תכנות פונקציונלי בפייתון

מסוכם משיעורי מרן הרב

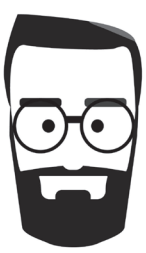
ד"ר משה גולדשטיין שליט"א

עם פירוש

"רי"ח טוב"

מאיתי הצב"י יוחנן חאיק

שנת 01011001111111



"כשם שאי אפשר לבר בלא תבן, כך אי אפשר לתוכנית בלי שגיאות של הבודק האוטומטי"

להערות, הארות ותיקונים:
yohananha@gmail.com
@yohanan - בטלגרם

ניתן להשתמש בסיכום באופן חופשי לכולם!!

תוכן עניינים

הכרת השפה פייתון[1]

פייתון היא שפת סקריפט – כלומר כותבים תרחישים מסוימים ואותם מעבירים לאינטרפרטר שיריץ אותם בסביבה המתאימה. בניגוד לשפות שנתקלנו בהם עד עכשיו, אותם כתבנו בסביבת עבודה הכוללת קומפיילר (ויז'ואל), לפייתון אין קומפיילר, אלא רק אינטפרטרים (מפרשים) שלוקחים את התוכנה הנכתבת ומריצים אותה, ורק אז אנחנו יכולים לגלות כל מיני טעויות שקיימות בקוד בזמן הריצה. אמנם יש היום כבר אינטפטרים מספיק משוכללים שיכולים לעבוד בצורה פשוטה יותר עם פייתון (PyCharm, VSCode), אך זה עדיין לא מספיק בשביל להוציא את השפה מההגדרה הזאת.

השפה נכתבה בשנות ה-90 כפיתוח של שפת ABC של גוגל, במטרה ליצור שפה שתהיה יותר נקיה וברורה, ו על מנת ליצור פשטות, נכנסו כל מיני אפשרויות לשפה שהן שונות ברעיון מהרבה שפות אחרת. על אף שהשפה היא מונחית עצמים, היכולות שלה שונות משפות אחרות בכך שאין בה כימוס (הכנסה של שדות/פונקציות תחת private), כך שהכל יהיה מונח על השולחן.

לתכנה יש כרגע שתי גרסאות פעילות – 2.7.X שהיא הגרסה הישנה יותר, וכבר לא בפיתוח מתמשך, וגרסה 3.7.3 שהיא הגרסה שאנו לומדים, לה יש יכולות קצת יותר גבוהות, ומקבלת עדכונים שוטפים. הסיבה שעברו גרסה, היא כי גרסה 2 הכילה יותר מידי פגמים, אך מאחר וכבר יותר מידי קוד היה מבוסס על הגרסה, והמעבר בין הגרסאות הוא לא ישיר, עדיין גרסה 2 קיימת.

השפה עצמה היא Open-Source וכל אחד יכול לכתוב לה מודולים ולפתח אותה, אך יש וועדה היושבת ומגדירה אילו שינויים יהיו רשמיים בשפה, ואילו לא.

הדבר החשוב ביותר לדעת לגבי השפה – שם השפה "פייתון" הוא מחווה לחבורת ההומור הבריטית "מונטי פייתון". ועכשיו למשהו שונה לגמרי-

שפת סקריפט

כמו שאמרנו קודם, בשפת פייתון, אין קישור של תכנית והידור של הטקסט הנכתב (מעבר על בדיקת הקוד עצמו), אלא פשוט ניתן לכתוב ולהריץ. כל סביבת עבודה מגיעה עם אינטרפטר משלה, המסוגלת להגיב לסקריפטים הנכתבים בה, ואף יש אפשרות להריץ פקודות ישירות מתוך האינטרפטר גם ללא תכנית ולקבל תשובות. דבר זה יעיל מאוד הן ללימוד השפה, והן לכך, שאין צורך תמיד לכתוב את כל התכנית, ואם יש חישוב קטן ניתן לעשות אותו בקלות.

תהליך תרגום השפה

אם ניקח תכנית בשפת C שכתבנו, ונרצה להריץ אותה, הקוד עובר מספר שלבים, שאנחנו מכירים באופן כללי (ויוסברו באופן מפורט בקורסים מתקדמים יותר), ובגדול ניתן להראות את המעבר מהקוד למכונה באופן הבא-

[1] בחלק הזה אנחנו לא נתמקד בתכנות הפונקציונלי, אלא רק בשפה עצמה. נראה בהמשך שפייתון היא לא שפה פונקציונלית בהגדרתה, אך יש בה פיצ'רים שאפשר לעבוד איתם בצורה פונקציונלית.
כאן אנחנו נתמקד רק בבסיס השפה ותו לא.

הקוד המקורי אותו אנחנו כותבים, עובר שלב עיבוד בקומפיילר, המועבר לשפת מכונה, שזו מעין שפה שמתארת את כל הפעולות הדרושות לביצוע, הקוד הזה הוא יחסית משותף, להרבה מכונות, כלומר אם כתבנו קוד בשפת C או בג'אווה, השלב הזה אמור להיראות פחות או יותר אותו דבר, כי כאן אמור להגיע בעצם רק שלב ביצוע הפעולות.

בשלב הבא, קוד הביניים נכנס לאסמבלר, שם הוא עובר המרה נוספת שהיא כבר תלוית מכונה, שם אנחנו מכניסים את כל הקצאת הזיכרון באופן פרטני, ובגדול מתעסקים יותר איך המחשב יגיב לקוד הנקלט.

לעומת זאת, בפייתון התמונה נראית שונה לגמרי –

כל ההמרות מתבצעות באופן שונה, למרות, שבסוף מדובר על ירידה לפרטי המכונה, אך הבייט קוד לכשעצמו הוא כבר יותר קרוב למכונה מאשר קוד הביניים הרגיל של שפת C.

ערכים בפייתון

אחד הדברים המיוחדים בשפה זו, הוא הרעיון שכל דבר המופיע בשפה הוא אובייקט. החל מהמשתנים הרגילים, המובנים בשפה, ועד לפונקציות מוגדרות במערכת או שנכתבות על ידי המשתמש.

נסקור את קבוצות הערכים השונות, ונדגיש לכל דבר את המיוחד בו בהמשך.

טיפוסי נתונים בסיסיים

bool – טיפוס המבטא משתנה בוליאני. תחת המשתנים האלה, אנחנו מכניסים את False ואת True, שמבטאים את התוצאה. (הערה: פייתון רגיש לקלט של אותיות גדולות וקטנות, כך ש-False יבטא 0 בוליאני, ו-false לא יבטא כלום)

<u>טיפוסים מספריים:</u>

int – מספר שלם.

long – מספר שלם ארוך.

float – מספר עשרוני.

complex – מספר מרוכב.

ערכים איטרביליים

תחת קבוצה זאת, נכנסים כל טיפוסי הנתונים, עליהם ניתן לעבור בצורה איטרטיבית. כלומר – לסרוק אותם באופן שוטף ולבצע עליהם פעולות שונות. (בניגוד למספר ארוך שאותו אנחנו קוראים כאחר, מחרוזת, למשל, יכולה להיקרא כאוסף אותיות).

str – מחרוזת (המבוטאת כאוסף של אותיות/סמלים).

<u>מערכים הטרוגניים (יוסבר בהמשך)</u>

tuple – רשומה.

list – רשימה.

dictionary – מילון.

set – קבוצה.

פונקציות

גם בפונקציות, לאחר שהגדרנו אותם, אנחנו מסוגלים להשתמש בהם בצורה של ערכים ולשלוח אותם כמו אובייקט לכל דבר ועניין, וכן לקבל אותן בחזרה מפונקציה אחרת. הפונקציות הקיימות בפייתן הן –

פונקציות מובנות במערכת.

פונקציות משתמש.

למבדא.

מתודות מובנות במערכת.

מתודות משתמש.

מחלקות

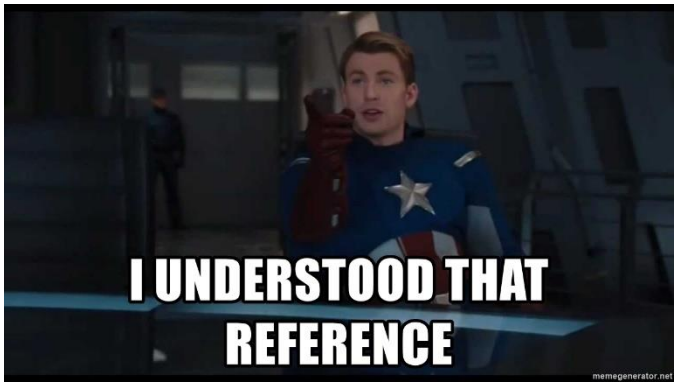
גם מחלקות וכל הנגזרת שלהם הם ערכים חוקיים בשפת פייתון.

משתנים בפייתון

כאשר אנחנו מגדירים משתנה בשפה, אנחנו לוקחים את האובייקט עליו אנחנו מכריזים, ומקשרים אותו לשם שאנחנו רוצים. ברגע שנבצע קישור בין שם משתנה לערך כלשהו, רק אז המשתנה נחשב קשור וניתן להתחיל להשתמש בו. כאשר נסתכל על טיפוס נתונים של משתנה, חשוב לשים לב, שהטיפוס מוגדר רק כפי הערך הקיים במשתנה. כלומר, אם נגדיר בשפת C++ משתנה ללא קישור של ערך –

```
int num;
```

כבר ברגע שהגדרנו אותו באופן כזה, המערכת מפנה את המקום המתאים בזיכרון, ורק מחכה שנשים שם ערך כלשהו. לעומת זאת, בפייתון, אם נגדיר שם משתנה אנחנו מגדירים רפרנס שלא קשור לכלום. ורק ברגע ששמים עליו ערך כלשהו, גם אז אנחנו לא מקצים מקום חדש, אלא רק מקשרים את השם שהגדרנו עם הערך אליו הוא מצביע, ורק אז הוא יבין את הרפרנס.



הנגזר מזה הוא, שאם יש לנו שני שמות של משתנים בעלי אותו ערך, הערך עצמו לא שמור פעמיים תחת כל שם משתנה, אלא שני המשתנים מצביעים בדיוק לאותו מקום (כמובן, זה רק באופן כללי ותלוי משתנים וכו').

מעבר לזה, אנחנו יכולים להגדיר עבור משתנה ערך מטיפוס מסוים, ולאחר מכן לעשות השמה של ערך מטיפוס שונה לגמרי. למשל –

```
>> x = 7
```

```
>> x
7
>> x='a'
>> x
'a'
```

ואפילו שאנחנו משתמשים באותו שם, ברגע ששינינו לו את הטיפוס, הוא פשוט העביר אותו למקום אחר.

טיפוסי נתונים בסיסים

מספרים

בחלק זה, נעבור על כל טיפוסי הנתונים השונים לפרטיהם. אמנם אנחנו יודעים כבר מה אומר int וכמה הוא אמור לתפוס בזיכרון, אך מאחר ויש שינויים קלים במימוש כדאי לשים לב אליהם.

בגדול, את המשתנים בפייתון ניתן למיין כ-mutable, וכ-immutable. ישנם ערכים שניתנים לשינוי פנימי (mutable), ואז ניתן לעבוד עליהם בצורה חופשית, וכאלה שהם בלתי ניתנים לשינוי, וברגע שהגדרנו אותם פעם אחת, אין לנו אפשרות להמשיך ולעבוד עליהם(אלה כמובן ה-immutable).

המספרים שייכים לקבוצת הבלתי-ניתנים-לשינוי. מה המשמעות של זה? כמובן שאין שום דרך להגדיר בשום מקום שמספר יהיה שווה למשהו שהוא לא עצמו (נגיד משהו כמו 8=7). אבל בפייתון המשמעות היא קצת שונה. השוני נובע מהגדרת המשתנים כרפרנסים ולא כמקום בזיכרון. אם עד היום היינו יכולים להגדיר "תיסוף" עבור משתנה, כלומר פקודה כזאת –

```
x = x+1;
```

שהיתה לוקחת לנו את הערך של שמוגדר בתוך x, מוסיפה לו את הערך הדרוש ושומרת שוב על אותו המקום, בפייתון זה דבר שהוא לא אפשרי. כמובן, שאנחנו יכולים להגדיר פקודה כזאת, אך המשמעות האמיתית שלה תהיה שונה. בשביל לראות את ההבדלים השונים, נכתוב תוכנית מאוד פשוטה –

```
>>> x = 3.5
>>> y = x
>>> x
3.5
>>> y
3.5
>>> id(x)
55724816
>>> id(y)
55724816
```

הגדרנו את x להיות משתנה מסוג float שמכיל את הערך 3.5. ברגע שאנחנו מגדירים את y=x, אנחנו לא רק מגדירים שהערך שלהם יהיה שווה, אלא ששניהם למעשה הם בדיוק אותו דבר. איפה זה בא לידי ביטוי? כשאנחנו מבצעים את הפונקציה המובנית id, שמחזירה לנו את הכתובת של המשתנה, אנחנו יכולים לראות שאכן שני המשתנים מגיעים לאותו מקום.

אבל המצב יהיה שונה ברגע שננסה לשנות את הערך –

```
>>> x = x+1
>>> x
4.5
>>> y
3.5
>>> id(x)
51248256
>>> id(y)
55724816
```

אם אנחנו משנים וכותבים x=x+1, אנחנו לא רק משנים את הערך שאליו x מצביע. אנחנו משנים את ההצבעה עצמה – ה-id של y נשאר במקום המקורי ובערך המקורי, אך לעומת זאת, x לא רק שינה את הערך, אלא הכתובת שלו נמצא במקום אחר לגמרי.

משתנים בוליאניים

ההגדרה הבסיסית יותר של המשתנים הבוליאניים, הם True ו-False, כאשר נזכיר שוב, שהאות הראשונה צריכה להיות גדולה. מעבר לזה, הגדירו בשפת את הערך False, להיות שווה לכל אחד מה"ערכים הריקים" הבאים –

0, 0.0 – המשתנים המספריים, בהם לא רק 0 של int אלא גם 0.0 של ה-float (כזכור, כל מספר float גם אם הוא שלם, יוצג בצורה עשרונית, ו-0 אינו שונה מהשאר).

"", str() - המחרוזת הריקות.

[], list() –רשימות ריקות.

(), tuple() - רשומה ריקה.

{} set() – קבוצה ריקה

{}, dict() – מילון ריק.

None – משתנה ייחודי לפייתון, המשמש בתור NULL.

כל אלו מוגדרים כשקר.

כל ערך אחר יוגדר כאמת.

ניתן לבצע על הערכים הבוליאניים פעולות לוגיות (AND, OR) כאשר כל הכללים הלוגיים שאנחנו מכירים עובדים גם כאן, עם מספר שינויים קלים –

```
>>> True and False

False

>>> False or True

True

>>> 7 and 14

14

>>> None and 2

>>> None or 2

2
```

בעוד שני החלקים הראשונים הם ברורים, למה 7 וגם 14 שווים 14?

יש לזכור, שמבחינה לוגית בשביל לקבל ערך אמת תחת AND צריך ששני התנאים יקויימו, ולכן, כאשר האינטרפרטר בודק את 7, שהוא אמת (מאחר והוא לא 0), הוא לא יקבל ערך אמת עד שהוא לא יבדוק את 14. ומאחר ש-14 הוא הערך המגדיר את התוצאה כאמת, זה אכן הערך שיוחזר.

מבחינת הערך None, הוא לא ממש מוגדר ערך שקר, אלא כלום. ולכן אם יהיה ביחס AND עם ערך אחר לא יחזור כלום, כי כלום ו-2, זה לא ערך מוגדר, לעומת OR שם מספיק שרק אחד מהתנאים יתקיימו, ולכן חוזר לנו 2.

טיפוסי נתונים מורכבים

מחרוזות

כמו שנאמר קודם, עלינו לזכור שמחרוזת היא מערך הומוגני. כלומר – בניגוד לכל שאר הטיפוסים המורכבים שנראה בהמשך, הוא היחיד שמכיל בתוכו רק משתנים מסוג אחד – סמלים (אותיות ומספרים).

המערך שומר את כל מה שהכנסנו בו, כולל רווחים, וכך אם נרצה לחתוך אותו או לעשות עליו פעולות, אנחנו נוכל לעשות את זה תוך כדי ידיעה שהרווחים משמשים לנו איזה עוגן.

מבחינת הגדרת המחרוזת עצמה, יש לנו 3 אפשרויות- שתי האפשרויות הראשונות, הן כאלה שכבר נתקלנו בהם בעבר – על ידי גרש אחד (')‚ או גרשיים (") בתחילת ובסוף הרצף. אין שום הבדל או משמעות בין לעשות את זה או את זה. אז למה זה טוב? אם נרצה להכניס את המשפט הבא – "ז'ורז'ט ג'יעג'עה את הג'יפ", וננסה לכתוב את זה רק עם גרש אחד, זה לא ממש יעבוד לנו, אבל אם נתחום את המחרוזת עם גרשיים, נוכל לכתוב בפנים כמה גרשים בודדים שרק נרצה. וגם להיפך, אם נרשום "קמר"ש הלור"ל", נוכל לתחום את זה בגרש בודד ולהמשיך הלאה.

ומה אם נרצה לרשום גם גרש אחד וגם גרשיים? שתי אפשרויות – נוכל להוסיף לפני התוים האלו בטקסט קו נטוי (\), ואז הם יפורשו בצורה ברורה. לחילופין – נוכל לבחור באפשרות השלישית להגדרת המחרוזות – """ – שלוש גרשיים ברצף. מה זה נותן לנו? מלבד סתם אפשרות נוספת להגדרת מחרוזת, אם יהיה לנו טקסט שיהיה יותר משורה אחת, ונרצה לרשום אותו בצורה יותר אינטואיטיבית מאשר לדחוף \n בכל סוף שורה, נוכל לתחום את קטע הטקסט ב"""‚ ובכל פעם שנרד שורה, הפייתון יקרא את זה כירידת שורה אמיתית ויבצע בהתאם.

פעולות על מחרוזות

אם נרצה לשרשר מחרוזות שונות אחת לשניה נוכל להשתמש בסימן ה+. למשל:

```
>>> x = 'hello'

>>> x = x + ' there'

>>> x

'hello there'
```

אם לחילופין, נרצה לשרשר את אותה מחרוזת, או להכפיל אותה ברצף מספר פעמים, נוכל פשוט להשתמש בסימן הכפל * , ואז לכתוב כמה פעמים נרצה –

```
>>> y = 2*x

>>> y

'hello therehello there'
```

שליפת איבר מהמחרוזת

בדומה למערכים רגילים, גם כאן, נשלוף איבר על ידי שם המחרוזת, והאינדקס בתוך סוגריים מרובעות. מה שמיוחד באינדקסים של פייתון, הוא שכאן הקריאה יכולה להיות דו כיוונית. כלומר – מעבר למיספור הרגיל החל מ-0 ועד n-1, יש מיספור נוסף בכיוון ההפוך, שמתחיל באינדקס האחרון בסדרה, שם ממספרים 1-, ועד האיבר במקום הראשון (האמיתי), שם הוא ימוספר n-. כך שנוכל לגשת למערך בכל האופנים הבאים –

```
>> s= 'Hello World!'

>> s

'Hello World!'

>> s[0]

'H'

>> s[11]
'!'

>> s[-1]

'!'
```

חיתוך מחרוזת

פייתון מכיל בתוכו פונקצייה מובנית לחיתוך מחרוזות. בשונה מהרבה שפות, שם צריך להגדיר slice ואז את הטווחים, בפייתון אנחנו מבצעים את כל זה בתוך הסוגריים המרובעות. כאשר, את הטווחים מגדירים בין נקודותיים, בסדר הבא – [start : end : move by]. כאשר הטווח המוגדר הוא החל מהתו הראשון ועד האחרון (ולא עד בכלל), והחלק השלישי מתייחס לדילוגי האותיות על המחרוזת, אם להגדיר 1 (או לא להגדיר) בשביל להביא את כל מה שבטווח, או להגדיר מספר ובעבורו יהיו דילוגים (החל מתחילת הטווח). נראה מספר דוגמאות –

```
>>> s = 'Hello World!'
>>> s
'Hello World!'


>>> s[9:11]        >>> s[-12]
'ld'               'H'
>>> s[1:12:2]      >>> s[-1:-12:-1]
'el ol!'           '!dlroW olle'
>>> s[-11:-1]      >>> s[::-1]
'ello World'       '!dlroW olleH'
```

מה שמיוחד, הוא שלא חייבים להגדיר חלקים שאין לנו בהם צורך – בדוגמה הראשונה, לא הגדרנו דילוג, אז פשוט הלכנו ברצף, בדוגמאות האחרונות, אנחנו רואים שאפשר גם להפוך את הרשימה ברברס, על ידי קביעת הדילוגים שיהיו 1-. אם עושים את זה, כדאי לשים לב ששמים את הטווחים נכון, אחרת תחזור לנו מחרוזת ריקה.

