



 python TM
זוהר זילברמן



מעה הסייבר הצעה לי

www.cyber.org.il

2.7 – גרסה Python

זהר זילברמן

גרסה 1.2

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

© תשע"ד – 2014. כל הזכויות שמורות למיטה הסייבר הצה"ל.
הודפס בישראל.

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

תוכן עניינים

תוכן עניינים

4 חלק 1: מבוא

5	הפעלת Python
5	Interpreter ומאחורי הקלעים
6	משתנים והשמות (Assignments)
8	טיפוסי נתונים ואופרטורים
21.....	תנאים: if, elif, else
24.....	לולאות
27.....	range, xrange
28.....	pass

30 חלק 2: ביטויים ופונקציות

30.....	הגדרת פונקציות
30.....	None
31.....	pass בפונקציות
31.....	תיעוד
33.....	פרמטרים לפונקציות
35.....	פרמטרים אופציוניים
37.....	Conditional Expression
38.....	פרמטרים בעיתים
39.....	משתנים בפונקציות
40.....	החוות Tuple-ם מפונקציה

41 חלק 3: עיבוד מידע

41.....	map, reduce, filter, lambda
44.....	קלט מהמקלדת
44.....	List Comprehensions
45.....	קבצים
46.....	print
47.....	פירומות מחרוזות

50 חלק 4: אובייקטים

50.....	is-1 id()
51.....	type
52.....	Type Objects
53.....	Immutable-1 Mutable
54.....	dir()-1 Attributes
55.....	Reference Counting
56.....	קבצים

57 חלק 5: מחלקות

57.....	class
58.....	__init__
59.....	פנימיים Attributes
60.....	__repr__, __str__
61.....	__getitem__, __setitem__, __delitem__
63.....	ירושה

65.....	MRO
66.....	isinstance, issubclass
67.....	super
68.....	ירושה מרובה
71.....	__getattr__, __setattr__, __delattr__
72.....	__dict__
74.....	object
75.....	__slot__ים נוספים .. __Slot

76	Exceptions :6
76.....	אובייקט Exception
77.....	זריקת exception :raise
78.....	תפיסה try...except :except
80.....	ה-exception Python של
85.....	בלוק finally
86.....	בלוק else
88.....	sys.exc_info()

91	7 מודולים
91.....	import
92.....	Doc-string
93.....	NameSpace
95.....	משתנים גלובליים
97.....	global
98.....	AIR import עובד
100.....	import *
102.....	reload
102.....	סקרייפטים
105.....	פרמטרים לתוכנית פיתון
107.....	Packages

112	8 מודולים נפוצים
112.....	os
113.....	sys
113.....	os.path
115.....	math
115.....	time
116.....	datetime
117.....	random
117.....	struct
119.....	argparse
121.....	subprocess

124	9 איטרטורוֹם
124.....	איטרטורים Iterator protocol
125.....	iter()
127.....	AIR לולאת for עובדת
128.....	Generator Expressions
130.....	המודול itertools
132.....	גנרטורים

חלק 1: מבוא

שפת Scripting Python היא שפה נוחה לשימוש שהפכה למאוד פופולרית בשנים האחרונות. Python נוצרה בשנת ה-90 המוקדמות ע"י גuido ואן רוסם (Guido van Rossum) ב-Stichting Mathematisch Centrum (CWI), Corporation for National Research (CNRI), והוציאה מספר גירסאות של השפה. בשנת 1995 גuido המשיך את פיתוח השפה בחברה שנקראה ABC. במאי 2000 גuido וצוות הפיתוח של Python עברו ל-Digital Creations (DC), אשר ארגן ללא מטרות רווח. בשנת 2001 נוסד Python Software Foundation (PSF) היא אחת הממונות של PSF. כל מה הקשור לPython הוא Open Source, כולל הקוד פתוח ונגיש לכלום, וכלל אחד מותר לבצע בו שינויים ולהփיצו מחדש. אתר הבית של PSF הוא <http://www.python.org/psf/>

כאשר אומרים "שפה Scripting" מתכוונים שאין צורך בהידור (Compilation) וקיים (Linkage) של תוכניות. ב-*Python* פשוט כתובים ומריצים. כל סבירות העבודה של *Python* מגיעה עם Interpreter שמאפשר כתיבה ישירה של פקודות אלו, אףלו בלי צורך לכתוב תוכנית או קובץ. בדומה ציוו מואוד נוח להתנסות בשפה, לבצע חישובים מהירים (עוד תוכנה חזקה של *Python*) ולבדוק דברים קטנים בmahירות, בלי הצורך לכתוב תוכנית שלמה.

ל *Python* יתרונות רבים על פני שפות Scripting אחרות. ראשית, *Python* פשוטה בהרבה ללימוד, לקריאה ולכתיבה משפות אחרות (שהלן מتعلלות במתכנת ובמי צריך לקרוא קוד כתוב). *Python* גורמת לך שכתוב בה להיראות מסודר ופשוט יותר בגלל שהוא מובלט סיבוכים מיוחדים שקיימים בשפות אחרות.

שנית, *Python* מכילה בתוכה אוסף מכובד מאד של ספריות טנדריות, כולל ספריות שנitin להסתמך על קיומן בכל שימוש של *Python*. בין הכלים השימושיים ניתן למצוא יכולות התמודדות עם קבצים דחוסים (gzip), תכונות לתקשורת TCP/IP, כלים רבים לתכנות בסביבת האינטרנט, משק משתמש גרפי (GUI), ספריות לטיפול בטקסט ומחרוזות (כמו כן עיבוד טקסטואלי למיניו), עובודה עם פונקציות מתקדמות של מערכת הפעלה, תמיכה בתכונות מודולריות (Distributed Development), יכולת טיפול אינטנסיבית בקבצי XML ונתזותיהם, ועוד הרבה הרבה דברים! כמו כן *Python* מטפלת במידע בינארי (מידע שאינו טקסטואלי) בצורה הרבה יותר טובה משפות אחרות.

בנוסף, כמו שנitin לכתוב תוכניות קטנות ב-*Python*, קיימים בה כלים לכנת תוכניות גדולות, או מודלים חיצוניים. *Python* תומכת בתכונות מכון עצמים (OOP), Exceptions, מבני נתונים (Data Structures) וקישורות גם ל-C, או כל שפה שנייה לקרויה לה מ-C.

דבר נוסף שחייב לידע על *Python* הוא שיש לה מספר גרסאות. אמם יש תאימות לאחר, אבל בגרסאות החדשות (2.5 ומעלה) יש הרבה תכונות חדשות ונוחות לשימוש. בחוברת זו נלמד את גרסה 2.7.

על Python-3

אם אנחנו נלמד את *Python-2.7*, אבל חשוב לציין שהאנשים הטובים שעשו את *Python* המשיכו לפתח את השפה, וכבר ברגע כתיבת שורות אלה יוצאה לעולם גרסה 3.3 של השפה. *Python* בגרסה 3.0 ואילך שונה מהותית מגרסאות ה-X. Shallow, ולמרות שיש הרבה מושות בינהן, לא ניתן לכתוב קוד שירוץ על שתי השפות, אלא אם הוא מאוד פשוט. הסיבה שעדיין צריך ללמוד את גרסה 2.7 היא שעדין לא הומרו ממספרים ספריות ל-3.3, כך שייקחו עד הרבה חדשים עד שנראה את *Python-2* נעלמת מהעולם.

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 מגניבת ביקום שמאפשרת לעשות כל דבר بكلות ובמהירות, בלי להסתבר עם קומפיילרים ולינקרים ושמאפשרת אפילו לא לכתוב את התוכניות בקבצים אם רוצים. אז זהו, ש-C עדיין נחוצה. *Python* היא לא סתם שפה, היא סביבה עבודה שלמה, ובסביבת 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 זמן מהזמן ובשאר הזמן מחכים לקלט מהמשתמש (ואז אין סיכוי שהמשתמש ישים לב שהוא *Python* או C), אבל אם נרצה לכתוב תוכנית שמבצעת הרבה חישובים נגלה ש-*Python* לעומת C זה כמו חמור ז肯 ועייף לעומת סוס מירוצים צער ושם.

הפעלת Python

לפני שנמשיך, וודאו שאתם יודעים איך להפעיל את *Python*:

כיוון, *Python* מופצת יחד עם כל ה-*Linux*-Distribution'ים הפופולריים של *Ubuntu*, *Redhat* Linux (Ubuntu, Redhat) או כל מי שMRI'ץ לינוקס בבית לא צריך להוריד כלום. משתמשי Windows צריכים להוריד סביבות פיתוח של Python ל-Windows היישר מ-[.http://www.python.org/](http://www.python.org/).

שימוש לב-*Ubuntu* החל מגרסה 12.10 ניתן עם Python 3.0, וההבדלים בין פיתון 2.7 ל-3.0 הם גדולים, לכן וודאו שגםם מתקיימים את החבילה `python2.7` ומሪיצים מפורשת את הפקודה `.python2.7`.

כדי להפעיל את *Python* בלינוקס, פשוט מקlidים את הפקודה `python` ב-*console* ומקבלים את ה-*prompt* של

```
$ python
Python 2.7.3 (default, Apr 20 2012, 22:39:59)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

כדי להפעיל את *Python* ב-*Windows*, נטרף בדומה להריץ את `python.exe` מ-*C:\Python27*.
במהלך כל החוברת, הדוגמאות שיירשמו יילקחו מסביבת העבודה בלינוקס, אבל אין שום הבדל בין שתי סביבות העבודה.

ה-Interpreter ומחרורי הקלעים

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

```
>>>
```

ב-prompt זהה ניתן למעשה לכתוב כל פקודת *Python*, והוא 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 מרגש. אמרנו של-*Python* יש יכולות חישוב מהירות. בואו ננסה משהו קצת יותר מסובך:

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

את התוצאה כבר לא נרשם כאן... אם הרצת את השורה الأخيرة על המחשב שלך, נראה לך כמה שניות להציג לתוכה, אבל הוא כתוב על המסך ממש הרבה מספרים. מה שעשינו בשורה الأخيرة היה:

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

וכמו שבודאי הבנת, 2^{∞} זה חזקה.

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

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

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

הדבר האחרון שכדי להכיר בהקשר של ה-Interpreter (_Interpreter) ב-underscore (_). סימן ה- underscore מציין את התוצאה לאחרונה שחוושבה ע"יInterpreter ה-_. כך למשל:

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

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

משתנים והשמות (Assignments)

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

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

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

בדוגמה זאת נוצר משתנה בשם x, וערך ההתחלתי הוא 5. כיוון ש-5 הוא מספרשלם (יש לשים לב להבדל בין 5 ל-5.0), כמו ב-C (סוג המשתנה הוא `.int`).

את סוג המשתנה מקבלים באמצעות הפונקציה המובנית `type`, המתקבל בסוגרים את המשתנה ומוחזירה את הסוג שלו. את נשא ה"סוג" של משתנים ומה בדיק זה אומר נסקור בהמשך.

כאשר מציבים למשנה ערך חדש, גם סוג המשתנה מתעדכן בהתאם. אם x היה int עד עכשו, ונציב לתוכו ערך חדש, סוג המשתנה ישנה בהתאם לערך החדש:

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

ועכשיו הסוג של x הוא float.

איולוצי שמות המשתנים עובדים כמו ב-C (שם המשתנה חייב להתחיל באות או קו-הח��ן ויכול להכיל אותיות, מספרים וקוויים תחתוניים):

```
>>> 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
```

הפקודה print, בנוסף לפקודות אחרות (יש להבדיל בין פקודות לבין פונקציות) מופיעה ללא סוגרים. זאת משום שגם פקודה ולא פונקציה. ההבדל בין פונקציה לפקודה הוא שפקודה היא כל מילת מפתח של Python שאיתה כתבים את בקרת הזרימה של התוכנית (הכוונה לפקודות if, print, while, וכו'). פונקציה היא אוסף של פקודות, פונקציות אחרות ופונקציות מובנות שאותן אספנו ביחד כדי ליצור פעולה בסיסית.

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

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

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

cut יש רשימה בשם l (האות L קטנה) שמכילה את המספרים 1, 2, 3. האינדקס של האיבר הראשון הוא אפס, ונינו 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 עיטה הסימן =

המשתנים ב-**Python** הם לא יותר מאשר מבני נתונים – כל משתנה הוא מבני נתונים לאובייקט. לכל אובייקט יש סוג (אובייקט של מספר, אובייקט של מחרוזת, ועוד). משום שימושו כמבנה נתונים נוהג לשלוט "ערכים" של משתנים – רק מושנים את הערכים אליהם המשתנה מצביע, והוא "ערך" שלו המשתנה.

כאשר משתנה מצביע לעצם, וויצרים השמה למשתנה אחר ($y = x$), 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]
```

במהלך נראה אילו דברים טובים אפשר לעשות בעזרת הצבאות האלה, וגם אילו בעיות הן יכולות ליזור אם לא משתמשים לשימה הקיימת. כאשר שינוינו את הרשימה הקיימת, כל מי שהביע אליה הושפע מכך. כפי שניתן לראות, יצרנו רשימה במשתנה `x`, והשכנו אותה למשתנה `u`. כתוצאה מכך, לא נוצרה רשימה חדשה, אלא הצבעה בהן כמו שצירף.

טיפול נתוניים ואופרטורים

מספרים שלמים

אוначילהלמספריםשלמים, הטיפוס הפשטוט ביוותר. הטיפוס של מספר שלם הוא `int` (קייזר של Integer) והם מאוד דומים למספרים שלמים ב-C. פיריטון מייצרת מספר שלם כשאנחנו כתובים רק ספרות בלי שום תוספות (שבר עשרוני, או אותיות בעלות משמעות, אותן נראה בהמשך):

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

קיבלו את המשתנה x מסוג int.

לפעמים. כשהנסה לאיור מספרים מאוד גדולים. נקבע את המספר שרצינו. אבל יהיה לנו בסופו:

הסיבה לכך היא ש-int הוא טיפוס עם מגבלה – הוא יכול להחזיק מספרים רק עד גבול מסוים (2,147,483,647 בסביבת 32-בית או 9,223,372,036,854,775,807 ב-64-בית). כאשר超出了 הגבול הזה, Python צריכה להחזיר את המספר שלנו באורח קצת אחרית, הזרה הזאת נקראת `long` וזה גם שם ה-type:

```
>>> 100000000000000000000000000000000000000000000000000L  
100000000000000000000000000000000000000000000000000000000L  
>>> type(_)  
<type 'long'>
```

כמו כן, אם כבר קיים משתנה שלם ומונחים לחשב באמצעותו ערך שחרוג מהתחום של משתנה שלם, הוא יומר אוטומטית למשתנה ארוור (שימו לב ל-L בסוף המספר האחרון):

```
>>> 2 ** 62  
4611686018427387904  
>>> _ * 2  
9223372036854775808L
```

למעשה אין באמת מגבלה על גודל מספר בפיתון, מכיוון שבמידת הצורך מופיע מחרה ל-long so שיכל לתמוך במספר כל גודל שהוא. ההבדל בין long ל-int לא משנה לנו אלא אם נרצה לעשות שימושים מאוד ספציפיים במספרים, אבל זה יהיה בעתיד הרחוק איז לבינתיים מספר שלם הוא פשוט מסטר שלם.

למרות שאין בכר צורך אמיתי, אפשר ליציר משטנה אරוך כמו שיזכרים משטנה שלם רגיל. ההבדל היחיד הוא הוספה האות **L** אחרי המספר (**L** עברו Long). אין משמעות להוספה **L** קטנה או גדולה, אבל מומלץ בטור הרגל כתוב תמיד **L** גדולה – הרבה יותר ברור שהכוונה ליצירת מספר ארוך, ולא הספרה אחת (1).

```
>>> 2L  
2L  
>>> type(_)  
<type 'long'>
```

וכך יצרנו משטנה ארוך.

מספרים עשרוניים

אchio החרוג של המספר השלם הוא המספר העשרוני. ייצוג המספר העשרוני שונה מאשר ממספר שלם, וכך חלות עליו מגבלות מסוימות. גם למספר עשרוני יש גבולות (הגבלות תלויות במימושו של המספר העשרוני – האם זה מספר עשרוני של 32-ביט או של 64-ביט). אבל אין לו טיפוס מקביל "ארור" כמו למספר השלם.

יצירת משנה עשרוני היא כמו ב-C, ע"י כתיבת מספר עם נקודה עשרונית ושבר (אפשר שבר 0, רק כדי להציג 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
```

כasher shelbilim bchishuv mafkar aror vmasper ushroni yekol lehiozur makra bo htotzaah gedola mudi kdi lihiot miyetzat bechora ushroni. Am nitnu lchabat htotzaah velaachan anotta, al tihya shem beuya;

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

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

```
>>> (2L**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
```

חשיבותם של פערם במספרים עשרוניים מבדים את הספרות הת ת |וניות של המספר ב- ל שנו^רצתה אף שגיאה:

ה-א זאת דרך קצרה להגיאד "10 בחזקת" (במקרה זהה 10 בחזקת 50).

הסיבה להתנגדות זו היא שמספר עשרוני מוחזק בצורה שבה יהיה עלוי חישובים, אבל הוא לעיתים 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*

בסיסי ספירה ב-**Python** נכתבים ומתחנגים כמו ב-C: כאשר רוצים ליצג מספר בסיס 16 (Hexadecimal), או בקיצור Hex) משתמשים בקידומת 0x:

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

Python מבון תציג את המספר ביצוג עשרוני, וזהו דרך מזוהה להמיר מבסיס לבסיס ממש מהר. אפשר לעשות גם את הפעולה הפוכה, בעזרת הפונקציה המובנית (`hex`) שמקבלת מספר ומוחזירה מחרוזת המכילה את המספר ביצוג הקסדצימלי:

```
>>> 0xBEE  
3054  
>>> hex(_  
'0xbeef'
```

יצוג מספרים בבסיס 8 (Octal) נעשה ע"י הוספה אפס לפני המספר אותו רוצים ליצג (כמו ב-C) או ע"י הוספה 00 (אפס ואז האות 0):

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

כמו הנקודות (hex). קיימת פונקציית מובנית בשם (`oct`) שמחזירה מספר דואלי לערך מסוים אוקטלי:

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

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

* אם איןכם מוכרים בסיסי ספרה, תוכלו לדלג על הטעיף או לקרוא עליו Wikipedia-ב-/<http://he.wikipedia.org> (חפשו את הערך "בסיסי ספרה" ב-/<http://he.wikipedia.org>) וומלץ לקרוא ב-Wikipedia-ב-/<http://he.wikipedia.org>. אל מילון, זה גורם מוגני!

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

פעולות חשבון על מספרים

חיבור (+), חיסור (-) וכפל (*) פועלם כמו בשפת C. כמובן, כאשר יש פעולה בין 2 אופרנדים זהים, התוצאה היא מהסוג של האופרנדים. כאשר האופרנדים לא זהים, התוצאה תהיה מהטיפוס "יותר מתקדם": בכל פעולה בה מעורב מספר עשרוני התוצאה תהיה עשרונית. אם בפעולה לא מעורב משתנה עשרוני, אבל מעורב משתנה אחר, התוצאה תהיה ארוכה. חילוק (/) גם פועל כמו ב-C, וכך שטיפוס נתונים שונים שימושיים בחישוב, הכלל לגבי חיבור חיסור וכפל קבוע גם כן. כאשר יש חילוק בין שלמים, התוצאה תמיד שלמה. לעומת זאת, אם רוצים תוצאה עשרונית, חייב להיות מעורב מספר עשרוני.

אם רוצים לאlez את Python ליצר תוצאה עשרונית, אבל משתמשים רק במשתנים שלמים, ניתן להשתמש בפונקציה float() כדי להמיר את אחד המשתנים השלמים למספר עשרוני, ועל-ידי כך לארום 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 y/x מחושב, ורק אז הוא מומר למשתנה עשרוני.

מודולו (%) לא מתנהג כמו בשפת C, הוא קצר יותר משוכלל. אופרטור המודולו יכול לקבל מספרים שלמים ורגילים, ארוכים ו גם משתנים עשרוניים. התוצאה במקיריים 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, והטיפה הננה בצורה מס' 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 השנייה חושב המודולו ההפוך (והטיפה בהתאם בהתאם).

דוגמה השלישייה לעומת זאת, חושבה השארית של 2.5 לחלק 3.5-6.

בוליאניים

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

לדוגמה:

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

ניסינו לבדוק האם 5 שווה ל-6 וקיבliśmy את התשובה "לא". האופרטור == בדוגמה שלנו משווה בין שני מספרים ומחזיר True או False בהתאם לערכי המספרים. את == נפגש הרבה בהמשך ונשווה באמצעותם גם דברים שאינם מספרים. הסיבה שבבוליאנים דומים למספרים היא שפיתון יודעת לתרגם, במידת הצורך, בין בוליאנים למספרים שלמים רגילים. מתרגם ל-True, False מתרגם ל-0, והתרגומים מתבצע רק כאשר מנסים לבצע פעולות חשבון 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 الأخيرة 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 נתוני מובנה, ותו הוא מחרצת באורך 1. ייצוג זה מקל מאד על העבודה עם מחרוזות – מה שמצריך ב-C שימוש בפונקציות של הספרייה `<string.h>` מצרייך כאן שורת קוד אחת קצרה.

יצירת מחרוזת תיועשה באמצעות המחרוזות הבאות:

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

כמו שניתן לראות, מחרוזות ניתנת ליצור בשתי דרכים: המחרוזת נמצאת בין 2 תווים של גרשימים או בין 2 תווים של גרש בודד. כמובן, אם מחרוזת נמצאת בין 2 תווים של גרשימים, ניתן לכלול בה גרש בודד ('בלוי'), ולהפוך. 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, '\t' לתחילת השורה, '\b' לחזרתו אחד אחרת, '\t' עברו טאב, וכו'...

כמו כן, ב-**Python** יש מגנון מובנה לפירמות (Formatting) של מחרוזות (מה שעשו `printf()`). זה נעשה ע"י כתיבת כל ה-% המוכרים כבר מ-C לתוכן 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 מعتبرים Tuple (הטיפוס יוצג בהמשך פרק זה) של כל הפרמטרים, לפי הסדר. אם מعتبرים מספר לא מדובר של פרמטרים (למשל, רשנו רק % בחרוזת, והעברנו שני פרמטרים), יש שגיאה. כמו כן, אם מعتبرים פרמטר שהוא לא מהסוג שהעבכנו ב-% (מחרוזת עברו %, וכו'), גם יש שגיאה.

פירמות המחרוזות של **Python** הרבה יותר מוגן ונוח מזה של C, הוא לא נדרש לקרוא לפונקציה, ונינתן לשלב אותו בכל מקום, אפילו בתוך קראיה לפונקציה (בפרמטר, במקרה סתם מחרוזת רגילה, מכנים מחרוזת מפורמתת):
אין צורך להбелת מהדוגמה, פונקציית ייצגו בפרק הבא...

```
>>> def func(s):
...     print s * 5
...
>>> func("yanti%s" % ("parazi"))
yantiparaziyantiparaziyantiparaziyantiparaziyantiparazi
```

עת נסקר כמה פונקציות שימושיות מאד. חלק מפונקציות אלה מובנות ב-**Python** עצמה, וולכן של טיפוס המחרוזת: `()` מקבלת כקלט מספר המציג את ערך ה-ASCII של TWO מסויים, ומחזירה מחרוזת ובה TWO אחד שערך ה-ASCII שלו הוא הערך שקיבלה הפונקציה כפרמטר:

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

`str()` לעומת זאת מקבלת משתנה ומנסה להמיר אותו למחרוזת. הפונקציה מסוגלת לקבל יותר מטיפוסים בסיסיים, אפילו רשימות ומלונים (נראית 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.

הfonקציה `split()` מקבלת מחוץ ותו ומפרצלת את המחרצת לשימה, כאשר התו שהוא קיבל מפרד בין האיברים ברשימת הפלט. לדוגמה:

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

הfonקציה `join()`, לעומת זאת, עשויה לבדוק ההפק – היא מקבלת רשימה, ומופעלת על מחוץ. הפלט הוא מחרצת המורכבת מכל איברי הרשימה, ומהחרצת הנ吐ונה בינהם:

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

דבר שלא ניתן לעשות עם מחרצות הוא לשנות את ערכו של תו בודד. הסיבה לכך נועזה בימוש של Python, שכן שכך ציינו בנווי מצביים. מחרצת היא מצביע לעצם שמכיל את המחרצת. הביטוי הבא:

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

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

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

בנוסף, קיימות במחרצות פונקציות לשינוי ה-`hosition`. `Capitalization` של מחרצות, כולל כל נושא האותיות הקטנות והגדלות באנגלית. הפונקציה `capitalize()` מקבלת מחרצת ומחזירה את אותה המחרצת, כאשר האות הראשונה במילה הראשונה מהחליה באות גדולה והשאר קטנות.

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

מסירה מהחרצת את תוכי ה-`whitespace` שנמצאים בקצוות המחרצת. למשל, הנה המחרצת הבאה אחרי `strip`

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

הfonקציה `index()` תת-מחרצת כפרטט, ומחזירה את המיקום הראשון בו תת-המחרצת מופיעה בתוך המחרצת עליה הרצינו את `index`. אם לא נמצא מקום צזה, הפונקציה זורקת שגיאה (ييلמד בהמשך בפרק על Exceptions). הפונקציה `find()` עושה בדיקת אותו הדבר, אבל אם היא לא מוצאת את תת-המחרצת היא מחזירה -1:

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

מחרצות Unicode

דבר נוסף שנתרטט ע"י Python הוא מחרצות Unicode – מחרצות הן מחרצות בהן כל תו מיוצג ע"י שני בתים, וכך ניתן לייצג יותר תווים בטיפוס אחד. תמיינה ב-Unicode חשובה לדוגמה כדי לאפשר לנו לעבוד עם שפות כמו עברית, וב-

(אותה אנחנו לא לומדים כאן) קיימת תמייה נרחבת בהרבה לטקסט שמקודד ב-Unicode. אנחנו לא נלמד על Unicode, אך תוכל לקרוא עוד על הנושא ב-`help` של `str.decode` ו-`str.encode`.

מחרוזת Unicode נוצרת בבדיקה כמו מהירות רגילה, רק שלפנוי 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, ונitinן לעבור על כל אחד מהאיברים שברשימה. ב-חומרה Python, רשימה היא טיפוס מובנה, ובנוסף, רשימה אחת יכולה להחזיק איזה טיפוס שנרצה, ואפילו כמה סוגים טיפוסים בבת-אחת.

יצירת רשימה נעשית ע"י כתיבת האיברים שלה בין סוגרים מרובעים:

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

זאת הרשימה שמכילה את המספרים 1 עד 5. עכשו 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]
```

והשנייה היא להשתמש בפונקציה `append()` הפעלה על רשימה:

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

הדרך הראשונה יכולה לגרום להקצאה של רשימה חדשה – וזה בגלל יכול להיות שנוצרת רשימה חדשה אליה מוכנסים האיברים של שתי הרשימות המקוריות, ובמה שמתמשים מאותו הרגע. הדרך השנייה מבטיחה שלא תהיה יצרה 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 קיבל את אורך הרשימה באמצעות (`len`):

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

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

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

המספר הראשון (לפני הנקודות) הוא האינדקס ממנו מתחילה החיתוך, והמספר השני הוא האינדקס שבו יסת内幕 החיתוך, אבל הוא בעצמו לא יוכל ברשותה הנוצרת.

