

ΚΕΦΑΛΑΙΟ 6

Συμβολοσειρές, λίστες, πλειάδες, λεξικά

Σύνοψη

Στο κεφάλαιο αυτό θα εμβαθύνουμε στην κατανόηση σημαντικών δομών δεδομένων, όπως οι συμβολοσειρές (*strings*), οι λίστες (*lists*), οι πλειάδες (*tuples*) και τα λεξικά (*dictionaries*), καθώς και της διαχείρισής τους.

Προαπαιτούμενη γνώση

Κεφάλαια 1-5 του παρόντος συγγράμματος.

6.1 Εισαγωγή

Σε αυτό το κεφάλαιο θα εμβαθύνουμε στην κατανόηση σημαντικών δομών δεδομένων, όπως οι συμβολοσειρές (*strings*), οι λίστες (*lists*), οι πλειάδες (*tuples*) και τα λεξικά (*dictionaries*), και της διαχείρισής τους. Ο σκοπός του κεφαλαίου είναι να εξοικειώσει τον αναγνώστη με τη χρήση συλλογών (*collections*) αντικειμένων οι οποίες αποτελούν βασικά εργαλεία τους στην διαχείριση δεδομένων.

6.2 Συμβολοσειρές

Μια συμβολοσειρά είναι μια ακολουθία από χαρακτήρες. Οι συμβολοσειρές ανήκουν στον τύπο δεδομένων `str`. Μπορείτε να ορίσετε συμβολοσειρές με μονά (') ή διπλά (") εισαγωγικά:

```
>>> s1='Hello'
>>> s2="Hello"
>>> print(s1)
Hello
>>> print(s2)
Hello
>>> type('Hello')
<type 'str'>
```

Εικόνα 6.1 Ορισμός συμβολοσειρών με μονά ή διπλά εισαγωγικά

Τα μονά εισαγωγικά (') μπορείτε να τα χρησιμοποιείτε ελεύθερα μέσα σε διπλά εισαγωγικά ("), όπως στην Εικόνα 6.2:

```
>>> print("Let's go")
Let's go
```

Εικόνα 6.2 Χρήση μονών εισαγωγικών μέσα σε διπλά εισαγωγικά

Αν θέλουμε να χρησιμοποιήσουμε το μονό εισαγωγικό σαν εκτυπώσιμο χαρακτήρα εντός μιας συμβολοσειράς, θα ακολουθήσουμε την ακολουθία διαφυγής (escape sequence) \' (ο χαρακτήρας \ ακολουθούμενος από το μονό εισαγωγικό). Αντίστοιχα, χρησιμοποιούμε ακολουθίες διαφυγής για την εισαγωγή νέας γραμμής (new line, \n) ή το tab (\t). Παραδείγματα ακολουθιών διαφυγής παρουσιάζονται στην Εικόνα 6.3.

```
>>> print('Hello\nWorld!')
Hello
World!
>>> print('Let\'s go')
Let's go
>>> print('Hello\tWorld!')
Hello   World!
```

Εικόνα 6.3 Παραδείγματα ακολουθιών διαφυγής

Μπορείτε να ορίσετε συμβολοσειρές πολλαπλών γραμμών με τριπλά εισαγωγικά (τρία διπλά “” ή τρία μονά “”), όπως στην Εικόνα 6.4. Βλέπετε ότι με τα τριπλά εισαγωγικά μπορούμε να αλλάξουμε και γραμμή, χωρίς να χρειάζεται να βάλουμε την ακολουθία διαφυγής slash n (\n):

```
>>> s='''Hello  
world!'''  
  
>>> print(s)  
Hello  
world!
```

Εικόνα 6.4 Συμβολοσειρές πολλαπλών γραμμών

Μπορείτε να χρησιμοποιείτε ελεύθερα μονά και διπλά εισαγωγικά μέσα σε τριπλά εισαγωγικά.

6.2.1 Ο σύνθετος τύπος δεδομένων str

Σε αντίθεση με τους τύπους int και float που έχουμε δει ως τώρα, οι οποίοι δεν αναλύονται σε απλούστερους τύπους, ο τύπος str αποτελείται από μικρότερα συστατικά στοιχεία, τους χαρακτήρες. Ένας τέτοιος τύπος ονομάζεται **σύνθετος**. Ανάλογα με το τι θέλουμε να κάνουμε, μεταχειριζόμαστε ένα σύνθετο τύπο είτε σαν ένα ενιαίο σύνολο είτε σαν αποτελούμενο από διακριτά μέρη.

Μια συμβολοσειρά μπορεί να περιέχει και αριθμούς (π.χ. ‘ctr2’), η Python ωστόσο καταλαβαίνει όλα τα σύμβολα εντός της συμβολοσειράς σαν χαρακτήρες, όχι σαν ακεραίους ή πραγματικούς. Μάλιστα η Python αντιστοιχεί τους χαρακτήρες στη συγκεκριμένη σειρά με την οποία εμφανίζονται. Η σειρά έχει σημασία, και γι’ αυτό και η Python μας προσφέρει έναν τελεστή, τις τετράγωνες παρενθέσεις ([], Εικόνα 6.5), με τον οποίο μπορούμε να επιλέγουμε κάποιον χαρακτήρα μέσα στη συμβολοσειρά. Για παράδειγμα στην Εικόνα 6.5) δίνουμε στη μεταβλητή fruit την τιμή ‘banana’ και με την έκφραση fruit[1] επιλέγουμε έναν από τους χαρακτήρες.

```
>>> fruit='banana'
>>> letter=fruit[1]
>>> print(letter)
```

Εικόνα 6.5 Ο τελεστής []

Η έκφραση `fruit[1]` διαλέγει το χαρακτήρα στη θέση 1 του `fruit`. Το αποτέλεσμα είναι έκπληξη:

```
>>> print(letter)
a
```

Εικόνα 6.6 Εκτύπωση χαρακτήρα

Το πρώτο γράμμα, βέβαια, του `'banana'` δεν είναι `a` αλλά `b`. Όπως και σε άλλες γλώσσες προγραμματισμού έτσι και στην Python, σκεφτόμαστε την έκφραση `fruit[1]` σαν μετάθεση (`offset`) από την αρχή της συμβολοσειράς, οπότε η μετάθεση του πρώτου γράμματος είναι μηδέν. Άρα, το `b` είναι το μηδενικό γράμμα της `'banana'`, το `a` είναι το πρώτο, το `n` είναι το δεύτερο, ..., κλπ.

Η έκφραση μέσα στις αγκύλες `[]` λέγεται και δείκτης (`index`). Ένας δείκτης αναφέρεται σε μέλος ενός διατεταγμένου συνόλου, στη δική μας περίπτωση του συνόλου χαρακτήρων μιας συμβολοσειράς. Δείκτης μπορεί να είναι κάθε ακέραια έκφραση. Ο λόγος για τον οποίο ο δείκτης εντός της συμβολοσειράς ξεκινά από το 0 είναι γιατί η συμβολοσειρά αποθηκεύεται στη μνήμη σειριακά και ο τελεστής δείχνει σε ποια μετάθεση μέσα στη μνήμη βρίσκεται ο συγκεκριμένος χαρακτήρας. Για τον πρώτο χαρακτήρα δε θα πρέπει να μετατοπιστεί καθόλου ο δείκτης, άρα είναι 0, για το δεύτερο χαρακτήρα θα μετατοπιστεί κατά μία θέση, κ.ο.κ. Άρα το `b` είναι το μηδενικό γράμμα της `'banana'`, `a` το πρώτο, `n` το δεύτερο κ.ο.κ.:

```
>>> letter=fruit[0]
>>> print(letter)
b
```

Εικόνα 6.7 Εκτύπωση χαρακτήρα

6.2.2 Συνάρτηση len και δείκτες συμβολοσειρών

Η ενσωματωμένη συνάρτηση *len* της Python μάς επιστρέφει τον αριθμό των χαρακτήρων σε μια συμβολοσειρά (μήκος). Στην Εικόνα 6.8 βλέπουμε ότι το μήκος της συμβολοσειράς *fruit* είναι 6:

```
>>> fruit='banana'
>>> print(len(fruit))
6
```

Εικόνα 6.8 Η συνάρτηση *len*

Ωστόσο, σημειώστε ότι για να επιλέξουμε τον τελευταίο χαρακτήρα σε μια συμβολοσειρά είναι λάθος να πούμε:

```
>>> length=len(fruit)
>>> last_letter=fruit[length]

Traceback (most recent call last):
  File "<pyshe11#19>", line 1, in <module>
    last_letter=fruit[length]
IndexError: string index out of range
```

Εικόνα 6.9 Λάθος τρόπος επιλογής χαρακτήρα συμβολοσειράς

Ο λόγος είναι ότι δεν υπάρχει 6ος χαρακτήρας στο 'banana', γιατί αρχίσαμε να μετράμε από το μηδέν και άρα ο τελευταίος χαρακτήρας είναι στη θέση 5. Με την έκφραση *fruit[length]* στοχεύουμε εκτός του εύρους της συμβολοσειράς. Μπορούμε να αναφερθούμε στον τελευταίο χαρακτήρα, αν αφαιρέσουμε μία μονάδα από το μήκος, όπως στην Εικόνα 6.10:

```
>>> length=len(fruit)
>>> last_letter=fruit[length-1]
>>> print(last_letter)
a
```

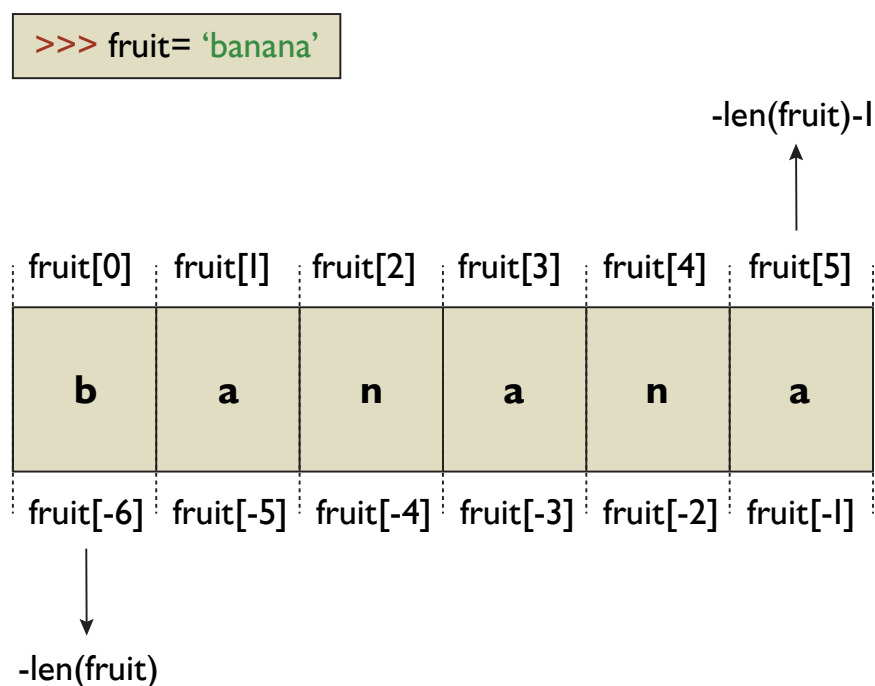
Εικόνα 6.10 Σωστός τρόπος επιλογής χαρακτήρα συμβολοσειράς

Η Python μάς δίνει τη δυνατότητα να χρησιμοποιήσουμε *αρνητικούς* δείκτες. Για παράδειγμα, *fruit[-1]* είναι ο τελευταίος χαρακτήρας του *fruit*, *fruit[-2]* είναι ο προ-

τελευταίος, κ.ο.κ. (Εικόνα 6.11). Άρα, όταν έχουμε θετικούς δείκτες κινούμαστε από τα αριστερά προς τα δεξιά, ενώ όταν έχουμε αρνητικούς δείκτες κινούμαστε από τον τελευταίο χαρακτήρα που είναι ο -1 προς την αρχή που θα είναι ο -6. Είναι, λοιπόν, το ίδιο πράγμα να πούμε fruit[5] με το να πούμε fruit[-1]. Το -1 διαλέγει πάντα τον τελευταίο χαρακτήρα της συμβολοσειράς και μετά κάνουμε μεταθέσεις προς τα πίσω. Στην Εικόνα 6.12 βλέπετε αυτές τις έννοιες σχηματικά.

```
>>> print(fruit[-1])
a
>>> print(fruit[-2])
n
>>> print(fruit[-length])
b
```

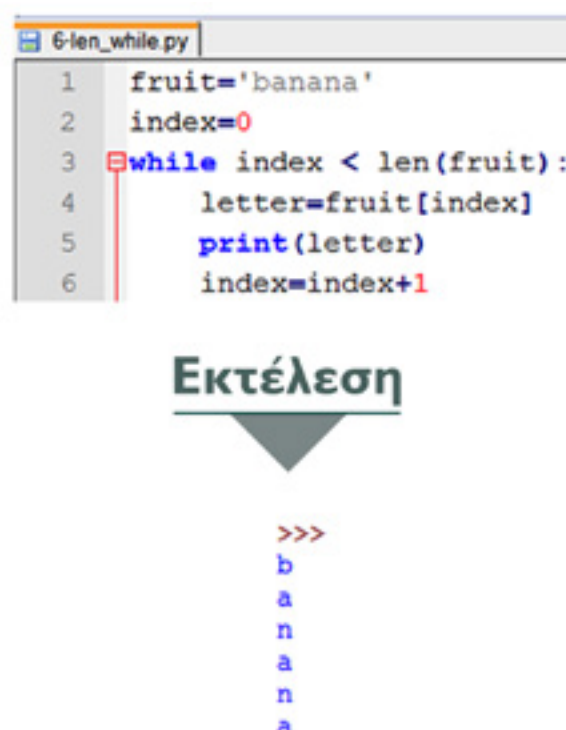
Εικόνα 6.11 Χρήση αρνητικών δεικτών για επιλογή χαρακτήρα



Εικόνα 6.12 Δείκτες συμβολοσειράς

6.2.3 Πέρασμα συμβολοσειράς και βρόχος while

Μια συχνή ανάγκη σε πολλά προγράμματα Python είναι η επεξεργασία συμβολοσειρών, χαρακτήρα-χαρακτήρα. Πολλές φορές ξεκινάνε από την αρχή, επιλέγουν ένα χαρακτήρα τη φορά, κάνουν κάποιον υπολογισμό με αυτόν, μετά πηγαίνουν στον επόμενο και συνεχίζουν μέχρι το τέλος. Αυτός ο τρόπος (pattern) υπολογισμού λέγεται πέρασμα (traversal). Ένας τρόπος να γράψουμε πρόγραμμα για το πέρασμα μιας συμβολοσειράς είναι με χρήση της εντολής while:



Εικόνα 6.13 Πέρασμα μιας συμβολοσειράς με χρήση της εντολής while

Ο βρόχος while διατρέχει τη συμβολοσειρά fruit και δείχνει ένα χαρακτήρα σε διαφορετική γραμμή κάθε φορά. Η συνθήκη του βρόχου είναι `index < len(fruit)` και επομένως, όταν το `index` γίνει ίσο με το μήκος της συμβολοσειράς, η συνθήκη γίνεται ψευδής και το σώμα του βρόχου δεν εκτελείται. Ο τελευταίος χαρακτήρας που τυπώνεται, έχει δείκτη `len(fruit)-1`, και αυτός είναι ο τελευταίος χαρακτήρας της συμβολοσειράς.

6.2.4 Πέρασμα συμβολοσειράς και βρόχος for

Η χρήση δείκτη για το πέρασμα ενός συνόλου τιμών είναι τόσο κοινή στον προγραμματισμό, ώστε η Python προσφέρει μια εναλλακτική σύνταξη, το βρόγχο for, έναν διαφορετικό (και πιο συνοπτικό) τρόπο από το while για να εκτελεστεί η ίδια λειτουργία.

Στην Εικόνα 6.14 βλέπουμε ένα παράδειγμα περάσματος συμβολοσειράς με χρήση της for. Προσέξτε ότι δε χρειάζεται να αρχικοποιήσουμε τη μεταβλητή char. Σύμφωνα με τον κανόνα αν δεν αρχικοποιήσουμε, τότε αρχίζουμε πάντα από το 0. Για τη μεταβλητή char, το in σημαίνει ότι «παίρνει τιμές» και λειτουργεί σαν δείκτης μέσα στη μεταβλητή fruit. Άρα η τιμή της fruit πρέπει να εκφράζει κάτι που να έχει στη σειρά κάποια αντικείμενα, ώστε να μπορεί ο δείκτης να έχει νόημα, να παίρνει τιμές 0,1,2,3 κλπ. Στην προκειμένη περίπτωση εφαρμόζουμε το for ... in στις συμβολοσειρές, αλλά θα δούμε και άλλους σύνθετους τύπους στους οποίους μπορεί να εφαρμοστεί αυτή η σύνταξη.

Δύο είναι τα απαραίτητα πράγματα που πρέπει να έχει η εντολή βρόγχου for. Τη διατρέχουσα μεταβλητή (εδώ η char), δηλαδή, αυτή που διατρέχει κάτι, και τη διατρεχόμενη μεταβλητή (εδώ fruit), δηλαδή αυτή μέσα στην οποία η διατρέχουσα θα διατρέξει και θα πάρει τιμές. Στην απλούστερη περίπτωση, ο βρόχος αρχίζει από το 0 και φτάνει μέχρι το μήκος της διατρεχόμενης μεταβλητής -1.

Οι ειδικές λέξεις της σύνταξης for (το for και το in) είναι δεσμευμένα ονόματα και δεν μπορούν να χρησιμοποιηθούν σαν ονόματα μεταβλητών, όπως αναφέραμε στο Κεφάλαιο 2. Η διατρέχουσα μεταβλητή δεν παίρνει τιμές 0,1,2,3 (αυτές είναι οι τιμές του δείκτη στο fruit, ο οποίος δείκτης υπονοείται εδώ), αλλά παίρνει τις τιμές των αντικειμένων τα οποία βρίσκονται στη σειρά της διατρεχόμενης μεταβλητής εν προκειμένω των χαρακτήρων, άρα πρώτα b, μετά a, μετά n, κ.ο.κ. (Εικόνα 6.14). Είναι σημαντικό ο αναγνώστης να εκτιμήσει εδώ τη συνοπτικότητα αυτής της έκφρασης σε αντιπαράθεση με την πιο κλασική σύνταξη for i=1:10 κλπ. Ο βρόχος συνεχίζει, μέχρι να τελειώσουν οι χαρακτήρες.



