Σημειωματάριο Τετάρτης 18 Οκτ. 2017

Περισσότερα για λίστες και ανακύκλωση

Είδαμε σήμερα διάφορα προβλήματα και λύσεις για λίστες. Είδαμε επίσης την ανακύκλωση `while`.

Στο επόμενο βλέπουμε πώς μπορούμε να τροποποιήσουμε ένα στοιχείο μιας λίστας αν γνωρίζουμε το δείκτη του (τη θέση του).

In [2]:

```
L = [1, 2, 3, 4]
L[1] = 3.3
print(L)
```

```
[1, 3.3, 3, 4]
```

Παρά τις ομοιότητες ανάμεσα σε λίστες και strings αυτό δεν είναι δυνατό για strings. Δε μπορούμε δηλ. να αλλάξουμε ένα γράμμα ενός string. Ο μόνος τρόπος είναι να φτιάξουμε καινούργιο string.

In [3]:

```
s = "abcdef"
s[1] = 'x'
print(s)
```

```
---------------------------------------------------------------------
--------
TypeError                                 Traceback (most recent ca
ll last)
<ipython-input-3-2b65678c86a5> in <module>()
      1 s = "abcdef"
----> 2 s[1] = 'x'
      3 print(s)

TypeError: 'str' object does not support item assignment
```

Εδώ βλέπουμε πώς μπορούμε με μια απλή ανακύκλωση `for` να αλλάξουμε όλα τα 0 που εμφανίζονται ως στοιχεία μιας λίστας, σε κάτι άλλο, π.χ. τη λέξη `"zero"`.

In [6]:

```
L = [1, 2, 0, 0, 3, 3, 0, -1]
for i in range(len(L)): # Εδώ το i παίρνει ως τιμές όλες τις θέσεις (όχι τα στοι
χεία) της λίστας
    if L[i]==0:
        L[i] = 'zero'
print(L)
```

```
[1, 2, 'zero', 'zero', 3, 3, 'zero', -1]
```

Για να σβήσουμε ένα στοιχείο μιας λίστας του οποίου γνωρίζουμε τη θέση η εντολή είναι `del L[i]`, όπου `L` είναι το όνομα της λίστας και `i` η θέση του στοιχείου.

In [1]:

```
X = ['a', 'b', 'c']
del X[1]
print (X)
```

```
['a', 'c']
```

Δοκιμάζουμε τώρα να σβήσουμε όλα τα μηδενικά της λίστας L με ένα απλό `for` και αποτυγχάνουμε οικτρά.

Ο λόγος είναι ότι το εύρος τιμών της μεταβλητής `i` στο `for` (στην περίπτωσή μας 0 έως 7) αποφασίζεται μια φορά και μάλιστα τη στιγμή που ξεκινά να τρέχει η ανακύκλωση. Όμως η λίστα μας διαρκώς κονταίνει μετά από τα σβησίματά μας με αποτέλεσμα ο δείκτης `i` να φτάνει μέχρι την τιμή 7 (την αρχικά τελευταία θέση της λίστας) αλλά όταν αυτό συμβαίνει η λίστα δεν έχει πλέον μήκος 7, οπότε προκύπτει λάθος.

In [8]:

```
L = [1, 2, 0, 0, 3, 3, 0, -1]
for i in range(len(L)):
    if L[i]==0:
        del L[i]
print(L)
```

```
---------------------------------------------------------------------
--------
IndexError                                Traceback (most recent ca
ll last)
<ipython-input-8-9b3e48f25056> in <module>()
      1 L = [1, 2, 0, 0, 3, 3, 0, -1]
      2 for i in range(len(L)):
----> 3     if L[i]==0:
      4         del L[i]
      5 print(L)

IndexError: list index out of range
```

Εδώ λύνουμε το προηγούμενο πρόβλημα με χρήση της ανακύκλωσης `while`. Η ανακύκλωση `while` γράφεται ως εξής:

```
while συνθήκη:
    εντολή 1
    εντολή 2
    ...
    εντολή N
```

και εκτελεί τις εντολές που υπάγονται σε αυτήν για όσο εξακολουθεί να ισχύει η συνθήκη. Πρώτα λοιπόν ελέγχεται η συνθήκη. Αν αυτή δεν ισχύει το `while` τελειώνει αμέσως και η ροή του προγράμματος συνεχίζει μετά από αυτό. Αν η συνθήκη ισχύει τότε οι εντολές που υπάγονται στο `while` εκτελούνται κι αυτές και μετά ξαναελέγχει το `while` αν ισχύει η συνθήκη του κλπ. Μπορεί ακόμη και να μην τελειώσει ποτέ να εκτελείται (*άπειρο loop*) πράγμα που δε συμβαίνει με το `for`.