מטעמי נוחות, כאשר רוצים להעתיק קטע רשותה שמתחליל מיד בהתחלה שלו, או מסתיים מיד בסופה, אין צורך לציין את האינדקס הראשון (תמיד 0) או את האינדקס האחרון (תמיד אורך הרשותה), וניתן לה שימושו אותו. לדוגמה, הנה רשותה עד אינדקס 4:

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

והנה רשותה 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 זו אינדקס ההתחלה היה 0 (אנו יכולים לדלג עליו כמו בדוגמה הקודמת), האינדקס האחרון היה סוף הרשותה (גם עליו נוכל לדלג) והפרמטר האחרון מציין ערך להגדיל את האינדקס בכל העתקה.

נוכל לשככל את הקפיצה ולציין גם קפיצה שלילית, מה שיגרום ל-*Python* ללקת אחרת:

```
>>> [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 ברשותה מבלי לדעת את האורך שלה:

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

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

דוגמה חשובה ל-*slicing* היא המקרה שבו משמשים גם את התחלת הרשותה וגם את סופה. במקרה זה נקבל את כל הרשותה:

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

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

כמו שכבר ציינו קודם, ב-`Python` משתנים הם בסך הכל מצביים, כלומר שם של משתנה הוא מצביע לטיפוס מסוים. כאשר משתנה מצביע לטיפוס כלשהו – בין אם זה טיפוס מובנה או טיפוס שנוצר ע"י המשתמש (בmarshmallow הכוורת נראת כיצד עושים את זה), הוא מצביע לכטובת מסוימת בזיכרון. כאשר מבצעים השמה (עושים =) בין שני משתנים, והמשתנים הם לא מטיפוס "פשוט" (לא מחרזת ולא `Tuple` שמייד נפցוש), ההשמה מעשית מעתיקת את המצביע ממשתנה אחד למשתנה الآخر.

בדוגמה הבאה ניצור רשימה בשם `x`, נשים אותה במשתנה `y`, ונוסף 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 תכמה חזקה מאוד של `Python` – אין בה צורך במצבים, כיון שאין בה משהו אחר במצבים. 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.

תכמה זו מאוד נוחה, אבל לעיתים נרצה ליצור עותק של משתנה קיים – מה נעשה אז? השיטה הכי טובה להשתמש ב-`Slicing` אותו לדנו קודם, ובו ראיינו שכאשר ערים `Slicing` שכולל את כל האיברים (בלי ציון התחלת ובלי ציון סוף), נוצר עותק חדש של העצם המקורי. כתת נראה איך תבוצע ההשמה עם `Slicing`:

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

יש לציין שה-`Slicing` יעבד 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 שאורךה לא משתנה. כמו כן, לא ניתן לשנות את האיברים שה-Tuple מצביע אליהם.

יצירת `Tuple` נעשית בדיקות כמו יצירת רשימה, רק שבמקרים מסוימים משתמשים בסוגרים עגולות:

* העתקה כזו של רשימה לרשימה חדשה נקראת shallow copy.

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

קיבלה איבר מה-Tuple היא כמו איבר מרשימה – ע"י סוגרים מרובעים:

```
>>> 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 נכתב פשטן סוגרים בלי שום דבר ביןיה:

```
>>> 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 :Tuple על

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

ובדומה לרשימה, השם Tuple לא תיזור עותק שלו אלא תعبיר מצבי 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 الأخيرة 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 ההתחלתיים 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 الأخيرة פשוט החליפה בין הערכים של x ו-y, ללא צורך במשתנה זמני. שימוש לב גם שהורדנו את הסוגרים, כי Python מאפשר להשmidt סוגרים בכל מקום שבו אין דו-משמעות. השימוש העיקרי בtuple הוא Tuple Assignments, כמו כן, בהמשך החוברת נראה שפונקציה יכולה להחזיר רק איבר אחד. מושם Tuple טיפוס, ניתן להחזיר מפונקציה Tuple המכיל מספר איברים, וע"י כך להחזיר יותר מאיבר אחד מפונקציה.

מילון

מילון (Dictionary) הוא טיפוס שמטרתו היא המראה בין מפתחות לבין ערכים (ב- C++ משתמשים ב-map לכך). מילון למעשה מחזק ערכים, מפתחות ואת המיפוי ביניהם. כאשר המשתמש פונה למילון, הוא מצין מפתח, ומתקבל עבورو ערך. היתרון בכך הוא שבמקום מספרים אינדקסים (שלא ממש אומרים שהוא למשהו), ניתן להשתמש בכל ערך שרק רצים, אՓילו מחרוזות.

יצירת מילון נעשית כמו רישמה, אבל עם סוגרים מסולסלים:

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

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

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

שימוש לב-sh- Python מדפיס את המילון שלנו בצורה {'1': 'Sunday'}, ונוכל לבנות מילון בדיק באותה צורה בעצמו מבלי להזין את האיברים אחד אחרי השני:

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

בהמשך נשתמש בשיטה זו כדי ליצור מילונים כחלק מהקוד שלנו.

עכשו, כשהamilon d יודע שהוא "ערק" של 1 הוא "Sunday", יוכל לבקש ערך עבור מפתח מסוים כמו ברישמה:

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

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

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

בעת פניה למילון נוכל לציין מיד 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 ?:

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

היא פונקציה שמחזירה את רשימת המפתחות במילון:

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

היא פונקציה שמחזירה את רשימת הערךים 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 ישמר אותם בסדר פנימי ולא ידוע משלו. לכן, אי-אפשר להמיר בין מילון לשימה או Tuple, בעוד שכן אפשר להמיר בין Tuple לשימה לדוגמה.

לבסוף, נוכל גם לקבל מהמילון רשימת tuples-ים, כאשר כל tuple זוג של (key, value). בדוגמה ערי הבירה שלנו:

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

set

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

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

קובוצה נוכל להוסיף איברים, ואם ננסה להוסיף איבר שכבר קיים 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 לקובוצה 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 השנייה להסיר איבר היא לבקש מהקובוצה 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 כפיליות:

```
>>> 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 כМОבן פקודת ההטעינה (Conditioning), כמו ברוב שפות התכנות. מבנה הפקודה דומה לזה שבשפת C מלבד העובדהαι:

שאין חובה בסוגרים. דוגמה די פשוטה ל-if:

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

בשורה הראשונה יצרנו משתנה בשם `num` שערך 5. השורה השנייה היא שורת `if` עצמה, ובננה התחנכה דומה לה שבדיקת `C` (מיד נראה מספר הבדלים). בסוף השורה השנייה יש נקודות.תו ה-`:` ב-`Python` מורה על פתיחת בלוק חדש, בדיקת `num` הסוגרים-המסולסלות ב-`C`.

בלוקים

ואיך `Python` יודעתמתי בלוק נגמר? את זה היא עשוה באמצעות הרווחים ששמנו בתחילת השורה בבלוק `if`. בדוגמה לעליה, השורה השלישייה מוחצת ימינה ב-4 רווחים, ולא סתם בשביל שהיא כל יותר לקרוא הקוד (כמו בשפות אחרות). הוספה הרווחים לפני כל שורה בקוד `Python` משיכת אותה לבלוק מסוים.

השיטה הזו נקראת אינדנטציה (Indentation), והיא אומרת שבлок שתלויה בבלוק קודם ייכתב "פנימה" יותר בקוד. ב-`Python` אינדנטציה היא חלק מהמבנה שלו, ולכן לא ניתן לכתוב בлок חדש בלי להוסיף רווח לפני כל שורה בבלוק. רווח הוא בסה"כ כמוות רווחים מסוימת, וכדי ש-`Python` לא תתבלבל אנחנו צריכים להקפיד על רווח אחד. לא נוכל לכתוב בлок שבו חלק מהשורות יהיו עם רווח של 3 רווחים וחלק אחר עם 4 רווחים. אם נעשה כך דבר `Python` לא יוכל לדעת איפה שורה שייכת לאיזה בלוק, ולכן علينا להקפיד שכל שורה תתחיל עם מספר רווחים זהה שמתאים לבlok בה היא נמצאת.

אפשר גם להשתמש בתוו TAB עצמו, אבל אז `Python` תהפוך כל TAB ל-8 רווחים. שימושTAB בטעות ולבוד בעורך קוד שמצויג TAB-ים כ-4 רווחים, כי אז הקוד שאתם רואים שונה מהקוד שאתם מרצים. באופן כללי, העדריפו לא להשתמש ב-TAB-ים מאחר שהם מייצרים הבלבול בין אנשים שונים שנוהגים להציג את TAB בתו כמות רווחים שונה. מקובל בעולם לעבוד עם אינדנטציה של 4 רווחים רגילים, וכך גם אנחנו עושים.

and, or, not

בנוסף לתנאי `if` רגילים, אפשר ליצור גם תנאים לוגיים. ב-`Python` קיימות מילوت 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 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 בLOCk 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 הוסיף 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 ווד נראה בהמשך, אחרי `if` שמנוע כל מיני תנאים. לעיתים אלה היו תנאים עם `==`, לעיתים עם `>`, וכמובן בעודו בעודו נוכל גם לשים משתנים בפני עצם, 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 שהוא פוגשת ל-`True` או `False`. כמו שראינו מוקדם, נוכל לרשום את התנאים שבאים אחרי `if` ב-`interpreter` ולקבל את הערך שלהם. לדוגמה:

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

כשאנחנו רושמים משתנה בפני עצמו, Python פשוט ממיר את המשתנה או האובייקט לبولיאני ע"י-כך שהוא בודקת האם אותו אובייקט נחשב מבחינתי ל-`False` או `True`. אובייקטים שמורמים אוטומטית ל-`False` הם `0, 0.0, None`, רשיימה ריקה, `tuple` ריק, מילון ריק, `set` ריק, `None` ומהריזת 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

לפנינו שנסקרו את פקודת `for`, נגידר מהו **רצף** (Sequence) – רצף הוא עצם שמכיל בתוכו איברים, בסדר מסוים, כאשר העצם מכיל פונקציות לקבלת כל-אחד מהאיברים (או פונקציות לקבלת איבר ראשון וアイבר עוקב לאיבר קיומ). דוגמאות לרצפים שאנו חזו כבר מכיריהם ב-`Python` הן רשימות, `Tuple`ים, מחרוזות ומילונים.

פקודת הלולאה `for` שונה מזו שב-C. נחזור על זה: פקודת הלולאה `for` שונה מזו שב-C. במילים אחרות לאמרי – לולאת `for` של Python זה 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 המקבילות האלה. `for` ב-`Python` לא מבצעת לולאה על תחום ערכאים (כמו בפסקל) וגם לא מתחדשת בתוכנה משפט-אתחול, תנאי ופקודה לביצוע בכל איטרציה (כמו ב-C). אם אתם לא מכירם אף אחת מהשפות האלה, הכי טוב. ככה תוכלו ללמידה מהי לולאת `for` כמו שהטבע התכוון אליה.

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

דוגמאות פשוטה ל-`for`:

```

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

```

בדוגמה, הלולאה יוצרה 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, וערך יהיה הערך האחרון שקיבל במהלך ריצת הלולאה.

אם מריםים לולאת `for` עם רשימה ריקה, כלום לא יקרה והמשנה `day` (או כל משתנה אחר שייכתב שם) לא ייווצר. יש לשים לב שאם מריםים כמה לולאות, אחת אחרי השנייה, עם אותו משתנה (גיאז `i`), ובאחת הלולאות יש רשימה ריקה, אין הדבר אומר שהמשנה `i` יוושם. זה רק אומר שהוא לא ישנה עקב הלולאה זו. אם הלולאה הזו היא הלולאה הראשונה, נאמר שהיא `for` `i` `in` `[]`:

```
>>> 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 **מילוניים** ומחזורות. להלן שתי דוגמאות להmphשת העבודה עם מחרוזות ומילוניים:

```

>>> 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 מיליוןם – בהגדרת המילון לא נאמר שהסדר בו האיברים מופיעים בו הוא הסדר בו הם יוחסנו בוללתה `for`. לכן, אין להניח הנחות על סדר האיברים בטיפוס מסוים, אלא אם נאמר אחרת בהגדרת הטיפוס.

הנחה שכן ניתן להניח (וגם זאת בזיהירות רבה מאוד) היא שאם הרצנו לולאת `for` על רשימה מסוימת, והרשימה לא השתנתה, ריצת `for` נספתחת על אותה הרשימה תבצע באותו הסדר. לקרأت סוף החוברת נראה באילו מקרים ההנחה הזאת לא מתקינה יותר.

break, continue

משפט `break` נועד ליציאה מlolאות `for` או `while` רק מבлок ה-`for`-`while` בו הוא נמצא, ולא מסוגל לצאת החוצה יותר מבлок אחד.

משפט `continue` מורה להפסיק מיד את האיטרציה הנוכחיית של הלולאה ולהתחלף את האיטרציה הבאה. כמו `break`, גם `continue` יכול לצאת רק מבлок 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 הבתנים עד הסוף מה קרה כאן, העתיקו את הדוגמה למחשב, הוסיפו `print`-ים ושנו את המספרים בדוגמה כדי להבין יותר טוב מה הקוד עשו.

בנוספ', אפשר להוסיף בлок `else` לולאות `for` ו-`while`. הבלוק של `else` מורץ במקרה בו סימנו את ריצת הלולאה כולה ולא עצרנו בעקבות `break`. דוגמה לקרה בו לולאה מצילה להסתיים בשלום ולכן בлок `else` שלא מורץ:

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

והדוגמה הפוכה:

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

range, xrange

טוב, אז יש לולאת `for` שמודעת 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 מש蒿, כי הרי איך ניצור את הרשימה? בעזרת לולאת `while`? (ואז מה עשינו בדיקו...?) או שניצור אותה ישירות בקוד (זהה מגעיל).

Python פותרת עבורנו את הבעיה באמצעות הfonקציות המובנות `range` ו-`xrange`. שתי הfonקציות עושות את אותו הדבר, אבל בצורה שונה. `range` יוצרת רשימה חדשה עם ערך התחלתי, ערך סופי וקפיצה (Step, Step) ההפרש בין צ"א מהאיברים).

מבנה הfonקציה `:range`

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

- `start` – זהו פרמטר אופציוני, ומציין את ערך ההתחלה (כולל) של הרשימה.

- `end` – ערך זה תמיד יוכל ב-`range` ומציין את הערך האחרון ברשימה (לא כולל). כמובן, הרשימה מגיעה עד הערך זהה, ולא כוללת אותו.

- step – ההפרש בין כל שני איברים סמוכים ברשימה. כמו כן, הפרש זה חייב להיות גדול מ-0.

דוגמאות ל-range:

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

כלומר, `range` יוצרת רשימה חדשה עבורנו. הרשימה יכולה למעשה להכיל כל סדרה חשבונית שהיא.

כמו כן, ניתן לחת את הרשימה שנוצרה ולהציגו אליה עם משתנה, ואז נקבל רשימה חדשה במשה מעת קד:

```
k = range(1024)
```

אז מה עושה `?xrange`

`xrange` לא יוצרת עבורנו רשימה, אלא יוצרת אובייקט רצף, שנייה לקבל ממנו התחלת, סוף ואת האיבר הבא לאיבר קיימ. בצורה אז, אובייקט `xrange` מחזיק את תחום הערבים 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 `xrange` מודיע על סוף הרשימה.

הסיבה שנרצה בכלל להשתמש ב-`xrange` היא שצצ יוצרת רשימה חדשה, רק עבור הלולאה, ולא כדי להשתמש בה מאוחר יותר, היא שיצירת רשימה לוקחת זיכרון, ואם משתמש ב-`range` עם תחום ערבים גדול, סתם נזבז זיכרון, ובמקרים של רשימה ממש גדולה היוצרה גם תיקח זמן או תקorris את התוכנית שלו במידה וייגמר הזיכרון.

כלל אגב, משתמש ב-`xrange` כנראה רק בתחום של מספרים (כמו בולולות `for`) וב-`range` כאשר נרצה רשימה מפורשת שמכילה סדרת מספרים.

המבנה של `xrange` זהה לזה של `range`.

pass

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

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

בגלל ש-`Python` מ Chapman אתתו TAB בבלוק חדש, זאת תהיה שגיאה לא לפתח 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 להזכיר שהלולאה תבוצע – היא תרוץ 100 פעמים ובכל אחת מהריצות של הלולאה לא תעשה שום דבר. אם נשים מספר מאד גדול במקום ה-500 (נגיד 50000000000) יוכל אפיו 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 ונחמדהים 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 תוכניות גדולות של קוד, או סתם תוכניות רגילות, תבוצע ע"י חלוקה לפונקציות, מודולים, וכו'.

פרק זה עוסק **בביטויים ובפונקציות.**

הגדרת פונקציות

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

כמו כן, חוקי **the 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 זו לא מקבלת שם פרמטר, וגם לא מחזירה כלום (אין בה משפט `return`). אם ניקח את ערך ההחזרה של הפונקציה ונשים אותו בתחום המשתנה, המשתנה יכיל "כלום":

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

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

None

ה-"כלום" שהרגע ראיינו מכונה ב-**Python** `"None"`. `None` הוא עצם ייחיד (**Singleton**), שמצויבים אליו כשרוצים לציין שאין לנו למה להציבו. `None` לא אומר שאין לנו מה להחזיר (כמו `void` ב-C), אלא אומר שאחנו לא מחזירים כלום (או יותר דיווק, מחזירים "כלום"). כמו כן, `None` הוא ערך חוקי לכל דבר, וניתן להשים אותו למשתנים באוטו באופן נתון להחזיר אותו מפונקציות.

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

- .1. לא לרשום שם 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. פשרו רושמים pass במקום בЛОק הפונקציה:

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

גם כאן, בדיק כmo בלולאת ה-for מסוף הפרק הקודם, pass הוא לא return או משאו 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 והוא קיימת רק כדי שנוכל להגיד פונקציות או בלוקים בלי שנחיה חייבים לשים בהם תוכן.

תיעוד

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

יש שני סוגים גדולים של תיעוד ב-**Python**:

1. סתם תיעוד שזורקים באמצעות 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 לזכור, כך גם ב-**Python** יש את התו "#" שאומר שאין להתייחס למה שכתוב החל מזה ואילך:

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

הסוג השני של תיעוד הוא סוג מיוחד, ואליו **Python** מתיחס בצורה שונה – בתחילת פונקציה, שורה אחת אחרי שורה ה-def (בלי שום רווחים מסתוריים), ניתן שם מחרוזת (כמובן, עם TAB לפניה). המחרוזת הזאת היא מחרוזת התיעוד של הפונקציה. כאשר מישרו אחר (או אנחנו) ורצה יומם אחד לדעת מרוי הפונקציה הזה, הוא יוכל לקבל את המחרוזת בקלות רבה. העובדה שהתייעוד נמצא בתוך הקוד עצמו דבר מאד חזק – זה אומר שאם יש לך את הקוד, יש לך את התיעוד. אין צורך לחפש קובץ תיעוד במקומם שלא קשור לקוד או ספר מודפס. מושג מחרוזות התיעוד בתחילת פונקציות שאל משפט התכוות **LISP**, וקרו **DocStrings** (Documentation Strings) או **DocStrings**.

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

از את מחרוזת התיעוד מקבלים ע"י כתיבת שם הפונקציה, נקודה, ו- `__doc__`. ניתן גם להשתמש בפונקציה המובנית `help` שiodעת 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
```

במהשך כשלמד 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/non 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 'ח'-'ים כאשר רוצים לרדת שורה:

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

כדי להדפיס את מחרוזת התיעוד כמו שצריך (ולא את תוכן המחרוזת עצמה), משתמש 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 long documentation
```

קצת יותר טוב, אבל הדרך הנוכחית ביותר שפיתון מציעה היא באמצעות 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 להסתבר עם ח'-'ים מגעילים, משתמשים במחרוזת עם 3 גרשימים (כפולים או בודדים), העיקר שהגרשימים הפתוחים והסגורים יהיו סימטריים). המחרוזת מסתיים כאשר פיתון פוגשת שוב 3 גרשימים, וכל מה שבאמת, כולל ח'-'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 ש כדי לקרוא את התיעוד של פונקציה משתמש ב-`help()` ולא נסתכל ישירות על המחרוזת `__doc__` שלהן. יודעת לפרט את מחרצת התיעוד בצורה יפה, בלי ה-`\n` או הרווחים שהתווסףו 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 שرك מדפסות דברים קבועים הן לא ממש שימושיות. בשילגראם לפונקציות להיות יותר מעולות נדרש להעביר להן פרמטרים. העברת פרמטרים תיעשה בסוגרים 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, מחרצת...). אם לא נהגים בחופש זה בזיהירות, ניתן להקליד קוד די פשוט.

הפרמטרים שמועברים לפונקציה מועברים אליה בדיקת כמה השם רגילה של משתנה. ההבדל הוא שהמשנה ה"חדש" שוגדר בתוך הפונקציה לא קשור למשנה שהעבכנו לפונקציה מבחוץ. לדוגמה:

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

קרהנו לפונקציה עם המשתנה `x`, וכאשר הפונקציה קיבלה אותו הוא נקרא אצלם `num`. המשתנה `num` בפונקציה `f` הוא לא אותו המשתנה `x`. שני המשתנים מצביעים באותו אובייקט (בmarker זהה המספר 1), אבל אם נשנה את `num` מתוך הפונקציה, הערך של `x` לא ישנה. כמו שאפשר לראות בדוגמה, `x` מצביע על-1 בעוד ש-`num` מצביע על-3 ברגע שהפונקציה יצאה.

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

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

אוקי, זה מאד בסיסי. מה אם נרצה לקבל מספר לא קבוע של פרמטרים? למשל, היינו רוצים לקבל כל כמה פרמטרים, או 3 פרמטרים ויתר? Python מאפשר לנו להגדיר את הפרמטרים שנחיה חייבים לקביל, וגם להגדיר פרמטר מיוחד את כל הפרמטרים ה"מיוחדים" שהעבכנו לפונקציה:

```
>>> def greet(first_name, last_name, *more_names):
...     return 'Welcome, %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 שאנו חיבים להעביר לפונקציה כדי לקרוא לה. ומה ה-`type` של הפרמטר המוחיד זהה? ניחשتم נכון – `tuple`. את הפרמטרים העודפים קיבל ב-`tuple` פשוט ונכל להשתמש בהם. ה-`tuple` יכול להיות מבן גם ריק, אם לא נעביר אף פרמטר מעבר لما שהוא חייבים. בד"כ מקובל לקרוא לפרק זה `args` (ונכתב `args` כשרצתה להיות אפיו יותר מפורשים), אבל השם הזה הוא סתם קובנצייה ולא חייבים להשתמש בו אם יש לנו שם יותר שימושי, כמו בפונקציה `greet` שכתבנו קודם.

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 שנייה על הדוגמה הזאת. איך היינו יכולים למש 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 למש.

אבל, רק בשביל הדוגמה, נניח שהיינו צריכים למש את אותה הפונקציה בדרך קצת יותר מתחכמת: בחלק, אנחנו תמיד מחברים את `num1` ו-`num2` ואז מוסיפים את מה שיש ב-`args`. אבל אפשר להסתכל על זה אחרת – אנחנו תמיד ממחברים בין `num1` לשזהו אחר. המשהו الآخر הזה הוא `num2` בלבד (אם אין עוד מספרים) או הסכום של `num2` ושאר המספרים ב-`args`. מאחר שהסכום של `num2` ושאר המספרים הוא בעצם מה שהפונקציה ע查明ה עשויה (היא סוכמת שניים או יותר מספרים, ויש לנו שניים או יותר מספרים), נוכל לקרוא ל-`sum_two_or_more_numbers` במקום לבצע את הלולאה*. טרייך נחמד, רק שבשביל זה אנחנו צריכים להיות מסוגלים לקרוא ל-`sum_two_or_more_numbers` עם מספר משתנה של פרמטרים. זה בעצם ההפך מהגדירה של פונקציה עם פונקציה עם `*args`, ו-`args` מאפשרת לנו לעשות את זה. זה נראה בדיקות

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

אותו הדבר כמו ההגדירה: 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 מוגן. עכשיו נמשח 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!'
```

שימוש 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 מחילות, והיא תחבר גם רשימות ו-tuple-ים.

