Κεφάλαιο 7

7. Ενσωματωμένες σύνθετες δομές δεδομένων

Τελειώνοντας αυτό το μάθημα θα έχεις μάθει

- 🎯 **Τί είναι μια σύνθετη δομή δεδομένων στην Python και ποιες είναι οι ενσωματωμένες σύνθετες δομές σε αυτή**
- 🎯 **Τί είναι συλλογή και τί ακολουθία**
- 🎯 **Τί είναι οι πλειάδες και πώς τις χρησιμοποιούμε**
- 🎯 **Τί είναι οι λίστες και γιατί είναι από τις πιο σημαντικές δομές στην Python**
- 🎯 **Τι είναι λεξικό και πώς το χρησιμοποιούμε**
- 🎯 **Τί είναι σύνολο και πώς το χρησιμοποιούμε**
- 🎯 **Τους τύπους συνόλων στην Python και το λόγο ύπαρξης τους**
- 🎯 **Την αλγοριθμική δομή της εξαίρεσης και τη χρήση της**
- 🎯 **Τις συναρτήσεις που εφαρμόζονται στις συλλογές και τις ακολουθίες**
- 🎯 **Τις μεθόδους κάθε ενσωματωμένης δομής δεδομένων**

Περιεχόμενα

7.1 Σύνθετες δομές δεδομένων

Στην προηγούμενη ενότητα είδαμε μία σύνθετη δομή δεδομένων τη συμβολοσειρά. Στην Python έχουμε αρκετές ακόμα ενσωματωμένες (δηλαδή ορισμένες από την ίδια τη γλώσσα) σύνθετες δομές δεδομένων. Στη συμβολοσειρά τα στοιχεία που τη συνθέτουν είναι άλλες συμβολοσειρές που στην τελική ανάλυση, στις αριθμημένες «θέσεις», έχουν μέγεθος ένα – είναι με άλλα λόγια χαρακτήρες. Επιπλέον η συμβολοσειρά είναι μη μετατρέψιμη (immutable) στην Python, με άλλα λόγια δεν είναι φτιαγμένη για να τροποποιείται, ενώ η θέση μέσα στη συμβολοσειρά έχει σημασία, με άλλα λόγια η δομή είναι αυτό που ονομάζουμε ακολουθία. Αλλάζοντας οποιαδήποτε από αυτές τις ιδιότητες έχουμε μια διαφορετική, ιδιαίτερα χρήσιμη δομή. Παίρνουμε με αυτό τον τρόπο την πλειάδα, τη λίστα και το σύνολο.

Μια ακόμα ενδιαφέρουσα δομή που όμως αντιστοιχίζει ζευγάρια συμβόλων ή κλειδιών και τιμών είναι το λεξικό. Δε χρειάζεται κανείς να επιχειρηματολογήσει πόσο διαδεδομένα και χρήσιμα είναι τα λεξικά πέρα από τις εφαρμογές. Εδώ θα δούμε πώς αξιοποιούνται και εντός του πλαισίου μιας εφαρμογής.

Στη γλώσσα Python, όλα είναι αντικείμενα. Αυτό περιλαμβάνει και τις συμβολοσειρές. Περιλαμβάνει ακόμα τις πλειάδες, τις λίστες, τα λεξικά και τα σύνολα που θα δούμε σε αυτό το κεφάλαιο. Για αυτό και όλα τους έχουν βασικές λειτουργίες ή αλλιώς μεθόδους με τις οποίες, για παράδειγμα, τα δημιουργούμε, τα προσπελαύνουμε και, εφόσον το επιτρέπουν, τα τροποποιούμε. Για κάθε δομή θα δούμε σύντομα και τις αντίστοιχες μεθόδους.

7.2 Η δομή της πλειάδας

Μια πλειάδα (tuple) διαφέρει από μια συμβολοσειρά μόνο στο ότι τα στοιχεία της δεν είναι αποκλειστικά χαρακτήρες. Μια ακολουθία στοιχείων, είτε αυτά είναι αριθμοί, είτε συμβολοσειρές, είτε και οποιουδήποτε άλλου τύπου, μπορούν να ομαδοποιηθούν σε μία δομή αυτήν της πλειάδας.

Αν θέλουμε να είμαστε πιο τυπικοί η πλειάδα είναι μια δομή δεδομένων με τα εξής χαρακτηριστικά:

- ✓ είναι σύνθετη, δηλαδή χρησιμοποιείται για την αποθήκευση πολλαπλών δεδομένων ταυτόχρονα
- ✓ είναι διατεταγμένη, δηλαδή η σειρά των μελών είναι σημαντική (ακολουθία)
- ✓ είναι μη μετατρέψιμη, δηλαδή τα στοιχεία της δεν μπορούν να αλλάξουν μόλις εκχωρηθούν σε αυτά τιμές (δεδομένα)
- ✓ τα δεδομένα της μπορούν να είναι ετερογενή, δηλαδή πολλών διαφορετικών τύπων χωρίς περιορισμό

Αυτό το τελευταίο χαρακτηριστικό είναι αυτό που καθιστά την πλειάδα ένα πολύ σημαντικό εργαλείο για την Python.

Λόγω της φύσης της η πλειάδα χρησιμοποιείται όταν οι τιμές παραμένουν σταθερές και δεν μπορούν να αλλάξουν, για την αναπαράσταση σταθερών συνόλων τιμών ή για την ομαδοποίηση στοιχείων που δεν πρέπει να αλλάξουν. Είναι απλούστερη ως προς τη χρήση και πιο αποτελεσματική ως προς τις επιδόσεις της μνήμης από τη λίστα. Προτιμάται ιδιαίτερα όταν θέλουμε να εισάγουμε σταθερές ή μεταβλητές μιας χρήσης.

Ας δούμε μερικά παραδείγματα διαφορετικών τρόπων δημιουργίας πλειάδων (tuple):

```
# Τρόποι δημιουργίας πλειάδων (tuples)
tuple1 = (1, 2, 3, 4, 5, 6)    # Κλασική δημιουργία πλειάδας (tuple)
# που χρησιμοποιεί παρενθέσεις για τη δημιουργία
# της.
tuple2 = tuple([10, 20, 30])    # Εναλλακτικός τρόπος δημιουργίας
# πλειάδας χρησιμοποιώντας τη συνάρτηση tuple().
# Σημειώστε τις αγκύλες μέσα στην παρένθεση, θα τις εξηγήσουμε
# σε επόμενη ενότητα.
tuple3 = 1, 2, 'τρία'    # Όταν η Python βλέπει να ανατίθεται
# μια ακολουθία σε μια μεταβλητή χωρίς παρενθέσεις
# αγκύλες ή άγκιστρα, υποθέτει ότι πρόκειται για
# πλειάδα. Για αυτό και μπορούμε και με αυτό τον τρόπο
# να ορίσουμε μια πλειάδα. Έτσι αν ρωτήσουμε για την
# tuple3, γράφοντας:
 print(type(tuple3)
# θα μας επιστρέψει:
<class 'tuple'>
```

Σημείωση: Όταν δημιουργούμε μια μοναδιαία πλειάδα δηλ. μια πλειάδα με ένα μόνο μέλος, ή αλλιώς στοιχείο, ιδιαίτερα όταν πρόκειται για αριθμό ή μεταβλητή, πρέπει μετά το στοιχείο να τοποθετήσουμε και κόμμα. Αυτό γιατί αλλιώς θα εκληφθεί ως αριθμητική παράσταση.

```
  x = 3
  tuple4 = (3)    # Μια απλή μεταβλητή, όχι tuple. Πράγματι αν
# ζητήσουμε τον τύπο της:
  print(type(tuple4))
<class 'integer'>    # Θα είναι ακέραιος ! Με τιμή 3.
  tuple4 = 3,    # Προσέξτε το κόμμα, το tuple4 είναι πλέον tuple
  print(type(tuple4))
<class 'tuple'>
  tuple1[3] = 14    # Προσπαθούμε να αλλάξουμε την τιμή 4 που
# υπάρχει στη θέση 3 (4η) της λίστας tuple1 σε 14. Δεν
# επιτρέπεται. Θα προκληθεί σφάλμα TypeError !
```

Είδαμε μόλις ότι δεν μπορούμε να αλλάξουμε μια πλειάδα. Το ίδιο ισχύει και για τα στοιχεία της. Έτσι, αν προσπαθούσαμε να αλλάξουμε ένα από τα στοιχεία της tuple1 στο πρώτο παράδειγμα θα ήταν και πάλι ανέφικτο.

Πράγματι η εντολή που ακολουθεί και που προσπαθεί να αλλάξει το 4ο στοιχείο της πλειάδας tuple1, η οποία θυμίζουμε είναι η (1, 2, 3, 4, 5, 6) από 4 σε 5, θα δώσει πάλι σφάλμα TypeError!

```
  tuple1[3] = 5    # Θα προκληθεί και πάλι σφάλμα TypeError!
```

Προσοχή! Έχουμε πει ότι μέλος/ στοιχείο μιας πλειάδας μπορεί να είναι ο,τιδήποτε: ένας ακέραιος, ένα string αλλά ακόμα και μια μεταβλητή ή ένα αντικείμενο. Ας πάρουμε την περίπτωση της μεταβλητής. Η μεταβλητή που αποτελεί το 3ο μέλος της πλειάδας μας δεν μπορεί να αλλάξει. Η τιμή της όμως; Ας δούμε:

```
  x = 3
  y = 5
  tuple5 = (x, y)
  print(x + ' και ' + y)
3 και 5
  x = 7    # Τροποποιούμε την τιμή της μιας μεταβλητής
  print(x + ' και ' + y)
7 και 5
```

Επομένως, εφόσον ως στοιχεία μιας πλειάδας έχουν οριστεί μεταβλητές, οι μεταβλητές αυτές δεν μπορούν να αλλαχθούν, καθώς η πλειάδα είναι μη μετατρέψιμη (immutable). Οι τιμές όμως των μεταβλητών αυτών μπορούν φυσικά να αλλάζουν αφού είναι μεταβλητές (mutable).

7.2.1 Στοιχεία μιας πλειάδας (πρόσβαση, διάσχιση)

Η πρόσβαση στα στοιχεία μιας πλειάδας γίνεται με τον ίδιο ακριβώς τρόπο που γίνεται η πρόσβαση και στα στοιχεία μιας συμβολοσειράς. Η αρίθμηση των θέσεων των στοιχείων ξεκινά από 0 και τελειώνει σε (μέγεθος -1). Λέμε ότι ο δείκτης, ή το ευρετήριο, της πλειάδας ξεκινάει από 0 και φτάνει μέχρι μέγεθος -1. Παρότι τη λίστα την ορίζουμε με παρενθέσεις, η πρόσβαση στα στοιχεία της μέσω δεικτοδότησης, ή αλλιώς ευρετηρίου, γίνεται με αγκύλες. Μερικά παραδείγματα:

```
new_tuple = (1, 2, 3, 4, 5, 6)
print(new_tuple[1])    # Εκτύπωση του 2ου στοιχείου της πλειάδας
2
print(new_tuple[5])    # Εκτύπωση του 6ου στοιχείου της πλειάδας
6
```

Η πρόσβαση μπορεί να γίνει τόσο με θετική όσο και με αρνητική δεικτοδότηση. Η θέση -1 είναι η τελευταία θέση της πλειάδας, η -2 η προτελευταία και ούτω καθ΄ εξής. Μερικά ακόμα παραδείγματα:

```
print(new_tuple[-2])    # Εκτύπωση του προτελευταίου
# στοιχείου της πλειάδας
5
print(new_tuple[-1])    # Εκτύπωση του τελευταίου στοιχείου
# της πλειάδας
6    # Προσέξτε ότι δίνει το ίδιο αποτέλεσμα με την print(new_tuple[5])
```

Μπορούμε επίσης να προσπελάσουμε ένα υποσύνολο μιας πλειάδας αν υποδείξουμε ένα διάστημα για τον δείκτη. Το κομμάτι (slice) που θα προσπελάσουμε ορίζεται από δύο ορίσματα, την αρχική και την τελική θέση του στην πλειάδα. Όταν εκτελείται αυτή η διεργασία, την οποία ονομάζουμε κατάτμηση (στα αγγλικά slicing), επιστρέφει μια νέα πλειάδα που περιλαμβάνει τα επιλεγμένα στοιχεία.

```
tuple_slice = new_tuple[0:3]'
print(tuple_slice)
(1, 2, 3, 4)
```

Για να επισκεφτούμε τα στοιχεία μιας πλειάδας μπορούμε να χρησιμοποιήσουμε ένα βρόγχο σταθερού μεγέθους (δηλαδή ένα βρόγχο for). Κάθε στοιχείο της πλειάδας θα ανατεθεί διαδοχικά στη μεταβλητή που ορίστηκε για αυτό το σκοπό, επιτρέποντας τη χρησιμοποίησή του μέσα στο βρόγχο:

```
for x in new_tuple:
      print(x)
```

7.2.2 Βασικές λειτουργίες πλειάδας, πράξεις και συναρτήσεις που εφαρμόζονται σε αυτή

Οι ακολουθίες στην Python περιλαμβάνουν τις συμβολοσειρές, τις πλειάδες, τις λίστες, και το προϊόν της εφαρμογής της range σε ένα εύρος τιμών. Υπάρχουν μια σειρά από πράξεις και συναρτήσεις που εφαρμόζονται σε όλες τις ακολουθίες (iterables) και επομένως και στις πλειάδες. Επίσης υπάρχουν ενσωματωμένες συναρτήσεις που μετατρέπουν μία ακολουθία σε άλλη διαφορετικού τύπου. Είδαμε ήδη τη δεικτοδότηση και την κατάτμηση που ανήκουν σε αυτή την κατηγορία. Ας δούμε εδώ αναλυτικά μερικές ακόμα δυνατότητες.