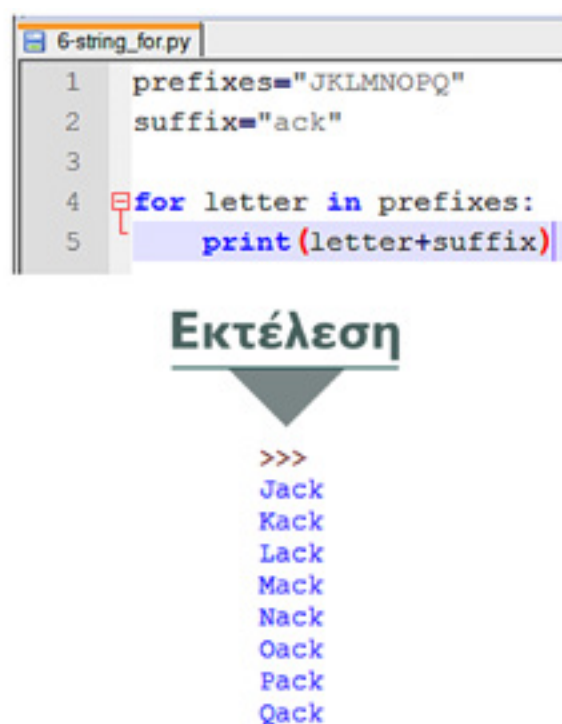
Εικόνα 6.14 Πέρασμα μιας συμβολοσειράς με χρήση της εντολής *for*

Στην Εικόνα 6.15 βλέπουμε μια άλλη εκδοχή του *for loop* στην οποία χρησιμοποιείται η έννοια της αλληλουχίας (ή concatenation) συμβολοσειρών. Αυτό σημαίνει ότι ενώνουμε μια συμβολοσειρά με μια άλλη, βάζοντας στο τέλος της πρώτης συμβολοσειράς τον 1ο χαρακτήρα της 2ης συμβολοσειράς, μετά τον 2ο, κλπ. και παράγουμε μια νέα συμβολοσειρά. Αυτό εκφράζεται με τον τελεστή του αθροίσματος '+', ο οποίος έχει διαφορετικό νόημα όταν τον χρησιμοποιούμε για ακεραίους και πραγματικούς (τότε είναι η κλασική μας πρόσθεση) και διαφορετικό, όταν τον χρησιμοποιούμε στις συμβολοσειρές (αλληλουχία).

Προσέξτε ότι οι ιδιότητες του '+' στις συμβολοσειρές, επίσης, διαφέρουν μεταξύ αριθμών και συμβολοσειρών, π.χ. η αντιμετάθεση ισχύει για πραγματικούς και ακεραίους (το $2+3$ είναι το ίδιο πράγμα με το $3+2$ ή το $2.1+4.3$ είναι το ίδιο με το $4.3+2.1$), αλλά δεν ισχύει για τις συμβολοσειρές, επειδή η σειρά σε αυτές έχει σημασία.

Στην Εικόνα 6.15 η διατρέχουσα μεταβλητή είναι η *letter* και η διατρεχόμενη μεταβλητή είναι η συμβολοσειρά *prefixes*. Η διατρέχουσα μεταβλητή, λοιπόν, παίρνει τιμές πρώτα J, μετά K, μετά L, μετά M, κλπ. μέχρι το Q. Την πρώτη φορά το *letter*

παίρνει την τιμή J, βάζουμε και προσθέτουμε και την τιμή “ack”, επομένως αυτό που τυπώνουμε είναι: Jack, Kack, Lack κλπ.



Εικόνα 6.15 Πέρασμα μιας συμβολοσειράς με χρήση της εντολής *for*

Στη συνέχεια θα δούμε μια επέκταση του τελεστή επιλογής (`[]`), για να επιλέγουμε μια σειρά από χαρακτήρες (μια υπό-συμβολοσειρά στη θέση της συμβολοσειράς) αντί του ενός χαρακτήρα. Μια τέτοια υπο-συμβολοσειρά θα την αναφέρουμε και ως «φέτα» (slice).

6.2.5 Φέτα συμβολοσειράς

Παρόμοια με την επιλογή χαρακτήρα, η επιλογή φέτας συμβολοσειράς λειτουργεί όπως φαίνεται στο παράδειγμα της Εικόνας 6.16. Εδώ βλέπετε το παράδειγμα της συμβολοσειράς “Peter, Paul and Mary”, η οποία είναι η τιμή της μεταβλητής *s*. Εάν ζητήσουμε να τυπώσει ο διερμηνέας το περιεχόμενο της συμβολοσειράς από 0 έως 5, τυπώνει Peter, δηλαδή, μετράει από το δείκτη 0 (το P του Peter) ως και το 4^ο δηλαδή το r, αλλά δε φτάνει ως τον 5^ο χαρακτήρα, το κόμμα. Το διάστημα που ορίζουμε, λοιπόν, στην έκφραση 0:5 είναι (όπως λέμε στα μαθηματικά) κλειστό από την κάτω μεριά (συμπεριλαμβάνεται το κάτω άκρο), αλλά ανοιχτό από την πάνω μεριά (δηλαδή, δε συμπεριλαμβάνεται το άνω άκρο). Άρα, όταν λέμε θέλου-

με να πάρουμε τη συμβολοσειρά στο διάστημα από 0 ως 5 στην Python, εννοούμε 0,1,2,3,4 κι όχι 5 κατά αντιστοιχία της μετάθεσης. Σημειώστε ότι ως χαρακτήρας μετράει και το κενό, και η αλλαγή γραμμής, και το tab, και αντίστοιχα όλοι οι χαρακτήρες ελέγχου. Όταν ζητάμε να τυπώσουμε από [7:11], εννοούμε τους δείκτες 7,8,9,10 και όχι το 11, άρα τη «φέτα» “Paul”. Αντίστοιχα, όταν ζητάμε να τυπώσουμε από [17:21], τότε εννοούμε 17,18,19,20 και όχι 21, άρα “Mary”.

```
>>> s="Peter, Paul, and Mary"
>>> print(s[0:5])
Peter
>>> print(s[7:11])
Paul
>>> print(s[17:21])
Mary
```

Εικόνα 6.16 Φέτα συμβολοσειράς

Ο τελεστής [n:m] επιστρέφει το τμήμα της συμβολοσειράς από το n-στό χαρακτήρα μέχρι το m-στό χαρακτήρα, περιλαμβάνοντας τον πρώτο αλλά αποκλείοντας τον τελευταίο.

s	P	e	t	e	r	,		P	a	u	l	,		a	n	d		M	a	r	y
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Εικόνα 6.17 Το index της συμβολοσειράς.

Εάν παραλείψετε τον πρώτο δείκτη, τότε η φέτα ξεκινάει από την αρχή της συμβολοσειράς, αν παραλείψετε το δεύτερο δείκτη, τότε η φέτα φτάνει μέχρι το τέλος της συμβολοσειράς. Έτσι:

```
>>> fruit="banana"
>>> print(fruit[:3])
ban
>>> print(fruit[3:])
ana
```

Εικόνα 6.18 Παράλειψη ενός από τους δύο δείκτες σε φέτα συμβολοσειράς

6.2.6 Συγκρίσεις μεταξύ συμβολοσειρών

Μπορούμε, επίσης, να χρησιμοποιήσουμε τελεστές σύγκρισης πάνω σε συμβολοσειρές (ή άλλες ομάδες δεδομένων), π.χ. μπορούμε να συγκρίνουμε τη μεταβλητή `word` με την ομάδα χαρακτήρων `banana`, δηλαδή, είναι η τιμή της λέξης ‘`banana`’ (Εικόνα 6.19). Ωστόσο τι σημαίνει ότι μια συμβολοσειρά είναι μεγαλύτερη από μια άλλη συμβολοσειρά ή μικρότερη από μια άλλη συμβολοσειρά; Σημαίνει ότι στη σειρά κατάταξης των γραμμάτων χαρακτήρων της, αλφαριθμηκά η πρώτη συμβολοσειρά εμφανίζεται πριν από τη δεύτερη συμβολοσειρά, δηλαδή, αν τα βάλουμε όλα αυτά σε ένα λεξικό, πρώτα θα δούμε την πρώτη συμβολοσειρά και μετά θα βάλουμε την επόμενη συμβολοσειρά. Σημειώστε ότι σε μια τέτοια σύγκριση τα κεφαλαία γράμματα προηγούνται από τα μικρά. Το “Give” λοιπόν προηγείται από το “banana” γιατί το G είναι κεφαλαίο, ενώ το b μικρό. Αντίστοιχα το “Banana” προηγείται από το “banana”. Επίσης, τα νούμερα προηγούνται των γραμμάτων.



Εικόνα 6.19 Χρήση τελεστών σύγκρισης για συμβολοσειρές

Εφόσον, λοιπόν, η Python θεωρεί ότι όλες οι λέξεις με κεφαλαία προηγούνται όλων των λέξεων με μικρά γράμματα, μπορούμε να συγκρίνουμε συμβολοσειρές ανεξάρτητα από αυτό μετατρέποντάς τις στην ίδια μορφή, παραδείγματος χάριν σε μικρά γράμματα, πριν κάνουμε συγκρίσεις.

6.2.7 Οι συμβολοσειρές είναι αμετάτρεπτες (immutable)

Είναι ακόμη σημαντικό να τονιστεί ότι κάποιοι τύποι δεδομένων είναι αμετάτρεπτοι (immutable), ενώ κάποιοι άλλοι μετατρέψιμοι (mutable). Για παράδειγμα, δεν μπορούμε να πάρουμε μια μεταβλητή που έχει σαν τιμή ομάδα χαρακτήρων και να αλλάξουμε τον 3ο χαρακτήρα από a σε b. Αυτό δεν μπορεί να γίνει στις συμβολοσειρές, ωστόσο επιτρέπεται σε άλλους τύπους, όπως οι λίστες. Έτσι δεν μπορούμε να χρησιμοποιήσουμε τον τελεστή [] στην αριστερή πλευρά μιας εκχώρησης (π.χ. να πάρουμε τον πρώτο χαρακτήρα της μεταβλητής 'Hello, world!', δηλαδή το H, και να ορίσουμε από H να είναι J, όπως στην Εικόνα 6.20), με σκοπό να αλλάξουμε ένα χαρακτήρα σε μια συμβολοσειρά. Στην περίπτωση αυτή η Python δε θα μας αφήσει (δείτε το μήνυμα λάθους στην Εικόνα 6.20) γιατί οι συμβολοσειρές είναι αμετάβλητες.

```
>>> greeting='Hello, world!'
>>> greeting[0]='J'

Traceback (most recent call last):
  File "<pyshell#34>", line 1, in <module>
    greeting[0]='J'
TypeError: 'str' object does not support item assignment
```

Εικόνα 6.20 Οι συμβολοσειρές είναι αμετάβλητες

Μπορούμε, βέβαια, να υλοποιήσουμε αυτήν την αλλαγή με άλλο τρόπο, αν ορίσουμε μια νέα μεταβλητή new_greeting, η οποία έχει σαν πρώτο γράμμα το J ακολουθούμενο από τα υπόλοιπα της greeting από το δεύτερο χαρακτήρα μέχρι το τέλος (Εικόνα 6.21).

```
>>> greeting='Hello, world!'
>>> new_greeting='J'+greeting[1:]
>>> print(new_greeting)
Jello, world!
```

Εικόνα 6.21 Δημιουργία νέας συμβολοσειράς

Στη συνέχεια θα δούμε ένα παράδειγμα μιας συνάρτησης της οποίας η λειτουργία είναι να βρίσκει αν υπάρχει κάποιος συγκεκριμένος χαρακτήρας μέσα σε μια συμβολοσειρά.

Ένα παράδειγμα

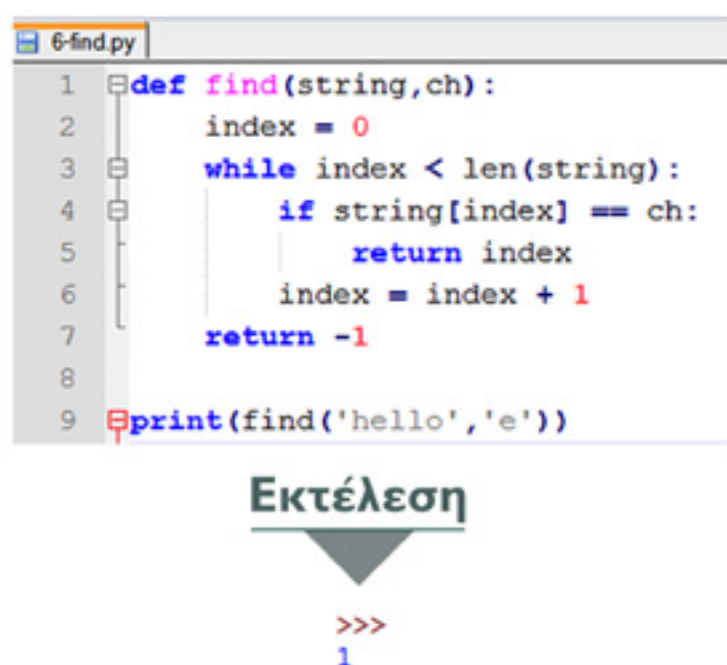
Γράψτε τον ορισμό μιας συνάρτησης με όνομα `find`, η οποία να έχει δύο παραμέτρους: μια συμβολοσειρά `string` και ένα χαρακτήρα `ch`. Η συνάρτηση αυτή να επιστρέφει τον πρώτο δείκτη της συμβολοσειράς `string` στον οποίο εμφανίζεται χαρακτήρας ίσος με τον `ch`, αλλιώς να επιστρέφει `-1`.

Στο πρόγραμμα της Εικόνας 6.22 ορίζουμε το σώμα της συνάρτησης `def find(string,ch)`. Στη συνέχεια ορίζουμε το δείκτη ο οποίος θα διατρέχει τη συμβολοσειρά `index=0`. Μετά έχουμε το `while loop` το οποίο ελέγχει κάθε φορά αν έχουμε φτάσει στο τέλος του `string` (έλεγχος με `length-1`). Αν, λοιπόν, δεν έχουμε φτάσει στο τέλος της συμβολοσειράς, ελέγχουμε στην τρέχουσα θέση αν ο χαρακτήρας είναι ίσος με το χαρακτήρα εισόδου. Εάν ισχύει αυτό, επιστρέφουμε τον τρέχοντα δείκτη. Διαφορετικά, αυξάνουμε το δείκτη κατά μία μονάδα και επιστρέφουμε πάλι πίσω για να δούμε μήπως φτάσαμε στο τέλος της συμβολοσειράς.

Αν έχουμε τελειώσει όλη τη συμβολοσειρά, μπορούμε να επιστρέψουμε `-1`. Ας αναρωτηθούμε τώρα: Υπάρχει περίπτωση το `while` να ξεπεράσει το μήκος της συμβολοσειράς; Η απάντηση είναι «όχι», γιατί ο έλεγχος του `while` μάς αναγκάζει να βγούμε από το `loop`, όταν η συνθήκη του κριθεί αληθής. Υπάρχει περίπτωση το `while loop` να συνεχίσει να εκτελείται επ' άπειρον και να μην τελειώνει ποτέ; Και αυτό δεν μπορεί να συμβεί, γιατί αυξάνουμε το `index` κάθε φορά που μπαίνουμε στο `while loop`: αν πετύχει ο έλεγχος του `while` βγαίνουμε από το `loop`, αν δεν πετύχει στη συνέχεια θα αυξήσουμε οπωσδήποτε την τιμή του `index` κατά μία μονάδα, οπότε κάθε φορά που ξαναγυρίζουμε στην κορυφή του `while loop`, το `index` θα έχει αυξηθεί τουλάχιστον κατά μία μονάδα. Άρα το `index` θα αυξάνεται, συνεχώς, καθώς προχωράει η εκτέλεση του `while loop`, και άρα αναπόφευκτα κάποια στιγμή το `index` θα γίνει ίσο με το μήκος. Αυτή είναι μια άτυπη (informal) απόδειξη, γιατί αναμένουμε ότι το πρόγραμμα θα τελειώσει κάποτε. Γενικότερα, είναι πολύ σημαντικό να προσπαθούμε η ορθότητα των προγραμμάτων μας να βασίζεται σε λογικά επιχειρήματα.

Μια εναλλακτική υλοποίηση της `find` θα μπορούσε να γίνει και με χρήση της `for` αλλά θα χρειαζόμασταν πρόσθετους μηχανισμούς για να πετύχουμε το ίδιο αποτέλεσμα. Ας έχουμε κατά νου ότι η `for` έχει μια διατρέχουσα και μια διατρεχόμενη

μεταβλητή. Αν η διατρεχόμενη είναι συμβολοσειρά, η διατρέχουσα θα διατρέχει χαρακτήρες, οπότε εγείρεται το ερώτημα: πώς θα υπολογίζουμε τη θέση; Συμπερασματικά, θα πρέπει να έχουμε και μια άλλη μεταβλητή μέσα στο `loop`, αντίστοιχη της `index`, η οποία κάθε φορά που τελειώνει μια επανάληψη του `for loop` να αυξάνεται κατά 1, με αντιστοίχιση στην τρέχουσα θέση της συμβολοσειράς. Ένας άλλος τρόπος είναι να ορίσουμε μια ισοδύναμη λίστα αριθμών, τους δείκτες της συμβολοσειράς 0,1,2,3 μέχρι το τέλος της συμβολοσειράς -1, και να διατρέξει αυτήν τη λίστα η μεταβλητή του `for`, χρησιμοποιώντας την σαν δείκτη για τη συμβολοσειρά, και κάνοντας τη σύγκριση. Ωστόσο, για τις λίστες, θα αποκτήσουμε τη γνώση στη συνέχεια σε αυτό το κεφάλαιο.



Εικόνα 6.22 Μία συνάρτηση αναζήτησης χαρακτήρα σε συμβολοσειρά

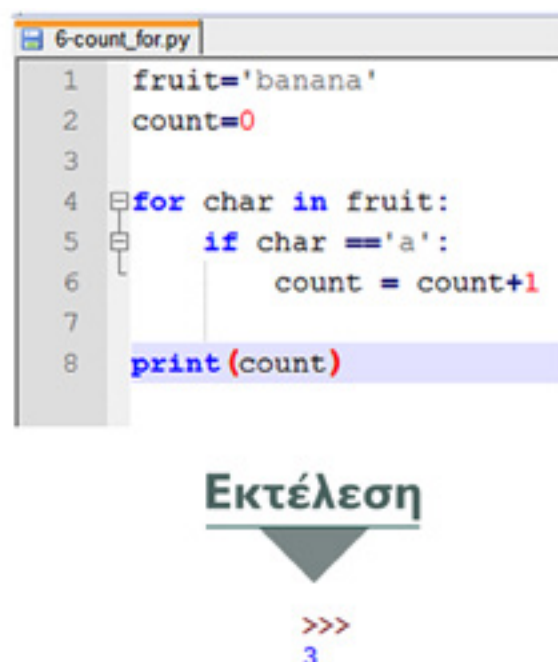
Κατά μια έννοια, η συνάρτηση `find` είναι η αντίστροφη του τελεστή `[]`. Αντί να παίρνει ως όρισμα ένα δείκτη και να επιστρέφει τον αντίστοιχο χαρακτήρα, παίρνει ένα χαρακτήρα και επιστρέφει τη θέση όπου βρίσκεται ο χαρακτήρας αυτός μέσα στη συμβολοσειρά. Αν ο χαρακτήρας δε βρεθεί, επιστρέφει -1.