פרמטרים אופציונליים

כרגע אנחנו יכולים להעביר כל כמות של פרמטרים לפונקציה, ואני יכולם לציין בהגדירת הפונקציה אילו פרמטרים היא צריכה 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. ערך Default-י אומר שאם לא העברנו ערך לאוותו פרמטר בקריאה לפונקציה, יהיה לו ערך תחתיו כלשהו. בכלל הזרה שבה מגדירים פרמטרים עם ערך תחתי, הם יכולים להופיע רק אחרי פרמטרים רגילים, אחרת Python לא ידע איזה ערך להכניס לכל פרטיה.

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 הקראיה לפונקציה, אי-אפשר להשאיר "פרמטרים ריקים" (כלומר, לשים פסיק כדי לדלג על משתנה), וחובה להעביר את כל הפרמטרים לפונקציה. אך לא נוכל לרשום מותציות מחרשת 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 הזרה הזאת בכלל לא מיוחדת והפונקציה תתבלב 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, אופציוני או "רגיל", לפי השם שלו. תכף נחזיר לנושא של העברת פרמטרים לפי שם, כי לפני זה אנחנו חייבים להבין עד מהו אחד אחרון.

בסייר הקודם הצגנו את הנושא של פרמטרים, וזה שכלנו את הפונקציות שלנו בכר שאפשרו להן לקבל כל מספר של פרמטרים. בצורה דומה, אנחנו יכולים לשככל 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` קיבל מילון שיכיל מיפוי בין כל הפרמטרים שהעבכנו לפונקציה לפי שם ולא צינו 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
```

הפייצ'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 שקראננו 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

```

עכשו 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 מיוחדים וגם פרמטרים מיוחדים 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}

```

זה המקהלה הכללי של קריאה לפונקציה. כשפונקציה נקראת, Python אוספת את כל הפרמטרים שהעבנו לפונקציה ואז:

- הפרמטרים שהועברו בלי שם ספציפי (אלה נקראים Positional Arguments) מוכנסים לפרמטרים של הפונקציה לפי הסדר בו הם הוגדרו.
 - הפרמטרים ה-Positional-Sharedeli אף משתנה מוכנסים ל-args.*.
 - אם הועברו פרמטרים לפי שם (Keyword Arguments), הערכים שלהם מוכנסים לשניים של הפונקציה בהתאם לפי השם.
 - שאר ה-Keyword arguments שאין פרמטר שמקורו לקביל אותם יוכנסו ל-kwargs**.
- אם יש איזושהי התנגשות (למשל, העבנו 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, נזכיר את ה-Conditional Expression, או בביטויי "ביטוי מותנה". ביטוי מותנה מקביל ל-ternary operator משפט C (הו if else בביטוי אחד. לדוגמה, במקום לרשום)

```

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

```

נוכן פשוט לרשום:

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

שזאת צורה הרבה יותר קצרה וקלה לקרוא, וכן נשתמש בה הרבה.

הביטוי בניו בצורה `Y if X else C`, כלומר אם `X` נכון, מספר שאנו `0`, רשימה לא ריקה, מחרצת לא ריקה או כל `list/tuple/set` שאינו ריק) אז הביטוי יחזיר את הערך `X`, אחרת הביטוי יהיה `Y`.

שימוש לב שnocn לכתוב מה שנרצה בביטויים עצם, כולל ביטויים וקריאה לפונקציות. הסיבה לכך היא ש-`X` או `Y` לא מודרים ע"י Python עד שהוא משתמש 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 בעיתויים

בפרק המבוא ראיינו שכשאנו מעתינים 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 התנהגות קיימת גם בפונקציות. אם ניצור פונקציה שמקבלת רשימה ונשנה את הרשימה מתוך הפונקציה, הרשימה تعدכן גם עבור מי שקרה לפונקציה (כי זאת אותה הרשימה):

```

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

```

אין כאן שום דבר מפתיע. בהינתן התנהגות 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

```

לכוארה 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. מהهو אחר למחרי קורה כאן:

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

בפועל, כשהגדרנו את הפונקציה לא גרמנו לה "לייצר רשימה אם לא העברנו פרמטר לפונקציה". הערך הדיפולטי של `f` הוא אותו הערך בכל פעם. זאת אותה רשימה. לא נוצרת רשימה חדשה כל פעם, וכך אם לא נעביר פרמטר לפונקציה עם רשימה ספציפית שהתכוונו לשנות, נקבל בחזרה איזושהי רשימה מלאה באיברים שלא ביקשו בכלל.

כדי לפתור את זה, מקובל ב-`Python` להעביר את הערך `None` כפרמטר לרשימה, וכך אם `p` יהיה `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]
```

שים 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 המקומיי בפונקציה, וכך הקורא לפונקציה לא ישפיע מכך.

כמו כן, שים לב שכשבדכנו האם `p` הוא `None`, לא השתמשנו באופרטור `==` אלא ב-`=`. בהמשך נפגש את זו ונבין בדוק מה הוא עשו, אבל כרגע רק תזכרו ש כדי לבדוק האם משתנה הוא `None` משתמשים ב-`=`.

משתנים בפונקציות

אחריו שהסתכלנו על ייצרת פונקציות, על תיעוד ועל הפרמטרים לפונקציות, נצלול פנימה ונראה מה קורה כאשר מכרזים על משתנים בתוך מקומות שונים בפונקציה.

הקרה פשוט ביותר, כאשר כתבים פונקציה, ופתאום מכרזים בה על משתנה:

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

בפונקציה הזאת, הוכרו משתנה בשם `i`. המסתנה זו יהיה קיים עד שהפונקציה תגיע לסופה, ואם יושמד. אם נקרא לפונקציה אחרת מתחוץ הפונקציה `func`, הן לא יכירו את המשתנה זו. יותר מכך, אם פונקציה אחרת תכריז על משתנה באותו השם, לכ"א מהפונקציות יהיה משתנה זה 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 בתחום הבלוק של הפונקציה, אלא באחד מהתחום-הблוקים שלה, בתחום תנאי `if` או לולאת `for` או `while`. במקרה זה, המשתנה שנוצר ימשיך להתקיים גם אחרי היציאה מתחום-הבלוק, בלי שום קשר לזה שהוא נוצר בתחום הבלוק (בשונה מאוד מ-`C`-וב-`C++`: Scopes-`C`-ב-`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` קיימים טיפוס הנתונים `Tuple`. בפרק על טיפוסי הנתונים סקרנו את הטיפוס, וראינו שניתן לאגד מספר ערכים ב-`Tuple` אחד. ניתן להשתמש בתוכנה 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 משתמש בפונקציה, יוכל לקבל ממנה מיד את שני הערכים בתחום 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 הבורורים יוכל להשתמש את הסוגרים:

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

והכל יעבד בדיק על 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, ועוד') עליו לבצעים עיבוד מסוים. העיבוד יכול להיות כמעט כל דבר (קריאה לפונקציות, פעולות בין שני איברים עוקבים, ועוד'). ברוב המקרים, העיבוד הזה יישנה בלולה.

פרק זה נראה אולי פונקציות וכליים 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 בעוררת לולאה. כדי שהקוד שלנו לא יתנפח מלוואות זהות שעושות את אותה עבודה שוב ושוב, נוצרו הפונקציות `map`, `reduce` ו-`filter`.

שלוש הפונקציות האלה הן פונקציות מובנות ב-`Python`, שככל מטרתן היא לחתך רשימה (או `Tuple`), להריץ פונקציה על האיברים שבה (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 שגורם כתיבה מחדש של לולאה טריוויאלית בכל פעם מחדש (ולפעמים קצת יותר מלוואה טריוויאלית).

map

פונקציית `map` 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 המקורי. בעברית: `map` לוקחת את האיבר הראשון, מריצה עליו את הפונקציה, ושם את התוצאה ברשימה החדשה. אח"כ היא לוקחת את האיבר השני, מריצה עליו את הפונקציה, ושם 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

פונקציית `map` שנייה.

`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 (ambil להריץ את הפונקציה). אם הרשימה ריקה, `reduce` תקروس ותדפיס הודעה שגיאה.

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

בדוגמה זו הודפסו 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 שאוטו `reduce` תכניס לפני כל האיברים ברשימה שלנו, ובכך 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

פונקציית `filter` קסם שלישי.

מקבלת רשימה ופונקציה. הפונקציה חייבת לקבל פרמטר אחד. `filter` תיקח כל איברי הרשימה ותריץ את הפונקציה עם האיבר זהה. אם התוצאה של ריצת הפונקציה היא `True` (אינה אפס, איננה מחרוזת ריקה, ולא הופכת לאפס אם מmirirs אותה ל-`int()`), האיבר (האיבר מהרשימה שהוא עברת ל-`filter`, לא ערך החזרה של הפונקציה) יתווסף לרשימה התוצאה.

filter מעשה מספקת 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 זו הודפסו כל המספרים בין 0 ל-19 ששארית החלוקה שלהם ב-2 איננה אפס. בKİצ'ור, כל המספרים הא-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

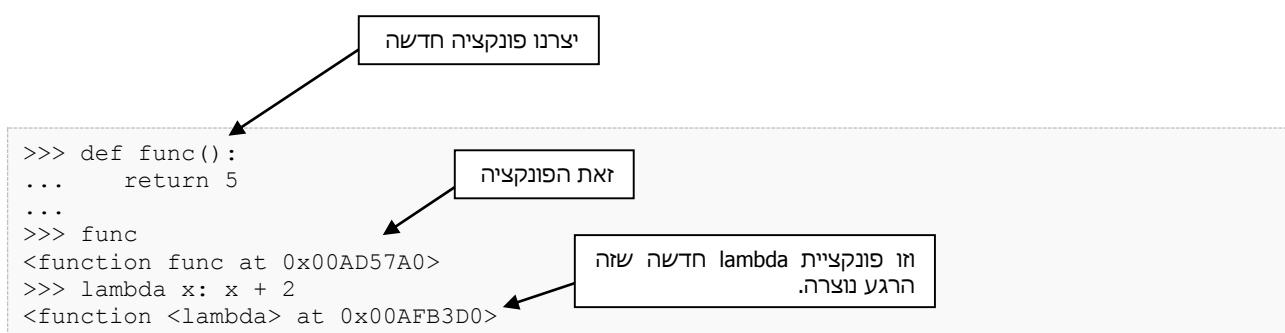
از מה `lambda` עושים בכלל ולמה היא קשורה לפרקי זהה?

כמו שאפשר לראות בדוגמה מעלה, עבור כ"א מההרצות של `map`, `filter` או `reduce` הינו 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 שמנוע אותה במקום אחר).

lambda מאפשרת לנו לחסוך עוד יותר בຄמות הקוד – במקרה כתיבת פונקציה חדשה בכל פעם, `lambda` יוצרת עבורנו פונקציה ללא-שם, כבר בתוך שורת הקוד של `map`, `filter` או `reduce`.

בתכלס, כאשר יוצרים פונקציה, השם שנוטנים לפונקציה הוא מצביע לפונקציה עצמה. זאת גם הסיבה שכאשר כותבים שם של פונקציה, בלי סוגרים, Python כותבת לנו את שם הפונקציה (היא בעצם מדפיסה לנו את האובייקט של הפונקציה). באופן זה, `lambda` יוצרת עבורנו פונקציה חדשה, ומוחירה את המצביע לפונקציה החדשה. לפונקציה החדשה אין שם, וכאשר תסתים הרצת `map`, `filter` או `reduce`, הפונקציה חסרת-השם תיעלם.

דוגמה קטנה כדי להבין יותר טוב מה קורה:



קצת על המבנה של `lambda`: כתיבת פונקציה `lambda` כולל רק את הפרמטרים לפונקציה ואת ערך ההחזרה שלה. הנה דוגמה לשימוש ב-`map` ו-`lambda` בלבד. הדוגמה עשויה את מה שעשו הדוגמה הראשונה – מקבלת רשימה מספרים ומוחירה 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 זו כתבנו פונקציה `lambda` מקבלת פרמטר אחד, `x`, ומוחירה את 2^*x . כמובן, אפשר לכתוב כל ביטוי שתלו依 ב-`x`, ואפשר גם לכלול פונקציות בביטוי זהה.

למעשה, כאשר אנו כותבים את פונקציית `lambda` הבאה:

```

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

יצרנו פונקציה, אותה יכוליםנו ליצור באמצעות `def`:

```

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

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

שימוש ב-`map`, `filter` ו-`reduce` בלבד

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

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

כasher תוכנית מעוניינת לקבל קלט מהמשתמש (דרך המקלדת בד"כ), ניתן להשתמש בפונקציה המובנית `input`.

הפונקציה מקבלת כפרמטר מחרוזת שהיא מדפסה למסך, ואז היא ממתינה לשורת קלט מהמשתמש. הקלט ישתיים כאשר המשתמש יקיש על `Enter`. דוגמה לשימוש ב-`input`:

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

ערך ההוצאה של הפונקציה הוא מחרוזת.

קיימת פונקציה משוללת יותר בשם `input` שפועלת באותה דרך, אבל במקום להחזיר מחרוזת היא מרים את הקלט שהמשמש הקlid:

```
>>> >>> input('--> ')
--> 5 + 5
10
```

נראה די מגניב בהתחלה, אבל במידה שימותה המשמש גם יכול למחוק לנו את כל הקבצים בדיסק, ולכן לא נשתמש ב-`input` כדי לקבל קלט מהמשתמש סתם ככה.

List Comprehensions

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

לשםחנותו, Python הלה צעד אחד קדימה והכניסה את `map` כתחביר של ממש בשפה. כדוגמה, במקום כתובות 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 שהבייטוי יחזיר (בדוגמה זה $x * 2$), ואנחנו קובעים את השם של האיברים בעזרת תחביר דמיי לולאת-for (במקרה זה, ... in x).

עד כאן זה לא נראה משהו, כי בעצם הצלפנו את העונש של כתיבת "lambda" בעונש אחר שבו ניאלץ לכתוב "for". זה נכון חלקיים, כי קיבלנו קוד יותר קל לקרוא. אבל הכוח האמתי הוא אם נרצה להכניס map אחד בתוך השני, וכך זה כבר מתחילה להיות ממש קל:

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

לא נכתב את כל התוצאה כי היא די ארוכה... קיבלנו כאן רשימה שמקילה מחרוזות, שבכל מחרוזת יש את השעה בכל דקה של היום:

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

בעצם, התחביר הזה הרבה יותר טוב מ-map כי הוא מאפשר לנוukan כמה לולאות בבייטוי אחד ולהתייחס לכל המשתנים של כל הלולאות בבת-אחת בבייטוי 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]
```

ולדוגמא, בואו נמצא את כל הזמןים ביום שהם 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']
```

קובצים

טיפול בקובצים ב-Python הוא די פשוט. כדי לפתוח קובץ علينا לקרוא לפונקציה open:

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

כזכור, הפונקציה open מחזירה אובייקט מסווג 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!')
```

בסוף העבודה עם הקובץ יש לסגור אותו באמצעות הפונקציה close:

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

ניתן גם לפתח קובץ לקריאה ולקראות ממנו את כל התוכן שלו לשנתה אחת:

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

עת המשנה `content` מכיל את תוכן הקובץ שיצרנו בדוגמה הקודמת:

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

בhamster נראה כיצד מטפלים ב-`Python` ביצור של שמות הקבצים (הרוי לא נפתח סתם קובץ בשם `txt.x...`) וAIR ליצר קבצים זמינים.

כמו כן, בהhamster נראה שnitin ליותר על הקריאה ל-`(close)` ע"י-כך שnitin ל-`Python` לטפל עבורנו בסגירת הקובץ.

print

את `print` פגשנו מקודם – היא מדפסה למסך. אבל עכשיו שאנחנו כבר מכירים את `Python` קצת יותר טוב, בואו נבון מה אנחנו עושים שאנחנו קוראים ל-`print`. השימוש הכל פשוט הוא להעביר ל-`print` מחוזחת 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 שנitin לראות, `print` מוסיפה אוטומטית רווחים בין הפרמטרים שנשלחים להדפסה. כמו כן, ניתן לתת כפרמטר כל סוג משתנה, כל עוד ניתן להמיר אותו למחוזחת כדי להדפיס אותו.

בנוסף, הפקודה תמיד מדפסת תוו סיום שורה בסופה, אלא אם כן שמיים פסיק בסוף השורה:

```
>>> if True:
...     print 'look!',
...     print 'this is the same line!'
...
look! this is the same line!
```

השימוש ב-`if` בדוגמה الأخيرة היה כדי ששתי הפקודות יבוצעו אחת אחרי השנייה, כי אם נכתב את הפקודות אחרת שנייה ב-`Interpreter`, `Interpreter` יירד שורה באופן אוטומטי לאחר כל ביצוע פקודה, כדי לחזור לסמן של הכנסתה הפקודה הבאה (>>).

אם נרצה, נוכל להגיד ל-`print` לכתוב את הפלט שלו לקובץ (במקום למסך), בעזרת אופרטור <>:

```
>>> f = file('/tmp/blah.txt', 'w')
>>> print >>f, 'Hello file!'
>>> f.close()
>>> f = file('/tmp/blah.txt', 'r')
>>> f.read()
'Hello file!\n'
```

כמו כן, ניתן לחת ל-`print` מחרוזת מפורמתה:

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

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

פירמות מחרוזות

אופרטור %

הדרך הראשונה והישנה יותר לפירמות מחרוזות היא השימוש ב-`%`. כבר ראינו כיצד הפירמות מתבצע – כתובים `%` אחרי המחרוזת, ואז `tuple` שמכיל את הפרמטרים שנרצה לפרט:

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

הדוגמה הזאת קצרה שונה מהדוגמאות שראינו עד עכשיו – במקרה כתוב סתם 20 כתבונו (20). הת לחבר המוחר הזה הוא דרך `-h`-Python ליצור `tuple` עם איבר אחד בלבד. בעצם, אם נכתוב (20) לא ניצרך `tuple` אלא את המספר 20, כי `tuple` מוריידה כל סוג סוגרים סימטריים (זה גם הרבה יותר הגיוני ש-(20) אומר 20 ולא `tuple` שמכיל 20).

באופן כללי, בכל מקום ב-`Python` שמצפה לאיברים מופרדים בפסיקים, יוכל תמיד לשים פסיק אחרי האיבר האחרון. מאחר ש-`Python` מנסה לקבל `tuple` אחרי ה-`%` במחרוזת, תמיד ניצור עבורה `tuple`. הסיבה לכך היא שלא תמיד ידוע מה הטיפוס שאנו מנסים לפרט. לדוגמה, אחד הפורמטים שניתן לציין במחרוזת הוא `z%` (שלא קיים ב-C), והוא אומר לפירט את הפרמטר שקיבלנו Caino היה מודפס ב-`Interpreter`. נסתכל על הדוגמה הבאה:

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

עד כאן הכל מcron. ביקשו לפירט 8 וקיבלו 8. אבל מה יקרה אם נעשה את זה:

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

קיבלו שגיאה. הסיבה לכך היא ש-`Python` מאפשר לנו להעביר פרמטר אחד בלבד של המחרוזת, אבל זה מתוקן איזושהי נחמדות לא מוסברת כלפי המשתמש. היא תשתחם בפרמטר האחד הזה Caino הוא פרמטר אחד רק אם הוא לא `tuple`. אם

הפרמטר הוא כן `tuple`, מניהה שהעבכנו לה רשימת ארגומנטים ולכון היא אוטומטית מפרקת את ה-`tuple`. בעצם יכלנו לקבל תוצאה יותר גורואה מושגיה, והיא ש-`Python` תפרמטר את המחרוזת, אך תעשה את זה לא נכון:

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

זהו בכלל לא מה שביקשנו...

לכן, מעכשיו תמיד נعتبر `tuple` כמספרם מחרוזות עם %:

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

str.format()

פירמווט מחרוזות בעזרת % היא ה"דרך הקלאסית" למספר מחרוזות ב-`Python`. הסיבה שהיא נכנסה לשימוש היא שהמבנה של המחרוזות מאד דומה לזה שב-C (השימוש ב-`%d`, `%s`, וכו') ולכון היה קל להטעין אותו בקרוב משתמשי `Python`. עם זאת, המבנה היישן הווא... טוב נו, ישן... ובגלל זה הוא קצת לא אינטואיטיבי וגם לא מאפשר להרחב אוטומטי כדי שימושו בפתרונות של `Python`.

לכן, אימצאו ב-`Python` מבנה חדש ל-`Format Strings` שקיים במקביל למבנה היישן. במבנה היישן תוכלו להמשיך להשתמש וכונראה שגם נמשיך לואות 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 לפירמווט מחרוזות:

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

במקרה זה אמרנו ל-`Python` שאנחנו רוצים שהוא תחליף את {0} בארגומנט הראשון שנעביר לו `format`. אם היוינו רוצים לתמוך בשני ארגומנטים, היוינו כתובים:

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

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

אבל אם היא פונקציה רגילה, אז כבר למדנו הרי איך עובדות הפונקציות המשוכפלות האלה ב-`Python`: הן מקבלות `*args` ו-`**kwargs` וככה הן יכולות להיות גמישות בכמות הפרמטרים שלهن. 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!'
```

זהו כבר ממשמעותית יותר נוח מ-%.

אם נרצה, יוכל להשתמש בהעברת **:Keyword Arguments**

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

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

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

כמובן, **format** יתלוון 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 זו אנו יכולים לתת יותר פרמטרים ממה שהיינו צריכים, ו-**format** יקח רק את אלה שהוא חייב לצורך לפרט את המחרוזת. כך לא נדרש ליצור אובייקט מיוחד רק עבור פירמות מחרוזות.

כשנראה בהמשך איך יוצרים אובייקטים מסוימים בין היתר טוב מה **format** הרבה יותר %%.

חלק 4: אובייקטים

ב-Python כל דבר הוא אובייקט. התכוונה הזאת נקראת ברוב שפות התכנות Object-Oriented, אך בוגוד להרבה שפות אחרות שרק תומכות באובייקטים, Python באמת מייצגת כל דבר בה בעזרת אובייקטים. מספרים, מחרוזות, רשימות, מילונים, `None`, ופילו פונקציות – כולן אובייקטים.

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

id() ו-is()

בפרקם הקודמים אמרנו שאנו לא משוים ל-`None` עם האופרטור `==` אלא בעזרת האופרטור `is`. אפילו ציינו את הסיבה לכך, והוא ש-`None` הוא `Singleton`, כלומר יש רק אחד כזה. הוכחדה שיש רק `None` אחד אומרת שלא מעוניין אותנו להשווות משתנה מסוים ל-`None`. לצורך העניין, ההשווה לא רלוונטי כאן, כי השוואה בודקת את התוכן של האובייקט. אנחנו רוצים לבדוק האם המשתנה שלנו מצביע לאובייקט, וכך התוכן בכלל לא משנה.

כדי שנוכל לדעת האם שני אובייקטים הם אותו אובייקט בבדיקה 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 יוכל לבדוק בודאות שהם מצביעים באותו 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
```

צר נוכל להשווות בין ה-`(id)`-ים של שני משתנים ולדעת האם הם מצביעים באותו אובייקט:

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

마חר שהשווות `(id)`-ים היא פעולה נפוצה, ב-`Python` קיים אופרטור עבור ההשווה זו, והוא האופרטור `is`:

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

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

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

אמנם ניתן למש את "Y is not X is Y" בעזרת "not", אך הגרסה "Y is not X is not Y" יותר קלה לקרוא ולבן 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, בדיק כmo שהיינו מצפים:

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

אבל, הם לא אותו האובייקט:

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

כשכתבנו `x = 1000` גרמו 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 ליצור שני משתנים שמצביעים לאותו אובייקט, נכתוב:

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

ואז נקבל את התוצאה הרצוייה:

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

נקודת אחורונה שחשוב לבחיר לפני שימושה נמייר היא שאם ננסה להריץ את הדוגמאות האלה עבור מספרים קטנים יותר, ניתקל בתופעה שונה:

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

לכוארה מה שאמרנו כאן לא נכון.

הסיבה לכך היא ש-`Python` שומרת עותקים מוכנים של המספרים השלמים שבין 5- ל-256. הסיבה לכך היא שאליה מספרים דינמיים ו-`Python` מעדיפה לשמר עותקים שלהם במקום ליצור אותם כל פעם מחדש.

type

עכשו כשים בידינו את דרך לבחין בין אובייקטים שונים, נמייר ונסתכל על אחת התכונות של אובייקטים שכבר פגשנו קודם: `type`. `type` הוא אובייקט (מספר, מחרוזת, פונקציה, ...). יש סוג (`type`) של אובייקט נוכל לקבל ע"י קראיה `:type` לפונקציה המובנית:

```
>>> 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 `list`. אבל הרי אמרנו שכל דבר ב-`Python` הוא אובייקט, ולכן גם הтиיפוס של אובייקט הוא אובייקט. בעצם, `type` זאת לא פונקציה שמדפיסה על המסר טקסט שאומר מה הтиיפוס של משהו, היא מוחזירה את ה-`type object` שלו. אז אם פונקציה מוחזירה ערך, אנחנו יכולים לשמר את הערך זהה במשתנה:

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

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

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

שימוש לב שימוש ב-`is`: אנחנו הרי רוצים לדעת שהтиיפוס הוא אותו טיפוס (אין משמעות לתוכן של ה-`type object`).

Type Objects

יש הרבה מספרים בעולם. יש גם הרבה רשימות, והרבה tuple-ים ורבה מיליוןים. אבל לכל אחד מלאה 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 במאמר (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 (שים לב שבדוגמה הזה מודפסות מחרוזות שמקילות את הייצוג הטקסטואלי של האובייקטים שהמранו למחרוזת, ולמרות שהן נראהות 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]'
```

از בעצם, גם מעין פונקציות שמייצרות אובייקטים חדשים מהסוג של עצמן, וגם משתמשים אותו כדי שנוכל לזהות את האובייקטים אחר-כך ולהגיד מה סוג של האובייקט.