Πράξεις για ακολουθίες

Δύο πράξεις εφαρμόζονται σε όλες τις ακολουθίες η συνένωση (concatenation) και ο πολλαπλασιασμός. Η πρώτη εφαρμόζεται ανάμεσα σε δύο ομοειδείς ακολουθίες τις οποίες και συνενώνει σε μία νέα. Η δεύτερη εφαρμόζεται ανάμεσα σε ένα ακέραιο και μία ακολουθία και ουσιαστικά επαναλαμβάνει σε μία νέα ακολουθία, πάντα ίδιου τύπου, τα στοιχεία της ακολουθίας τόσες φορές όσες υποδεικνύει ο ακέραιος. Στις δύο αυτές πράξεις αντιστοιχούν οι τελεστές + και * αντίστοιχα. Τέλος μπορούμε να έχουμε μια ακολουθία από ακολουθίες. Ας δούμε μερικά παραδείγματα:

```
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)
print(tuple1 + tuple2)    # Εκτύπωση της συνένωσης των δύο πλειάδων
(1, 2, 3, 4, 5, 6)
print(2 * tuple1)    # Εκτύπωση του πολλαπλασιασμού της
# πλειάδας επί δύο (θα έδινε το ίδιο αποτέλεσμα και το
# tuple1 * 2)
(1, 2, 3, 1, 2, 3)
tuple3 = (tuple1, tuple2)
print(tuple3)
((1, 2, 3), (4, 5, 6))    # Εκτύπωση μιας πλειάδας πλειάδων
```

Φυσικά μπορεί κάποια στοιχεία να είναι πλειάδες και κάποια άλλα όχι. Αν αναρωτιέστε που μπορεί να είναι χρήσιμο αυτό, ας δούμε άλλο ένα παράδειγμα. Ένα σημείο στο επίπεδο ορίζεται από τις δύο συντεταγμένες του στον άξονα x και τον άξονα y. Μία απόσταση ανάμεσα σε ένα σημείο και μια γραμμή ορίζεται από την απόσταση του σημείου από το πλησιέστερο σημείο της γραμμής. Αν η γραμμή είναι κύκλος και το σημείο το κέντρο του, η απόσταση αυτή είναι ίδια με όποιο σημείο του κύκλου και αν μετρήσουμε και είναι η ακτίνα του κύκλου. Ας ορίσουμε τώρα το κέντρο, την ακτίνα και τον ίδιο τον κύκλο:

```
center = (10, 20)    # Η πλειάδα που ορίζει το κέντρο του
# κύκλου
r = 5
circle = (center, r)    # Η πλειάδα που ορίζει τον κύκλο
print(circle)
```

```
((10, 20), 5 )    # Η πλειάδα που ορίζει τον κύκλο αναλυτικά
```

Έλεγχος συμπερίληψης

Μπορούμε επίσης να ελέγξουμε αν ένα στοιχείο ανήκει σε μία πλειάδα ή και το αντίθετο. Αυτό γίνεται με την εφαρμογή των τελεστών `in` και `not in` για έλεγχο συμπερίληψης ή μη αντίστοιχα. Και οι δύο τελεστές εφαρμόζονται ανάμεσα σε ένα πιθανό στοιχείο και μία πλειάδα, δημιουργώντας μία παράσταση που μπορεί να είναι είτε αληθής είτε ψευδής. Όπως όλες οι παραστάσεις μπορεί να χρησιμοποιηθεί ως συνθήκη, για παράδειγμα σε μία εντολή επιλογής ή επανάληψης. Θυμηθείτε το έχουμε ήδη χρησιμοποιήσει στην προηγούμενη παράγραφο σε ένα `for`. Ας δούμε άλλο ένα παράδειγμα:

```
for x in tuple1
     print(x)  # Θα τυπώσει διαδοχικά τους αριθμούς 1,2 και 3
```

Συγκρίσεις

Και οι συγκριτικοί τελεστές μπορούν να εφαρμοστούν στις ακολουθίες. Αρκεί οι δύο ακολουθίες να είναι συγκρίσιμες. Εφόσον αυτό συμβαίνει, συγκρίνονται τα στοιχεία στην ίδια θέση (ξεκινώντας από τη θέση μηδέν) μέχρι να βρεθεί διαφορετικό και τότε η ακολουθία που έχει το στοιχείο με τη μεγαλύτερη τιμή θα αντιμετωπιστεί ως μεγαλύτερη. Περισσότερα για αυτό θα πούμε σε άλλο σημείο.

Συναρτήσεις

Υπάρχουν όπως έχουμε ήδη αναφέρει μια σειρά από συναρτήσεις που εφαρμόζονται σε όλες τις ακολουθίες, άρα και στις πλειάδες. Αυτές περιλαμβάνουν την εύρεση του μήκους της ακολουθίας δηλαδή τη συνάρτηση `len( )` και, εφόσον οι τιμές είναι συγκρίσιμες, το μέγιστο και ελάχιστο στοιχείο με τις συναρτήσεις `max( )`και `min( )` αντίστοιχα. Ακόμα η, επίσης ενσωματωμένη, συνάρτηση `sorted( )` επιστρέφει μια ταξινομημένη λίστα με τα ίδια στοιχεία με την πλειάδα μας. Ας δούμε κι εδώ κάποια παραδείγματα:

```
tuple1 = (2, 1, 3, 5)
print(len(tuple1 ))    # Θα τυπώσει το μέγεθος της πλειάδας
4
print(max(tuple1 ))    # Θα τυπώσει τη μεγαλύτερη τιμή

5
print(min(tuple1 ))    # Θα τυπώσει τη μικρότερη τιμή

1
print(sorted(tuple1 ))    # Θα τυπώσει μια λίστα με τα ίδια
# στοιχεία ταξινομημένα.
[1, 2, 3, 5]
```

Μέθοδοι

Όλες οι ακολουθίες, όπως όλοι οι τύποι δεδομένων στην Python είναι αντικείμενα. Επομένως έχουν μεθόδους. Τις λιγότερες μεθόδους από οποιονδήποτε τύπο ακολουθίες τις έχει η πλειάδα και μάλιστα είναι κοινές με όλους τους άλλους τύπους. Για να δούμε τις μεθόδους ενός αντικειμένου χρησιμοποιούμε την ενσωματωμένη συνάρτηση `dir( )`. Η συνάρτηση επιστρέφει μια λίστα με τα ονόματα των μεθόδων του αντικειμένου μας. Αν την εφαρμόσουμε σε μία πλειάδα θα μας επιστρέψει μόλις δύο μεθόδους τις `count( )` και `index( )`. Και οι δύο μέθοδοι εφαρμόζονται σε ένα πιθανό στοιχείο της πλειάδας και επιστρέφουν η μεν πρώτη τον αριθμό των εμφανίσεών του στην πλειάδα, η δε δεύτερη τη θέση της πρώτης εμφάνισης αποδιδόμενη με τον αντίστοιχο δείκτη.

```
tuple1 = (1, 2, 3, 3)
print(dir(tuple1 ))    # Θα τυπώσει τις μεθόδους της πλειάδας
['count', 'index']
print(tuple1.count(3))    # Θα τυπώσει τη μεγαλύτερη τιμή
2
print(tuple1.index(3))    # Θα τυπώσει τη θέση πρώτης εμφάνισης

2
```

Άλλες χρήσεις των πλειάδων

Οι πλειάδες χρησιμοποιούνται για πολλαπλή εκχώρηση τιμών – με ή χωρίς τις παρενθέσεις. Φυσικά, το πλήθος των μεταβλητών και τιμών στις δύο πλευρές της εκχώρησης πρέπει να είναι το ίδιο:

```
x, y, z = (1, 2, 3)    # Θα ήταν το ίδιο με παρενθέσεις μπροστά # ή χωρίς
πίσω.
print(x)    # Θα τυπώσει την τιμή του x
1
print(y)    # Θα τυπώσει την τιμή του y
2
print(z)    # Θα τυπώσει την τιμή του z

3
```

Χρησιμοποιούνται ακόμα για να επιστρέψει μια συνάρτηση πολλαπλές τιμές χρησιμοποιώντας τη `return`. Δε χρειάζεται καμία τροποποίηση, επιστρέφει απλά μια πλειάδα.

```
def limits(a)
      x = min(a)
      y = max(a)
      return x, y
tuple1 = (1, 2, 3, 10, 7)
print(limits(tuple1))    # Θα τυπώσει τις min και max τιμές της πλειάδας
(1, 10)
```

Χρησιμοποιούνται επίσης στην ανταλλαγή τιμών, ένα ιδιαίτερο χαρακτηριστικό της Python.

```
X = 'Νίκος'
Y = 'Μαρία'
Y, X = X, Y    # Εκχώρηση τιμών μέσω ανταλλαγής
print(X)    # Θα τυπώσει την τιμή του X
'Μαρία'    # Αντίστοιχα η τιμή του Y θα είναι 'Νίκος'
```

Οι τρεις αυτές χρήσεις δεν είναι πολύ διαφορετικές όπως μπορεί κάποιος να δεί από την ακόλουθη υλοποίηση.

```
def swap(x,y):
      return y, x     # Συνάρτηση ανταλλαγής

x = 1
y = 2
x,y = swap(x,y)    # Θα ανταλλάξει τις δύο τιμές
print(x, ' και ', y)    # Θα τυπώσει τις νέες τιμές
 2 και 1
```

Μετατροπή ακολουθίας σε διαφορετικού τύπου.

Κάθε είδους ακολουθία στην Python μπορεί να μετατραπεί εύκολα σε μια άλλου τύπου απλά χρησιμοποιώντας μια συνάρτηση με το όνομα του τύπου. Αυτό που παίρνουμε είναι μια νέα ακολουθία του τύπου που προσδιορίζει η συνάρτηση που χρησιμοποιήσαμε με στοιχεία τα στοιχεία της ακολουθίας που χρησιμοποιήσαμε σαν βάση.

```
list1 = [1, 2, 4]    # Ορίζει μια λίστα με τρία στοιχεία
tuple1 = tuple(list1)    # Δημιουργεί μια πλειάδα από τη λίστα
print(tuple1)    # Θα τυπώσει την πλειάδα
(1, 2, 4)
```

Τέλος οι πλειάδες χρησιμοποιούνται και στα λεξικά αλλά αυτό θα το δούμε στην αντίστοιχη ενότητα.

7.2.3 Να θυμάμαι

Πλειάδες

Η πλειάδα είναι μια δομή δεδομένων με τα εξής χαρακτηριστικά:

- ✓ είναι σύνθετη, δηλαδή χρησιμοποιείται για την αποθήκευση πολλαπλών δεδομένων ταυτόχρονα
- ✓ είναι διατεταγμένη, δηλαδή η σειρά των μελών είναι σημαντική (ακολουθία)
- ✓ είναι μη μετατρέψιμη (immutable), δηλαδή τα στοιχεία της δεν μπορούν να αλλάξουν μόλις εκχωρηθούν σε αυτά τιμές (δεδομένα)
- ✓ τα δεδομένα της μπορούν να είναι ετερογενή, δηλαδή πολλών διαφορετικών τύπων χωρίς περιορισμό

Πράξεις για ακολουθίες

Δύο πράξεις εφαρμόζονται σε όλες τις ακολουθίες η συνένωση (concatenation) και ο πολλαπλασιασμός:

```
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)
print(tuple1 + tuple2)
(1, 2, 3, 4, 5, 6)
print(2 * tuple1)
(1, 2, 3, 1, 2, 3)
```

Έλεγχος συμπερίληψης

Μπορούμε επίσης να ελέγξουμε αν ένα στοιχείο ανήκει σε μία πλειάδα ή και το αντίθετο. Αυτό γίνεται με την εφαρμογή των τελεστών `in` και `not in` για έλεγχο συμπερίληψης ή μη.

Συγκρίσεις

Και οι συγκριτικοί τελεστές (`==`, `!=`, `>=`, `>`, `<=`, `<`)μπορούν να εφαρμοστούν στις πλειάδες όπως σε όλες τις ακολουθίες. Αρκεί οι δύο πλειάδες να είναι συγκρίσιμες.

Συναρτήσεις

Μια σειρά από συναρτήσεις εφαρμόζονται σε όλες τις ακολουθίες, άρα και στις πλειάδες. Αυτές περιλαμβάνουν την εύρεση του μήκους της ακολουθίας με τη συνάρτηση `len( )` και, εφόσον οι τιμές είναι συγκρίσιμες, το μέγιστο και ελάχιστο με τις `max( )`και `min( )` αντίστοιχα.

Μέθοδοι

Μία πλειάδα, ως δομή, έχει δύο μεθόδους, τις `count( )` και `index( )`. Και οι δύο μέθοδοι εφαρμόζονται σε ένα πιθανό στοιχείο της πλειάδας και επιστρέφουν η μεν πρώτη τον αριθμό των εμφανίσεών του στην πλειάδα, η δε δεύτερη τη θέση της πρώτης εμφάνισης αποδιδόμενη με τον αντίστοιχο δείκτη.

7.2.4 Ερωτήσεις Κατανόησης

1. Η πλειάδα στην Python είναι μια απλή δομή δεδομένων.

   ΝΑΙ ΟΧΙ

2. Η πλειάδα στην Python έχει μόνο δύο μεθόδους, τις `count( )` και `index( )`.

   ΝΑΙ ΟΧΙ

3. Η πλειάδα χρησιμοποιείται κυρίως για την αποθήκευση μεταβλητών τιμών.

   ΝΑΙ ΟΧΙ

4. Σε πλειάδες αποθηκεύουμε ομοειδή δεδομένα.

   ΝΑΙ ΟΧΙ

5. Η γλώσσα Python διαθέτει συναρτήσεις που εφαρμόζονται στις ακολουθίες, όπως οι `len( )`, `max( )`και `min( )`

   ΝΑΙ ΟΧΙ

6. Οι συγκρίσεις δεν εφαρμόζονται στις πλειάδες.

   ΝΑΙ ΟΧΙ

7.2.5 Ερωτήσεις Ανάπτυξης

1. Ποια είναι τα χαρακτηριστικά της πλειάδας στην Python;
2. Ποιες ιδιαίτερες λειτουργίες της Python στηρίζονται στις πλειάδες;
3. Περιγράψτε τον έλεγχο συμπερίληψης στην Python.

7.2.6 Ασκήσεις

7.3 Η δομή της λίστας

Μια λίστα (list) διαφέρει από μια πλειάδα μόνο στο ότι είναι μετατρέψιμη (mutable), δηλαδή τα στοιχεία της μπορούν να αλλάξουν όσες φορές θέλουμε. Όταν αλλάζει κάποιο από τα στοιχεία της η λίστα παραμένει η ίδια. Σχετικά με μία συμβολοσειρά μια λίστα έχει δύο διαφορές, το ότι είναι μετατρέψιμη και το ότι τα στοιχεία της δεν είναι αποκλειστικά χαρακτήρες. Μια ακολουθία στοιχείων, είτε αυτά είναι αριθμοί, είτε συμβολοσειρές, είτε και οποιουδήποτε άλλου τύπου, μπορούν να ομαδοποιηθούν στη δομή της λίστας και να τροποποιηθούν χωρίς περιορισμούς μετά.

Αν θέλουμε να είμαστε πιο τυπικοί η λίστα είναι μια δομή δεδομένων με τα εξής χαρακτηριστικά:

- ✓ είναι σύνθετη, δηλαδή χρησιμοποιείται για την αποθήκευση πολλαπλών δεδομένων ταυτόχρονα
- ✓ είναι διατεταγμένη, δηλαδή η σειρά των μελών είναι σημαντική (ακολουθία)
- ✓ είναι μετατρέψιμη, δηλαδή τα στοιχεία της μπορούν να αλλάξουν και αφού εκχωρηθούν σε αυτά τιμές (δεδομένα)
- ✓ τα δεδομένα της μπορούν να είναι ετερογενή, δηλαδή πολλών διαφορετικών τύπων χωρίς περιορισμό

Η μεταβλητότητα είναι ένα από τα χαρακτηριστικά που καθιστούν την λίστα το πλέον, ίσως, δυναμικό εργαλείο για την Python.