פונקציות נוספות על מחרוזות

- **len**(*string*) - מחזיר את אורך המחרוזת (כולל רווחים).

```
>>> x = 'I love Python'

>>> len(x)

13
```

- (*object*)**str** - המרת אובייקט כלשהו (שאינו מחרוזת) למחרוזת.

```
>>> str(10.3)
'10.3'
```

- ( )**upper**.*string* - המרת המחרוזת לאותיות גדולות בלבד (קיימת גם אופציה מקבילה להמרה לאותיות קטנות (lower))
- ([sep])**split**.*string* - חיתוך המחרוזת, והחזרתה כמערך המחולק למילים. כברירת המחדל, החיתוך יתבצע בכל רווח, אך אם רוצים, ניתן להכניס בסוגריים תו שהחיתוך יהיה לפיו - רלוונטי אם רוצים לחתוך למשל כתובת לקובץ.

```
>>> x.split(' ')
['I', 'love', 'Python']
```

- (*list_of_strings*)**join**.*string* - לוקח מערך של מחרוזות ומחזיר מחרות אחת ארוכה.

```
>>> ' '.join(x.split(' '))
'I love Python'
```

- (*substr*)**find**.***string*** – מחפש האם תת-המחרוזת הדרושה נמצאת במחרוזת המקורית. אם כן, מחזיר את האינדקס (הראשון) של המופע של תת המחרזות. אם לא, מחזיר 1-. ניתן גם להתחיל לחפש מהסוף, על ידי findr(subst)

```
>>> s = "Jerusalem is my city"
>>> s.find("e")  #  s.index("e") returns the same value
1
>>> s.rfind("e") # s.rindex("e") returns the same value
7
>>> s.find("salam")
-1
```

(*substr*)**index**.***string*** - מבצע את אותו הדבר כמו find, אך אם הוא לא מוצא, נזרקת חריגה (שאותה ניתן לתפוס במידת הצורך.

```
>>> s.index("salam")
Traceback ….
….
Value Error: substring not found
```

- ***string.count***(*substr*). – מחזיר את מספר ההופעות הלא-חופפות של תת המחרוזת (הלא חופפת, במקרה שמחפשים aba אז ה-a השניה לא נספרת כתת מחרוזת חדשה.

```
>>> s.count("e")
2
>>> s.count("er")
1
>>> s.count("era")
0
```

רשימות

בניגוד למחרוזות שם אנחנו לוקחים כל איבר במחרוזות כCHAR, רשימות הן הטרוגניות ויכולות להכיל איברים שונים מטיפוסים שונים, כאשר יכול להיות ששתי רשימות יצביעו לאותו ערך משותף.

יש פונקציות דומות לאלו של המחרוזות – גישה למערך נעשית עם סוגריים מרובעות [], ועל מנת לעשות חיתוך, ניתן להשתמש באותו אופן חיתוך בדיוק כמו שאנחנו עושים במחרוזת.

כאשר אנחנו מבצעים "סלייס", מה שנעשה הוא יצירת עותק של הרשימה, ולא חיתוך על הרשימה המקורית, כך שאם נרצה לקבל את החיתוך, נצטרך לדאוג שתת-הרשימה תושם במשתנה חדש. כמו במחרוזת, אם לא מגדירים התחלה או סוף, אז מתכוונים לקצה של כל כיוון, והדילוג (החלק הימני) הוא לא הכרחי בזמן חיתוך. כמו כן, אם משנים את תחילת הרשימה המועתקת, אז הקפיצות מתייחסות לרשימה החדשה ולא המקורית, כך שאם נבקש את הקפיצות ב-2, אז לא נקבל דווקא את הזוגיים, אלא את הזוגיים מאותה נקודה שהגדרנו כהתחלה.

```
>>> x = [1,'hello', (3 + 2j)]
>>> x
[1, 'hello', (3+2j)]
>>> x[2]
(3+2j)
>>> x[:2]
[1, 'hello']
>>> x[::2]
[1,(3+2j)]
```

ניתן גם להציב מחרוזת כאיבר בתוך מחרוזת, וכך ליצור רשימות מקוננות –

```
>>> y = [5, x, ‘abc’]   # y is a nested list

>>> y

[5,[1, 'hello', (3+2j)],’abc’]

>>> id(x)

2516447480072

>>> id(y[1])

2516447480072
```

שימו לב, הגדרנו את הרשימה x בתור איבר ברשימה y. המצביע של האיבר x נמצא בדיוק על אותו אחד של המצביע בתוך הרשימה. כלומר – אנחנו לא הגדרנו עותק של x בתוך y, אלא ממש שמנו שם את אותו אחד.

רשימות ניתנות לשינוי, לכן ניתן להוסיף מידע לתוך רשימה, ולהישאר באותו רפרנס של כתובת –

```
>> x = 3
>> x = x+1

>> list = [1,2,3]

>> list[2] = 55

>> list

[1,2,55]
```

כמו במחרוזות יש פונקציית אינדקס שבאמצעותה ניתן לחפש איבר בתוך הרשימה, ולקבל את האינדקס למקום שאותו איבר נמצא בו. כך שניתן ליצור רשימה של מילים במשפט מסוים על ידי שימוש בספליט, שבאופן אוטומטי חותך ומחזיר מערך לפי הרווחים[2]- אם יהיה ניקוד שצמוד מילה אז הוא יופיע כאותה מילה עד לרוח), ולחפש את המילים כאינדקס מסוים.

כמו כן, אנחנו יכולים לספור כמויות של ערך מסוים בתוך רשימה –

```
>> L = "Jerusalem is my city, my love".split()

>> L.index("my") // יוחזר לנו כאן המופע הראשון שנמצא ברשימה

2

>> L.count("my")

2

L.index("lobe")

Traceback..

….
```

[2] גם פה ניתן גם להגדיר מה שאנחנו רוצים שייחתך בספליט על ידי הכנסה של התו הדרוש לחיתוך לתוך הסוגריים. למשל- x.split('w')

יחתוך לנו בכל פעם שתופיע האות w במהלך המחרוזת הגדולה.

Value Error: 'lobe' is not in list

מתודות לביצוע שינויים ברשימה

Append

הוספה של איבר לסוף הרשימה. אם אנחנו נשלח רשימה בפונקציה בתור איבר, אז אנחנו נוסיף את הרשימה את כאיבר כמות שהיא לסוף הרשימה קיימת –

```
>>> x = [1, 15, (3+2j)]
>>> x
[1,15,(3 + 2j)]
>>> x.append([1,2,3])
>>> x
[1,15, (3 + 2j), [1,2,3]]
```

Pop

הוצאה של האיבר האחרון ברשימה (בפועל זה עובד כמו מחסנית, שמוסיפים איברים לראש המחסנית, ואז פופ יוציא את האחרון שנכנס (LIFO)-

ניתן גם להעביר פרמטר לפופ, ואז הוא ייקח את האיבר המסוים, ולאו דווקא את הראשון ברשימה

```
>>> topitem = x.pop()
>>> topitem
[1,2,3]
```

Extend

העברה של רשימה לתוך רשימה קיימת. בשונה מ-append, כאן הרשימה תפורק לתוך הרשימה המקורית –

```
>>> x
[1,15, (3 + 2j)]
>>> x.extend(['a',23])
>>> x
[1,15, (3 + 2j), 'a',23]
```

Insert

הכנסה של איבר במיקום מסוים. כאשר מגיעים לאינדקס הנתון, לא מחליפים את האיבר המסוים כמו בגישה לאינדקס, אלא דוחפים את הרשימה החל מהמיקום המסוים שביקשנו, והאיבר החדש יהיה האינדקס הדרוש –

```
>>> x.insert(2,"yes")
>>> x
[1,15, "yes",(3 + 2j), 'a',23]
```

Del

מחיקת האיבר לפי מיקום מסוים (אינדקס) ברשימה.

```
>>> del x[2]
>>> x
[1,15, (3 + 2j),'a',23]
```

Remove

מחיקת איבר לפי ערך, שולחים ערך מסוים, והפונקציה מוחקת את המופע הראשון (בלבד) ברשימה. אם יש כמה ורוצים למחוק את הכל, צריך לעשות את זה באופן עצמאי – לעשות לולאה שרצה עד שלא מוצאים יותר את הערך הדרוש. אם האיבר לא נמצא אז נזרקת חריגה של ValueError, כך שאם עושים לולאה יש לשים לב לא להיזרק החוצה –

```
>>> x.remove('a')
>>> x
[1,15, (3 + 2j),23]
>>>
```

דבר חשוב- כל המתודות האלה מחזירות None – מבצעות פעולה על הרשימה עצמה ומחזירות None, כך שבניגוד לחיתוך שראינו קודם, יש לשים לב ולהמנע מהטעות של ניסיון להחזיר תוצאה. אם ננסה לעשות דבר כזה, אז הערך המקורי אכן יתבצע בדיוק כמו שביקשנו, אבל הערך שביקשנו לעשות לתוכו השמה, יקבל לתוכו None. למשל -

```
L1 = L2.append('a')
```

במקרה הזה השינוי הדרוש ב-L2 יתבצע, אך L1 יקבל None.

רשומות - Tuples

רשומות הן "רשימות שלא ניתנות לשינוי". כלומר – מבחינת מה שהרשומה יכולה להכיל, מדובר בדיוק באותה צורה של רשימות רגילות, כל סוגי הטיפוסים להכנס בצורה מעורבבת, וכן ניתן לגשת למיקום ברשומה והכל. אך לעומת הרשימות, שהן דינמיות, הרשומות הן קבועות ולא ניתנות לשינוי.

ברשומות, כמו במקרה של רשימות, אם ניצור רשומה שבתוכו יש רשימה, אז נוכל ליצור רשומה מקוננת, וכן לגשת ולשנות את המידע ברשימה הפנימית, ואפילו להוסיף שם איברים בצורה חופשית.

```
>>> a=('a','b',1,2,['a','b','c'])
>>> a[4][0]='b'
>>> a
('a', 'b', 1, 2, ['b', 'b', 'c'])
```

מבחינה סינטקטית אנחנו מציינים רשומה בעזרת סוגריים רגילות - (), כאשר גישה לאיבר מתבצעת על ידי [] באותו אופן כמו שאנחנו מכירים ורגילים.

אמנם רשומות אינן ניתנות לשינוי, אך אם נרצה להכפיל רשומה או לשרשר שתי רשומות אחת לשניה, זה לא ייחשב כ"שינוי ברשימה", וניתן לעשות את זה באופן חופשי.

השוואה בין רשימה לרשומה

מבחינת האינטרפרטר, רשומה ורשימה הם שני טיפוסים שונים לגמרי. כך שאם ניקח בדיוק את אותו סדר איברים, ונגדיר אותם פעם כרשומה ופעם כרשימה, אז השוואה ביניהם תיתן False, אך אם נשווה ערכים ספציפיים מתוך הטיפוסים השונים, זה אפשרי, אבל כמובן רק ביחס לאותם איברים מתאימים.

אם רוצים לשחק עם רשומה ולשנות בה ערכים, כמובן שזה בלתי אפשרי, אך אם נרצה בכל אופן, אנחנו יכולים להגדיר רשומה חדש על ידי סלייסים – חיתוך הרשומה באופן שלמדנו, ולאחר מכן, להגדיר רשומה חדשה על ידי שרשור והוספת איברים כדרוש.

ניתן להגדיר רשומה עם איבר אחד, אך מבחינה סינטקטית נצטרך להכניס פסיק אחרי אותו איבר.

```
>>> x = (1,2,3)
>>> x[1:]
(2, 3)
>>> y = (2,)  # a
```

קבוצות – SETS

לעומת רשימה ורשומה, קבוצות זה אוסף הטרוגני של נתונים ללא שום סדר וללא כפילויות. לא הזכרנו קודם שרשימות לסוגיה יכולות להכיל כפילויות אבל זה דבר שנשמע לנו ברור, אך השוני בקבוצה, הוא שאנחנו מדברים על קבוצה בהגדרתה המתמטית, המתייחסת לאיברים ללא כפילויות ואף ללא סדר. כפועל יוצא מכך, אין לנו שום אפשרות לגשת לאיבר על פי אינדקס. יותר מזה, אם ניקח קבוצה ונדפיס אותה כמה פעמים, נקבל בכל פעם סדר טיפה שונה.

קבוצות מסומנות על ידי סוגריים מסולסלים – {}, או באמצעות פונקציית ההמרה set, שמשנה את הרשימה לצורה של קבוצה, כך שאם יש כפילויות, אז הכל יורד במהלך ההעברה (פתרון להורדת כפילויות).

הנתונים שאנחנו מכניסים לקבוצה חייבים להיות כאלה שניתן לשמור אותם בצורת Hash Table. מאחר ואין סדר פנימי של ראשון ואחרון, בעת הגדרת הקבוצה האיברים מוכנסים בצורה מעורבלת על יד הפייתון עם פונקציית HASH פנימית.

הקבוצות ניתנות לשינוי – הוספה, הורדה(-), חיתוך(&), איחוד (|), הפרש סימטרי XOR (^) וכך כל פעולות הקבוצות –

```
>>> s1 = {1, 'two', 'blah', 5, 1}

>>> s1

{'two', 1, 'blah', 5}

>>> s2 = set(['was', 4, 5, 4, 'two', (5,3,8)])

>>> s2

{'two', (5, 3, 8), 4, 'was', 5}

>>> s1 & s2

{5, 'two'}

>>> s1 | s2

{'two', 1, 'blah', 5, (5, 3, 8), 4, 'was'}
```

בשונה ממה שאמרנו לגבי פונקציות על רשימות, ניתן להכניס השמה לתוך סט בשילוב של פעולות ולהגדיר כרצף של חיתוכים, איחודים וכו'.

פונקציות של קבוצות

add – הוספת איבר לקבוצה.

remove – מחיקת איבר מהקבוצה. אם האיבר לא נמצא נזרקת חריגה.

discard – מחיקה קצת יותר בטוחה ללא חריגה (אם לא נמצא, יחזור None).

pop – הוצאה של איבר אקראי מהקבוצה.

clear – מחיקת כל האיברים מהקבוצה.

הקבוצות הן איטרביליות, כך שאפשר לעבור עליהן עם לולאת for. אם מחפשים איבר בקבוצה, ניתן פשוט לעבור על כל האיברים בקבוצה עם לולאה, עד שמוצאים את האיבר הדרוש

מילון – Dict

קבוצה של זוגות "מפתח:ערך", כלומר מדובר פה בהרחבה של מושג הקבוצה שראינו עכשיו לכך שאנחנו יכולים לגשת לערכים בעזרת מפתחות שאנחנו מגדירים. המילונים הם ניתנים לשינוי, כאשר ניתן להגדיר כמפתח גם מספרים וגם מחרוזות. מותר להשתמש בכל סוג מפתח שהוא לא ניתן לשינוי – כלומר - אי אפשר להגדיר מפתח בעזרת רשימה, אך אפשר להגדיר מפתח עם רשומה.

ניתן להגדיר גם בצורה של פונקציה, שמכניסים בה את הערכים. זה שימושי להחזרה של מילון מתוך פונקציה, שיכולה להבנות בזמן ריצה, ואז להחזיר אותה.

פעולות על מילון

איך מוסיפים או משנים זוג במילון?

```
x[k] = a
```

ניגשים למילון x בעזרת המפתח k, ומכניסים לתוכו את הערך הרצוי a. במידה והמפתח לא יהיה קיים במילון, אז הערך ייכנס בכל זאת עם המפתח החדש שהגדרנו.

משיכת איבר מתוך המילון גם מתבצעת באופן דומה –

```
n = x[k]
```

הפונקציה del מאפשרת למחוק ערך בעזרת המפתח שלו.

```
>>> d
{1: 'hello', 2: 'there', 10: 'world'}
>>> del(d[2])
>>> d
{1: 'hello', 10: 'world'}
```

אמרנו שערכים במילון ובקבוצות חייבים להיות hashable כלומר שיכולים להיות כאלה שניתן לערבל אותם בפונקצייה הפנימית של פייתון. מה הכוונה לערך שהוא hashable?

טיפוסי נתוני Immutable, כמו שכבר אמרנו, וכן מופעים של מחלקה שנוצרו על ידי המשתמש. מותר ששני מופעים של אותה מחלקה יהיה כל אחד מפתח שונה, אפילו אם הם מכילים את אותו תוכן (אם שני מופעים של מחלקה

מכילים את אותו תוכן, אז מבחינת פייתון הם מצביעים לאותו מקום, אך בכל זאת ההגדרה השונה שלהם גורמת לכך שיוכלו להיות כל אחד מפתח בפני עצמו).

מדוע אסור לנו שהמפתחות יהיו ניתנים לשינוי? אמרנו שההכנה של קבוצות מתבצעת על ידי פונקצית ערבול. כשאנחנו מגדירים מילון עם מפתח, אנחנו מסייעים לעצמנו בזה שאנחנו יכולים לגשת לערכים. אבל אם נכניס מפתח שהוא ניתן לשינוי, אז הערבול שלו יגיע לנקודה מסוימת, ואם נשנה עכשיו את הערך, אז הפונקציה לא תוכל להגיע לאותה נקודה, כי הערך השתנה.

Range

יש פונקציה ששמה range (טווח) שבאמצעותה ניתן ליצור ערכים בטווח מסוים. כאשר אנחנו מגדירים טווח מסוים ומכניסים לתוך משתנה, המשתנה מקבל את הטווח הזה, ואז ניתן להמיר אותו לרשימה או למבנה שאנחנו צריכים. אנחנו נשתמש בפונקציה הזאת גם בהמשך שנגדיר לולאות for, אבל על זה נרחיב אחר כך.

הגדרה פשוטה של טווח תהיה פשוט לרשום עד איפה אנחנו רוצים להגיע –

```
>> x  =list(range(5))

>> x

[0,1,2,3,4]
```

אם אנחנו צריכים טווח שהוא קצת יותר מתקדם, אנחנו יכולים גם ליצור טווחים באופן דומה לפונקציית ה"סלייס", רק שבמקום להפריד את הקצוות עם נקודותיים, אנחנו מפרידים עם פסיק –

```
>> x = list(range(2,5))

>> x

[2,3,4]
```

הטווח הוא איטרטור, כך שאם נעשה עליו לולאה, נוכל לקבל בכל פעם איבר אחד מתוך הטווח לפי הסדר עד שנגיע לסוף. זה יכול לחסוך בזיכרון, אם רוצים לחסוך ולא ליצור רשימה ענקית ומיותרת.

כמו שראינו קודם, ניתן לעשות המרה של הטווח לרשימה, וכמו כן ניתן להעביר אותו גם להיות רשומה או קבוצה

Zip

דרך נוחה ליצירת מילון הוא על ידי הפונקציה זיפ. הפונקציה מקבלת שני אובייקטים של רשימות, ומאחדת אותם לצורה של ערך:מפתח בצורת ריץ'-רץ' – ערך ראשון מהרשימה הראשונה מפתח, ערך ראשון מהרשימה השניה ערך, וכן הלאה. אבל גם פה בדומה לטווח, אנחנו לא יכולים ישר להשתמש בזיפ הזה אלא אנחנו צריכים להעביר אותו ולעשות איזה המרה למילון ורק אז נוכל להשתמש בו –

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

>> strList = ['one', 'two', 'three']

>> result = zip(numberList, strList)

>> resultSet = dict(result)

>> print(resultSet)

{1: 'one', 2: 'two', 3: 'three'}
```

פונקציות על רצפים/קונטיינרים

יש פונקציות / מתודות שמוגדרות בשפה, שעובדים בצורה רחבה לכל סוגי הרצפים (מחרוזות, רשימות וכו'). על חלק גדול מהן עברנו בחלקים הקודמים, ונעבור עכשיו על עוד כמה בצורה מהירה –

פונקציה	פעולה	על מה מוגדר
x in s	מחזיר ערך אמת אם האיבר x הוא חלק מ-s, אחרת מחזיר שקר	על כל טיפוסי הנתונים המורכבים
x not in s	בדיוק הפוך מהקודם – אמת אם האיבר לא נמצא.	על כל טיפוסי הנתונים המורכבים
s + t	שרשור	על רצפים בלבד (לא על מילונים או קבוצות)
s * n, n * s	הכפלה ושרשור של העתקות רדודות	על רצפים בלבד (לא על מילונים או קבוצות)
s[i]	החזרת איבר i בתוך s	על הכל מלבד קבוצות. ברצפים i ייצג אינדקס, ובמילון הוא מייצג מפתח
s[i:j]	חיתוך בטווח [i,j)	על כל הרצפים
s[i:j:k]	חיתוך בטווח [i,j) בדילוגים של k צעדים	על כל הרצפים
len(s)	אורך (או גודל) של s	על כל טיפוסי הנתונים המורכבים
min(s) **max(s)**	איבר מינימלי ב-s	על כל טיפוסי הנתונים המורכבים. במילון מתייחס למפתח המינימלי
s.index(x[, i[, j]])	החזרת האינדקס של המופע הראשון של x בתוך s.	לא מוגדר על מילונים וקבוצות.
s.count(i)	החזרת מספר המופעים של i בתוך s.	על כל הרצפים
s[i] = x	החלפת המקום/המפתח i בערך x. במילון – אם המפתח לא קיים, מוסיפים את הערך	הכל מלבד קבוצות
s[i:j] = t	החלפת הטווח [i,j) בערך איטרבילי של t	ברשימות בלבד
del s[i:j]	מחיקת הטווח	ברשימות בלבד
s[i:j:k] = tt	החלפת הטווח בערך מסוים	ברשימות בלבד. אם tt זה מחרוזת, אז כל תו יקבל תא נפרד
del s[i:j:k]	מחיקת הטווח עם דילוגים	ברשימות בלבד
s.append(x)	הוספה של איבר לסוף הרשימה (אם מדובר על רשימה בסוף, הוא ייכנס כרשימה מקוננת. מבחינה סינטקטית זה הוספה של [x] לרשימה s הקיימת	ברשימות בלבד
s.extend(x)	הוספת משתנה לסוף הרשימה, אם מדובר על רשימה אז היתה תיכנס כאיברים בודדים.	ברשימות בלבד
s.insert(I, x)	הכנסת איבר בודד למיקום ברשימה	ברשימות בלבד

s.pop([i])	הוצאת איבר במיקום i, ומחיקתו מהרשימה.	ברשימות בלבד
s.remove(x)	מחיקת איבר i	

בדיקת טיפוס נתונים

בפייתון יש שתי פונקציות עם משמעות קצת שונה – type מקבלת משתנה, והוא מחזיר את סוג הטיפוס המוגדר עליו, כמובן שמדובר רק על הערך המושם באותו רגע – אם נשנה את הערך ה-type גם הוא ישתנה.