Παρακάτω χρησιμοποιούμε μια μεταβλητή `i` για να κρατάμε την τρέχουσα θέση μας στη λίστα, την οποία διανύουμε από την αρχή προς το τέλος της. Η `i` είναι αρχικά 0. Αν το στοιχείο που κοιτάμε είναι 0 τότε το σβήνουμε από τη λίστα με την εντολή `del`. Σε αυτή την περίπτωση όμως το `i` (η θέση μας στη λίστα) δεν πρέπει να αυξηθεί γιατί αυτό που συνέβη με το σβήσιμο είναι ότι το στοιχείο `L[i+1]` ήρθε στη θέση `i`, το στοιχείο `L[i+2]` ήρθε στη θέση `i+1`, κλπ, και το μήκος της λίστας μειώθηκε κατά 1. Άρα και πάλι πρέπει να εξετάσουμε (στην επόμενη εκτέλεση της ανακύκλωσης) τη θέση `i`, γι' αυτό και δεν αυξάνουμε το `i`. Αν όμως δε σβήσαμε το στοιχείο αυτό (δεν ήταν 0) τότε πρέπει να αυξήσουμε το `i` κατά 1 αν θέλουμε να εξετάσουμε μετά το επόμενο στοιχείο.

Αυτό εξακολουθεί να γίνεται όσο η θέση που πάμε να εξετάσουμε στην ανακύκλωση είναι μια δεκτή θέση της λίστας, όσο δηλ. ισχύει `i < len(L)`. Παρατηρείστε ότι από φορά σε φορά που ελέγχουμε αν ισχύει αυτό αλλάζει όχι μόνο το `i` αλλά και το `len(L)`, εν δυνάμει, αν έχει συμβεί κάποιο σβήσιμο.

In [9]:

```
L = [1, 2, 0, 0, 3, 3, 0, -1]
i=0
while i<len(L):
    if L[i]==0:
        del L[i]
    else:
        i = i+1
print(L)
```

```
[1, 2, 3, 3, -1]
```

Στα επόμενα δύο κελιά δείχνουμε πώς μπορούμε να υλοποιήσουμε την ίδια ανακύκλωση (μέτρημα από 1 έως 10) με το `for` και με το `while`. Το `while` είναι γενικά λίγο πιο πολύπλοκο στην υλοποίησή του (γι' αυτό και σχεδόν πάντα κοιτάμε αν μπορούμε να υλοποιήσουμε την ανακύκλωσή μας με `for`) αλλά είναι πιο δυναμικό (η συμπεριφορά του δηλ. αλλάζει στο χρόνο που τρέχει το πρόγραμμα) και μπορεί να κάνει κανείς περισσότερα με αυτό.

In [10]:

```
for i in range(1,11):
    print(i)
```

```
1
2
3
4
5
6
7
8
9
10
```

In [12]:

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

```
1
2
3
4
5
6
7
8
9
10
```

Μια σημαντική παρατήρηση για μεταβλητές τύπου λίστας

Παρατηρείστε στα επόμενα δύο κελιά φαίνεται μια σημαντική διαφορά ανάμεσα σε μεταβλητές αριθμητικού τύπου και μεταβλητές τύπου λίστας.

Η συμπεριφορά στο πρώτο κελί είναι η αναμενόμενη και αυτή που συμβαδίζει με την αντίληψή μας ότι μια μεταβλητή δεν είναι τίποτ' άλλο από ένα κουτί που έχει ένα όνομα για να αναφερόμαστε σε αυτό. Η τιμή της μεταβλητής είναι το περιεχόμενο του κουτιού και εκχώρηση τιμής στη μεταβλητή σημαίνει ότι βάζουμε μια τιμή μέσα στο κουτί αυτό. Έτσι το ότι αλλάξαμε την τιμή της μεταβλητής a δεν επηρεάζει στο παραμικρό τη μεταβλητή b.

Δε συμβαίνει όμως αυτό στο δεύτερο κελί όπου οι μεταβλητές a και b είναι τύπου λίστας. Βλέπουμε εκεί ότι μετά την εκχώρηση b = a μια οποιαδήποτε τροποποίηση (όχι όμως εκχώρηση νέας τιμής, όπως φαίνεται στο μεθεπόμενο κελί) της λίστας a συνεπάγεται την ίδια αλλαγή στη λίστα b. Με άλλα λόγια, μετά την εκχώρηση b = a τα περιεχόμενα των δύο λιστών είναι πάντα τα ίδια, όποια μεταβλητή από τις δύο και να τροποποιήσουμε.