Λόγω της φύσης της η λίστα χρησιμοποιείται για να ομαδοποιήσει μια ακολουθία μεταβλητών που μπορούν να αλλάξουν ανά πάσα στιγμή. Κατ΄ αυτόν τον τρόπο μπορεί να χρησιμοποιηθεί για την αποτύπωση και ομαδοποίηση των τιμών κάποιων μεταβλητών του περιβάλλοντος και άλλων ασταθών τιμών. Έχει λιγότερους περιορισμούς από την πλειάδα, πολλές περισσότερες μεθόδους αλλά και μεγαλύτερες απαιτήσεις διαχείρισης μνήμης. Προτιμάται ιδιαίτερα όταν θέλουμε να εισάγουμε αρχικές τιμές που θα τροποποιηθούν αργότερα.

Ας δούμε τους διαφορετικούς τρόπους δημιουργίας λιστών:

```
# Τρόποι δημιουργίας λιστών (lists)
list1 = [1, 2, 3, 4, 5, 6]    # Κλασική δημιουργία λίστας (list)
# που χρησιμοποιεί αγκύλες για τη δημιουργία της.
list2 = list((10, 20, 30))    # Εναλλακτικός τρόπος δημιουργίας
# λίστας χρησιμοποιώντας τη συνάρτηση list() σε κάποια
# ακολουθία.
list3 = [ ]    # Δημιουργία κενής λίστας. Μπορεί να γεμίσει
# αργότερα.
# Προσέξτε τις αγκύλες παντού εκτός από τη συνάρτηση
# μετατροπής. Δεν μπορούμε να τις παραλείψουμε γιατί τότε η Python θα
# θεωρήσει ότι πρόκειται για πλειάδα.

# Γράφοντας για τη list2:
 print(type(list2))
```

```
# θα μας επιστρέψει:
<class 'list'>
```

```
 list1[3] = 14    # Αλλάζουμε την τιμή 4 που είναι στη θέση 3
# (4η) της λίστας list1 σε 14. Και αυτό δεν προκαλεί κανένα
# σφάλμα
```

Είδαμε μόλις ότι μπορούμε να αλλάξουμε μια τιμή στη λίστα. Το ίδιο ισχύει και για ολόκληρη τη λίστα φυσικά.

Έχουμε πεί ότι μέλος/ στοιχείο μιας λίστας, μπορεί να είναι ο,τιδήποτε: ένας ακέραιος, ένα string αλλά ακόμα και μια μεταβλητή ή ένα αντικείμενο. Στην περίπτωση της μεταβλητής, μπορούμε όχι μόνο να αλλάξουμε την τιμή της μεταβλητής όπως σε μία πλειάδα αλλά και την ίδια τη μεταβλητή. Ας δούμε:

```
  x = 3
  y = 5
  list4 = [x, y]
  print(x + ' και ' + y)
3 και 5
  x = 7    # Τροποποιούμε την τιμή της μιας μεταβλητής
  print(x + ' και ' + y)
7 και 5
  z = 9
  list4 = (y, z)    # Τροποποιούμε την λίστα
  print(list4)
[5, 9]    # και όλα καλά.
```

Εφόσον και η λίστα και τα στοιχεία της μπορούν να τροποποιηθούν, αν ως στοιχεία της έχουν οριστεί μεταβλητές, τόσο οι μεταβλητές αυτές όσο και οι τιμές τους, μπορούν να αλλαχθούν, καθώς η λίστα είναι μετατρέψιμη (mutable).

7.3.1 Στοιχεία μιας λίστας (πρόσβαση, διάσχιση)

Η πρόσβαση στα στοιχεία μιας λίστας γίνεται με τον ίδιο τρόπο που γίνεται η πρόσβαση και στα στοιχεία μιας συμβολοσειράς ή μιας πλειάδας. Η αρίθμηση των θέσεων των στοιχείων ξεκινά από 0 και τελειώνει σε μέγεθος -1. Η λίστα, την οποία εξ' αρχής ορίζουμε με αγκύλες, τις διατηρεί και στη δεικτοδότηση. Μερικά παραδείγματα:

```
new_list = [1, 2, 3, 4, 5, 6]
print(new_tuple[1])    # Εκτύπωση του 2ου στοιχείου της λίστας
2
print(new_tuple[5])    # Εκτύπωση του 6ου στοιχείου της λίστας
6
```

Η πρόσβαση μπορεί να γίνει τόσο με θετική όσο και με αρνητική δεικτοδότηση. Η θέση -1 είναι η τελευταία θέση της λίστας, η -2 η προτελευταία και ούτω καθ΄ εξής. Μερικά ακόμα παραδείγματα:

```
print(new_list[-2])    # Εκτύπωση του προτελευταίου
# στοιχείου της πλειάδας
5
print(new_list [-1])    # Εκτύπωση του τελευταίου στοιχείου
# της πλειάδας
6    # Προσέξτε ότι δίνει το ίδιο αποτέλεσμα με την
# εντολή print(new_tuple[5])
```

Μπορούμε επίσης να προσπελάσουμε ένα υποσύνολο μιας λίστας, όπως ακριβώς και με την πλειάδα, αν υποδείξουμε ένα διάστημα για τον δείκτη. Το κομμάτι (slice) που θα προσπελάσουμε ορίζεται από δύο ορίσματα, την αρχική και την τελική θέση του στην πλειάδα. Όταν εκτελείται αυτή η διεργασία (slicing) σε μια λίστα, επιστρέφει μια νέα λίστα που περιλαμβάνει τα επιλεγμένα στοιχεία.

```
list_slice = new_list[0:2]'
print(list_slice)
(1, 2, 3)
```

Όπως ακριβώς στην περίπτωση της πλειάδας, για να επισκεφτούμε τα στοιχεία μιας λίστας μπορούμε να χρησιμοποιήσουμε ένα βρόγχο for. Κάθε στοιχείο της λίστας θα ανατεθεί διαδοχικά στη μεταβλητή που ορίστηκε για αυτό το σκοπό, επιτρέποντας τη χρησιμοποίησή του μέσα στο βρόγχο:

```
for x in new_list :
     print(x)  # Θα τυπώσει διαδοχικά τους αριθμούς 1,2 και 3
```

7.3.2 Βασικές λειτουργίες λίστας, πράξεις και συναρτήσεις που εφαρμόζονται σε αυτή

Όλες οι πράξεις, συναρτήσεις και τελεστές που εφαρμόζονται στις ακολουθίες εφαρμόζονται φυσικά και στις λίστες. Ας δούμε πάλι μερικές δυνατότητες.

Πράξεις για ακολουθίες

Έχουμε ήδη αναφέρει ότι δύο πράξεις εφαρμόζονται σε όλες τις ακολουθίες, άρα και στις λίστες: η συνένωση (concatenation) και ο πολλαπλασιασμός. Χρησιμοποιούνται για αυτές οι τελεστές `+` και `*` αντίστοιχα. Τέλος μπορούμε να έχουμε μια λίστα από λίστες. Ας δούμε πάλι μερικά παραδείγματα:

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
print(list1 + list2)    # Εκτύπωση της συνένωσης των δύο λιστών
[1, 2, 3, 4, 5, 6]
print(2 * list1)    # Εκτύπωση του πολλαπλασιασμού της
# λίστας επί δύο (θα έδινε το ίδιο αποτέλεσμα και το
# list1 * 2)
[1, 2, 3, 1, 2, 3]
list3 = (list1, list2)
print(list3)
[[1, 2, 3], [4, 5, 6]]    # Εκτύπωση μιας λίστας λιστών
```

Έλεγχος συμπερίληψης

Μπορούμε επίσης, όπως και στην πλειάδα, να ελέγξουμε αν ένα στοιχείο ανήκει σε μία λίστα ή και το αντίθετο. Αυτό γίνεται με την εφαρμογή των τελεστών `in` και `not in` για έλεγχο συμπερίληψης ή μη αντίστοιχα. Και οι δύο τελεστές εφαρμόζονται ανάμεσα σε ένα πιθανό στοιχείο και μία πλειάδα, δημιουργώντας μία παράσταση που μπορεί να είναι είτε αληθής είτε ψευδής. Όπως όλες οι παραστάσεις μπορεί να χρησιμοποιηθεί ως συνθήκη, για παράδειγμα σε μία εντολή επιλογής ή επανάληψης. Θυμηθείτε το έχουμε ήδη χρησιμοποιήσει τόσο στην προηγούμενη παράγραφο όσο και στις πλειάδες. Ας δούμε άλλο ένα παράδειγμα:

```
newlist = [x for x in list2 if x <= 5]    # Διατρέχουμε τη λίστα με
# έλεγχο συμπερίληψης και περιλαμβάνουμε στη νέα τα
# στοιχεία που ικανοποιούν την επιπλέον συνθήκη.
print(newlist)    # Εκτύπωση της νέας λίστας που προέκυψε από
# εφαρμογή συνθήκης στα στοιχεία της παλαιής λίστας
[4, 5]
```

Συγκρίσεις

Και στις λίστες μπορούν να εφαρμοστούν οι συγκριτικοί τελεστές. Εφόσον οι δύο λίστες είναι συγκρίσιμες, συγκρίνονται τα στοιχεία στην ίδια θέση – ξεκινώντας φυσικά από τη θέση μηδέν – μέχρι να βρεθεί διαφορετικό και τότε η ακολουθία που έχει το στοιχείο με τη μεγαλύτερη τιμή θα αντιμετωπιστεί ως μεγαλύτερη. Τα επόμενα στοιχεία, ο αριθμός τους ή οι τιμές τους δεν παίζουν ρόλο. Περίπου έτσι κατατάσσονται αλφαβητικά οι εγγραφές με βάση το επώνυμο (που είναι μία συμβολοσειρά) σε ένα τηλεφωνικό κατάλογο (που συνήθως είναι ένα λεξικό μπορεί όμως να είναι και μια λίστα αντικειμένων). Περισσότερα για αυτό αργότερα.

Συναρτήσεις

Υπάρχουν όπως έχουμε ήδη αναφέρει μια σειρά από συναρτήσεις που εφαρμόζονται και στις λίστες. Αυτές περιλαμβάνουν την εύρεση του μήκους της λίστας δηλαδή τη συνάρτηση `len( )` και, εφόσον οι τιμές είναι συγκρίσιμες μεταξύ τους, το μέγιστο και ελάχιστο στοιχείο με τις συναρτήσεις `max( )` και `min( )` αντίστοιχα. Ακόμα η, επίσης ενσωματωμένη, συνάρτηση `sorted( )` επιστρέφει μια ταξινομημένη λίστα με τα ίδια στοιχεία. Ας δούμε πώς λειτουργούν:

```
list1 = [0, 2, 1, 3, 4]
print(len(list1))    # Θα τυπώσει το μέγεθος της λίστας
5
print(max(list1))    # Θα τυπώσει τη μεγαλύτερη τιμή

4
print(min(list1))    # Θα τυπώσει τη μικρότερη τιμή

0
print(sorted(list1))    # Θα τυπώσει τη λίστα μας αλλά

# ταξινομημένη με αύξουσα σειρά. Προσοχή δεν

# αλλάζει τη λίστα απλά επιστρέφει μια άλλη,

# ταξινομημένη εκδοχή της

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

Μέθοδοι

Όλες οι ακολουθίες στην Python, άρα και οι λίστες, είναι αντικείμενα και έχουν μεθόδους. Τις λιγότερες μεθόδους από οποιονδήποτε τύπο ακολουθίας τις έχει όπως είδαμε η πλειάδα. Αντίθετα η λίστα έχει μια ποικιλία μεθόδων. Αν εφαρμόσουμε την ενσωματωμένη συνάρτηση `dir( )` σε μια λίστα, η συνάρτηση επιστρέφει μια μακροσκελή λίστα από μεθόδους. Σε αυτή περιλαμβάνονται πολλές με διπλή κάτω παύλα που είναι ιδιωτικές και δεν θα μας απασχολήσουν εδώ. Ακόμα κι έτσι μένουν έντεκα (11) δημόσιες μέθοδοι έναντι μόλις δύο (2) της πλειάδας. Αυτό είναι ένα ακόμα χαρακτηριστικό που καθιστά τη λίστα μια από τις πιο δυναμικές δομές δεδομένων στην Python. Ας δούμε ποιες είναι αυτές οι μέθοδοι και τι κάνουν:

`.append( )` Δέχεται ως όρισμα ένα στοιχείο που θέλουμε να προσθέσουμε στη λίστα και το προσθέτει στο τέλος της, αυξάνοντας έτσι το συνολικό μήκος της κατά 1.

`.clear()` Χωρίς όρισμα, διαγράφει όλα τα στοιχεία από τη λίστα.

`.copy()` Χωρίς όρισμα, επιστρέφει ένα αντίγραφο της λίστας (με αντιγραφή by value).

`.count()` Δέχεται ως όρισμα μία τιμή και επιστρέφει τον αριθμό των στοιχείων της λίστας με αυτή την τιμή.

`.extend()` Δέχεται ως όρισμα μία λίστα, ή οποιαδήποτε άλλη ακολουθία, και προσθέτει τα στοιχεία της στο τέλος της υπάρχουσας λίστας.

`.index()` Δέχεται ως όρισμα μία τιμή και επιστρέφει τη θέση στη λίστα που αυτή εμφανίζεται για πρώτη φορά.

`.insert( )` Δέχεται ως ορίσματα μία θέση στη λίστα και μια τιμή και εισάγει στη συγκεκριμένη θέση ένα στοιχείο με αυτή την τιμή. Το στοιχείο που τυχόν υπήρχε στη συγκεκριμένη θέση τοποθετείται στην επόμενη και όλα τα επόμενα σε μία επόμενη από την τρέχουσα θέση.

`.pop( )` Δέχεται ως όρισμα μία θέση στη λίστα και απομακρύνει από τη λίστα το στοιχείο που βρίσκεται στη συγκεκριμένη θέση. Οι θέσεις των επόμενων στοιχείων στη λίστα τροποποιούνται κατάλληλα ώστε στη νέα λίστα να υπάρχει συνέχεια στην αρίθμηση των θέσεων (δεικτοδότηση). Χωρίς όρισμα απομακρύνει το τελευταίο στοιχείο στη λίστα.

`.remove()` Χωρίς όρισμα, διαγράφει το πρώτο στοιχείο της λίστας και τροποποιεί κατάλληλα τη δεικτοδότηση. Μπορεί να δεχτεί ως όρισμα τιμή, την οποία θα αναζητήσει στη λίστα και θα διαγράψει το στοιχείο στο οποίο θα τη συναντήσει για πρώτη φορά.

`.reverse()` Χωρίς όρισμα, αντιστρέφει τη σειρά των στοιχείων της λίστας. Επειδή η λίστα είναι μετατρέψιμη, το αποτέλεσμα της ενέργειας είναι μόνιμο.

`.sort()` Χωρίς όρισμα, ταξινομεί τα στοιχεία της λίστας. Αντίθετα από τη συνάρτηση sorted( ), η μέθοδος .sort( ) έχει μόνιμο αποτέλεσμα στη λίστα, η οποία είναι πλέον ταξινομημένη με αύξουσα σειρά. Μπορούμε επίσης να δώσουμε ως όρισμα reverse = True και τότε θα πάρουμε τη λίστα με φθίνουσα ταξινόμηση.