```
>> x = 5
>> print(type(x))
<class 'int'>
>> x= 'a'
>> print(type(x))
<class 'str'>
```

Isinstance – בודק האם המשתנה הוא מסוג מסוים לפי מה שאנחנו רוצים לבדוק, אנחנו שואלים על משתנה וסוג טיפוס, ומקבלים ערך אמת אם ההשמה אכן בטיפוס המאים. אנחנו יכולים גם להכניס מספר ערכים בתוך סוגריים ואז לבדוק האם המשתנה שייך לאחד מהם

```
>> x = 5
>> print(isinstance(x,int))
True
>> x= 'a'
>> print(isinstance(x,(str,int,tuple)))
True
```

.

העתקה

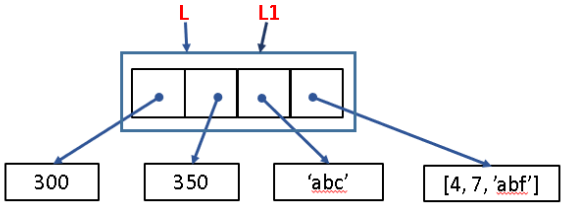
כאשר אנחנו מעוניינים להעתיק מאגרים (Containers), צורת ההעתקה מתחלקת לשתי אפשרויות – העתקה רדודה והעתקה עמוקה. בגדול, השוני בין רמות ההעתקה, הוא עד כמה שני המופעים של הרשימה יהיו קשורים האחד בשני – בהעתקה רדודה המופעים יהיו קשורים יותר, כלומר יצביעו לאותם מקומות בזיכרון (עד שלב מסוים), ובהעתקה עמוקה ערכים מסויימים ייכתבו מחדש לתוך המאגר. נראה את ההבדלים בהעתקות ואת המימושים השונים.

נסתכל על קטע הקוד שמופיע במצגת, ונראה את ההבדלים –

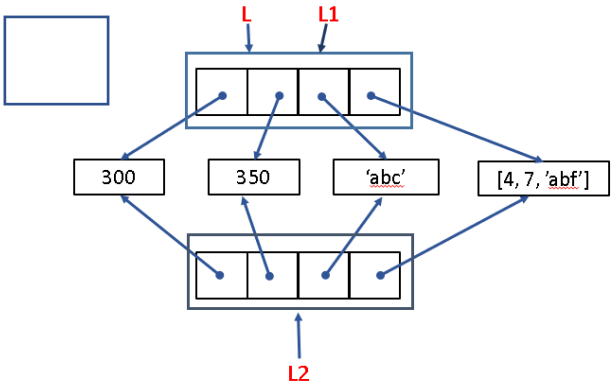
העתקה רדודה

```
>>> L = [300, 350, 'abc', [4, 7, 'abf']]
>>> L1 = L
>>> L2 = L1[:]
>>> L2
[300, 350, 'abc', [4, 7, 'abf']]
>>> id(L1)
2620735606984
>>> id(L2)
2620735607240
>>> L1[0]
300
>>> id(L1[0])
2620734897072
>>> id(L2[0])
2620734897072
>>> id(L1[1])
2620734895408
>>> id(L2[1])
2620734895408
>>> id(L1[2])
2620725558608
>>> id(L2[2])
2620725558608
>>> id(L1[3])
2620735458504
>>> id(L2[3])
2620735458504
>>> L1
[300, 350, 'abc', [4, 7, 'abf']]
>>> L2
[300, 350, 'abc', [4, 7, 'abf']]
```

קודם כל, יש לציין ששורת ההעתקה הראשונה (השרה השניה בקוד), היא ההעתקה בצורתה הרדודה ביותר – שתי הרשימות הן **בדיוק** אותו דבר, רק עם שם שונה, המונח המתאים לזה הוא Aliasing, כלומר כינוי, זה כמו לקרוא לאברהם "אבי", שום שינוי לא נעשה פה באמת, ואפשר לראות בתרשים העזר, ששני המשתנים מצביעים לאותם ריבועים.



אם רוצים לעשות העתקה רדודה, שהיא כבר קצת יותר מנותקת ממה שראינו עכשיו, יש לנו אופציה להשתמש בפונקציית החיתוך בתצורה מיוחדת. אם אנחנו שולחים העתקה בתוספת "חיתוך ריק" כלומר [:], אנחנו מבצעים העתקה רדודה. כלומר אנחנו מפנים מקום חדש, שיצביע בדיוק לאותו מקום ואותם ערכים כמו קודם, דבר שאנחנו יכולים לראות בקוד שנבדק, ואכן כל מקום ברשימה אחת, מופיע בדיוק באותה כתובת ברשימה השניה. מבחינת התרשים, אנחנו יכולים להראות את הדבר הבא –



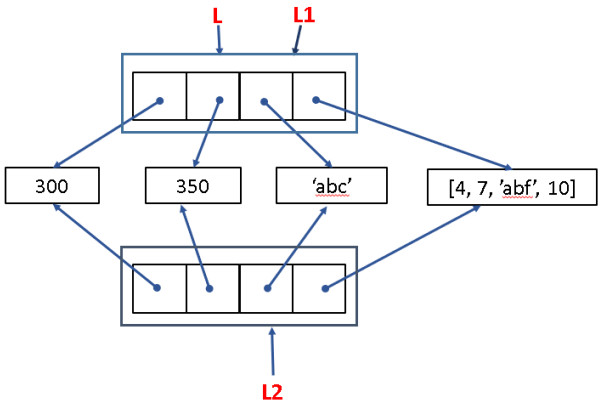
כלומר, הרשימות עצמן נמצאות במקומות שונים, אבל הערכים נשארו באותה כתובת. הערה: במילון יש לנו פונקציה מובנית בשם copy, שמבצעת בדיוק אותו דבר (כי כזכור, אין חיתוך במילון), שנראה כך-

```
>>> d = {1 : 10}

>>> d2 = d.copy()

>>> d[1] = 22

>>> d

{1: 22}

>>> d2

{1: 10}
```

מה שמראים פה בנוסף, הוא שאם אנחנו משנים את הערך שהוא immutable, אז זה משנה רק ברשימה אותה שינינו, אך המילון השני עדיין יצביע למקור.

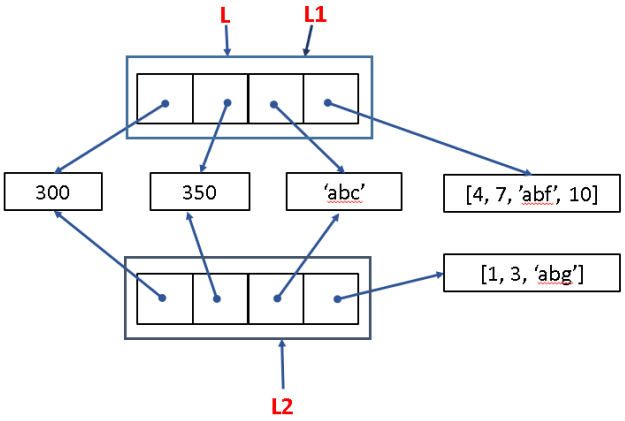
לעומת זאת, אם ניקח את הרשימה הפנימית בקוד שהכנסנו למעלה, ונשרשר אליה עוד ערכים, אנחנו עדיין נשארים תחת אותה הגדרה, ורק שינינו את הערך הפנימי, כמו שניתן לראות בקוד, ובתרשים מתחתיו–

```
>>> L1[3].append(10)
>>> L1
[300, 350, 'abc', [4, 7, 'abf', 10]]
>>> L2
[300, 350, 'abc', [4, 7, 'abf', 10]]
```



כמובן, שאם נרצה לשנות לגמרי את **כל** הרשימה המקוננת, אנחנו יכולים לעשות את זה, אבל בשלב הזה הרשימה בה אנחנו משנים את זאת הפנימית תצביע למקום אחר לגמרי – שוב, מאחר שאם אנחנו מגדירים רשימה חדשה, אנחנו מגדירים מצביע חדש לכתובת שיהיה במקום אחר. קוד ותרשים –

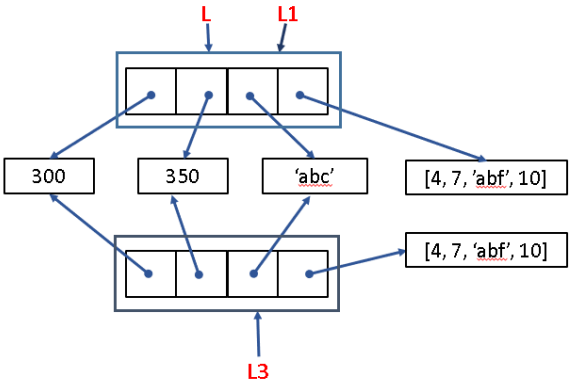
```
>>> L2[3] = [1,3,'abg']
>>> L1
[300, 350, 'abc', [4, 7, 'abf', 10]]
>>> L2
[300, 350, 'abc', [1, 3, 'abg']]
```



העתקה עמוקה

יש מודול מיוחד בפייתון שניקרא copy שניתן לייבא אותו ולהשתמש במתודה המוגדרת בו שנקראת deepcopy. לעומת ההעתקה הרדודה, שאת כל נגזרותיה ראינו עכשיו, כאן אנחנו יוצרים עותק חדש של הרשימה, כאשר כל האיברים שהם Mutable כמו מספרים עדיין ישארו במקום, כמו ההגדרה הרגילה, אך משתנה שאינו Immutable כמו למשל רשימות מועתק מחדש –

```
>>> import copy
>>> L3 = copy.deepcopy(L1)
>>> id(L1)
2620735606984
>>> id(L3)
2620735458056
>>> L1[3]
[4, 7, 'abf', 10]
>>> id(L1[3])
2620735458504
>>> L3[3]
[4, 7, 'abf', 10]
>>> id(L3[3])
2620735632520
```



כמו שאנחנו רואים, אמנם מדובר על אותה רשימה בעלת אותם ערכים, אך ההעתקה העמוקה גרמה להם להיות שונים כך שנוכל לעשות שינויים ברשימה L3, מבלי לפגוע בערכי הרשימה L1.

משפטי בקרה

הדבר החשוב ביותר שצריך לזכרו בפייתון לעומת שפות אחרות שהכרנו עד עכשיו, הוא בשפייתון אין לנו תיחום של איזור הכרה (Scope) על ידי סוגריים מסולסלות כמו שאנחנו רגילים, אלא על ידי הזחה. כלומר, אם אנחנו מגדירים איזה פונקציה, תנאי, או לולאה כלשהי, עלינו לוודא שכל החלק הרלוונטי לפעולה, יופיע בהזחה ביחס לכותרת. מבחינה סינטקטית, גם בשפות אחרות אנחנו עושים הזחה כזאת, אבל זה סתם מטעמי נוחות וקריאות של השפה, אבל כאן זה מוכרח – אם הגדרנו תנאי וכל הפעולה נמצאת באותו מרחק מהקצה, לא עשינו פה כלום.

נעבור עכשיו על רשימה של משפטי בקרה שעוזרים לנו בזרימת התוכנית.

If

על מנת להגדיר פעולה תחת תנאי, אנחנו רושמים בתצורה הבאה: **if** cond: אין שום דבר שצריך להכניס בסוגריים, אלא מציינים את התנאי וממשיכים בפעולות שיתקיימו על ידי קיום התנאי, בשורות מתחת (והצידה)-

```
desc = b**2 - 4*a*c
if desc == 0:
  root = -b / (2*a)
```

אם אנחנו רוצים להכניס אפשרויות נוספות, אנחנו יכולים להכניס עוד (**el**se **if**) **elif**, אחרי הפעולות האלה, שצריך להיות באותה הזחה של ה-if, המקורי, וכך ניתן להמשיך ולשרשר ולהגיע גם בסוף ל-else ללא שום תנאי.

While

בנוי באותה תצורה של if – **while** cond: והפעולות בהזחה מתחת -

```
x = 1
while x < 10:
    print (x)
    x = x + 1
```

ניתן לשלוט בלולאה על ידי מספר פקודות שיכולות ליצור זרימה שונה קצת מההגדרה המקורית. אם אנחנו רוצים, אנחנו יכולים להכניס למשל עוד תנאים בתוך ה-while וככה להחליט מתי יוצאים או מתי חוזרים לתנאי הראשון -

Break – שבירה של הלולאה ויציאה ללא כל תלות בתנאי הלולאה.

Continue – קפיצה בחזרה לראש הבלוק, בדיקה אם התנאי חוזר ומתקיים והמשך הרצה.

Pass – להמשיך הלאה בתוך ה-while ללא שום פעולה מיוחדת

ניתן כמובן גם לקונן תנאים אחד בתוך השני, כאשר כל שיש לשים לב הוא רק הריווח שיתאים לכל לבלוק.

אם רוצים שפעולה מסוימת תתבצע בסוף ריצת ה-while אפשר להגדיר else בסופו (ובאותה הזחה), כך שאם הלולאה תסיים את הריצה בהצלחה (או במילים אחרות – לא תישבר החוצה), גם התנאי הזה יתבצע –

```
#
# whilelse.py
#
x = 1
while x < 3 :
    print (x)
    x = x + 1
else:
    print ('hello')
```

```
1
2
hello
```

For

לולאה איטריבית שעובדת בצורה מחזורית. על מנת להגדיר את הלולאה הזאת, אנחנו משתמשים בסינטקס שדומה יותר ללולאת foreach מאשר לולאות for רגילות שאנחנו מכירים. ה-for של פייתון צריך לעבוד על טיפוס איטרבילי, כמו רשימה או משהו בסגנון. הדרך הנוחה ביותר לעשות את זה הוא על ידי הגדרה של פונקציית טווח (Range) שראינו קודם שמגדירה את טווח הריצה ואת הדילוגים הדרושים. יש לזכור, שמבחינת המערכת זה לא משנה אם הגדרנו שירוץ 5 פעמים, או רק על זוגיים עד 10, אבל אם יש לנו סיבה נוחה יותר להשתמש בזוגיים, אז אנחנו יכולים להגדיר את זה כך.

אנחנו יכולים להגדיר לולאה שלא תלויה במשתנה, אלא לקבוע מראש את מספר האיטרציות. בשביל זה ה-range מאוד יעיל בשביל לסרוק את טווח הערכים הדרוש. (ה-while למשל, אינו מוגדר, כי יכול להיות שנבצע מיליון פעמים, יכול להיות שבכלל לא ניכנס אליו).

גם כאן אנחנו יכולים להגדיר else: שירוץ בסיום הלולאה, בהנחה שהיא הסתיימה בהצלחה, רק יש להקפיד שוב על ההזחה הנכונה לכל דבר (יש הרבה אינטרפרטרים שכבר עושים את ההזחה בצורה אוטומטית, אבל כדאי לוודא ולא לסמוך על זה).

בנוסף, ניתן לתת למשתמש להגדיר את הערכים ואז לקבוע תוך כדי ריצה את הטווח –

```
print ("=== Accumulating a range of values ===")
initVal = int(input("Enter the range initial value : "))
stopVal = int(input("Enter the range stop value: "))
stepVal = int(input("Enter the range step value>: "))
#
acc = 0
for i in range(initVal, stopVal, stepVal):
   acc += i
print ("=== The accumulated value is ",acc, " ===")
```

קבצים

כמו כל שפה, גם פייתון מכיל יכולת עבודה על קבצים. תהליך העבודה על קבצים דומה מאוד למה שכבר פגשנו בשפות C – קודם כל "פותחים" את הקובץ על ידי הגדרה של משתנה שיכיל את הייצוג של הקובץ, ולתוך פונקציית הפתיחה אנחנו מכניסים את הנתיב של הקובץ, ואת סוג הפתיחה שאנחנו מעוניינים בה – קריאה (r), כתיבה (w), או הוספה לקובץ קיים (a – append), כאשר לצורות הכתיבה, אם הקובץ לא קיים, ניתן להגדיר + ליד סוג הפתיחה, ואז אם הקובץ לא קיים כבר, אז פייתון ייצר אחד בשם שהגדרנו.

קריאה מקובץ

בשביל לקרוא קובץ, מגדירים קודם כל את הקובץ ממנו קוראים, ואז מגדירים את סוג הקריאה הדרושה. את הקריאה מבצעים לתוך מחרוזת חדשה. על מנת לקרוא יש לנו שלוש אפשרויות שונות –

- **read** – קריאת כל הקובץ בשלמותו.
- **read(N)** – קריאת N ביטים מהקובץ.
- **readlines** – קריאת כל שורה בנפרד. מה שיתקבל הוא רשימה של רשימות, כאשר הרשימה הראשית תהיה כל השורות, וכל שורה תהיה רשימה בפני עצמה.

```
f=open('text1.txt','r')
first = f.read().split()
f.close()

f=open('text2.txt','r')
second = f.read().split()
f.close()

d1 = histogram(first)
d2 = histogram(second)

print( 'a:','\nfirst file: ', d1, '\n\n\nsecond file: ', d2)

print( '\nNumber of words in file1', len(first) )
print( '\nNumber of words in file2', len(second) )
print( '-'*25 )

f = set(first)
s = set(second)

print( '\nb1 (common words - intersection): ','\n',f&s )
print( '-'*25 )
print( '\nb2 (unique words - xor): ','\n',f^s )
```

ברגע שאנחנו מבצעים קריאה מהקובץ, אנחנו מקבלים סטרינג, ככה שאנחנו יכולים להפעיל עליו את כל המתודות של הסטרינג. ולכן, אנחנו יכולים לעבוד בצורה נוחה אם נגדיר את הקריאה כ-split, ונקבל מערך של כל מילה בנפרד. מה שהקוד כאן מבצע הוא קריאה של כל המילים מהקובץ, ויצירת מילון שיכיל כמפתח את המילה, והערך יהיה מונה של מספר המופעים של המילה בטקסט, כמו שאפשר לראות בקוד הנפרד של הפונקציה –

```
def histogram(L):
    d = dict()
    for word in L:
        if word not in d:
            d[word] = 1
        else:
            d[word] = d[word] + 1
    return d
```

ה-L המתקבל בפונקציה הוא רשימת המילים, שהוא טיפוס איטרבילי. אנחנו מגדירים מילון ריק, ובעבור כל מילה ברשימה, אנחנו בודקים האם כבר קיים מפתח באותו שם. במידה ויש, אנחנו מעלים את הספירה באחד, ואם אין אז אנחנו מגדירים אותו עם ערך ראשוני 1.

אחרי זה אנחנו יכולים גם לראות את העבודה עם הקבוצות שם אנחנו מראים את המילים המשותפות בכל קובץ בעזרת האיחוד &, ולאחר מכן את המילים הייחודיות לכל קובץ בעזרת XOR.

כתיבה לקובץ

כתיבת מחרוזות לתוך קובץ מתבצעת בדיוק באותו אופן כמו קריאה, רק הפוך. גם כאן עלינו לפתוח קובץ לכתיבה, ואז אנחנו מגדירים את סוג הכתיבה שאנחנו מעוניינים:

- **write(S) –** כתיבה של מחרוזת לתוך הקובץ.
- **writelines(L) –** כתיבה של רשימת מחרוזות לתוך קובץ.

לאחר שאנחנו מסיימים את הכתיבה, אנחנו צריכים לסגור את הקובץ לשינויים, על מנת לשחרר את הקובץ.

פונקציות

הצורה הבסיסית להגדרת פונקציה הוא באופן הבא – ראשית מגדירים את הפונקציה עם המילה def, אחרי זה מגדירים את שם הפונקצייה, ואז בסוגריים את שמות הערכים המוכנסים לפונקציה.

הדבר המיוחד בדרך הכתיבה של הפונקציה, הוא שבאופן הזה אנחנו לא מגדירים את הטיפוס אותו אנחנו אמורים לקבל, וגם כאשר אנחנו מחזירים את המשתנה הסופי, אין לנו שם טיפוס מוחזר, כך שהפונקציות מוגדרות פולימורפיות בלי שום הגדרה מיוחדת.

הערה: מבחינת הערך המוחזר – אם לא נגדיר במפורש ערך מוחזר, יחזור לנו None.

חשוב מאוד גם פה לשים לב לאינדנטציה – המגדירה לנו את ה-Scope של הפונקציה. ברגע שאנחנו נחזור לאותה רמה ראשונית של הגדרת הפונקציה, מבחינת האינטרפרטר אנחנו כבר בחוץ.

נסתכל על דוגמת קוד של פונקציה המבצעת השוואה פולימורפית של טיפוסים –