בפרק הבא נראה גם שקבועים את ההתנהוגות של האובייקטים, ונראה מהי בדיקת התנהוגות של אובייקט ואיך אפשר לשנות את התנהוגות האלה ע"י-כך שינוי type objects משלנו.

Immutable ו-Mutable

בינתיים נסתפק בכך ש-type objects קיימים ובכך שאנו יודעים להבדיל בין סוגי שונים של אובייקטים. אבל לא נסתפק בכך שעדין 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 שייצרנו 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 לשנתנה החדש.

התנהוגות הזאת היא הפך הגמור מההתנהוגות של שאר האובייקטים. לדוגמה, נסתכל על רישומות:

```
>>> 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 כשהאנחנו מושנים את תוכן של הרשימה. אז מה בעצם ההבדל בין מספרים לרישומות? ההבדל הוא שאובייקטים מתחלקים לשני סוגים:

- אובייקטים שהם Immutable objects לא ניתנים לשינוי מהרגע שייצרנו אותם. אלה הם המספרים, החזירים None (שכבר נוצר עבורונו) ומחרוזות.
- Mutable objects: אלה כל שאר האובייקטים, ואפשר לשנות אותם אחרי שנוצרו.

חלוקת זאת קיימת בעיקר בעבר מילוניים. כשפגשנו את המילוניים בפעם הראשונה לא טרחנו לצייןעובדת דיאט שפוגעת ב对他们. אם ננסה להכנס למילון key mutable שהוא keys, נקבל שגיאה:

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

(לפni שנסמץ'יך, וודאו שהבנתם שניינו ליצור מילון שבו יש מפתח [] שמצוין לערך []).

הסיבה שmailto לא אהוב שה-keys הם mutable היא שהmailto לא יכול להרשות עצמו שנקnis לתוכו זוג (key, value) ואז נשנה את ה-key מתוך לרגלים שלו. אם נשנה את ה-key אז יוכל ליצור התנטשה (שני keys שונים בתוך אותו מילון), וגם נפריע לסדר הפנימי של המילון (איך הוא ימצא את ה-key אם נשנה את הערך שלו?).
לכן, יש לנו טיפוסים שהם immutable שביהם בטוח להשתמש כключи במילון.

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

dir() ו-Attributes

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

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

הו append Attribute של-listים. אובייקט append (כי הרוי כל דבר ב-Attribute הוא אובייקט):

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

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

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

אז Attribute (לעתים נשתמש בקיצור attr) הוא בסה"כ אובייקט שמוכל בתחום אובייקט אחר. ההבדל בין attr לבין כל אובייקט ש"סתם" מוכל בתחום אובייקט אחר (למשל, רשימה מכילה אובייקטים שהכנסנו לתוכה) הוא של attr-attribים יש שם. כאשר אנחנו רוצים לפנות לattr 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__', '__setattribute__', '__setitem__', '__setslice__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

attributes שמהחילים ונגמרים בשני קווים תחתונים נקראים slots, ועל רובם נדבר בפרק הבא.

אחרי ה-slots למיניהם אפשר לראות את כל ה-list attributes ש-`list` מזיהה לנו: `insert`, `index`, `extend`, `count`, `append`, `remove`, `pop` ו-`reverse`. כדי לראות מה עושה כל אחד מה-attributes האלה, נוכל לקרוא `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 `l()` sort ואילו פרמטרים `sort` מקבלת במידה ונרצה לשלוט על הזרה שבנה המיןון מתבצע. באופן כללי נוכל לקרוא `l().help()` ולקבל את התיעוד של ה-`attribues` של הרשימה, ושל כל אובייקט שנגנש.

Reference Counting

כשאנחנו כתובים שורה קוד כמו זו:

```
x = 1000
```

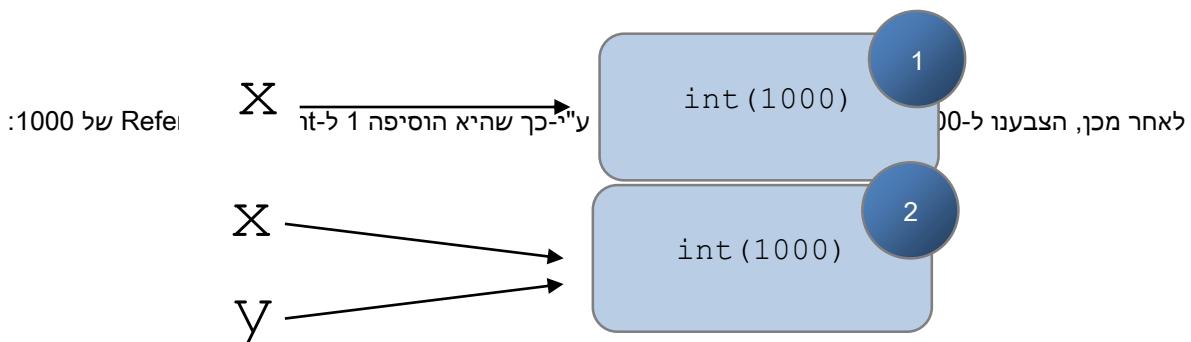
טורחת ויזכרת עבורנו אובייקט חדש מסוג `int` ומכוונה לתוכו את הערך 1000. נוכל גם להמשיך ולהשתמש במשתנה `x`:

```
y = x
```

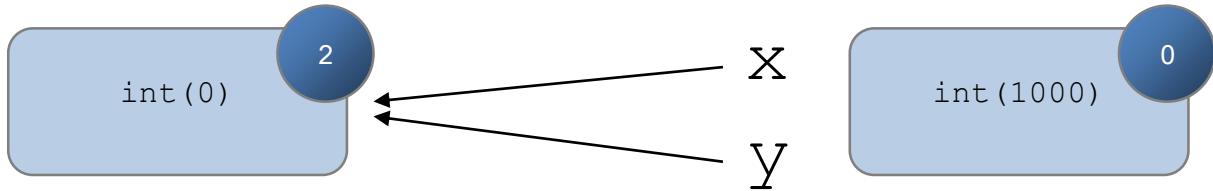
ו-`Python` יודעת ש-`x` הוא אובייקט כלשהו ו-`y` מצביע לו אותו אובייקט ש-`x` מצביע אליו. אבל מה יקרה אם נכתוב פתאום את השורה הבאה:

```
x = y = 0
```

אף אחד לא צריך יותר את ה-1000 שיצרנו קודם. יותר גרווע מזה, אין לנו איך לגשת ל-1000 זהה אפילו אם נרצה. האם Python יודעת שה-1000 זהה לא נחוץ יותר או שאולי היינו צריכים לעשות משהו כדי לסיים את השימוש באותו מספר? ברור ש-`Python` יודעת ובורור שלא נדרש לעשות כלום. הדרך שהיא שיפרנו 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 והציבענו אליו עם `x`, יצרנו את המצביע הבא:



לבסוף, כששינו את הצבעה של `x` ו-`y` לאובייקט אחר, ה-`Reference count` של 1000 ירד ל-0:



בນקודה זו, Python יודעת שה-`int` שלנו לא בשימוש יותר והוא מסמנת אותו. באיזהו נקודה שפה יהיה ה-`int` now, היא תנקה את האובייקטים המיותרים. תהליך הנקוי זהה נקרא Garbage collection ולא נכסה את התהליך הזה במסגרת הלימוד שלנו.

עלינו לזכור רק ש-`Python` אין לנו דרך לדעת מתי יבוצע Garbage collection, לאחר שהוא עצמה את הזכות לשנות את השיטה לפיה היא מבצעת נספחים שעדי לא ראיינו (כמו Exceptions) וכן Python שומרת לעצמה את הזכות לשנות את השיטה לפיה היא מבצעת Garbage collection כדי שאפשר יהיה לשנות אותה בין גרסאות מוביל לפגוע בקוד קיימ.

קבצים

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

קובץ הוא אובייקט די פשוט. אחרי שיצרנו אובייקט `file`, אנחנו יכולים לבצע עליו מספר די מצומצם של פעולות (read/write/seek) ובסיום 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 לנטרש את אובייקט הקובץ (לדוגמא ע"י קר שפושט נצא מפונקציה שבה הקובץ הוא משתנה מקומי) ואז Python תשמיד את האובייקט מתישחו ועל הדרך גם תסגור את הקובץ.

במקרים בהם כן נרצה לבדוק מתי הקובץ נסגר – למשל אם נרצה לקרוא קובץ שהרגע סימנו לכתוב, או אם נרצה למחוק קובץ כשהוא עדין פתוח – נהיה חייבים לסגור את הקובץ בזורה מפורשת ע"י קריאה ל-`close()`.

בד"כ נפתח קבצים לקריאה או לכתיבה בלבד, ולכן לא יעניין אותנו לוודא שהקובץ נסגר. כדוגמה, במקרה הפחות 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 ניתן להשתמש בקבצים תוך כדי שימוש בכל העולמות – גם נדע בדיקת אם הם נסגרים, אך מוביל לשמור אותם במצב מיוחד או לקרוא ל-`close()`.

לבינתיים נסתפק בכך שאין צורך לקרוא ל-`close()` מפורשות, לאחר שזהו המקורה הנפוץ.

חלק 5: מחלקות

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

class

כדי להתגבר על הדרישות שיכולה להיות לכל מפתח שהוא, Python מאפשרת לנו ליצור סוגים אובייקטיבים משלנו. כדי ליצור סוג חדש של אובייקט, משתמש במילה השמורה `class`:

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

בaya הגדרה של מחלקה חדשה. למחלקה החדש קוראים Greeter במקורה זהה, ויש בתוכה מתודה אחת בשם `greet` שמודפיסה למסך הודעה. מתודה (method) היא פונקציה שפעולת על אובייקט מסוים (לבדיל מ"סתם" פונקציה). הפטר self ש-`greet` מקבל הוא פרמטר שחייב להיות הפרמטר הראשון בכל מתודה במחלקה. self הוא למעשה משתנה המכיל את האובייקט עליו תורץ הפונקציה, כי אם נרים "סתם פונקציה", היא לא יכולה לדעת על איזה מופע של Greeter עלייה לפועל.

את המשמעות של `(object)` נבהיר בהמשך.

השימוש במחלקה שייצרנו מאוד פשוט:

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

ונקרא מופע (Instance) של Greeter. אם נרצה, יוכל ליצור כמה `Greeter`-ים של :

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

וקיבלנו Greeter אחר. x ו-y הם שני אובייקטים שונים, אבל יש להם משווה אחד משותף, הרוי שניהם `Greeter`-ים של Greeter. נוכל לדעת זאת ע"י קריאה לפונקציה `type()`:

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

בעצם, הוא `type-object` Greeter חדש שאנו 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 זו רואים מודול בירור ש-x מועבר כפרמטר self כשםתודה נקראת, ו-self הוא ה-`instance` עליו פועלת המתודה כשהיא נקראת. במקרה של המחלקה שלנו לא עשינו שם דבר עם self, אך מיד נראה מדוע אנחנו צריכים את ה-.`instance`

init

אם אתם זוכרים, כשפגשנו את (`dir()` ראיינו של אובייקטים יש הרבה attributes שמהחילים ונגמרם בשני קווים תחתוניים (מעכשו פשוט נכתב `__`). ה-`attributes` האלה נקראים slots, ומיד נראה שהוא הרבה ה-`attributes` האלה הם מותודות. המתודה הראשונה שנזכיר היא `__init__`, והיא המתודה שנקראת כשהונוצר 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 זו, `__init__` יכולה לקבל פרמטרים: כל פרמטר שנעביר בסוגרים ל-`Greeter` בעת יצירת האובייקט יועבר כפרמטר ל-`__init__` בנוסף לפרמטר self (אותו Python מעבירה עבורנו בצורה אוטומטית).

בדוגמה זו נראה העברת פרמטר ל-`__init__` וכייד המחלקה שלנו שומרת את הפרמטר זהה 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 שמחישה את אותה נקודה בצורה אחרת:

```

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

```

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

cut נשלל מעט את הדוגמה של המחלקה :

```

>>> 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 מתוך מתחם המחלקה ע"י שימוש ב-*self*. לאחר מכן אפשר לקרוא מתוך מתחם המחלקה עצמה, אנחנו משתמשים במתודה (*set_greeting()*) כדי לאתחל את *self._greeting*. ובוצרה הזו אנחנו גםאפשרים להשתמש לשנות את הברכה שתודפס כשהוא יקרא ל-*(greet)*.

בנוסף, את הברכה שהמחלקה מדיפה אחסנו ב-*attribute* *_greeting* ולא *greeting* כמו בדוגמה הקודמת. הסיבה לכך היא שב-*Python* אין באמת דרך להחביא *attributes* כך שלא ניתן יהיה לגשת אליהם מחוץ למחלקה. אמן קיימת שיטה להקשוט על המשתמש לגשת ל-*attributes* פנימיים (ומיד נראה איך), אבל מאחר שהשיטה רק מקשה ולא באמת פתרת את בעיית הגישה מבוצע, לא ממקובל להשתמש ב-*attribute* לשם שמן שהוא שומרן פנימי של המחלקה פשוט *נשיים* לפני השם שלו.

Attributes פנימיים

כמו שראינו לפני שורות, לעיתים גרצה שמיישו שימוש במחלקה שלנו בצורה חיצונית לא יוכל לגעת ב-*attributes* פנימיים למחלקה. דוגמה ל-*attribute* פנימי הוא הברכה ש-*Greeter* מדיפה, כי יכול להיות שבונוסף לברכה אנחנו גם שומרים במחלקה את השפה שבה עליינו להדפיס ואם המשמש ישנה את הברכה בלי לשנות את השפה נדפיס ברכה לא נcona (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 מחלוקת שסופרת כמה פעמים היא הדפסה את הברכה שאמרו לה להדפס. בכל פעם שהברכה מודפסת, מdad הnimos ("politeness") עולה ב-1, כי כמה שנברך יותר פעמים ניחשב יותר מונומסים. אבל, אם נשנה את הברכה ניאלץ לאפס את מdad הnimos שלנו, כי הוא כבר לא יהיה נכון לאחר 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']

```

הגדשנו הם בעצם __greeting ו-__politeness attributes. כשאנחנו ניגשים ל-greet __attribute שמתחליל בשני קווים תחתונים אבל לא מסתיים בשני קווים תחתונים, Python תמיד מוסיפה לפני קו תחתון ואת שם המחלוקת שבה אנחנו רצים כרגע. אם אין מחלוקת, מקבל שגיאה וכן 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 מנגן הינה מפני טעויות, ולא מפני משתמש שיזכר מה הוא עושה. לכן, מקובל להשתמש בשיטה שראינו בסוף הסעיף הקודם (הוספה קו תחתון בודד לפני שמות של attributes פנימיים) ולא במנגן ש-*Python* מספקת.

repr, str

אם ננסה להדפיס אובייקטים של המחלוקת Greeter ב-*interpreter* נגלה שהיא interpreter לא יודעת להגיד עליה יותר מדי:

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

וגם קיבל את אותה התוצאה אם ננסה להמיר את `greeter` למחרצת:

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

כדי שנוכל להמיר מחלוקת למחרצת וכדי שהוא תודפס כמו שצריך ב-`interpreter` נדרש למש את ה-slot-ים `-str` ו-`-repr`. המתודה `_repr_` תקרא לצורך הדפסה ב-`interpreter` או כשנקרא מפורשת לפונקציה (`repr()`). המתודה `_str_` תקרא כשנקרא `str()` עם האובייקט שלנו. אם לא נמש את `_str_`, Python תקרא עבורנו לימוש של `_repr_`, אך שם אין לנו רעיון טוב למשוך עבור `_str_` מספיק למש רק את `_repr_`. לצורך הדוגמה נמש מחלוקת 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)'
```

از מה ההבדל בין `_str_` ל-`_repr_`? למה לא היה מספיק שהייה רק אחד מהם? ההבדל הוא ש-`_str_` הוא למתודה שתחזיר מחרצת שתיצג את האובייקט שלנו עבור המשתמש, ככלומר עבור בן-אדם. שימוש אפשרי ל-`_str_` הוא להדפיס את התוצאה שלו למסך כחלק מאיזושהי אינטראקציה עם המשתמש.

לעומת זאת הוא `_repr_` למתודה שתחזיר, בתקווה, מחרצת שמייצגת את האובייקט כאובייקט. המטרה כאן היא שאם ניקח את התוצאה של `_repr_` ונעתיק אותה ל-`interpreter` נקבל בחזרה את האובייקט שהיה לנו המקורי. במקרה של המחלוקת `Snake` עשינו את זה די בקלות, אבל יש הרבה מחלוקות שעבורן לא יוכל למש מתודה `_repr_` שתאפשר לנו לשחזר את האובייקט המקורי שהוא לנו ולכן רק נשתדל להחזיר מחרצת שתתאים את האובייקט בצורה מספיק אינפורטטיבית לצורך הדפסה ב-`interpreter`.

`getitem`, `setitem`, `delitem`

שלושה slot-ים שימושיים הם `_getitem_`, `_setitem_` ו-`_delitem_`, והם ה-slot-ים שנקרוים כשהאנחנו פונים לאובייקט עם סוגרים מרובעים. כדוגמה, נמש 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)
...

```

נבדק קודם שהメתודת `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. כתה נראה איך `__getitem__`עובדת:

```

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

```

שימוש לב ש-`__getitem__`עובדת בדיקת כמו רשיימה רגילה (תוכלו לבדוק את אותו הקוד על רשיימה רגילה ב-interpreter בעצמכם). כתה נבדוק את `__setitem__`.

```

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

```

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

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

בדוגמה זו מימושו אובייקט שמגיב ל-items מספריים, אבל באותה קלות נוכל למש 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 במרקחה זהה הוא מילון מרושע שהופך את key בכל פעם שאנו מכניסים לתוכו איבר, כך שנוצרת להפוך את key כשרצה לקבל את האיבר מהמילון. לא שימושי במיוחד בחוכנות אמיתית אבל אחהלה דוגמה.
שימוש לב שללא מימושו את __delitem__ ולאחר מכן האובייקט שלנו לא תומך ב-del. בעצם נוכל למש רק את-slotים שנראות לנו, וכל slot שלא נ渺ש הוא פונקציונליות שהאובייקט שלנו לא יתמוך בה. כמובן שם נסה לבצע del במרקחה זהה נקבל שגיאה, ונתקבל שגיאה אם נסה להשתמש ב-set/get __getitem__/_setitem__.

ירשה

בדוגמה האחרונות פגשנו את ScumbagDict. בסה"כ הוא די בסדר, חוץ מ-slot אחד קצת דפוק. נגיד שנרצה לתקן אותו, האם נדרש למש את כל המילון מחדש? הרי רק מתודה אחד לא טובה בו וכל השאר בסדר. Python יש מנגנון בשם **ירשה** (Inheritance). ירשה מאפשרת לנו לקחת אובייקטים קיימים ולהחליפם 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}
```

בקarakה זהה הוא השם של המחלקה והיא יורשת מ-ScumbagDict. NiceDict היא subclass של ScumbagDict, ומושגים, ScumbagDict היא superclass של NiceDict. הסוגרים 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 שימושות הירושה ממנו.

הפעולה שביצענו עבורי __setitem__ נקראת "דרישה" (override), כלומר דרשו את __setitem__ של ScumbagDict וחלפנו אותו בגרסה אחרת משלנו.

cut, נשתככל קצת. נתחילה מחלוקת 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
```

מחלקה מייצגת חיים, כאשר החיה יכולה לגודל, לאכול, לשחק, ללקת לשירותים ולישון. שימוש לב שמלבד העבודה שהחיה שלו לא עושה הרבה חוץ מלעדכן כמה מונחים, לא היינו רוצים להחליף אף מתודה בה.

cut נגידיר מחלוקת 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 מגדרה, Dog מכילה 2 מתודות חדשות – מתודה לנביחה ומתודה לכשכווש בזנב. יש לשים לב שבמתודה __init__ של Dog יש קרייה למתודה __init__ של Animal. דבר זה נעשה כדי לאפשר לא-Animal לאתחל את כל ה-attributes של פונקציית כליה. קרייה זו לא נעשית אוטומטית לאחר שיכשאנחנו דורסים מתודה אנחנו מחליפים אותה וכך נרצה לקרוא למתודה של superclass ניאlez לעשות את זה ידנית.

שימוש במחלוקת Dog הוא בדיקת כמו שימוש במחלוקת 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 שמדוברים הרבה יותר מאשר מצלבים 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 מетодה של `dog`, צריכה לחפש את ה-`bark` Python, `aging_dog` אומנם נקרה למетодה (`grow`), נוכל למצאו אותה מיד במחלקה `AgingDog`. אבל מה יקרה אם נקרה לא-שרצינו לקבל. למשל, אם נקרה למетодה (`bark`), נוכל למצאו אותה מיד במחלקה `Dog`. אולם מטרתנו היא למצוא את `bark` (`aging_dog`) ולא `Dog`.

```

>>> aging_dog.bark()
Waff Waff

```

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

```

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

```

כעת, ככל ש-`type object` יש גם מוחדר בשם `__bases__`:

```

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

```

הוא בסה"כ שמכיל את כל המחלקות שירשנו מהן. במקרה שלנו זהה מחלוקת אחת, ובהמשך נראה מה יקרה כשהנירש מכמה מחלקות.

אז מכאן זה כבר די פשוט – בכל פעם שנחפשים מוסויים 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 יהיה יותר).(`__bases__`

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

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

הטכנית היזת נקראת MRO (Method Resolution Order), והיא דואגת לשני דברים. קודם כל, היא מודדת שנמצא את מה שchipshnu. בנוסף, היא מודדת שנמצא את המתוודה הנכונה. אם בכל אחד מה-objects type (object) בעץ הירושה שלנו) $\text{AgingDog} \rightarrow \text{Dog} \rightarrow \text{Animal} \rightarrow \text{object}$, היינו צריכים לדעת שכשאנו קוראים ל-`age` מ-`AgingDog` (היתה המתוודה לא קיבל בטעות את `age` (), ומארח שהחיפוש מתבצע לפי סדר הירושה אנחנו יודעים שתמיד קיבל את המתוודה הנכונה.

isinstance, issubclass

שתי פונקציות שימושיות שקיימות ב-`object`-`issubclass`. שתיהן יודעות לטויל על המסלול של ה-

`__bases__` שנוצרים בירושות שלנו, וכך אנחנו יכולים לקבל בזמן ריצת התוכנית מידע על האובייקטים שלנו.

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

:`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 גם מוגדר כ-`instance` של `AgingDog`, אבל הוא אינו יורש מ-`Dog`. וזה הרי די הגיוני, כי אם יורש מ-`Dog` אז כל מי שיקבל אובייקט מסווג `AgingDog` יוכל לעשות אליו כל דבר ש-`Dog` ידע לעשות (ולא אמרו להיות לו אפשרות שהוא יורש מ-`Dog` שהוא קצת יותר משוכל).

באופן דומה, 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
```

שים לב שהסדר חשוב בשתי הפונקציות – האובייקט הראשון הוא תמיד האובייקט שעליו אנחנו שואלים האם הוא subclass או not, והאובייקט השני הוא הסוג אליו אנחנו משוורם. אם נתבלבל נקבל תשובה לא נכונה או שגיאה.

از מה אפשר לעשות עם שתי הפונקציות האלה? נראה דוגמה: במחלקה `Animal` מימשנו מетодת `eat`, אבל ה-`eat` של החיות שלנו מאוד מוזר כי הוא לא מקבל אוכל. חייה בדרך כלל אוכלות אוכל, ומארח שאוכל הוא אובייקט הוא צריך מחלוקת משלו:

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

וופי. עכשו 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
```

עכשו החיים שלנו הרבה יותר הגיונית. כשהיא מקבלת לאכול משהו שאינו אוכל היא לא עורשה אותו כלום ופשט חזרה. רק אם ניתן לה אוכל, היא תאכל אותו עד שהיא תשבע. אם האוכל לא יספיק לה שביל לשבע היא תישאר רעהה במידה מסוימת. נראה דוגמה לישירות על `:Animal`

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

super

עכשו יש לנו חיים שיעודו לאוכל ולא דברים אחרים. אבל נניח שהכלבים אוכלים לתרנגולות את כל האוכל (הרוי כלבים הם חזקים ותרנגולות הן חלשות ומסכנות). עלינו לשנות את המימוש של הכלבים כך שאכלו רק אוכל של כלבים.

מה בעצם נרצה לעשות? נרצה שהמחלקה `Dog` תירש מ-`Animal`, תדרוס את `eat` ותודא 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-`eat` של `Animal`. אלא למתחודה "ה-`eat` של מי שירשנו ממן". כמובן, 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)` (שים לב שהחłówנו גם את המימוש של `super`). יודע להחזיר אובייקט שמשמש כמתוך שיעודו להחזיר לנו את המתודה הנקונה עbor ה-`superclass`-`__init__`.).

שלאו. ה-`super` נקבע בצורה דינמית לפי ה-`__bases__` של ה-`object` העברנו ל-`super`, במקרה זהה זהו `Dog`.

super חוסך לנו הרבה חישבה, כי כל מה שעליינו לעשות הוא לציין מיהו ה-`class` הנוכחי ו-`Python` עשו בשבלנו את כל העבודה. אבל למה בעצם צריך לציין את ה-`class`? הרי אנחנו יודעים באיזה מחלקה אנחנו. למה `Python` לא יכולה לקרוא בלבד ל-`type(self)` ולהוסיף 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)
...

