```
reduce(lambda x,y: x+1, L, 0) - אורך רשימה
def mylen(L):
    if L==[]:
        return 0
    return 1+myfunc(L[1:])
def mylen1(L):
    def rec(L1, acc):
        if L1==[]:
            return acc
        return rec(L1[1:],[1+acc]. (1+acc) מחזיר את הרשימה המקורית עם אורכה בכל תא.
    return rec(L,0)
def mylen2(L):
    if L==[]:
        return 0
    elif isinstance(L[0],list):
        return mylen2(L[1:])+mylen2(L[0])
    else:
        return 1+mylen2(L[1:])
def myreverse(L):
    if len(L)==0:
        return []
    elif isinstance(L[0],list):
        return myreverse(L[1:])+[myreverse(L[0])]
    else:
        return myreverse(L[1:])+[L[0]]
def reducerec(func, L):
    if len(L)==1:
        return L[0]
    return func(reducerec(func, L[:-1]), L[-1])
```

* אלגוריתם רקורסיבי - פונקציה שקוראת לעצמה עם קלט קטן יותר.
* בסיס - התנאי שבו הפונקציה מפסיקה לקרוא לעצמה (מונע לולאה אינסופית).
* Stateless - לא משתמשים במשתנים גלובליים, כל הנתונים עוברים דרך הפרמטרים.
* mylen - מחזיר את אורך הרשימה.
* mylen1 - כמו mylen אבל עם פרמטר נוסף (acc) שצובר את התוצאה.
* myreverse - הופך את הרשימה ברקורסיה.
* reducerec - מיישם את reduce ברקורסיה.

סכם

לא לשכוח!

* רשימה - List: אוסף ממוין של מידע מגוון. רשימה ניתנת לשינוי. דוגמא: a=[1,2,(3+2i)]

* רשומה - Tuple: רשימה שלא ניתנת לשינוי - immutable list (אם הרשומה מכילה איברים שהם mutable - הערכים האלו ניתנים לשינוי) דוגמא לרשומה - a=('a',2,['a','b','c'])

* קבוצות - set: אוסף לא ממוין של מידע מגוון - ללא כפילויות. קבוצות ניתנות לשינוי.
דוגמא: s={1,'two',5,1}

- אופרטורים לפעולות בין קבוצות: & חיתוך, | איחוד, - הפרש, ^ הפרש סימטרי: xor

* מילונים - Dict: קבוצה של זוגות - מפתח וערך (key, value) מילונים ניתנים לשינוי.
דוגמא: d={1:'hello', 42:'hi', 'two':[1,2]}

* המחולל Range: כאמצעי ליצירת רשימות, רשומות, קבוצות ומילונים, טווח בין x ל-y בקפיצות z. Range(x,y,z)

* המחולל zip: פונקציה zip כאמצעי ליצירת רשימות, רשומות, קבוצות ומילונים zip(v,w)
- יוצרת זוגות של tuple האיבר הראשון הוא מאיטרטור v והאיבר השני מאיטרטור w.

* עיקרון הסגור - סגור הוא פונקציה יחד עם סביבה, יחס עבור המשתנים שאינם מקומיים בפונקציה.
כלומר: זה הרעיון (העיקרון) שהוא קיים ברגע שהפונק' נוצרה. סגור מאפשר לפונקציה לגשת למשתנים שאינם בטווח ההכרזה המיידי שלה.

* פונקציה מסדר גבוה - פונק' שמקבלת פונק' אחרת כפרמטר, נחשבת פונק' מסדר גבוה.
לדוג: מינ, מאקס. ובפונק' תבניתיות: map, filter, reduce

* תבניות מיפוי - מבצעת פונקציה על כל איבר במבנה הנתונים. הפונק' מקבלת פונק' ו-Iterable ומפעילה על כל איבר מאיברי האוסף את הפונק' שקיבלה כפרמטר. פונק' המערכת לא מחזירה אוסף אלא מחולל (כדי לקבל תוצאה צריך להמיר ל list).
דוגמא: הפלט: [10,8,6,4,2] הקלט: list(map(lambda x: 2*x,[1,2,3,4,5]))

* תבניות סינון - filter - מבצעת סינון. פונקציה שמקבלת פונק' בוליאנית ואוסף, ומחזירה את האיברים באוסף שעונים בחיובי (שמקיימים הבוליאנית). גם הפונק' הזו מחזירה מחולל ולא אוסף מסונן. כל ערך בפיתון יש לו משמעות בוליאנית.
דוגמא: הפלט: [0,2,4,6,8] הקלט: list(filter(lambda x: x%2==0, range(10)))

* תבניות צמצום - מבצעת פעולה מצטברת על כל האיברים. פונק' שמקבלת פונק' ואוסף ומחזירה 'מצטבר'. התוצאה של הפעלת הפונק' על איברי הרשימה בזה אחר זה, במצטבר.
(כדי להשתמש בתבנית יש לייבא את functools). התבנית הזו לא פועלת על כל איבר ברשימה בנפרד אלא על כל האיברים יחד (אין צורך בהמרה לרשימה)
דוגמא: הפלט: 45 הקלט: reduce(lambda x,y: x+y, range(10))

סעיף 2

א) func1([H|T],RES):- !, H2 is 2*H+1, K=[H,H2],
func1(T,RES2), append(RES2,[K],RES).
func1([],[]).