```
def mymax(x, y):
    if x < y:
        retval = y
    else:
        retval = x
    return retval
```

דרך הבדיקה פה היא מאוד פשוטה, אנחנו מקבלים שני ערכים, ומחזירים ב-retval את הערך הגדול מביניהם. כמובן שמאחר ואנחנו מדברים על פייתון, אנחנו לא חייבים להגדיר את הטיפוס, אלא הוא יוגדר בעצמו בזמן הריצה, ולכן אנחנו נוכל לשלוח לכאן מספרים ומחרוזות באופן חופשי. וכן לשלוח קבוצות ורשימות ולעשות עליהם השוואה. אך אם ננסה לשלוח מילון שהוא לא בר השוואה, תיזרק לנו חריגה ולא נוכל להשוות את זה -

```
>>> mymax('abc','defr')
'defr'
>>> mymax([1,2,3], [4,21,7])
[4, 21, 7]
>>> mymax((1,2,3), (4,21,7))
(4, 21, 7)
>>> mymax(set([1,2,3,1]), set([4,21,21,7]))
{1, 2, 3}
>>> mymax({1:'a', 2:'b'}, dict([(3,'c'),(5,'e'), (6,'gg')]))
```

מבחינת סדר הגדרת הפונקציות, יש לוודא שאנחנו מגדירים קודם כל את הפונקציות בהם אנחנו הולכים להשתמש, ובסוף את התכנית הראשית. זאת מאחר, שבזמן הריצה אנחנו צריכים להכיר כבר את הפונקציות ברגע שאנחנו נקראים להשתמש בהם.

ניתן לעשות הגדרת ברירת מחדל לערכים בפונקציה. דבר זה נעשה בעת ההכרזה על הפונקציה, על ידי השמת ערכים ברשימת הארגומנטים, כך שאם נקרא לפונקציה עם ערכים מושמים אחרי, אז נקבל את מה שהכנסנו, ואם לא יהיו לנו ערכים אותם הגדרנו מחוץ למשתמש.

דרך נוספת להעביר ארגומנטים (למשל כאלה שהמשתמש הגדיר), היא לקרוא לפונקציה ולהשים את הערכים בתוך השמות של הארגומנטים הפונקציה המוגדרת. מה שמיוחד פה, הוא שזה לא חייב להיות באותו הסדר שהגדרנו בפונקציה. דבר זה ייראה כך –

```
def accFunc(init, stop, step):
   acc = 0
   for value in range(init, stop, step):
      acc += value
   return acc
#
initVal = int(input("Enter the range initial value : "))
stopVal = int(input("Enter the range stop value : "))
stepVal = int(input("Enter the range step value : "))
acc = accFunc(stop = stopVal, step = stepVal, init = initVal)
print (acc)
#
```

*args

אפשר להעביר רשימת ארגומנטים לא מוגדרת באורכה. מה הכוונה? אם אנחנו לא יודעים, או רוצים לא לדעת כמה ארגומנטים יכניס המשתמש, ניתן להשתמש במשתנה עם * כמובן שלא חייבים לקרוא לזה *args, אלא זה סתם שם מוסכם כזה.

את הפונקציה עצמה אנחנו יכולים להגדיר כמובן בצורה פולימורפית ככה שיגיבו גם למחרוזות וגם למספרים. כדאי מאוד לוודא איזה השמה הכנסנו לערכים על נת לעבוד עליהם בעזרת הפונקציה isinstance, ולעבוד בהתאם –

```
#
# unkeyUnlimitedPrm.py
#
def unlimitadd(*values):
   if values:
     if isinstance(values[0], (int, float)):
       result = 0
     elif isinstance(values[0], str):
       result = ''
     for val in values:
        result += val
   else:
     result = 0
   return result
```

לאחר שהגדרנו כך את התכנית אנחנו יכולים להריץ עליו מספר פרמוטציות שונות ובכל פעם לקבל את הדרוש –

```
>>> unlimitadd(3,5,2,7,10,45)
72
>>> unlimitadd()
0
>>> unlimitadd(3,5,2,7,10,45,50,65,23)
210
>>> unlimitadd(*range(10))
45
>>> unlimitadd(*"hfsdhfiuwehrewo")
'hfsdhfiuwehrewo'
>>> unlimitadd(*(3,5,2,7,10,45))
72
>>> unlimitadd(*[3,5,2,7,10,45])
72
```

באותו אופן, אפשר להעביר גם kwargs** ולהעביר מילון עם ערכים ומפתחות ולעשות עליהם פעולות שונות על ידי הגדרת שמות משתנים לערך:מפתח ואז גם את ערך ההחזרה אפשר להגדיר בצורה של יותר מארגומנט אחד –

```
#
# keyUnlimitedPrm.py
#
def orderTotal(**order):
   # every formal parameter is a part name
   # every actual parameter is the qty of bought units
   inStock = {'p1':100.0, 'p2':250.0, 'p3':50.0}
   notInStock = dict([])
   total = 0.0
   if order:
     for pName, pQty in order.items():
         if pName in inStock:
           total += inStock[pName]*pQty
         elif pName not in notInStock:
           notInStock[pName] = pQty
         else:
           notInStock[pName] += pQty
   return (total, notInStock)
#

>>> partsToOrder = dict([('p1',3),('p3',2),('p2',5),('p4',2),('p5',1)])
>>> partsToOrder
{'p1': 3, 'p3': 2, 'p2': 5, 'p4': 2, 'p5': 1}
>>> orderTotal(**partsToOrder)
(1650.0, {'p4': 2, 'p5': 1})
>>> orderTotal(p1 = 3, p3 = 2, p2 = 5, p4 = 2, p5 =1)
(1650.0, {'p4': 2, 'p5': 1})
```

__name__

דיברנו על כך שפייתון היא שפת סקריפטים, כלומר אין לנו איזה תכנה ש"עובדת" אלא אוסף קבצים אותם אנחנו יכולים להריץ. אם אנחנו מעוניינים להשתמש בקבצים אותם הגדרנו בעבר (ששמורים תחת אותה תיקייה), אנחנו יכולים לייבא את הגדרות התיקייה לתוך הקובץ עליו אנחנו עובדים.

עכשיו, כאשר אנחנו עובדים על קובץ, אנחנו יכולים להריץ אותו, ואת כל מה שמוגדר בו. מה קורה אם אנחנו מייבאים קובץ נוסף להרצה? יותר מזה, נגיד ובקובץ המיובא עשינו כל מיני "טסטים" וכתבנו שם קוד ברמת הזחה 0. כלומר הוא ממש "שייך" לכל החלק המיובא, ואם יש לנו שם הדפסה או פעולה זה עלול להתבצע (הדבר הזה דומה

לתרגילים שעשינו בסדנא עם ההדפסות בתוך הקונסטרקטורים והבדיקה מה יוצא בזמן ירושה). נסתכל על דוגמה פשוטה . נגדיר קובץ בשם mymath.py ובתוכו נוסיף פונקציית הדפסה פשוטה –

```
// mymath.py

def add(a,b):

        return a+b

print (add(3,4)
```

עכשיו בנוסף לפונקציה הוספנו גם שורה הרצה שתדפיס לנו את תוצאת החיבור של 3 ו-4.

נגדיר כעת קובץ בשם test.py. נייבא לתוכו את הקובץ השני, ונדפיס עוד תרגיל חיבור –

```
// test.py

Import mymath.py

Print(mymath.add(5,4))
```

לפי מה שאמרנו עד עכשיו, אם נריץ רק את הקובץ הראשון mymath, האינטרפרטר יעבור שורה-שורה, ובסוף ידפיס לנו את התוצאה7. אבל מה יקרה אם נריץ את test?

הדבר הראשון שיקרה הוא ייבוא של mymath, ולאחר שנעבור על השורות שם, מה שיודפס יהיה זה –

```
7

9
```

מדוע? כי אין שום סיבה שהאינטרפרטר לא יעבור על השורה הזאת.

בשביל להמנע ממקרים כאלה, בפייתון יש מילה שמורה בשם __name__ שמבטא את מרחב ההכרה, ויכול להגיד לנו מהו הקובץ הראשי שמפעיל את הסקריפטים השונים. המודול הראשי, נקרא באופן מפתיע "__main__". איך זה תורם לנו?

אם אנחנו רוצים להגדיר שחלק מהקוד לא יורץ תמיד, אנחנו יכולים להגדיר את החלק הזה תחת בדיקה האם אנחנו נמצאים בקובץ ההרצה או לא. נראה דוגמא על הקובץ mymath שראינו קודם –

```
// mymath.py

def add(a,b):

        return a+b

if __name__ == "__main__":

        print (add(3,4)
```

עכשיו, אם נריץ את הקובץ mymath, פקודת ההדפסה תרוץ כי הוא ראשי. אבל אם מצד שני, נריץ את test, אז בזמן הייבוא, כשנגיע לתנאי, פייתון יבדוק מה ה"שם" של הקובץ mymathעליו הוא רץ כרגע, התשובה שהוא יקבל תהיה "mymath.py", וזה כמובן שונה מ-main. מה שגורם להדפסה לא להתבצע.

תכנות פונקציונלי

לאחר היכרות עם השפה יש לציין - פייתון אינה שפה שבעיקרה מיועדת לתכנות פונקציונלי, אך למרות שהיא לא תוכננה להיות פונקציונלית, יש לה לא מעט תכונות שנותנות את האפשרות לעבוד בצורה כזו.

מה זה תכנות פונקציונלי?

צורת החשיבה בתכנות פונקציונלי שונה מזו האיטרטיבית, התכנות האיטרטיבי הוא זה שנפגשנו בו עד עכשיו, ועיקרו הוא האיטרציות – החזרות על פעולות שונות עד לביצוע הדרוש. בתכנות איטרטיבי אנחנו מסוגלים לקחת משתנה ולעבור עליו שוב ושוב על אותה פונקציה, אך בכל פעם לקבל לפלט שונה.

בתכנות פונקציונלי לאף פונקציה אין תופעות לוואי חיצוניות. למה אנחנו מתכוונים במונח "תופעת לוואי"? אם אנחנו כותבים למשל ב-C:

```
Int f (int x){
        Int z;
        z = y+x;
        y +=1;
        Return (z);
}
```

ולצורך הדיון, נניח ש-y הוא משתנה חיצוני לפונקציה, ובשורה מסוימת אנחנו קוראים לפונקציה הנ"ל בעזרת-

```
n = f(3);
```

במקרה זה, n יקבל 3, ואם נמשיך ונקרא לפונקציה שוב בהמשך, הערך יהיה שונה, מאחר ו-y משתנה בכל כניסה לפונקציה. באופן כללי, ניתן לומר שמקריאה פשוטה של התוכנית, אנחנו לא יכולים לדעת בוודאות מה יהיה הערך המוחזר של הפונקציה, למרות שאנחנו בכל פעם קוראים לה עם אותו קלט. זה דוגמא של תופעת לוואי אחת, ויכולות להיות עוד כמה נוספות שנרחיב עליהן בהמשך. אנחנו דורשים שהפונקציה תהיה "טהורה", כלומר שחישוב התוצאה לא גורם לשינוי חיצוני, וזה אחד מאבני היסוד של תכנות פונקציונלי – אין תופעות לוואי והמידע הוא לא ניתן לשינוי. אנחנו נראה איך עושים את זה בדיוק.

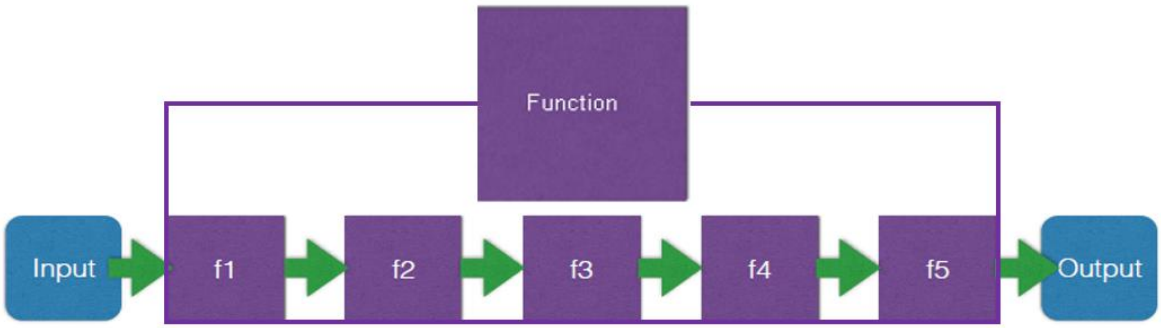
על פי ויקיפדיה – "במחשבים ,תכנות פונקציונלי היא פרדיגמת תכנות השמה דגש על חישוב ביטוי תוך שימוש בפונקציות ככלי ההפשטה העיקריים. זאת בניגוד לפקודות (Statements) שהן הכלי העיקרי של שפות התכנות מהפרדיגמה הנפוצה יותר,הפרדיגמה האימפרטיבית."

"פרדיגמה" זה סט של הסכמות מדעיות מסוימות הנוגעות לתחום מסוים. פרדיגמת התכנות מדברת על דרך הכתיבה, ארגון הקוד וכו'.

פרדיגמת תכנות פונקציונלי, מסתכלת על חישוב בתוכנית כמו על חישוב מתמטי, בו אין "מצב" של תוכן משתנה ונשמר – אם נחפשה הקבלה מתמטית, אנחנו יכולים להבין שאם אנחנו ננסה לבצע הכפלה בין שני משתנים, לצורך העניין, 3 ו-7, לא משנה כמה פעמים נכפיל אותם, התוצאה תמיד תהיה 21. באותו אופן, אם אנחנו נרצה לבצע שימוש בפונקציה כלשהי, אנחנו רוצים לדעת שתמיד נקבל את הפלט האחיד והמתאים לנו, וזאת לעומת תכנות אימפרטיבי, בו יש השמה כמעט לכל שורה.

התרשים הבא מתאר לנו את דרך העבודה הכללי המצופה מאיתנו – יש לנו פונקציה אחת גדולה, אליה אנחנו מכניסים את הקלט. כל שלב של תת הפונקציה, עושה איזה חישוב מסוים ומוציא פלט. אותו הפלט יהיה הקלט של

הפונקציה הבאה, וכן הלאה. למעשה, אפשר לומר שיש פה הרכבה מאוד ארוכה של פונקציות, בדיוק כמו ההרכבה המתמטית, אנחנו משתמשים בפלט של פונקציה אחת בשביל להגדיר את האחרת. מה שיצא בסוף בפלט הכללי, הוא החישוב שעבר בכל הצינור הארוך, וקיבל את השינויים הנצרכים.



Referential Transparency

אם אנחנו קובעים שהנתונים לא ניתנים לשינוי, ושכל פונקציה תהיה טהורה ללא תופעות לוואי, שני הדברים ביחד מהווים את מושג ה"Referential Transparency", שזה השקיפות המובטחת לקבל את אותו הפלט באופן תמידי, עבור אותו קלט.

צורת העבודה הזאת גוררת את המודל החישובי של התכנות הפונקציונלי. מאחר והחישוב הוא כל כך קבוע, אנחנו יכולים להתייחס אליו כאובייקט לכל דבר ועניין. מה שאומר שאנחנו יכולים להשתמש אפילו בפלט של פונקציה לתוך הקלט של עצמה.

יתרונות

- **קביעות תוצאה (idempotence) –** לא משנה כמה פעמים נקרא לפונקציה, נקבל את אותו הקלט.
- **קלות דיבוג (Ease of Debugging) –** אם יש לנו באג, יש לנו רק מקום אחד אותו אנחנו צריכים לבדוק. אין לנו צורך לעבור על באג ארוך עם לולאות ולבדוק באיזה שלב הוא נדפק, כי פשוט אין לנו לולאות.
באימפרטיבי, לא רק שאנחנו יכולים ליפול בלולאה, הנפילה יכולה לבוא בכלל ממקום אחר לגמרי, מאחר ושינויים יכולים לקרות מכל מקום בקוד.
- **שמירת חישובים (Memoization)** – אם יש חישוב משנה שחוזר על עצמו מספר פעמים, במקום לחזור ולחשב אותו בכל פעם מחדש, מה שיגרור לנו זמן ריצה ארוך, אחנו יכולים לשמור את הערך החוזר הזה באיזה מילון מסוים, ופשוט לבקש את התוצאה מהמילון או מבנה הנתונים בו הוא שמור.
- **מודולריות (Modulatiztion) -** אנחנו יכולים להתייחס לפונקציות כקופסה שחורה אם אנחנו יודעים מה יצא אנחנו יכולים לקחת את הפונקציות האלה למקומות נוספים ולאו דווקא למה שהוא נכתב במיוחד. כך שהמודולריות היא לא רק ביחס לאותו הקוד שאנחנו כתבנו, אלא היכולת להשתמש בפונקציות האלה בתור מודולים שונים.
- **מקביליות (Parallelization)** – אם פונקציה אחת לא בדיוק תלויה בשניה, אז תוכנית פונקציונלית יכולה לרוץ במקביל (ואפילו על מעבדים שונים), בלי לחשוש לפגיעות. ואנחנו יכולים להריץ בשתי פונקציות שונות את אותו משתנה בלי לפחד או להתחשב בשינוי שעלול לקרות, אלא שניהם יכולים לעבוד במקביל. למשל, אם יש לנו שורה שכתובה באופן הבא - Res =fun1(a,b)+fun2(a,c). תכנות אימפרטיבי לא יעז לנסות להריץ את שתי הפונקציות האלה במקביל, מאחר ושימוש ב-a באחת מהפונקציות הללו עלול לגרום לשינוי בערך, מה שיגרום תגובת שרשרת של חישוב לא נכון. אך אם מובטח לנו שהמשתנה a יישאר תמיד עם אותו הערך, אין לנו סיבה שלא להריץ את שתי הפונקציות האלו כאחד.
- **חפיפה (Cuncurrency)** – אין מידע משותף, מה שמבטיח ששתי פונקציות יכולות לכתוב במקביל ללא בעיית קוראים-כותבים ואין צורך בנעילות שונות. דבר זה הוא בעצם הרחבה של היתרון הקודם – לא רק

שאנחנו יכולים להשתמש באותו משתנה בשתי פונקציות שונות, אנחנו אפילו יכולים להשתמש בשתי פונקציות שהן שוות לגמרי ולא לחשוש שיהיה באחד מהם שינוי שעלול לפגום בשני.

מה זה משתנה?

במתמטיקה, משתנה הוא שם שאנחנו נותנים לערכים מסוגים שונים. יכול להיות ערכים אטומים, שלמים , רציונליים מרוכבים וכו', ויכול להיות גם ווקטורים, מטריצות וכו' – אנחנו נגדיר עבור כל משתנה ממטי, את שלישיית ההגדרות הבאה, שתכיל לנו את כל המידע הדרוש לעבוד איתו <שם, ערך, סוג של ערך>. יש לשים לב – ה"סוג" של הערך הוא סוג הטיפוס המספרי – מספר טבעי, עשרוני, מרוכב וכו'.

בתכנות אנחנו מדברים על רביעיה - <שם, כתובת, ערך, טיפוס>. אנחנו צריכים לדעת גם היכן שמור כל משתנה בנפרד, ככה שאם נרצה לעשות בו שינויים נוכל להגיע אליו בצורה מהירה.

המשתנה שאנחנו מדברים עליו, בתכנות הפונקציונלי, הוא משהו שנמצא בתווך שבין שתי ההגדרות האלה – אנחנו מגדירים את זה כשלשה של <שם, ערך, טיפוס>, לזיכרון ולכתובת אין משמעות בסוג תכנות כזה, כי אנחנו מסתכלים על הנתונים ברמת הפשטה גבוהה, ואנחנו לא מסתכלים על מקום האחסון, אליו אנחנו ממילא לא אמורים להגיע בשביל לשנות בו משהו.

חישוב נטול מצבים – Stateless Computation

בחלק זה נסתכל על חישוב חסר מצבים. המטרה העיקרית בדבר זה, הוא הרעיון שאנחנו לא צריכים להקצות זיכרון או לשמור מידע שאנחנו לא צריכים אותו, אלא להגיב כמה שיותר מהר על הדרישות.

נסתכל על דוגמה תכנותית ראשונה –

```
def fullName1(firstName, lastName):
   # functional (stateless) style
   return firstName + ", " + lastName

def fullName2(firstName, lastName):
   # functional (once-only assignment) style
   Name = firstName + ", " + lastName
   return Name
```

אם נרצה ליצור פונקציה שמקבלת שם פרטי ומשפחה ומחזירה אותו כשם אחד ארוך, יש לנו מספר אופציות שאנחנו יכולים לעשות את זה – נתחיל דווקא בפונקציה השניה – שם אנחנו שמרנו את הערכים בתוך משתנה פנימי אותו החזרנו. מבחינה תכנותית אין עם זה שום בעיה, ודבר כזה גם מותר (בצורה מוגבלת) בתכנות פונקציונלי, אבל ברור שהדרך היותר נכונה תהיה לקחת את הערכים ולהחזיר אותם בצורה ישירה, כמו בפונקציה הראשונה. אמנם הגדרה של משתנה זמני הוא לא משהו שאמור לעשות בעיה גדולה, אבל בכל כניסה לפונקציה כזו, אנחנו נצטרך להקצות זיכרון לכל המשתנים הקיימים בפונקציה, ובכל יציאה מהפונקציה נצטרך לשחרר אותם. בחישוב הכללי, דבר כזה יהיה ארוך ומיותר.

עוד דוגמא –

```
#
# Dict example
def popAges1(names, ages):
   # functional (stateless) style
   return dict(zip(names, ages))

def popAges2(names, ages):
   # functional (once-only assignment) style
   pairsLst = zip(names, ages)
   return dict(pairsLst)
```

הפעולה המתבצעת בפונקציות אלה, היא שידוך של שתי רשימות לכדי מילון שיכיל את השם כמפתח ואת הגיל כערך. גם כאן, אנחנו מבחינים בשתי דרכי הפעולה השונים – בפעם הראשונה, החזרה ישירה של המילון, בצורה שאפילו לא נשמרת לרגע אחד על משתנה כלשהו. הפונקציה השניה, שומרת לנו קודם את הצמדים הדרושים, ורק אחר כך מגדירה אותו כמילון תוך כדי ההחזרה. כדאי לשים לב, שיש פה השמה יחידנית – אין השמה של הזיפ, המרה למילון ואז החזרה, שכבר עושה לנו בעצם שתי השמות, אלא השמה אחת והחזרה.