```

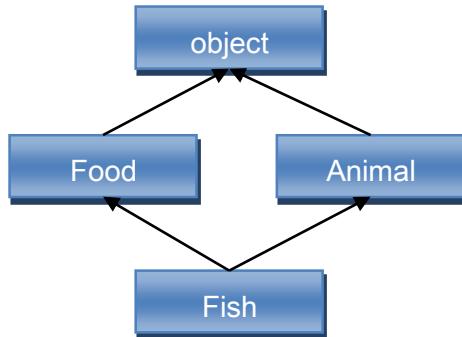
כאשר נקרא ל-`eat` עם האוכל המתאים, המתודה נקראת `super` ל-`super`s. במקרה הספציפי זהה הוא אכן `type(self)`. אבל כשהגענו ל-`Dog.eat` מה יהיה `type(self)`? הוא גם יהיה `AgingDog`, הרי ה-`type` של אובייקט לא משתנה. אם לא נגיד ל-`Python` איפה אנחנו בשרשראת הקריאות של הפונקציות, היא לא תוכל לדעת מאיזה `type object` לחת 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` והוא שאנו לא תמיד יודעים מייה ה-`superclass` שלו.

ירושה מרובה

חיות שאוכלות אוכל... כמה תמים. שכחנו דבר אחד קטן: יש חיים שאין להם אוכל.

בסעיף הזה ניאלץ לפתח מחדש את כל המחלקות שכתבנו, ונראה כמה חמור יכול להיות מצב בו לא קראונו ל-`super` מ-`__init__`. כמו כן, בסעיף הזה נראה **בירשה מרובה היא דבר רע**. הוא מיצרת קוד מסובך ולכך רצוי להמנע ממנו. אז נתחיל מהסוף – מה המטרה שלנו? להגיע למחלקה שירשת מ-`Food` ומ-`Animal`:



נגדיר את המחלקה הזאת:

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

כעת עלינו לחשב למי super(self) הולך – ל-Food או ל-Animal. לפי ה-MRO Python לוקחת את `__bases__` ועובדת עליהם אחד אחריו השני. נבדוק מהו `Fish.__bases__`:

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

כלומר, `super(Fish, self)` יקרא ל-Food. אבל מישחו צריך לקרוא ל-`Animal.__init__`, וכך זה מתחיל להסתבר. המשמעות של `super(Fish, self)` היא "תקרא לתור בשרשרת הירושה". אז אם שרשרת שלנו היא ש-Fish יירש מ-Food ומ-Animal, כאשר `__init__` של Fish קורא ל-`__init__` של Food, ההגדרה אומרת שה"תור" של Food ו-Animal, ייקרא ע"י `Animal.__init__`. כלומר, `Animal.__init__` ייקרא ע"י `Food.__init__`.

נזכיר על זה: `super(Fish, self)` יקרא ע"י `Food.__init__`. כך, למרות ש-object יורש מ-Animal, לא שמע על Food (פחות 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 מ-Animal:

```
>>> 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 ליצר `instance` של `Dog`, 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 שיכל לאכול את עצמו:

```

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

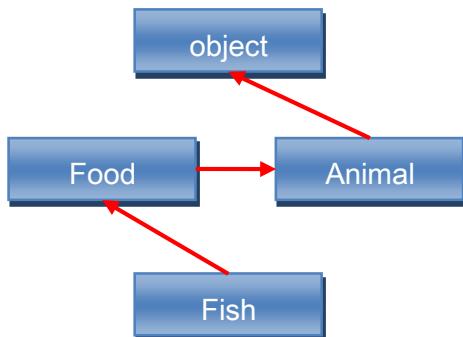
```

נסביר שוב מה קרה כאן בamilims אחרות, לאחר שקצת קשה להבין את ההתנהגות הזאת בפעם הראשונה: ירושה רגילה שבה מחלוקת יורשת בלבד היא מקרה פשוט. יש `__init__` אחד, קוראים לו וממשיכים להשתמש באובייקט כמו שציריך. לרוב `__init__` מקבל פרמטרים, ולכן נעביר לו את הפרמטרים שהוא יצפה להם.

בירושה מרובה ב-Python אנחנו מייצרים לעצמנו עיטה בכך שאנחנו יירושים מחלוקת ודורסים את ה-`__init__` שלו, כי דרשו יותר מ-`__init__` אחד. פתרון אחד לבעה (שהוא הפתרון הקל, אבל לא בהכרח הכי נכון) הוא לדרוש מהמתכנתן לקרוא לכ"א מפונקציות ה-`__init__` של `superclass`-ים ישירות. ככה לא נשכח אף `__init__` ולא נוכל לטעת 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 `__getattribute__`)? נctrיך תמיד לקרוא מפורשת למחלקה `super` שלנו, וזה לא תמיד אותה מחלוקת. העיטה מחמירה אם נירש בצורה מרובה מכמה מחלוקות שלכלן מחלוקת `super` משותפת, כאשר רק חלק מהמחלקות דורסות מתודה מסוימת*.

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

לכן ב-`Python` החליטו ללקת על הפתרון הכללי יותר. בפועל נוצרך לכתוב פחות קוד, אבל ניאלץ להבין טוב טוב מה אנחנו עושים. הפתרון של `Chos` הוא לחת את הירושה המורובה ולנסות להפוך אותה לירושה רגילה, ע"י-כך שבמקרים לדמיין עצ' ירושה מסווג, 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 זהו הוא שכאשר אנחנו קוראים למethודה ב-`superclass`, נחפש את המethודה קודם כל במחלקות שירשנו מהן, אחר כך במחלקות שהן ירשו מהן, וכך נטיל על הרמות בעז ולא נבצע סתם רקורסיה כלפי מעלה. אפשר גם לדמיין שאם המחלקה שלנו היא יلد או `Python` מחפשת מתחודה אצל ההורים, אחר כך אצל הסבים והסבנות, אחר כך אצל סבא וסבתא הרבה, וכך הלאה.

ambilhetnu, גודיף להמנע מירושה מרוביה, ותמיד נזכיר להשתמש ב-`(super)` כשןקרה למתחודות של ה-`superclass` שלנו ולא להשתמש שירות 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 סוגרים מרובעים אחרי האובייקט. אבל יש דבר הרבה יותר בסיסי מסוגרים מרובעים והוא `__get/set/delitem`. מה אם נרצה ליצור אובייקט שיש לו את אותה גמישות שקיבלנו ב-`__attributes`, רק במקומות `?attributes` עם `items`:

אפשרת לנו לעשות זאת בעזרת `__delattr__`, `__getattribute__` ו-`__setattr__`. 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}

```

methode ה-`__init__` אמרה להיות די ברורה. המתוודות `__getitem__` ו-`__setitem__` עובדות בדומה ל-`__getattribute__`, מלבד הבדל אחד קטן. גם Python משתמש בהן כדי לגשת ל-`attributes`. הרזה כל הקטע של `__setitem__`, אנחנו בוררים את ה-`attributes` שנקניש ל-`_items` (אלה הם שמתחילים בקוו תחתון). כל מי שלא מתחיל בקוו תחתון מופנה ל-`_items`. כל מי שמתחיל בקוו תחתון מופנה ל-`super`, כי אנחנו רוצים تحت למימוש המקורי לטפל בו. וכך שאפשר לראות, הדוגמה הזאתעובדת בדיקן כמו שרצינו.

dict

כדי למש את מגנון ה-`attributes`, Python צריכה להחזיק מיפוי בין שם של attribute לבין האובייקט שהוא מייצג. למשלנו כבר יש לנו סוג כזה של אובייקט – מילון. מילון הרי יודע לשומר מיפוי בין keys ל-values.

במקרה הספציפי של `attributes` אנחנו רוצים להחזיק מיפוי בין keys str ל-values כלשהם. זה בדיק מה ש-`dict` – לכל אובייקט עם `attributes` מוחזק attribute בשם `__dict__` ששמור בין שמות ה-`attributes` לעצם. אם נסתכל על(`dir`) שהרץנו 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 שלו. הרি את כל דברים היא יכולה להשלים מה-`object`,`type`, זה בדוק מה שהוא ערשו. כשאנו פונים לאיישו `attribute`, מחפשת אותו ב-`__dict__`, ואם הוא לא נמצא היא פונה ל-MRO.

כרגע בדיקת `__getattribute__` ו-`__setattr__` עובדים בדוגמה מה壽יר הקודם. כל פניה ל-`attribute` שלא שמרנו לעצמו (אלה הינו הפניות ל-`attributes` שהכילה קו תחתון בתחילת השם שלהם) הופנתה למימוש הדיפולטי של Python.

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

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

שלושת הפונקציות האלה מאפשרות לנו לגשת ל-`attributes` בצורה דינמית, ככל מרבי שנדע בזמן כתיבת הקוד את שם ה-`attribute`. למשל, נוכל לעבור על כל ה-`attributes` באובייקט מסוים ולבדק האםcolm 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 זכרה – `dir` היא פונקציה שמחזירה רשימה של מחרוזות, כשל מחרוזת מכילה שם של `attribute` באובייקט. נסביר אגב את מה שכבר הבנו בצורה צאת או אחרת – `dir`, הפונקציה הפשוטה לכארה, בעצם עשויה די הרבה. היא עוברת על כל ה-`keys` של `__dict__`, על כל ה-`object`,`type` `attributes` של ה-`__bases__` של ה-`type`. על ה-`object` ועל ה-`attributes` שלו בצורה רקורסיבית, ובסיום מחזירה רשימה אחת יפה וממוינת עם כל ה-`attributes` שנוכנלו למצוא באובייקט שהעברנו ל-`dir`.

לසימן, נזכיר שיש אובייקטים שאין להם `__dict__`. דוגמה אחת היא `int`. בהקשר זה אנחנו יכולים להגיד של-`int`-ים אין מה-`object`. זה נכון בקשר בו ל-`int` אין `__dict__`, כל עוד ברור לנו שהכוונה ל-`attributes` דינמיים ולא אלה שמופיעים מה-`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

עכשו שהבנו איך אובייקטים אמורים להתנהג ואנחנו מבינים מהם type-objects ומה העבודה שלהם, נוכל סוף סוף להבין מיהו אותו object מסטורי שנאלצנו לרשף ממנו בתחילת הפרק שאפילו לא ידעו מהי ירושה.

ב-Python יש שני סוגי מחלקות – Old-style ו-New-style. כשהאנחנו יורשים מ-object אנחנו מייצרים אובייקט מסווג-new-style-class. כמו שאפשר לנחש, אם לא נירש מ-object נקבל אובייקט מסווג-old-style-class. כמו שם מרמז, זה לא דבר חיווי אחרית השם שלו היה יותר מושך. ניצור מחלוקת style-old 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 עכשו היינו מקבלים הרבה יותר attributes. הסיבה העיקרית לכך (שלא נתעמק בה) היא שב-new-style-objects המתוודות של האופרטורים (לדוגמה __add__, __getitem__ ו __init__) ממומשות במחלקה האב (object) ונקראות ממש. בעולם ה-old-style לאובייקט יש רק את המתוודות שהוא הגדר לעצמו וניתן אפילו לדروس אותן בזמן ריצה.

נקרא כתה `l-` עם האובייקט שלנו:

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

עכשו ברור שמעבר להתנהגות המשונה של dir(), אין סימטריה בין type-object ל-(type). כזכור ב-Old-style-classes, סימטריה בין type-object ל-(type) מאפשרת לנו לקבל פיצ'רים מתקדמים בשפה (כמו __getattr__) וכן היא מאפשרת לנו לרשף מהטיפוסים המובנים של Python (אפשר לדוגמה לרשף מ-dict וליצור לעצמנו סוג חדש של מילון).

ב-3.0 של Python לא תהיה תמייהה בין-old-style-classes לבין-new-style-classes והוא רק/new-style-classes. כדי להיות מוכנים לעתיד הקרב ובא, ניצור רק מחלקות `new-style-classes` כדי שהקוד שלנו ירוז בסביבת Python 3.0 גם אם לא נשתמש `new-style-classes`.

slot-ים נוספים

Python מאפשרת לנו לדرس הרבה slot-ים נוספים כדי שנוכל לאובייקטים שלנו להשתלב יפה בסביבה פיתונית. בסעיף זה נציג את-slot-ים ומה תפקידם, אך לא נפרט עליהם כי עוד לא למדנו על Exception-ים (אותם נכסה בפרק הבא). בעזרה החומר שלמדו עד כה ואחרי שנלמד על Exception-ים תוכלו בקלות להבין כיצד להשתמש ב-slot-ים שנציגן כאן.

- `__cmp__`, `__le__`, `__lt__`, `__ne__`, `__eq__`, `__ge__`, `__gt__` הם slot-ים להשוואה בין שני אובייקטים. שנייהם מקבלים תמיד שני פרמטרים (self, other) וצרכיהם להחזיר True או False, מלבד `__cmp__` שמחזיר 1 או -1 ועל-column תוכלו לקרוא בתיעוד של Python.
- `__call__` הוא slot שנקרא כאשר אנחנו "קוראים" למחלקה שלנו כailo היא פונקציה.
- `__mul__`, `__add__`, `__sub__` ועוד רבים אחרים משמשים כדי לתמוך בפעולות חישוביות. כך נוכל לדוגמה לחבר בין שני `instance`-ים של המחלקה שלנו.
- `__contains__` מאפשר לנו למש את האופרטור `in` עבור המחלקה שלנו.
- `__len__` יחזיר את אורך המחלקה.
- `__str__`, `__int__`, `__long__`, `__float__` עושים בדיק מה ש-`str` עושה עבור מחרוזות, ומאפשרים למספר מתודה שאומרת איך להמיר את האובייקט שלנו למספר, מספר ארוך או float.
- `__enter__`, `__exit__`אפשרים לנו למש `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" שימושי מילוניים כדי לדעת איפה לאחסן ואריך למצוא אובייקטים בהתאם keys במילון.

חלק 6 : Exceptions

בודאי שמתם לב שעד עכשו אמרנו הרבה פעמים "שגיאה" על כל מיני מקרים שלא התאפשרו בהם, אבל לא הגדרנו בקורס מדוקט מהי שגיאה. יותר מכך, לא אמרנו מה אפשר לעשות עם השגיאות האלה. בפרק זה נזכיר מנגנון שימורshi מאד ב-Exceptions Python שנראה Exceptions.

או שניים של Python ולכן `Exception`-ים כבר לא ייראו כמו שהוא חדש במיוחד.

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

דרך אחת בה `int` יכולה להגיד לנו שהיא לא יכולה לקבל את הקלט 'a' היא ש-`int` תמיד הייתה מחזירה `tuple`. באיבר הראשון היה הימרה מוחזירה `True` או `False` כדי להגיד האם הצלחה או נכשלה, ובאיבר השני היה הימרה מוחזירה את התוצאה אם היא הצלחה או `None` אם לא. אבל אז תמיד היו צריכים לבדוק, בכל קראיה ל-`int`, האם היא הצלחה או לא. כל פעם. לא משנה ש-`int` כמעט אף פעם לא נכשלת, תמיד נצטרך לבדוק מה היא החזירה.

יותר גורע מזה, לעיתים אנחנו עשויים לשכוח לבדוק את ערך ההחזרה ואז נהיה חסופים לבאגים מוחרים ששתחמש בערכיים לא נוכונים בכל מיני מקומות בלי שנדע מאיפה הם הגיעו באמצעות (הרי יש עוד פונקציות חוץ מ-`int`...).

מה שאנו רוצים הוא פתרון פשוט ידיע לנו שימושה לא הייתה בסדר, וככה גם לא נצטרך לבדוק האם כל שורת קוד הצליחה וגם לא נוכל לשכוח שום שגיאה.

אובייקט Exception

ນשייך בדוגמה שמננה 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'
```

הדף דרש דבר שנראה `Traceback`, והוא קורה כשתחרחת שגיאה ואף אחד לא מטפל בה. השורה הראשונה היא כותרת קבועה שתמיד אומרת זהה `Traceback`. השורות האמצעיות אומרות לנו מאייפה בקוד קורתה השגיאה, ככלומר אילו פונקציות נקראו עד שהגענו למצב שלו הגענו. השורה الأخيرة והכי מעניינת היא השורה שאומרת לנו מה קרה, ובמקרה שלנו היא מתחילה ב-`ValueError`.

נקlid `ValueError` ב-`interpreter` וונגלי:

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

:Exception. ספציפית, נוכל גם להגיד שהוא יורש ממחלקה קדומה יותר שנראית `ValueError`

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

וגם נוכל להגיד ש-`Exception` (ולכן גם `ValueError`) יורש ממחלקה נוספת שנראית `BaseException`

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

השגיאה שלנו היא בסה"כ ירוש `instance` של `ValueError`, `ValueError` מ-`BaseException`.
בגודל, `BaseException`-ים הם בסך הכל אובייקטים. אובייקט `exception` חייב לרשת מ-`superclass` שנקרא `exception`. Python דורשת רק דבר אחד והוא שאי-שם בראש עץ הירושה של המ-`BaseException` יהיה `exception`.

אם נרצה נוכל ליצר אובייקטי `exception` סתם ככה:

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

וכמו כל אובייקט רגיל נוכל לשמר אותו במשתנים, לקבל את המ-`type` שלהם וקומו ליצר `exception`-ים مثلנו.
כדי ליצר `exception`-ים مثلנו, כל מה שעליינו לעשות הוא להגדיר מחלקה שירושת מ-`Exception`. לאחר מכן:

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

וקומו נוכל ליצר `exception`-ים של המ-`instance` החדש שלנו:

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

זריקת exception-ים: raise

עכשו כשראינו איך ליצר אובייקט עבור המ-`exception` שלנו, ננסה לבצע את ההתנהגות של `int`. אנחנו נעשה ממשו קצר יותר מעניין והוא לכתוב פונקציה שמכינה לקבל רק מספרים שמתחלקים ב-3. אם המשתמש נותן פונקציה מספר שמתחלק ב-3, היא מחזירה את המספר מחולק ב-3. אם לא הוא מתחלק ב-3, ניצור שגיאה בדיקת 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
```

שימוש לב שבקורה זהה לא הגדרנו `exception` משלנו, אלא השתמשנו ב-`ValueError` שקיים ב-`Python`. מקובל להשתמש במ-`exception`-ים המובנים של `Python` כי גם מישחו שלא מכיר את הקוד שלנו יוכל להבין מה השגיאות הנפוצות אומרות.
cut 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 שהוא 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
```

הfonקציה mysterious_func שלנו עשתה דבר שנקרא "לזרוק exception". הפקודה raise ל-*Python* בקצת את exception ה-*Python* שהיא זריקה 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 מנגןון לתפיסת exception-ים.

trap...except :Exception

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 כניתיקל באיזושהי שגיאה נרצה לטפל בה. אם נשאר עם הדוגמה מהסעיף הקודם, נניח שיש לנו את func_mysterious וונניח שלא ראיינו את הקוד של הfonקציה אלא רק קראנו את ה-*doc-string* שלה, נרצה לתפוס את exception-הן ו לטפל 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 קוד שמקבל קלט מהמשתמש, ממיר אותו ל-*int* (כי קלט מ-*raw_input* כ-*str*) ואם הצלחנו להמיר אותו ל-*int* נקרא ל-*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 שקלטת מספר מהמשתמש, ופונקציה שקוראת ל-`statements` עד שהוא מצליח לקבל ערך בחזרה, ובשתי הפונקציות אפשר לראות שהשתמשו בשני `try-except`:

כאשר אנחנו כותבים `try` בקוד שלנו, Python מצפה למצואו בлок של קוד, בדיקות כמו פונקציה או קטע `if`. מה שמיוחד ב-`try` הוא שאם ייזרק `exception` מתוך הקוד בבלוק-`try`, הוא ייחסם כך שנוכל להגדיר בדיקות איך לטפל בו. זה לא משנה אם הקוד בבלוק-`try` יעשה `raise` שירות, או שאולי ייזרק `exception` מתוך קריאה לפונקציה. אם `exception` כשלשו ייזרק מתוך הקוד ב-`try`, הוא ייחסם בرمאה הזאת כדי שנוכל להגיד איך לטפל בו.

שיטת הטיפול הראונה שנכיר היא `except`. כשאנו כותבים בлок `except` בקוד שלנו, אנחנו צורכים לציין את ה-`type` של `exception` שנרצה לטפל בו. אם נסתכל שוב על `get_int_from_user`, נוכל לראות שכתבנו `Python` בעצמם. כלומר בлок `try-except` יטפל רק ב-`exceptions` מסוימים – `ValueError` או שם `ValueError` עצמו. `except` משתמש ב-`isinstance` כדי לוודא את סוג ה-`exception`.

נסתכל לדוגמה על קטע הקוד הבא שמתבצע עד שפקודת `raise` מתרחשת. ברגע שmaguiim ל-`raise`, השיטה עוברת לבlok 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 occurred'
... except Error2:
...     print 'Error #2 has occurred'
... except Error3:
...     print 'Error #3 has occurred'
...
Hello
World
My
Name
Error #2 has occurred

```

קטע ה-try מתחילה לרוֹץ...

נזרקת שגיאה והשליטה
עוברת מיד לקטע הקוד
המתאים.

ואכן, השגיאה שנזרקה הייתה מסוג `Error2`, וקטע ה-`except` המתאים ה被执行.

ה-Exceptionים של Python

ל-Python יש הרבה `exception`-ים מובנים שמשמעותם עם השפה, וברוב המקרים בכלל לא נדרש להגיד `BaseException` ומשלו.Cut נפגוש את ה-`exception`-ים השימושיים יותר, ובתייעוד של Python תוכלו למצוא את `BaseException` ואת כל ה-`exception`-ים שיורשים ממנו ומובנים בשפה.

OSError

הו `exception` שנזירק כמשמעותו רע קורה בקריאה לפונקציה של מערכת הפעלה. בפרק הבא נלמד על מודולים ונפגוש שם את המודול `os` אך לבינתיים נסתפק בכך שאם נכתב `"import os"` יוכל להשתמש במודול `os` ולקרוא לפונקציות של מערכת הפעלה ממנו.

נתחיל בכך שמיוצר קובץ (הקובץ יוצר בספרייה הנוכחית, אבל זה לא ממש מעניין כי עוד רגע נמחק אותו):

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

마חר שלא נזירק שום `exception` אנחנו יכולים לדעת שהקובץ נכתב בהצלחה. Cut נמחק את הקובץ:

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

ושוב, מאחר שלא נזירק שום `exception` אנחנו יודעים שהקובץ נמחק. Cut ננסה למחוק את הקובץ שוב, מה שאמור לגרום ל-`exception`:

```
>>> 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 להתחל לחשוב מה קורה בטור האובייקט OSError, כי OSError נשמע כמו שם מאוד כללי שיכל לעמוד על איזושהי שגיאת מערכת-הפעלה, ושגיאת מערכת הפעלה לא בהכרח אומרת שהקובץ לא נמצא, היא גם יכולה להגיד שאין לנו גישה אליו או שהקובץ עדרין פתוח ע"י מישהו אחר.

מהסיבהazzo אובייקטי OSError מכילים attribute בשם errno שמכיל קוד שאומר מה הייתה השגיאה. במקרה שלנו הקוד הזה הוא 2, אך ממש לא צריך לזכור את הקודים האלה מכיוון שיש מודול נוסף בשם errno שמכיל קבועים עבור השגיאות שמערכת הפעלה 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 שיצבע לאובייקט ה-exception שקיבנו. בנוסף, השתמשנו ב-`raise` בלי פרמטרים מותר לעשות רק בתוך בлок והוא גורם ל-`exception` הנוכחי להפסיק לפעע לבוקי ה-`try` שימושינו, וממשים בו במקרה שבו החלטנו שאנו לא יודעים איך לטפל ב-`exception` ולכן נמשיך לזרוק אותה למעלה.

במקרה שבו בדקנו האם errno הוא ENOENT, שהוא קבוע שאומר שלא מצאנו את הקובץ שchipshnu (קיצור של ENTry NO). אם errno שקיבלו הוא לא ENOENT אנחנו עושים `raise` ולא מטפלים ב-`exception` זהה, כי הרוי העבודה של הפונקציה שלנו הייתה לוודא שמחוקים קובץ ומתחומים ממנו אם הוא לא קיים, ואם לא החלטנו למחוק את הקובץ בטוח לא עשינו את העבודה שלנו וכן אנחנו חייבים להפסיק לזרוק את ה-exception הלאה.

IOError

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

לדוגמה, נוכל לקבל IOError אם ננסה לפתח 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'
```

שימוש לב שכיבוכו היו אמורים לקבל OSError עם errno של ENOENT, כי הרוי זאת בדיק או שהיא כמו בחלק הקודם – ניסינו לפתח 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 צפוי:

```
>>> 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'
```

שימוש ב-sh.open.os בכלל לא מחרירה file-object אלא מספר, וכאמור די מסובך להשתמש בה בהשוואה ל-file ולכן השתמש ב-file-objects כמו שלמדנו עד עכשיו.