ב) func2(L,RES):- func2(L,[],RES).
func2([],RES,RES):- !.
func2([H|T],ACC,RES):- H2 is 2*H+1, func2(T,[[H,H2]|ACC],RES).

ג) func3([H|T],RES):- listf(H), !, func3(H,R1), func3(T,R2), append(R2,[R1],RES).
func3([H|T],RES):- H2 is 2*H+1, K=[H,H2], !, func3(T,RES2),
append(RES2,[K],RES).
func3([],[]).
listf([_|_]):- !.
listf([]).

ד) job([1,5,4,[2,7],13],L2,L3). כלומר L2=[1,5,7,13], L3=[4,2]
job([H|T1],L2,L3):- is_list(H), !, job(H,H2,H3), job(T1,T2,T3),
append(H2,T2,L2), append(H3,T3,L3).

job([H|T1],[H,T2],L3):- 0 is H mod 2, job(T1,T2,L3).
job([H|T1],L2,[H,T3]):- 1 is H mod 2, job(T1,L2,T3).
job([],[],[]).

- Referential Transparency - פונקציה שתמיד מחזירה אותו ערך עבור אותם ארגומנטים.
  לפונקציה אין תופעות לוואי כמו שינוי משתנה גלובלי או קלט/פלט.
  ניתן להחליף קריאה לפונקציה בערך שהיא מחזירה בלי לשנות את התוכנית.
  (למשל [] זה אחד הדברים שיוצרים אובייקט חדש בכל קריאה).
  `f(L[1:],[L[0]]+acc)` (הוספת האיבר הראשון לסוף הרשימה) `f(L[1:],L[0]+acc)` (הוספת האיבר הראשון לתחילת הרשימה)
  `list(filter(lambda y:y!=x,L))+list(filter(lambda y:y==x,L))` (העברת כל המופעים של x לסוף הרשימה).
  `list(filter(lambda y:y!=0,L))+list(filter(lambda y:y==0,L))` (העברת כל האפסים לסוף הרשימה).
- כתיבה פונקציונלית (ללא לולאות, רקורסיה בלבד) - כל ביטוי חוזר יחיד.
  אי אפשר לעדכן משתנים קיימים (כולל פרמטרים של פונקציות).
  פונקציה יכולה להחזיר ביטוי מורכב שמכיל קריאות לעצמה.
  כל שינוי מצב חייב להיות מועבר כפרמטר חדש.
- `assert(x)` / `assertz(x)` - מוסיף את הפרדיקט x למסד הנתונים.
  `retract(x)` - מוחק את הפרדיקט x ממסד הנתונים.
- השוואות לוגיות - אופרטורים (`=<`, `>=`, `=:=`, `=\=`)
  משמשים לבדיקת ערכים מספריים בתוך ביטויים לוגיים או `yield`.
- `yield` - שומר את מצב הפונקציה, מחזיר ערך אך ממשיך לרוץ בקריאה הבאה.
  דוגמה: `f(a,b)=7` כלומר `f(c,d)=7` כש: `[c,d]=[a,b]`.
- `next(a)` - קריאה ראשונה: מקדם את האיטרטור ומחזיר את האיבר הראשון.
  דוגמה: `a=(x**2 for x in [1,2,3])`, `print(next(a))` כלומר: 1
  הפונקציה יכולה להחזיר ערכים מרובים באמצעות list comprehension.
- generator - פונקציה שיוצרת איטרטור (מחזיר ערכים אחד אחרי השני).
  משמש כדי לחסוך בזיכרון (לא יוצר את כל הרשימה בבת אחת).
  `[x for x in range(0,50) if x%7==0]` כלומר `[0,7,14,21,28,35,42,49]`
  דוגמה: `[x for x in range(0,50)]==l` כלומר `[בדוק מחזיר אותו דבר]`
  כלומר הפונקציה מחזירה את כל האיברים ברשימה l, ולכן התנאי מתקיים.
  ניתן להשתמש ב-list comprehension כדי לבנות רשימה חדשה מתוך רשימה קיימת.
- list comprehension - `[ערך if תנאי else ערך for איבר in רשימה if תנאי]`
  `any` - מחזיר true אם לפחות איבר אחד ברשימה מקיים את התנאי.
  `all` - מחזיר true אם כל האיברים ברשימה מקיימים את התנאי.