Ας δούμε μερικά παραδείγματα:

```
list = [2,3,5]    # Ορίζουμε μια λίστα
list.append(6)
print(list)
[2, 3, 5, 6]
list.clear()
print(list)
 []
a = [3,4,5]
list.extend(a)
print(list)
[3, 4, 5]
list.pop()
```

```
a = list.copy()
a.insert(7,2)
a.pop(1)
print(a)
[3, 2]
list.insert(1,9)
list.reverse()
print(list)
[4, 9, 3]
list.sort(reverse = True)
[9, 4, 3]
```

Ταξινόμηση λίστας

Εφόσον η λίστα είναι μια μετατρέψιμη δομή η ταξινόμησή της είναι εφικτή, όπως είδαμε χρησιμοποιώντας τη μέθοδο .sort( ). Παρά το γεγονός ότι υπάρχει αυτή η μέθοδος μπορεί σε κάποιες περιπτώσεις να θελήσουμε να χρησιμοποιήσουμε μια συγκεκριμένη μέθοδο ταξινόμησης αλλά και εναλλακτικούς αλγόριθμους για την κάθε μία, φτιάχνοντας αντίστοιχες συναρτήσεις. Μια ενδιαφέρουσα τέτοια περίπτωση είναι η ταξινόμηση εισαγωγής.

Η συνάρτηση insertionSort που ακολουθεί δέχεται ως είσοδο μία αταξινόμητη λίστα ulist. Αρχικά υπολογίζει το μήκος της λίστας (length). Εφόσον η λίστα έχει περισσότερα από ένα στοιχεία, η συνάρτηση διασχίζει τον πίνακα ξεκινώντας από το δεύτερο στοιχείο. Λαμβάνει το τρέχον στοιχείο (current) και το συγκρίνει με τα στοιχεία στο ταξινομημένο τμήμα του πίνακα που προηγείται. Εάν το τρέχον είναι μικρότερο από ένα στοιχείο στο ταξινομημένο τμήμα, η συνάρτηση τοποθετεί το τρέχον πριν από το στοιχείο αυτό. Αυτή η διαδικασία συνεχίζεται μέχρι να βρεθεί η σωστή θέση για το τρέχον στοιχείο και να εισαχθεί σε αυτή.

```
def insertionSort(ulist):
    length = len(ulist)    # Βάλε το μήκος της λίστας σε μια μεταβλητή
    if length <= 1:
        return     # Στις τετριμμένες περιπτώσεις (0 ή 1 στοιχεία) δε
        # χρειάζεται ταξινόμηση

        for i in range(1, length):    # Διέσχισε τη λίστα ξεκινώντας από
            # το 2ο στοιχείο
        current = ulist[i]    # Αποθήκευσε το τρέχον προς ταξινόμηση
        # στοιχείο στη μεταβλητή current
        j = i-1
        while j >= 0 and current < ulist[j]:    # Όσο συναντάς
        # μεγαλύτερα στοιχεία πήγαινέ τα μια θέση δεξιά
            ulist[j+1] = ulist[j]    # Μεταφορά στοιχείων δεξιά
            j -= 1
```

```
            ulist[j+1] = current  # Βάλε το τρέχον στη σωστή θέση
```

Στην πραγματικότητα, μια που δουλεύουμε με λίστες, δε χρειάζεται να πάμε τα στοιχεία μία θέση δεξιά. Αρκεί να βάλουμε το τρέχον (current) στη σωστή θέση με την insert, που θα πάει όλα τα επόμενα στοιχεία μια θέση δεξιά, και να το διαγράψουμε από την αρχική του θέση με τη pop. Το for μας γίνεται τώρα:

```
for i in range(1, length):  # Διέσχισε τη λίστα ξεκινώντας από
     # το 2ο στοιχείο
current = ulist[i]  # Αποθήκευσε το τρέχον προς ταξινόμηση
# στοιχείο στη μεταβλητή current
j = i-1
while j >= 0 and current < ulist[j]:  # Όσο συναντάς
# μεγαλύτερα στοιχεία πηγαινέ τα μια θέση δεξιά
j -= 1 # Μετακίνηση του δείκτη στη επόμενη θέση
ulist.insert(j+1, current) # Βάλε το τρέχον στη θέση του
ulist.pop(i+1) # Διέγραψε το από την παλιά. Βάζουμε +1
# γιατί προστέθηκε στοιχείο μπροστά
```

Αναζήτηση στοιχείου σε ταξινομημένη λίστα

Και για την αναζήτηση είδαμε ότι υπάρχει η μέθοδος .index( ) που εντοπίζει τουλάχιστον την πρώτη εμφάνιση ενός στοιχείου στη λίστα. Μας επιστρέφει μάλιστα τη θέση του. Όμως υπάρχουν αρκετοί τρόποι αναζήτησης, διαφορετικών δυνατοτήτων και κυρίως ταχυτήτων. Μπορούμε να χρησιμοποιήσουμε εκτός από την .index( ), μία επανάληψη for ή ακόμα μια συνάρτηση. Πιθανόν να θέλουμε να χρησιμοποιήσουμε μια συγκεκριμένη μέθοδο αναζήτησης. Ας υποθέσουμε ότι μας ενδιαφέρει η δυαδική αναζήτηση:

```
def binary_search(list,item):
      start = 0
      end = len(list) - 1

      while start <= end :
            mid = (start + end) // 2
            current = list[mid]

            if current == item :
                  return mid
            elif current > item :
                  end = mid – 1
            else :
                  start = mid + 1
      return -1
```

```
list = [7, 8, 9, 10, 11 ]
answer = binary_search(list, 10)
if answer == -1:
      print('Item not found')
else :
      print('Item found at position', answer)
Item found at position 3
```

Μετατροπή ακολουθίας σε διαφορετικού τύπου

Όπως είδαμε στην προηγούμενη ενότητα οι ακολουθίες μπορούν να μετατραπούν σε άλλες διαφορετικού τύπου. Στην περίπτωση της προηγούμενης ενότητας μετατρέψαμε μια λίστα σε πλειάδα. Το ίδιο εύκολο είναι και το αντίστροφο:

```
tuple1 = (1, 2, 4)     # Ορίζει μια πλειάδα με τρία στοιχεία
list1= list(tuple1)     # Δημιουργεί μια λίστα από την πλειάδα
print(list1)     # Θα τυπώσει την λίστα
[1, 2, 4]
```

Αυτή η μετατροπή είναι πιο συνηθισμένη καθώς οι λίστες επιτρέπουν περισσότερους χειρισμούς των στοιχείων τους.

7.3.3 Να θυμάμαι

Λίστες

Η λίστα είναι μια δομή δεδομένων με τα εξής χαρακτηριστικά:

- ✓ είναι σύνθετη, δηλαδή χρησιμοποιείται για την αποθήκευση πολλαπλών δεδομένων ταυτόχρονα
- ✓ είναι διατεταγμένη, δηλαδή η σειρά των μελών είναι σημαντική (ακολουθία)
- ✓ είναι μετατρέψιμη (mutable), δηλαδή τα στοιχεία της μπορούν να αλλάξουν και αφού εκχωρηθούν σε αυτά τιμές
- ✓ τα δεδομένα της μπορούν να είναι ετερογενή, δηλαδή πολλών διαφορετικών τύπων χωρίς περιορισμό

Πράξεις, έλεγχος συμπερίληψης και συγκρίσεις

Όπως σε κάθε ακολουθία έτσι και στις λίστες εφαρμόζονται η συνένωση (concatenation, με τελεστή +) και ο πολλαπλασιασμός (με θετικό ακέραιο, τελεστής *). Εφαρμόζονται επίσης οι τελεστές in και not in για τον έλεγχο συμπερίληψης ή μη.

Τέλος, και με την προϋπόθεση ότι οι λίστες είναι συγκρίσιμες εφαρμόζονται και οι συγκριτικοί τελεστές όπως της ισότητας (==), διαφορετικότητας (!=), μεγαλύτερου ίσου (>=), γνήσια μεγαλύτερου (>),, μικρότερου ίσου (<=), γνήσια μικρότερου (<) κλπ. Εφόσον οι δύο λίστες είναι συγκρίσιμες, συγκρίνονται τα στοιχεία στην ίδια θέση. Αν βρεθεί διαφορετικό στοιχείο και τότε η ακολουθία που έχει το στοιχείο με τη μεγαλύτερη τιμή θα αντιμετωπιστεί ως μεγαλύτερη. Αν εξαντληθούν τα στοιχεία της μίας τότε η άλλη θεωρείται μεγαλύτερη.

```
list1 = [1, 2, 3, 4]
list2 = [1, 2, 3]
print(list1 < list2)
```

```
False
```

```
print(list1 != list2)
```

```
True
```

```
print(list1 > list2)
```

```
True
```

Μέθοδοι λίστας

Η λίστα έχει 11 δημόσιες μεθόδους, τις `.clear()` Χωρίς όρισμα, διαγράφει όλα τα στοιχεία από τη λίστα.

`.copy()` Χωρίς όρισμα, επιστρέφει ένα αντίγραφο της λίστας (με αντιγραφή by value).

`.count()` Δέχεται ως όρισμα μία τιμή και επιστρέφει τον αριθμό των στοιχείων της λίστας με αυτή την τιμή.

`.extend()` Δέχεται ως όρισμα μία λίστα, ή οποιαδήποτε άλλη ακολουθία, και προσθέτει τα στοιχεία της στο τέλος της υπάρχουσας λίστας.

`.index()` Δέχεται ως όρισμα μία τιμή και επιστρέφει τη θέση στη λίστα που αυτή εμφανίζεται για πρώτη φορά.

`.insert()` Δέχεται ως ορίσματα μία θέση στη λίστα και μια τιμή και εισάγει στη συγκεκριμένη θέση ένα στοιχείο με αυτή την τιμή. Το στοιχείο που τυχόν υπήρχε στη συγκεκριμένη θέση τοποθετείται στην επόμενη και όλα τα επόμενα σε μία επόμενη από την τρέχουσα θέση.

`.pop()` Δέχεται ως όρισμα μία θέση στη λίστα και απομακρύνει από τη λίστα το στοιχείο που βρίσκεται στη συγκεκριμένη θέση. Οι θέσεις των επόμενων στοιχείων στη λίστα τροποποιούνται κατάλληλα ώστε στη νέα λίστα να υπάρχει συνέχεια στην αρίθμηση των θέσεων (δεικτοδότηση). Χωρίς όρισμα απομακρύνει το τελευταίο στοιχείο στη λίστα.

`.remove()` Χωρίς όρισμα, διαγράφει το πρώτο στοιχείο της λίστας και τροποποιεί κατάλληλα τη δεικτοδότηση. Μπορεί να δεχτεί ως όρισμα τιμή, την οποία θα αναζητήσει στη λίστα και θα διαγράψει το στοιχείο στο οποίο θα τη συναντήσει για πρώτη φορά.

`.reverse()` Χωρίς όρισμα, αντιστρέφει τη σειρά των στοιχείων της λίστας. Επειδή η λίστα είναι μετατρέψιμη, το αποτέλεσμα της ενέργειας είναι μόνιμο.

`.sort()` Χωρίς όρισμα, ταξινομεί τα στοιχεία της λίστας. Αντίθετα από τη συνάρτηση sorted( ), η μέθοδος .sort( ) έχει μόνιμο αποτέλεσμα στη λίστα, η οποία είναι πλέον ταξινομημένη με αύξουσα σειρά. Μπορούμε επίσης να δώσουμε ως όρισμα reverse = True και τότε θα πάρουμε τη λίστα με φθίνουσα ταξινόμηση.

Ταξινόμηση και αναζήτηση

Εκτός από την συνάρτηση `sorted( )` και τη μέθοδο `.sort()` για την ταξινόμηση αλλά και τον τελεστή `in` και τις μεθόδους `.index( )` και `.count( )` για την αναζήτηση, υπάρχουν αρκετοί αλγόριθμοι ταξινόμησης και ιδιαίτερα για ταξινομημένες λίστες αρκετοί αλγόριθμοι αναζήτησης. Η ταξινόμηση με εισαγωγή (**insertion sort**) και η δυαδική αναζήτηση (**binary search**) αντίστοιχα είναι δύο τέτοια παραδείγματα.

7.3.4 Ερωτήσεις Κατανόησης

1. Η λίστα στην Python έχει μη τροποποιήσιμα δεδομένα.

ΝΑΙ ΟΧΙ

2. Στις λίστες δεν αποθηκεύουμε υποχρεωτικά ομοειδή δεδομένα.

ΝΑΙ ΟΧΙ

3. Η λίστα στην Python έχει μόνο δύο (2) μεθόδους.

ΝΑΙ ΟΧΙ

4. Η λίστα στην Python έχει δεδομένο μέγεθος (πλήθος στοιχείων) από τη στιγμή της δημιουργίας της.

ΝΑΙ ΟΧΙ

5. Ανάμεσα σε λίστες μπορούμε να εφαρμόσουμε συγκρίσεις;

ΝΑΙ ΟΧΙ

7.3.5 Ερωτήσεις Ανάπτυξης

1. Ποια είναι τα χαρακτηριστικά της λίστας στην Python;

2. Περιγράψτε τη συνένωση (concatenation) δύο λιστών στην Python.

3. Είναι, κατά την άποψή σας, η λίστα κατάλληλη δομή για εφαρμογές Internet of Things; Γιατί;

4. Περιγράψτε την απομάκρυνση ενός στοιχείου από μια λίστα.

5. Περιγράψτε τρεις (3) τρόπους για την αναζήτηση ενός στοιχείου σε μια λίστα.

6. Περιγράψτε την ταξινόμηση με εισαγωγή (insertion sort) χωρίς να χρησιμοποιήσετε κώδικα.

7. Περιγράψτε την δυαδική αναζήτηση (binary search) χωρίς να χρησιμοποιήσετε κώδικα.

7.3.6 Ασκήσεις

1. Η ταξινόμηση φυσαλίδας (bubble sort) ταξινομεί μια λίστα συγκρίνοντας επανειλημμένα διπλανά στοιχεία και ανταλλάσσοντάς τα αν η σειρά είναι λάθος. Ο αλγόριθμος διασχίζει τη λίστα πολλές φορές, κάθε φορά καταλήγοντας να μεταφέρει το εκάστοτε μεγαλύτερο αταξινόμητο στοιχείο στη σωστή του θέση προς το τέλος της λίστας (για αύξουσα ταξινόμηση). Αν έχουμε τον ακόλουθο κώδικα:

```
def bubbleSort(list):
    n = len(list)
    swapped = False
    for i in range(n-1):
        for j in range(0, n-i-1):
            if list[j] > list[j + 1]:
                swapped = True
                list[j], list[j + 1] = list[j + 1], list[j]
```

α. Σχολιάστε τι κάνει κάθε γραμμή κώδικα.

β. Εξηγείστε γιατί ο κώδικας χάνει χρόνο δηλ. δεν είναι βελτιστοποιημένος

γ. Αν ξέρουμε ότι ο κώδικας για την βελτιστοποίηση είναι ο ακόλουθος, εντοπίστε που πρέπει να μπεί (θέση και εσοχή):

```
if not swapped:
    return
```

2. Τρέξτε τον παραπάνω κώδικα για τη λίστα [60, 40, 30 20, 10, 50, 70, 80] με και χωρίς τη βελτιστοποίηση και μετρείστε πόσες φορές επαναλήφθηκε το κάθε for στην κάθε περίπτωση.