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

NameError

NameError קורה כאשרנו מנסים למשתנה שלא קיים. להבדיל מ-exception-ים כמו OSError או IOError NameError קורה כתוצאה לכך שתכתבנו קוד שלא עובד כמו שצריך (לפעמים קוראים לזה באג...).

למשל (שימוש לב לשגיאת הכתיבה):

```
>>> flei('/etc/passwd').read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'flei' is not defined
```

ואת אותו exception קיבל גם אם נפנה למשתנה שלא קיים בפונקציה או מתודה.

הדבר חשוב להבין כאן הוא ש-Python מוחפש משתנים בזמן הריצה של הקוד, ומאחר שאין כאן תהליך של קומpileציה כמו ב-C, Python מוצאת משתנה לא קיים רק כשהיא מגיעה להריצת הקוד שכתבנו. לכן, שגיאות שהן כתוצאה חלק מהקוד הן גם exception-ים רגילים ו מבחינות Python אין הבדל בין הדרך שבה נגדיד למשתמש שקובץ לא קיים ובין הדרך שנגיד לו שחווסף 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
```

ובbor 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`, כשנכחות איזושהי פיסת קוד ש-`Python` לא יכולה להבין, יירק `SyntaxError`. ה-`error` הזה אומר שקיימת שגיאה כלשהי בתחום, ככלומר שרשותנו טקסט שאינו קוד `Python`. לדוגמה:

```
>>> if 1
  File "<stdin>", line 1
    if 1
      ^
SyntaxError: invalid syntax
```

בדוגמה זו רשותנו "if". אין עם זה שם בעיה, חוץ מהעובדת ששכחנו לשים נקודתיים בסוף השורה. מבחינת `Python` זה לא יכול להיות ולכן קיבלנו `SyntaxError`. דוגמה קצרה 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 שמננו אינדנטציה לא נכונה, ולכן `Python` התלוננה של שורת ה-`print` הוא סוג של `SyntaxError` כי:

```
>>> issubclass(IndentationError, SyntaxError)
True
```

דוגמה נוספת כדי להמחיש את הנקודה ולסייע:

```
>>> if x < 1:
...     print 'Blah'
... else:
...     print 'Bleh'
... else:
  File "<stdin>", line 5
    else:
      ^
SyntaxError: invalid syntax
```

כি ברור שאי-אפשר לשים שני `else`-ים באותו בLOC `if`.

KeyboardInterrupt

את `KeyboardInterrupt` מקבל כשהמשתמש יקיש `CTRL+C`. `CTRL+C` הוא צירוף מקשים שמותר למשתמש להקיש כשהוא רוצה להפסיק, ובದ"כ תוכניות יוצאות כשהמשתמש מקיש `Python`. `CTRL+C` מmirrh את ההקשה הזאת ל-`exception` שפושט נזרק במקום שבו הקוד שלנו רץ כרגע.

זאת דוגמה מצוינת לכך שגם אם נכתב קוד מושלם עדין יהיו exception-ים, כי זה מנגנון שמשמש את Python כדי להודיע לנו שקרה משהו, ולא בהכרח שהיתה איזושהי שגיאה.

כדוגמה, נקרא ל-`get_int_from_user()` 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, לאחר 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'
```

זהה דוגמה נוספת למקורה בו אפשר לשים כמה בלוקי `except` כדי לתפוס שגיאות מסוגים שונים. ננסה את הפונקציה החדשת 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 לציין ש-`Python` היא אمنה שפה מוגיבה 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 occurred'
...     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 שרשמו את `except` `Exception` לפני `ValueError`. כש-`Python` מחפשות את בלוק ה-`except` שהוא צריך כדי לטפל בשגיאה, היא עוברת בлок עד שהיא מוצאת בлок `except` שבו יש `type` שלו שעבורו מתקיים `isinstance` 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 כשנקרה לפונקציה ונקליד `CTRL+C` מקבל:

```
>>> 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 "עליה" של ירשה בין ה-`try`-`exception`.

TypeError

הו `exception` די נפוץ, ונקבל אותו במקרה 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)
```

בעצם מאוד דומה `ValueError`, רק ש-`ValueError` היא עברו `TypeError` ו-`TypeError` היא עברו `ValueError`. בנוסף, חשוב לציין ש-`TypeError` לא מיועדת לשימוש של Python בלבד, ומומלץ מאוד לזרוק `TypeError` אם ציפינו למשתנה מסווג מסוים וקיים בו סוג אחר. מואוד מקובל ב-`Python` לזרוק `TypeError` במקרים כאלה ומתקנות אחרים מצפים לקבל `exception` ולא ספציפי שנוצר ע"י המתכונת. אם נזכיר בדוגמה החותם שוכלות 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 בлок נוסף שיכל להגיע אחרי `try` והוא בлок `finally`. אחרי בлок `try` אנחנו יכולים לשים כמה בלוקי `except` שונים, או לא לשים `except` בכלל, ואחריהם בлок `finally`. מה שבлок `finally` עושים הוא לחת לנו הزادנות אחרונה להריץ קוד לפני שנמצא מבlok 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 בлок finally היא לאפשר לנו להריץ קוד ביציאה מבלוק ה-try בלי קשר לשאלת האם נזדק exception. המקרה פשוט הוא המקרה בו בлок ה-try הסתיים בהצלחה: במקרה זהה הקוד ממשיר לרווח מסוף בлок ה-try ישירות לבlok ה- finally ומשם מסתיים בлок ה-try והקוד הנלווה אליו.

אם נזדק exception מוצאת את בлок ה-except המתאים (אם יש אחד כזה) ומሪיצה את הקוד שבו. אחרי שבlok ה-except מסיים את הרצת הקוד שלו, Python מחפשת בлок finally ואם יש אחד כזה היא מרייצה את הקוד שבו אחרי בлок ה-.except

בנוסף, בлок ה-finally יתבצע אפילו אם בлок ה-except זורק exception (לדוגמה ע"י raise). לפניו זריקת ה-.exception. החידש Python תריץ את הקוד של finally ורק אז תמשיך לזרוק את ה-.exception.

ההגון הוא שבлок finally מאפשר לנו לכתוב קוד שմבצע סיום או ניקוי של מה שהתחלנו בבלוק ה-try. לדוגמה, יכול להיות שנרצה לוודא שסגורנו קובץ לפני שנצא מקטע הקוד שלנו:

```

>>> f = file('/etc/passwd')
>>> try:
...     content = f.read()
... finally:
...     f.close()

```

– מאלצת אותנו לשים את בлок ה-finally אחרי כל בлокי ה-.except, וזאת כדי להציג שהוא תמיד רץ אחרון – אחרי בлок ה-try ואחריו בлок ה-except שאולי יירוץ.

בלוק else

בלוק try מאפשר לנו לעטוף קוד ולהגיד מה נעשה אליו במקרה מקרים: נשים בлокי exception כדי להריץ קוד ספציפי אם היה exception. נשים בлок finally כדי להריץ חלק מהקוד בכל מקרה (אם היה exception וגם לא היה exception). חסר רק עוד משагה אחד – בлок שניים אם לא היה exception.

אם שמננו לפחות בлок exception אחד, נוכל להוסיף אחרי בлокי ה-.except (finally) גם בлок else של קוד שיירוץ רק אם בлок 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 בлок `else`? הרי הוא קורה אם לא היה `exception`... יכולנו פשוט לכתוב את הקוד ב-`.try...except` בלי `else`, פשוט אחרי בлок ה-`authenticate`

זה נכון, ובמקרה הספציפי זהה באמת היינו יכולים ממש את בлок ה-`try` בלי `else`, ע"י כך שהיינו משתמשים ב-`break` בблок ה-`except`. אבל אם היינו מוסיפים גם בлок `finally`, זאת הייתה הדרך היחידה להכניס קוד שיתבצע בין סיום בлок ה-`try` לתחילת בлок ה-`except`, שירץ במקרה שלא היה אף `exception` ושלא יטופל ע"י בлокי ה-`try`-`except` של בлок ה-`try` שלו. דוגמה:

```

>>> 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 מוקדם, בפרק הבא נלמד על מודולים, אך כדי לסייע את הפרק על exception-ים علينا להכיר את `.exc_info()` במודול sys יש פונקציה חשובה שנקראת `exc_info()` שכשנקרה לה נקבל:

```
>>> import sys
>>> sys.exc_info()
(None, None, None)
```

במקרה שלנו, קראנו ל-`sys.exc_info()` כשהיינו "סתם בקוד רגיל". אבל, אם נהייה 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 exception, זכרת את המידע על exception הנוכחי במקומות גלובליים בזיכרון ומאפשרת לנו לגשת לו אותו מקום ע"י קריאה ל-`sys.exc_info()`.

מה שאנחנו מקבלים הוא tuple עם שלושה איברים. הראשון הוא ה-type של exception, השני הוא ה-`instance` והשלישי הוא אובייקט traceback. כן, אףלו traceback שמודפס למסך בסה"כ אובייקט.

בגלל שיש לנו גישה ל-exception הנוכחי, אנחנו יכולים לדעת כמה דברים. קודם כל, אנחנו יכולים לדעת האם יש כרגע exception בתהליכי טיפול מכל מקום שנחיה בו בקוד. בנוסף, אנחנו יכולים לדעת לבדוק אם היה הקוד היה exception ע"י-כך שנבחן את אובייקט 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 שmagינה על עצמה מפני משתמשים רעים, בלי שהמשתמש בכלל יודע שהסתכלנו האם יש exception שמטופל כרגע.

דבר נוסף שהוא יכול 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
  File "<stdin>", line 1, in <module>
    main__.BadPassword

```

כולם, מה שאמרנו מוקדם לגבי היכולת לעשות raise מתחוץ בлок except תקין גם מתחוץ פונקציות. Python לא צריכה לראות שאנו עוזרים exception ספציפית מתחוץ בлок, 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 exception אחד שמטופל בכל רגע נתון. זה אומר שאנו נגרום לזריקת exception מתחוץ 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 סימנת `uk`, אך קיימות גם סימנות אחרות, לדוגמה `lll` או `os` (ב-`Windows` או `UNIX`), שמצוינות קובץ שמכיל פונקציות Python שכתובות ב-C (או באיזושהי שפה מקומפלט אחרת). לדוגמה, מרבית הפונקציות במודול `os` כתובות בשפת C.

כאשר קובץ הוא בעל סימנת `py`, יודעת שהקובץ מכיל טקסט רגיל, וצורך להתייחס אליו כאל קוד Python. קוד Python הוא כל דבר שהכרנו עד עכשו – משתנים, פונקציות, שימושים במודולים אחרים, `if`, `for`, `else`, `while`, `class`-ים. אם ניצור פונקציה בקובץ כלשהו, ונשמר אותו עם סימנת `uk`, נוכל לטען את הקובץ ל-Interpreter או לモודול אחר ולהשתמש בפונקציה שכתבנו.

לדוגמה, נניח שיש לנו את הקובץ הבא, שנקרא `py.example`:

```
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
```

ואחריו 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]
```

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` ולמצאה שהוא ממומש בקובץ `example.py`. לנו בתור משתמשים thereby לא אכפת אם המודול הוא עם סימנת `uk` או `lll` או `os`.

Python מבchinתה יודעת למצוא את המודול הנכון ולעשות לו `import`.

דרך נוספת לעשות `import` היא לציין 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'

```

בדרכו זה Python מקבלת בדיקת השמות שהיא צריכה ליבא, ולאחר היבוא אפשר להשתמש באובייקטים שיבאונו 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 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

```

נՐץ 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.

```

Namespaces

לפנינו בדיקת מה ההבדל בין שתי שיטות `import` שראינו מקודם נctrיך להכיר מושג חדש-ישן שנקרא `.namespace`. בעםם כבר פגשנו `namespace` בעבר אבל קראנו לו `__dict__` – אם אתם זוכרים, בכל פעם שאנו 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 שאנו ייצרים ב-`.interpreter`. אם נבדוק, נוכל לראות של `example` גם יש `__dict__` ונווכל 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` כראנו:

```

>>> 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, הוא כל מיפוי של שמות משתנים לאובייקטים, וב-*namespace* הוא ממומש ע"י מילון. אם נחשוב על זה קצת, ב-*Python* יש שני סוגי של *namespace*-ים: ה-"nocopy" שבו אנחנו רצים בכל רגע ו-*namespace*-ים אחרים שבו אנחנו צריכים לפנות אליהם בצורה מפורשת (לדוגמה המודול *example*).

כasher אנחנו רצים בתחום פונקציה ופונים למשתנה, *Python* ידעת בדיק איפה לחפש את המשתנה. *Python* גם ידעת לשומר את המשתנים המקומיים של הפונקציה כך שכשהיא תסיט לרווח המשתנים האלה יושמדו ולא נראה אותם יותר. לכן, יש עוד *namespace* שמחזב מאיתנו ושבד היום לא שמו לב אליו. ה-*locals* זהה נקרא *locals*.

начילה בפונקציה פשוטה שמחזירה רשימה:

```
>>> def f():
...     x = []
...     return x
```

בפונקציה זו נוצרת רשימה חדשה שמוכנסת למשתנה *x*, ולאחר מכן *x* מוחזר למי שקרה לפונקציה. בפועל, האובייקט היחיד שוגן בכל סיפורו הזה הוא הרשימה, כי רק על הרשימה יש *reference-count* שטoper כמה משתנים מצביעים אליה.

אם ננסה לבדוק ב-*interpreter*, נוכל לראות שכמה קריאות עוקבות ל-*f* נקבל את "אותה הרשימה":

```
>>> id(f())
139935593701816
>>> id(f())
139935593701816
>>> id(f())
139935593701816
```

הסיבה לכך היא מאד פשוטה: מיד כשהרשימה מוחזרת מ-*f* אנחנו בודקים את *id* שלה, ולאחר מכן הרשימה מושמדת כי אף אחד לא מצביע אליה יותר. בקריאה הבאה נוצרת שוב רשימה חדשה כתגובה שבה הייתה הרשימה הישנה שהושמדה.

עתה נשנה קצת את *f*:

```
>>> def f():
...     x = []
...     return locals()
```

locals היא פונקציה שמחזירה את ה-*namespace* המקומי, כלומר את ה-*locals* שמחזיק את המשתנים המקומיים. נראה מה *f* ממחזירה:

```
>>> f()
{'x': []}
```

כלומר השם וקריאה 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 יורד 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**, בኒיגוד למשתנים **локליים**, מוגדרים ברמת המודול כולה. 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 **CONST** יודעת בזמן הריצה למצבו את המשתנה **CONST**? בזמן שהפונקציה רצתה היא מחפשת את המשתנה **CONST** ב-(**locals**) ולא מוצאת. במקרה זהו היא עוברת לחפש את **CONST** במשתנים הגלובליים, ככלمر ב-**namespace** של המודול כולה. באוטו namespace מוגדרים גם לדוגמה **multiply** עצמה וה-**__doc__** של המודול כמו שראינו קודם.

כדי לגשת ל-**dict** של המשתנים הגלובליים נוכל לקרוא לפונקציה **(globals)**. כדי להציג את (**globals**), נוסיף את הפונקציה **get_globals** ל-**mod1**:

```
mod1.py  
CONST = 17  
  
def multiply(num):  
    return num * CONST  
  
def get_globals():  
    return globals()
```

ונראה ל-**get_globals()** 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**, **multiply** ו-**get_globals**, בנוסף לנימה **attributes** נוספים שמוגדרים בכל מודול:

name

הו **attribute** מסוג **str** שמכיל את השם של המודול. בד"כ קיבל מחרוזות פשוטה כמו '**mod1**', אך בהמשך דוגמאות פחות טריוויאליות בהן קראנו למודול **mod1** מפייה שם קצר יותר מסובך.

file

מחרוזת המכילה את השם של הקובץ ממנו המודול יובא. שם הקובץ שנקלט בחרוזת זו לא תמיד **path** אבסולוטי, למשל במקרה של **mod1**:

```
>>> mod1.__file__  
'mod1.pyc'
```

כלומר שם הקובץ כאן הוא ביחס לסדרית העבודה הנוכחית, ולכן __file__ שימoshi רק בהנחה שלא שינו את סדרית העבודה הנוכחית בעזרת os.chdir().

package

מצין את שם ה-package שבו המודול נמצא. בהמשך הפרק נלמד על package-ים.

builtins

עד עכשיו פגשנו הרבה פונקציות ואובייקטים שהיו "מובנים", ככל פרשטו השתמשנו בהם והם הגיעו מאיפושהו. מעתה מוקומיים הגיעו מ-dict שנקרא locals, מעתה מ-dict הגיעו globals, אז נראה שכל האובייקטים האלה גם צריכים להיות מוגדרים באיזשהו dict. זה dict הנקרא __builtins__ והוא מכנס אוטומטית לכל global-namespace. בכל פעם שמודול מיובא.

כשאנחנו כותבים שם כלשהו, Python ממחפש אותו ב-locals. אם היא לא מוצאת היא עוברת לחפש ב-globals. אם היא לא מוצאת את השם שchipשנו גם שם, היא עוברת לחפש ב-__builtins__, ואם גם שם לא נמצא אז יזרק .NameError.

הו מודול לכל דבר, ונוכל להתסכל על כל השמות שמוגדרים בו:

```
>>> dir(__builtins__)  
['ArithmetError', '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']
```

שימוש לב שאתם כבר צריכים להכיר חלק לא קטן מהאובייקטים שיש כאן, ועכשו 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 לבדוק האם רגע אנחנו רצים עם `locals`-ים או `globals` שמתאים לקטע הקוד שמורץ כרגע. הרי לא נרצה להיות בתוך הפונקציה `x` עם משתנים מקומיים של הפונקציה `u`...

ב חלק מזמן אין משמעות למשתנים מקומיים, למשל כאשרנו מרים קוד ב-`global-namespace`, דוגמה להרצת קוד ב-`mod1`.
CONST=17 הינה הוראה CONST=17 ב-`global-namespace` בנקודת כללה, () מוחזר את ה-`global-namespace`, ככלומר אם נריץ ב-`interpreter`:

```
>>> locals() is globals()
True
```

נוכל לראות שלפעמים המשתנים המקומיים והגLOBליים הם בדיקו 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
```

הוא אמן משתנה גלובלי, אבל כשאנו מרים את הוראה `CONST = 18` נוצר משתנה מקומי ב-(`f()`) עם הערך 18, וברגע שאנו יוצאים מהפונקציה המשנה מושמד ו-`CONST` הגלובלי נשאר כמו שהוא.

אם נרצה לשנות את `CONST`, נצטרך להגיד `CONST`-ה-`Python` קיים, ואת זה עושים בעזרת הצהרת `global`:

```
>>> def f():
...     global CONST
...     CONST = 18
...
>>> f()
>>> CONST
18
```

שימוש לב ש-`global` היה statement ולא נוכל להציב בה ערך ל-`CONST`. `CONST` בסה"כ אומרת ל-`Python` את המשנה זהה תקחי מ-(`locals()` ולא מ-(`globals()`).

כמו כן, נוכל ליצור הצהרת `global` למשתנים שעוד לא קיימים וכך לאתחל אותם (הרי `Python` לא ידעה אם `CONST` קיים שככתבנו `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

```

AIR import עובד

שאנו עושים import לモודול 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 נכתבו שני מודולים באופן הבא:

m1.py	m2.py
<pre> import m2 def f1(a, b): return m2.f2(a) + m2.f2(b) </pre>	<pre> import m1 def f2(x): if x < 0: return 0 return f1(x - 1, x - 1) + 7 </pre>

האם נוכל לעשות import ל-m1 או ל-m2 בלי שנקבל exception? כן. הסיבה לכך היא ש-Python זוכרת את המודולים שהיा עשתה להם import במקומות מרכזי שנקרו .sys.modules.

את sys כבר פגשנו קודם, וראינו שהוא סיפק לנו מידע על ה-exception הנוכחי שבטיפול. למעשה, sys מספק לנו הרבה אחרים שcoli משפטים על ההתקנות של Python עצמה. sys הוא המודול שמייצג את "מערכת ה-attributes" שאנו רצימ עליה. במקרה שלנו, sys.modules הוא מילון שמכיל מיפוי בין שם מודול לאובייקט המודול שמננו המודול יoba.

sys.modules

שאנו עושים import, Python לא מחרת לחפש קובץ ששמם את המודול שלנו. קודם כל היא משתמשת ב-sys.modules כדי לראות אם המודול כבר יoba בעבר. אם כן, היא פשוט מחזירה לנו את המודול. אם לא, היא מ Chapman 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.alternatives', 'exceptions', 'sre_parse', 'os', '__weakref__']

```

כאשר Python מוצאת את הקובץ שרצינו לעשות לו `import`, היא מייצרת אובייקט מודול ריק ב-`sys.modules` ומשיכת בתהlixir ה-`import`. בשיטה זו, אם פקודות שנריצ' בזמן ה-`import` יגרמו לפניה מעגלית למודול שלנו (כמו במקרה של `m1` שמייבא את `m2` שמייבא את `m1` בחרה), ה-`import` השני יוחזר אובייקט המודול שכבר ייצרנו ב-`Python` ו-`sys.modules` לא תצטרך לעשות דבר מעבר לכך.

למעשה, בזמן ש-`Python` מבצעת את ה-`import` הפנימי, אין לה מושג שהוא במהלך import אחר, כי היא מצאה את המודול ב-`sys.modules` וזה כל מה שחשוב במהלך import מבחינת השפה.

[sys.path](#)

נקודה נוספת היא אם ה-`import` היא איפה Python מוצאת את המודולים שביקשנו לעשות להם `import`. מודולים כמו `mod` ו-`example` שכתבנו במהלך הפרויקט הם מקרה ייחודי פשוט, כי שמו אותו בספריה מסוימת והרצינו את `os` מהספריה ה-`os`. אבל מה עם `sys` או `os`, או מודולים אחרים שנראות בהמשך כמו `itertools`?
ב-`sys` יש 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 מחוץ לארון ריקה, והוא מסמנת את הספריה הנוכחית. אם נרצה, נוכל להוריד את האיבר זה מ-`sys.path` ואז לא יתבצע import מהספריה הנוכחית.

למעשה, נוכל לעורר את `sys.path` בזמן הריצה של Python כדי לאפשר לה למצוא ספריות מקומיות נוספות שבהם היא לא ידעה שעלייה לחפש. דוגמה ניצור ספריה בשם `lib` בספריה הנוכחית שבה אנחנו רצים ונעביר אליה את `m1.py` ואת `:m2.py`. כתע ננסה לייבא את `m1` ו-`m2`:

* שימוש 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 מרכיבים Windows).

```
>>> import m1, m2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named m1
```

ברור שלא הצלחנו. כתה נוסיף את `lib` ל-`sys.path`:

```
>>> import sys  
>>> sys.path.append('lib')  
>>> import m1, m2
```

כלי ידיים מוכן להימנע מכך ולהשתמש ב-packages, עליהם נלמד בהמשך הפרק הזה.

import *

ראינו מוקדם שאפשר להשתמש `import` לשם של מודול או ל-`attributes` מתוך המודול. אם נסתכל על דוגמה בדרך השנייה מביניהן:

```
>>> from sys import path  
>>> from sys import modules
```

נראה שעשינו import ל-path ול-modules מ-sys ל-local-namespace שלנו. עכשו כשרצח לפנות ל-sys.path לא נדרש לרשום "sys.path" אלא נוכל להסתפק ב-path. כמו כן, בשיטה זו 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
```

ארח שobox לזכור שבשתי שיטות ה-`import` השימוש ב-`sys.modules` תמייד זהה, כלומר ב-`sys.modules` יוצר בסוף המקרה שלנו מופע של `sys`, ושיטת ה-`import` השנייה היא בסה"כ שיטה קצר יותר לנוכח ליבא `attributes` שנשימוש בהרבה וולכן לא נרצה לציין בכל שימוש שלהם את שם המודול מןו הם באו.

* import, והוא נראה ככה:

```
>>> from sys import *
```

מה שעשינו הרגע הוא להגיד ל-Python "קחי כל מה שיש ב-sys ותשיימי אותו ב-namespace שלי". זה לא סבבה, כי אין לנו מושג מה יש ב-sys. אנחנו מכירים את path,modules,info,exc_* אבל יש שם עוד הרבה דברים שאנו לא צריכים, והרגע זיהמנו את ה-namespace שלו.

יותר מכך, אנחנו יכולים לחשב שהכל בסדר ולעשות * import לעוד מודול:

```
>>> from sys import *
>>> from os import *
```

כשגם ב-os וגם ב-sys יש attribute בשם path, וכצפוי sys.path וain לנו מושג שזה קרה.

לכן, נעדיף להימנע מקוד שערשה *.import

אבל, אם בכלל זאת נרצה לאפשר למי שימושה במודול שאנו כותבים לעשות * import, נרצה לציין את השמות שהמשמש יעשה להם import, ואת זה נעשה באמצעות __all__. אם ניקח לדוגמה את m1 ו-m2 שראינו קודם, יוכל להווסף להם את __all__ כדי שאם מישו יעשה * import לאחד מהם הוא יקבל רק את הפונקציות שתכתבו ולא את המודולים שיש ב-namespace-ים שלהם:

m1.py	m2.py
<pre>__all__ = ['f1'] import m2 def f1(a, b): return m2.f2(a) + m2.f2(b)</pre>	<pre>__all__ = ['f2'] import m1 def f2(x): if x < 0: return 0 return f1(x - 1, x - 1) + 7</pre>

ובעת נרץ 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 מופיע מ-m1) ולא את m1 (שהיה מופיע מ-m2) אלא רק את ה-attributes שכתב המודול החלייט שנדאי שניבא.

רק לצורך הדוגמה, כך זה יהיה 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 interpreter. אחרי שהבנו שאנחנו לא יכולים לעשות import פונקימים (כי Python משתמש בעותק שב-sys.modules ולא תסתכם על הקובץ המעודכן שלנו), צריכה להיות דרך reload Python לטעינה מחדש מהקובץ המקורי. הדרך זאת נקראת import.reload.

reload היא פונקציה שמקבלת מודול ומבצעת לו טעינה מחדש מהקובץ המקורי במקור:

```
>>> reload(m1)
<module 'm1' from 'm1.pyc'>
```

reload גם מוחירה את האובייקט של המודול, אך אין בו צורך לאחר import קורה על האובייקט המקורי של המודול. הסיבה לכך היא שב-Python הרבה יותר חשוב שוותק אחד מכל מודול מאשר שוותק אחד מכל מודול אחר. אםreload מפעיל import-again על המודול ה"חדש" כי כל מה שהוא עשה הוא לקרוא את הקובץ ולהריץ אותו על גבי אובייקט המודול שכבר היה לה. לדוגמה, אם נכתבת את m1 מחדש (אבל נשאיר את m2 כמו שהוא בזיכרון אחר import reload):

m1.py	m2.py
<pre>import m2 def totally_new_func(x): return x + 1</pre>	<pre>import m1 def f2(x): if x < 0: return 0 return f1(x - 1, x - 1) + 7</pre>

או אחרי reload, m2.f2 תרוץ בלי שום בעיה כי המודול m1 יוכל גם את f1 וגם את totally_new_func. נכון, מומלץ מאוד לא להשתמש reload אלא אם אנחנו מודעים להתנהגות הזו ויודעים שרק עדכנו attributes ולא מחקנו attribute בזורה שתשפיע על ההתנהגות של הקוד שלנו.

סקריפטים

אפשרה לנו רק ליצור מודולים וליבא אותם, אלא גם להריץ קבצי Python בתור תוכניות. הדרך פשוטה ביותר ליצור תוכנית Python היא ליצור קובץ עם סימנת # ולהריץ אותו באמצעות ה-command-line של Python (ב-Linux). פשטן מקישים python ובס-Windows נוצרת נציג את ה-path, שהוא בד"כ Python.exe (C:\Python27\Python.exe).

שאנו כותבים קובץ Python שכוכונתו להריץ אנחנו קוראים לו скריפט (script).

נכתב לדוגמה את הסkrיפט 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)
```

ונרץ 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 (שmailto:הינה ב-\$) היא שורת הפקודה שהקלדנו בלינוקס כדי להריץ את הסkrיפט שלנו.