Εδώ έχουμε και ένα παράδειγμα χρήσης της εντολής `return` μέσα σε βρόγχο. Αν `string[index] == ch`, τότε η συνάρτηση επιστρέφει αμέσως σταματώντας και το βρόγχο! Αν από την άλλη μεριά ο χαρακτήρας δε βρεθεί, τότε το πρόγραμμα βγαίνει από το βρόγχο και επιστρέφει -1. Αυτή η μορφή (pattern) υπολογισμού λέγεται

και *eureka traversal*, γιατί μόλις βρούμε αυτό που ψάχνουμε, φωνάζουμε όπως ο Αρχιμήδης «Εύρηκα» και σταματάμε την αναζήτησή μας.

6.2.8 Επανάληψη και μέτρηση

Το επόμενο πρόγραμμα (Εικόνα 6.23) μετράει τις φορές που ένας χαρακτήρας εμφανίζεται σε μια συμβολοσειρά χρησιμοποιώντας ένα `for loop`, υποδεικνύοντας μια άλλη μορφή υπολογισμού που λέγεται μετρητής (*counter*). Η μεταβλητή `count` ξεκινά με αρχική τιμή 0 και αυξάνεται κατά ένα, κάθε φορά που το πρόγραμμα βρίσκει το γράμμα `a`. Όταν το πρόγραμμα βγει από το βρόγχο, η `count` περιέχει το συνολικό αριθμό χαρακτήρων `'a'` στη συμβολοσειρά `'banana'`, άρα 3.



Εικόνα 6.23 Πρόγραμμα μέτρησης εμφάνισης χαρακτήρα σε συμβολοσειρά

Τέλος, θα αναφερθούμε και στον τελεστή `in`, με τον οποίο ελέγχουμε αν ένας χαρακτήρας ή μια συμβολοσειρά είναι κάπου μέσα σε μια άλλη συμβολοσειρά.

6.2.9 Ο τελεστής `in`

Το `in`, λοιπόν, είναι ένας **λογικός τελεστής** που εφαρμόζεται σε δύο συμβολοσειρές και ελέγχει αν η μία εμφανίζεται μέσα στην άλλη. Δεν είναι όπως το *find*, το

οποίο επιστρέφει τη θέση ενός χαρακτήρα, αν υπάρχει μέσα σε μια συμβολοσειρά. Ο τελεστής *in* ελέγχει αν ένας χαρακτήρας ή μια συμβολοσειρά εμφανίζεται κάπου, έστω και μια φορά σε άλλη συμβολοσειρά. Για παράδειγμα (Εικόνα 6.24), το 'a' *in* 'banana' επιστρέφει *TRUE*. Το 'na' *in* 'banana' επίσης επιστρέφει *TRUE*. Το 'w' δεν είναι μέσα στο 'banana' και άρα αυτό επιστρέφει *FALSE*.

```
>>> 'a' in 'banana'
True
>>> 'na' in 'banana'
True
>>> 'w' in 'banana'
False
```

Εικόνα 6.24 Ο λογικός τελεστής *in*

Στη συνέχεια θα αναφερθούμε σε μια πολύ σημαντική έννοια της Python, αυτή της μεθόδου. Σε αυτό το Κεφάλαιο θα κατανοήσουμε την έννοια της μεθόδου στα πλαίσια των συμβολοσειρών, στη συνέχεια του συγγράμματος, ωστόσο, θα δούμε ότι η έννοια αυτή είναι γενικότερη και κεντρική στην Python.

6.2.10 Μέθοδοι για συμβολοσειρές

Οι μέθοδοι μοιάζουν πολύ με τις συναρτήσεις (παίρνουν ορίσματα και επιστρέφουν κάποια τιμή), αλλά διαφέρουν στον τρόπο με τον οποίο συντάσσονται. Ουσιαστικά θεωρούνται ιδιοκτησία κάποιων τύπων (εν προκειμένω των συμβολοσειρών) και άρα μπορούν να χρησιμοποιηθούν μόνο στα πλαίσια αυτών των τύπων, ενώ οι γενικές συναρτήσεις δεν υπόκεινται σε αυτόν τον κανόνα.

Ένα παράδειγμα μεθόδου συμβολοσειράς είναι η *find*, (σημειώστε ότι είναι διαφορετική *find* από τη συνάρτηση με το ίδιο όνομα που ορίσαμε νωρίτερα (πρβ. Εικόνα 6.22), ωστόσο έχει παρόμοια συμπεριφορά. Για να επικαλεστούμε τη μέθοδο *find*, χρησιμοποιούμε το λεγόμενο *dot notation*, το οποίο τονίζει ότι η μέθοδος είναι ιδιοκτησία της συμβολοσειράς. Αυτή η τυπική χρήση φαίνεται στα παραδείγματα της Εικόνας 6.25. Αρχικά πρέπει να ορίσουμε ένα στιγμιότυπο (*instance*) του τύπου «συμβολοσειρά» (στην Εικόνα 6.25 το *fruit*) αναφορικά με το οποίο θα εκτελεστεί η μέθοδος. Στη συνέχεια καλούμε τη μέθοδο (π.χ. *fruit.find('a')* κλπ.).

Βλέπουμε ότι κατά τα άλλα η συμπεριφορά της `find` είναι πολύ παρόμοια με τη συνάρτηση `find` που είχαμε φτιάξει. Μια σημαντική διαφορά είναι ότι, ενώ η `find` συνάρτηση που είχαμε ορίσει, λάμβανε δύο παραμέτρους: τη συμβολοσειρά και το χαρακτήρα τον οποίο θέλαμε να βρούμε μέσα στη συμβολοσειρά, εδώ παίρνει μόνο μία παράμετρο, γιατί η συμβολοσειρά, πλέον, ορίζεται εξωτερικά και είναι το αριστερό κομμάτι του dot notation, εν προκειμένω `fruit`.

```
>>> fruit='banana'
>>> print(fruit.find('a'))
1
>>> print(fruit.find('na'))
2
>>> print(fruit.find('na', 3, 6))
4
```

Επιστροφή του πιο αριστερού δείκτη, όπου εμφανίζονται το **a** και το **na** στη **fruit**.

Επιστροφή του δείκτη στη φέτα 3:6, όπου εμφανίζεται το **na** στη **fruit**.

Εικόνα 6.25 Η μέθοδος `find`

Οι συμβολοσειρές υποστηρίζουν και μια σειρά από άλλες μεθόδους που θα δούμε παρακάτω. Για παράδειγμα, οι μέθοδοι `islower` και `isupper` μπορούν να χρησιμοποιηθούν για να ελέγξουμε αν μια συμβολοσειρά αποτελείται από μικρά ή κεφαλαία γράμματα. Η `islower` δίνει `True`, αν όλα τα γράμματα της συμβολοσειράς είναι μικρά, διαφορετικά `False`, αν έστω και ένα είναι κεφαλαίο. Το `isupper` έχει αντίστοιχη λειτουργικότητα για κεφαλαία. Αν είναι όλα κεφαλαία, τότε `True`, αν έστω και ένα είναι μικρό, τότε `False`.

```
>>> s='hello'
>>> s.islower()
True
>>> s='Hello'
>>> s.islower()
False
>>> s.isupper()
False
>>> s='HELLO'
>>> s.isupper()
True
```

Εικόνα 6.26 Οι μέθοδοι `islower` και `isupper`

Άλλες μέθοδοι που ελέγχουν κατά πόσο μια συμβολοσειρά αποτελείται από γράμματα ή ψηφία είναι οι `isalpha` και `isdigit`. Η `isalpha` επιστρέφει `True`, αν η συμβολοσειρά αποτελείται μόνον από γράμματα, διαφορετικά `False`. Η `isdigit` επιστρέφει

True, αν η συμβολοσειρά αποτελείται μόνον από ψηφία, False αν δεν αποτελείται μόνον από ψηφία (π.χ. αν υπάρχει υποδιαστολή).

```
>>> s='hello'
>>> s.isalpha()
True
>>> s='hello1'
>>> s.isalpha()
False
>>> s='1.5'
>>> s.isdigit()
False
>>> s='1209821'
>>> s.isdigit()
True
```

Εικόνα 6.27 Οι μέθοδοι *isalpha* και *isdigit*

Βεβαίως, υπάρχει και η αρίθμηση της εμφάνισης μιας συμβολοσειράς μέσα σε μια άλλη με τη μέθοδο `count`: Πόσες φορές εμφανίζεται η υπο-συμβολοσειρά 'na' στη μεγάλη συμβολοσειρά 'banana'. Μπορούμε να αναφερθούμε σε ολόκληρη τη συμβολοσειρά ή σε ένα κομμάτι της συμβολοσειράς.

```
>>> s='banana'
>>> s.count('na')
2
>>> s.count('na', 3, 6)
1
```

Εικόνα 6.28 Η μέθοδος *count*

Μπορούμε, επίσης, να ελέγξουμε αν υπάρχουν κενά διαστήματα (white spaces) στη συμβολοσειρά (Εικόνα 6.29) με τη μέθοδο `isspace`. Στο πρώτο παράδειγμα όπου η συμβολοσειρά είναι κενή, η `isspace` επιστρέφει False. Ο λόγος είναι ότι άλλο είναι το κενό διάστημα και άλλο η κενή συμβολοσειρά. Σ' αυτό το παράδειγμα έχουμε μια κενή συμβολοσειρά η οποία δεν έχει κανένα χαρακτήρα. Στο επόμενο παράδειγμα έχουμε μια συμβολοσειρά η οποία δεν είναι κενή, απλώς έχει έναν κενό χαρακτήρα: το κενό διάστημα, και επομένως η απάντηση είναι True. Στη συνέχεια και το \n θεωρείται κενό διάστημα, όπως και το \t, άρα η `isspace` επιστρέφει True και στις δύο περιπτώσεις.

```

>>> s=' '
>>> s.isspace()
False
>>> s='  '
>>> s.isspace()
True
>>> s='\n'
>>> s.isspace()
True
>>> s='\t'
>>> s.isspace()
True

```

Εικόνα 6.29 Η μέθοδος *isspace*

Στη συνέχεια θα δούμε τη μορφοποίηση συμβολοσειράς με τη μέθοδο `format`. Στο παράδειγμα της Εικόνας 6.30, αντί να χρησιμοποιήσουμε κάποια μεταβλητή για τη συμβολοσειρά, την αναθέτουμε σε μια σταθερά περικλείοντάς την μεταξύ των δύο εισαγωγικών και καλούμε τη μέθοδο. Προσέξτε ότι σε αυτήν την περίπτωση η συμβολοσειρά δεν είναι μια τυπική σταθερά: περιέχει δύο θέσεις, τις `{0}` και `{1}`, οι οποίες θα αντικατασταθούν με τις τιμές των μεταβλητών που δίνουμε εδώ. Δηλαδή, όταν εφαρμόσουμε τη μέθοδο `format` πάνω στη συμβολοσειρά, το σύμβολο `{0}` θα αντικατασταθεί με την τιμή της συμβολοσειράς `name`. Αντί για το σύμβολο `{1}` θα μπει η τιμή της μεταβλητής `age` (20), και άρα θα τυπωθεί Ο John είναι 20 ετών.

```

>>> age=20
>>> name='John'
>>> print('Ο {0} είναι {1} ετών.'.format(name,age))
Ο John είναι 20 ετών.

```

Εικόνα 6.30 Η μέθοδος *format*.

Εκτός από τις προαναφερθείσες, υπάρχουν αρκετές άλλες μέθοδοι έτοιμες προς χρήση. Για να τις δείτε, μπορείτε να ζητήσετε βοήθεια από το διερμηνευτή με την εντολή `help`:

```
>>> help(str)
```

Το module `string` περιέχει πολλές χρήσιμες σταθερές, κάποια παραδείγματα των οποίων βλέπουμε στην Εικόνα 6.31.


```
>>> import string
>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'
>>> string.ascii_uppercase
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> string.ascii_letters
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> string.digits
'0123456789'
```

Εικόνα 6.31 Χρήσιμες σταθερές συμβολοσειρές, διαθέσιμες μέσω του *module string*

6.3 Λίστες

Η λίστα είναι ο δεύτερος σύνθετος τύπος (μετά τις συμβολοσειρές) στον οποίο αναφερόμαστε στην Python. Οι λίστες (lists), δηλώνουν πράγματα καθώς και διατεταγμένα σύνολα τιμών. Οι τιμές (μέλη μιας λίστας) λέγονται στοιχεία (elements). Ο όρος «διατεταγμένα» σημαίνει ότι μπορούμε να ορίσουμε ότι αυτό είναι το πρώτο, αυτό είναι δεύτερο, αυτό τρίτο κλπ. Οι λίστες είναι παρόμοιες με τις συμβολοσειρές, οι οποίες είναι διατεταγμένα σύνολα χαρακτήρων, ωστόσο τα στοιχεία μιας λίστας μπορεί να είναι οποιουδήποτε τύπου. Μπορεί να είναι νούμερα, ακέραιοι, πραγματικοί, μπορεί να είναι άλλες λίστες. Μπορεί να περιέχουν στοιχεία οποιουδήποτε τύπου της Python και όλα αυτά να τα βάζουμε μαζί γιατί πιστεύουμε ότι πρέπει να είναι μαζί. Θα δούμε αργότερα ότι υπάρχουν άλλου τύπου σύνολα τα οποία δεν είναι διατεταγμένα κατά αυτόν τον τρόπο, ωστόσο και οι λίστες και συμβολοσειρές είναι διατεταγμένα σύνολα τα οποία γενικότερα ονομάζουμε ακολουθίες (sequences).

6.3.1 Τιμές τύπου λίστας

Ο απλούστερος τρόπος να δημιουργήσει κανείς μια λίστα, είναι να βάλει τα στοιχεία της μέσα σε `[]`. Παρακάτω βλέπουμε κάποια παραδείγματα λιστών. Οι πρώτες δύο λίστες (Εικόνα 6.32) είναι ομοιογενείς, δηλαδή, έχουν στοιχεία ίδιου είδους, η πρώτη ακεραίους, η δεύτερη συμβολοσειρές:

```
[1, 10, 15, 20, 30]
['apple', 'banana', 'mango', 'watermelon']
```

Εικόνα 6.32 Δημιουργία λίστας

Τα στοιχεία μιας λίστας δε χρειάζεται να είναι ίδιου τύπου. Για παράδειγμα η παρακάτω λίστα είναι ανομοιογενής και περιέχει έναν ακέραιο, έναν πραγματικό αριθμό, μια συμβολοσειρά, αλλά και μια άλλη λίστα. Μια λίστα μέσα σε μια άλλη λίστα λέγεται εμφωλευμένη λίστα (nested list). Έτσι η λίστα [2,4,5] είναι εμφωλευμένη μέσα στη μεγάλη λίστα:

```
[1, 2.5, 'cherry', [2,4,5]]
```

Εικόνα 6.33 Λίστα με στοιχεία διαφορετικού τύπου

Μια λίστα η οποία δεν περιέχει στοιχεία, ονομάζεται άδεια λίστα και συμβολίζεται με []. Μπορείτε να εκχωρήσετε τιμές τύπου λίστας σε μεταβλητές, όπως φαίνεται στην Εικόνα 6.34. Οι τύποι των αντικειμένων cheeses, numbers, empty στην Εικόνα 6.34 είναι list. Βλέπουμε ότι μπορούμε να δώσουμε ονόματα λιστών σαν ορίσματα, στη συνάρτηση print, π.χ. στην Εικόνα 6.34 το πρώτο όρισμα στην print είναι cheeses, μετά αλλαγή γραμμής ('\n'), μετά numbers, αλλαγή γραμμής ('\n'), και τέλος empty.

```
>>> cheeses=['Cheddar', 'Edam', 'Gouda']
>>> numbers=[17,100]
>>> empty=[]
>>> print cheeses, '\n', numbers, '\n', empty
['Cheddar', 'Edam', 'Gouda']
[17, 100]
[]
```

Εικόνα 6.34 Εκχώρηση τιμών τύπου λίστας σε μεταβλητές ή πέρασμα ως ορίσματα σε συναρτήσεις

Μια πολύ χρήσιμη λειτουργία είναι η αυτόματη παραγωγή από λίστες με ακεραίους βάσει της συνάρτησης range. Τέτοιες λίστες ακεραίων έχουν χρήση σε πολλά προγράμματα, οπότε αυτή η λειτουργικότητα είναι πολύ σημαντική. Η συνάρτηση range, λοιπόν, μάς φτιάχνει ένα εύρος (range) τιμών, μια ακολουθία τιμών από

το πρώτο όρισμα της συνάρτησης `range` (το `1` στο παράδειγμα της Εικόνας 6.35) έως το δεύτερο όρισμα «μείον ένα» (όπως ισχύει και σε άλλες περιπτώσεις στην Python, τα διαστήματα είναι κλειστά στο κάτω άκρο και ανοιχτά στο πάνω άκρο), άρα θα φτιάξει μια ακολουθία τιμών που θα είναι `1, 2, 3, 4, 5, 6, 7, 8, 9`. Αυτήν την ακολουθία τιμών μετά θα την πάρει η συνάρτηση `list` (που λέγεται και «γεννήτρια»-συνάρτηση, γιατί «γεννάει» λίστες) και θα την μετατρέψει σε λίστα. Έπειτα μπορούμε να τυπώσουμε τη νέα λίστα `numbers`. Προσέξτε εδώ ότι το `range` είναι μια συνάρτηση, το αποτέλεσμα του `range` δεν είναι λίστα, είναι τύπου `range`, το οποίο το παίρνει η «γεννήτρια»-συνάρτηση της λίστας και το μετατρέπει σε λίστα.

```
>>> numbers = list(range(1,10))
>>> print numbers
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Εικόνα 6.35 Δημιουργία λίστας που περιέχει συνεχόμενους ακεραίους

Η συνάρτηση `range` έχει ως ορίσματα δύο ακεραίους και επιστρέφει την ακολουθία των ακέραιων αριθμών στο διάστημα από το πρώτο έως το δεύτερό της όρισμα, συμπεριλαμβάνοντας το πρώτο όρισμα, αλλά αποκλείοντας το δεύτερο. Η συνάρτηση `list` αρχικοποιεί μια νέα λίστα η οποία περιέχει τους ακέραιους αριθμούς της ακολουθίας που δίνεται ως όρισμα. Χωρίς όρισμα η `list` επιστρέφει άδεια λίστα.

Υπάρχουν και δύο άλλες μορφές της `range`:

- Με ένα μόνο όρισμα, επιστρέφει μια ακολουθία τιμών που ξεκινάει από το `0`. Αν βάλουμε ένα μόνο όρισμα στο `range`, «καταλαβαίνει» ότι αρχίζει από το `0` και με βηματισμό `1` φτάνει μέχρι το όρισμα που του έχουμε δώσει `-1`. Άρα θα είναι `0,1,2,3,4,5,6,7,8,9`, και σταματά εκεί.
- Αν υπάρχει και τρίτο όρισμα, τότε αυτό το όρισμα προσδιορίζει το «βηματισμό» της ακολουθίας τιμών. Αν, λοιπόν, βάλουμε και τρίτο όρισμα, π.χ. το `(1,10,2)`, αυτό ορίζει το βηματισμό `2` και άρα δίνει `1,3,5,7,9`. Αν ήταν `(1,10,3)` θα έδινε `(1,4,7)`. Αν παραλειφθεί το τρίτο όρισμα, τότε εξ ορισμού το βήμα είναι `1`.

Παράδειγμα:

```
>>> numbers = list(range(1,10))
>>> print numbers
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> numbers = list(range(1,10,2))
>>> print numbers
[1, 3, 5, 7, 9]
```

Εικόνα 6.36 Δύο ακόμη μορφές της συνάρτησης *range*

6.3.2 Προσπέλαση στοιχείων λίστας

Για την προσπέλαση των στοιχείων μιας λίστας η σύνταξη είναι ίδια με αυτήν της προσπέλασης χαρακτήρων σε συμβολοσειρά:

- Η έκφραση μέσα στις αγκύλες προσδιορίζει το δείκτη.
- Οι δείκτες αρχίζουν από το 0 ως το `length-1`.