```
#
# List example
def yearsList1 (From, To):
   # functional (stateless) style
   return list(range(From, To+1))

def yearsList2 (From, To):
   # functional (once-only assignment) style
   years = list(range(From, To+1))
   return years
```

גם בפונקציה הזאת אין איזה חידוש מיוחד, אלא הדגמה של שתי האפשרויות השונות בתכנות הפונקציונלי – צורה חסרת מצבים והשמה יחידנית.

פונקציות כאובייקטים

דיברנו כבר על כך, שאנחנו יכולים להתייחס לפונקציות בתור אובייקטים. למעשה, כאשר אנחנו כותבים את ההגדרה def ומכריזים על פונקציה, אנחנו בעצם עושים השמה – אנחנו קושרים את שם הפונקציה לכתובת המוקצית לו. במה זה בא לידי ביטוי?

קודם כל, בגלל שזה אובייקט, אפשר להעביר אותו לתוך פונקציות בתוף אובייקטים, וכמובן שאפשר גם להחזיר את הערך של הפונקציה. מעבר לזה, כאשר אנחנו מגדירים איזה מבנה נתונים כלשהו (רשימה, רשומה וכו') אנחנו יכולים להגדיר בתוך השדות ערכים של פונקציה.

נראה דוגמא –

```
>>> x = 10

>>> x

10

>>> def y(name):

        print ("Hello", name)

>>> y
```

```
<function y at 0x000002385A7A5A60>

>>> y("Anna")

Hello Anna

>>> z = 'Sarah'

>>> z

'Sarah'

>>> y(z)

Hello Sarah

>>> z = y

>>> z

<function y at 0x000002385A7A5A60>

>>> z("Anna")

Hello Anna
```

בדוגמא הנ"ל אנחנו יכולים לראות שאנחנו מתייחסים בדיוק באותו אופן הן לשמות המשתנים, והן לשמות הפונקציות. המשתנה Z שהופיע בהתחלה בתור מחרוזת, מופיע בהמשך בתור שם פונקציה

למבדא

נפגשנו בעבר בפונקציות למבדא, שעיקרן הוא הגדרה פשוטה ומהירה לפונקציות שמבצעות פעולות (יחסית) פשוטות. למעשה, מבחינת השפה פייתון, הלמבדא הוא ביטוי ש**מחזיר** לנו ערך של פונקציה, והוא בעצם הביטוי הכי מופשט של טיפוס הפונקציות. אפשר לומר שבעצם הגדרה של ביטוי למבדא זה שליחה לבנאי של הטיפוס "פונקציה" ויצירת מופע מתאים.

מבחינת הסינטקס, כתיבת למבדא מתבצעת באופן הבא –

lambda $a_1, a_2 \ldots a_n$: expression

קודם כל מכריזים על הלמבדא, ורושמים אחר כך את רשימת הארגומנטים שהפונקציה מקבלת. לאחר מכן, מכניסים נקודותיים, ואז רושמים את הביטוי. חשוב לשים לב, למבדא מקבלת רק ביטויים! לא מכניסים פה הצהרות ודברים דומים – הביטוי תמיד מחזיר ערך, וההצהרה לא תמיד מחזירה ערך. אם נכתוב –

```
>> x =5
```

אנחנו נוכל להמשיך כרגיל וכלום לא חוזר, מאחר והיתה השמה אבל היא לא עושה משהו בפועל. אך אם נכתוב משהו כמו –

```
>> x+3

8
```

אז יחזור לנו ערך החישוב, כי אנחנו עשינו פה ביטוי שמחזיר ערך, ולא רק את ההשמה הפשוטה.

אם למבדא יוצרת לנו אובייקט מסוג פונקציה, ואנחנו לא רוצים לתת שם, אז נכתוב את הביטוי למבדא, ואז ישר נכניס בסוגריים את הארגומנטים המועברים.

נסתכל על דוגמא לשתי כתיבות שונות לביטוי למבדא–

```
>>> f = lambda x,y : x + y
>>> f(2,3)
5
>>> lst = ['one', lambda x : x * x, 3]
['one', <function <lambda> at 0x0000023BCBBF6AE8>, 3]
>>> lst[1](4)
16
>>> f2 = lst[1]
>>> f2(4)
16
```

הדוגמא הראשונה היא פשוטה ואינטואיטיבית – אנחנו מגדירים פעולת למבדא, ומכניסים את האובייקט הזה לשם משתנה (f) שאנחנו יכולים להכניס אליו אחר כך ערכים. בדוגמת הרשימה כבר יש חידוש רעיוני – מאחר שהפונקציה היא אובייקט לכל דבר, אנחנו יכולים להעביר אותה לתוך הרשימה בתור איבר. מה שזה ייצור לנו, הוא האפשרות לגשת לאיבר ולשלוח אליו ערכים שיחזירו לנו את תוצאת החישוב. כמו כן, אנחנו יכולים להעביר את ערך האיבר/האובייקט ולתת לו השמה למשתנה חדש שיעשה את אותה פעולה.

סגור (Closure)

לפני שמתחילים את הדיון על "סגור", בדרך כלל נהוג להיסגר על פונקציות מקוננות ומשתנים מקומיים. אנחנו יודעים כבר שיש לנו את האפשרות להגדיר פונקציה בתוך פונקציה, ולגשת לתוך החלק הפנימי יותר ולבצע שם את הפעולות השונות. כאשר אנחנו מגדירים דבר כזה, תחום ההכרה מקבל טווח חדש של הפונקציה הראשית שהתחילה הכל, כך שגם פונקציות המשנה, המקוננות, יכולות לגשת לערכים ה"שייכים" לפונקציה הראשית. נראה דוגמא פשוטה –

```
def outerFunction(text):
    text = text
    def innerFunction():
        print(text)
    innerFunction()
if __name__ == '__main__':
    outerFunction('Hey!')
```

הפונקציה הפנימית מקבלת את ערך הטקסט של הפונקציה החיצונית ומשתמשת בה בלי בעיה.

ה"סגור" הוא אפשרות בה אנחנו משתמשים בתכנות פונקציונלי, המתבצעת בסיוע של פונקציות מקוננות. מה שהסגור נותן לנו הוא האפשרות ליצור אובייקט עם משתנה פנימי אליו אנחנו יכולים לגשת מתוך הפונקציה המקוננת. היוחד פה, הוא זה שש לנו את היכולת להמשיך ולגשת גם לאר שלכאורה הגדרנו ויצאנו מהפונקציה.

נראה גם בהמשך, שהסגור לא מוגבל לפונקציה אחת בלבד, אלא יכול לקבל כמה סוגי ערכים שונים. למעשה, יש כאן בניה דומה למה שאנחנו מכירים בבניית מחלקה עם מידע ופונקציות, רק שכאן אנחנו מסתכלים הפוך על כל העבודה – במחלקות, אנחנו שומרים מידע עם הפעולות האפשריות לביצוע עליו (למשל – יצירת סטודנט, עדכון שם וכו'), ואילו הסגור נותן לנו להגדיר סט של פעולות שיקושר לערכים מסויימים. זה קצת מבלבל, אז נראה דוגמאות כיצד אנחנו עובדים עם זה –

```
>> def print_msg(msg):
>>     def printer():
>>         print(msg)
>>     printer()
>> print_msg("Hello")
```

אנחנו הגדרנו פונקציה בשם print_msg שמקבלת הודעה כלשהי (מחרוזת). בתוכה הגדרנו פונקציה שמדפיסה את ההודעה הזאת, וקראנו לה מקצה הפונקציה הראשית. שימו לב, אנחנו לא קיבלנו לתוך הפונקציה printer את המחרוזת ולא העברנו אותה בשום דרך, אך מאחר והגדרנו את המחרוזת הזאת כמשתנה כללי לפונקציה אנחנו יכולים לגשת אליו גם בפנים.

עכשיו ניקח עוד צעד לכיוון הסגור הפונקציונלי. כמו שחזרנו והדגשנו, אנחנו יכולים להגדיר פונקציות כאובייקטים. מה שאומר, שאנחנו יכולים להגדיר איזה אובייקט שייקח את הפונקציה הזאת כמו שהיא וימשיך להתייחס אליה, השינוי היחיד שנעשה כרגע, הוא במקום לקרוא לפונקציה printer, אנחנו נחזיר אותה כערך לתוך אובייקט -

```
>> def print_msg(msg):
>>     def printer():
>>         print(msg)
>>     return printer
>> another = print_msg("Hello")
>> another()
```

ברגע שהגדרנו את המשתנה another בתוך הפונקציה עם המחרוזת הזאת, אז בכל פעם שנקרא לאוביקט הזה, יודפס לנו "Hello".

על מנת לבצע את הסגור הזה בצורה נכונה, יש לנו מספר תנאים אותם אנחנו צריכים לקיים:

- אסור לנו לשנות את ערך המשתנה בו אנחנו משתמשים, אלא רק לשמור אותו לשימושים חוזרים.
- עלינו להחזיר פונקציה אחת בלבד.
- ברגע שהסגור מוגדר, הוא עובר בצורה פונקציונלית טהורה (לא משנה ערכים מבחוץ לו, ומחזיר תמיד את אותו הערך).

נסתכל על דוגמאות השימוש המופיעות במצגת –

```
def paintPlan(*Colors) :
      return lambda color : color in Colors
#
clients = ('MrsCohen','MrsLevi','MrsKeshet')
plans   = (paintPlan('white', 'light blue', 'yellow'),\
           paintPlan('black', 'red', 'blue', 'white'),\
           paintPlan('gray','brown','dark blue', 'white'))
paints = dict(zip(clients, plans))
#
print (paints['MrsCohen']('white'))
print (paints['MrsCohen')
```

אנחנו מגדירים כאן פונקצית למבדא, שמקבלת כמות לא מוגדרת של צבעים (תזכורת ה-* לפני הארגומנט, אומר שמספר הארגומנטים עלול להשתנות ואינו קבוע), ופונקציית הלמבדא עצמה מקבלת ערך של צבע בודד ובודקת האם הצבע מופיע ברשימת הצבעים שנכנסו קודם. לאחר מכן אנחנו יוצרים רשימה, של "תוכניות הצבעים" האלה שמוגדרות כפונקציה, ומרצ'רצ'ים את המידע לפי שם לקוח כמפתח לתוך מילון. לאחר מכן אנחנו ניגשים למילון הזה (הגישה מתבצעת עם השם בין []), ושולחים צבע לבדיקה (על ידי שליחת צבע בסוגריים), ומה שמוחזר לנו הוא האם הצבע הזה נמצא או לא בתוך תוכנית הצבעים.

השימוש פה בסגור הוא בעצם כמה רמות מעל מה שציפינו לו - בעצם המפתח של הלקוח הוא איזה ערך אמורפי של פונקציית למבדא כלשהי, שכידוע גם זה אובייקט לכל דבר בפייתון, אם נסתכל עכשיו על ההדפסה של הפונקציה (שעשיתי שינוי קטן בשורה השניה של ההדפסה), נקבל את הדבר הבא –

```
True
<function paintPlan.<locals>.<lambda> at 0x00ABB780>
```

כלומר – אם נקרא למפתח עם ארגומנט מתאים כלשהו, נקבל אמת או שקר, תלוי בתוצאה, אבל אם נבקש רק את הערך עצמו לאותו מפתח, נקבל את הפונקציית הלמבדא בצורה הטהורה שלה.

יש דוגמאות בהמשך, כיצד אפשר להשתמש בסגור בשביל לקבל תוצאות דומות רק עם אמת/שקר או ערכים שונים, אבל הדוגמא המקיפה ביותר היא דוגמת המחשבון –

```
def makeCalc():
   def add(x,y):
      return x+y
   def mult(x,y):
      return x*y
   def div (x,y):
      return x/y
   def minus (x,y):
      return x-y
   def dispatch(op):
      opNames   = ('add', 'mult', 'div', 'minus')
      Ops = (add, mult, div, minus)
      if op in opNames:
        return Ops[opNames.index(op)]
      else:
        print ("invalid operator")
        sys.exit()
```

```
    return dispatch
#
c1 = makeCalc()
c2 = makeCalc()
print (c1('add')(2,3))
print (c2('mult')(2,3))
```

כאן אנחנו מגדירים שני אובייקטים של מחשבון. כאשר אנחנו מגדירים את האובייקט הנ"ל, מה שחוזר לנו הוא הפונקציה dispatch, שמגדירה את הפעולות האפשריות, ומגדירה שאם אנחנו שולחים לאובייקט פעולה ומספר, לבצע את הפונקציה המתאימה. ככה אנחנו יכולים לבצע בעצם פעולות אריתמטיות בצורה של הגדרה מראש (prefix).

בעצם ההגדרה פה היא דומה מאוד להגדרת מחלקה של מחשבון שהיינו עושים בעבר, אבל הגדרת הפונקציה בצורה הזאת, הרבה יותר נוחה ופחות תופסת מקום.

פונקציות כפרמטר

במסגרת השימוש בפונקציות כאובייקטים, אנחנו יכולים לכתוב פונקציה שחלק מהפרמטרים המתקבלים שלה זה פונקציה –

```
def foo(f, L) :
    L1 = list(L[:])
    for i in range(len(L)):
        L1[i] = f(L[i])
    return L1

def bar(x) :
    return x * x
```

בדוגמה לעיל, הפונקציה foo מקבלת רשימה כלשהי, אותה היא מעבירה לרשימה חדשה, ומכניסה לכל תא את ערך הפעולה שהפונקציה f מבצעת על תא מסוים. למשל, אם הגדרנו את הפונקציה bar כמחזירה הכפלה של איבר בעצמו, אנחנו נוכל אחר כך ליצור רשימה כלשהי של ערכים (בדוגמה במצגת אנחנו יוצרים איזה טווח) וההכנסה של הרשימה הזאת, עם הפונקציה bar, תית לנו בחזרה את הרשימה של חזקות 2 באותו טווח.

פונקציות כאלה, המקבלות כפרמטר פונקציה אחרת, נקראות **פונקציות-על**. המקרה הזה, בו אנחנו מקבלים פונקציה ורצים על קלט מסוים נקרא "מיפוי", ואנחנו נראה תיכף כיצד אנחנו יכולים להשתמש בפונקציות מוגדרות בשפה בשביל לעשות זאת בקלות.

חלק מהפונקציות שראינו, יכולות לקבל גם צורה של פונקציות על כאלה, בשביל להרחיב את האפשרויות ולבצע קצת מעבר למה שהן מסוגלות בצורה ה"פרימיטיבית" שלהן. נראה דוגמא –

```
>>> L1 = ['a',5,'abc',3]
>>> max(L1)
Traceback (most recent call last):
  File "<pyshell#197>", line 1, in <module>
    max(L1)
TypeError: '>' not supported between instances of 'int' and 'str'
>>> max(L1,key=str)
'abc'
```

אם נגדיר את הרשימה L1 בצורה של טיפוסים מעורבבים, כמובן שזה לא יהיה לנו פשוט להגדיר מה המקסימום של הרשימה. ואכן אם ננסה, תחזור לנו הודעת שגיאה, שאנחנו לא מסוגלים להשוות בין int ל-str. אבל אנחנו יכולים

להגדיר את הפונקציה שמוצאת מקסימום, גם באופן של הכנסת מפתח שיהווה מדד להשוואה. במקרה הזה נקבל את המקסימלי מבין כל הסטרינגים הקיימים ברשימה.

```
>>> sorted(L, key=str)
['Python', 'This', 'a', 'from', 'is', 'string', 'test']
>>> sorted(L, key=str.lower)
['a', 'from', 'is', 'Python', 'string', 'test', 'This']
>>> studLst = [('Jack', 'Math', 80), ('Naomi', 'Math', 75), ('Doron', 'CS', 90)]
>>> from operator import itemgetter
>>> sorted(studLst, key=itemgetter(0))
[('Doron', 'CS', 90), ('Jack', 'Math', 80), ('Naomi', 'Math', 75)]
>>> sorted(studLst, key=itemgetter(1))
[('Doron', 'CS', 90), ('Jack', 'Math', 80), ('Naomi', 'Math', 75)]
>>> sorted(studLst, key=itemgetter(2))
[('Naomi', 'Math', 75), ('Jack', 'Math', 80), ('Doron', 'CS', 90)]
>>> sorted(studLst, key=itemgetter(1,2))
[('Doron', 'CS', 90), ('Naomi', 'Math', 75), ('Jack', 'Math', 80)]
>>> sorted(studLst, key=itemgetter(1,2), reverse=True)
[('Jack', 'Math', 80), ('Naomi', 'Math', 75), ('Doron', 'CS', 90)]
```

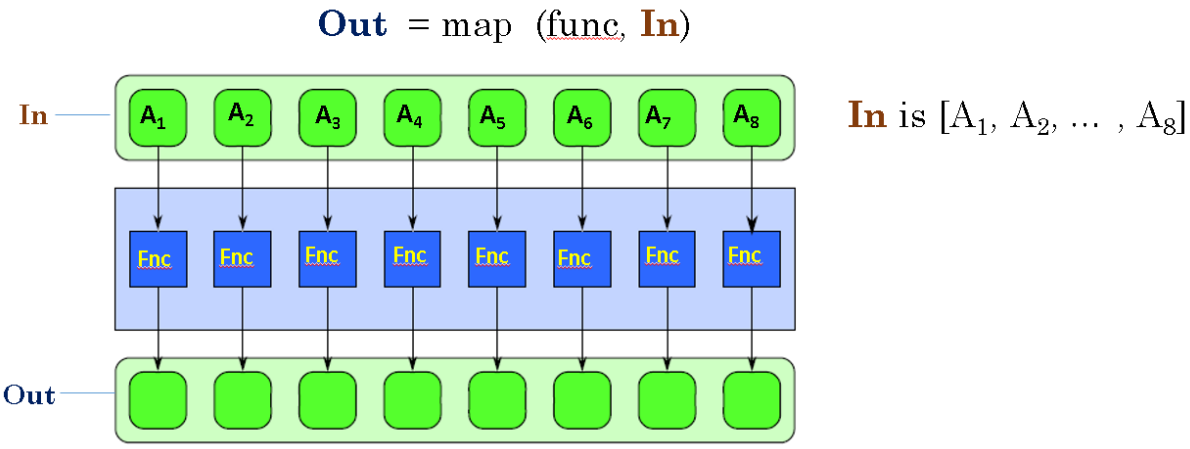
עכשיו, נניח שהגדרנו רשימה של מחרוזות, ואנחנו רוצים למיין אותה. אנחנו יכולים לעשות מיון רגיל לפי מפתח str. אבל אם נעשה ככה, אנחנו יכולים לראות שהמילים עם האותיות הגדולות בתחילתן, נמצאות לפני כל המילים עם האותיות הקטנות, מה שיוצר שהמילה 'a' אינה ראשונה. מבחינת פייתון זה נכון, כי הוא בודק לפי הערך ה-ascii שם אותיות גדולות מופיעות לפני אלו הקטנות. במקרה שבו נרצה לקחת את הצורה היחסית ולמיין לפי סדר אלפבתי רגיל, נוכל להגדיר את המפתח כ str.lower שבזמן ההשוואה יוריד את כל המחרוזות שיכילו רק אותיות קטנות ולפי זה למיין. יש לשים לב – אנחנו מדברים על פונקציות טהורות, כלומר שינוי המילים למילים "נמוכות" מתבצע על רשימה מקבילה ולא משנה את הערכים ברשימה המקורית.

פונקציה נוספת שאנחנו מכירים פה היא itemgetter. זה פונקציה מתוך מודול של אופרטורים, שבמקרה ויש לנו רשימה מקוננת, יכול לקחת ולהוריד את הזום לאיבר מסוים. כך שאם נרצה, נוכל לעשות מיון על הרשימה רק לפי ערך מסוים בכל איבר. אנחנו יכולים גם לקחת יותר מאיבר אחד ולעבוד על מיון לפי שני ערכים, כאשר אם הגדרנו שנעשה את המיון על המחלקה והציון (לפי הדוגמא (1,2)), אז הוא קודם ממי את הערך הראשון, ואחרי זה עושה מיון פנימי לתוך הקטגוריות האלו.

בדוגמא האחרונה, אנחנו יכולים לראות איך גם הוספנו מאפיין של מיון ברברס, שמחזיר בדיוק את אותה רשימה רק הפוכה.

פונקציות-על מוגדרות בשפה

Map



נסתכל על פונקציות על בשפה המוגדרת – map.

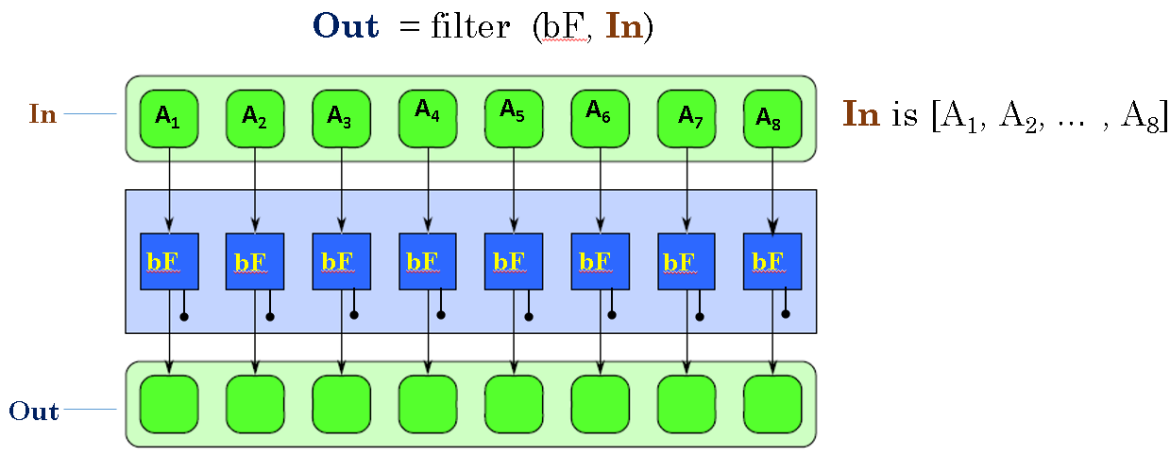
דיברנו כבר קודם על עצם המיפוי – קבלה של פונקציה ורשימת ערכים, והחזרה של כל תוצאות הערכים השונות. התרשים שלמעלה מתאר בעצם את דרך הקלט והפלט – עבור כל ערך שנקבל בפלט, נקבל ערך שיוצא ממנו בפלט. כדאי שוב לשים לב, שבדומה לפונקציות כמו טווח ופונקציית זיפ, גם פה אנחנו לא מחזירים רשימה או מילון או כל מבנה נתונים מסוים, אלא אנחנו מחזירים מיפוי. את אותו המיפוי אנחנו צריכים להעביר הלאה למבנה הנתונים שאנחנו רוצים לעבוד איתו.נראה דוגמא לעניין -