In [13]:

```
a = 1
b = a
a = 2
print(b)
```

```
1
```

In [15]:

```
a = [1, 2, 3]
b = a
a[1] = 7
print(b)
```

```
[1, 7, 3]
```

Τι συνέβη;

Ίσως το καλύτερο που έχουμε να κάνουμε αυτή τη στιγμή είναι να δούμε τη λειτουργία του κώδικα αυτού σε python με τη βοήθεια του Online Python Tutor, που είναι ένα εξαιρετικό εργαλείο που μας βοηθάει να βλέπουμε τις τιμές των διάφορων μεταβλητών σε ένα πρόγραμμα python όπως ο κώδικας εκτελείται.

Μπορείτε να δείτε εδώ (http://www.pythontutor.com/visualize.html#code=a%20%3D%20%5B1,%202,%203%5D%0Ab%20%3D%20a%
frontend.js&py=3&rawInputLstJSON=%5B%5D&textReferences=false) το πώς ακριβώς μεταβάλλονται οι μεταβλητές `a, b` όπως εκτελείται ο κώδικας (πατάτε συνεχώς το κουμπί Forward>).

Μια μεταβλητή στην python είναι απλά ένα όνομα. Το όνομα αυτό μπορεί να δείχνει σε ένα αντικείμενο (αριθμό, λίστα, string, κ.ά.). Ο λόγος ύπαρξης των μεταβλητών είναι ακριβώς για να μπορούμε να αναφερόμαστε στα αντικείμενα αυτά, τα οποία είναι αποθηκευμένα κάπου στη μνήμη του υπολογιστή.

Όταν π.χ. στο πρόγραμμα python που υπάρχει στο προπροηγούμενο κελί εμφανίζεται για πρώτη φορά η μεταβλητή `a` και εμφανίζεται σε μια εντολή του τύπου `a=1`, αυτό που συμβαίνει στο παρασκήνιο είναι ότι (α) δημιουργείται η νέα μεταβλητή `a` και (β) η μεταβλητή αυτή "δείχνει" σε μια θέση μνήμης όπου είναι αποθηκευμένη η τιμή 1. Αν έπεται η εντολή `b=a` τότε (α) δημιουργείται η νέα μεταβλητή `b` (αν δεν έχει χρησιμοποιηθεί πριν και δεν υπάρχει ήδη) και (β) η `b` δείχνει σε μια άλλη θέση μνήμης όπου είναι αποθηκευμένη η τιμή του δεξιού μέλους, δηλ. και πάλι στον αριθμό 1. (Το 1 έτσι υπάρχει σε δύο θέσεις μνήμης.) Όταν μετά εκτελεσθεί η εντολή `x=2` αυτό που συμβαίνει είναι ότι η μεταβλητή `a` δείχνει πλέον σε μια θέση μνήμης όπου είναι αποθηκευμένη η τιμή 2. Αυτή η τελευταία εντολή δεν επηρεάζει την τιμή της `b` η οποία εξακολουθεί να δείχνει στο 1.

Στο πρόγραμμα python που βρίσκεται στο επόμενο κελί η μεταβλητή `a` δημιουργείται και δείχνει σε μια θέση μνήμης όπου είναι αποθηκευμένη η λίστα `[1, 2, 3]`. Όταν εκτελεστεί η εντολή `b=a` αυτό που συμβαίνει διαφορετικό με πριν είναι ότι η τιμή της `b` τίθεται να είναι η τιμή της `a`, όμως η τιμή της `a` είναι απλά η θέση μνήμης όπου είναι αποθηκευμένη η λίστα (η διεύθυνση της λίστας όπως λέμε). Αυτό σημαίνει ότι τώρα οι `a, b` δείχνουν και οι δύο στην ίδια θέση μνήμης. Ό,τι πράξη λοιπόν κάνουμε από δω και πέρα που επηρεάζει τη λίστα στην οποία δείχνει η `a` (όπως η `a[1] = 7`) θα έχει άμεσο αντίκτυπο και στη λίστα την οποία δείχνει η `b`, απλούστατα γιατί οι `a, b` δείχνουν στην ίδια θέση μνήμης. Γι' αυτό και βλέπουμε αυτή τη διαφορετική συμπεριφορά ανάμεσα σε λίστες και αριθμούς.

Στο επόμενο κελί αντίθετα, όπου η λίστα `a` δεν τροποποιείται αλλά αλλάζει τελείως και δείχνει σε μια άλλη λίστα δεν παρατηρείται το φαινόμενο αυτό, και ο λόγος είναι ότι μετά τη δεύτερη εκχώρηση στη μεταβλητή `a` η μεταβλητή αυτή δείχνει πλέον σε άλλη θέση στη μνήμη.