```
>>> numbers=[2,7,12,15]
>>> print(numbers[0])
2
>>> print(numbers[1])
7
```

Εικόνα 6.37 Προσπέλαση στοιχείων λίστας

Ακόμη, σαν δείκτης μπορεί να χρησιμοποιηθεί κάθε ακέραια έκφραση (π.χ. η έκφραση `3-1`). Θα πάρουμε λάθος αποτέλεσμα, ωστόσο, αν προσπαθήσουμε να χρησιμοποιήσουμε για δείκτη πραγματικό αριθμό:

```
>>> numbers=[2,7,12,15]
>>> print(numbers[3-1])
12
>>> print(numbers[1.0])

Traceback (most recent call last):
  File "<pyshell#110>", line 1, in <module>
    print(numbers[1.0])
TypeError: list indices must be integers, not float
```

Εικόνα 6.38 Παραδείγματα δεικτών σε λίστα

Είναι, επίσης, λάθος να προσπαθήσετε να γράψετε ή να διαβάσετε στοιχείο το οποίο δεν υπάρχει στη λίστα. Σε αυτήν την περίπτωση θα πάρουμε το error `IndexError: list assignment index out of range`, το οποίο σημαίνει ότι πάμε να αναθέσουμε τιμή σε μια θέση η οποία δεν υπάρχει. Στο παράδειγμα της Εικόνας 6.39, το λάθος οφείλεται στην προσπάθεια να προσπελαστεί η 8η θέση (`number[7]`) σε μια λίστα με 4 στοιχεία.

```
>>> numbers=[2,7,12,15]
>>> print(numbers[7])

Traceback (most recent call last):
  File "<pyshell#112>", line 1, in <module>
    print(numbers[7])
IndexError: list index out of range
```

Εικόνα 6.39 Μήνυμα λάθους

Όπως και στις συμβολοσειρές, μια αρνητική τιμή δείκτη μετράει από το τέλος της λίστας προς την αρχή, άρα το `-1` πάει στο τελευταίο στοιχείο, το `-2` στο προτελευταίο στοιχείο κ.ο.κ. Στην Εικόνα 6.40, λοιπόν, η θέση με δείκτη `-1` τυπώνει 15 και αυτή με `-2` τυπώνει 12. Μια σημαντική χρήση αυτής της λειτουργικότητας είναι ότι δε χρειάζεται να ξέρουμε το μήκος της λίστας (αν και το βρίσκουμε εύκολα με το `length`) για να πάμε στο τέλος της λίστας, μπορεί να γίνει εύκολα με δείκτη `-1`.

```
>>> numbers=[2,7,12,15]
>>> print(numbers[-1])
15
>>> print(numbers[-2])
12
```

Εικόνα 6.40 Αρνητική τιμή δείκτη

Στην Εικόνα 6.41 φαίνεται ένα παράδειγμα χρήσης μιας μεταβλητής βρόγχου ως δείκτη σε λίστα. Σ' αυτό το παράδειγμα η μεταβλητή βρόγχου `i` (αρχικοποιημένη στο 0) χρησιμοποιείται για να διαπεράσουμε όλα τα στοιχεία της λίστας `fruits`. Για όσο `i<4` (δηλαδή για 0,1,2,3) τυπώνουμε `fruits[0]`, `fruits[1]`, `fruits[2]`, `fruits[3]`.

```
6-list_access_elements2.py
1  fruits=['apple', 'banana', 'mango', 'melon']
2  i=0
3  while i<4:
4      print(fruits[i])
5      i=i+1
```

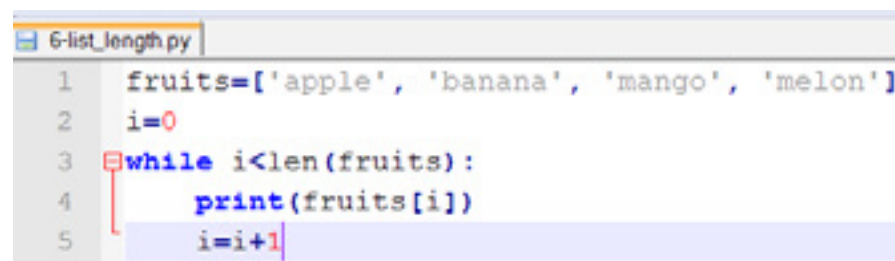
Εκτέλεση

```
>>>
apple
banana
mango
melon
```

Εικόνα 6.4I Μεταβλητή βρόχου ως δείκτης σε λίστα

6.3.3 Μήκος λίστας

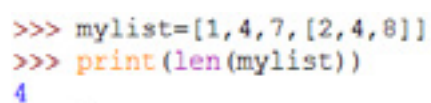
Όπως είχαμε δει και με τις συμβολοσειρές, έτσι και στις λίστες έχουμε μια συνάρτηση `len`, η οποία επιστρέφει το μήκος μιας λίστας. Σε βρόγχους οι οποίοι διατρέχουν λίστες, είναι καλύτερα να χρησιμοποιούμε την τιμή αυτής της συνάρτησης αντί μιας σταθεράς, γιατί η πρώτη προσαρμόζεται αυτόματα σε αυξομειώσεις του μήκους της λίστας. Έτσι, αν το μήκος της λίστας αλλάξει, δε θα χρειάζεται να κάνουμε αλλαγές στο πρόγραμμα. Έτσι στην Εικόνα 6.42 φαίνεται ότι χρησιμοποιώντας τη συνάρτηση `len`, μπορούμε να γενικεύσουμε το προηγούμενο πρόγραμμά μας, όπου είχαμε βάλει 4, ώστε αντί για 4 να χρησιμοποιεί τη συνάρτηση `len(fruits)`:



```
6-list_length.py
1  fruits=['apple', 'banana', 'mango', 'melon']
2  i=0
3  while i<len(fruits):
4      print(fruits[i])
5      i=i+1
```

Εικόνα 6.42 Η συνάρτηση `len` επιστρέφει το μήκος της λίστας

Στην περίπτωση εμφωλευμένης λίστας, αυτή μετράει σαν απλό στοιχείο. Στο συγκεκριμένο παράδειγμα της Εικόνας 6.43, έχουμε τρεις ακεραίους και μια λίστα εμφωλευμένη μέσα στην πιο μεγάλη λίστα που έχουμε. Αν κάνουμε `len`, το μήκος της λίστας `mylist` φαίνεται να είναι 4 και όχι 6, όπως θα μπορούσε να νομίσει κανείς, εάν μετρούσε και τα στοιχεία της εμφωλευμένης λίστας. Αν βέβαια ζητήσουμε το `length` του στοιχείου `mylist[3]` που είναι λίστα το αποτέλεσμα θα είναι 3.



```
>>> mylist=[1,4,7,[2,4,8]]
>>> print(len(mylist))
4
```

Εικόνα 6.43 Η συνάρτηση `len` με εμφωλευμένες λίστες

6.3.4 Έλεγχος του “ανήκειν”

Ο `in` είναι λογικός τελεστής ο οποίος ελέγχει αν μια τιμή ανήκει σε μια λίστα (επιστρέφοντας `true` ή `false`) και δουλεύει όπως και στις συμβολοσειρές. Μπορούμε, επίσης, να χρησιμοποιήσουμε την έκφραση `not in`. Παραδείγματα:

```
>>> fruits=['apple', 'banana', 'mango', 'melon']
>>> 'mango' in fruits
True
>>> 'zebra' in fruits
False
>>> 'zebra' not in fruits
True
```

Εικόνα 6.44 Ο λογικός τελεστής `in`

6.3.5 Λίστες και βρόχος `for`

Αντίστοιχα με τις συμβολοσειρές, μπορούμε να χρησιμοποιήσουμε το βρόχο `for` και με τις λίστες:


```
for VARIABLE in LIST:
    BODY
```

Η παραπάνω εντολή είναι ισοδύναμη με το παρακάτω `while` loop:

```
i = 0
while i < len(LIST):
    VARIABLE = LIST[i]
    BODY
    i = i + 1
```

Ωστόσο, ο βρόχος `for` έχει πιο συνοπτική και απλή έκφραση, επειδή μπορούμε να καταργήσουμε τη μεταβλητή βρόχου `i` (και άρα να αποφύγουμε τη διαχείρισή της, όπως αρχικοποίηση, αύξηση, κλπ.).

Ο προηγούμενος βρόγχος (Εικόνα 6.42) μπορεί, λοιπόν, να γραφεί ως εξής:



The image shows a code editor window titled '6-list_for.py' containing the following Python code:

```
1 fruits=['apple', 'banana', 'mango', 'melon']
2
3 for fruit in fruits:
4     print(fruit)
```

Below the code editor, the word "Εκτέλεση" (Execution) is written in a large, bold, green font. Underneath it, the output of the script is displayed in a monospaced font:

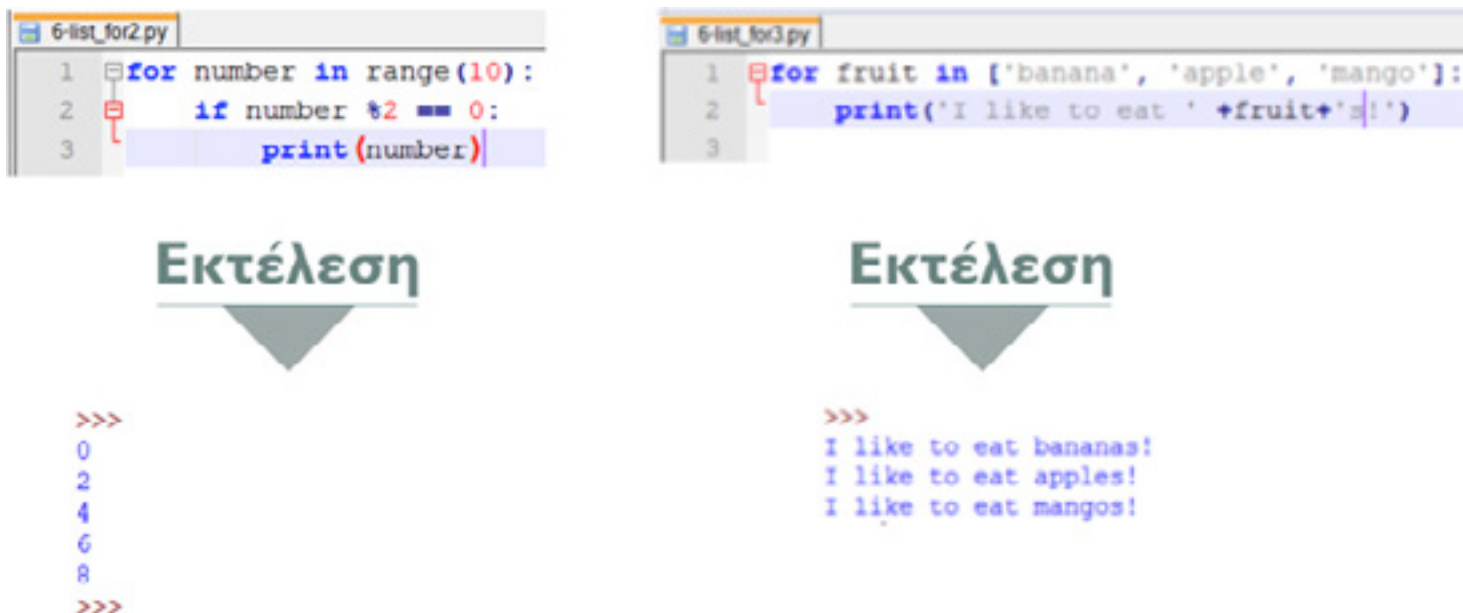
```
>>>
apple
banana
mango
melon
```

Εικόνα 6.45 Ο βρόγχος *for* σε συνδυασμό με λίστα

Ο βρόγχος σχεδόν διαβάζεται σαν κείμενο σε φυσική γλώσσα: “For (every) fruit in (the list of) fruits, print (the name of the) fruit.”

Στη συνέχεια θα δούμε μια άλλη χρήση του *for*, όχι με λίστες ή με συμβολοσειρές, αλλά με το *range* (Εικόνα 6.46), το οποίο και αυτό παράγει μια ακολουθία τιμών, εν προκειμένω από το 0 μέχρι το 9. Το *number*, λοιπόν, παίρνει τιμές στο 0, 1, 2, 3, 4, ..., 9 και εάν είναι ζυγός αριθμός (το υπόλοιπο της διαίρεσής του με το δύο είναι μηδέν) τότε τον τυπώνω. Άρα θα τυπώσει 0, 2, 4, 6, 8, και δε θα τυπώσει 10.

Αντίστοιχα στην Εικόνα 6.46 βλέπετε και το παράδειγμα με τα φρούτα. Το *fruit* παίρνει τις τιμές ‘banana’, ‘apple’, ‘mango’. Αυτό που κάνουμε στη συνέχεια είναι αλληλουχίες (concatenation) μεταξύ του πρώτου κομματιού της εκτύπωσης ‘I like to eat’, της συμβολοσειράς η οποία έχει την τρέχουσα τιμή του *fruit*, και του ‘s!’, οπότε π.χ. ‘I like to eat bananas!’.



Εικόνα 6.46 Ο βρόχος for με ακολουθία τιμών

6.3.6 Πράξεις σε λίστες

Ο τελεστής + συνενώνει λίστες (αλληλουχία, concatenation). Όπως βλέπουμε στο παράδειγμα της Εικόνας 6.47, η λίστα a έχει τα στοιχεία 1,2,3 και η b τα 4,5,6,7. Αν κάνω a+b, αυτό θα δημιουργήσει μια νέα λίστα η οποία θα την βάλει στη c που είναι αποτέλεσμα το οποίο δίνει η συνένωση με τις δύο λίστες 1,2,3 + 4,5,6,7. Προσέξτε ότι αν είχαμε εδώ 4,5,6,7 + 1,2,3 (ανάποδα) και κάναμε a+b, τότε θα ήταν 4,5,6,7,1,2,3 δηλαδή η Python δεν τα βγάζει τα στοιχεία κατά αύξουσα σειρά, αλλά κατά τη σειρά που είναι, ήδη, στις αρχικές λίστες. Άρα με τον τελεστή + στις λίστες δεν ισχύει η αντιμεταθετικότητα, όπως στην πρόσθεση ακεραίων και πραγματικών αριθμών.

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

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

Εικόνα 6.47 Ο τελεστής + σε λίστες

Ας δούμε, όμως, τι κάνει ο τελεστής *. Ο τελεστής * επαναλαμβάνει μια λίστα όσες φορές υποδεικνύει το όριο δεξιά του. Δηλαδή, η λίστα [0]*4 είναι μια νέα λίστα με 4 φορές το στοιχείο 0 [0,0,0,0]. Η λίστα [1,2,3]*3 είναι μια νέα λίστα με στοιχεία

[1,2,3, 1,2,3, 1,2,3,] με αυτήν τη σειρά. Προσέξτε ότι τα στοιχεία δε διατάσσονται αυτόματα κατ' αύξουσα σειρά [1,1,1, 2,2,2, 3,3,3]. Ουσιαστικά, λοιπόν, ο τελεστής * κάνει concatenate την ίδια λίστα τόσες φορές, όσες είναι το δεξί όριο. Ένα ακόμα παράδειγμα φαίνεται στην Εικόνα 6.48:

```
>>> c
[1, 2, 3, 4, 5, 6, 7]
>>> c*2
[1, 2, 3, 4, 5, 6, 7, 1, 2, 3, 4, 5, 6, 7]
```

Εικόνα 6.48 Ο τελεστής * σε λίστες

6.3.7 Φέτες από λίστες

Όπως και στις συμβολοσειρές, ο τελεστής φέτας (slice) δουλεύει επίσης και με τις λίστες. Στο παράδειγμα της Εικόνας 6.49 έχουμε τη συγκεκριμένη λίστα η οποία περιέχει τους χαρακτήρες από 'a' ως 'f'. Βλέπουμε αρχικά τη φέτα mylist[1:3]. Εδώ ισχύει πάλι ο κανόνας διαστημάτων (κλειστό από αριστερά, ανοικτό από δεξιά), άρα παίρνει το 2^ο στοιχείο ('b') και το 3^ο στοιχείο ('c') της λίστας, αλλά όχι το 4^ο. Επίσης, ισχύει ότι και στις συμβολοσειρές για τα παρελειπόμενα άνω και κάτω άκρα (δεικτών) της φέτας: Αν παραλείψουμε τον πρώτο δείκτη, η φέτα ξεκινά από την αρχή. Αν παραλείψουμε το δεύτερο, η φέτα φτάνει στο τέλος. Αν παραλείψουμε και τους δύο, η φέτα είναι αντίγραφο όλης της λίστας.