```
>>> map(lambda x : 2*x, [5,8,3,9])
<map object at 0x000001A33E7F3F28>
>>> list(map(lambda x : 2*x, [5,8,3,9]))
[10, 16, 6, 18]
>>> list(map(lambda x,y : x+y, [5,8,3,9], [1,4,2,3]))
[6, 12, 5, 12]

>>> from math import pi, sin
>>> list(map(sin, [pi/2,pi,3*pi/2,2*pi]))
[1.0, 1.2246467991473532e-16, -1.0, -2.4492935982947064e-16]
>>> |
```

אם נמפה פונקציית למבדא שמכפילה ערך נתון ב-2, ונכניס רשימה מתאימה, אנחנו נקבל כערך מוחזר את העובדה שיש לנו אובייקט מסוג map. נשלח אותו שוב תחת מסגרת של רשימה, ונוכל לראות גם את התוצאות המתאימות. באופן דומה, אנחנו יכולים להגדיר שהלמבדא תקבל שני ערכים, והמיפוי יעבור על כל איבר i ברשימה ויחזיר לנו תוצאות.

שימוש בפונקציה הזאת חוסך לנו המון בכתיבה – במקום ליצור איזה לולאה עם מונה שתרוץ על הכל ותעשה בדיקות וכו', אנחנו פשוט מגדירים את הכל בפשטות בשורה אחת ומחזירים תוצאה.

Filter



פילטר מממש גם הוא את התבנית של פונקצית על. בשונה מפונקצית מיפוי רגילה, הפילטר מקבל פונקציה בוליאנית. כך שמה שיחזור לאחר הסינון, הם רק הערכים המתקיימים תחת התנאי שהבאנו.

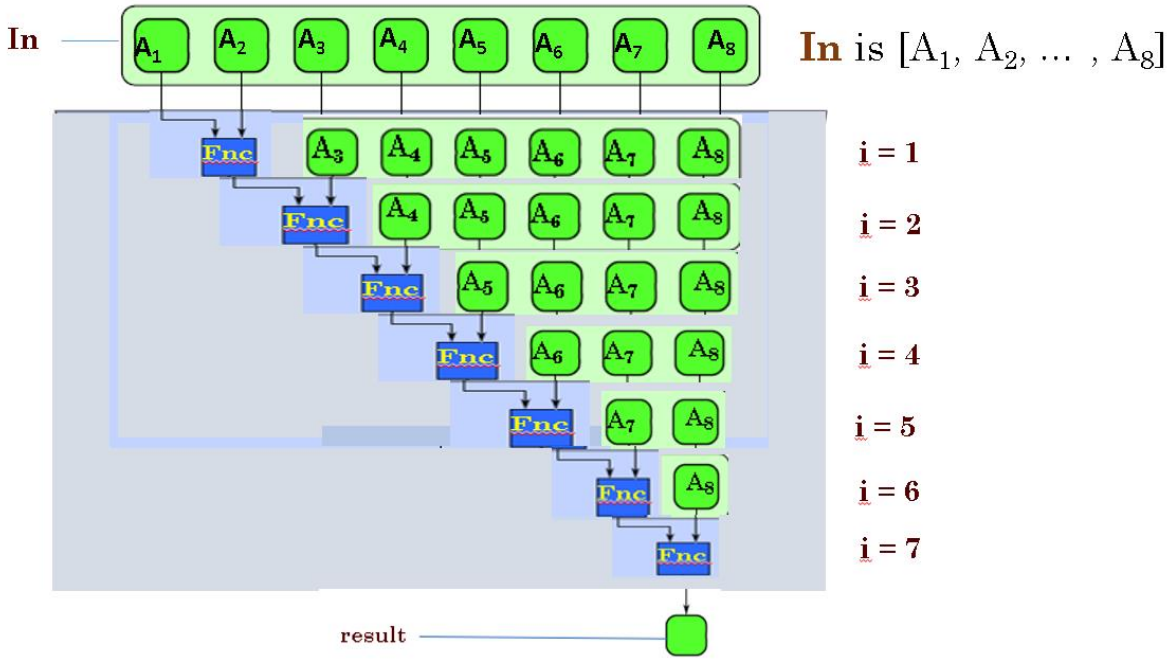
```
>>> filter(lambda x : isinstance(x,str), [4,3,"abc",5,"grty"])
<filter object at 0x000001A33E77F748>
>>> list(filter(lambda x : isinstance(x,str), [4,3,"abc",5,"grty"]))
['abc', 'grty']
>>> def isEven(x):
        return isinstance(x,int) and x%2 == 0

>>> isEven(3)
False
>>> isEven(4)
True
>>> list(filter(isEven, [4,3,"abc",5,"grty"]))
[4]
```

הסינון הראשון שאנחנו מבצעים,הוא בדיקה לפי טיפוס ערך – מה שמקיים את התנאי, והוא מהטיפוס הדרוש יעבור את הסינון, ומה שלא – לא.

אנחנו רואים בנוסף, אפשרות להגדיר פונקציה בוליאנית שלמה ולא רק בצורה של למבדא, ואליה להכניס רשימה המורכבת מטיפוסים שונים, ומה שיוחזר אלו רק ערכי הספרים השלמים המתחלקים ב-2.

Reduce

פונקציית ה-reduce, או הצמצום, הייתה פונקציה מוגדרת בשפה בגרסאות המוקדמות יותר של פייתון, ועכשיו יש לייבא אותה מתוך מודול שנקרא functools. פעולת הפונקציה היא קבלה של פונקציה שמקבלת מספר ערכים, ומעבר על כל רשימת הקלט שהכנסנו בצורה מדורגת. כלומר- אם יש לנו פונקציה שמקבל שני ערכים, אנחנו מתחילים בשני הערכים הראשונים ברשימת הקלט, ומפעילים עליהם את הפונקציה שהבאנו. את התוצאה של השימוש בהם, אנחנו מעבירים הלאה, ומשתמשים עם האיבר העוקב ברשימה. כך אנחנו ממשיכים, עד שהתוצאה הסופית תהיה מורכבת מהפעלת הפונקציה על כל הקלט. נתון התרשים הבא, שמתאר בצורה כללית את ההשתלשלות -



נראה דוגמת שימוש פשוטה בפילטר -

```
>>> from functools import reduce
>>> reduce(lambda x,y : x+y, range(10))
45
>>> reduce(lambda x,y : x*y, range(10))
0
>>> reduce(lambda x,y : x*y, range(1,10))
362880
>>> fact1 = lambda N : reduce(lambda x,y : x*y, range(1,N+1))
>>> fact1(5)
120
>>> def fact2(N):
        return reduce(lambda x,y : x*y, range(1,N+1))

>>> fact2(5)
120
>>> s1 = "this "
>>> s2 = "is "
>>> s3 = "a string."
>>> reduce(lambda x,y : x+y, [s1, s2, s3])
'this is a string.'
```

אנחנו יכולים בעזרת ה-reduce להגדיר פונקציות פשוטות של סכימה ועצרת (רק יש לשים לב, כמו בדוגמה, להתחיל את העצרת מ-1 ולא מ-0), וכן להגדיר את הטווח בצורה של סגור שלמדנו קודם.

השלב הבא, הוא כמובן האפשרות לשלב בין שלושת הפונקציות, מה שבעצם מביא אותן לידי הכוח האולטימטיבי שלהן -

```
>>> reduce(lambda x,y : x+y, filter(lambda x : x%2 == 0, range(10)))
20
>>> reduce(lambda x,y : x+y, filter(lambda x : x%2 != 0, range(10)))
25
>>> L = list(map(lambda x : 2*x, filter(lambda x : x%2 != 0, range(10))))
>>> L
[2, 6, 10, 14, 18]
>>> map(lambda x : pow(x,2), L)
<map object at 0x000001A33E7F3F28>
>>> list(map(lambda x : pow(x,2), L))
[4, 36, 100, 196, 324]
>>> reduce(lambda x,y : x+y, list(map(lambda x : pow(x,2), L)))
660
```

הדוגמאות האלה הן כמובן פשוטות מאוד, ואפילו קצת מיותרות -במקום לעשות פילטר של מה שמתחלק ב-2 (זוגי), אנחנו יכולים פשוט את הטווח להגדיר בצורה יותר נכונה, לחסוך לנו קצת. אבל הרעיון הוא להבין את היכולת לבצע סינון על גבי סינון של הקלט. בנוסף, אנחנו יכולים להרכיב את התוצאות וליצור תוצאה סופית שמורכבת מהכל.

All\Any

שתי פונקציות בוליאניות, שאנחנו יכולים לנחש פחות או יותר מה הן עושות, הפונקציות מקבלות פונקציה בוליאנית כתנאי, רשימת ערכים, ובודקת את התנאי עבור כל הערכים. הפונקציה all תחזיר אמת אם **כל** הערכים עומדים בתנאי שהגדרנו. לעומת זאת הפונקציה Any תחזיר אמת אם **קיים** איזה ערך אחד שמתקיים ברשימה –

```
>>> L = [1,4,5,7,9]
>>> map(lambda x : isinstance(x,int), L)
<map object at 0x000001A33E77F748>
>>> list(map(lambda x : isinstance(x,int), L))
[True, True, True, True, True]
>>> all(map(lambda x : isinstance(x,int), L))
True
>>> L.append(1.5)
>>> L
[1, 4, 5, 7, 9, 1.5]
>>> list(map(lambda x : isinstance(x,int), L))
[True, True, True, True, True, False]
>>> all(map(lambda x : isinstance(x,int), L))
False
>>> any(map(lambda x : isinstance(x,int), L))
True
```

פונקציה בתוך פונקציה

ראינו כבר דוגמאות של הגדרת פונקציה פנימית. אנחנו יכולים להגדיר פונקציה מקוננת, ובצורה מקומית להשתמש בזה בתור ערך מוחזר. כמובן שמחוץ לתחום ההכרה של הפונקציה אי אפשר להשתמש בפונקציה הזאת – אני לא יכול לקרוא לפונקציה מקוננת מתחום ההכרה הראשי, אלא רק לגשת אליה דרך הפונקציה הראשית שלה. נראה וגמה פשוטה לכך -

```
def foo (x,y) :
    def bar (z) :
        return z * 2
    return bar(x) + y
```

לעומת זאת אם אנחנו מחזירים ממש את ערך הפונקציה, אנחנו יכולים לשלוח ערך כפול לתוך הפונקציה, כך שאחד מהם יגדיר את הפנימי יותר –

אנחנו שולחים את שני הערכים, כאשר כל מסגרת של סוגריים מבטאת הגדרה של תחום הכרה של פונקציה וכניסה לעומק הפונקציות. לפי איך שהגדרנו גם את הפונקציה foo, אם נשלח אליה רק ערך בודד, מה שנגדיר יהיה דווקא הערך הפנימי של bar, מאחר ואת ה-x אנחנו הגדרנו כבר כברירת מחדל 0.

```
def foo (x = 0) :
    def bar(y) :
        return x + y
    return bar
# main
print (foo(1)(3))
f = foo(3)
print (f)
print (f(2))
```

ביטוי תהליכים איטרטיביים בעזרת רקורסיה

אנחנו יודעים שלפתרון בעיות באופן כללי, אנחנו לוקחים את הבעיה המורכבת, הקשה, ובשביל לפתור אותה אנחנו מזהים את תת-הבעיות הקטנות יותר שהן קלות יותר לפתרון. כל בעיה כזאת שתהיה לנו עדיין קשה, אנחנו נמשיך ונפרק עד שנגיע לבעיות טריוויאליות שמהוות ממש את הבסיס הכי קטן של הבעיה. ירידה כזאת לפתרון בעיה, יוצר לנו מעין עץ של בעיות, כאשר הענפים השונים יהיה המעברים מהבעיות הקשות, והעלים יהיו לנו בעיות שהן קלות לפתרון. כך שאם נעבור על כל העץ שנפרש לנו בצורה של bottom-up (כלומר, לפתר את הבעיות הקטנות, העלים, ואז בעזרת התשובה המתקבלת לפתור את הבעיות הגדולות יותר) אחנו יכולים להרכיב בחזרה את הפתרון לבעיה כולה, וזה שיטת "הפרד ומשול" לפתרון בעיות. השיטה הזו כמובן טובה לאו דווקא ייחודית לתכנות אלא בחיים בכלל.

רקורסיה היא מקרה פרטי של הפרד ומשול. יש לנו בעיה שאנחנו לא מסוגלים לפתור, אבל אנחנו כן יכולים לזהות את תת-הבעיה, שהא בדיוק אותו דבר רק קטן יותר, ואנחנו ממשיכים לחפש שלב בוא אנחנו יכולים לעצור ולפתור את הבעיה ולחזור לשאלה המקורית. כמובן שברקורסיה יש תנאי נוסף לפירוק הבעיה - אנחנו יוצאים מנקודת הנחה שאנחנו מדברים על תת-בעיה עם מבנה דומה לזו הגדולה, ובמקרה הקטן ביותר אנחנו מדברים על המקרה הבסיסי, שהוא השתקפות של כל ביה גדולה שלא תהיה.

דבר זה דומה להוכחה באינדוקציה – אנחנו יודעים שיש מקרה בסיסי אליו אנחנו יודעים את הפתרון מראש, ובהנחה שיש פתרון לתת-הבעיה הקטנה, אנחנו יכולים לפתור את הבעיה הגדולה בעצמה.

נראה מספר דוגמאות פשוטות ומוכרות של רקורסיה –

$$0! = 1! = 1$$

$$n! = n * (n-1)!$$

כאשר אנחנו רוצים לחשב עצרת של מספר כלשהו, אנחנו יודעים שמה שעלינו לעשות הוא להכפיל את המספר n שקיבלנו, בעצרת של המספר הקודם לו. ומה המספר הזה שווה? הכפלה שלו בעצרת הקודם, וכן הלאה. מתי אנחנו יודעים שאנחנו יכולים להפסיק ולהקטין את הערכים? אם נגיע לעצרת של 1 (או 0). במקרה כזה אנחנו יודעים שאנחנו צריכים להחזיר 1. ואם נכנסנו בצורה רקורסיבית, אז נעלה למעלה בחזרה, נכפיל ב-2, ואת התוצאה ב-3, וכן הלאה.

אותו הדבר עם חזקה - חזקת 0 זה 1, חזקת 1 זה המספר עצמו, וכל השאר זה הכפלת הערך הקודם בעוד מופע של המספר –

$$b^0 = 1$$
$$b^1 = b$$
$$b^n = b^{n-1} * b$$

דבר שראינו גם בפרולוג – חישוב אורך של רשימה – רשימה ריקה זה 0. כל דבר אחר, זה תוספת של 1 לערך הקודם.

```
Lst == [] → len(Lst) = 0
Lst != [] → 1 + len(Lst[1:])
```

אנחנו מעדיפים להשתמש ברקורסיה על פני לולאת שונות, מאחר ובלולאות יש שינוי מצב (ולו רק שינוי פנימי בערך המונה) ואנחנו שואפים שהתכנות הפונקציונלי יהיה ללא שינוי מצב. בעזרת הרקורסיה ושימוש בפונקציות על, אנחנו יכולים לתכנת ללא שינוי מצב.

כמובן שאפשר לשאול, שהרי בעצם קריאה לרקורסיה מתבצעת עם שינוי של המספר עליו אנחנו מחפשים תוצאה? אבל בעצם, כאשר אנחנו קוראים בצורה רקורסיבית, אנחנו מעבירים בכל פעם מופעים חדשים ולא את אותם ערכים שעוברים שינוי. (אין איזה מונה של ערכים, אלא שינוי של קריאה לרקורסיה חדשה עם משתנה חדש).

כאשר ניתן לפרק בעיה לבעיות קטנות יותר, אז הבעיה היא ניתנת לפתרון רקורסיבי, וזהו הנקודה שאנחנו צריכים לקחת בחשבון – יש בעיות שאינן רקורסיביות, ואנחנו נצטרך לפתור אותם בצורה אחרת.

אנחנו שואפים לעבוד כמה שאפשר עם רקורסיות זנביות. הסיבה היא שבאופן כזה, אנחנו חוסכים לנו מקום בזמן הריצה – אם יש לנו רקורסיה לא זנבית, אז בכל כניסה לרקורסיה, אנחנו דורשים מהמערכת לבנות איזה מתחם הכרה חדש עבור הפונקציה הנקראת. אך אם אנחנו קוראים בצורה של רקורסיה זנבית אנחנו מרימים רק רמה אחת של מחסנית, ולא קוראים הלאה לרמות נוספות.

פייתון לא תומך באופן בנוי בשפה ברקורסיה זנבית, אבל יש מודול שנקרא tailrecurse, שמאפשר לנו לתכנת רקורסיות זנביות שעובדות בצורה הראויה ללא תוספת למחסנית קריאות. אחרת, גם אם נגדיר רקורסיה זנבית, על פי כל החוקים שלמדנו, פייתון ימשיך לעבוד בצורה הרגילה ועל כל כניסה לרקורסיה יפתח לנו רמה חדשה במחסנית הקריאות, טיפ לתרגילי הבית – אם אתם מנסים לעשות רקורסיה זנבית והיא לא עובדת, תוודאו שהשתמשתם במודול הזה, כי אחרת לא רק שזה יעבוד בצורה לא חסכונית, הוא גם יעבור לא נכון (כמובן שהוראות אלו נכתבו בדם).

נראה דוגמא לאותה פעולה בצורות שונות –

