~/example.py')
'/home/demo/example.py'
```

- os.path.splitext מפצלת שם קובץ לשם הבסיסי שלו ולסיומת (שימו לב שהפונקציה פועלת על מחרוזת ולא מוודאת האם העברנו לה שם של ספריה או קובץ אמיתי):

```
>>> os.path.splitext('/usr/local/lib/python2.7/re.py')
('/usr/local/lib/python2.7/re', '.py')
>>> os.path.splitext('/usr/local/lib/python2.7')
('/usr/local/lib/python2', '.7')
>>> os.path.splitext('/usr/local/lib/python2.7/')
('/usr/local/lib/python2.7/', '')
```

- os.path.join היא כנראה אחת הפונקציות השימושיות ב-os.path. היא מאפשרת לנו להעביר כמה פרמטרים שנרצה, ומחברת אותם לשם של path אחד. לדוגמה:

```
>>> os.path.join('/usr', 'lib', 'python2.7', 're.py')
'/usr/lib/python2.7/re.py'
```

- os.path.realpath מחזירה את ה"ספריה האמיתית" בהינתן path מסוים. לדוגמה, הפונקציה פותחת link-ים וספריות רלטיביות (כמו ..) ומחזירה path אבסולוטי לקובץ שביקשנו. את הדוגמה הזו הרצנו מהספריה /home/demo:

```
>>> os.path.realpath('../x/../y/blah.py')
'/home/y/blah.py'
```

- os.path.isdir מחזירה האם path מסוים הוא ספריה או קובץ:

```
>>> os.path.isdir('/etc')
True
>>> os.path.isdir('/etc/passwd')
False
```

חשוב לציין שהחוזק האמיתי של os.path הוא לא רק כל פונקציה בפני עצמה, אלא שילוב של כמה פונקציות ביחד. לדוגמה, אם נרצה למצוא את הקובץ settings.json שנמצא באותה ספריה כמו המודול שלנו, פשוט נכתוב:

```
os.path.join(os.path.dirname(__file__), 'settings.json')
```

math

מודול זה מכיל מספר פונקציות שימושיות וקבועים לשימושים מתמטיים, כמו חישובים עם π או e, או משחקים משונים עם float-ים:

- math.pi ו-math.e הם שני קבועים המייצגים את π ואת e בדיוק של 11 ספרות אחרי הנקודה.
- math.ceil מעגלת מספר כלפי מעלה.
- math.floor מעגלת מספר כלפי מטה.
- math.sin, math.cos, math.tan מבצעות סינוס, קוסינוס וטנגנס. math.asin, math.acos ו-math.atan הן הפונקציות ההופכיות להן.
- math.sqrt מחזירה שורש ריבועי של מספר.
- math.log מחזירה לוגריתם של מספר.

הרעיון די ברור, ואפשר לעבור על ה-help של המודול כדי לראות את כל הפונקציות שקיימות בו.

time

לא במפתיע, המודול time מכיל פונקציות לטיפול בזמן:

- time.time() מחזירה את הזמן הנוכחי בשניות, כאשר הטיפוס שמוחזר הוא float וערך ההחזרה הוא מספר השניות מ-1.1.1970 בחצות, עד לדיוק של 9 ספרות אחרי הנקודה:

```
>>> time.time()
1347987874.865145
```

- time.sleep() היא פונקציה שישנה את כמות השניות שניתן לה (מקבלת int או float):

```
>>> time.sleep(1)
>>> time.sleep(0.1)
```

- time.localtime() מקבלת שניות מ-1.1.1970 (מעכשיו נקרא לזה epoch-time) ומחזירה אובייקט נוח לעבודה שמייצג את הזמן הנוכחי לפי ה-Timezone שמכוון במחשב שעליו אנחנו עובדים:

```
>>> time.localtime(time.time())
time.struct_time(tm_year=2012, tm_mon=9, tm_mday=18, tm_hour=20, tm_min=7, tm_sec=39,
tm_wday=1, tm_yday=262, tm_isdst=1)
```

- time.gmtime() עושה אותו הדבר, רק עבור GMT-Time:

```
>>> time.gmtime(time.time())
time.struct_time(tm_year=2012, tm_mon=9, tm_mday=18, tm_hour=17, tm_min=8, tm_sec=37,
tm_wday=1, tm_yday=262, tm_isdst=0)
```

- time.asctime() מייצרת מחרוזת להדפסה למשתמש:

```
>>> time.asctime(time.localtime())
'Tue Sep 18 20:10:14 2012'
```

- time.ctime() מקבלת זמן בשניות מאז 1.1.1970 ומחזירה מחרוזת בדומה ל-asctime():

```
>>> time.ctime(time.time())
'Tue Sep 18 20:10:14 2012'
```

datetime

בוודאי שמתם לב שהפונקציות ב-time הן קצת קשות לשימוש. הסיבה לכך היא שהפונקציות ב-time מאוד דומות לפונקציות ב-os – הן רק עוטפות פונקציות ספריה שקיימות כבר הרבה שנים ואיתן Python נבנית.

כדי לאפשר לנו לתכנת כמו בני אדם, הוסיפו ב-Python את המודול datetime שמכיל אובייקטים לטיפול בשעות ותאריכים. קודם כל, המודול מכיל את האובייקט datetime, אותו אפשר לבנות עם kwargs כמו שאנחנו אוהבים:

```
>>> datetime.datetime(year=2012, month=12, day=31, hour=23, minute=59, second=59)
datetime.datetime(2012, 12, 31, 23, 59, 59)
```

כמו כן האובייקט מכיל מתודה שמאפשרת לנו לקבל את הזמן הנוכחי:

```
>>> datetime.datetime.now()
datetime.datetime(2012, 9, 18, 20, 45, 59, 461001)
```

והוא מאפשר לנו לבצע חישובים בין אובייקטי datetime. התוצאה היא אובייקט timedelta שמייצג הפרשי זמן:

```
>>> datetime.datetime(2012, 12, 31, 23, 59, 59) - datetime.datetime.now()
datetime.timedelta(104, 11577, 758527)
>>> _.days
104
```

כלומר נשארו עוד 104 ימים וקצת עד סוף השנה.

אפשר גם להוסיף ל-datetime. לדוגמה, נחשב את התאריך בעוד 10 ימים:

```
>>> datetime.datetime.now() + datetime.timedelta(days=10)
datetime.datetime(2012, 9, 28, 20, 48, 20, 847100)
```

הממ... זה התאריך והשעה, אבל רצינו רק את התאריך. בנוסף ל-datetime קיימים גם אובייקטים רק עבור תאריך ורק עבור שעה:

```
>>> x = datetime.datetime.now() + datetime.timedelta(days=10)
>>> x.date()
datetime.date(2012, 9, 28)
>>> x.time()
datetime.time(20, 49, 20, 523190)
```

ובנוסף לכך, נוכל גם לבדוק האם זמן אחד גדול מזמן אחר:

```
>>> datetime.datetime(2012, 9, 1) > datetime.datetime.now()
False
```

random

מודול זה מכיל פונקציות ליצירה של מספרים אקראיים. הפונקציה שהכי דומה לפונקציות מקבילות מ-C היא ()random, שמחזירה מספר אקראי מסוג float בין 0.0 ל-1.0.

אבל זה לא נוח במיוחד, ולכן הוסיפו ב-Python פונקציות נוחות במיוחד:

- ()random.randrange שמחזירה מספר אקראי בתחום מסוים:

```
>>> import random
>>> random.randrange(4, 800, 9)
121
```

- ()random.choice בוחרת איבר אקראי מרשימה:

```
>>> random.choice(['a', 'b', 'c'])
'c'
```

- ()random.getrandbits מקבלת מספר x ומחזירה מספר (int או long) באורך x ביטים אקראיים:

```
>>> random.getrandbits(7)
64L
>>> random.getrandbits(1000)
931789400067504998996045000155738447875894642727270320122851002736040968767404282268720056184104406195729215380033855510626272296561601893985769122276490938789322616718684317856051748207769003753055733180398850011408802557733970324003815051362801729872167438339327330157326078112678780934259070293232 2L
```

struct

המודול struct מאפשר להרכיב ולפרק מבני נתונים בינאריים כמו header-ים של פרוטוקול. ב-struct יש לתאר את מבנה הנתונים בתור מחרוזת, ובעזרת המחרוזת הזו אפשר להרכיב (pack) או לפרק (unpack) רצף של נתונים. ה"רצף" הזה גם מיוצג בתור מחרוזת. לדוגמה, נארוז את המספר 7 בתור long:

```
 >>> struct.pack('L', 7)
'\x07\x00\x00\x00\x00\x00\x00\x00'
```

נסביר את מה שקרה כאן: המספר ב-Python הוא מספר סתמי, כלומר הוא "סתם 7". אין לו גודל, ולכן אם נרצה לשלוח את ה-7 הזה לאנשהו (למישהו אחר דרך כבל רשת, או סתם לשמור אותו לקובץ), נצטרך לקודד אותו באיזושהי צורה כך שנוכל לקרוא אותו אחר כך.

נכון, יכלנו לפתוח קובץ ולשמור בו את הטקסט 7. אבל אז מספרים היו תופסים די הרבה מקום, ולכן קידוד בינארי הוא הרבה יותר יעיל. נדגים את זה עבור int-ים:

```
>>> struct.pack('I', 7)
'\x07\x00\x00\x00'
>>> struct.pack('I', 700)
'\xbc\x02\x00\x00'
>>> struct.pack('I', 70000)
'p\x11\x01\x00'
>>> struct.pack('I', 7000000)
'\xc0\xcfj\x00'
```

שימו לב שגם את 7 וגם את 7000000 אפשר לקודד למחרוזת באורך 4 תווים. ה-x\-ים שאנחנו רואים כאן הם הדרך של Python להדפיס תווים שאין להם תו דפיס, כמו 0xc0 שמודפס בתור \xc0.

כעת נוכל לשכלל קצת את הדוגמה ולארוז מחרוזת שמכילה 4 מספרים (int-ים) ותו אחד (byte):

```
>>> struct.pack('IIIIB', 1, 2, 4500, 9, 9)
'\x01\x00\x00\x00\x02\x00\x00\x00\x94\x11\x00\x00\t\x00\x00\x00\t'
```

היתרון הוא שה-str שיוצא לנו הוא יחסית קצר, וקבוע באורך:

```
>>> len(_)
17
```

כעת, נוכל לקחת את מה שקיבלנו ולפתוח אותו בחזרה ל-tuple:

```
>>> struct.unpack('IIIIB',
'\x01\x00\x00\x00\x02\x00\x00\x00\x94\x11\x00\x00\t\x00\x00\x00\t')
(1, 2, 4500, 9, 9)
```

שימו לב שחשוב לציין איך המידע מקודד (במקרה שלנו 'IIIIB'), אחרת Python לא תוכל לדעת איך לפתוח את המחרוזת שהיא קיבלה. לדוגמה, בואו נכניס מחרוזת אחרת לגמרי ונראה מה יקרה:

```
>>> struct.unpack('17B',
'\x01\x00\x00\x00\x02\x00\x00\x00\x94\x11\x00\x00\t\x00\x00\x00\t')
(1, 0, 0, 0, 2, 0, 0, 0, 148, 17, 0, 0, 9, 0, 0, 0, 9)
```

הפעם אמרנו שהמחרוזת שלנו היא קידוד של 17 מספרים באורך בית אחד. באופן כללי התיאור של המחרוזת יכול להיות מורכב מהתווים הבאים:

- c: תו דפיס, יכול לייצג לדוגמה את האות 'a'.
- b: מספר בגודל בית אחד עם סימן.
- B: מספר בגודל בית אחד בלי סימן.
- h: מספר בגודל שני בתים עם סימן.
- H: מספר באורך שני בתים בלי סימן.
- i: מספר באורך 4 בתים עם סימן.
- I: מספר באורך 4 בתים בלי סימן.
- l (האות L קטנה): מספר באורך 8 בתים עם סימן בסביבת 64-ביט או 4 בתים ב-32-ביט.
- L: מספר באורך 8 בתים בלי סימן ב-64-ביט או 4 בתים ב-32-ביט.
- f: מספר float באורך 4 בתים.
- d: מספר float באורך 8 בתים.

argparse

המודול argparse מאפשר לנו להשתמש ב-sys.argv בצורה מאוד נוחה עבורנו ומאוד ידידותית עבור מי שישתמש בתוכנית שלנו.

נתחיל במבוא קצר: בתוכניות שמספקות ממשק ב-command-line מקובל להפריד בין שני סוגי פרמטרים שמקבלים בשורת הפקודה: אופציות (options או switches) וארגומנטים (args). ההפרדה מאוד דומה ל-*args ו-**kwargs ב-Python, עם כמה הבדלים קלים.

מקובל לתת לכל אופציה בתוכנית שם קצר שמכיל אות אחת, ושם ארוך שמכיל אותיות או מספרים מופרדים במקפים. לדוגמה, לאופציה cycles ניתן את השם הקצר c- (מינוס c) ואת השם הארוך cycles– (מינוס מינוס cycles). בדוגמה הזו אם המשתמש ירצה להעביר לתוכנית שלנו (שנקראת prog.py) את האופציה cycles, הוא יצטרך לכתוב:

```
./prog.py --cycles=7
```

או:

```
./prog.py --cycles 7
```

או:

```
./prog.py -c 7
```

ומקובל לתמוך בשני הפורמטים (כלומר לקבל גם רווח וגם =).

בנוסף, הארגומנטים הם כל הפרמטרים לתוכנית שאין להם קידומת יפה כמו c-, ואותם אוספים בתור positional-arguments. לדוגמה: 1, 2 ו-3 הם ארגומנטים לתוכנית שלנו:

```
./prog.py 1 2 3
```

כמובן שאפשר יהיה גם לשלב בין אופציות לארגומנטים:

```
./prog.py 1 2 3 -c 7
```

כעת נכיר את argparse שחוסכת מאיתנו את הצורך לתכנת את הלוגיקה הזו בכל פעם מחדש:

prog.py

```
#!/usr/bin/python
import argparse

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('source_file')
    parser.add_argument('dest_file')
    args = parser.parse_args()

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

אם נריץ את prog בלי פרמטרים נקבל:

```
~$ ./prog.py
usage: prog.py [-h] source_file dest_file
prog.py: error: too few arguments
```

כלומר prog.py מצפה לקבל שני ארגומנטים בדיוק, והיא קוראית להם source_file ו-dest_file. אם נריץ את התוכנית שלנו עם h- או עם help-- נקבל את מסך העזרה ש-argparse מייצר באופן אוטומטי:

```
$ ./prog.py -h
usage: prog.py [-h] source_file dest_file

positional arguments:
  source_file
  dest_file

optional arguments:
  -h, --help   show this help message and exit
```

ואם נואיל להעביר ל-prog.py את הארגומנטים שהיא ציפתה להם היא תמשיך לרוץ כרגיל (ובמקרה שלנו לא תעשה כלום ומיד תצא):

```
$ ./prog.py 1 2
$
```

נוכל לגעת לארגומנטים לפי השם שנתנו להם ב-args, כלומר args.source_file ו-args.dest_file.

כעת, נוכל להוסיף גם אופציות לתוכניות שלנו (נשנה רק את הפונקציה main):

prog.py

```
def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('source_file', help='Path to source file')
    parser.add_argument('dest_file', help='Path to destination file')
    parser.add_argument('-c', '--copy', help='Copy source path onto destination path')
    parser.add_argument('-m', '--move', help='Move source path to destination path')
    args = parser.parse_args()
```

קיבלנו שתי אופציות שאפשר להפעיל את התוכנית בלעדיהן, וגם הוספנו את הפרמטר help לכל קריאה ל-add_argument כדי לספק תיאור לפרמטרים של התוכנית שלנו. כעת מסך העזרה ייראה כך:

```
~$ ./prog.py --help
usage: prog.py [-h] [-c COPY] [-m MOVE] source_file dest_file

positional arguments:
  source_file           Path to source file
  dest_file             Path to destination file

optional arguments:
  -h, --help            show this help message and exit
  -c COPY, --copy COPY  Copy source path onto destination path
  -m MOVE, --move MOVE  Move source path to destination path
```

כמה אפשרויות נוספות ש-argparse נותן לנו:

- אפשר להעביר ל-add_argument פרמטר בשם type שמאפשר לנו לציין לאיזה type אנחנו מצפים. אם לדוגמה נרצה לקבל מספר נציין type=int. אם המשתמש יעביר פרמטר שאינו מספר התוכנית שלנו תציג לו שגיאה ולא ניאלץ לבדוק את זה בעצמנו.
- אפשר להעביר ל-add_argument פרמטר בשם choices שמכיל list או tuple עם אפשרויות לערכים חוקיים לפרמטר.
- נוכל להעביר פרמטר בשם default עם ערך התחלתי, למקרה שהמשתמש לא יציין אופציה כלשהי.

בנוסף, נוכל לציין עבור אופציות מסוימות פעולה כמו store_true:

prog.py

```
def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('source_file', help='Path to source file')
    parser.add_argument('dest_file', help='Path to destination file')
    parser.add_argument('-c', '--copy', help='Copy source path onto destination path',
                        default=False, action='store_true')
    parser.add_argument('-m', '--move', help='Move source path to destination path',
                        default=False, action='store_true')
    args = parser.parse_args()
```

עכשיו args.copy ו-args.move יכילו True או False בהתאם להאם המשתמש העביר copy-- או move-- לתוכנית, והמשתמש מצידו לא צריך להעביר 1 או איזשהו ערך אחרי copy-- או move--.