```
>>> mylist = ['a','b','c','d','e','f']
>>> mylist[1:3]
['b', 'c']
>>> mylist[:4]
['a', 'b', 'c', 'd']
>>> mylist[3:]
['d', 'e', 'f']
>>> mylist[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

Εικόνα 6.49 Ο τελεστής φέτας σε λίστες

6.3.8 Οι λίστες είναι μετατρέψιμες (mutable)

Μια άλλη σημαντική διαφορά ανάμεσα στις λίστες και τις συμβολοσειρές είναι ότι οι λίστες είναι μετατρέψιμες, σε αντίθεση με τις συμβολοσειρές. Δηλαδή, αν θέλουμε να αλλάξουμε κάτι σε μια συμβολοσειρά, πρέπει ουσιαστικά να φτιάξουμε μια καινούργια όπου να ενσωματώσουμε τις αλλαγές· τις λίστες μπορούμε πολύ εύκολα να τις αλλάξουμε διαλέγοντας το στοιχείο που θέλουμε να αλλάξουμε και αλλάζοντάς το «επί τόπου» (in place). Π.χ. στην Εικόνα 6.70 βλέπουμε πώς μπορούμε να αλλάξουμε το πρώτο στοιχείο της λίστας και από apple να το κάνουμε orange. Επίσης, μπορούμε να αλλάξουμε το τελευταίο στοιχείο και από mango να το κάνουμε melon:

```
>>> fruits=['apple','banana','mango']
>>> fruits[0]='orange'
>>> fruits[-1]='melon'
>>> print(fruits)
['orange', 'banana', 'melon']
```

Εικόνα 6.70 Αλλαγή στοιχείων λίστας

Με τον τελεστή φέτας μπορούμε να αλλάξουμε περισσότερα του ενός στοιχεία ταυτόχρονα. Στην Εικόνα 6.71 ορίζουμε να αλλάξει η τιμή των στοιχείων με δείκτες 1 και 2 (αρχικές τιμές 'b' και 'c' αντίστοιχα) σε 'x' και 'y'. Μπορούμε, λοιπόν, έτσι να κάνουμε πολλαπλές εκχωρήσεις σε λίστα:

```
>>> mylist = ['a','b','c','d','e','f']
>>> mylist[1:3] = ['x','y']
>>> print(mylist)
['a', 'x', 'y', 'd', 'e', 'f']
```

Εικόνα 6.71 Αλλαγή στοιχείων λίστας με χρήση του τελεστή φέτας

Μπορούμε αντίστοιχα να αφαιρέσουμε στοιχεία από μια λίστα, εκχωρώντας τη φέτα τους, εφόσον βέβαια είναι συνεχόμενα, στην άδεια λίστα. Στην Εικόνα 6.72 έχουμε την ίδια λίστα από 'a' ως 'f' και ζητάμε να βάλουμε την κενή λίστα στο 1:3. Αυτό κάνει το λεγόμενο delete. Δηλαδή, διαγράφει τα στοιχεία που υπάρχουν από 1 έως 3. συγκεκριμένα το 'b' και το 'c':

```
>>> mylist = ['a', 'b', 'c', 'd', 'e', 'f']
>>> mylist[1:3] = []
>>> print(mylist)
['a', 'd', 'e', 'f']
```

Εικόνα 6.72 Αφαίρεση στοιχείων λίστας με εκχώρηση άδειας λίστας

Ας δούμε τώρα έναν τρόπο με τον οποίο μπορούμε να προσθέσουμε στοιχεία σε μια λίστα. Στην Εικόνα 6.73 έχουμε τη λίστα `mylist` και θέλουμε να βάλουμε δύο στοιχεία: το 'b' και το 'c' σαν δεδομένα μιας άλλης λίστας, στη θέση 1. Βλέπουμε, λοιπόν, ότι έτσι μπορούμε να προσθέσουμε στοιχεία σε μια λίστα «στριμώχνοντας» τα σε μια άδεια φέτα, και αυξάνοντας το μέγεθος της λίστας μας από 3 σε 5:

```
>>> mylist = ['a', 'd', 'f']
>>> mylist[1:1] = ['b', 'c']
>>> print(mylist)
['a', 'b', 'c', 'd', 'f']
>>> mylist[4:4] = ['e']
>>> print(mylist)
['a', 'b', 'c', 'd', 'e', 'f']
```

Εικόνα 6.73 Προσθήκη στοιχείων λίστας με χρήση άδειας φέτας

6.3.9 Διαγραφή στοιχείων λίστας

Στο παράδειγμα της Εικόνας 6.72 είδαμε πώς σβήνουμε στοιχεία μιας λίστας χρησιμοποιώντας την `empty list`, ωστόσο αυτός ο τρόπος δεν είναι ο ενδεδειγμένος και μπορεί να οδηγήσει σε λάθη. Η Python έχει εναλλακτικά μια έτοιμη συνάρτηση, την `del` η οποία χρησιμοποιείται ως εξής (Εικόνα 6.74): Έχουμε τη λίστα `a` και θέλουμε να σβήσουμε το δεύτερο στοιχείο που έχει θέση (δείκτη) 1. Κάνουμε, λοιπόν, `del a[1]`, και αν τυπώσουμε την `a`, βλέπουμε ότι το 'two' έχει διαγραφεί:

```
>>> a = ['one', 'two', 'three']
>>> del a[1]
>>> print(a)
['one', 'three']
```

Εικόνα 6.74 Διαγραφή στοιχείων λίστας με χρήση του τελεστή `del`

Η `del` χειρίζεται και αυτή αρνητικούς δείκτες και μπορεί να χειριστεί φέτες, κλπ., όπως βλέπουμε στο παράδειγμα της Εικόνας 6.75. Όταν σβήσουμε το τελευταίο στοιχείο (θέση -1) του `mylist` 'a', 'b', 'c', 'd', 'e', 'f' θα μείνουν τα 'a', 'b', 'c', 'd', 'e'. Μετά σβήνουμε από το 2ο μέχρι το 5ο στοιχείο, δηλαδή σβήνουμε το 'bcd' και άρα μένει μόνον το πρώτο και το τελευταίο στοιχείο (σε νέες θέσεις 0 και 1). Αν τώρα ζητήσουμε να σβήσει το στοιχείο 3 (θέση 4) το οποίο δεν υπάρχει πια, αυτό μάς επιστρέφει λάθος για δείκτη εκτός διαστήματος.

```
>>> mylist = ['a','b','c','d','e','f']
>>> del mylist[-1]
>>> print(mylist)
['a', 'b', 'c', 'd', 'e']
>>> del mylist[1:4]
>>> print(mylist)
['a', 'e']
>>> del mylist[3]

Traceback (most recent call last):
  File "<pyshell#168>", line 1, in <module>
    del mylist[3]
IndexError: list assignment index out of range
```

Εικόνα 6.75 Ο τελεστής `del` και μια περίπτωση λάθους

6.3.10 Αντικείμενα και τιμές

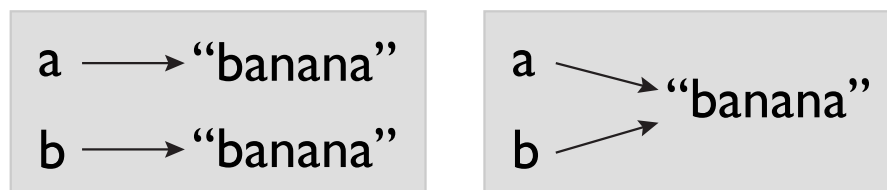
Ας υποθέσουμε ότι έχουμε τη συμβολοσειρά "banana" και δύο μεταβλητές `a` και `b` στις οποίες εκχωρούμε την τιμή "banana".

Δηλαδή:

```
a = "banana"
```

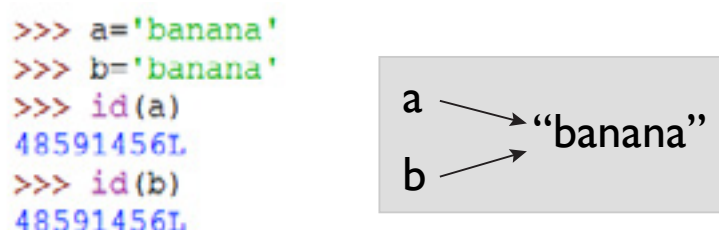
```
b = "banana"
```

Ξέρουμε, λοιπόν, ότι και η `a` και η `b` αναφέρονται στη συμβολοσειρά "banana", αλλά δε γνωρίζουμε αν αναφέρονται («δείχνουν») στην ίδια συμβολοσειρά! (Εικόνα 6.76) Στην πρώτη περίπτωση (αριστερά) οι `a` και `b` αναφέρονται σε δύο διαφορετικά «πράγματα» που έχουν την ίδια τιμή, ενώ στη δεύτερη περίπτωση αναφέρονται στο ίδιο «πράγμα». Τα «πράγματα» αυτά, συνήθως, ονομάζονται αντικείμενα (objects). Αντικείμενο είναι καθετί στο οποίο μπορεί να αναφερθεί μια μεταβλητή.



Εικόνα 6.76 Δύο διαφορετικές εκχωρήσεις

Στην Python το να ελέγχεις απευθείας τη μνήμη – όπως για παράδειγμα είναι δυνατό στην C – είναι δύσκολο, και άρα δε θα μπορούσαμε να εξακριβώσουμε ποια από τις δύο περιπτώσεις της Εικόνας 6.76 ισχύει. Ωστόσο στην Python κάθε αντικείμενο έχει μια μοναδική ταυτότητα (identifier), την οποία μπορούμε να δούμε με τη συνάρτηση `id`. Έτσι, συγκρίνοντας τις ταυτότητες των `a` και `b` μπορούμε να δούμε κατά πόσο αναφέρονται στο ίδιο αντικείμενο. Βλέπουμε, λοιπόν, στην Εικόνα 6.77 ότι η ταυτότητα είναι η ίδια και άρα οι `a` και `b` αναφέρονται στο ίδιο αντικείμενο, εν προκειμένω στην ίδια συμβολοσειρά.



Εικόνα 6.77 Δύο μεταβλητές που αναφέρονται στο ίδιο αντικείμενο

Δε συμβαίνει, ωστόσο, το ίδιο με τις λίστες. Αν υποθέσουμε ότι έχουμε δύο λίστες που έχουν τις ίδιες τιμές (όπως στην Εικόνα 6.78), μπορούμε να διαπιστώσουμε ότι αντιστοιχούν σε διαφορετικά αντικείμενα. Όσες τέτοιες λίστες και να είχαμε (περιέχοντας τα στοιχεία 1,2,3), θα μας έδιναν διαφορετικά `id`. Οι `a` και `b` έχουν, λοιπόν, την ίδια τιμή, αλλά αναφέρονται σε διαφορετικά αντικείμενα: οπότε είναι *ισότιμες*, αλλά όχι *ταυτόσημες* (Εικόνα 6.79).

```
>>> a=[1,2,3]
>>> b=[1,2,3]
>>> id(a)
36822152L
>>> id(b)
48079688L
```

Εικόνα 6.78 Μεταβλητές που δείχνουν λίστες

```
a —→ [1,2,3]
b —→ [1,2,3]
```

Εικόνα 6.79 Λίστες ισότιμες όχι ταυτόσημες

Τι θα γινόταν, ωστόσο, αν κάναμε την ανάθεση $b = a$; Τότε b και a θα έδειχναν στην ίδια τιμή, και άρα η λίστα στην οποία έδειχνε προηγουμένως το b , εξαφανίζεται, παύει να υπάρχει, χάνεται από τη μνήμη και a και b , πλέον, δείχνουν στην ίδια λίστα. Σε αυτήν την περίπτωση λέμε ότι το b είναι *ψευδώνυμο* του a .

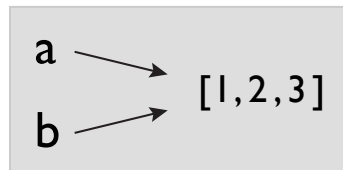
6.3.II Ψευδώνυμο

Εφόσον, λοιπόν, οι μεταβλητές αναφέρονται σε αντικείμενα, αν εκχωρήσουμε μεταβλητή b στην a , τότε και οι δύο αναφέρονται στο ίδιο αντικείμενο. Η ανάθεση, λοιπόν, $b=a$ δημιουργεί και δεύτερο όνομα (το b) μιας λίστας, το οποίο μέχρι πριν λίγο είχε το μοναδικό όνομα a . Έτσι λέμε ότι η συγκεκριμένη λίστα έχει *ψευδώνυμα* (*aliases*), τα a και b . Στην Εικόνα 6.80 αυτό φαίνεται και από το ότι τα a και b αντιστοιχούν σε αντικείμενο με τον ίδιο identifier, άρα το ίδιο αντικείμενο.

```
>>> a=[1,2,3]
>>> b=a
>>> id(a)
48079560L
>>> id(b)
48079560L
```

Εικόνα 6.80 Ψευδώνυμο

Το διάγραμμα κατάστασης, λοιπόν, θα είναι ως εξής:



Εικόνα 6.81 Διάγραμμα κατάστασης

Χρειάζεται ιδιαίτερη προσοχή στο γεγονός ότι αλλαγές που γίνονται στο αντικείμενο μέσω του ενός ψευδωνύμου, θα επηρεάζουν την εικόνα του αντικειμένου και μέσω του άλλου ψευδωνύμου (εφόσον το αντικείμενο στο οποίο αναφέρονται είναι το ίδιο). Αυτό φαίνεται στο παράδειγμα της Εικόνας 6.82, στο οποίο αλλάζοντας το αντικείμενο μέσω του `b` (`b[0]=3`) και κάνοντας `print a`, βλέπουμε αυτήν την αλλαγή.

```
>>> a=[1,2,3]
>>> b=a
>>> b[0]=3
>>> print(a)
[3, 2, 3]
```

Εικόνα 6.82 Αλλαγές σε ψευδώνυμο

Αυτή η συμπεριφορά των λιστών είναι ενίοτε χρήσιμη, ωστόσο, επειδή είναι δύσκολο για τον προγραμματιστή να υπολογίσει ποιες μεταβλητές είναι ψευδώνυμα και ποιες όχι, ιδιαίτερα όταν χειρίζεται πολλές από αυτές, και αυτή η συμπεριφορά μπορεί να αποτελέσει και πηγή λαθών. Γι' αυτό καλύτερα να αποφεύγουμε τη χρήση των ψευδωνύμων για τη μετατροπή αντικειμένων. Για τις συμβολοσειρές, ωστόσο, των οποίων η συμπεριφορά είναι πιο απλή (δεν είναι μετατρέψιμες), τα πράγματα είναι λιγότερο περίπλοκα.

6.3.12 Λίστες-κλώνοι

Ας υποθέσουμε τώρα ότι θέλουμε να *κλωνοποιήσουμε* μια λίστα, δηλαδή, να δημιουργήσουμε ένα νέο αντικείμενο τύπου λίστας το οποίο να είναι πανομοιότυπο με το αρχικό. Αυτό δεν μπορεί να συμβεί με την εκχώρηση, γιατί είδαμε ότι κάτι

τέτοιο δημιουργεί ψευδώνυμα προς το ίδιο αντικείμενο. Ο ευκολότερος τρόπος για να κλωνοποιήσουμε μια λίστα είναι με χρήση του τελεστή φέτας (slice), όπως φαίνεται στην Εικόνα 6.83. Με αυτόν τον τρόπο μπορούμε να κάνουμε αλλαγές στη λίστα b, χωρίς να ανησυχούμε για την a:

```
>>> a=[1,2,3]
>>> b=a[:]
>>> print(b)
[1, 2, 3]
>>> b[0]=3
>>> print(b)
[3, 2, 3]
>>> print(a)
[1, 2, 3]
```

Εικόνα 6.83 Κλωνοποίηση λίστας

Στη συνέχεια θα εξετάσουμε τι συμβαίνει όταν περνάμε μια λίστα ως παράμετρο συνάρτησης.

6.3.13 Η λίστα ως παράμετρος

Όταν περνάμε μια λίστα ως παράμετρο σε μια συνάρτηση, στην πραγματικότητα περνάμε μια αναφορά (ένα ψευδώνυμο) σε αυτήν και όχι ένα αντίγραφό της. Η πολύ σημαντική σημασία αυτής της αρχής είναι ότι αλλαγές που κάνουμε στη λίστα, μέσα στο «σώμα» της συνάρτησης, αλλάζουν τη λίστα και έξω από τη συνάρτηση. Το κύριο πλεονέκτημα αυτής της αρχής είναι η οικονομία μνήμης και χρόνου· σκεφτείτε για παράδειγμα την περίπτωση όπου μια λίστα έχει ένα δισεκατομμύριο στοιχεία και την περνάμε σαν παράμετρο σε μια συνάρτηση η οποία θα αλλάξει μόνο μερικά στοιχεία της. Η δημιουργία ενός αντιγράφου σε κάθε κλήση της συνάρτησης θα δημιουργούσε τεράστια σπατάλη στη μνήμη καθώς και χρονική καθυστέρηση, εφόσον η δημιουργία αντιγράφου είναι μια χρονοβόρα διαδικασία. Στο παράδειγμα της Εικόνας 6.85 η συνάρτηση head παίρνει ως όρισμα μια λίστα και επιστρέφει το πρώτο στοιχείο της. Εδώ περνάμε τη λίστα numbers ως παράμετρο με στοιχεία [1,2,3]. Η Python κάνει την εξωτερική μεταβλητή της συνάρτησης mylist να δείχνει στο αντικείμενο το οποίο δείχνει η numbers. Το διάγραμμα κατά-

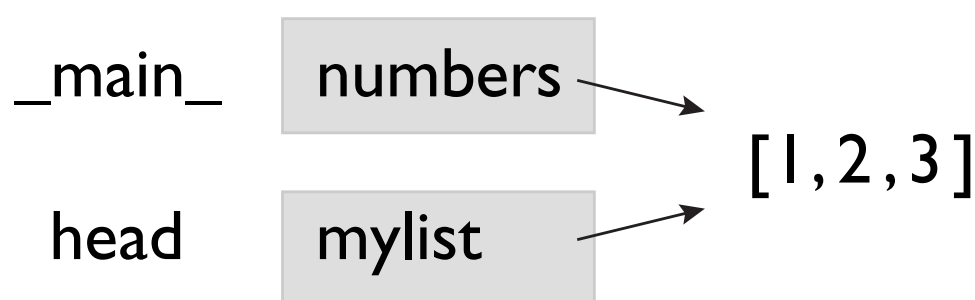
στασης της Εικόνας 6.86 δείχνει το ψευδώνυμο (alias) που δημιουργείται, και έτσι η συνάρτηση επιστρέφει το πρώτο στοιχείο της numbers, το οποίο και τυπώνει η print.

```
6-list_head.py
1 def head(mylist):
2     return mylist[0]
3
4 numbers=[1,2,3]
5 print(head(numbers))
```

Εκτέλεση

```
>>>
1
```

Εικόνα 6.84 Μια λίστα ως παράμετρος στη συνάρτηση head



Εικόνα 6.85 Το διάγραμμα κατάστασης

Στην Εικόνα 6.86 έχουμε το παράδειγμα μιας συνάρτησης η οποία κάνει και αλλαγή στη λίστα, σβήνοντας το πρώτο στοιχείο της λίστας. Έχουμε, λοιπόν, τη συνάρτηση deleteHead, η οποία σβήνει το πρώτο στοιχείο μιας λίστας. Χρησιμοποιούμε πάλι τη λίστα numbers με τιμή [1,2,3] και αφού καλέσουμε το deleteHead(numbers) και τυπώσουμε το αποτέλεσμα, μένει μόνο [2,3].

```

6-list_deleteHead.py
1 def deleteHead(mylist):
2     del mylist[0]
3
4 numbers=[1,2,3]
5 deleteHead(numbers)
6 print(numbers)