3. Μπορείτε να σκεφτείτε ένα τρόπο ταξινόμησης της παραπάνω λίστας που θα αξιοποιεί τη συνάρτηση max( ) και τη μέθοδο .index( ) για να ταξινομήσει τη λίστα κατά αύξουσα σειρά;

7.4 Η δομή του λεξικού

Ένα λεξικό (dictionary) διαφέρει από μια λίστα στο ότι δεν έχει ακέραια δεικτοδότηση. Μάλιστα, όχι μόνο οι δείκτες των στοιχείων του δεν είναι ακέραιοι αλλά επίσης δεν έχουν καν αρίθμηση. Αντίθετα έχουν ονόματα που λέγονται κλειδιά. Τα κλειδιά αυτά μπορούν να ανήκουν σε οποιοδήποτε μη τροποποιήσιμο τύπο.

Στην πράξη ένα λεξικό χρησιμοποιεί μια γενίκευση της έννοιας του δείκτη για να προσδιορίζει τα στοιχεία του. Εφόσον όμως δεν υπάρχει μια προϋπάρχουσα αριθμητική διάταξη των δεικτών (αυτή των ακέραιων αριθμών) όπως στις άλλες σύνθετες δομές που εξετάσαμε η σύνδεση μεταξύ του κλειδιού και της τιμής που προσδιορίζει πρέπει να γίνει από το ίδιο το λεξικό. Ο τρόπος που έχει επιλεγεί στην Python είναι να θεωρείται στοιχείο ενός λεξικού το ζεύγος ενός κλειδιού και μιας τιμής. Έτσι η διάταξη του στοιχείων ενός λεξικού είναι απλά αυτή της εισαγωγής των ζευγαριών σε αυτό. Η διάταξη μάλιστα ισχύει από την Python 3.7 και μετά καθώς σε προηγούμενες εκδόσεις τα λεξικά δεν ήταν καν διατεταγμένα.

Αν θέλουμε να είμαστε πιο τυπικοί το λεξικό είναι μια δομή δεδομένων με τα εξής χαρακτηριστικά:

- ✓ είναι σύνθετη, δηλαδή χρησιμοποιείται για την αποθήκευση πολλαπλών δεδομένων ταυτόχρονα
- ✓ είναι διατεταγμένη, δηλαδή η σειρά των μελών είναι συγκεκριμένη και είναι αυτή της εισαγωγής τους (ισχύει από την 3.7 και μετά)
- ✓ είναι μετατρέψιμη, δηλαδή τα στοιχεία του λεξικού μπορούν να προστεθούν, να αφαιρεθούν και να αλλάξουν και μετά την πρώτη εισαγωγή
- ✓ τα στοιχεία/ μέλη της αποτελούνται από ζεύγη κλειδιών και τιμών (δεδομένων). Τα δεδομένα μπορούν να είναι ετερογενή, δηλαδή πολλών διαφορετικών τύπων χωρίς περιορισμό, ενώ
- ✓ για τα κλειδιά υπάρχουν δύο περιορισμοί: να είναι μη τροποποιήσιμου τύπου και, κυρίως, να μην επαναλαμβάνονται τιμές κλειδιού.

Ας δούμε μερικά παραδείγματα δημιουργίας λεξικών :

```
# Τρόποι δημιουργίας λεξικών (dictionaries)
dict1 ={}    # Δημιουργία κενού λεξικού για να γεμίσουμε#
dict1['John'] = 'janitor'   # Προσθέτουμε ζευγάρια κλειδιού- τιμής
dict1['Mary'] = 'accountant'
dict1['George'] = 'guard'
dict1['Michael'] = 'programmer'
dict1['Jane'] = 'programmer'    # Ενώ το κλειδί δεν μπορεί να
# εμφανίζεται δύο φορές, η τιμή φυσικά μπορεί.
print(dict1)
{'John': 'janitor', 'Mary': 'accountant', 'George': 'guard', 'Michael': 'programmer',
'Jane': 'programmer'}
dict2 = { 'one' : 'uno', 'two' : 'due', 'three' : 'tre'}    # Απευθείας
# ορισμός με τα ζεύγη κλειδιών-τιμών. Προσέξτε ότι
```

```
# αντί για ίσον ( = ) χρησιμοποιούμε άνω-κάτω τελεία ( : )
dict3 = dict(one = 'unos', two = 'dos', three = 'tres')     # Ορισμός μέσω
# της συνάρτησης dict(). Προσέξτε ότι εδώ χρησιμοποιούμε
# και πάλι το ίσον ( = ) και τα κλειδιά δεν μπαίνουν σε
# εισαγωγικά.

print(type(dict1))    # Αν αναζητήσουμε τον τύπο της λίστας θα
# πάρουμε σε όλες τις περιπτώσεις:
<class 'dict'>
```

Επίσης μπορούμε να αξιοποιήσουμε την συνάρτηση `dict( )` εάν έχουμε είτε μια ακολουθία από πλειάδες των δύο είτε δύο ακολουθίες με ίδιο μέγεθος (σε συνδυασμό με τη `zip( )` σε αυτή την περίπτωση) για να δημιουργήσουμε ένα λεξικό.

```
list1 = [('one', 1),('two', 2), ('three', 3)]
dict3 = dict(list1)
print(dict3)
{'one': 1, 'two': 2, 'three': 3}
dict4 = dict(zip(['a', 'b', 'c'],[1,2,3]))
print(dict4)
{'a': 1, 'b': 2, 'c': 3}
```

7.4.1 Στοιχεία ενός λεξικού (πρόσβαση, διάσχιση)

Στην περίπτωση των λεξικών η πρόσβαση στα στοιχεία τους μπορεί να είναι τριών ειδών: πρόσβαση στα κλειδιά (keys), στις τιμές (values), ή συνολικά στα στοιχεία (items).

Η πρόσβαση στις τιμές είναι η ευκολότερη αφού – όπως όλες οι ακολουθίες – το λεξικό έχει κατασκευαστεί έχοντας υπόψη την πρόσβαση στις τιμές που αποθηκεύει. Αξιοποιεί μάλιστα για το σκοπό αυτό τα κλειδιά όπως οι προηγούμενες ακολουθίες που εξετάσαμε αξιοποιούν τους δείκτες θέσης. Μερικά παραδείγματα (υποθέτοντας το dict1 της προηγούμενης παραγράφου):

```
print( dict1[“John”])     # Εκτύπωση τιμής του κλειδιού “John”
janitor
```

Υπάρχει και η μέθοδος get( ) του λεξικού με την οποία επίσης δίνεται πρόσβαση στις τιμές, με παρόμοιο τρόπο:

```
print( dict1.get(“Michael”))
programmer
```

Για πρόσβαση γενικά στα κλειδιά ή στις τιμές το λεξικό μας προσφέρει δύο ακόμα μεθόδους τις .keys( ) και .values( ). Η πρώτη μας επιστρέφει όλα τα κλειδιά και η δεύτερη όλες τις τιμές του λεξικού μας. Επιστρέφουν μια συλλογή κλειδιών με όνομα dict_keys και dict_values αντίστοιχα. Ατυχώς αυτά έχουν οριστεί ως αντικείμενα με το αντίστοιχο όνομα και δεν είναι ορισμένα με τρόπο ώστε να διασχίζονται με βάση τη θέση Φυσικά οι τιμές δεν θα μπορούσαν εύκολα να οργανωθούν έτσι εφόσον επιτρέπουν την επανάληψη και οι ακολουθίες στην Python όχι . Ας τα δούμε όμως λίγο:

```
print( dict1.keys( ))     # Ζητάμε τα κλειδιά της λίστας μας
dict_keys(['John', 'Mary', 'George', 'Michael', 'Jane'])
print( dict1.values( ))     )))     # Ζητάμε τις τιμές της λίστας μας
dict_values(['janitor', 'accountant', 'guard', 'programmer', 'programmer'])
print( type(dict1.keys( )))     # Ζητάμε τον τύπο του dict1.keys
<class 'dict_keys'>
```

Για να επισκεφτούμε με τη σειρά (διάσχιση) τα στοιχεία ενός λεξικού μπορούμε να χρησιμοποιήσουμε ένα βρόγχο σταθερού μεγέθους (δηλαδή ένα βρόγχο for). Η διάσχιση μπορεί να γίνει ως προς τα κλειδιά, ως προς τις τιμές ή ως προς τα στοιχεία.

Αν δοκιμάσουμε πρώτα με το κλειδί, κάθε κλειδί στο λεξικό μας θα ανατεθεί διαδοχικά στη μεταβλητή που ορίστηκε για αυτό το σκοπό, επιτρέποντας τη χρησιμοποίησή του μέσα στο βρόγχο:

```
for x in dict1.keys( ):     # Το x διασχίζει τα κλειδιά του
# λεξικού μας
      print(x)
      print(dict1[x])     # Για να βρούμε την τιμή πρέπει να πάμε
# μέσω του κλειδιού
John janitor Mary accountant George guard Michael programmer Jane programmer
for x in dict1:     # Το x και πάλι διασχίζει τα κλειδιά του
# λεξικού μας
```

```
      print(x)
      print(dict1[x])
# Με το ίδιο ακριβώς αποτέλεσμα:
John janitor Mary accountant George guard Michael programmer Jane programmer
```

Με έμμεσο τρόπο – από το κλειδί – μας δόθηκε πρόσβαση και στην αντίστοιχη τιμή. Ας δούμε όμως τι γίνεται αν διασχίσουμε τις τιμές:

```
for y in dict1.values( ):    # Το y διασχίζει τις τιμές του λεξικού μας
      print(x)
janitor accountant guard programmer programmer
```

Δεν υπάρχει όμως τώρα άμεσος τρόπος να εντοπίσουμε πιο είναι το αντίστοιχο κλειδί, οπότε δεν μπορούμε άμεσα να το επεξεργαστούμε.

Υπάρχουν όμως τρόποι να διασχίσουμε τα στοιχεία (items) του λεξικού μας. Τον απλούστερο μας τον προσφέρει μία ακόμα μέθοδος του αντικειμένου λεξικό η `.items( )`.

```
for x,y in dict1.items( ):    # Το x και πάλι διασχίζει τα κλειδιά
# ενώ το y διασχίζει τις τιμές του λεξικού μας
      print(x ,y)
John janitor Mary accountant George guard Michael programmer Jane programmer
# Αν όμως δε χρειαζόμαστε να επεξεργαστούμε
# ξεχωριστά κλειδί και τιμή, υπάρχει απλούστερη λύση:
for x in dict1.items( ):    # Το x αυτή τη φορά διασχίζει τα
# στοιχεία του λεξικού μας
      print(x )
('John', 'janitor') ('Mary', 'accountant') ('George', 'guard') ('Michael', 'programmer')
('Jane', 'programmer')
```

7.4.2 Βασικές λειτουργίες λεξικού, πράξεις και συναρτήσεις που εφαρμόζονται σε αυτό

Τα λεξικά παρά το γεγονός ότι είναι πλέον διατεταγμένα, δεν αντιμετωπίζονται στην Python ως ακολουθίες. Ως εκ τούτου οι πράξεις που εφαρμόζονται σε όλες τις ακολουθίες (iterables) δεν εφαρμόζονται σε αυτά. Επομένως δεν υπάρχει συνένωση (concatenation) ούτε και πολλαπλασιασμός λεξικών. Επίσης δεν υπάρχει δεικτοδότηση (πέραν των κλειδιών) και κατάτμηση. Αυτό δεν σημαίνει ότι δεν προσφέρονται άλλες δυνατότητες. Ας δούμε μερικές:

Έλεγχος συμπερίληψης

Μπορούμε να ελέγξουμε τρία πράγματα: αν ένα στοιχείο ανήκει σε ένα λεξικό, αν ένα κλειδί περιλαμβάνεται στα κλειδιά του λεξικού και αν μια τιμή εμφανίζεται στις τιμές του λεξικού. Μπορούμε επίσης να ελέγξουμε και το αντίθετο. Αυτό γίνεται με την εφαρμογή των τελεστών `in` και `not in` για έλεγχο συμπερίληψης ή μη αντίστοιχα. Τον πρώτο τελεστή τον έχουμε ήδη χρησιμοποιήσει στη διάσχιση του λεξικού. Αντίστοιχα λειτουργεί και ο not in.

```
print("Simon" not in dict1)
# Χρησιμοποιήσαμε εδώ τη «σύντομη» διατύπωση για τα κλειδιά
```

Συγκρίσεις

Δυο λεξικά μπορούν να συγκριθούν ως προς την ισότητα (τελεστής ==) και βρίσκονται ίσα (True) τότε και μόνο τότε αν έχουν ακριβώς τα ίδια ζεύγη κλειδιών-τιμών (περιέργως ασχέτως σειράς). Σε κάθε άλλη περίπτωση κρίνονται άνισα (False). Επίσης, κάθε άλλη σύγκριση, ανάλογα με την έκδοση Python που χρησιμοποιείται θα δώσει False ή Error.

```
dict1 = {'one':'uno', 'two':'due', 'three':'tre'}
dict2 = {'three':'tre', 'one':'uno', 'two':'due'}
print(dict1 == dict2)
dict2['three'] = 'tres'
print(dict1 == dict2)
```

Συναρτήσεις

Μια σειρά από συναρτήσεις εφαρμόζονται στα λεξικά. Αυτές περιλαμβάνουν την εύρεση του μήκους του λεξικού δηλαδή τη συνάρτηση `len( )` και, το μέγιστο και ελάχιστο με τις συναρτήσεις `max( )`και `min( )` αντίστοιχα. Οι τελευταίες επιστρέφουν όμως το μέγιστο και ελάχιστο **κλειδί**. Ακόμα η, επίσης ενσωματωμένη, συνάρτηση `sorted( )` επιστρέφει μια ταξινομημένη λίστα με τα κλειδιά του λεξικού μας αν δεν επιλέξουμε κλειδιά, τιμές ή στοιχεία. Αν επιλέξουμε `.keys( )` ή `.items( )` η διάταξη είναι με βάση τις τιμές των κλειδιών. Αν επιλέξουμε `.values( )` παίρνουμε διατεταγμένες τις τιμές. Ας δούμε κι εδώ κάποια παραδείγματα θεωρώντας ήδη ορισμένα τα λεξικά της προηγούμενης ενότητας: sorted, dict, set

```
dict1 = {'one':'uno', 'two':'due', 'three':'tre'}
dict2 = {'three':'tre', 'one':'uno', 'two':'due'}
print(dict1.keys())
```

```
dict_keys(['one', 'two', 'three'])
print(max(dict1))
two     # επειδή τα συγκρίνει αλφαβητικά και όχι ως αριθμούς, οπότε 'w' > 'h'
print(sorted(dict2.values()))
['due', 'tres', 'uno']
print(sorted(dict2.items()))
[('one', 'uno'), ('three', 'tres'), ('two', 'due')]
```

Τέλος μπορούμε να χρησιμοποιήσουμε τη συνάρτηση set( ) στην περίπτωση που θέλουμε, ας πούμε, να δούμε τις τιμές του λεξικού μας χωρίς επαναλήψεις.

```
dict2['two'] = 'tres'
set1 = set(dict2.values())
print( set1 )
{'uno', 'tres'}
# Το ένα 'tres'παραλείπεται από το σύνολο
```

Μέθοδοι

Και εδώ για να δούμε τις μεθόδους ενός αντικειμένου μπορούμε να χρησιμοποιήσουμε την ενσωματωμένη συνάρτηση dir( ). Μας ενδιαφέρουν οι φανερές χωρίς διπλή κάτω παύλα. Έχουμε ήδη συναντήσει τις .keys( ), .values( ), .items( ) και .get( ). Για λόγους πληρότητας θα τις αναφέρουμε όλες:

.copy( ) Χωρίς όρισμα, επιστρέφει ένα αντίγραφο της λίστας (με αντιγραφή by value).