זהו מבוא יחסית קצר למודול, וקיימים עוד פיצ'רים יותר מתקדמים בתיעוד המלא של המודול.

subprocess

המודול subprocess מכיל אובייקט בשם Popen וכמה קבועים שמיד נראה מה תפקידם. האובייקט Popen מאפשר לנו להריץ תהליך, לתקשר איתו ולבסוף לראות האם וכיצד התהליך הסתיים.

נתחיל בדוגמה פשוטה – נריץ את הפקודה ls בתהליך חדש דרך Python:

```
>>> import subprocess
>>> result = subprocess.Popen('ls').wait()
Desktop  Documents  Downloads  git  Music  Pictures  Public  Templates  Videos
>>> result
0
```

מה שעשינו היה להריץ תהליך חדש ששורת הפקודה שלו הייתה "ls", חיכינו שהתהליך יסתיים, ושמרנו את ערך החזרה של התהליך במשתנה בשם result. שימו לב שהתהליך הדפיס את הפלט שלו למסך שבו אנחנו עובדים.

ערך החזרה של התהליך הוא 0, וזה אומר שהוא הצליח. רוב התהליכים מחזירים 0 כשהם מצליחים, למעט כמה תהליכים מיוחדים שלא נסקור כאן.

עכשיו היינו רוצים לשמור את הפלט של התוכנית שלנו במשתנה. הרי כשצריך תוכנית כנראה שלא נרצה את ערך החזרה שלה אלא דווקא את הפלט:

```
>>> proc = subprocess.Popen('ls', stdout=subprocess.PIPE)
>>> output = proc.stdout.read()
>>> proc.wait()
0
>>> output
'Desktop\nDocuments\nDownloads\ngit\nMusic\nPictures\nPublic\nTemplates\nVideos\n'
```

אובייקטי Popen מכילים שלושה attributes מאוד חשובים – stdin, stdout ו-stderr. אלה אובייקטים דמויי file שמייצגים את הקלט, הפלט ו-stream השגיאות של התהליך שהפעלנו. אם נרצה, נוכל לכתוב ל-stdin של התהליך ע"י-כך שנקרא ל-proc.stdin.write() ונוכל לקרוא את הפלט של התהליך ע"י קריאה ל-proc.stdout.read() כמו שעשינו.

שימו לב ש-proc.stdin יהיה מוגדר רק העברנו stdin=subprocess.PIPE כפרמטר. אם לא נעביר את הפרמטר PIPE התוכנית תורץ בלי קלט, ולכן לא נוכל להשתמש ב-proc.stdin.

חשוב לשים לב לכך שבד"כ נוכל להריץ תהליך ולקרוא את ה-stdout שלו בלבד, אך אם התהליך יכתוב ל-stderr ולא ל-stdout הוא עלול להתקע אם לא נקרא את הפלט שהוא כתב ל-stderr. במקרה הזה נוכל לגרום לתוכנית שלנו להתנהג כמו בטרמינל רגיל, ולכן נגיד לה לכתוב את ה-stderr ל-stdout:

```
>>> proc = subprocess.Popen('ls', stdout=subprocess.PIPE, stderr=subprocess.STDOUT)
>>> output = proc.stdout.read()
>>> proc.wait()
0
>>> output
'Desktop\nDocuments\nDownloads\ngit\nMusic\nPictures\nPublic\nTemplates\nVideos\n'
```

אך מה עושה המתודה wait()? התפקיד של wait הוא לחכות עד שהתהליך יסתיים, וכשהוא מסתיים לקבל את ערך החזרה שלו ממערכת ההפעלה. אם לא נקרא ל-wait() התהליך שלנו יישאר רץ במצב מיוחד שנקרא זומבי (zombie), כי מערכת ההפעלה שומרת עבורנו את כל התהליכים שלא אספנו את ערך החזרה שלהם עד שנאסוף אותם. לכן, תמיד נקרא ל-wait() ונבדוק את ערך החזרה של התהליך, כי הרי נהיה חייבים לוודא שהוא הצליח.

בנוסף לקבלת הפלט של תוכנית, נוכל גם לשלוח לה קלט. אם נציין ש-stdin הוא PIPE התוכנית שלנו תוכל גם לקבל קלט. הפעם נריץ את התוכנית cat שמדפיסה ל-stdout כל מה שהיא מקבלת ב-stdin:

```
>>> proc = subprocess.Popen('cat', stdin=subprocess.PIPE, stdout=subprocess.PIPE,
stderr=subprocess.STDOUT)
>>> proc.stdin.write('Blah\nBlah\nBlah\n')
>>> proc.stdin.close()
>>> proc.stdout.read()
'Blah\nBlah\nBlah\n'
>>> proc.wait()
0
```

התוכנית cat רצה עד שהיא מקבלת EOF (End-Of-File) ולכן היינו חייבים לקרוא ל-proc.stdin.close(), אחרת התוכנית לא הייתה רואה שהקלט שלה נגמר. כאשר סיימנו לתת לה קלט, קראנו את הפלט שלה וחיכינו שהיא תסתיים.

אבל כאן יש עוד בעיה – מה היה קורה אם בזמן שהיינו מזינים קלט לתוכנית שלנו היא כבר הייתה כותבת יותר מדי פלט והייתה נתקעת? הרי אמרנו שזה יכול לקרות...