```

Εκτέλεση

Εικόνα 6.86 Η συνάρτηση *deleteHead* σβήνει το πρώτο στοιχείο της λίστας

Στο επόμενο παράδειγμα (Εικόνα 6.87) η συνάρτηση επιστρέφει την ουρά της λίστας. Προσέξτε ότι η συνάρτηση δε μεταβάλει τη λίστα, μόνον επιστρέφει το slice από το 1 και μετά. Θυμηθείτε ότι ο τελεστής φέτας (slice) δημιουργεί μια νέα λίστα. Άρα εδώ η *rest* είναι κλώνος, είναι μια νέα λίστα. Επειδή είναι μια νέα λίστα, οποιεσδήποτε αλλαγές στη *rest* δεν επηρεάζουν τη *numbers*.

```

6-list_tail.py
1 def tail(mylist):
2     return mylist[1:]
3
4 numbers=[1,2,3]
5 rest =tail(numbers)
6 print(rest)
7 print(numbers)

```

Εκτέλεση

Εικόνα 6.87 Παράλειψη ενός από τους δύο δείκτες σε φέτα συμβολοσειράς

6.3.14 Εμφωλευμένες λίστες

Είδαμε νωρίτερα την έννοια της εμφωλευμένης λίστας, δηλαδή, μιας λίστας μέσα σε μια άλλη λίστα. Θα δούμε εδώ ένα ακόμα παράδειγμα το οποίο θα εμβαθύνει την κατανόηση αυτής της έννοιας. Στην Εικόνα 6.88 διαλέγοντας το 4ο στοιχείο της λίστας (`mylist[3]`) και αναθέτοντάς το στο `inlist`, βλέπουμε με το `print(inlist)` ότι δείχνει πράγματι στην εσωτερική λίστα `[10,20]` και μετά μπορούμε να διαλέξουμε το πρώτο ή δεύτερο στοιχείο της `inlist`, όπως γνωρίζουμε. Στη συνέχεια, βλέπουμε ότι μπορούμε να χρησιμοποιούμε τους τελεστές επιλογής διαδοχικά (`mylist[3][0]`): με αρχική λίστα τη `mylist` και με στοιχείο μια εμφωλευμένη λίστα μέσα σε αυτή, επιλέγουμε κάποιο στοιχείο της εμφωλευμένης. Θα κάναμε κάτι αντίστοιχο αν είχαμε συμβολοσειρά. Σημειώστε ότι οι τελεστές `[]` υπολογίζουν από αριστερά προς τα δεξιά.

```
>>> mylist = ['hello',2.0,5,[10,20]]
>>> inlist = mylist[3]
>>> print(inlist)
[10, 20]
>>> print(inlist[0])
10
>>> print(inlist[1])
20
>>> print(mylist[3][0])
10
>>> print(mylist[3][1])
20
```

Εικόνα 6.88 Εμφωλευμένες λίστες

6.4 Πίνακες

Η έννοια των εμφωλευμένων λιστών και του τρόπου πρόσβασής τους μπορεί να γενικευτεί και να μας οδηγήσει στην έννοια των πινάκων. Παρακάτω βλέπουμε έναν δισδιάστατο πίνακα, ο οποίος μπορεί να αποτυπωθεί στην Python όπως φαίνεται στην Εικόνα 6.89.

Ο πίνακας:

1	2	3
4	5	6
7	8	9

μπορεί να παρασταθεί στην Python ως εξής:

```
>>> matrix = [[1,2,3],[4,5,6],[7,8,9]]
```

Εικόνα 6.89 Αποτύπωση πίνακα στην Python

Για να επιλέξουμε μια γραμμή πίνακα προχωράμε στο εξής:

```
>>> print(matrix[0])  
[1, 2, 3]
```

Εικόνα 6.90 Επιλογή γραμμής πίνακα

Ή μπορούμε να εξαγάγουμε ένα απλό στοιχείο πίνακα.

```
>>> print(matrix[1][2])  
6
```

Εικόνα 6.91 Εξαγωγή ενός απλού στοιχείου πίνακα

6.5 Λίστες και συμβολοσειρές

Οι λίστες και οι συμβολοσειρές, συχνά, χρησιμοποιούνται συνδυαστικά σε περιπτώσεις όπου θέλουμε να διαχειριστούμε κάποιο κείμενο. Για παράδειγμα, μια πολύ χρήσιμη λειτουργία είναι η δημιουργία μιας λίστας από μια συμβολοσειρά χρησιμοποιώντας τη μέθοδο `split` του module `string`, η οποία σπάει σε μια λίστα το κείμενο της συμβολοσειράς. Τα όρια των λέξεων – ή εν γένει των κομματιών κειμένου που μας ενδιαφέρει να διαχωρίσουμε – θεωρούνται οι χαρακτήρες `whitespace`. Προαιρετικά, μπορεί να χρησιμοποιηθεί όρισμα το οποίο ορίζει ποιοι άλλοι χαρακτήρες οριοθετούν λέξεις, αν όχι το `whitespace`. Ένα παράδειγμα παρουσιάζεται στην ακόλουθη Εικόνα 6.92, όπου η `split` δημιουργεί μια λίστα με στοιχεία: `'The'`, `'rain'`, `'in'`, `'Spain'`. Στη συνέχεια ορίζουμε το `'ai'` σαν το όριο μεταξύ λέξεων και το `split` έτσι δημιουργεί τα στοιχεία `'The r'`, `'n in Sp'`, `'n'`. Προσέξτε, τέλος, στη χρήση της `split` τη διαφορά μεταξύ μεθόδων και συναρτήσεων· η `split` είναι μέθοδος του τύπου `string` και έτσι χρησιμοποιείται μόνο στα πλαίσια αυτού του τύπου.

```
>>> song = 'The rain in Spain...'  
>>> songwords = song.split()  
>>> print(songwords)  
['The', 'rain', 'in', 'Spain...']  
>>> songwords2 = song.split('ai')  
>>> print(songwords2)  
['The r', 'n in Sp', 'n...']
```

Εικόνα 6.92 Η μέθοδος `split`

Ας θυμηθούμε εδώ ότι μια κατασκευάστρια συνάρτηση των λιστών είναι η `list`. Είχαμε δει νωρίτερα για παράδειγμα ότι το `list(range(10))` κατασκευάζει μια λίστα με στοιχεία τους ακραίους από το 0 μέχρι το 9. Εδώ βλέπουμε μια άλλη εφαρμογή του `list` στην οποία παίρνει σαν όρισμα όχι ένα `range` αλλά μια συμβολοσειρά. Αυτό δημιουργεί μια λίστα με στοιχεία κάθε χαρακτήρα της συμβολοσειράς.

```
>>> s = 'spam'
>>> t = list(s)
>>> print(t)
['s', 'p', 'a', 'm']
```

Εικόνα 6.93 Η συνάρτηση *list*

Την αντίστροφη λειτουργία της *list* κάνει η μέθοδος *join*, η οποία συγχωνεύει μια λίστα που έχει ομάδες χαρακτήρων σε μια συμβολοσειρά. Στο παράδειγμα της Εικόνας 6.94 δημιουργεί τη συμβολοσειρά ‘The rain in Spain’, όταν ο οριοθέτης (delimiter) μεταξύ των λέξεων είναι ο προφανής (whitespace), ωστόσο μπορεί να χρησιμοποιηθεί με οποιονδήποτε οριοθέτη επιθυμούμε, όπως π.χ. με το *underscore*, δημιουργώντας τη συμβολοσειρά ‘The_rain_in_Spain’.

```
>>> songwords = ['The', 'rain', 'in', 'Spain...']
>>> delimiter = ' '
>>> song = delimiter.join(songwords)
>>> print(song)
The rain in Spain...
```

Εικόνα 6.94 Η συνάρτηση *join*

Στη συνέχεια θα δούμε τι μεθόδους προσφέρει η Python με τύπους που είναι ήδη ορισμένοι, όπως λίστες, πλειάδες, λεξικά, και πώς μπορούμε να τους χρησιμοποιούμε. Στο Κεφάλαιο 8, θα δούμε πώς μπορούμε να δημιουργούμε δικούς μας τύπους και μεθόδους, εμβαθύνοντας στην έννοια του αντικειμενοστραφούς προγραμματισμού.

6.6 Μέθοδοι για λίστες

Μια ακόμη μέθοδος για λίστες είναι η `append`, η οποία προσθέτει ένα νέο στοιχείο στο τέλος μιας λίστας.

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> t.append('e')
>>> print(t)
['a', 'b', 'c', 'd', 'e']
```

Εικόνα 6.95 Η μέθοδος `append`

Η μέθοδος `sort` ταξινομεί μια λίστα κατά αύξουσα σειρά.

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> print(t)
['a', 'b', 'c', 'd', 'e']
```

Εικόνα 6.96 Η μέθοδος `sort`

Υπάρχουν και άλλες πολλές μέθοδοι, όπως είναι οι: `extend`, `pop`, `remove`.

```
>>> t1 = ['a', 'b']
>>> t2 = ['c', 'd', 'e']
>>> t1.extend(t2)
>>> print(t1)
['a', 'b', 'c', 'd', 'e']
>>> letter = t1.pop(0)
>>> print(letter)
a
>>> print(t1)
['b', 'c', 'd', 'e']
>>> t1.remove('e')
>>> print(t1)
['b', 'c', 'd']
```

Εικόνα 6.97 Οι μέθοδοι `extend`, `pop`, `remove`

Μπορούμε να έχουμε μια πλήρη εικόνα όλων των μεθόδων των λιστών με την εντολή `help(list)` και προγραμματιστικά μέσω της `string.help(list)`, για παράδειγμα:

```
>>>help(list)
```

Ο αναγνώστης καλείται να παρακολουθήσει και το [διαδραστικό υλικό](#) που συνοδεύει αυτήν την ενότητα.

6.7 Πλειάδες

Έχοντας κατανοήσει την αναπαράσταση και λειτουργία των λιστών, θα προχωρήσουμε σε μια σχετικά απλή παραλλαγή τους, τις πλειάδες (tuples). Η έννοια των πλειάδων θυμίζει κάπως την έννοια του διανύσματος στα μαθηματικά, χωρίς ωστόσο να είναι το ίδιο πράγμα. Μια πλειάδα (tuple) είναι μια διατεταγμένη ακολουθία τιμών, οι οποίες αντιστοιχίζονται σε δείκτες. Οι τιμές οι οποίες είναι μέλη μιας πλειάδας, λέγονται στοιχεία (elements) και μπορεί να είναι οποιουδήποτε τύπου (αριθμοί, συμβολοσειρές, λίστες, πλειάδες).

Οι πλειάδες μοιάζουν με τις λίστες στη χρήση δεικτών, ειδικότερα στον τρόπο με τον οποίο διατρέχονται και στη χρήση του τελεστή φέτας. Όμως οι πλειάδες, όπως και οι συμβολοσειρές, είναι αμετάβλητες. Οι πλειάδες χρησιμοποιούνται, συνήθως, στις περιπτώσεις όπου πρόκειται να χρησιμοποιηθεί μια ακολουθία τιμών (πλειάδα) η οποία δεν πρόκειται να αλλάξει.

Συντακτικά μια πλειάδα είναι μια λίστα τιμών οι οποίες χωρίζονται με κόμμα (Εικόνα 6.98).

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

Εικόνα 6.98 Ορισμός πλειάδας με κόμμα

Παρόλο που δεν είναι αναγκαίο, είναι σύνηθες να περικλείουμε τις πλειάδες με παρενθέσεις (Εικόνα 6.99). Σημειώστε ότι για να μη συγχέουμε τις πλειάδες με τις λίστες, χρησιμοποιούμε απλές (αντί για τετράγωνες) παρενθέσεις.

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

Εικόνα 6.99 Ορισμός πλειάδας με παρενθέσεις

Για να δημιουργήσουμε μια άδεια πλειάδα, χρησιμοποιούμε άδειες παρενθέσεις:

```
>>> t1 = ()
>>> t1
()
>>> type(t1)
<type 'tuple'>
```

Εικόνα 6.100 Δημιουργία άδειας πλειάδας

Για να δημιουργήσουμε μια πλειάδα με ένα μόνο στοιχείο, πρέπει να προσθέσουμε κόμμα:

```
>>> t2 = ('a',)
>>> t2
('a',)
>>> type(t2)
<type 'tuple'>
```

Εικόνα 6.101 Δημιουργία πλειάδας μόνο με ένα στοιχείο

Χωρίς το κόμμα, η Python νομίζει ότι πρόκειται για συμβολοσειρά μέσα σε παρενθέσεις:

```
>>> t3 = ('a')
>>> t3
'a'
>>> type(t3)
<type 'str'>
```

Εικόνα 6.102 Χωρίς το κόμμα, έχουμε συμβολοσειρά μέσα σε παρενθέσεις

Οι πράξεις πάνω σε πλειάδες είναι παρόμοιες με τις πράξεις πάνω σε λίστες. Ο τελεστής `[]` («δείκτης») επιλέγει στοιχείο από μια πλειάδα. Να θυμάστε πάντα ότι αρχίζουμε από το 0.

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> t[0]
'a'
>>> t[1]
'b'
>>> t[-1]
'e'
```

Εικόνα 6.103 Ο τελεστής []

Ο τελεστής «φέτα» επιλέγει διάστημα στοιχείων (όπως ακριβώς και στις λίστες):

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> t[1:3]
('b', 'c')
```

Εικόνα 6.104 Ο τελεστής “φέτα”

Επίσης, μπορούμε να έχουμε πράξεις με τους τελεστές +, * και με τον τελεστή in.

```
>>> t1 = ('a', 'b')
>>> t2 = ('c', 'd')
>>> t3 = t1+t2
>>> print(t3)
('a', 'b', 'c', 'd')
>>> t4 = 2*t1
>>> print(t4)
('a', 'b', 'a', 'b')
```

Εικόνα 6.105 Οι τελεστές +, *

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> 'b' in t
True
>>> 'x' in t
False
>>> 'x' not in t
True
```

Εικόνα 6.106 Ο τελεστής in

6.7.1 Μήκος πλειάδας

Η συνάρτηση `len` επιστρέφει το μήκος μιας πλειάδας (τον αριθμό των στοιχείων που περιέχει). Για παράδειγμα:

```
>>> t1 = ('a', 'b', 'c', 'd')
>>> print(len(t1))
4
>>> t2 = (1, 4, 'a', 'c', 2.7, [3,4], (7,10,12))
>>> print(len(t2))
7
```

Εικόνα 6.107 Η συνάρτηση `len`

6.7.2 Πέρασμα πλειάδας

Για να διατρέξουμε μια πλειάδα, κάνουμε ακριβώς ό,τι κάναμε και με τις λίστες. Για παράδειγμα:



Εικόνα 6.108 Προσπέλαση στοιχείων λίστας

6.7.3 Οι πλειάδες είναι αμετάβλητες

Οι πλειάδες είναι αμετάβλητες (μη μετατρέψιμες). Αν προσπαθήσουμε να αλλάξουμε ένα από τα στοιχεία μιας πλειάδας, παίρνουμε μήνυμα σφάλματος από το διερμηνευτή της Python:

```
>>> t = ('a', 'b', 'c', 'd')
>>> t[0] = 'x'

Traceback (most recent call last):
  File "<pyshell#68>", line 1, in <module>
    t[0] = 'x'
TypeError: 'tuple' object does not support item assignment
```

Εικόνα 6.109 Μήνυμα σφάλματος σε περίπτωση προσπάθειας αλλαγής ενός από τα στοιχεία της πλειάδας

Μπορούμε, όμως, να αντικαταστήσουμε μια πλειάδα με μια άλλη:

```
>>> t = ('a', 'b', 'c', 'd')
>>> t = ('A',) + t[1:]
>>> print(t)
('A', 'b', 'c', 'd')
```

Εικόνα 6.110 Αντικατάσταση πλειάδας από μία άλλη

6.7.4 Εκχώρηση σε πλειάδες

Συχνά σε προγράμματα εμφανίζεται η ανάγκη να ανταλλάξουμε τις τιμές δύο μεταβλητών. Υποθέστε, λοιπόν, ότι έχουμε δύο μεταβλητές *a* και *b*, οι οποίες έχουν διαφορετικές τιμές και θέλουμε να ανταλλάξουμε τις τιμές τους. Συμβατικά χρησιμοποιούμε μια προσωρινή μεταβλητή, όπως φαίνεται στην Εικόνα 6.111. Δίνουμε σε αυτήν προσωρινά την τιμή της μεταβλητής *a*, αλλάζουμε την τιμή της *a*, και μετά δίνουμε στην *b* την αρχική τιμή της *a*:

```

>>> a = 2
>>> b = 4
>>> temp = a
>>> a = b
>>> b = temp
>>> print a,b
4 2

```

Εικόνα 6.III Χρήση προσωρινής μεταβλητής

Η Python μπορεί να εκφράσει την ίδια λειτουργία με αρκετά πιο συνοπτικό τρόπο χωρίς χρήση της ενδιάμεσης μεταβλητής. Ο τρόπος συνίσταται στη χρήση εκχώρησης με πλειάδες, όπως φαίνεται στο παράδειγμα της Εικόνας 6.II2. Το αριστερό μέρος της $a,b = b,a$ είναι πλειάδα μεταβλητών. Το δεξί μέρος είναι πλειάδα τιμών. Κάθε τιμή εκχωρείται στην αντίστοιχη μεταβλητή. Όλες οι εκφράσεις στη δεξιά μεριά υπολογίζονται **πρώτες** και μετά γίνονται οι εκχωρήσεις.