Και πάλι μπορείτε να το δείτε λεπτομερώς εδώ αυτό πατώντας συνεχως το κουμπί Forward>. (http://www.pythontutor.com/visualize.html#code=a%20%3D%20%5B1,%202,%203%5D%0Ab%20%3D%20a%
frontend.js&py=3&rawInputLstJSON=%5B%5D&textReferences=false)

In [16]:

```
a = [1, 2, 3]
b = a
a = [4, 5, 6, 7]
print(b)
```

```
[1, 2, 3]
```

Τι γίνεται αν θέλουμε να αντιγράψουμε τη λίστα a στη λίστα b χωρίς να κινδυνεύει η b να αλλάξει με την όποια αλλαγή συμβεί στην a; Ο πιο απλός τρόπος είναι να χρησιμοποιήσουμε ένα slice για να αντιγράψουμε τη λίστα a στη λίστα b όπως φαίνεται στο επόμενο κελί.

In [18]:

```
a = [1, 2, 3]
b = a[:]
a[1] = 7
print(b, a)
```

```
[1, 2, 3] [1, 7, 3]
```

Διάφορα προγράμματα με λίστες

Στο επόμενο πρόγραμμα από μια λίστα αριθμών X φτιάχνουμε μια άλλη λίστα Y που περιέχει εκείνα τα στοιχεία της X που ικανοποιούν κάποια συνθήκη, συγκεκριμένα να ανήκουν στο διάστημα [5, 8]. Ξεκινάμε με τη λίστα Y να είναι κενή, διανύουμε την X με το `for` loop και για κάθε στοιχείο x της λίστας X το βάζουμε στην Y αν και μόνο αν ικανοποιεί τη συνθήκη μας.

In [19]:

```
X = [1, 2, 3, 2, 1, -1,10, 7, 6, 3.4]
Y = []
for x in X:
    if 5<=x<=8:
        Y.append(x) # Αυτή η εντολή θα μπορούσε να έχει και την εξής μορφή: Y =
 Y+[x]
print(Y)
```

```
[7, 6]
```

Στο επόμενο διανύουμε μια λίστα αριθμών και κρατάμε όλα τα στοιχεία της από μια φορά το καθένα μόνο.

In [20]:

```
X = [1, 2, 3, 2, 1, -1,10, 7, 6, 3.4]
Y = []
for x in X:
    if not(x in Y): # αν το στοιχείο x είναι ήδη στη λίστα Y τότε δεν το προσθέτ
ουμε
        Y.append(x)
print(Y)
```

```
[1, 2, 3, -1, 10, 7, 6, 3.4]
```

Στο επόμενο κελί λύνουμε το εξής πρόβλημα. Έχουμε μια λίστα αριθμών X και θέλουμε για κάθε μπλοκ ίδιων τιμών να κρατήσουμε μόνο ένα αντιπρόσωπο. Αν δηλ. υπάρχει κάπου το μπλοκ τιμών `..,2,2,2,2,...` στην X τότε στην Y βάζουμε μόνο ένα 2.

Ο τρόπος που το πετυχαίνουμε αυτό είναι ότι προσθέτουμε το τρέχον στοιχείο της λίστας X στη λίστα Y μόνο αν δεν είναι το ίδιο με το τελευταίο στοιχείο της λίστας Y, δηλ. το `Y[len(Y)-1]` (το οποίο θα μπορούσαμε και να έχουμε γράψει ως `Y[-1]`).

In [24]:

```
X = [1, 2, 2, 2, 3, 3, 2, 1, -1,10, 7, 6, 3.4]
Y = [X[0]]
for x in X[1:]:
    if Y[len(Y)-1] != x:
        Y.append(x)
print(Y)
```

```
[1, 2, 3, 2, 1, -1, 10, 7, 6, 3.4]
```

Στο επόμενο προσθέτουμε όλα της στοιχεία της λίστας αριθμών L. "Μαζεύουμε" το άθροισμα σε μια μεταβλητή s, αρχικά 0, διανύοντας τη λίστα με τη μεταβλητή x και κάθε φορά αυξάνουμε το s κατά x (αυτό κάνει ο τελεστής += που βλέπουμε στην προτελευταία γραμμή).

Υπάρχει και έτοιμη συνάρτηση γι' αυτό, η `sum(L)`, η οποία επιστρέφει το άθροισμα των στοιχείων της L.