בשביל מקרים שבהם אנחנו צריכים לתקשר עם התוכנית שלנו בצורה אינטראקטיבית ממש, קיימת המתודה communicate שמקבלת מחרוזת קלט ודואגת לקרוא את stdout ו-stderr בנפרד עד שהתוכנית מסתיימת. אחרי ש-communicate תחזור נוכל לקרוא ל-wait() ולקבל את ערך החזרה של התוכנית:

```
>>> proc = subprocess.Popen('cat', stdin=subprocess.PIPE, stdout=subprocess.PIPE,
stderr=subprocess.PIPE)
>>> proc.communicate('Python\nIs\nAwesome!')
('Python\nIs\nAwesome!', '')
>>> proc.wait()
0
```

כמו שאפשר לראות communicate מחזירה tuple של (stdout, stderr).

חלק 9: איטרטורים

בפרקים הראשונים פגשנו פונקציה מאוד שימושית בשם range. מיד אחרי שפגשנו אותה הצגנו גם את xrange, והסברנו את ההבדל ביניהן – range היא פונקציה שמייצרת רשימה שמכילה את כל המספרים בתחום שביקשנו. לעומת זאת, xrange מייצרת אובייקט-דמה שלא מכיל את כל האיברים אלא זוכר את 3 הפרמטרים של הסדרה (start, stop, step) ויודע להחזיר לנו בכל פעם את האיבר הבא.

ההיגיון די ברור – אין כמעט אף מקרה שבו נצטרך את כל האיברים בזיכרון ברשימה אחת. לפעמים הרשימה הזאת יכולה להיות אפילו די גדולה, ולכן לא נרצה ליצור ולזכור את כל האיברים, אלא יהיה לנו הרבה יותר מהר לחשב את האיבור במקום x כשנצטרך אותו. xrange מספקת משהו הרבה יותר פשוט והוא רק את האיבר הבא בכל פעם. מסתבר שזה מספיק טוב.

אם נחשוב קצת הלאה, רשימות של מספרים זה רק מקרה אחד שבו אנחנו מייצרים עותק של משהו בזיכרון. נניח שיש לנו מילון גדול בשם d ואנחנו קוראים ל-()d.keys. המתודה keys של המילון תחזיר לנו רשימה שמכילה את כל המפתחות של המילון. מאחר שרשימה היא mutable (כלומר אפשר לשנות אותה), הרשימה הזו היא עותק. שוב יצרנו עותק של משהו שלא היינו צריכים. הרי לא יהיה כמעט מקרה שבו נצטרך את כל המפתחות, ובטוח נוכל להסתפק באובייקט שיחזיר לנו בכל פעם את המפתח הבא (זה מספיק טוב בשביל להדפיס, בשביל לחפש, ובשביל עוד הרבה משימות אחרות).

איטרטורים וה-Iterator protocol

מאחר שנראה שהפתרון של xrange שבו יש אובייקט שמחזיר בכל פעם את האיבר הבא הוא די טוב, אז כנראה שנרצה פתרון כזה עבור כל המקרים שבהם אנחנו צריכים לעבור על קבוצת איברים אחד אחרי השני.

הפתרון הזה נקרא איטרטור (iterator), ובתור התחלה נסתכל איך עובד איטרטור במקרה של xrange ואחר כך נבין את המקרה הכללי. לצורך הדוגמה ניצור לעצמנו אובייקט xrange ונעשה לו dir:

```
>>> x = xrange(7)
>>> dir(x)
['__class__', '__delattr__', '__doc__', '__format__', '__getattribute__',
'__getitem__', '__hash__', '__init__', '__iter__', '__len__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__reversed__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__']
```

מה שאפשר לראות הוא שבאובייקט מסוג xrange אין שום מתודה ואף attribute. למעשה, כל מה שהאובייקט הזה יודע לעשות הוא לממש slot אחד מאוד מיוחד בשם __iter__, שכשהוא ממומש אנחנו יכולים לקרוא ל-()iter על האובייקט שלנו (בדיוק כמו שאם נממש את __len__ נוכל לקרוא ל-()len). נקרא ל-iter(x) ונראה מה נקבל:

```
>>> y = iter(x)
>>> dir(y)
['__class__', '__delattr__', '__doc__', '__format__', '__getattribute__', '__hash__',
'__init__', '__iter__', '__length_hint__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'next']
```

אוקי, אז עכשיו יש לנו אובייקט חדש בשם y שמכיל מתודה אחת בשם ()next. ננסה לקרוא לה:

```
>>> y.next()
0
```

קיבלנו את האיבר הראשון בסדרה.

מאחר שיצרנו את xrange עם הפרמטר 7, הסדרה אמורה להיות באורך 7 ולכלול את המספרים מ-0 עד 6, כולל 6. בואו נקרא ל-()next עד שנגיע לסוף הסדרה:

```
>>> y.next()
1
>>> y.next()
2
>>> y.next()
3
>>> y.next()
4
>>> y.next()
5
>>> y.next()
6
```

אוקי, הגענו לסוף הסדרה. עד כאן האובייקט y עושה את העבודה שלו, אבל מה יקרה אם נמשיך לקרוא ל-()next? נגיד שלא טרחנו לקרוא ל-len(x) ולבדוק שהוא שווה ל-7, או שאולי xrange בכלל לא היה מממש את __len__ ולא היינו יכולים לקרוא ל-len(x):

```
>>> y.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

כמה הגיוני – קיבלנו exception שאומר שהגענו לסוף הסדרה.

בעצם, האובייקט y הוא איטרטור של x. כאשר אנחנו קוראים ל-iter(x) נוצר אובייקט שכל מה שהוא יודע לעשות הוא להחזיר את האיבר הבא בכל קריאה ל-()next, וכשאין עוד איברים לזרוק StopIteration.

ההתנהגות הזו שבה לאובייקט יש מתודת ()next שמתנהגת כמו שתיארנו עכשיו עכשיו נקראת Iterator Protocol, וכל אובייקט שמקיים אותה יכול להיות איטרטור. למעשה, אנחנו יכולים לממש איטרטורים בעצמנו ובהמשך נראה דוגמה לכך.

iter()

אחרי שהבנו מה זה איטרטור, כדאי שנבין קצת יותר טוב איך ()iter מתנהגת בכל מיני מצבים. בתור התחלה, ניצור לעצמנו רשימה ונייצר איטרטור שלה:

```
>>> l = [1, 2, 3, 4, 5]
>>> x = iter(l)
```

דבר ראשון שאפשר לעשות עם איטרטור הוא לפתוח אותו כרשימה או tuple. לכאורה זה נראה מטופש כי הרי רצינו בדיוק את ההפך, אבל זה די שימושי אם נרצה לדבג קוד שמכיל איטרטור, כי אם ננסה להדפיס את האיטרטור לא נקבל הרבה מידע שימושי:

```
>>> x
<listiterator object at 0x1831b50>
>>> tuple(x)
(1, 2, 3, 4, 5)
```

בנוסף, נוכל ליצור שני איטרטורים (או יותר משניים, למעשה כמה שנרצה) של אובייקט מסוים. האיטרטורים יתקדמו בנפרד ולא יהיו תלויים אחד בשני:

```
>>> x = iter(l)
>>> y = iter(l)
>>> x.next()
1
>>> x.next()
2
>>> y.next()
1
>>> y.next()
2
>>> y.next()
3
>>> x.next()
3
```

כמו כן, חשוב לשים לב שכשאנחנו מסיימים להשתמש באיטרטור הוא לא "חוזר להתחלה". אם לדוגמה ניצור איטרטור על הרשימה שלנו ונמיר אותו ל-tuple:

```
>>> x = iter(l)
>>> tuple(x)
(1, 2, 3, 4, 5)
```

לא נוכל להשתמש יותר ב-x. האיטרטור הזה "גמור", ובניסוח יותר איטרטורי, הוא consumed. התופעה הזו נקרא iterator consumption, כלומר המצב בו כבר סיימנו להשתמש באיטרטור:

```
>>> x.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

בהמשך נראה מקרים בהם נצטרך להיזהר מ-consumed iterators בלי שאפילו התכוונו להשתמש באיטרטורים.

דבר נוסף שכדאי לדעת על איטרטורים, ושהוא גם חלק מה-Iterator Protocol, הוא שאיטרטור חייב תמיד להסכים לספק איטרטור על עצמו, והוא חייב להיות עצמו. מבלבל... קוד יסביר את זה הרבה יותר טוב:

```
>>> x = iter(l)
>>> x is l
False
>>> y = iter(x)
>>> x is y
True
```

או בקיצור:

```
>>> x is iter(x)
True
```

כלומר, איטרטור תמיד חייב להחזיר את עצמו כשקוראים ל-()iter עליו. כמו כן, בדיוק כמו שאיטרטור שומר על המצב שלו, גם iter(x) ישמור על המצב של האיטרטור. לכן, במקרים שבהם נחשוב שאנחנו מייצרים איטרטור של אובייקט מסוים, יכול מאוד להיות שבכלל קיבלנו איטרטור ולכן הקריאה שלנו ל-()iter לא החזירה אובייקט חדש. במצב כזה יכול להיות שנעשה בטעות consume לאיטרטור או שנקבל איטרטור שהוא כבר consumed.

בהמשך נראה איך להתמודד עם מצבים כאלה.

לסיום הסעיף הזה, חשוב שנכיר שלוש מתודות שימושיות של מילונים – ()iterkeys, ()itervalues ו-()iteritems. המתודות האלה מקבילות ל-()keys, ()values ול-()items אבל מחזירות איטרטורים. לדוגמה:

```
>>> for name, age in d.iteritems():
...     print '{} is {} years old'.format(name, age)
...
moshe is 8 years old
haim is 99 years old
david is 40 years old
```

המתודות האלה שימושיות מאוד, ותמיד נעדיף להשתמש בהן, אבל חשוב להכיר הבדל קטן אחד בין המתודות המקוריות למתודות שמממשות איטרטורים והוא שבזמן איטרציה על טיפוס נתונים כמו מילון לא ניתן להוריד או להוסיף אליו איברים. הסיבה לכך היא שהאיטרטור משתמש במבנה הנתונים הפנימי של מילון ולכן הוא מסתמך על כך שהמילון נשאר קבוע (כלומר שה-keys של משתנים). לדוגמה:

```
>>> for name, age in d.iteritems():
...     d['Itzik'] = 9
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: dictionary changed size during iteration
```

איך לולאת for עובדת

איטרטורים די מזכירים משהו שראינו בפרק הראשון – לולאת for. בת'כלס, כל העבודה של לולאת for היא לקחת איזשהו אובייקט שמכיל אובייקטים אחרים ולעבור עליהם אחד אחרי השני, ובכל פעם לאפשר לנו לעשות משהו עם האובייקט הנוכחי. ומאחר שזאת איטרציה לכל דבר, גם לולאת for בעצמה משתמשת באיטרטורים.

בכל פעם שאנחנו עושים for על משהו, for מייצרת איטרטור של האובייקט שהעברנו אליה, קוראת ל-()next של האובייקט הזה ומציבה את ערך החזרה של ()next במשתנה הלולאה שלו.

אם ננסה לדוגמה לעשות for על מספר, נקבל ש:

```
>>> for i in 5:
...  pass
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
```

כלומר, אי-אפשר לבצע איטרציה על int. את אותה שגיאה בדיוק נקבל אם ננסה לעשות את זה:

```
>>> iter(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
```

והשגיאות לא סתם זהות – for קוראת ל-()iter כשהיא מתחילה לרוץ.