במקרה זהה, הסkrיפט היה אוסף פקודות ש-mailto:Python הריצה אחת אחרי השניה. אבל אם נרצה, יוכל לכתוב קוד קצר 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()
```

הבדל בין הגרסאות של yo_dog.py הוא שעכשיו הוא קובץ Python שמכיל פונקציה אחת בשם main() שעשו את העבודה של dog_dog.py. ברגע שאנחנו מרים את yo_dog.py, Python מגדירה את הפונקציה () main וرك בשורה الأخيرة מריצה אותה.

בשיטה זו היינו יכולים לעשות import yo_dog import main ולקראת main כמה פעמים שהיינו רוצים. יש רק בעיה אחת, והוא שם נעשה import yo_dog נגרום להריצה 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 שnamedו יורץ בתור סkrיפט, אבל גם לדאג שהמודול שלנו לא ירץ את main אם עשו לו import. ככה מי שיעשה import למודול יוכל לקרוא ל-main כמה פעמים שהוא ירצה ואיפה בקוד שהוא ירצה (ולא להזכיר אותו להכניס את השמות של הכלבים והחתולים שלו בזמן import.).

בשביל לעשות את זה, Python עושה דבר מאד נחמד: לשמורים 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 נרץ 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
>>>
```

הרכבת קבצי ב-Windows

ב-Windows קיימים כמה דברים נוספים שצדאי לדעת לשמורים קבצי `py`.

דבר ראשון הוא האם נרצה להריץ תוכניות `Python.exe` מ-`cmd.exe`, תמיד נצטרך לכתוב את ה-`path` לפני שם הסקריפט, לדוגמה:

```
C:\> C:\Python27\Python.exe yo_dog.py
```

אם אנחנו משתמשים ב-`Windows` הרבה, נוכל להוסיף את `C:\Python27` ל-\$PATH\$-הו דרך `Control Panel`, ואז נצטרך להקליד רק `yo_dog.py` לפני הסקריפט.

בנוסף, כאשר אנחנו מתקינים את `Python` על `Windows`, היא דואגת לרשום את עצמה עבור קבצי `py` כך שם נקליד פעמיים על קובץ `py` ייפתח חלון `cmd` ובו התוכנית שלנו תורץ. החסרון של השיטה זו הוא שאם יקופץ exception החלון ייסגר מיד ולא נוכל לראות אותו.

הרכבת קבצי ב-UNIX

הרכבת סקריפטי פירטו ב-UNIX דומה לריצה שלהם ב-Windows, הבדל אחד קטן – אנחנו יכולים לציין את ה-`path` ל-`executable` של Python בסקריפט עצמו, ואחריו שנסמן אותו כ-`cmd`-executable נוכל להריץ אותו כמו תוכנית רגילה.

למעשה, אחרי שהקובץ הוא executable אין גם צורך בסימנת `yc` ונוצרה להשאיר אותה רק אם נרצה שהמודול יוכל לעשות `import` נוספת בנוסף להיווטו סקריפט. נעדכן את `yo_dog.py` כך שירוץ מ-Linux-m-x86:

```
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()
```

נփוך 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 שאנו יכולים לעשות בפייתון הוא להעביר לסקריפט פרמטרים בשורת הפקודה, דוגמה לתוכנית בהרבה שפות אחרות. כל פרמטר שנעביר ייכנס ל-`sys.argv`, ונוכל לגשת לו-`sys.argv` כדי לראות את הפרמטרים האלה. לדוגמה, נכתבת תוכנית קצרה שמדפיסה את הפרמטרים שהיא מקבלת:

```
prints_args.py
#!/usr/bin/python
import sys
print sys.argv
```

ונריצ'ן:

```
$ ./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, נמשח תוכנית שמעתיקה קובץ ממוקם למקום אחר. התוכנית שלנו תקבל כפרמטר הראשון את הקובץ שצורך להעתיק וכפמטר שני את שם הקובץ החדש שצורך 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 זהה יש 3 דברים חדשים:

- פונקציה שימושית בשם `os.path.exists` מקבלת path ומחזירה True רק אם קיים קובץ זהה.
- כאשר אנחנו זורקים SystemExit התוכנית שלנו יוצאת. את SystemExit יוצרים עם ערך ההחזרה של התוכנית שלנו (ערך ההחזרה הוא הערך שמערכת הפעלה מקבלת מהתוכנית שלנו. הערך 0 אומר הצלחה וכל ערך אחד הוא כשלון).
- כדי לבדוק את `sys.argv` בדקנו את האורך שלו.

לגביה הנΚודה الأخيرة, בדיקת האורך של `sys.argv` היא טכניקה שמקובלת בשפות עתיקות כמו C, בהן אין רישימות ולכן מקבלים את `argv` ואת כמות הפרמטרים שהתוכנית שלנו קיבלה.

마חר ש-Python קצת יותר משוכלلت יוכל לכתוב את הקוד שלנו בצורה קצרה יותר קרייה באמצעות `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()

```

הזכורת: אפשר לעשות השמה של כמה משתנים באותו השורה, וכן השמה של משתנים מ-list/tuple/list למשתנים ספציפיים.
במקרה שלנו פרקנו את sys.argv לשולש משתנים בשם dest_file, exe ו-source_file. אם ב-argv[0] היה יותר או פחות מ-3 איברים, ראיינו מקבלים ValueError, ולכן אנחנו תופסים את ה-exception הזה ומניחים שאם הוא קפץ אז ה-command line לא הייתה תקינה.

בפרק הבא נכיר את המודול argparse שעוטף עבורנו את sys.argv ומאפשר לתוכניות שלנו להיות הרבה יותר נוחות לשימוש.

Packages

במהלך הפרק ראיינו שאחננו יכולות להפריד את הקוד שלנו לקבצים, זהה בפני עצמו דבר חשוב מאוד. בנוסף, ראיינו שאחננו יכולות לאחסן את הקבצים האלה בספריות שונות ולהוסיף path-ים ל-path.sys כדי ש-`sys` תמצא את המודולים שלנו.

אבל כאן נוצרות כמה בעיות:

- אנחנו לא רוצים לעורר את sys.path כי לא תמיד נדע איפה בדיק נמצא הקוד שלנו. גם אם ננסה למצוא את המיקום של הקוד שלנו בזיכרון דינמיית בעוזרת `__file__`, זה לא מבטיח לנו שהתווצה תהיה נכונה כי תמיד קיימים סיכומי שימושו ישנה את ספריית העבודה הנוכחיית.
- סיבה נוספת לא לשנות את sys.path היא שאחרי לא הרבה import-ים של כמה ספריות נסימן עם sys.path שמקיל הרבה "זבל" שאין לנו צורך בו יותר.
- שינוי של sys.path לא נותן פתרון למצarraה שבו שני מתכנים שונים כתובים שתי ספריות, ובשתייהן קיימים מודול באותו שם.

בגלל הבעיות האלה הרחיבו ב-Python את קונספט המודולים ויצרו Package-ים. بغداد, Package היא ספרייה המכילה קבצי Python. מה שמיוחד בספריה זו הוא שהיא מכילה קובץ בשם `__init__.py`.

כדוגמה, ניצור ספריה בשם `pkg` ובתוכה `pkg` ניצור קובץ ריק בשם `__init__.py`:

```
$ find pkg/  
pkg/  
pkg/  init  .py
```

cut ניצור את המודול הבא בתור pkg:

pkg/mod.py

```
def func1(x):  
    return x + 1  
  
def func2(y):  
    return func1(y) * 2
```

עכשו,倘若我们在Python中安装了名为pkg的包，我们就可以在命令行中输入“`import pkg`”来使用它。

```
>>> import pkg
>>> dir(pkg)
['__builtins__', '__doc__', '__file__', '__name__', '__package__', '__path__']
```

מזהר... ב-`pkg` אין כלום חוץ מהדברים הרגילים שיש במודול. ככלומר `pkg` הוא בסה"כ מודול ריק. כתעת נעשה `import :pkg.mod`

```
>>> import pkg.mod  
>>> dir(pkg.mod)  
['__builtins__', '__doc__', '__file__', '__name__', '__package__', 'func1', 'func2']
```

כלומר, כשאנו מיצרים מודולים בתחום `package`, אנחנו צריכים לעשות להם import בצורה מפורשת. הסיבה לכך היא שיש ב-`Python` כמה `package`-ים מאוד גדולים שב"כ נרצה לעשות import רק חלק מאוד קטן מהם ולכן 가능ה היא import מפושטת לנו לבחור דינית למה נרצה לעשות.

-package, אנחנו רואים ש-package מאפשר לנו ליצור היררכיה של מודולים. אפשר גם לשימוש package-ים בתוך package-ים וכן נוכל את הקוד שלנו בצוות שיהיה יותר נוח לשימוש.

ב-`mod` שirieז תמיד `csha-package` יעבור `import`. לדוגמה, נסנה את `py.../pkg` כך שיכיל `import package`

pkg/_init_.py

```
import mod
```

כעת נריצ' מחדש את ה-*interpreter*

```
>>> import pkg
>>> dir(pkg)
['__builtins__', '__doc__', '__file__', '__name__', '__package__', '__path__', 'mod']
```

זודר שימושית לחסוך משתמשים ב-`import package` שלנו `import`ים שוחזרים על עצם (במקרה שלנו `mod` הוא מאוד שימושי ולכו נרצה למסור מהמשתמש לעשות `import` ספציפי בכל פעם ל-`mod`).

כמו כן, ה-`import` ב-`py.mod` הוא ייחסי למספריה בה הקבצים האלה נמצאים, ולכן המודולים ב-`pkg` לא ארכיכים לדעת איפה הם נמצאים פיזית. הם יכולים להיות עשויים `import` אחד לשני ע"י שימוש ישירות בשם המודול.

from pkg import X

בדומה ל-importים הראשוניים שראינו, גם כאן נוכל לעשות import מתוך package:

```
>>> from pkg import mod
```

ואם נרצה נוכל גם ליבא שירות attribute של מודול מתוך ה-package:

```
>>> from pkg.mod import func2
```

sys.modules ב-Package

לפנינו סעיפים אמרנו שב-`sys.modules` נוכל למצוא מיופיע בין שמות מודולים למודולים. אז עכשו כשהוספנו `sub-package`ים לכל החגיגה הזאת, איך Python יודעת להבדיל בין `package` למודול (נגיד במקרה של `pkg` שהוא מודול לעומת `pkg` שהוא `package`?)

התשובה היא שאין הבדל. אחרי ש-import קורה, הוא בסה"כ מודול, והוא מופיע ב-`sys.modules` כמודול רגיל לחוטין. ההבדל הוא שמודולים שנמצאים בתחום `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 צזה 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 עכשו עשינו import רק לשם מפורטים, ככלומר:

```
>>> import pkg.mod
>>> from pkg import mod
```

לא משנה מה עשינו, תמיד היינו צריכים להגיד `Python` את כל המסלול עד למה שרצינו ליבא.

כל עוד אנחנו משתמשים ב-`package`, אין לנו הרבה ברירה, כי אם לא נבדיל בין `package`-ים ומודולים אז הפסדנו את אחד הדברים שרצינו להשתמש בהם המשמש ב-`package`-ים.

אבל נניח שדברים קצת מסתובבים וקיים 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
```

בפועל, 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 בתחום `x` של `sub1`. אבל מה יקרה אם נרצה לעשות `import` מתוך `sub2.x` לפונקציות מתוך `sub1`? נצטרך בלית ברירה לכתוב את זה:

```
pkg/sub2/x.py  
import pkg.sub1.x
```

אבל לא נרצה ש-`sub2` יידע שהוא ח' בתוך `pkg` (אולי נשנה את `pkg` לשם אחר בעתיד?). לכן, כדאי לבצע `:relative-import` מסוג אחר שנקרא `:relative-import`:

```
pkg/sub2/x.py  
from ..sub1 import x
```

בנוסף, נוכל לעשות `relative-imports` גם בין מודולים באוטו `: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 בדיקת imports-הו ציריך להתבצע. אם לדוגמה מחקנו את המודול `y` אבל `sys.path` קיים במקום אחר שאפשר למצוא בו-`sub1`.sys.path נקבל `ImportError` ולא נבצע `import` למודול לא נכון.

בנוסף, `relative-imports` מגינים علينا מפני `import` מחוץ ל-`package` בו אנחנו נמצא (`u"y"` כמובן). ננסה לעשות `import` כזה:

```
pkg/sub2/y.py  
from .....non_existent import non_existent
```

ונרץ:

```
>>> 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 נפרדים

הगראסאות העדכניות של Python מסופקות עם המון (מש המון) מודולים לשימוש של המתכנתים בה. ידע והכרה של המודולים חשובים לא פחות מהיירותם עם השפה, בגלל שכמו ש-Python חוסכת זמן בתוכנות ומאפשרת כתיבה של תוכניות חכומות וקצורות בהרבה, כך גם הספריות שמסופקות אליה – הן כוללות הרבה Möglichות ופונקציות סטנדרטיות, וחוסכות זמן בבניה של כלים שכבר קיימים ומוכנים לשימוש. בנוסף, שימוש בספריות קיימות הוא יתרון כי סביר להניח שעוזר אנשים כבר מכירים את הספריות האלה ולכן יהיה קל יותר לתחזק קוד שתכתבו איתן, וגם די סביר שהיו בהן פחות פ煦וט באגים מוקד שתכתבו בעצמכם.

פרק זה לא מכסה את כל המודוליםקיימים בשפה, אלא כמה מודולים אותם נהוג ללמוד בתחילת הבדיקה עם מודולים חיצוניים, כי הם שימושיים ולא דורשים הרבה ידע קודם ב-Python.

מומלץ להשתמש באינטרנט שנמצא ב-<http://docs.python.org/library/> כדי להכיר את המודולים שמוצגים כאן לעומק, וכך להכיר מודולים שלא מתועדים בפרק זה.

os

את os פגשנו כבר וראינו שהוא מכיל פונקציות לתקשורת עם מערכת הפעלה. המודול לא תלוי במערכת הפעלה עלייה אנחנו רצים, ולכן אפשר לסמוך על כך שאם נכתב את הקוד שלנו במערכת הפעלה אחת וניקח אותו למערכת הפעלה אחרת הוא ימשיך לפעול כמו שצריך.

בנוסף ל-os קיימים המודולים nt ו-xios לתקשורת הפעלה Windows-NT ו-UNIX-ים למיניהם ועל המודולים האלה לא נרחיב כאן.

כדי להשתמש במודול וכתו:

```
>>> import os
```

ב-os נוכל למצוא את ה-attribute `os.environ` שהוא מילון המכיל את משתני הסביבה של המערכת (כמו המיקום של ספריית-temp). משתני הסביבה משתנים מערכית, אך ניתן לראות את כל המשתנים הללו ע"י קרייה 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']
```

בעיר נוכל למצוא ב-os פונקציות לטיפול בתהליכיים וקבצים, והן מאוד בסיסיות ולכן קשה להשתמש בהן. יש ב-Python הרבה ספריות שעוטפות את הפונקציות ב-os כדי שלא נדרש להתעסק עם הפונקציות האלה, אך כן כדאי להכיר כמה מהן:

- `os.mkdir` מאפשר לנו ליצור ספירה.
- `os.rmdir`מוחקת ספירה ריקה (נקבל OSError אם בספריה יש עוד קבצים).
- `os.unlink` היא אורה הפונקציה כמו `os.remove`.
- `os.getcwd` מוחירה את הספריה הנוכחיית בה אנחנו רצים (שם לדוגמה נוצרים קבצים当我们 פותחים קובץ עם file ולא מצינמים ספריה ספציפית).
- `os.chdir` משנה את הספריה הנוכחיית.

sys

המודול `sys` מכיל את כל הֆונקציות הקשורות למערכת-h-`Python` ולהרצת תוכנית-h-`Python`.

- אובייקטי `file` שמייצגים את הקלט לתוכנית:
 - `sys.stdin` הוא קובץ שמייצג את הקלט לתוכנית. אם המשתמש עובד על terminal, הקובץ זהה יכול את הקלדות של המשמש, ואם הפעילו את התוכנית שלנו עם קלט מקובץ, `sys.stdin` יקרא מהקובץ זהה.
לדוגמה, `(raw_input()` משתמשת ב-`sys.stdin` כדי לקרוא קלט.
 - `sys.stdout` הוא קובץ שמייצג את הפלט לתוכנית. בדומה ל-`stdin`, זה יכול להיות טרמינל או קובץ.
`print` מדפסה ל-`sys.stdout` אם לא אמרנו לה כתוב אף 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 רוצחים להשתמש שайינו מתכנת ראה.
מידע על גרסה-h-`Python` עליה אנחנו רצימם כרגע.
- `sys.version` מכיל את גרסה-h-`Python` בצורת מחרוזת (לא כזה שימושי, אבל יפה להדפסה).
- `sys.version_info` הוא אובייקט שאפשר לקבל ממנו מידע על האגרסה שלנו.
`sys.platform` מכיל מחרוזת עם שם מערכת-ההפעלה עליה אנחנו רצימם.
שכיסינו בפרקם קודמים:

 - `sys.path`
 - `sys.modules`
 - `sys.argv`
 - `sys.exc_info()`

כמו כן, `sys` מכיל attributes לשימושים יותר מתקדמים, כמו:

- `sys.setprofile()`-`sys.getprofile()` שמאפשרות להגיד ל-`Python` להריץ פונקציה מסוימת בכל פעם שפונקציה פיתונית נקראת (זה משמש ליצירת Profiler שהוא כלי למדידת ביצועים).
- `sys.getrefcount()` שמאפשר לקבל את ה-`reference-count` לאובייקט מסוים.

os.path

במהלך החוברת התעסוקנו עם קבצים. לדוגמה, פתחנו קבצים עם `file` וגם עשינו `import file` אחרי שהוספנו איברים ל-`sys.path`. בcoliום היינו צריכים ליצור איזושהי צורה `path` לקובץ, ועד עכשיו נמנענו לעסוק בנושא בצורה יסודית.

המודול `os` (שמגייע עם `os`, אין צורך לטעות לו `import os` בנפרד) מכיל פונקציות לטיפול ב-`path`-ים לקבצים:

- `os.path.sep` מכיל את התו שמספריד בין שמות ספוריות ב-`path` (ב-`Unix` זה "/" וב-`Windows` זה "\").
- `os.path.pardir` מכיל את שם הספרייה שמצויה בספרייה שמעל הספרייה הנוכחית,
- `os.path.curdir` מכיל את שם הספרייה של הספרייה הנוכחית;

```
>>> os.path.sep  
'/'  
>>> os.path.pardir>>> os.path.curdir  
'.'
```

- `os.path.basename` מוחזיר את שם הספרייה בה קובץ מסויים נמצא, ו-`os.path.dirname` מוחזירה את שם הקובץ בלי הספרייה:

```
>>> os.path.dirname('/usr/local/lib/python2.7/re.py')  
'/usr/local/lib/python2.7'  
>>> os.path.basename('/etc/passwd')  
'passwd'
```

- `os.path.exists` מוחזיר האם קובץ מסויים 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` מפרצלת שם קובץ לשם הבסיסי שלו ולסיומת (שימוש לב שהפונקציה פועלת על מחרוזת ולא מודדת 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`. היא מאפשרת לנו להעביר כמה פרמטרים שנרצתה, 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` מוחזירה את ה"ספריה האמיתית" בהינתן path מסוים. לדוגמה, הפונקציה פותחת `link`-ים וספריות רלוונטיות (כמו ..) ומוחזירה path אבסולוטי לקובץ שביקשנו. את הדוגמה הזו הרצינו מהספרייה `:/home/demo`

```
>>> os.path.realpath('../x/../y/blah.py')  
'/home/y/blah.py'
```

- `os.path.isdir` מוחזירה האם path מסוים הוא ספריה או קובץ:

```
>>> os.path.isdir('/etc')
True
>>> os.path.isdir('/etc/passwd')
False
```

חשוב לציין שהחזק האמתי 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 סדרה כמו המודול שלנו, פשוט נכתב:

```
os.path.join(os.path.dirname(__file__), 'settings.json')
```

math

מודול זה מכיל מספר פונקציות שימושיות וקובעים לשימושים מתמטיים, כמו חישובים עם π או e , או משחקים שונים:
עם `float`:

- `math.pi` וה-`e` הם קבועים המייצגים את π ואת e בדיקות של 11 ספרות אחרי הנקודה.
- `math.ceil` מעגלת מספר כלפי מעלה.
- `math.floor` מעגלת מספר כלפי מטה.
- `math.sin`, `math.cos`, `math.tan` ו-`math.atan` מבצעות סינוס, קוסינוס וטנגנס.
- `math.log` מחזירה לוגריתם של מספר.
- `math.sqrt` מחזירה שורש ריבועי של מספר.
- `math.log` מחזירה הופכית להן.

הרעין די ברור, ואפשר לעבור על `help` של המודול כדי לראות את כל הפונקציות שקיימות בו.

time

לא במתיע, 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 בבחוץ, עד לדיקות של 9 ספרות אחרי הנקודה:

```
>>> time.time()
1347987874.865145
```

- `time.sleep()` היא פונקציה שি�ינה 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 `epoch-time`) ומחזיר אותה אובייקט נוח לעבודה שמייצג את הזמן הנוכחי לפי ה-`Timezone` שמכoon במחשב שלו לנו ועובדים:

```
>>> 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)
```

• מיצרת מחרוזת להדפסה באמצעות `time.asctime()`

```
>>> time.asctime(time.localtime())
'Tue Sep 18 20:10:14 2012'
```

• `time.ctime()` מקבלת זמן בשניות מאז 1.1.1970 ומחזירה מחרוזת בדמות `ctime()`

```
>>> time.ctime(time.time())
'Tue Sep 18 20:10:14 2012'
```

datetime

בודאי 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 להchnerות כמו בני אדם, הוסיףו ב-`Python` את המודול `datetime` שמכיל אובייקטים לטיפול בשעות ותאריכים. קודם כל, המודול מכיל את האובייקט `datetime`, אותו אפשר לבנות עם `kwargs` כמו שאנו אהבים:

```
>>> 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`.datetime, נחשב את התאריך בעוד 10 ימים:

```
>>> datetime.datetime.now() + datetime.timedelta(days=10)
datetime.datetime(2012, 9, 28, 20, 48, 20, 847100)
```

הmmm... 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 זמן 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 הוסיף 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)  
93178940006750499899604500015573844787589464272727032012285100273604096876740428226872  
00561841044061957292153800338555106262722965616018939857691222764909387893226167186843  
17856051748207769003753055733180398850011408802557733970324003815051362801729872167438  
3393273301573260781126787809342590702932322L
```

struct

המודול `struct` מאפשר להרכיב ולפרק מבני נתונים ביןאריים כמו `header`-ים של פרוטוקול. ב-`struct` יש לתאר את המבנה הנתונים בתור מחרוזת, ובעזרת המחרוזת זו אפשר להרכיב (`pack`) או לפרק (`unpack`) רצץ 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'
```

סביר את מה שקרה כאן: המספר ב-`Python` הוא מספר סטמי, כלומר הוא "סתם 7". אין לו גודל, ולכן אם נרצה לשולח את ה-7 זהה לאנשיו (למישו אחר דרך כבל רשת, או סתם לשומר אותו לקובץ), נצטרך לקודד אותו באיזושהי צורה כך שנוכל לקרוא אותו אחר כך.