Στο επόμενο κελί από αυτό βλέπουμε το ίδιο πράγμα αλλά με μεταβλητή που διατρέχει τις θέσεις της λίστας (0 έως το μήκος της πλην ένα) και όχι τα περιεχόμενα της λίστας.

Στο επόμενο κελί από αυτό βλέπουμε πώς να υπολογίσουμε το άθροισμα των τετραγώνων των στοιχείων της λίστας.

In [25]:

```
L = [1.1, 2, -3.5, 6, 8]
s = 0
for x in L:
    s += x
print(s)
```

```
13.6
```

In [26]:

```
L = [1.1, 2, -3.5, 6, 8]
s = 0
for i in range(len(L)):
    s += L[i]
print(s)
```

```
13.6
```

In [27]:

```
L = [1.1, 2, -3.5, 6, 8]
s = 0
for x in L:
    s += x*x
print(s)
```

```
117.46000000000001
```

Εδώ βλέπουμε πώς μπορούμε από μια λίστα X της οποίας τα στοιχεία είναι λίστες αριθμών να πάμε σε μια λίστα F που περιέχει (με τη σειρά που εμφανίζονται στην X) όλους τους αριθμούς της X.

Για να το κάνουμε αυτό αρχικοποιούμε την F σε κενή λίστα και για κάθε στοιχείο L της X διανύουμε όλα τα στοιχεία της L με τη μεταβλητή x και προσθέτουμε το κάθε x στη λίστα F.

Στο επόμενο κελί από αυτό βλέπουμε πώς αυτό μπορεί να γίνει εκμεταλλευόμενοι την πράξη + στις λίστες και απαλείφοντας έτσι το εσωτερικό `for` loop.

In [28]:

```
X = [[1, 2, 3], [4.1], [-1, -2, -11], [0, 0]]
F=[]
for L in X:
    for x in L:
        F.append(x)
print(F)
```

```
[1, 2, 3, 4.1, -1, -2, -11, 0, 0]
```

In [2]:

```
X = [[1, 2, 3], [4.1], [-1, -2, -11], [0, 0]]
F = []
for x in X:
    F += x
print(F)
```

```
[1, 2, 3, 4.1, -1, -2, -11, 0, 0]
```

Εδώ λύνουμε το ίδιο πρόβλημα αλλά τώρα επιτρέπουμε στα στοιχεία της X να είναι είτε αριθμοί είτε λίστες αριθμών.

Για να διακρίνουμε αν ένα αντικείμενο είναι λίστα έχουμε στη διάθεσή μας τη συνάρτηση `type`. Για να ελέγξουμε αν ένα αντικείμενο L είναι λίστα ή όχι υπολογίζουμε τη λογική ποσότητα `type(L) is list`, που είναι True αν το L είναι λίστα και False αλλιώς.

Έτσι αν το στοιχείο L της F είναι λίστα κάνουμε ό,τι κάναμε προηγουμένως (διανύουμε δηλ. τη λίστα αυτή και προσθέτουμε τα στοιχεία της στην F) ενώ αν δεν είναι τότε το στοιχείο αυτό είναι αριθμός οπότε το προσθέτουμε στη λίστα F.

In [33]:

```
X = [4, [1, 2, 3], [4.1], -2, [-1, -2, -11], [0, 0]]
F=[]
for L in X:
    if type(L) is list:
        for x in L:
            F.append(x)
    else:
        F.append(L)
print(F)
```

```
[4, 1, 2, 3, 4.1, -2, -1, -2, -11, 0, 0]
```

In [32]:

```
L = [1, 2, 3]
x = 3.4
print( type(L) is list )
print( type(x) is list )
```

```
True
False
```

Βλέπουμε εδώ πώς μπορούμε να χρησιμοποιήσουμε ένα `for` loop για να σχεδιάσουμε ένα ορθογώνιο από αστεράκια στην οθόνη με `m` γραμμές και `n` στήλες.

In [35]:

```
m = 5
n = 10
for i in range(m):
    print(n*'*')
```

```
**********
**********
**********
**********
**********
```

Σχεδιάζουμε τώρα ένα κάπως πιο περίπλοκο σχήμα, ένα ορθογώνιο με `m` γραμμές και `n` στήλες, που όμως από μέσα του λείπει ένα άλλο ορθογώνιο. Οι παράμετροι που περιγράφουν το σχήμα, πέρα από τα `m, n` είναι τα μήκη `k1, k2` (πόσες γραμμές είναι γεμάτες στην κορυφή και πόσες στον πάτο του ορθογωνίου) και τα μήκη `l1, l2` (πόσες στήλες είναι γεμάτες στην αριστερή πλευρά και πόσες στη δεξιά πλευρά του ορθογωνίου).