בנוסף, מאחר שאנחנו יכולים ליצור איטרטור של איטרטור (וכמו שאמרנו יוחזר האיטרטור עצמו), נוכל לעשות for על איטרטור:

```
>>> l = [1, 2, 3, 4, 5]
>>> x = iter(l)
>>> for i in x:
...     print i
...
1
2
3
4
5
```

וכמובן, נוכל לשכלל את הדוגמה כדי להמחיש את ההתנהגות של איטרטורים:

```
>>> l = [1, 2, 3, 4, 5]
>>> x = iter(l)
>>> x.next()
1
>>> x.next()
2
>>> for i in x:
...     print i
...
3
4
5
```

Generator Expressions

יצור נוסף שפגשנו בפרקים הראשונים היה List Comprehensions. זה היה syntax נוח שאיפשר לנו לאחד את היכולות של ()map ו-()filter. נזכיר איך זה נראה:

```
>>> [x * 2 for x in xrange(7) if x % 3 == 0]
[0, 6, 12]
```

אבל List Comprehensions עושים בדיוק את אותו העוול של range, ולכן היינו רוצים גם "גרסת xrange" שלהם. הגרסה הזו נקראת Generator Expressions והיא נראית בדיוק אותו הדבר, רק עם סוגריים עגולים:

```
>>> (x * 2 for x in xrange(7) if x % 3 == 0)
<generator object <genexpr> at 0x18411e0>
```

האובייקט שנוצר הוא מסוג גנרטור (generator) והוא מקיים את ה-Iterator Protocol:

```
>>> y = (x * 2 for x in xrange(7) if x % 3 == 0)
>>> y.next()
0
>>> y.next()
6
>>> y.next()
12
>>> y.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

מעבר לכך שהם הרבה יותר שימושיים מבחינה תכנותית, ל-Generator Expressions יש עוד יתרון חשוב על פני List Comprehensions. לדוגמה, נניח שאנחנו דווקא כן רוצים רשימה מפורשת בזיכרון, ואנחנו רוצים ליצור אותה בעזרת List Comprehensions:

```
>>> [num / 3 for num in xrange(10)]
[0, 0, 0, 1, 1, 1, 2, 2, 2, 3]
```

קיבלנו מה שרצינו, אבל גם נוצרה לנו שארית קטנה: List Comprehensions "דולף" את משתנה הלולאה בדיוק כמו לולאת for רגילה:

```
>>> num
9
```

כלומר, כשאנחנו משתמשים ב-List Comprehensions, תמיד יקרה מצב שבו משתנה הלולאה ידלוף לתוך ה-namespace שלנו. במקרה שבו לא השתמשנו במשתנה הזה זה עוד יחסית בסדר, אבל אם לא נשים לב נוכל להגיע למצבים שבהם שימוש תמים ב-List Comprehensions ידרוס לנו את אחד מהמשתנים בפונקציה.

כדי להתמודד עם התופעה הזו נוכל להשתמש ב-Generator Expression, מאחר ש-Generator Expressions מכילים namespace משלהם ואינם דולפים את משתנה הלולאה. כל מה שנצטרך הוא לעטוף את הגנרטור שנוצר ב-tuple או ב-list:

```
>>> tuple((num / 3 for num in xrange(10)))
(0, 0, 0, 1, 1, 1, 2, 2, 2, 3)
```

כמובן נוכל להשתמש ב-Generator Expressions גם בלולאות for ובתור פרמטרים לפונקציות, כמו במקרה של tuple.

שימו לב שהרגע ראינו שכשאנחנו קוראים לפונקציה שמסכימה לקבל איזשהו iterable נוכל גם להעביר לה גנרטור. אבל במקרה שבו אנחנו יוצרים גנרטור ע"י Generator Expression יוצא שאנחנו צריכים להעביר סוגריים מיותרים שברור שאין בהם צורך. לכן, מבחינת Python אפשר להוריד את הסוגריים האלה ולכן נוכל לקרוא ל-tuple מהדוגמה הקודמת פשוט ע"י:

```
>>> tuple(num / 3 for num in xrange(10))
(0, 0, 0, 1, 1, 1, 2, 2, 2, 3)
```

שתי פונקציות שימושיות שכדאי להכיר בהקשר הזה הן any ו-all. שתיהן מקבלות איזשהו iterable ומחזירות True או False. any מחזירה True אם איזשהו איבר שהיא קיבלה לא מיתרגם ל-False (כלומר לא אחד מ-0, "", {}, (), [] או (set()) ו-all מחזירה True אם כל האיברים שבה לא מיתרגמים ל-False. לדוגמה, האם יש איזשהו מספר בין 0 ל-100 שמתחלק ב-2?

```
>>> any(x % 2 == 0 for x in xrange(100))
True
```

ברור שכן. והאם כל המספרים בין 0 ל-100 מתחלקים ב-2?

```
>>> all(x % 2 == 0 for x in xrange(100))
False
```

ברור שלא.

שימו לב ש-Python ממש לא הייתה צריכה לעבור על כל המספרים בין 0 ל-100 בשני המקרים. במקרה הראשון הגנרטור שהעברנו לה בדק את המספר 0, החזיר True ולכן any חזרה מיד אחרי הניסיון הראשון. לא היה צריך לבדוק את שאר האיברים.

במקרה של all, היא ראתה שהאיבר הראשון אכן מקיים את התנאי והמשיכה לאיבר הבא. אבל ברגע שעברנו לבדוק את האיבר הבא קיבלנו False ולכן all מיד יוצאת, כי אין לה צורך לבדוק את שאר האיברים.

המודול itertools

איטרטורים הם אובייקטים מאוד שימושיים, ולכן קיים מודול בשם itertools שמכיל פונקציות כדי שיהפכו את החיים שלנו עם איטרטורים לעוד יותר קלים. נדגים כאן כמה מהפונקציות שנמצאות בשימוש נפוץ:

itertools.chain מאפשרת לנו לשרשר כמה איטרטורים. כשאנחנו משתמשים ברשימות, אפשר פשוט לחבר שתי רשימות כדי לקבל רשימה אחת גדולה. אבל במקרה של איטרטורים, אנחנו כמובן לא רוצים לחבר אותם (כי אז נצטרך לבצע consumption) ולכן נצטרך אובייקט שיעשה בשבילנו את העבודה:

```
>>> itertools.chain(xrange(10), xrange(100, 110))
<itertools.chain object at 0x1f33190>
>>> tuple(_)
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109)
```

itertools.cycle מקבל איטרטור ומחזיר איטרטור אינסופי (כזה שה-next() שלו לא זורק StopIteration אף פעם) ע"י-כך שהוא פשוט חוזר על האיברים שוב ושוב בלולאה. שימו לב שזהו איטרטור בלאי, כי בת'כלס הוא זוכר את האיברים ברשימה. לצערנו אין דרך אחרת (נסו לחשוב על איטרטור שמחזיר שלושה איברים אקראיים. יהיה מאוד קשה לחזור עליהם אם לא נשמור אותם בצד):

```
>>> itertools.cycle(xrange(3))
<itertools.cycle object at 0x1ed8710>
>>> x = _
>>> x.next()
0
>>> x.next()
1
>>> x.next()
2
>>> x.next()
0
>>> x.next()
1
>>> x.next()
2
>>> x.next()
0
>>> x.next()
1
>>> x.next()
2
>>> x.next()
0
```

itertools.count מקבל מספר וסופר החל מהמספר הזה והלאה. גם הוא לא מסתיים אף פעם.

```
>>> x = itertools.count(17)
>>> x.next()
17
>>> x.next()
18
>>> x.next()
19
>>> x.next()
20
>>> x.next()
21
```

לאלו מאיתנו שמתקשים להירדם, נוכל לממש בקלות גנרטור שסופר כבשים:

```
>>> ('{} sheeps'.format(num) for num in itertools.count(2))
<generator object <genexpr> at 0x1f368c0>
>>> x = _
>>> x.next()
'2 sheeps'
>>> x.next()
'3 sheeps'
>>> x.next()
'4 sheeps'
>>> x.next()
'5 sheeps'
>>> x.next()
'6 sheeps'
>>> x.next()
'7 sheeps'
```

תוכלו לקרוא על שאר הפונקציות שיש ב-itertools ב-help שלו.

גנרטורים

לקינוח, נכיר פיצ'ר שימושי ביותר ב-Python. עד עכשיו ראינו תחביר נוח ל-List Comprehensions שמאפשר לנו לקבל גנרטור עם אותו syntax.

אבל, דבר נוסף ש-Python מאפשרת לנו לעשות הוא לכתוב פונקציה רגילה לחלוטין, רק שבמקום שהפונקציה שלנו תחזיר ערך אחד, היא תוכל להחזיר כמה ערכים שתרצה, והיא תהיה עם אותו ממשק כמו של איטרטור. זה נראה ככה:

```
>>> def f():
...     yield 1
...     yield 2
...     yield 3
```

f נקראת גנרטור. הסיבה שהיא נקראת כך היא שהשתמשנו במילה yield בתוך הפונקציה. ברגע שעשינו כזה דבר זאת לא פונקציה יותר אלא גנרטור. שימו לב שאם נשתמש במילה yield בתוך פונקציה וננסה לעשות return לערך נקבל מיד exceptions:

```
>>> def bad_f():
...     yield 1
...     return 2
...
  File "<stdin>", line 3
SyntaxError: 'return' with argument inside generator
```

אם נרצה, נוכל לצאת מהפונקציה ע"י return ריק, בלי שום ערך:

```
>>> def f():
...     yield 1
...     yield 2
...     yield 3
...     return
...     yield 4
```

כמובן שהפונקציה פשוט תסתיים ולעולם לא תגיד להריץ את yield 4. כעת נראה איך הפונקציה פועלת:

```
>>> x = f()
>>> x.next()
1
>>> x.next()
2
>>> x.next()
3
>>> x.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

בדיוק כמו איטרטור – כשאנחנו קוראים לפונקציה, נוצר instance חדש של הגנרטור שלנו, ובכל קריאה ל-next() הקוד של הגנרטור רץ עד שהוא מגיע ל-yield הבא. בכל yield מוחזר הערך שנתנו ל-yield מ-next() והגנרטור "נתקע" עד הקריאה הבאה ל-next().

שימו לב שבדיוק כמו במקרה של איטרטורים, גם כאן נוכל ליצור כמה instance-ים שנרצה בלי שהם יהיו תלויים אחד בשני:

```
>>> x1 = f()
>>> x2 = f()
>>> x1.next()
1
>>> x2.next()
1
>>> x2.next()
2
>>> x1.next()
2
>>> x1.next()
3
>>> x1.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> x2.next()
3
```

כמובן שאיננו מוגבלים רק לקוד פשוט. נוכל לעשות בתוך גנרטור כל דבר שהיינו עושים בתוך פונקציה רגילה:

```
>>> def greet(*names):
...     for name in names:
...         yield 'Hello {}'.format(name)
...
>>> print '\n'.join(greet('Moshe', 'David'))
Hello Moshe
Hello David
```

וכמו שראינו מקודם, נוכל גם ליצור גנרטורים שלא מסתיימים לעולם פשוט ע"י-כך שנממש בהם לולאה אינסופית:

```
>>> import itertools
>>> def forever_alone(friends):
...     for friend in friends:
...         if friend is None:
...             yield 'Forever alone'
...         else:
...             break
...
>>> x = forever_alone(itertools.cycle([None]))
>>> x.next()
'Forever alone'
>>> x.next()
'Forever alone'
>>> x.next()
'Forever alone'
```

וכדוגמה אחרונה, נראה גנרטור אינסופי שימושי שמממש סדרת פיבונאצ'י (סדרה בה כל איבר הוא הסכום של השניים הקודמים):

```
>>> def fib():
...     a = b = 1
...     while True:
...         yield a
...         a, b = b, a + b
...
>>> x = fib()
>>> x.next()
1
>>> x.next()
1
>>> x.next()
2
>>> x.next()
3
>>> x.next()
5
>>> x.next()
8
>>> x.next()
13
>>> x.next()
21
>>> x.next()
34
```