.clear( ) Απομακρύνει όλα τα στοιχεία (items) από το λεξικό

.get( ) Με όρισμα ένα κλειδί. Επιστρέφει την τιμή που αντιστοιχεί στο κλειδί που δίνεται ως όρισμα αλλιώς επιστρέφει KeyError!. Μπορεί όμως να πάρει και δεύτερο όρισμα (υποχρεωτικά τη λέξη default ή την έκφραση default = τιμή) – με αυτόν τον τρόπο μπορούμε να δώσουμε μια τιμή για την περίπτωση που το κλειδί δεν υπάρχει στο λεξικό. Αν δεν έχει δοθεί τιμή, επιστρέφει none. Με αυτόν τον τρόπο δεν επιστρέφει ποτέ KeyError!

.items( ) Με δύο παραμέτρους, ένα κλειδί και μια τιμή. Μετατρέπει τα ζευγάρια κλειδιού-τιμής σε πλειάδες των δύο και επιστρέφει μια συλλογή. Η συλλογή μπορεί να χρησιμοποιηθεί σε επαναλήψεις (iterable).

.keys( ) Χωρίς παραμέτρους. Επιστρέφει μια διατρέξιμη συλλογή των κλειδιών του λεξικού μας

.values( ) Χωρίς παραμέτρους. Επιστρέφει μια διατρέξιμη συλλογή των τιμών του λεξικού μας

.pop( ) Με μία παράμετρο. Απομακρύνει από το λεξικό το ζευγάρι κλειδιού-τιμής του οποίου το κλειδί έχει δοθεί ως όρισμα

.popitem( ) Απομακρύνει από το λεξικό το ζευγάρι κλειδιού-τιμής που προστέθηκε τελευταίο

.set_default( ) Με δύο παραμέτρους, ένα κλειδί και μια τιμή. Επιστρέφει την τιμή που αντιστοιχεί στο κλειδί όρισμα. Αν το κλειδί δεν υπάρχει στο λεξικό τότε προσθέτει το κλειδί (πρώτο όρισμα) με τιμή αυτήν που δόθηκε (δεύτερο όρισμα).

.update( ) Με μία παράμετρο, ένα λεξικό, συνήθως με ένα κλειδί και μια τιμή μόνο. Προσθέτει το ζεύγος κλειδιού και τιμής που λαμβάνει ως ορίσματα στο τέλος του λεξικού. Στην περίπτωση των πολλαπλών ζευγών λειτουργεί σαν την συνένωση (concatenation) ακολουθιών: προσθέτει το λεξικό που παίρνει ως όρισμα μετά από το λεξικό που καλεί τη συνάρτηση.

```
worker = {‘name’:( ‘John’, ‘Daglas’), ‘age’: 43, ‘salary’: 30000 }
print(worker.get(‘name’))
('John', 'Daglas')   # Επιστρέφει την πλειάδα που δώσαμε
ως τιμή στο ‘name’
worker.update({ ‘office’: ‘central’})    # Προσέξτε τις εσωτερικές
# αγκύλες
print(worker)    # Θα τυπώσει το νέο λεξικό
{'name': ('John', 'Daglas'), 'age': 43, 'salary': 30000, 'office': 'central'}
worker.popitem( )
print(worker)    # Θα τυπώσει το νέο λεξικό
{'name': ('John', 'Daglas'), 'age': 43, 'salary': 30000}
worker.set_default(‘office’: ‘central’)    # Προσέξτε ότι δε
# χρειάζεται εσωτερικές αγκύλες (αφού δέχεται μόνο ένα
# ζεύγος)
print(worker)    # Θα τυπώσει το νέο λεξικό
{'name': ('John', 'Daglas'), 'age': 43, 'salary': 30000, 'office': 'central'}
worker.clear( )
{}
```

7.4.3 Να θυμάμαι

Λεξικά

Το λεξικό είναι μια δομή δεδομένων με τα εξής χαρακτηριστικά:

- ✓ είναι σύνθετη, δηλαδή χρησιμοποιείται για την αποθήκευση πολλαπλών δεδομένων ταυτόχρονα
- ✓ είναι διατεταγμένη, δηλαδή η σειρά των μελών είναι συγκεκριμένη και είναι αυτή της εισαγωγής τους (ισχύει από την 3.7 και μετά)
- ✓ είναι μετατρέψιμη, δηλαδή τα στοιχεία του λεξικού μπορούν να προστεθούν, να αφαιρεθούν και να αλλάξουν και μετά την πρώτη εισαγωγή
- ✓ τα στοιχεία/ μέλη της αποτελούνται από ζεύγη κλειδιών και τιμών (δεδομένων). Τα δεδομένα μπορούν να είναι ετερογενή, δηλαδή πολλών διαφορετικών τύπων χωρίς περιορισμό, ενώ
- ✓ για τα κλειδιά υπάρχουν δύο περιορισμοί: να είναι μη τροποποιήσιμου τύπου και, κυρίως, να μην επαναλαμβάνονται τιμές κλειδιού.

Δημιουργία λεξικού

Ανήγοντας ένα κενό λεξικό και ενημερώνοντας με τιμές για συγκεκριμένα κλειδιά:

```
dict1 ={}     # Δημιουργία κενού λεξικού για να γεμίσουμε#
dict1['John'] = 'janitor'    # Προσθέτουμε ζευγάρια κλειδιού- τιμής
dict1['Mary'] = 'accountant'
dict1['George'] = 'guard'
```

Με παράθεση ζευγών κλειδιού-τιμής:

```
dict2 = { 'one' : 'uno', 'two' : 'due', 'three' : 'tre'}
```

Χρησιμοποιώντας τη συνάρτηση `dict( )`

```
dict3 = dict(one = 'unos', two = 'dos', three = 'tres')
```

Συνδιάζοντας την συνάρτηση `dict( )`με την συνάρτηση `zip( )` και εφαρμόζοντας τον συνδυασμό σε δύο ισομεγέθεις ακολουθίες οποιουδήποτε τύπου.

```
dict4 = dict(zip(['a', 'b', 'c'],[1,2,3]))
```

Οι τρόποι δεν εξαντλούνται εδώ.

Διάσχιση ενός λεξικού

Τα λεξικά μας προσφέρουν τρεις μεθόδους, τις `.keys( )`,`.values( )` και `.items( )`, για τη διάσχισή τους. Οι `.keys( )` και `.items( )` διασχίζουν μέσω των κλειδιών και προσφέρουν πρόσβαση και στις τιμές, οι πρώτη έμμεσα και η δεύτερη άμεσα. Η `.values( )` μας επιτρέπει τη διάσχιση των τιμών, χωρίς όμως ταυτόχρονα να προσφέρει εύκολη πρόσβαση στα κλειδιά.

Και στις τρεις περιπτώσεις αξιοποιείται ο τελεστής `in` μέσα σε ένα βρόγχο σταθερών επαναλήψεων (`for`). Προφανώς είναι εφικτός έλεγχος συμπερίληψης.

Συγκρίσεις

Η μοναδική σύγκριση που έχει νόημα μεταξύ λεξικών είναι ως προς την ισότητα (`==`). Δύο λεξικά βρίσκονται ίσα (True) τότε και μόνο τότε αν έχουν ακριβώς τα ίδια ζεύγη κλειδιών-τιμών (ασχέτως σειράς)

Συναρτήσεις

Μια σειρά από ενσωματωμένες συναρτήσεις εφαρμόζονται σε όλες τις ακολουθίες, άρα και στα λεξικά. Αυτές περιλαμβάνουν την εύρεση του μήκους της ακολουθίας δηλαδή τη συνάρτηση `len( )` και, εφόσον οι τιμές είναι συγκρίσιμες, το μέγιστο και ελάχιστο στοιχείο με τις `max( )` και `min( )` αντίστοιχα. Εφαρμόζεται και η συνάρτηση `dir( )` που επιστρέφει όλες τις μεθόδους της δομής δεδομένων ως αντικείμενο. Επίσης εφαρμόζεται η συνάρτηση `sorted( )` που επιστρέφει μια ταξινομημένη λίστα με τα κλειδιά του λεξικού μας αν δεν επιλέξουμε κλειδιά, τιμές ή στοιχεία. Αν επιλέξουμε `.keys( )` ή `.items( )` η διάταξη είναι αυτή των κλειδιών. Αν επιλέξουμε `.values( )` παίρνουμε διατεταγμένες τις τιμές.

Μέθοδοι

Ένα λεξικό ως δομή , έχει δέκα (10) δημόσιες μεθόδους τις εξής:

`.copy( )` Χωρίς όρισμα, επιστρέφει ένα αντίγραφο της λίστας (με αντιγραφή by value).

`.clear( )` Απομακρύνει όλα τα στοιχεία (items) από το λεξικό

`.get( )` Επιστρέφει την τιμή που αντιστοιχεί στο κλειδί- όρισμα. Αν χρησιμοποιήσουμε `default` τότε δεν επιστρέφει KeyError! Επιστρέφει την τιμή που δώσαμε ή none.

`.items( )` Με δύο παραμέτρους, ένα κλειδί και μια τιμή. Μετατρέπει τα ζευγάρια κλειδιού-τιμής σε πλειάδες των δύο και επιστρέφει μια συλλογή που μπορεί να χρησιμοποιηθεί σε επαναλήψεις (iterable)

`.keys( )` Χωρίς παραμέτρους. Επιστρέφει μια διατρέξιμη συλλογή των κλειδιών του λεξικού.

`.values( )` Χωρίς παραμέτρους. Επιστρέφει μια διατρέξιμη συλλογή των τιμών του λεξικού.

`.pop( )` Με μία παράμετρο. Απομακρύνει από το λεξικό το ζευγάρι κλειδιού-τιμής του οποίου το κλειδί έχει δοθεί ως όρισμα

`.popitem( )` Απομακρύνει από το λεξικό το ζευγάρι κλειδιού-τιμής που προστέθηκε τελευταίο

`.set_default( )` Με δύο παραμέτρους, ένα κλειδί και μια τιμή. Επιστρέφει την τιμή που αντιστοιχεί στο κλειδί όρισμα. Αν το κλειδί δεν υπάρχει στο λεξικό τότε προσθέτει το κλειδί (πρώτο όρισμα) με τιμή αυτήν που δόθηκε (δεύτερο όρισμα).

`.update( )` Με μία παράμετρο, ένα λεξικό, συνήθως με ένα κλειδί και μια τιμή μόνο. Προσθέτει το ζεύγος κλειδιού και τιμής που λαμβάνει ως όρισμα στο τέλος του λεξικού. Στην περίπτωση των πολλαπλών ζευγών λειτουργεί σαν την συνένωση (concatenation) ακολουθιών: προσθέτει το λεξικό που παίρνει ως όρισμα μετά από το λεξικό που καλεί τη συνάρτηση..

7.4.4 Ερωτήσεις Κατανόησης

1. Οι τιμές του λεξικού δεν αλλάζουν μετά την πρώτη εισαγωγή.

   ΝΑΙ ΟΧΙ

2 Το λεξικό έχει δέκα (1) δημόσιες μεθόδους.

   ΝΑΙ ΟΧΙ

4. Τα λεξικά ορίζονται μόνο με ζευγάρια κλειδιών – τιμών (παράθεση στοιχείων).

   ΝΑΙ ΟΧΙ

5. Το λεξικό διασχίζεται με τα κλειδιά του μόνο.

   ΝΑΙ ΟΧΙ

6. Η σύγκριση λεξικών έχει νόημα μόνο ως προς την ισότητα.

   ΝΑΙ ΟΧΙ

7.4.5 Ερωτήσεις Ανάπτυξης

1. Ποια είναι τα χαρακτηριστικά του λεξικού στην Python;

2. Περιγράψτε τον τρόπο που ελέγχεται η ισότητα στην περίπτωση των λεξικών

3. Περιγράψτε τον έλεγχο συμπερίληψης σε λεξικό.

4. Περιγράψτε τους τρεις κύριους τρόπους διάσχισης ενός λεξικού. Μπορείτε να εντοπίσετε πλεονεκτήματα ή και μειονεκτήματα;

7.4.6 Ασκήσεις

1. Στο πρώτο παράδειγμα της παραγράφου (7.3) χρησιμοποιήσαμε τα ονόματα υπαλλήλων ως κλειδιά και τους ρόλους τους στην επιχείρηση ως τιμές. Προφανώς παραπάνω από ένα άτομα μπορούν να έχουν τον ίδιο ρόλο. Έτσι αν χρησιμοποιηθούν οι ρόλοι ως κλειδιά μπορούμε να έχουμε ως τιμές μια λίστα με τα ονόματα. Σκεφτείτε μια επιχείρηση με 10 υπαλλήλους, προσδιορείστε τους πιθανούς ρόλους και ενημερώστε το σχετικό λεξικό έτσι ώστε οι εργαζόμενοι με τον ίδιο ρόλο να εισάγονται σε μία τιμή λίστα για το κλειδί αυτό. Τυπώστε το ενημερωμένο λεξικό σας.

2. Υποθέστε ότι δύο ομάδες ποδοσφαίρου ενοποιούνται. Επιλέξτε τις αγαπημένες σας. Ευτυχώς τα λεξικά για τους παίχτες τους είναι οργανωμένα με τον ίδιο τρόπο, που περιγράφεται στην προηγούμενη άσκηση. Τα κλειδιά είναι και στις δύο περιπτώσεις μέλη της λίστας [‘επιθετικός’, ‘μέσος’, ‘αμυντικός’, ‘τερματοφύλακας’]. Οι παίκτες είναι τουλάχιστον 15 για την κάθε ομάδα.

α. Ενημερώστε τα λεξικά των ομάδων και ενώστε τα για να εξυπηρετείται η νέα.

β. Αν ο προπονητής έχει εντοπίσει έναν ‘λίμπερο’ με το όνομα ‘Ταχυπόδης’ μπορούμε να τον προσθέσουμε και πώς;

γ. τυπώστε το τελικό λεξικό.