כמובן, יכולנו לפתוח קובץ ולשמור בו את הטקסט 7. אבל אז מספרים היו תופסים די הרבה מקום, ולכן קידוד ביןארי הוא הרבה יותר יעיל. נדגים את זה עבורי `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\xcf\x00'
```

שימוש לב שאג את 7 וגם את 7000000 אפשר לקודד למחוזת באורך 4 תווים. ה-א-ים שאנו רואים כאן הם הדרך של Python להציג תווים שאין להם تو דפיים, כמו 0x00 שמודפס בתווך 0xa.

כעת נוכל לשכלל קצת את הדוגמה ולאורך מחוזת שמכילה 4 מספרים (int-ים) ותו אחד (byte):

```
>>> struct.pack('IIIB', 1, 2, 4500, 9, 9)
'\x01\x00\x00\x02\x00\x00\x00\x94\x11\x00\x00\t\x00\x00\x00\t'
```

היתרון הוא שהוא-str שיצא לנו הוא יחסית קצר, וקובע באורך:

```
>>> len(_)
17
```

כעת, נוכל לנקח את מה שקיבלו ולפתחו אותו בחזרה ל-tuple:

```
>>> struct.unpack('IIIB',
'\x01\x00\x00\x02\x00\x00\x00\x94\x11\x00\x00\t\x00\x00\x00\t')
(1, 2, 4500, 9, 9)
```

שימוש חשוב לציין איך המידע מקודד (במקרה שלנו 'IIIB'), לאחרת Python לא תוכל לדעת איך לפתח את המוחוזת שהיא קיבלה. לדוגמה, בואו נכניס מוחוזת אחרת למחרוזת ונראה מה יקרה:

```
>>> struct.unpack('17B',
'\x01\x00\x00\x02\x00\x00\x00\x94\x11\x00\x00\t\x00\x00\x00\t')
(1, 0, 0, 0, 2, 0, 0, 148, 17, 0, 0, 9, 0, 0, 0, 9)
```

הפעם אמרנו שהמוחוזת שלנו היא קידוד של 17 מספרים באורך בית אחד. באופן כללי התיאור של המוחוזת יכול להיות מורכב מההתווים הבאים:

- c: تو דפיים, יכול ליצג לדוגמה את האות 'a'.
- d: 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-בית או 4 בתים ב-32-בית.
- L: מספר באורך 8 בתים בלי סימן ב-64-bit או 4 בתים ב-32-bit.
- f: מספר float באורך 4 בתים.
- d: מספר float באורך 8 בתים.

argparse

המודול argparse מאפשר לנו להשתמש ב-`argparse` בצורה מודרנית ומאוד ידידותית עבור מי שישתמש בתוכנית שלנו.

נתחיל בicode קצר: בתוכנית שמספקת ממוקד `command-line` מקובל להפריד בין שני סוגי פרמטרים שמקבלים בשורת הפקודה: **אופציות (options)** או **ארגומנטים (arguments)**. ההפרדה מאוד דומה ל-`*args` ו-`**kwargs` ב-`Python`, עם כמה הבדלים קלים.

מקובל לתת לכל אופציה בתוכנית שם קצר שמכיל את אחת, ושם ארוך שמכיל יותר או מספרים מופרדים בפסיקים. לדוגמה, לאופציה `cycles` ניתן את השם הקצר `-c` (מינוס `c`) ואת השם הארוך `cycles` (מינוס `cycles`). בדוגמה זו: 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
```

ומקבול להמיר בשני הפרמטרים (כלומר לקבל גם רווח וגם `=`).

בנוספּ, הארגומנטים הם כל הפרמטרים לתוכנית שאין להם קידומת יפה כמו `-c`, ואוֹתם אוספים בתור-`positional`. לדוגמה: 1, 2 ו-3 הם ארגומנטים לתוכנית שלנו: `arguments`

```
./prog.py 1 2 3
```

כמובן שאפשר יהיה גם לשלב בין אופציות לארגומנטים:

```
./prog.py 1 2 3 -c 7
```

cut נזכיר את argparse שחווסף מאייתנו את הצורך בתכננת הלוגיקה זהה 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 נרץ את `prog` בלי פרמטרים נקבל:

```
~$ ./prog.py
usage: prog.py [-h] source_file dest_file
prog.py: error: too few arguments
```

כלומר prog מנסה לקבל שני ארגומנטים בדיק, והוא קוראיהם 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 נויאל להעביר ל-prog את הארגומנטים שהוא ציפתה להם היא תמשיך לרצף קריגל (ובמקרה שלנו לא תעשה כלום ומיד תצא):

```
$ ./prog.py 1 2
$
```

נוכל לגעת לארגומנטים לפי השם שנתנו להם ב-args, ככלומר main().args.dest_file ו-args.source_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()
```

קיים לנו שתי אופציות שאפשר להפעיל את התוכנית בלבד, וגם הוספנו את הפרמטר help לכל קריאה ל-args.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 מצלפים. אם לדוגמה נרצה לקבל מספר נציג `int`. אם המשמש עביר פרמטר שאינו מספר התוכנית שלנו תציג לו שגיאה ולא ניתן לבדוק את זה בעצמו.
- אפשר להעביר ל-`add_argument` פרמטר בשם `choices` שמכיל `list` או `tuple` עם אפשרות לערכים חוקיים לפרמטר.
- ונכל להעביר פרמטר בשם `default` עם ערך התחלתי, במקרה שהמשמש לא יציגו 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()
```

עכשו `args.move` ו-`args.copy` יכילו `True` או `False` בהתאם להאם המשתמש העביר `--copy` או `--move` לתוכנית, והמשמש מצדדיו לא צריך להעביר 1 או אישרנו ערך אחריו `--copy` או `--move`.
זהו מבוא יחסית קצר למודול, וקיים עוד פיצרים יותר מתקדמים בתיעוד המלא של המודול.

subprocess

המודול `subprocess` מכיל אובייקט בשם `Popen` וכמה קבועים שמיד נראה מה תפקידם. האובייקט `Popen` מאפשר לנו להריץ תחילה, לתקשר איתו ולבסוף לראות האם וכיצד התחילה הסתiens.
נתחיל בדוגמה פשוטה – נריץ את הפקודה `ls` בתחילה דרך דרך `Python`:

```
>>> import subprocess
>>> result = subprocess.Popen('ls').wait()
Desktop Documents Downloads git Music Pictures Public Templates Videos
>>> result
0
```

מה שעשינו היה להריץ תחילה חדש ששורת הפקודה שלו הייתה "ls", חיכינו שהתחילה יסתiens, ושמרנו את ערך החזרה של התחילה במשתנה בשם `result`. שימו לב שהתחילה הדפיס את הפלט שלו למסך שבו אנחנו עובדים. ערך החזרה של התחילה הוא 0, וזה אומר שהוא הצלחה. רוב התחilibים מחזירים 0 כשם מצלחים, למעט כמה תהליכיים מיוחדים שלא נסקרו כאן.
עכשו היינו רוצים לשמור את הפלט של התוכנית שלנו במשתנה. הרוי כנדריך תוכנית כנראה שלא נרצה את ערך החזרה שלה אלא דוקא 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'

```

אובייקט `Popen` מכילם שלושה attributes מאוד חשובים – `stdout`, `stdin` ו-`stderr`. אלה אובייקטים דמויי `file` שמייצגים את הקלט, הפלט ו-stream השגיאות של התהיליך שהפעלנו. אם נרצה, נוכל כתוב ל-`stdin` של התהיליך ע"י-כך שנקרה ל-`proc.stdout.read()` וنוכל לקרוא את הפלט של התהיליך ע"י קראיה ל-`proc.stdout.read()` כמו שעשינו.

שימוש לב ש-`stdin`proc יהיה מוגדר רק העברנו `stdin=subprocess.PIPE` כפרמטר. אם לא נעביר את הפרמטר PIPE התוכנית תורץ בלי קלט, וכך לא נוכל להשתמש ב-`proc.stdin`.

חשוב לציין לב שכ"כ נוכל להריץ תהיליך ולקראא את ה-`stdout` שלו בלבד, אך אם התהיליך יכתוב ל-`stderr` ולא ל-`stdout` הוא עלול להתקע אם לא נקרה את הפלט שהוא כתב ל-`stderr`. במקרה זה נוכל לגרום לתוכנית שלנו להתנהג כמו בטרמינל רגיל, וכך נגיד לה לכתוב את ה-`l`-`stderr`:

```

>>> 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 עשו המתוודה (`wait`)? התפקיד של `wait` הוא לחכות עד שהטהיליך יסתתיים, וכשהוא מסתיים מקבל את ערך החזרה שלו ממערכת הפעלה. אם לא נקרה ל-`wait` התהיליך שלנו יישאר רץ במצב מיוחד שנקרא זומבי (zombie), כי מערכת הפעלה שומרת עבורה את כל התהיליכים שלא אספנו את ערך החזרה שלהם עד שנא Sof אוטם. לכן, תמיד נקרה ל-`wait` ונבדוק את ערך החזרה של התהיליך, כי הרו נוה חיבים לוודא שהוא הצליח.

בנוסף לקבלת הפלט של תוכנית, נוכל גם לשולח לה קלט. אם נציין ש-`stdin` הוא PIPE התוכנית שלנו תוכל גם לקבל קלט. הפעם נריץ את התוכנית `cat` שמדפסה ל-`stdout` כל מה שהוא 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` רצתה עד שהיא מקבלת EOF (End-Of-File) ולקמן היינו חייבים לקרוא ל-`proc.stdin.close()`, אחרת התוכנית לא הייתה רואה שהקלט שלה נגמר. כאשר סיימנו לחת לה קלט, קראננו את הפלט שלו וחיכינו שהיא תסתתיים.

אבל כאן יש עוד בעיה – מה היה קורה אם בזמן שהיינו מזינים קלט לתוכנית שלנו היא כבר הייתה כותבת יותר מדי פלט והייתה נתקעת? הרו אמרנו שהה יכול לקרוא...
...

בשביל מקרים שבהם אנחנו צריכים לתקשר עם התוכנית שלנו בצורה אינטראקטיבית ממש, قيمة המתוודה `communicate` שמקבלת מחרוזת קלט ודואגת לקרוא את `stdout` ו-`stderr` בנפרד עד שהתוכנית מסתיימת. אחרי ש-`communicate` תחזיר 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 מוחזירה tuple של `communicate` של `(stdout, stderr)`.

חלק 9: איטרטורים

בפרקם הראשון פגשנו פונקציה מאוד שימושית בשם `range`. מיד אחרי שפגשנו אותה הצגנו גם את `xrange`, והסבירנו את ההבדל ביןיהן – `range` היא פונקציה שמייצרת רשימה שמכילה את כל המספרים בתחום שביקשנו. לעומת זאת, `xrange` מייצרת אובייקט-דממה שלא מכיל את כל האיברים אלא זוכר את 3 הפרמטרים של הסדרה (`start`, `stop`, `step`) וודיע על החזר לנו בכל פעם את האיבר הבא.

ההיגיון די ברור – אין כמעט אף מקרה שבו נדרש את כל האיברים בזיכרון ברשימה אחת. לעיתים הרשימה הזאת יכולה להיות אפילו די גדולה, ולכן לא נרצה ליצור ולזכור את כל האיברים, אלא יהיה לנו הרבה יותר מהר לחשב את האיבור במקומם `x` מאשרctrkr אותו. `xrange` מספקת משהו הרבה יותר פשוט והוא רק את האיבר הבא בכל פעם. מסתבר שהוא מספיק טוב.

אם נחשוב קצת הלאה, רישימות של מספרים זה רק מקרה אחד שבו אנחנו צריכים עותק של מהו בזיכרון. נניח שיש לנו מיליון גדול בשם `d` ואנחנו קוראים `l(d.keys)`. המתודה `keys` של המילון תחזיר לנו רשימה שמכילה את כל המפתחות של המילון. לאחר רשימה היא `mutable` (כלומר אפשר לשנות אותה), הרשימה הזה היא עותק. שוב יזכירנו עותק של מהו שלא היו צריים. הרי לא יהיה כמעט מקרה שבו נדרש את כל המפתחות, וב吐ו נוכל להסתפק באובייקט שיחזר לנו בכל פעם את המפתח הבא (זה מספיק טוב בשבייל להdfs, בשבייל לחפש, ובשביל עוד הרבה MERCHANTABILITY).

איטרטורים וה-`Iterator protocol`

מהחר שנראה שהפתרון של `xrange` שבו יש אובייקט שמחזיר בכל פעם את האיבר הבא הוא די טוב, או כנראה שנרצה פתרון זהה עבור כל המקרים שבהם אנחנו צריכים לעבור על קבוצת איברים אחד אחריו השני.

הפתרון זהה נקרא איטרטור (`iterator`), ובתוור התחלת נסתכל איך עובד איטרטור במקרה של `xrange` ולאחר מכן נבין את המקרה הכללי. לצורך הדוגמה ניזור 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 מסווג `xrange` אין שום מתודה ואף `.attribute` למשה, כל מה שהאובייקט הזה יודע לעשות הוא למש槽 אחד מיוחד בשם `__iter__`, שכשהוא ממושך אנחנו יכולים לקרוא `l(iter(x))` על האובייקט שלנו (בדיקה כמו שאמנם משמש את `__len__` נוכל לקרוא `l(len(x))`). נקרא `l(x)` ונראה מה קיבל:

```
>>> 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 שייצרנו את `xrange` עם הפעמטר 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)` ולבדק שהוא 7, או שאולי `xrange` בכלל לא היה מմASH את `len` ולא היינו יכולים לקרוא ל-`len(x)`:

```
>>> y.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

כמו הגיוני – קיבלנו exception שאומר שהגענו לסוף הסדרה.

בצם, 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`.

התוצאות הזו שפה לאובייקט יש מתודת `next` שמתנהגת כמו שתיארנו עכשו נקראת `Iterator Protocol`, וכל אובייקט שקיים אותה יכול להיות איטרטור. למעשה, אנחנו יכולים למש איטרטורים בעצמנו ובמה שיראה דוגמה לכך.

iter()

אחריו שהבנו מה זה איטרטור, כדאי שנבין קצת יותר טוב איך `iter` מתנהגת בכלל מיני מצבים. בטור התחלת, ניצור לעצמנו רשימה וניציר איטרטור עליה:

```
>>> l = [1, 2, 3, 4, 5]
>>> x = iter(l)
```

דבר ראשון שאפשר לעשות עם איטרטור הוא לפתח אותו כרשימה או `tuple`. כמובן זה נראה מטופש כי הרি רצינו לבדוק את הפך, 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 משנהים, למשל `list` ומושם). האיטרטורים יתקדמו בנפרד ולא יהיה 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 שכשאנו מסיים להשתמש באיטרטור הוא לא "חזר להתחלה". אם לדוגמה ניצור איטרטור על הרשימה שלנו ונמיר אותו ל-tuple:

```
>>> x = iter(l)
>>> tuple(x)
(1, 2, 3, 4, 5)
```

לא יוכל להשתמש יותר ב-x. האיטרטור הזה "גמר", ובນיסוח יותר איטרטורי, הוא consumed. התופעה זו נקרא consumption, כלומר המצביע בו כבר סיימנו להשתמש באיטרטור:

```
>>> x.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

במשך נראה מקרים בהם נctract להיזהר מ-consumed iterators בלבד שאפיינו התכוונו להשתמש באיטרטורים.

דבר נוסף שחייב לידע על איטרטורים, והוא גם חלק מה-Iterator Protocol, הוא שאיטרטור חייב תמיד להסכים לספק איטרטור על עצמו, והוא חייב להיות עצמו. מבבל... 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 כש庫ראים ל-(iter) עליו. כמו כן, בדיק כmo שאיטרטור שומר על המצביע שלו, גם (x) iter ישמר על המצביע של האיטרטור. לכן, במקרים שבהם נחשוב שאנו מיצרים איטרטור של אובייקט מסוים, יכול מאוד להיות שבכל קיבלנו איטרטור ולן הקריאה שלנו ל-(iter) לא החזירה אובייקט חדש. במצב זה יכול להיות שנעשה בעיות consume לאיטרטור או שנקבל איטרטור שהוא כבר consumed.

במשך נראה איך להתמודד עם מצבים כאלה.

לסיום הסעיף זהה, חשוב שנקיר שלוש מתודות שימושיות של מילונים – `iterkeys()`, `itervalues()` ו-`iteritems()`. המתודות האלה מקבילות ל-`values()`, `keys()` 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 כדאי להשתמש בהן, אבל חשוב להזכיר הבדל קטן אחד בין המתודות המקוריות למתודות שemmמשות איטרטורים והוא שבזמן איטרציה על טיפוס נתוניים כמו מילון לא ניתן להוריד או להוסיף אליו איברים. הסיבה לכך היא שהאיטטור משתמש במבנה הנתוניים הפנימי של מילון וכך הוא מסתמן על כך שהמילון נשאר קבוע (כלומר שה-`keys` של מנתנים). 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

אייטרטורים די מזכירים שהוא שראינו בפרק הראשון – לולאת `for`. בת'כלם, כל העבודה של לולאת `for` היא לחת איזה אווביקט שמכיל אובייקטים אחרים ולבור עליהם אחד אחרי השני, ובכל פעם לאפשר לנו לעשות משהו עם האובייקט הנוכחי. ולאחר מכן איטרציה לכל דבר, גם לולאת `for` עצמה משתמש באיטרטורים.

בכל פעם שאנו עושים `for` על משהו, `for` מייצרת איטטור של האובייקט שהעבכנו אליו, קוראת ל-`next()` של האובייקט זהה ומזכירה 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 בדיקו 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 – `korat next()` כשהיא מתחילה לרווח.

בנוספ', לאחר שאנו יכולים ליצור איטטור של איטטור (וכמו שאמרנו יוחזר האיטטור עצמו), נוכל לעשות `for` על איטטור:

```
>>> l = [1, 2, 3, 4, 5]
>>> x = iter(l)
>>> for i in x:
...     print i
...
1
2
3
4
5
```

וכמובן, נוכל לשכפל 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

ישור נוסף שפגשנו בפרקם הראשונים היה **List Comprehensions**. זה היה **syntax** נוח שאיפשר לנו לאחד את היכולות **filter()** ו-**map()**. נזכיר איך זה נראה:

```
>>> [x * 2 for x in xrange(7) if x % 3 == 0]
[0, 6, 12]
```

אבל **list comprehensions** עושים בדיק את אותו העול של **range**, ולכן היינו רוצים גם "גרסת **xrange**" שלהם. הגרסה זו נקראת **Generator Expressions**:

```
>>> (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** (generator).

```

>>> 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 תכנית, ל-Generator Expressions יש עד יתרון חשוב על פני List Comprehensions. לדוגמה, נניח שאנו דוקא כן רוצים רשימה מפורשת בזיכרון, ואנו רוצים ליצור אותה בעזרה :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: "دولף" את משתנה הלולאה בדיק כmo לולאת for רגילה:

```

>>> num
9

```

כלומר, כשאנו משתמשים ב-List Comprehensions, תמיד יקרה מצב שבו משתנה הלולאה ידולף לתוך ה-namespace שלו. במקרה שבו לא השתמשו במשתנה זה עוד יחסית בסדר, אבל אם לא נשים לב יוכל להגיע למצבים שבהם שימוש תמים ב-List Comprehensions ידרוס לנו את אחד מהמשתנים בפונקציה.

כדי להתמודד עם התופעה זו נוכל להשתמש ב-Generator Expression, לאחר ש-Generator Expression מכילים כדי להתחום עם התופעה הזו נוכל להשתמש ב-tuple, כל מה שנוצרת הוא בעצם את הגנרטור שנוצר ב-namespace :list

```

>>> tuple((num / 3 for num in xrange(10)))
(0, 0, 0, 1, 1, 1, 2, 2, 2, 3)

```

כובן נוכל להשתמש ב-Generator Expressions גם בלוואות for ובתוור פרמטרים לפונקציות, כמו במקרה של tuple. שימוש לב שהריגע ראיינו שכשאנו קוראים לפונקציה שמסכימה לקבל איזשהו iterable נוכל גם להעביר לה גנרטור. אבל במקרה שבו אנחנו יוצרים גנרטור ע"י Generator Expression יוצא לנו צרכיים להעביר סוגרים מיותרים שברורו שאין בהם צורך. לכן, מבחינת Python אפשר להוריד את הסוגרים 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 שצדדי להכיר בהקשר זה הן any ו-all. שתיהן מקבלות איזשהו iterable True או False מחזירה any אם איזשהו איבר שהוא קיבלה לא מתרגם ל-False (כלומר לא אחד מ-0, "", {}, [] או set()) או False ו-all מחזירה True אם כל האיברים שבה לא מתרגם ל-False. לדוגמה, אם יש איזשהו מספר בין 0 ל-100 שמתחלק ?2-ב-

```
>>> 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.

שימוש לב-`sh-`Python מושך לא להיותה צריכה לעבור על כל המספרים בין 0 ל-100 בשני המקרים. במקרה הראשון הגנרטור שהעבכנו לה לבדוק את המספר 0, החזיר `True` ולכון `any` חזרה מיד אחרי הניסיון הראשון. לא היה צורך לבדוק את שאר האיברים.

במקרה של `all`, היא ראתה שהאיבר הראשון אכן מקיים את התנאי והמשיכה לאיבר הבא. אבל ברגע שעבכנו לבדוק את האיבר הבא קיבלנו `False` ולכון `all` מיד יוצאה, כי אין לה צורך לבדוק את שאר האיברים.

המודול `itertools`

אייטרטורים הם אובייקטים מאד שימושיים, ולכן קיימים מודול בשם `itertools` שמכיל פונקציות כדי שייהפכו את החיים שלנו עם אייטרטורים לעד יותר קלים. נדגים כאן כמה מהfonקציות שנמצאות בשימוש נפוץ:

`itertools.chain` מאפשרת לנו לשרשר כמה אייטרטורים. כאשר אנחנו משתמשים ברשימת, אפשר פשוט לחבר שני רישימות כדי לקבל רשימה אחת גדולה. אבל במקרה של אייטרטורים, אנחנו כמובן לא רוצחים לחבר אותם (כי אין צורך לבצע `consumption`) ולכון נדרש אובייקט שיעשה בשביבינו 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 אייטרטור ומחייב אייטרטור אינסופי (כחזשה-`next()` שלו לא זורק `StopIteration` אף פעם) ע"י-כך שהוא פשוט חוזר על האיברים שוב ושוב בולולאה. שימוש לב-`sh-`Python באילו, כי בתכלס הוא זוכר את האיברים ברשימה. לצערנו אין דרך (נסו לחשב על אייטרטור שמחזיר שלושה איברים אקראיים. יהיה מאוד קשה לחזור עליהם אם לא נשמר 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
```

. מתקבל מספר וסופר החל מהמספר זהה 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 מאייתנו שמתקדים להירדם, יוכל למש בקבוצות גנרטור שסופר כבושים:

```
>>> ('{} 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 הfonקציות שיש ב-`itertools` ב-`help` שלו.

גנרטורים

לפינותו, נכיר פיצ'ר שימושי ביותר ב-*Python*. עד עכשיו ראיינו לחברנו ל-*List Comprehensions* שמאפשר לנו לקבל גנרטור עם אותו *syntax*:

אבל, דבר נוסף ש-*Python* מאפשרת לנו לעשות הוא לכתוב פונקציה רגילה לחוטין, רק שבמקום שהפונקציה שלנו תחזיר ערך אחד, היא תוכל להחזיר כמה ערכים שתרצה, והיא תהיה עם אותו משך כמו של איטרטור. זה נראה כך:

```
>>> def f():
...     yield 1
...     yield 2
...     yield 3
```

fn קראת גנרטור. הסיבה שהיא נקראת כך היא שהשתמשנו במילה *yield* בתוך הפונקציה. ברגע שעשינו כזה דבר זאת לא פונקציה יותר אלא גנרטור. שמו לב שגם השתמש במילה *yield* בתוך פונקציה וננסה לעשות *return* לערך נקלט מיד :exceptions

```
>>> def bad_f():
...     yield 1
...     return 2
...
File "<stdin>", line 3
SyntaxError: 'return' with argument inside generator
```

אם נרצה, יוכל לצאת מהפונקציה ע"י *return* ריק, בלי שום ערך:

```
>>> def f():
...     yield 1
...     yield 2
...     yield 3
...     return
...     yield 4
```

כמובן שהפונקציה פשוט תסתהים ולעולם לא תגיד להרץ את 4 *yield*. עצת 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
```

בדוק כמו איטרטור – כשאנו קוראים לפונקציה, נוצר *instance* חדש של הגנרטור שלנו, ובכל קריאה ל-*next()* הקוד של האיטרטור רץ עד שהוא מגיע ל-*yield* הבא. בכל *yield* מוחזר הערך שננתנו ל-*next()* והגנרטור "נטקע" עד הקראיה הבאה ל-*next()*.

שימו לב שבדוק 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 שאינו מוגבלים רק לקוד פשוט. נוכל לעשות בתחום גנרטור כל דבר שהיינו עושים בתחום 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 מסתירים לעולם פשוט ע"י-כך שנממש בהם לפחות 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'

```

וכdogמה אחרתה, נראה גנרטור אינסופי שימושי שممמש 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
```