```
#
# fact.py
#
from tailrecurse import *
#
# Non-recursive, functional version
from functools import reduce
def fact1(N):
        return reduce(lambda x,y : x*y, range(1,N+1))
#
# Non-tail recursive version
def fact2(N):
   if N in (0,1):
     return 1
   else:
     return N * fact2(N-1)
#
# Tail recursive version
@tail_call_optimized
def fact3(N):
   def Tfact(N,result):
      if N in (0,1):
        return result
      else:
        return Tfact(N-1, N*result)
   return Tfact(N,1)
#
```

בחלק הראשון, זה עם פונקציות על, כמו שכבר ראינו.

החלק השני הוא עצרת בצורה של רקורסיה לא זנבית – התוצאה מתקבלת רק לאחר שאנחנו יורדים לרמה הנמוכה ביותר, ומתחילים להחזיר למעלה את התוצאה המתאימה בצורה צוברת. מבחינת המחסנית, בכל פעם שאנחנו נכנסים לרקורסיה אנחנו שומרים רמה של מחסנית, שבדרך חזור אוספת את התוצאה שיש לנו ומכפילה ברמה מעל – כלומר, מאחר וחישוב התוצאה מתבצע רק בדרך חזור, אז אנחנו צריכים לשמור את כל הרמות הנכנסות, בשביל שנדע את מה להכפיל, ורק אחרי זה לשחרר את המקום שנתפס. כמובן שזה תקין, אבל זה תופס הרבה מקום בזיכרון, ובמקרה הגרוע עלול לסתום את הזיכרון.

במקרה השלישי, אנחנו עושים אופטימיזציה לצורה של רקורסיה זנבית, על ידי סימון @tail_call_optimized, שגורם למנוע של פייתון לחשב בצורה זנבית, בלי לשמור רמות מיותרות של רקורסיות. אחרי זה אנחנו צריכים להוסיף איזה ארגומנט שיהיה הצובר של התוצאה שתחזור בסוף הריצה. ואת הרקורסיה עצמה אנחנו בונים בפונקציה חדשה שם אנחנו שולחים את הפרמטר הצובר של המשוואה בכל פעם לתוך הכניסה לרקורסיה – ליתר דיוק, אנחנו מחשבים את מה שדרוש לנו, ורק אז אנחנו שולחים הכל בחרה לכיוון פונקציית הרקורסיה.

אפשר לשים לב, שברקורסיה הלא-זנבית תנאי העצירה מחזיר לנו את מקרה הבסיס – 1, ממנו אנחנו מחזירים את התוצאה כלפי מעלה. אך הרקורסיה הזנבית מחזירה לנו בתנאי העצירה את התוצאה בעצמה, אותה חישבנו כל הדרך למטה.

(הערה: הסימן @ מכונה decorator, ומבטא שימוש במודולים שונים על מנת לייעל או לעשות שינויים בריצת המערכת).

עד עכשיו, מה שהראינו היה חישוב אריתמטי יחסית פשוט. בדוגמאות הבאות אנחנו נראה שימוש במעבר על מבני נתונים איטרביליים, או בשמם המוכר יותר – רשימות. בגדול מה שאנחנו צריכים לזכור, זה שבעזרת ה-slice אנחנו

יכולים לגשת לערכים במקומות מסויימים (התחלה/סוף) בצורה קלה יחסית, וגם לשרשר חלקים שונים של הרשימות. גם פה אנחנו נראה את שלושת הפתרונות שאנחנו שואפים להשתמש בהם: פונקציות על, רקורסיה רגילה, ורקורסיה זנבית.

החזרת אורך רשימה

כדוגמא ראשונה, נסתכל על האפשרות להשיג אורך של רשימה נתונה (כמובן שלהחזיר Len של רשימה זה הדרך הנכונה, אבל אנחנו מתעסקים יותר במימוש, ולא בפתרון הבעיה.

פונקציות על

השימוש הפשוט ביותר, והקל לכתיבה, גם אם הוא לא הכי אינטואיטיבי –

```
def mylen1(L):
   return sum(map(lambda x : 1, L))
```

אנחנו יוצרים לנו פונקציית למבדה שממפה את הרשימה באופן מאוד פשוט – מרכיבים רשימה חדשה המכילה רק אחדות, אחד כנגד כל איבר ברשימה המקורית. לאחר שאנחנו יוצרים את הרשימה הזאת, אנחנו פשוט עוברים בעזרת פונקציית sum שסוכמת את כל איברי הרשימה (כזכור – אחדות), מה שיביא לנו למעשה את אורך הרשימה.

רקורסיה לא זנבית

```
def mylen4(L):
   if isempty(L):
     return 0
   else:
     return 1 + mylen4(L[1:])
```

בכל רקורסיה שמתעסקת ברשימות, עלינו לזכור שמקרה הבסיס זה רשימה ריקה. אם נקבל רשימה כזאת, כמובן שהאורך שלה הוא 0. אם נקבל כל רשימה שהיא לא-ריקה, אנחנו נחתוך את האיבר הראשון, ונשלח את שאר הרשימה לתוך הרקורסיה, כאשר בחזרה החוצה אנחנו נוסיף +1 על כל כניסה לרמה חדשה.

רקורסיה זנבית

```
@tail_call_optimized
def mylen1(L, length=0):
    if L == []:
      return length
    else:
      return mylen2(L[1:], length+1)

def myLen1b(L):
   @tail_call_optimized
   def Tlen(L,length):
      if L == []:
        return length
      else:
        return Tlen(L[1:], 1 + length)
   return Tlen(L,0)
```

יש לנו כאן שתי פונקציות עם רקורסיה זנבית, שהמימוש שלהם הוא כמעט אותו דבר. כמו שכבר אמרנו קודם, על מנת ליצור רקורסיה זנבית, עלינו ליצור משתנה נוסף שיהווה הצובר שלנו ואיתו נעבוד. נתחיל דווקא בפונקציה השניה – צורת העבודה שם דומה למה שראינו גם בפרולוג – אנחנו מגדירים פונקציה חדשה, ושם מכניסים את הפרמטר הנוסף, כאשר את הקריאה עצמה לרקורסיה, אנחנו מתחילים עם שליחה של מונה מאופס (או רשימה ריקה,

תלוי בצורך). בפונקציה הראשונה, אנחנו דווקא קצת מתחכמים, ומגדירים את מונה האורך בצורה דפולטיבית, שיהיה 0, כך שגם אם אנחנו שולחים פרמטר בודד, אנחנו נוכל להתחיל ישר לרוץ על הרקורסיה.

כך או כך, לפני שאנחנו נכנסים לרקורסיה עצמה, אנחנו דואגים להוסיף למונה האורך את ה+1 שיוסיף לנו לתוצאה הסופית.

החזרת רשימה בעלת איברים שהם רשימה מקוננת של שני איברים

אמנם הבקשה פה נראית מאוד ספציפית בצורה קצת מוגזמת, אבל זה יפתח לנו את הדיון להרבה פונקציות שנראה בהמשך – רשימות מקוננות.

מאחר ורשימה יכולה להכיל איברים שהם רשימות בעצמם, אנחנו צריכים להיות ערוכים לאפשרות שחלק מהאיברים הם אכן רשימות. יש לנו מספר כלים שכבר הכרנו, שמטרתם לבדוק את הטיפוס שאיתו אנחנו עובדים. נשתמש בפונקציות אלו, ונראה כיצד הפונקציות הללו עוזרות לנו לפתור את הבעיה.

שימוש בפונקציות על

```
from functools import reduce
def getpairs1(L):
   return filter(lambda isinstance(item,(list,tuple)) and len(item) == 2, L)
```

אנחנו משתמשים בפונקציית הסינון שראינו – אנחנו מגדירים פונקציה בוליאנית בעזרת הלמבדא שמקבלת שני תנאים – קודם כל צריך שה-item, כלומר האיבר ברשימה, יהיה מסוג רשימה (או רשומה), ואם הוא אכן כזה, אנחנו בודקים האם אורך הרשימה הזאת שווה ל-2. כזכור – הפילטר מחזיר רק את הערכים שמתקיימים בתנאי, וברגע שיש כאלה, אנחנו מחזירים את הרשימה הנתונה למי שקרא לפונקציה.

רקורסיה לא-זנבית

```
def getpairs2(L):
   if L == []:
     return []
   elif isinstance(L[0],(list,tuple)) and len(L[0]) == 2:
     return getpairs2(L[1:]) + [L[0]]
   else:
     return getpairs2(L[1:])
```

דבר שנשים אליו לב, בדרך כלל הרקורסיות יותר ארוכות מהשימוש בפונקציות על. מבחינת תנאי הבדיקה, אנחנו מחפשים בדיוק את אותו דבר, איברים שהם רשימות ושאורכם 2. אם נמצא אחד כזה, נדאג לצרף את האיבר הזה כשנחזור מהקריאה לאחר שנמצא שהרשימה התרוקנה. אם האיבר הנתון אינו עומד בתנאים – נמשיך ונחפש פשוט בדילוג האיבר הזה.

רקורסיה זנבית

```
def getpairs3(L):
   @tail_call_optimized
   def Tgetpairs(L,Lout):
      if L == []:
        return Lout
      elif isinstance(L[0], (list,tuple)) and len(L[0]) == 2:
        return Tgetpairs(L[1:], Lout+[L[0]])
      else:
        return Tgetpairs(L[1:], Lout)
   return Tgetpairs(L,[])
```

אנחנו מגדירים לנו פונקציית עזר, שם אנחנו מוסיפים משתנה שיהיה הרשימה המוחזרת. בכל פעם שאנחנו רואים משהו מתאים, אנחנו מוסיפים את האיבר הזה לרשימה (בגלל זה אגב, התוצאה הסופית תהיה הפוכה ברשימה לעומת הרקורסיה הלא זנבית – מאחר והוספת האיברים נעשית בצורה הפוכה). שוב, את הקריאה לרקורסיה הזנבית אנחנו מבצעים עם ערך "ריק", כמו שקודם שלחנו 0, כאן אנחנו שולחים רשימה ריקה.

רקורסיית עץ

כל מה שראינו עד עכשיו זה רקורסיה על רצף ליניארי של ערכים – בין עם אובייקטים רגילים, ובין אם אובייקטים איטרטיביים, את כל אלו אפשר לפתור עם רקורסיה פרימיטיבית (כניסה בודדת לתוך פונקציית רקורסיה). אבל כידוע לנו מכל פעם שמזכירים את הביטוי "רקורסיה", יש לא מעט מקרים בהם התוצאה תלויה ביותר מחישוב בודד אחורה, והדוגמה הקלאסית לזה, היא כמובן מספרי פיבונאצ'י – סדרת מספרים המוגדרת בכל פעם עבור ערך של איבר בתור חיבור שני האיברים שלפניו.

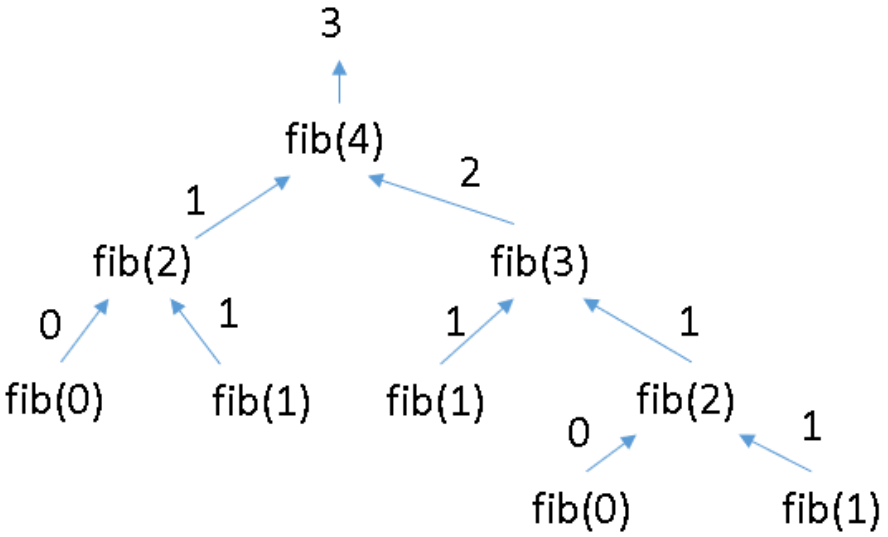
שוב, על מנת להבין איך הרקורסיה בנויה, אנחנו צריכים להסתכל על מקרה הבסיס (שבמקרה שלנו יורכב משני מקרים כאלה, ואז עלינו לבדוק את הנוסחה למקרה הכללי -

fib(0) = 0 fib(1) = 1	מקרים בסיסיים
fib(n) = fib(n-2) + fib(n-1)	מקרה כללי

אנחנו מדברים פה רק על מספרים חיוביים (החל מ-0), ולכן עבור 0 נחזיר 0, ועבור 1, נחזיר גם כן את עצמו. כל מספר אחר החל מאינדקס 2, יורכב משני האיברים שקדמו לו (כך שהאינדקס השני נשאר על 1, והחל משם זה מתחיל לגדול).

מבחינה ויזואלית, אנחנו מסתכלים על הרקורסיה הזאת בצורת עץ (בדומה למה שראינו במבנה נתונים ב'), כאשר כל רמה של העץ מתפצלת בהתאם לכמות הכניסות לרקורסיה – במקרה שלנו לשני ענפים בכל פעם.

כך שבשביל לחשב את מספר הפיבונאצ'י הרביעי, אנחנו צריכים לבדוק את חיבור המספרים השני והשלישי. את הראשון אנחנו מכירים (1), והשני הוא איחוד של הראשון וה-0, כלומר גם הוא 1. והמספר השלישי הוא איחוד של השני והראשון – כלומר 2, וחיבור של 1+2 הוא כמובן 3. תבדקו אפילו בעצמכם. נוכל לראות את העץ באופן הבא -



<u>רקורסיה לא-זנבית</u>

חישוב לא-זנבי של הרקורסיה הזאת, הוא יחסית פשוט – נכנסים בכל פעם עם n-2 ו n-1. ואת מקרי העצירה אנחנו מגדירים שיחזירו כמו שצריך –

```
def fib1(N):
   if (N == 0):
     return 0
   elif (N == 1):
     return 1
   else:
     return fib1(N-1) + fib1(N-2)
```

אם זה 0 או 1, אנחנו מחזירים אותם בעצמם, אחרת אנחנו בודקים רקורסיבית את הפיצולים. ברגע שנגיע בכל רמה לעלים, אנחנו מתחילים לחזור למעלה ולאחד את התוצאות עד לכניסה הראשית.

רקורסיה זנבית

רקורסיה זנבית היא יותר בעייתית כאן, מהסיבה הפשוטה שקשה לחשב את התוצאה של מספר שעוד לא הגענו אליו. מה שאנחנו עושים בשביל זה הוא כניסה סימולטנית עם חישוב, כך שהמספר שקיבלנו יהיה המונה למספר הרקורסיות, ואנחנו דואגים להכנס פעם אחת "יותר מידי", מה שיביא לנו את החישוב הסופי –

```
def fib2(n):
  @tail_call_optimized
  def fibIter(a, b, count):
     if (count == 0):
       return  b
     else:
       return fibIter(a+b, a, count-1)
  return fibIter(1, 0, n)
```

אנחנו מקבלי את n ושולחים אותו ביחד עם מקרי הבסיס של 0,1. ובתוצאה הסופית מחזירים את הנמוך מבין שניהם, ככה שאם נכנס לנו 0 ישר יחזור 0. בכל כניסה לרקורסיה אנחנו מזיזים את a שיהיה הנמוך יותר ברמה הבאה, ושמים במקומו חיבור של שני האיברים. נראה איזה דוגמת ריצה מייצגת (בערך) –

Fib2(4)

fibIter(1,0,4)

fibIter(1,1,3)

fibIter(2,1,2)

fibIter(3,2,1)

fibIter(5,3,0)

return 3

ואת זה כבר בדקנו שהוא נכון, ואנחנו יכולים להמשיך הלאה.

רקורסיית עץ על רשימות ורשימות מקוננות

שוב, אנחנו עכשיו עוברים לרמת הרשימות המקוננות. אם אנחנו רוצים לבדוק איזה תנאי מסוים ברשימה מקוננת, אנחנו צריכים הרבה פעמים להכנס לתוך האיברים ברשימה הפנימית יותר. ברמה הרקורסיבית, זה יוצר לנו גם כן עץ, רק שהוא לא אחיד – יכול להיות לנו שם רשימה באורך שלוש ואחרי זה רשימה ארוכה יותר או קצרה יותר, ואנחנו צריכים להתמודד עם כולם בלי קשר לאורך הרשימה. בדוגמאות הבאות, אנחנו מקבלים רשימות של איברים מטיפוסים מעורבים, ואנחנו צריכים לבדוק האם כל איברי הרשימה הם מספרים (שלמים או עשרוניים).

בשביל להשלים את המשימה ולחסוך לנו קצת בכתיבה, אנחנו קודם כל מגדירים פונקציה שתעשה את הבדיקה – תקבל איבר ותחזיר האם הוא מספר או לא -

```
def isnumber(X):
   return isinstance(X,(int,float))
```

עכשיו נטפל בצורות הכתיבה הפונקציונליות השונות (במצגת מופיעות פונקציות העל לפני הרקרוסיות, אבל אני אשאיר לפי הסדר שעשינו עד עכשיו)–

רקורסיה לא-זנבית

```
def allnumbersp3(L):
   if len(L) == 1:
     return isinstance(L[0], (int, float))
   else:
     return isinstance(L[0], (int, float)) and allnumbersp3(L[1:])
```

אם יש לנו החל מאיבר בודד ברשימה, אנחנו פשוט בודקים האם הוא מספר או לא, ומוסיפים לזה תנאי בוליאני, שיבדוק על שאר הרשימה. כמובן, שהתוצאה הסופית תהיה רק כאשר נחזור מסוף הרשימה ויחזרו לנו כל הערים הבוליאניים – אם הכל יהיה True, אז הרשימה אכן לפי מה שדרשנו.

רקורסיה זנבית

כאן יש לנו שתי אופציות לכתיבה. כדאי לשים לב להבדל הגדול בין שניהם, שהוא לא משמעותי מבחינת התוצאה, אבל כל המחשבה מאחורי הפונקציות היא שונה לגמרי –

```
@tail_call_optimized
def allnumbersp1(L): # without an accumulator parameter
   if L == []:
     return True
   elif isnumber(L[0]):
     return allnumbersp1(L[1:])  # this is a tail call
   else:
     return False
```

פה אנחנו מקבלים את הרשימה, ובכל פעם בודקים רק את האיבר הראשון, אם הוא לא מספר, אנחנו ישר מחזירים False, ואם הוא מספר אנחנו ממשיכים ושולחים את זנב הרשימה בצורה רקורסיבית. את ההחזרה הסופית שהרשימה מורכבת רק ממספרים, אנחנו מבצעים רק אם עברנו על כל הרשימה והגענו בסוף למסקנה שאין שם איברים לא מספריים.

נראה עכשיו את הדרך השניה, שהיא קצת יותר דומה למה שעשינו עד עכשיו בצורה הזנבית –

```
@tail_call_optimized
def allnumbersp2(L, result = True):
   if L == []:
     return result
   else:
     return allnumbersp2(L[1:], result and isinstance(L[0], (int, float)))
```

הגדרנו ערך ברירת מחדל של תוצאה בערך True. בכל כניסה לרקורסיה, אנחנו מכניסים לערך הזה את תוצאת הבדיקה עבור התוצאה הנוכחית, ותוצאת הבדיקה עבור האיבר הראשון. ומכניסים את זה הלאה. כאשר הרשימה ריקה אנחנו מחזירים את התוצאה הסופית האם פגשנו איברים שהם לא מספרים בדרך.

כדאי לשים לב – שתי התוכניות מחזירות לנו את אותה תוצאה עבור כל רשימה שלא תהיה, אבל מבחינת זמן ריצה, ברור שהפונקציה הראשונה הרבה יותר יעילה. הפונקציה השניה רצה על כל הרשימה ורק כשאנחנו מסיימים לרוץ על

הרשימה חוזר לנו התוצאה, אבל אם נבדוק רשימה שכבר האיבר הראשון שלה הוא לא מספר? יותר מזה – אם נקבל רשימה של אלף איברים בלי שום מספר אחד בתוכו?

הפונקציה השניה בכל מקרה תרוץ על כל הרשימה, לעומת הראשונה שברגע הראשון שהיא תראה איבר שהוא לא מתאים ישר תקפוץ החוצה. כך שכמובן הפונקציה הראשונה יעילה הרבה יותר (במקרה הממוצע) מהשניה.

פונקציות על

ברגע שהגדרנו את פונקציית הבדיקה, אנחנו יכולים להתחיל להשתמש בפונקציות העל, בשביל להחזיר לנו את התוצאה הזאת בצורה הרבה יותר קלה -

```
def allnumbersA(L): # using the filter high order function
  return len(L) == len(filter(isnumber, L))
```

אנחנו יכולים להשתמש בפונקציית הפילטר – לשלוח את הפונקצייה הבוליאנית, ולשלוח אותה שתרוץ על כל הרשימה. אם כל הרשימה אכן מורכבת ממספרים, אז אורך הרשימה המתקבלת מהפילטר יהיה בדיוק אותו אורך של הרשימה המקורית, ואז נדע שהכל בסדר.

אפשרות נוספת הוא לעשות מיפוי של כל הרשימה, ואז לצמצם את כל התוצאות הבוליאניות ולבוק את התוצאה הסופית -

```
def allnumbersB(L):
  # using a composition of map and reduce high order functions
  return reduce(lambda x,y : x and y,map(isnumber,L))
```

קודם כל אנחנו עושים את המיפוי על הרשימה, מה שיוצר לנו רשימה בוליאנית. אחרי זה אנחנו מגדירים את פונקציית הצמצום בעזרת למבדא שבודקת and על שני האיברים הראשונים ברשימה (ובהמשך על התוצאה והאיבר הראשון), אם הכל יהיה True, אז זה מה שיחזור מהפונקציה, ואם יהיה אפילו False אחד, הכל ייפול.

האפשרות השלישית והקצרה ביותר הוא שימוש ב-all

```
def allnumbersC(L): # using a composition of the "all" function and
                    # the map high order function
  return all(map(isnumber, L))
```

אנחנו ממפים את הרשימה באותו אופן כמו קודם, אבל אז מעבירים את כל זה לפונקציית-העל all שבודקת האם **כל** הערכים ברשימה הם True. כמובן, שזה קצת יותר יעיל, באותו אופן שדיברנו על הפונקציות הקודמות – אם ניתקל בFalse אחד, כבר אז נוכל לדעת שלא הכל אמת, ולהחזיר תשובה.

עכשיו אנחנו נעלה רמה לרשימה מקוננת. כאן, אנחנו נוכל להראות אפשרויות רק על ידי רקורסיות ופונקציות על.

רקורסיה

```
def nestedallnumbers1(L):
   if L == []:
     return True
   elif isinstance(L[0],list):
     return nestedallnumbers1(L[0]) and nestedallnumbers1(L[1:])
   elif isnumber(L[0]):
     return nestedallnumbers1(L[1:])
   else:
     return False
```

קודם כל אנחנו בודקים אם האיבר הראשון הוא רשימה. אם כן, אנחנו מכניסים גם אותו וגם את שאר הרשימה לתוך הפונקציה שבודקת (כמובן שיכולות להיות כמה רמות של קינון, ולכן אנחנו נבדוק את זה בכל פעם מחדש). אם

האיבר הוא לא רשימה, אנחנו נבדוק אם הוא מספר – אם כן, אנחנו נשלח את שאר הרשימה להמשך הבדיקה. הפונקציה תסיים לרוץ באחת משתי אפשרויות – אם הרשימה ריקה, אז הכל בסדר, ומחזירים True, ואם מגיעים לאיבר שהוא לא מספר (ולא רשימה) מחזירים False, שישורשר למעלה.

אותו דבר עם פונקציות על – באופן דומה למה שראינו כבר במקרים קודמים. אנחנו מגדירים פונקציה שבודקת האם האיברים ברשימה הם מספרים, ועושים את זה גם בצורה רקורסיבית. אחרי זה אנחנו שולחים הכל למיפוי שבודק לנו ומחזיר רשימה שקולה עם ערכים בוליאניים, ואז אנחנו מצמצים הכל עם reduce שעובר ובודק עם למבדא שהכל עם ערכי אמת.