3. Κατασκευάστε ένα λεξικό με κλειδί μια πλειάδα με το ονοματεπώνυμο και τιμή μια λίστα με τα τηλέφωνα των φίλων σας. Εισάγετε τους πλησιέστερους. Είναι εξυπηρετική η οργάνωση αυτή;

7.5 Η δομή του συνόλου

Ένα σύνολο έχει μία κύρια διαφορά από μία πλειάδα: είναι αδιάτακτο. Όχι μη διατεταγμένο αλλά αδιάτακτο, επειδή αυτή είναι μια επιλογή που έχει γίνει με πρόθεση. Η επιλογή προσφέρει μεγάλα πλεονεκτήματα σε επίπεδο ταχύτητας αλλά εδώ δεν θα ασχοληθούμε με αυτά. Μας ενδιαφέρει περισσότερο να εντοπίσουμε πράγματα που γίνονται ευκολότερα με ένα σύνολο από ότι με μια πλειάδα ή μια λίστα.

Στην πραγματικότητα υπάρχουν περισσότερες διαφορές, αν και κάποιες είναι άμεση συνέπεια αυτής της πρώτης. Για παράδειγμα σε ένα σύνολο δεν επιτρέπονται επαναλήψεις. Καθόλου. Σε βαθμό που το True και το 1, πιθανότατα επειδή λειτουργούν με τον ίδιο τρόπο σε λογικές εκφράσεις, θεωρούνται επανάληψη και δεν μπορούν να υπάρχουν ταυτόχρονα σε ένα σύνολο.

Υπάρχουν όμως και άλλες. Για παράδειγμα το σύνολο είναι ανάποδο από την πλειάδα σε τούτο: το ίδιο είναι τροποποιήσιμο (mutable) αφού μπορείς να αφαιρέσεις ή να προσθέσεις στοιχεία. Τα στοιχεία του όμως είναι μη τροποποιήσιμα: από τη στιγμή που θα βάλεις ένα στοιχείο σε ένα σύνολο μπορείς μόνο να το βγάλεις, όχι να το τροποποιήσεις. Γι αυτό και θα βρεις πολλές αναφορές να προσπαθούν να το περιγράψουν με μια λέξη όπως παράδειγμα 'αμετάβλητο' (unchangeable).

Με βάση τα παραπάνω το σύνολο είναι ένας ενσωματωμένος σύνθετος τύπος δεδομένων που περιλαμβάνει ένα μεταβλητό αριθμό αμετάβλητων στοιχείων. Επιπλέον τα στοιχεία (μέλη) ενός συνόλου οφείλουν να είναι και συγκρίσιμα – μεταξύ τους και με άλλα δεδομένα. Μπορούμε να πάμε σε μεγάλο βάθος περιγραφής χρησιμοποιώντας όρους όπως hashable και μεθόδους **__hash__** και **__eq__** αλλά η ουσία του πράγματος είναι αυτή.

Αν θέλουμε να είμαστε πιο τυπικοί το σύνολο είναι μια δομή δεδομένων με τα εξής χαρακτηριστικά:

- ✓ είναι σύνθετη, δηλαδή χρησιμοποιείται για την αποθήκευση πολλαπλών δεδομένων ταυτόχρονα
- ✓ είναι αδιάτακτη, δηλαδή η σειρά των μελών της είναι αδιάφορη
- ✓ είναι τροποποιήσιμη (mutable), με την έννοια ότι μπορούμε να προσθέσουμε και να αφαιρέσουμε στοιχεία όποτε θέλουμε
- ✓ τα στοιχεία/ δεδομένα της, από την άλλη, είναι μη τροποποιήσιμα (immutable)
- ✓ δεν επιτρέπεται η επανάληψη στοιχείου
- ✓ τα στοιχεία της πρέπει να είναι συγκρίσιμα (ισχύει και για τις ακολουθίες αλλά όχι για τα λεξικά)
- ✓ τα στοιχεία/ δεδομένα της μπορούν να είναι ετερογενή

Αυτό τα δύο τελευταία χαρακτηριστικά, σε συνδυασμό με μια πληθώρα μεθόδων αλλά και την ταχύτητα των υλοποιήσεών τους, είναι που καθιστούν το σύνολο ένα πολυεργαλείο για την Python.

Ας δούμε μερικά παραδείγματα διαφορετικών τρόπων δημιουργίας συνόλων (sets):

```
# Τρόποι δημιουργίας συνόλων (sets)
set1 = {1, 2, 3, 4, 5, 6}    # Κλασική δημιουργία συνόλου (set)
# με αγκύλες και απαρίθμηση των στοιχείων του
```

```
set2 = set([10, 20, 30])    # Εναλλακτικός τρόπος δημιουργίας
# συνόλου χρησιμοποιώντας τη συνάρτηση set() και
# κάποια ακολουθία.
set3 = {5}    # Αρχική δημιουργία ενός υποσυνόλου
# Προσοχή: Δεν μπορεί να είναι το κενό γιατί,
# περιέργως, θα το θεωρήσει λεξικό
set3.add(10)
set3.add(15)    # Διαδοχικές προσθήκες στοιχείων με τη
# μέθοδο .add()
print(set1)    # Ακολουθούν εκτυπώσεις των τριών συνόλων
{1, 2, 3, 4, 5, 6}
print(set2)
{10, 20, 30}
print(set3)
{10, 5, 15}
print(type(set3))
<class 'set'>
```

Προσοχή! Όπως αναφέραμε τα στοιχεία ενός συνόλου δεν μπορεί να είναι τροποποιήσιμα. Δεδομένου ότι ένα σύνολο είναι τροποποιήσιμο συμπεραίνουμε ότι δεν μπορούμε να έχουμε π.χ. ένα σύνολο συνόλων. Επίσης, δεν μπορούμε να χρησιμοποιήσουμε ένα σύνολο ως κλειδί σε ένα λεξικό. Επειδή όμως και οι δύο αυτές ιδιαίτερες περιπτώσεις έχουν ένα ενδιαφέρον υπάρχει ένας επιπλέον τύπος συνόλων που είναι μη τροποποιήσιμος (immutable) και χρησιμοποιείται ειδικά για αυτές τις περιπτώσεις. Τα σύνολα αυτά ονομάζονται πολύ σωστά frozensets. Δεν θα μας απασχολήσουν άλλο εδώ απλά αν συναντήσουμε κάπου απρόοπτα ένα σύνολο, γνωρίζουμε ότι αυτό θα ανήκει στη συγκεκριμένη κατηγορία.

7.5.1 Στοιχεία ενός συνόλου (πρόσβαση, διάσχιση)

Εφόσον το σύνολο είναι αδιάτακτο δεν μπορούμε να έχουμε πρόσβαση στα στοιχεία του μέσα από ένα δείκτη ή κλειδί ή κάποιο άλλο προσδιοριστικό θέσης. Τα στοιχεία του δεν έχουν κάποια δεδομένη σειρά. Μπορούμε όμως να χρησιμοποιήσουμε το γνωστό βρόγχο for σε συνδυασμό με τον τελεστή αναζήτησης in. Κάθε στοιχείο του συνόλου, με τυχαία σειρά, θα ανατεθεί διαδοχικά στη μεταβλητή που ορίστηκε για αυτό το σκοπό, επιτρέποντας τη χρησιμοποίησή του μέσα στο βρόγχο:

```
new_set = {‘pencil’, ‘pen’, ‘rubber’, ‘book’}
for x in new_set:
      print(x)
pen rubber pencil book
```

Η σειρά, εάν το ξανατρέξουμε, θα είναι διαφορετική.

Ένας άλλος τρόπος για πρόσβαση σε όλα τα στοιχεία ενός συνόλου είναι η συνάρτηση iter( ). Τη συνάρτηση αυτή μπορούμε να χρησιμοποιήσουμε σε κάθε συλλογή. Δεν την αναφέραμε στις περιπτώσεις πλειάδας, λίστας και λεξικού γιατί εκεί είχαμε ποικιλία τρόπων πρόσβασης. Ας δούμε πώς δουλεύει. Υποθέτουμε το ίδιο σύνολο new_set.

```
new_set_iter = iter(new_set)
for i in range(len(new_set)):    # Χρησιμοποιούμε το μέγεθος του
# συνόλου - που ονομάζεται μήκος (length) μόνο για λόγους
# ομοιομορφίας – γιατί ο iterator που κατασκευάσαμε με
# τη συνάρτηση iter( ) δεν έχει τέτοιο χαρακτηριστικό.
      print(next(new_set_iter))    # Η next δίνει το επόμενο στοιχείο
# του iterator
pen rubber pencil book
```

7.5.2 Βασικές λειτουργίες συνόλου, πράξεις και συναρτήσεις που εφαρμόζονται σε αυτό

Για τα σύνολα υπάρχει ένας μεγάλος αριθμός πράξεων που τα αφορά. Επίσης, όπως είδαμε ήδη στην περίπτωση του μεγέθους (len( )), οι συναρτήσεις που εφαρμόζονται στις ακολουθίες συχνά εφαρμόζονται και στα σύνολα. Σε ένα βαθμό αφορούν όλες τις συλλογές απλά έχουν πάρει τα ονόματά τους από τις ακολουθίες. Λέμε μήκος και εννοούμε μέγεθος, μεγαλύτερο και εννοούμε επόμενο στην κατάταξη (συχνά αλφαβητική) και ούτω καθεξής. Τέλος υπάρχει ένας μεγάλος αριθμός δημόσιων μεθόδων των συνόλων που κάποιες από αυτές αποτελούν εναλλακτικές υλοποιήσεις των πράξεων μεταξύ συνόλων, κάποιες εξυπηρετούν τη διαχείριση των στοιχείων τους, δημιουργούν αντίγραφα και λοιπά.

Πράξεις για σύνολα

Οι πράξεις μεταξύ συνόλων, όπως η ένωση, η τομή, οι έλεγχοι για υποσύνολα και υπερσύνολα αλλά και άλλες εκδοχές λογικών πράξεων που υλοποιούνται με σύνολα έχουν όπως είπαμε τουλάχιστον δύο τρόπους υλοποίησης: με τελεστές και με μεθόδους (της κλάσης του) συνόλου. Ας δούμε μερικά παραδείγματα της πρώτης περίπτωσης:

```
set1 = { 'pencil', 'pen', 'rubber', 'book' }
set2 = { 'diary', 'book'}
set3 = { 'pen', 'ruler'}
set3 = set1 | set2    # Η ένωση των δύο συνόλων
set4 = set1 & set2    # Η τομή των δύο συνόλων
print(set4)
{'book'}
for x in set3:
     print(x)
diary pencil pen rubber book
print(max(set1))    # Επιστρέφει τη μέγιστη αλφαριθμητικά
# τιμή στο πρώτο σύνολο
rubber
set5 = set1 - set2 - set3
print(set5)    # Το σύνολο είναι κενό οπότε θα μας το γράψει
# αντί να τυπώσει ένα κενό
set()
set6 = { 1, 'one'}
print(max(set5))    # Θα επιστρέψει ValueError αφού στη max() δώσαμε όρισμα
κενό σύνολο
print(max(set6))    # Θα επιστρέψει TypeError: αφού ανάμεσα σε
# 'str' και 'int' τύπου στοιχεία δεν προβλέπεται
# σύγκριση
```

Να σημειωθεί ότι τελεστές όπως | της ένωσης, - της διαφοράς, ^ της συμμετρικής διαφοράς και & της τομής, εφαρμόζονται και επαναληπτικά μέσα στις συνολοθεωρητικές εκφράσεις της Python.

Έλεγχος συμπερίληψης

Μπορούμε επίσης να ελέγξουμε αν ένα στοιχείο ανήκει σε ένα σύνολο ή και το αντίθετο όπως και για τις υπόλοιπες δομές που αποτελούν συλλογές στοιχείων. Αυτό γίνεται με την εφαρμογή των γνωστών τελεστών in και not in για έλεγχο συμπερίληψης ή μη αντίστοιχα. Κατά τα γνωστά, οι δύο τελεστές εφαρμόζονται ανάμεσα σε ένα πιθανό στοιχείο και ένα σύνολο, δημιουργώντας μία παράσταση που μπορεί να είναι είτε αληθής είτε ψευδής. Όπως και για κάθε άλλη συλλογή στοιχείων, μπορεί να χρησιμοποιηθεί ως συνθήκη, για παράδειγμα σε μία εντολή επιλογής ή επανάληψης (κυρίως όταν πρόκειται για in). Ας δούμε άλλο ένα παράδειγμα:

```
for x in set1
     print(x)  # Θα τυπώσει διαδοχικά 'pencil','pen','rubber','book'
```

Συγκρίσεις

Οι συγκριτικοί τελεστές μπορούν να εφαρμοστούν και στα σύνολα. Έχουν όμως μια διαφορετική σημασία – στην πραγματικότητα ελέγχουν τη σχέση των συνόλων. Συγκεκριμένα, θεωρώντας τους ορισμούς συνόλων από την παράγραφο «Πράξεις για σύνολα», ας δούμε μερικά παραδείγματα:

```
set1 == set2    # Ελέγχει αν τα δύο σύνολα έχουν ακριβώς τα
# ίδια στοιχεία
set1 != set2    # Ελέγχει αν τα δύο σύνολα έχουν τουλάχιστον
# ένα στοιχείο διαφορετικό.
set1 <= set2    # Ελέγχει αν το set1 είναι υποσύνολο του set2,
# δηλαδή αν κάθε στοιχείο του set1 περιέχεται στο set2
set1 < set2    # Ελέγχει αν το set1 είναι γνήσιο υποσύνολο
# του set2, δηλαδή αν set1 <= set2 και ταυτόχρονα set1 != set2
set1 >= set2    # Ελέγχει αν το set2 είναι υποσύνολο του set1,
# δηλαδή αν κάθε στοιχείο του set2 περιέχεται στο set1
set1 > set2    # Ελέγχει αν το set2 είναι γνήσιο υποσύνολο
# του set1, δηλαδή αν set1 >= set2 και ταυτόχρονα set1 != set2
print(set1 == set2)
False
print(set1 <= set3)
True
print(set1 > set4)
True
```

Συναρτήσεις

Υπάρχουν όπως έχουμε ήδη αναφέρει μια σειρά από συναρτήσεις που εφαρμόζονται στις ακολουθίες, όμως εφαρμόζονται επίσης και σε άλλες συλλογές. Αυτές περιλαμβάνουν την εύρεση του μεγέθους του συνόλου με τη συνάρτηση len( ) και, εφόσον οι τιμές είναι συγκρίσιμες, το μέγιστο και ελάχιστο στοιχείο με τις συναρτήσεις max( )και min( ) αντίστοιχα. Ακόμα η, επίσης

ενσωματωμένη, συνάρτηση sorted( ) επιστρέφει μια ταξινομημένη λίστα με τα ίδια στοιχεία με το σύνολό μας και τέλος η iter( ) που είδαμε νωρίτερα. Ας δούμε κι εδώ κάποια παραδείγματα:

```
set1 = {'pencil','pen','rubber','book' }
print(len(set1))    # Θα τυπώσει το μέγεθος (αριθμό στοιχείων)
# του συνόλου
4
print(max(set1))    # Θα τυπώσει τη «μεγαλύτερη» τιμή

rubber

print(min(set1))    # Θα τυπώσει τη «μικρότερη» τιμή

book
print(sorted(set1 ))    # Θα τυπώσει μια λίστα με τα στοιχεία
# του συνόλου ταξινομημένα.

['book', 'pen', 'pencil', 'rubber']
print(iter(set1 ))    # Θα μας επιστρέψει τη θέση του
# αντικειμένου που δημιούργησε η iter( ) στη μνήμη
# καθώς δεν προβλέπεται εκτύπωση για έναν iterator που
# κατασκευάστηκε έτσι. Π.χ.
<set_iterator object at 0x7f380f6d6640>
for x in iter(set1):    #
     print(x)    # Αντίθετα τυπώνει κανονικά τα επί μέρους
# στοιχεία του αντικειμένου (σε τυχαία εννοείται
# σειρά)
book rubber pen pencil
```

Μέθοδοι

Όπως έχουμε και αλλού αναφέρει τα σύνολα έχουν μια πληθώρα δημόσιων μεθόδων που κάποιες από αυτές αποτελούν εναλλακτικές υλοποιήσεις των πράξεων και συγκρίσεων μεταξύ συνόλων, κάποιες εξυπηρετούν τη διαχείριση των στοιχείων τους και φυσικά η ενσωματωμένη συνάρτηση `dir( )`, η οποία επιστρέφει μια λίστα με τα ονόματα των μεθόδων (δημόσιων και ιδιωτικών) του αντικειμένου μας.