```
def nestedallnumbers2(L):
   # using high-order functions reduce and map, together with recursion
   def allnumbersInItem(item):
      if isinstance(item,list):
        return nestedallnumbers2(item)
      elif isnumber(item):
        return True
      else:
        return False
   return reduce(lambda x,y : x and y, map(allnumbersInItem,L))
```

אפשרות נוספת רק עם פילטר –

```
def nestedallnumbers3(L):
   # using high-order function filter, together with recursion
   def allnumbersFilter(L):
      def numbersInItem(item):
         if type(item) != list:
           if isnumber(item):
             return True
           else:
             return False
         else:
           return (len(item) == len(allnumbersFilter(item)))
      return filter(numbersInItem, L)
   return (len(L) == len(allnumbersFilter(L)))
```

גם זה בדיוק כמו מה שראינו קודם – רצים על הרשימה עם פילטר, ובודקים ששום דבר לא השתנה ברמת אורך הרשימה ומחזירים את התוצאה הסופית המשוקללת מכל ההחזרות המקוננות.

ועכשיו – היפוך רשימה –

<u>רקורסיה לא זנבית</u>

```
def recReverse(L):
   if L == []:
     return L
   return ([L[-1]] + recReverse(L[:-1]))
```

הפונקציות האלה יחסית פשוטות – אנחנו משתמשים ביכולת של פייתון לעשות את הסלייס גם מהסוף, ואנחנו דואגים שמה שיחזור בכל פעם הוא האיבר האחרון, ואת שאר ההיפוך של כל הרשימה מלבד האיבר האחרון. ברגע שנגיע

לרשימה הריקה, מה שייכנס ראשון יהיה האיבר הראשון ברשימה המקורית, ואחריו השני וכו', מה שייצור בעצם היפוך של הרשימה.

רקורסיה זנבית

```
@tail_call_optimized
def tailReverse(L, resL = []):
    if L == []:
      return resL
    return tailReverse(L[:-1], resL + [L[-1]])
```

אנחנו מתחילים עם רשימה שתהיה הרשימה המוחזרת, ומתחילים את הריצה עם רשימה ריקה. בכל שלב של כניסה לרקורסיה, אנחנו חותכים את האיבר האחרון מהרשימה ומשרשרים אותו לסוף הרשימה המוחזרת, כך שהאיבר האחרון יהיה ראשון, והכמעט אחרון יהיה שני, עד שרשימת המקור מתרוקנת, ואנחנו יכולים להחזיר את התוצאה.

מה קורה עם רשימה מקוננת?

```
def deepReverse(L):
    if L == []:
      return L
    if isinstance(L[-1],list):
      return [deepReverse(L[-1])]+deepReverse(L[:-1])
    else:
      return ([L[-1]]+deepReverse(L[:-1]))
```

את הצורה הזאת כבר ראינו בכל המקרים הקודמים – אם האיבר האחרון הוא רשימה, אנחנו שולחים אותו להיפוך, ואז מחזירים אותו משורשר לסוף, לאחר שכבר הפכנו את האיברים הפנימיים שלו. זאת בניגוד לפונקציות הרקורסיביות שמתייחסות רק להיפוך בכל רמה. כדאי לשים לב, שאם נשלח את אותה רשימה מקוננת לכל הפונקציות שהראינו, אז רק הפונקציה deepReverse שראינו עכשיו, תדאג לטפל ברשימות הפנימיות יותר.

תבניות תכנותיות

אנחנו נדבר על תבניות תכנותיות, בדגש על היישום שלהם בתכנות פונקציונלי.

כמו שאמרנו קודם, אנחנו שואפים להגיע לתכנות נטול-מצבים, כלומר ליצור פעולות שישפיעו כמה שפחות מעבר לפעולה שאנחנו רוצים לבצע – אם אנחנו רוצים לקבל רשימה עם אספקט מסוים של ערכים, אנחנו רוצים לקבל אותה באופן ישיר, ולא על ידי השמות שונות. עד עכשיו על מנת לפתור את הדרישות האלה, השתמשנו בפונקציות-על או ברקורסיות, ועכשיו אנחנו נעבור לכלי שנקרא List-Comprehension (תפיסת רשימה, או משהו בסגנון).

List comprehension

ה-List Comprehension הוא כלי מאוד חזק בפייתון, ש"הושאל" מהשפה הפונקציונלית Haskell. היא עובדת בצורה כל כך מוצלחת ויעילה, שזה הסיבה לכך שלא ראו צורך לממש בשפה את האפשרות של רקורסיה זנבית. ככה טוב. המימוש של זה הוא בצורה שהיא נטולת-מצבים לגמרי, ולכן מושלם לתכנות הפונקציונלי.

מבחינה סינטקטית, אנחנו כותבים עם סוגריים מרובעות, ששני חלקים הם חובה והשלישי הוא אופציונלי –

L = [output-expr **for** item **in** input-iterable **if** Boolean-expr]

את החלקים אנחנו מגדירים לפי מילות הקישור (המודגשות) – output-expr – הפונקציה אותה אנחנו נפעיל על האיברים. Item **in** input-iterable – הגדרת הקלט והאיברים שמרכיבים את הקלט האיטריבי. **If** boolean-expr – חלק זה הוא חלק הרשות, אנחנו יכולים לשלוח ישירות את כל האיברים במכה אחת לתוך פונקציה, ואנחנו יכולים להציב איזה תנאי, עבורו רק ערכים שיקיימו אותו יבצעו את הפעולה הנתונה.

עבור כל תבנית תכנותית, אנחנו נציג את ההגדרה המתמטית שלה, ואת המימוש התכנותי המתאים.

תבנית Map

את הרעיון של map כבר הסברנו קודם – הפעלת פונקציה על כל הערכים. אנחנו כעת נכנה את התבנית הזאת כתבנית "לכל" – לכל איבר באיטרבל, בצע פעולה מסויימת.

ההגדרה המתמטית – $Out = \{x \mid \forall A\ (A \in In \ \wedge x = F(A))\}$

כלומר, הפלט x מתקיים עבור כל A ששייך לקלט ונעשתה עליו פעולה של הפונקציה F. Out יכיל מאגר של x-ים שכאלה שעבור שינוי בפונקציה.

מבחינת הסינטקס התכנותי, אנחנו כותבים את זה באופן הבא –

Out = [F(A) for A in In]

שימו לב – אנחנו לא כותבים פה את המילה map. אנחנו מדברים פה על תבנית תכנותית, והביטוי שכתבנו תחת הסוגריים המרובעות מחזיר לנו בדיוק את אותו הדבר, ומאחר שלא הגדרנו איזה תנאי, הפעולה אכן מתבצעת כ**מיפוי של כל הערכים**.

אם היינו רוצים לממש את התבנית הזאת בצורה של פונקציה פשוטה, כל שנצטרך הוא לכתוב את הדבר הבא -

```
def MapLCompr(Fnc, In):
    return [Fnc(item)  for item in In]
```

פשוט נקבל פונקציה וקלט, ונחזיר את התוצאה של הפעולה על כל איבר.

בעבור הצורה הרקורסיבית של המימוש, אנחנו מחזירים רשימה שלתוכה אנחנו משרשרים בכל פעם את תוצאה הביצוע של האיבר הראשון, ואת שאר הרשימה –

```
def MapRec(Fnc, In):
```

```
    if In == []:
      return []
    else:
      return [Fnc(In[0])] + MapRec(Fnc, In[1:])
```

כמובן שזה רקורסיה לא זנבית. רק כשנגיע לרשימה ריקה, אנחנו נחזור בחזרה ונשרשר את התוצאות של המיפוי. אם נרצה רקורסיה זנבית, נצטרך ליצור רשימה ריקה כבסיס, ואז להכניס אותה ולהוסיף אליה בכל כניסה לרמה נוספת של הרקורסיה –

```
@tail_call_optimized
def MapTailRec(Fnc, In, Result = []):
    if In == []:
      return Result
    else:
      return MapTailRec(Fnc, In[1:], Result + [Fnc(In[0])])
```

כדאי לזכור – כל אלו הם רק הצעות למימוש, שהיינו צריכים לבצע אם לא היתה לנו את האפשרות של ה- List Comprehension, אבל מאחר ויש לנו אנחנו יכולים לחסוך לנו המון ולעבוד בצורה נקיה.

תבנית filter

כמו בפונקציית-העל, גם כאן אנחנו מחפשים להחזיר רק את הערכים שיקיימו תנאי בוליאני כלשהו (bF עומד בתור פונקציה F שהיא בוליאנית b). נגדיר את זה מתמטית בעזרת הביטוי הבא –

$$Out = \{x \mid \forall A\ (A \in In \wedge bF(A) \wedge x = A)\}$$

ובמילים – אנחנו מחזירים x כלשהו, כאשר ה-A המקורי ממנו יצאנו שייך לחלק הקלט In, ואם נכניס את A לתוך הפונקציה הבוליאנית bF, יחזור לנו ערך אמת, ובנוסף לכל, הערך המוחזר הוא אותו A בעצמו.

שימו לב – עדיין לא ביצענו כאן שום פעולה, אלא רק הגדרנו את הסנן על פי התנאי הבוליאני.

הסינטקס בצורת ה-List Comprehension ייראה כך –

Out = [A for A in In if bF(A)]

שוב – המוחזר הוא A מתוך In אם הוא מחזיר אמת מ-bF.

אם נרצה לבנות פונקציה שתחזיר לנו רשימה מסוננת (כדאי לזכור, אנחנו מחפשים ליצור תבניות שיתאימו ליותר ממקרה פרטי אחד, ולכן יצירת פונקציה שמבצעת סינון היא הדבר המתבקש והחסכוני יותר), נכתוב את הדבר הבא –

```
def FilterLCompr(bF, In):
    return [item  for item in In  if bF(item)]
```

פשוט נחזיר את רשימה האייטמים שמקיימים את התנאי, קל ופשוט.

עכשיו נראה את הצורות הרקורסיביות שהיינו יכולים לעשות בשביל לממש את אותו הרעיון –

```
def FilterRec(bF, In):
    if In == []:
      return []
    elif bF(In[0]):
      return [In[0]] + FilterRec(bF, In[1:])
    else:
      return FilterRec(bF, In[1:])
```

כאן בעצם יש לנו שלושה תנאים –

1. אם הרשימה ריקה מחזירה אותה.
2. אם האיבר הראשון עומד בתנאי, מחזירים אותו בשרשור לתוצאות שאר הרשימה.
3. אם האיבר הראשון לא מחזיר אמת, אז מכניסים את זנב הרשימה שוב לרקורסיה ללא האיבר הראשון.

פה אנחנו כבר רואים יותר כמה נחסך לנו בזה שאנחנו כותבים משפט אחד רציף.

נראה עכשיו אתת הרקורסיה הזנבית –

```
@tail_call_optimized
def FilterTailRec(bF, In, Result = []):
   if In == []:
     return Result
   elif bF(In[0]):
     return FilterTailRec(bF, In[1:], Result + [In[0]])
   else:
     return FilterTailRec(bF, In[1:], Result)
```

עושים את אותם פעולות ואותם בדיקות, רק שאנחנו דואגים ליצור רשימה להחזרה, אנחנו מתחילים עם רשימה ריקה, ומשרשרים לה בכל פעם עד שאנחנו מגיעים לריקון רשימת הקלט, שם אנחנו מחזירים את התוצאה.

תבנית "לכל"+מסנן

שילוב שתי התבניות הקודמות, מאפשר לנו **לסנן** תוצאות מתוך רשימה, ועל **כל** האיברים המוחזרים להפעיל פעולה מסוימת. מבחינה מתמטית מה שאנחנו דורשים הוא כזה –

$Out = \{ x \mid \forall A\ (A \in In \wedge bF(A) \wedge x \in F(A))\}$

שימו לב, שמבחינת ההגדרה המתמטית השינוי היחיד שהתבצע הוא השייכות של x לפונקציה על האיבר ולא לאיבר עצמו. באותו אופן, הביטוי התכנותי יהיה דומה מאוד –

Out = [F(A) **for** A **in** In **if** bF(A)]

אנחנו מכניסים את שלושת החלקים, ומפעילים את הפונקציה על התוצאה.

נעבור מהר על המימושים השונים –

```
def mapFilterLCompr(bF, func, In):
   return [func(item)  for item in In  if bF(item)]
```

צריך לזכור שברגע שאנחנו רוצים לעשות map+filter, אנחנו צריכים לדאוג לקבל חוץ מהקלט גם את הפונקציה הבוליאנית לסיון, וגם את הפעולה על האיברים.

נראה את המימושים הרקורסיביים –

```
def mapFilterRec(bF, func, In):
   if In == []:
     return []
   elif bF(In[0]):
     return [func(In[0])] + mapFilterRec(bF, func, In[1:])
   else:
     return mapFilterRec(bF, func, In[1:])
```

שוב, ההבדל היחיד פה במימוש לעומת הסינון הרגיל, הוא שאת האיבר הראשון (במידה והוא יעבור את הסינון) אנחנו מכניסים לרשימה לאחר שהפעלנו עליו את הפונקציה הדרושה.

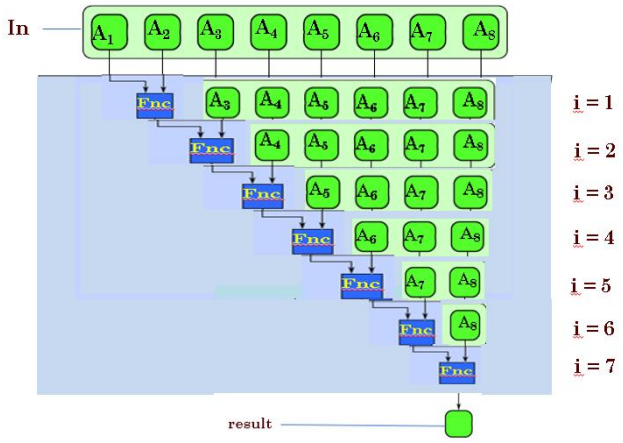
אותו דבר גם ברקורסיה זנבית –

```
@tail_call_optimized
def mapFilterTailRec(bF, func, In, Result = []):
    if In == []:
      return Result
    elif bF(In[0]):
      return mapFilterTailRec(bF, In[1:], Result + [func(In[0])])
    else:
      return mapFilterTailRec(bF, In[1:], Result)
```

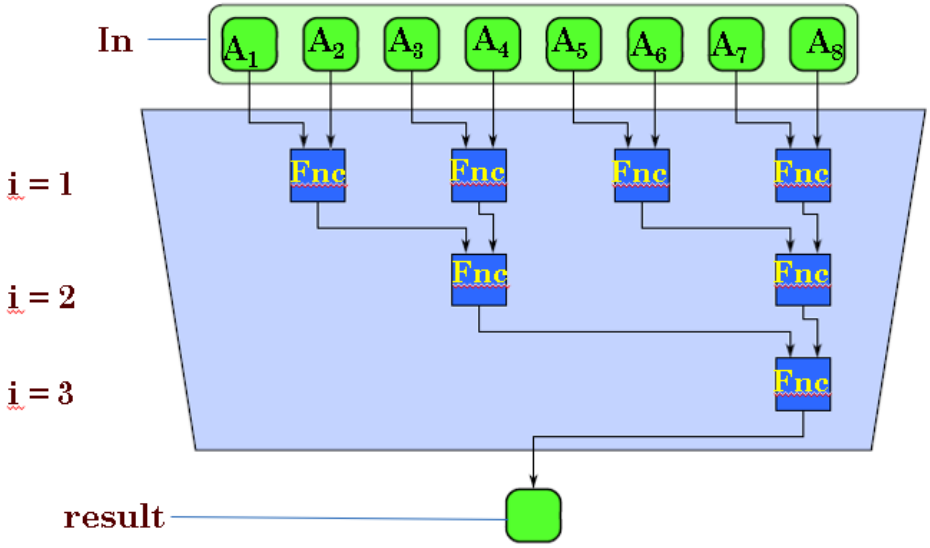
אנחנו מכניסים לרשימת ההחזרה את תוצאת הפונקציה על האיבר הראשון.

תבנית צמצום Reduce

פה יש לנו שוני קטן לעומת פונקציית-העל. אם בפונקציית העל עשינו צמצום בין האיבר הראשון והשני, ואז את התוצאה צמצמנו עם השלישי וכו', כאן אנחנו עושים צמצום בזוגות – אנחנו עוברים על כל רמה, מצמצים זוגות, וברמה הבאה אנחנו ממשיכים לצמצם את הזוגות הקיימים עד שנגיע לזוג הבודד אותו נצמצם. מבחינה ויזואלית, אם הצמצום של פונקציית העל נראה כמו משולש ישר זוית –



הצמצום של ה-List comprehension נראה יותר כמו פירמידה הפוכה –



בגלל השינוי במימוש, אם אנחנו רוצים להשתמש בצמצום אנחנו צריכים לזכור לעבוד רק עם זוגות. כך שאנחנו נצטרך להוסיף תנאי שבודק את אורך הרשימה – אם היא לא-זוגית, אנחנו נדרשים להוסיף משתנה לרשימה שיהיה ניטרלי, שזה כמובן תלוי באופי הצמצום שאנחנו רוצים (0 יתאים לסכימה, אבל פחות יתאים למכפלת מספרים), ורק אז להמשיך ולרוץ.

המימוש ב-List comprehension ייראה כך –

```
@tail_call_optimized
def reduceLCompr(func, In, neutral=0):
   if len(In) == 1:
     return In[0]
   if len(In) % 2 != 0:
     In = In + [neutral]
   funcPairs = [func(In[i],In[i+1]) for i in range(0,len(In)-1,2)]
   return reduceLCompr(func, funcPairs, neutral)
```

כאשר Func הוא הפונקציה שמקבלת שני ערכים, אליה אנחנו מכניסים את שני האיברים המיועדים, ואז שולחים את שאר הרשימה להמשך המימוש (מכאן גם אפשר להבין למה אנחנו דורשים את הזוגיות – אנחנו שולחים בכל פעם שני איברים, ומריצים את השאר בצורה רקורסיבית זנבית).

דווקא כאן, במפתיע, הרקורסיות קצרות יותר לכתיבה –

```
def reduceRec(Fnc, In):
   if len(In) == 1:
     return In[0]
   else:
     return Fnc(In[0], reduceRec(Fnc, In[1:]))
```

כאן אנחנו מכניסים לפונקציה את האיבר האשון, ואת התוצאה של הצמצום על שאר הרשימה. בעצם, בצורה הלא-זנבית אנחנו עושים את הצמצום לא מהאיבר הראשון, אלא מהאחרון, אבל בפועל אין בזה הבדל משמעותי.

```
@tail_call_optimized
def reduceTailRec(Fnc, In, Result = 0):
   if In == []:
     return Result
   else:
     return reduceTailRec(Fnc, In[1:], Fnc(Result, In[0]))
```

הרקורסיה הזנבית שולחת בכל פעם את הפונקציה עצמה, שכמובן לא מקבלת שום שינוי, את זנב הרשימה, ואת התוצאה המופעלת על ראש הרשימה והתוצאה הנוכחית.