Μέθοδοι υλοποίησης συγκρίσεων

`.isdisjoint( )` Δέχεται ως όρισμα ένα άλλο σύνολο και επιστρέφει True αν και μόνο αν το σύνολο αυτό δεν έχει κοινά στοιχεία με το αρχικό. Τα δύο σύνολα πρέπει να έχουν κενή τομή.

`.issubset( )` Δέχεται ως όρισμα ένα άλλο σύνολο και επιστρέφει True αν και μόνο αν κάθε στοιχείο στο αρχικό σύνολο ανήκει στο σύνολο όρισμα. Αποτελεί εναλλακτική υλοποίηση της σύγκρισης υποσύνολο (set1 <= set2).

`.issuperset( )` Δέχεται ως όρισμα ένα άλλο σύνολο και επιστρέφει True αν και μόνο αν κάθε στοιχείο στο σύνολο όρισμα ανήκει στο αρχικό σύνολο. Αποτελεί εναλλακτική υλοποίηση της σύγκρισης υπερσύνολο (set1 >= set2).

Μέθοδοι υλοποίησης πράξεων

`.union( )` Δέχεται ως όρισμα ένα ή περισσότερα σύνολα χωρισμένα με κόμμα (,) και επιστρέφει ένα νέο σύνολο που περιλαμβάνει όλα τα στοιχεία που ανήκουν είτε στο αρχικό σύνολο είτε σε οποιοδήποτε από τα σύνολα ορίσματα. Αποτελεί εναλλακτική υλοποίηση της ένωσης συνόλων ή αν θέλετε του λογικού ΄Η (set1 | set2 | set3 ...).

`.intersection( )` Δέχεται ως όρισμα ένα ή περισσότερα σύνολα χωρισμένα με κόμμα (,) και επιστρέφει ένα νέο σύνολο που περιλαμβάνει όλα τα στοιχεία που ανήκουν τόσο στο αρχικό σύνολο όσο και σε οποιοδήποτε από τα σύνολα ορίσματα. Αποτελεί εναλλακτική υλοποίηση της τομής συνόλων ή αλλιώς του λογικού ΚΑΙ (set1 & set2 & set3 ...).

`.difference( )` Δέχεται ως όρισμα ένα ή περισσότερα σύνολα χωρισμένα με κόμμα (,) και επιστρέφει ένα νέο σύνολο που περιλαμβάνει όλα τα στοιχεία που ανήκουν στο αρχικό σύνολο αλλά όχι σε κάποιο από τα σύνολα ορίσματα. Αποτελεί εναλλακτική υλοποίηση της set1 – set2 – set3 ...

`.symmetric_difference( )` Δέχεται ως όρισμα ένα σύνολο και επιστρέφει ένα νέο σύνολο που περιλαμβάνει όλα τα στοιχεία που ανήκουν είτε στο αρχικό σύνολο είτε στο σύνολο όρισμα αλλά όχι κα στα δύο. Αποτελεί εναλλακτική υλοποίηση της set1 ^ set2.

`.update( )` Δέχεται ως όρισμα ένα ή περισσότερα σύνολα χωρισμένα με κόμμα (,) και ενημερώνει το αρχικό σύνολο ώστε να περιλαμβάνει όλα τα στοιχεία που ανήκουν είτε στο αρχικό σύνολο είτε σε οποιοδήποτε από τα σύνολα ορίσματα. Αποτελεί εναλλακτική υλοποίηση της ένωσης συνόλων ή αν θέλετε του λογικού ΄Η που όμως αποθηκεύει το αποτέλεσμα στο αρχικό σύνολο(set1 |= set2 |= set3 ...).

`.intersection_update( )` Δέχεται ως όρισμα ένα ή περισσότερα σύνολα χωρισμένα με κόμμα (,) και ενημερώνει το αρχικό σύνολο ώστε να περιλαμβάνει όλα τα στοιχεία που ανήκουν τόσο στο αρχικό σύνολο όσο και σε οποιοδήποτε από τα σύνολα ορίσματα. Αποτελεί εναλλακτική υλοποίηση

της τομής συνόλων ή αλλιώς του λογικού ΚΑΙ που όμως αποθηκεύει το αποτέλεσμα στο αρχικό σύνολο (set1 &= set2 &= set3 ...).

`.difference_update( )` Δέχεται ως όρισμα ένα ή περισσότερα σύνολα χωρισμένα με κόμμα (,) και ενημερώνει το αρχικό σύνολο αφαιρώντας όλα τα στοιχεία που ανήκουν και σε κάποιο από τα σύνολα ορίσματα. Αποτελεί εναλλακτική υλοποίηση της set1 –= set2 | set3 ...

`.symmetric_difference_update( )` Δέχεται ως όρισμα ένα σύνολο και ενημερώνει το αρχικό σύνολο ώστε να περιλαμβάνει όλα τα στοιχεία που ανήκουν σε οποιοδήποτε από τα δύο σύνολα αλλά όχι και στα δύο. Αποτελεί εναλλακτική υλοποίηση της set1 ^= set2.

Διαχείριση στοιχείων

`.add( )` Δέχεται ως όρισμα ένα στοιχείο και το προσθέτει στο σύνολο.

`.remove( )` Δέχεται ως όρισμα ένα στοιχείο και το απομακρύνει από το σύνολο. Εάν το στοιχείο δεν περιλαμβάνεται στο σύνολο επιστρέφει KeyError.

`.discard( )` Δέχεται ως όρισμα ένα στοιχείο και το απομακρύνει από το σύνολο αν ανήκει σε αυτό.

`.pop( )` Απομακρύνει ένα τυχαίο στοιχείο από το σύνολο. Εάν το σύνολο είναι κενό επιστρέφει KeyError.

`.clear( )` Απομακρύνει όλα τα στοιχεία από το σύνολο.

`.copy( )` Επιστρέφει ένα «ρηχό» (by value) αντίγραφο του συνόλου.

.

7.5.3 Να θυμάμαι

Σύνολα

Το σύνολο είναι μια δομή δεδομένων με τα εξής χαρακτηριστικά:

- ✓ είναι σύνθετη, δηλαδή χρησιμοποιείται για την αποθήκευση πολλαπλών δεδομένων ταυτόχρονα
- ✓ είναι αδιάτακτη, δηλαδή η σειρά των μελών της είναι αδιάφορη
- ✓ είναι τροποποιήσιμη (mutable), με την έννοια ότι μπορούμε να προσθέσουμε και να αφαιρέσουμε στοιχεία όποτε θέλουμε
- ✓ τα στοιχεία/ μέλη της, από την άλλη, είναι μη τροποποιήσιμα (immutable)
- ✓ δεν επιτρέπεται η επανάληψη στοιχείου
- ✓ τα στοιχεία της πρέπει να είναι συγκρίσιμα
- ✓ τα στοιχεία/ μέλη της μπορούν να είναι ετερογενή

Τρόποι δημιουργίας συνόλων

Με απαρίθμηση των στοιχείων του:

```
set1 = {1, 2, 3, 4, 5, 6}
```

Με εφαρμογή της συνάρτησης σε κατάλληλη ακολουθία (όχι λεξικό στο σύνολό του):

```
set2 = set([10, 20, 30])
```

Με αρχική δημιουργία υποσυνόλου και διαδοχικές προσθήκες των μελών/ στοιχείων του (π.χ. με τη μέθοδο `.add( )`:

```
set3 = {5}
set3.add(10)
set3.add(15)
```

Προσοχή! Δεν μπορεί να είναι το κενό γιατί θα το θεωρήσει λεξικό

Πράξεις ανάμεσα σε σύνολα

Στα σύνολα εφαρμόζονται οι κλασσικές συνολοθεωρητικές πράξεις με τελεστές όπως `|` της ένωσης, `-` της διαφοράς, `^` της συμμετρικής διαφοράς και `&` της τομής, δεν εφαρμόζονται σε αυτά η συνένωση (concatenation) και ο πολλαπλασιασμός, για προφανείς λόγους.

Έλεγχος συμπερίληψης

Μπορούμε να ελέγξουμε αν ένα στοιχείο ανήκει σε ένα σύνολο ή και το αντίθετο. Αυτό γίνεται με την εφαρμογή των τελεστών `in` και `not in` για έλεγχο συμπερίληψης ή μη αντίστοιχα.

Συγκρίσεις

Και οι συγκριτικοί τελεστές μπορούν να εφαρμοστούν στα σύνολα, αν και έχουν διαφορετική σημασία:

```
set1 == set2    # Ελέγχει αν τα δύο σύνολα έχουν ακριβώς τα  ίδια στοιχεία
set1 != set2    # Ελέγχει αν έχουν τουλάχιστον ένα στοιχείο διαφορετικό.
set1 <= set2    # Ελέγχει αν το set1 είναι υποσύνολο του set2.
set1 < set2    # Ελέγχει αν το set1 είναι γνήσιο υποσύνολο του set2
set1 >= set2    # Ελέγχει αν το set2 είναι υποσύνολο του set1.
set1 > set2    # Ελέγχει αν το set2 είναι γνήσιο υποσύνολο του set1.
```

Συναρτήσεις

Κάποιες συναρτήσεις που αφορούν στις ακολουθίες εφαρμόζονται με διασταλτικό τρόπο και σε άλλες συλλογές όπως τα σύνολα. Εξυπηρετείται έτσι η εύρεση του μεγέθους του συνόλου (το πλήθος των στοιχείων του) από τη συνάρτηση `len( )` και, εφόσον οι τιμές είναι συγκρίσιμες, το μέγιστο και ελάχιστο στοιχείο με τις συναρτήσεις `max( )`και `min( )` αντίστοιχα. Ακόμα η, επίσης ενσωματωμένη, συνάρτηση `sorted( )` επιστρέφει μια ταξινομημένη λίστα με τα ίδια στοιχεία με το σύνολό μας και τέλος η `iter( )` επιτρέπει τη δημιουργία ενός iterator τόσο για τη διάσχιση του συνόλου μας όσο και για άλλες χρήσεις.

Μέθοδοι

Ένα σύνολο έχει τις περισσότερες δημόσιες μεθόδους από όλους τους ενσωματωμένους σύνθετους τύπους. Και αυτό γιατί περιλαμβάνει πολλαπλές κατηγορίες όπως:

Μεθόδους υλοποίησης συγκρίσεων,

Μεθόδους υλοποίησης πράξεων,

Μεθόδους διαχείριση στοιχείων.

7.5.4 Ερωτήσεις Κατανόησης

1. Το σύνολο στην Python είναι μια σύνθετη δομή δεδομένων.

   ΝΑΙ ΟΧΙ

2. Το σύνολο στην Python είναι διατεταγμένο.

   ΝΑΙ ΟΧΙ

3. Υπάρχει μόνο ένας τρόπος εκτέλεσης συνολοθεωρητικών πράξεων στην Python.

   ΝΑΙ ΟΧΙ

4. Σε σύνολα αποθηκεύουμε υποχρεωτικά ομοειδή στοιχεία.

   ΝΑΙ ΟΧΙ

5. Η γλώσσα Python διαθέτει συναρτήσεις που εφαρμόζονται εκτός από τις ακολουθίες και σε άλλες συλλογές, όπως οι . `len( )`, `max( )` και `min( )`

   ΝΑΙ ΟΧΙ

6. Οι συγκρίσεις δεν εφαρμόζονται στα σύνολα.

   ΝΑΙ ΟΧΙ

7.5.5 Ερωτήσεις Ανάπτυξης

1. Ποια είναι τα χαρακτηριστικά του συνόλου στην Python;
2. Με ποιους τρόπους κατασκευάζουμε σύνολα στην Python;
3. Μπορούν οι συνολοθεωρητικές πράξεις να χρησιμοποιηθούν για το σκοπό αυτό;
4. Με ποιο τρόπο μπορούμε να δούμε όλα τα στοιχεία ενός συνόλου;
5. Προτείνετε τέσσερις τρόπους για να προσθέσουμε στοιχεία σε ένα σύνολο.
6. Προτείνετε τρεις τρόπους για να απομακρύνουμε ένα στοιχείο από ένα σύνολο

7.5.6 Ασκήσεις

1. Δεδομένου ότι κάθε φορά που χρησιμοποιώ ένα σύνολο η σειρά των αριθμών είναι τυχαία, αν έχω ένα σύνολο set = {1,2,3,4,5,6} κάθε εκτύπωσή π.χ. θα μας δίνει πρώτο έναν διαφορετικό αριθμό. Άρα ο επόμενος κώδικας θεωρητικά προσομοιώνει τη ρήψη ενός ζαριού:

```
set = {1,2,3,4,5,6}

zari = list(set)

print(zari[0])
```

Αν αντί για print σώσουμε την κάθε τιμή σε μια θέση μιας νέας λίστας num μπορούμε να φτιάξουμε μια επανάληψη π.χ. 1000 ή 10000 επαναλήψεων και τυπώνοντας, όταν τελειώσει, τη συχνότητα εμφάνισης κάθε αριθμού στο set στη νέα μας λίστα μπορούμε να ελέγξουμε πόσο καλά προσομοιώνει την τυχαιότητα. Για τη συχνότητα μπορούμε να διαιρέσουμε το .count( ) του στοιχείου με τον αριθμό των επαναλήψεων.

Υλοποιείστε και ελέγξτε.

2. Αν θελήσουμε να προσομοιώσουμε με παρόμοιο τρόπο τη λειτουργία του ΛΟΤΤΟ πρέπει να αντιμετωπίσουμε το πρόβλημα ότι τα 6 νούμερα που θα επιλέξει το πρόγραμμά μας πρέπει να είναι διαφορετικά. Σκέψου έναν τρόπο να το παρακάμψουμε και φτιάξε ένα πρόγραμμα που όσο του ζητάμε θα μας επιστρέφει εξάδες.