```

>>> a = (2,)
>>> b = (4,)
>>> a,b=b,a
>>> a
(4,)
>>> b
(2,)

```

Εικόνα 6.II2 Ανταλλαγή τιμών στην Python

Προφανώς, το πλήθος των μεταβλητών και τιμών στις δύο πλευρές της εκχώρησης πρέπει να είναι το ίδιο:

```

>>> a,b = 1,2,3

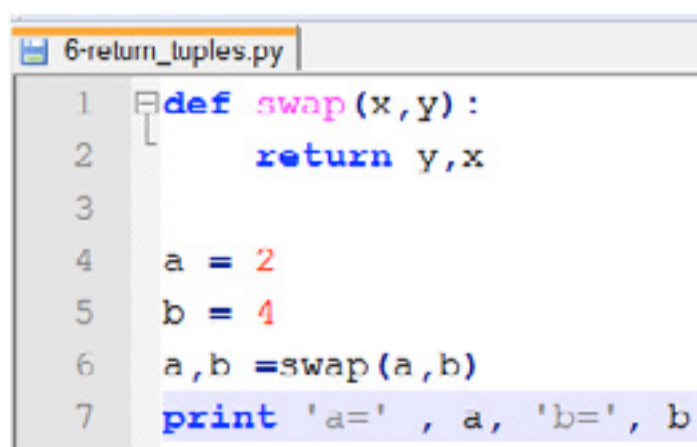
Traceback (most recent call last):
  File "<pyshell#85>", line 1, in <module>
    a,b = 1,2,3
ValueError: too many values to unpack

```

Εικόνα 6.II3 Σφάλμα ανταλλαγής τιμών στην Python

6.7.5 Οι πλειάδες ως επιστρεφόμενες τιμές

Οι συναρτήσεις μπορούν να επιστρέφουν πλειάδες ως τιμές. Για παράδειγμα η συνάρτηση `swap` (Εικόνα 6.114) παίρνει δύο τιμές, μια πλειάδα `x, y` και επιστρέφει την πλειάδα `y, x`. Άρα, κάποιος που καλεί την **`swap`** (`x, y`) μπορεί να εκχωρήσει το αποτέλεσμα της `swap (x, y)` στην πλειάδα `x, y`, οπότε πετυχαίνει την αλλαγή.

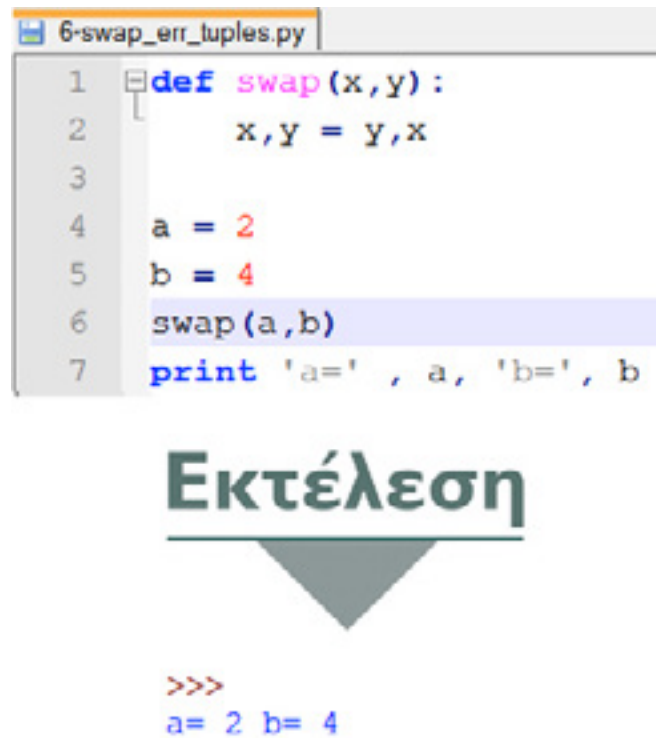


```
1 def swap(x,y):  
2     return y,x  
3  
4 a = 2  
5 b = 4  
6 a,b = swap(a,b)  
7 print 'a=' , a, 'b=' , b
```

Εκτέλεση

Εικόνα 6.114 Οι συναρτήσεις επιστρέφουν πλειάδες ως τιμές

Στην Εικόνα 6.115 βλέπουμε μια λάθος υλοποίηση της `swap`. Είναι λάθος, γιατί κατ'αρχήν δεν επιστρέφει τίποτα η συνάρτηση, οπότε θα υποθέταμε ότι γίνεται έμμεσα η αλλαγή. Ωστόσο δε γίνεται, και αυτό οφείλεται στο ότι χρησιμοποιούνται τοπικές μεταβλητές. Οι `x` και `y` είναι τοπικές μεταβλητές της `swap` και η αλλαγή τους μένει μέσα στη συνάρτηση. Εδώ φαίνεται ότι η συνάρτηση είναι *συντακτικά σωστή*, ωστόσο δεν κάνει αυτό που θέλουμε. Αυτό είναι μια περίπτωση ενός *λάθους σημαντικής ή σημασιολογίας* του προγράμματος. Αυτά είναι από τα δυσκολότερα λάθη που μπορεί να ανακαλύψει κανείς, γιατί δεν μπορεί να τα διαγνώσει ο διερμηνέας και έτσι αδυνατεί να μάς βοηθήσει.



Εικόνα 6.115 Λάθος σημαντικής

6.7.6 Η συνάρτηση tuple

Η συνάρτηση tuple δέχεται ως όρισμα μια συμβολοσειρά ή μια λίστα (ακολουθία) και επιστρέφει μια πλειάδα. Με κενό όρισμα επιστρέφει μια άδεια πλειάδα.

```
>>> t1 = tuple()
>>> print(t1)
()
>>> t2=tuple('spam')
>>> print(t2)
('s', 'p', 'a', 'm')
>>> t3=tuple([1,2,3,4])
>>> print(t3)
(1, 2, 3, 4)
```

Εικόνα 6.116 Η συνάρτηση tuple

6.7.7 Μέθοδοι για πλειάδες

Χρήσιμες μέθοδοι στις πλειάδες είναι οι `count` και `index`. Η μέθοδος `count` επιστρέφει τον αριθμό των εμφανίσεων μιας τιμής σε μια πλειάδα. Για παράδειγμα:

```
>>> t1 = ('a', 'b', 'c', 'd')
>>> t1.count('a')
1
```

Εικόνα 6.117 Η μέθοδος `count`

Η μέθοδος `index` επιστρέφει τον πρώτο δείκτη στον οποίο αντιστοιχεί μια τιμή. Αν δεν υπάρχει η τιμή εμφανίζεται μήνυμα λάθους. Για παράδειγμα:

```
>>> t1 = ('a', 'b', 'c', 'd')
>>> t1.index('a')
0
>>> t1.index('f')

Traceback (most recent call last):
  File "<pyshell#96>", line 1, in <module>
    t1.index('f')
ValueError: tuple.index(x): x not in tuple
```

Εικόνα 6.118 Η μέθοδος `index`

Ανακεφαλαιώνοντας, είδαμε ως τώρα σε αυτό το κεφάλαιο τις λίστες και τις πλειάδες, οι οποίες είναι διατεταγμένα σύνολα. Θυμηθείτε ότι η λίστα είναι μετατρέψιμη (μπορούμε να αλλάζουμε επιμέρους στοιχεία) ενώ η πλειάδα δεν είναι.

Ο αναγνώστης καλείται να παρακολουθήσει και το [διαδραστικό υλικό](#) που συνοδεύει αυτήν την ενότητα. Στη συνέχεια θα μιλήσουμε για τον τελευταίο προκατασκευασμένο τύπο δεδομένων στην Python τον οποίο θα δούμε σε αυτό το σύγγραμμα, τα λεξικά (*dictionaries*).

6.8 Λεξικά

Τα λεξικά (dictionaries) είναι διατεταγμένα σύνολα, όπως οι λίστες και οι πλειάδες. Μια χαρακτηριστική διαφορά τους είναι ότι στις λίστες και στις πλειάδες οι δείκτες οι οποίοι μας επιτρέπουν να προσπελάσουμε στοιχεία, είναι αυστηρά ακέραιοι (π.χ. `t[index]`), όπου `t` είναι μια λίστα ή μια πλειάδα και `index` ένας ακέραιος), ενώ στα λεξικά μπορούν να χρησιμοποιηθούν και άλλοι τύποι (ωστόσο μη μετατρέψιμοι).

Τα λεξικά είναι, λοιπόν, ένας γενικευμένος σύνθετος τύπος όπως οι λίστες, αλλά χωρίς διάταξη (η οποία υπάρχει στις λίστες και τις πλειάδες), και άρα στα λεξικά δεν ορίζουμε το 1^ο, 2^ο, κλπ. στοιχείο τους. Εφόσον, λοιπόν, δεν υπάρχει διάταξη, θα πρέπει να γενικεύσουμε τον τρόπο επιλογής των αντικειμένων μέσα στα λεξικά. Ο τρόπος αυτός είναι η χρήση του κλειδιού, το οποίο είναι ένας δείκτης γενικού τύπου και άρα κάθε στοιχείο ενός λεξικού έχει δύο μέρη, το κλειδί και την τιμή (περιεχόμενο) του στοιχείου.

Αυτή η δυνατότητα αντιστοίχισης (κλειδί–τιμή) είναι σε ευρεία χρήση σε πληροφορικά συστήματα εκτός της Python, όπως για παράδειγμα στις βάσεις δεδομένων, οι οποίες προσφέρουν την αποθήκευση και ανάσυρση πληροφορίας βάσει κλειδιών. Στη συνέχεια θα χρησιμοποιήσουμε ένα απλό παράδειγμα για να εμβαθύνουμε στη χρήση και διαχείριση των λεξικών στην Python. Ας πάρουμε, λοιπόν, ως παράδειγμα ένα αγγλο-ισπανικό λεξικό στο οποίο χρησιμοποιούμε ως κλειδιά τις αγγλικές λέξεις και σαν τιμές του λεξικού τις αντίστοιχες ισπανικές λέξεις. Σε αυτήν την περίπτωση η διάταξη δεν έχει κανένα νόημα. Οπότε η επιλογή της τιμής (ισπανικής λέξης) γίνεται με βάση το κλειδί (αγγλική λέξη), το οποίο εδώ είναι μια συμβολοσειρά. Ωστόσο, όπως θα δούμε αργότερα, κλειδιά και τιμές μπορούν να είναι και σύνθετοι τύποι.

Στο παράδειγμα της Εικόνας 6.119 βλέπουμε τη δημιουργία ενός αγγλο-ισπανικού λεξικού. Αρχίζουμε με το άδειο λεξικό (το οποίο γράφεται με ένα ζευγάρι αγκίστρων `{ }`) και προσθέτουμε στοιχεία:

```
>>> eng2sp = {}  
>>> eng2sp['one'] = 'uno'  
>>> eng2sp['two'] = 'dos'
```

Εικόνα 6.119 Δημιουργία λεξικού

Τα στοιχεία του λεξικού εμφανίζονται σε λίστα, χωρισμένα με κόμματα. Κάθε στοιχείο περιλαμβάνει ένα δείκτη και μια τιμή που διαχωρίζονται μεταξύ τους με μια άνω και κάτω τελεία. Σε ένα λεξικό, οι δείκτες ονομάζονται κλειδιά, οπότε τα στοιχεία ονομάζονται ζεύγη κλειδιών-τιμών. Ο δείκτης (κλειδί) της τιμής 'uno' είναι η αγγλική λέξη 'one' και αντίστοιχα της τιμής 'dos', η αγγλική λέξη 'two'. Αν στη συνέχεια τυπώσουμε το λεξικό, βλέπουμε ότι αποτελείται από ζευγάρια αντικειμένων (εν γένει σύνθετων αντικειμένων):

```
>>> print(eng2sp)  
{ 'two': 'dos', 'one': 'uno' }
```

Εικόνα 6.120 Εκτύπωση τρέχουσας τιμής του λεξικού

Εφόσον τα λεξικά δεν υποστηρίζουν κάποια a priori διάταξη των στοιχείων τους, η σειρά εκτύπωσης την οποία επιλέγει ο διερμηνευτής δεν είναι απαραίτητο να ακολουθεί συγκεκριμένη πολιτική, και κάθε υλοποίηση του διερμηνευτή μπορεί να επιλέγει τη σειρά εκτύπωσης (θα μπορούσε, ενδεχομένως, να είναι και τυχαία). Νεότερες εκδόσεις της Python, υποστηρίζουν και διατεταγμένα λεξικά σε αρκετά modules, αλλά δε θα ασχοληθούμε με αυτά στο παρόν σύγγραμμα.

Το κλειδί πρέπει να είναι αμετάβλητη (μη μετατρέψιμη) τιμή, μπορεί να είναι συμβολοσειρά, αλλά μπορεί να είναι και πλειάδα. Οι τιμές του λεξικού μπορούν να είναι αμετάβλητα ή μετατρέψιμα αντικείμενα. Τα λεξικά, ωστόσο, είναι μετατρέψιμα και μπορούμε εύκολα να προσθέσουμε και να διαγράψουμε στοιχεία. Σε ένα λεξικό δεν υπάρχει η έννοια της θέσης- δείκτη και έτσι δεν μπορούμε να κάνουμε πέρασμα (scan) ή να χρησιμοποιήσουμε φέτες (slices).

Ένα παράδειγμα δημιουργίας λεξικού με πιο συνοπτικό τρόπο συγκριτικά με αυτό της Εικόνας 6.120 είναι το εξής:

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
>>> print(eng2sp)
{'three': 'tres', 'two': 'dos', 'one': 'uno'}
```

Εικόνα 6.121 Παράδειγμα υλοποίησης λεξικού

Μπορούμε να δημιουργήσουμε ένα λεξικό και με τη χρήση της ενσωματωμένης συνάρτησης `dict`, η οποία όταν καλεστεί χωρίς ορίσματα, παράγει το κενό λεξικό (Εικόνα 6.122). Όταν καλεστεί με μια σειρά τιμών, όπως φαίνεται στην Εικόνα 6.122, η συνάρτηση `dict` δημιουργεί ένα λεξικό με αυτά τα αντικείμενα σε αντιστοιχία:

```
>>> d1 = dict()
>>> print(d1)
{}
>>> d2 = dict(one='uno', two='dos', three='tres')
>>> print(d2)
{'three': 'tres', 'two': 'dos', 'one': 'uno'}
```

Εικόνα 6.122 Η συνάρτηση `dict`

Στην Εικόνα 6.123 φαίνεται ότι ο προγραμματιστής μπορεί να επιλέξει την τιμή ενός κλειδιού: σε αυτό το παράδειγμα τα κλειδιά `'one'`, `'two'`, και `'five'`. Παρατηρήστε ότι αν επιλεγθεί κάποιο κλειδί που δεν υπάρχει (όπως εδώ το `'five'`), παράγεται μήνυμα σφάλματος:

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
>>> print(eng2sp['one'])
uno
>>> print(eng2sp['two'])
dos
>>> print(eng2sp['five'])

Traceback (most recent call last):
  File "<pyshell#108>", line 1, in <module>
    print(eng2sp['five'])
KeyError: 'five'
```

Εικόνα 6.123 Εκτύπωση τιμής που αντιστοιχεί σε κλειδί

6.8.1 Πράξεις στα λεξικά

Ο τελεστής `del` αφαιρεί ένα ζευγάρι κλειδιού-τιμής από ένα λεξικό. Για παράδειγμα, στην Εικόνα 6.124 το λεξικό που δημιουργούμε, περιέχει ονόματα φρούτων καθώς και την πληροφορία πόσα από το καθένα έχουμε αποθηκευμένα. Έχουμε μήλα 430, μπανάνες 312, πορτοκάλια 525 και αχλάδια 217.

```
>>> inventory = {'apples':430, 'bananas': 312, 'oranges': 525, 'pears':217}
>>> print(inventory)
{'pears': 217, 'apples': 430, 'oranges': 525, 'bananas': 312}
```

Εικόνα 6.124 Λεξικό με ονόματα φρούτων

Αν κάποιος αγοράσει όλα τα αχλάδια μας, τότε μπορούμε να αφαιρέσουμε αυτήν την εγγραφή από τον κατάλογό μας, όπως φαίνεται στην Εικόνα 6.125. Αν ξανατυπώσουμε το λεξικό, θα δούμε ότι δεν περιέχει πια αχλάδια.

```
>>> del inventory['pears']
>>> print(inventory)
{'apples': 430, 'oranges': 525, 'bananas': 312}
```

Εικόνα 6.125 Ο τελεστής `del`

Αν θέλουμε μπορούμε να περιμένουμε μέχρι να μάς φέρουν νέα αχλάδια, οπότε δε βγάζουμε από το λεξικό μας το ζευγάρι αυτό, αλλά, απλώς, μηδενίζουμε το νούμερό τους, δείχνοντας ότι τελείωσαν τα αχλάδια και, επομένως, περιμένουμε άλλα. Κι όταν έρθουν νέα, προφανώς, βάζουμε το νούμερο το οποίο μας ήρθε. Βλέπετε ότι αυτός είναι ένας τρόπος να εισαγάγει κανείς μια καινούρια τιμή και αυτό δείχνει, επίσης, ότι τα λεξικά είναι μετατρέψιμα:

```
>>> inventory['pears']=0
>>> print(inventory)
{'pears': 0, 'apples': 430, 'oranges': 525, 'bananas': 312}
```

Εικόνα 6.126 Προσθήκη ζεύγους κλειδιού-τιμής σε λεξικό

Στα λεξικά έχουμε και την έννοια του μήκους, πόσα στοιχεία, πόσα ζευγάρια τέτοια κλειδιών-τιμών έχουμε μέσα σε αυτό, καθώς και τη δυνατότητα να ελέγξουμε αν υπάρχει ένα κλειδί μέσα στο λεξικό μας με τον τελεστή **in**. Η συνάρτηση **len** επιστρέφει τον αριθμό των ζευγαριών κλειδιών-τιμών του λεξικού:

```
>>> print(len(inventory))
4
```

Εικόνα 6.127 Η συνάρτηση *len* σε λεξικό

Ο τελεστής **in** αντίστοιχα ελέγχει αν υπάρχει ένα κλειδί (και όχι τιμή) σε ένα λεξικό:

```
>>> 'apples' in inventory
True
```

Εικόνα 6.128 Ο τελεστής *in* σε λεξικό

6.8.2 Μέθοδοι στα λεξικά

Στη συνέχεια θα δούμε μερικές χρήσιμες μεθόδους τις οποίες πρέπει οπωσδήποτε να θυμόμαστε αναφορικά με τα λεξικά. Όπως έχουμε αναφέρει, οι **μέθοδοι** είναι παρόμοιες με τις συναρτήσεις (παίρνουν ορίσματα και επιστρέφουν τιμές), αλλά η σύνταξή τους είναι διαφορετική (με βάση το dot notation). Μέσω του **help (dict)** μπορούμε να βρούμε τις μεθόδους που προσφέρει η Python για τα λεξικά. Μια σημαντική έννοια στα λεξικά είναι αυτή της *εικόνας* (view), η οποία προσφέρει ένα διαφορετικό τρόπο να βλέπει κανείς το περιεχόμενο του λεξικού. Θα δούμε τρεις διαφορετικές εικόνες: αυτή των κλειδιών (keys) (αναπαρίσταται σαν μια λίστα από όλα τα κλειδιά), αυτή των τιμών (values) (μια λίστα από όλες τις τιμές), και των πραγμάτων (items), η οποία είναι μια λίστα από πλειάδες (τα ζευγάρια μέσα στο λεξικό). Αυτές τις τρεις εικόνες μπορούμε να τις αποκτήσουμε επικαλούμενοι τις μεθόδους **keys**, **values**, **items**:


```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
>>> keys_view = eng2sp.keys()
>>> print(keys_view)
['three', 'two', 'one']
>>> values_view = eng2sp.values()
>>> print(values_view)
['tres', 'dos', 'uno']
>>> items_view = eng2sp.items()
>>> print(items_view)
[('three', 'tres'), ('two', 'dos'), ('one', 'uno')]
```

Εικόνα 6.129 Οι μέθοδοι *keys*, *values*, *items* σε λεξικό

Μπορούμε να χρησιμοποιήσουμε το γνωστό μας βρόγχο *for* για να τυπώσουμε τα κλειδιά, τις τιμές και τα ζεύγη κλειδιών-τιμών ενός λεξικού. Βάσει πάλι του παραδείγματος του αγγλο-ισπανικού λεξικού (όπως φαίνεται στην Εικόνα 6.130) ορίζουμε αρχικά τις μεταβλητές που αντιστοιχούν στα κλειδιά, τις τιμές, και τα ζευγάρια κλειδιών-τιμών. Ας προσέξουμε εδώ ότι οι μέθοδοι *keys()*, *values()*, και *items()*, παράγουν νέους τύπους της Python (όχι τις γνωστές μας λίστες), οι οποίοι έχουν ιδιάζουσα συμπεριφορά. Φαινομενικά, αυτοί μοιάζουν με λίστες, αλλά διαφέρουν στο ότι δεν μπορούμε να επιλέξουμε συγκεκριμένα στοιχεία μέσω ενός ακέραιου δείκτη, π.χ., το 0, 1, 2, κτλ. Μπορούμε, όμως, να διατρέξουμε αυτές τις «περίπου λίστες» σε ένα *for loop*, όπως φαίνεται στο παράδειγμα 6.130:


```

6-dictionary_for_function.py
1  eng2sp = {'one':'uno','two':'dos','three':'tres'}
2  keys_view = eng2sp.keys()
3  values_view = eng2sp.values()
4  items_view = eng2sp.items()
5
6  for key in keys_view:
7      print(key)
8
9  for value in values_view:
10     print(value)
11
12 for key,value in items_view:
13     print key,value

```

Εκτέλεση

```

>>>
three
two
one
tres
dos
uno
three tres
two dos
one uno

```

Εικόνα 6.130 Χρήση της *for* σε λεξικό

Μπορούμε να μετατρέψουμε αυτούς τους τύπους σε λίστες βάσει της συνάρτησης-γεννήτριας *list*, όπως φαίνεται στην Εικόνα 6.131:

```

>>> eng2sp = {'one':'uno','two':'dos','three':'tres'}
>>> keys_list = list(eng2sp.keys())
>>> print(keys_list)
['three', 'two', 'one']
>>> values_list = list(eng2sp.values())
>>> print(values_list)
['tres', 'dos', 'uno']
>>> items_list = list(eng2sp.items())
>>> print(items_list)
[('three', 'tres'), ('two', 'dos'), ('one', 'uno')]

```

Εικόνα 6.131 Χρήση της συνάρτησης *list* για αποθήκευση πληροφορίας ενός λεξικού

Μπορείτε να αντλήσετε πληροφορίες για άλλες μεθόδους των λεξικών μέσω της εντολής

```
>>> help(dict)
```

6.9 Ψευδώνυμα και αντιγραφή

Σ' αυτό το σημείο επανερχόμαστε στο σημαντικό θέμα των αντιγράφων και των ψευδωνύμων αλλά και της πληροφορίας: πότε χρησιμοποιούμε το ένα και πότε το άλλο. Αυτό έχει μεγάλη σημασία όταν αναφερόμαστε σε μετατρέψιμους τύπους δεδομένων, γιατί μπορεί εύκολα να γίνει πηγή λαθών αλλάζοντας μεταβλητές χωρίς να το καταλάβουμε.

Ας πάρουμε για παράδειγμα το λεξικό `opposites`, το οποίο περιέχει σαν στοιχεία αντίθετες έννοιες: «Πάνω-κάτω», «σωστό-λάθος», «αληθές-ψευδές». Η μεταβλητή `alias` ορίζεται σαν ψευδώνυμο του λεξικού `opposites`. Με άλλα λόγια, το `alias` και το `opposites` δείχνουν στην ίδια θέση της μνήμης. Πιο κάτω στο πρόγραμμα ορίζουμε τη μεταβλητή `copy`, η οποία είναι ένα αντίγραφο του λεξικού `opposites` και παράγεται βάσει της μεθόδου `copy()` που υπάρχει προκατασκευασμένη στην Python. Η `copy` εφαρμόζεται στο αντικείμενο `opposites` τύπου λεξικού και αντιγράφει το αντικείμενό της αυτό σε μια άλλη θέση της μνήμης. Έτσι, ενώ το `alias` δείχνει στο αντικείμενο `opposites`, το `copy` είναι ένα αντίγραφο το οποίο μπορεί να εξελιχθεί ανεξάρτητα μέσω ενημερώσεων.

```
>>> opposites = {'up':'down', 'right':'wrong', 'true':'false'}
>>> alias = opposites
>>> copy = opposites.copy()
```

Εικόνα 6.132 Η μέθοδος `copy` για αντιγραφή λεξικού

Αν, λοιπόν, χρησιμοποιήσουμε το `alias` για να αλλάξουμε την τιμή του κλειδιού `'right'` (απο `'wrong'` σε `'left'`), θα αλλάξει και το `opposites` —επειδή είναι το ίδιο αντικείμενο— όπως φαίνεται στο παρακάτω παράδειγμα:

```
>>> alias['right']='left'
>>> opposites['right']
'left'
```

Εικόνα 6.133 Χρήση ψευδώνυμου σε λεξικό

Αντίστροφα, αν αλλάξουμε την τιμή του κλειδιού `'right'` στο `copy`, το `opposites` παραμένει αμετάβλητο, επειδή είναι ανεξάρτητο αντίγραφο:

```
>>> copy['right']='privilege'
>>> opposites['right']
'left'
```

Εικόνα 6.134 Αλλαγή σε αντίγραφο λεξικού

Στη συνέχεια θα δούμε μια τελευταία εφαρμογή των λεξικών στους λεγόμενους *αραιούς πίνακες*, τους οποίους συναντάμε σε εφαρμογές των υπολογιστικών και εφαρμοσμένων μαθηματικών και σε μια περιοχή τους η οποία ονομάζεται Αριθμητική Ανάλυση.

6.10 Αραιοί πίνακες

Νωρίτερα σ' αυτό το κεφάλαιο είχαμε αναφέρει ότι ένας δισδιάστατος πίνακας μπορεί να αναπαρασταθεί ως μια λίστα λιστών. Σε πολλές περιοχές των εφαρμοσμένων και υπολογιστικών μαθηματικών συναντούμε τεράστιους πίνακες, των οποίων η διάσταση μπορεί να είναι εκατομμύρια επί εκατομμύρια στοιχεία εκ των οποίων τα πιο πολλά είναι μηδενικά, με ελάχιστα μόνο να είναι μη μηδενικά. Αυτοί οι πίνακες λέγονται *αραιοί πίνακες* (sparse matrices). Συνήθως, στις πράξεις επί τέτοιων πινάκων ενδιαφέρουν μόνον τα μη μηδενικά στοιχεία. Το κύριο πρόβλημα όταν αναπαριστούμε τέτοιους πίνακες ως λίστες λιστών είναι η σπατάλη μνήμης, λόγω της ανάγκης σε αυτήν την περίπτωση να αποθηκεύουμε όλα τα στοιχεία, ακόμα και τα μηδενικά τα οποία έχουν ελάχιστο ενδιαφέρον.

Παρακάτω βλέπουμε ένα παράδειγμα *αραιού πίνακα*, όπου οι πιο πολλές τιμές είναι 0:

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \end{bmatrix}$$

Εικόνα 6.135 Αραιός πίνακας

Η αναπαράσταση με λίστες περιέχει πολλά μηδενικά:

```
matrix = [ [0,0,0,1,0], [0,0,0,0,0], [0,2,0,0,0], [0,0,0,0,0], [0,0,0,3,0] ]
```

Μια εναλλακτική λύση είναι να θεωρήσουμε ότι υπάρχει μια έννοια συντεταγμένων των στοιχείων του πίνακα, όπως οι καρτεσιανές συντεταγμένες.

Εναλλακτικά, μπορούμε να χρησιμοποιήσουμε ένα λεξικό το οποίο για κλειδιά να χρησιμοποιεί πλειάδες οι οποίες περιέχουν τους αριθμούς γραμμών και στηλών των μη μηδενικών στοιχείων του πίνακα. Υποθέτουμε, βέβαια, και ότι όλα τα άλλα στοιχεία του πίνακα είναι μηδέν. Βλέπουμε στο παρακάτω παράδειγμα την αναπαράσταση του πίνακα σε αυτήν τη μορφή.

```
matrix = {(0,3): 1, (2, 1): 2, (4, 3): 3}
```

Σημειώστε ότι ακόμη δεν έχει σημασία η σειρά. Βάζουμε, λοιπόν, συντεταγμένη (0,3) τιμή 1, συντεταγμένη (2,1) τιμή 2, κλπ. Αυτό οδηγεί σε μια πολύ πιο συμπαγή αναπαράσταση ενός πίνακα, όπου τα πιο πολλά στοιχεία του είναι 0.

```
>>> matrix = {(0,3): 1, (2, 1): 2, (4, 3): 3}
```

Ωστόσο, στο θέμα της πρόσβασης στα στοιχεία του πίνακα αντιμετωπίζουμε ένα μειονέκτημα αυτής της αναπαράστασης. Για να το καταλάβουμε αυτό, ας ξεκινήσουμε παρατηρώντας ότι τα κλειδιά στο λεξικό `matrix` είναι πλειάδες. Επειδή στην Python μπορούμε να αναπαραστήσουμε μια πλειάδα με παρένθεση ή χωρίς, μπορούμε αντίστοιχα να έχουμε (0, 3) ή 0, 3. Οπότε, για να προσπελάσουμε το στοιχείο με συντεταγμένες (0, 3) χρησιμοποιούμε τον τελεστή επιλογής (`[]`) βάζοντας μέσα σε αυτόν την πλειάδα την οποία θέλουμε χωρίς παρένθεση, οπότε έχουμε:

```
>>> matrix[0,3]
1
```

Εικόνα 6.136 Πρόσβαση σε ένα στοιχείο πίνακα

Τι γίνεται, ωστόσο, αν χρησιμοποιήσουμε συντεταγμένη ενός στοιχείου το οποίο είναι μηδενικό και άρα δεν υπάρχει αντίστοιχο στοιχείο στη «συμπυκνωμένη»

έκδοση του `matrix` (π.χ. το `(1, 3)` όπως φαίνεται στην Εικόνα 6.137). Εφόσον το αντίστοιχο κλειδί δεν υπάρχει στο λεξικό μας, τότε ο διερμηνευτής θα μας δώσει μήνυμα λάθους.

```
>>> matrix[1,3]
Traceback (most recent call last):
  File "<pyshell#148>", line 1, in <module>
    matrix[1,3]
KeyError: (1, 3)
```

Εικόνα 6.137 Πρόβλημα πρόσβασης σε στοιχείο πίνακα

Θα μπορούσαμε, ενδεχομένως, να ανεχτούμε αυτές τις περιπτώσεις αντιμετωπίζοντας τα λάθη σαν *εξαιρέσεις*, οι οποίες εμφανίζονται σχετικά σπάνια και τις οποίες μπορούμε να διαχειριστούμε αντί να αφήσουμε να τερματίσει το πρόγραμμα. Ωστόσο, η Python μάς διευκολύνει εδώ (και έτσι αφήνουμε τη διαχείριση εξαιρέσεων για τις περιπτώσεις πραγματικά σπάνιων γεγονότων) προσφέροντάς μας μια ακόμα μέθοδο για τα λεξικά η οποία λέγεται `get` και η οποία λαμβάνει δύο ορίσματα: Το πρώτο όρισμα είναι η τιμή ενός κλειδιού και το δεύτερο είναι μια τιμή την οποία η `get` επιστρέφει αν το κλειδί που ορίσαμε σαν πρώτο όρισμα δεν υπάρχει στο λεξικό, διαφορετικά μάς επιστρέφει την τιμή του κλειδιού.

Στην Εικόνα 6.138 βλέπουμε ένα παράδειγμα στο οποίο η μέθοδος `get` καλείται για το κλειδί `(0, 3)` με δεύτερο όρισμα το `0`. Εφόσον το κλειδί υπάρχει στο λεξικό, η μέθοδος `get` μάς επιστρέφει την τιμή του κλειδιού (`1`):

```
>>> matrix.get((0,3),0)
1
```

Εικόνα 6.138 Η μέθοδος `get`

Στη Εικόνα 6.139 έχουμε την περίπτωση που το κλειδί δεν υπάρχει στο λεξικό. Εδώ η μέθοδος μάς επιστρέφει την τιμή την οποία έχουμε περάσει σαν δεύτερο όρισμα. Σαν αποτέλεσμα αποφεύγουμε το σφάλμα σε αυτήν την περίπτωση.

```
>>> matrix.get((1,3),0)
0
```

Εικόνα 6.139 Η μέθοδος `get`

6.11 Μετρώντας γράμματα

Μια άλλη χρήσιμη και ενδιαφέρουσα εφαρμογή των λεξικών είναι η δημιουργία ενός *ιστογράμματος* που μετρά πόσες φορές εμφανίζονται τα διάφορα γράμματα ή κάποιες λέξεις σε ένα κείμενο. Για παράδειγμα, το *a* μπορεί να εμφανίζεται 10 φορές, το *b* να εμφανίζεται 50 φορές, κλπ. Ένα ιστόγραμμα αναπαρίσταται με στήλες, μία για κάθε γράμμα – αν μετράμε συχνότητα εμφάνισης γραμμάτων – των οποίων το μήκος είναι ανάλογο με το πόσες φορές εμφανίζεται κάποιο γράμμα στο κείμενο. Για παράδειγμα στη λέξη “Mississippi”, θέλουμε να λέει ότι το *i* εμφανίζεται 4 φορές, το *s* εμφανίζεται επίσης 4 φορές, το *p*, 2 φορές, κτλ. Μια χρήση των ιστογραμμάτων είναι στη συμπίεση των δεδομένων. Χωρίς να εμβαθύνουμε περαιτέρω, μπορεί κανείς να μειώσει τον όγκο δεδομένων που χρειάζεται να μεταδοθεί ή να αποθηκευθεί για την ίδια ποσότητα πληροφορίας, αν η αναπαράσταση των πιο συχνά εμφανιζόμενων χαρακτήρων γίνει με λιγότερα bits συγκριτικά με τους χαρακτήρες οι οποίοι εμφανίζονται λιγότερο συχνά σε ένα συγκεκριμένο κείμενο.

Ένα ιστόγραμμα μπορεί να αναπαρασταθεί σαν ένα λεξικό που έχει σαν κλειδιά τα γράμματα και τιμές τη συχνότητα με την οποία εμφανίζεται κάθε γράμμα στο κείμενο. Στο παράδειγμα της Εικόνας 6.141 ξεκινάμε με ένα κενό λεξικό, (στους σύνθετους τύπους πρέπει να ξεκινήσουμε πάντα από κάποια αρχική τιμή). Στο βρόγχο *for*, η μεταβλητή *letter* διατρέχει τη συμβολοσειρά ‘Mississippi’, άρα παίρνει τιμές -χαρακτήρες πρώτα ‘m’, μετά ‘i’, μετά ‘s’, μετά ‘s’, κτλ. Για κάθε τιμή του *letter* με τη μέθοδο *get* διαβάζουμε την τρέχουσα συχνότητα εμφάνισης αυτού του χαρακτήρα (αν δεν υπάρχει ακόμα στο λεξικό, η *get* επιστρέφει 0). Στη συνέχεια αυξάνουμε τη συχνότητα εμφάνισης αυτού του χαρακτήρα κατά ένα. Αυτή είναι μια συνοπτική έκφραση. Αν δοκιμάσετε να το κάνετε διαφορετικά (π.χ. με ένα *while loop*, με χρήση τοπικών μεταβλητών) θα διαπιστώσετε ότι ο κώδικάς σας θα είναι λιγότερο συνοπτικός από αυτόν της Εικόνας 6.141.


```
6 letterCounts.py
1 letterCounts = {}
2
3 for letter in 'Mississippi':
4     letterCounts[letter]=letterCounts.get(letter,0) +1
5
6 print(letterCounts)
```

Εκτέλεση

```
>>>
{'i': 4, 'p': 2, 's': 4, 'M': 1}
```

Εικόνα 6.141 Ιστόγραμμα των γραμμάτων σε μια συμβολοσειρά

Ο αναγνώστης καλείται να παρακολουθήσει και το [διαδραστικό υλικό](#) που συνοδεύει αυτήν την ενότητα.

6.12 Επίλογος

Σε αυτό το κεφάλαιο εμβαθύναμε στην κατανόηση σημαντικών δομών δεδομένων, όπως οι συμβολοσειρές (strings), οι λίστες (lists), οι πλειάδες (tuples) και τα λεξικά (dictionaries), καθώς και της διαχείρισής τους. Επίσης γνωρίσαμε την έννοια της μεθόδου. Η μελέτη των δομών δεδομένων και των αλγορίθμων διαχείρισής τους έχει μακρά ιστορία και αποτελεί κεντρικό αντικείμενο μελέτης και διδασκαλίας της επιστήμης των υπολογιστών. Ο αναγνώστης που ενδιαφέρεται να εμβαθύνει περισσότερο σε αυτό το πεδίο καλείται να μελετήσει συγγράμματα όπως τα (Cormen, Leiserson, Rivest, & Stein, 2009) και (Miller & Ranum, 2005).

Βιβλιογραφία/Αναφορές

Cormen, T., Leiserson, C., Rivest, R., & Stein, C. (2009). *Introduction to algorithms* (Third Edition). MIT Press.

Miller, B. N., & Ranum, D. L. (2011). *Problem Solving with Algorithms and Data Structures Using Python* (Second Edition). Franklin, Beedle & Associates Inc.

Κριτήρια αξιολόγησης

Κριτήριο αξιολόγησης Ι (Βαθμός δυσκολίας: ●)

Τι σημαίνουν τα `fruit[:]` και `fruit[3:3]`;

Κριτήριο αξιολόγησης 2 (Βαθμός δυσκολίας: ●)

Τι θα εκτυπώσει ο παρακάτω κώδικας:

```
names = ['Jo', 'Yo', 'Lo']
```

```
names[0] = 100
```

```
print names
```

Κριτήριο αξιολόγησης 3 (Βαθμός δυσκολίας: ● ●)

Πότε θα επιλέξουμε να χρησιμοποιήσουμε μία πλειάδα;

Κριτήριο αξιολόγησης 4 (Βαθμός δυσκολίας: ● ●)

Κατασκευάστε έναν τηλεφωνικό κατάλογο με δύο εγγραφές, στον οποίο θα βρίσκουμε το όνομα του συνδρομητή με βάση το τηλέφωνό του

Αφού μελετήσετε τους παρακάτω αλγορίθμους ταξινόμησης οι οποίοι περιγράφονται σε αντίστοιχες διαδραστικές παρουσιάσεις, υλοποιήστε τους σε κώδικα Python

- [Bubble Sort](#)
- [Insertion Sort](#)
- [Merge Sort](#)
- [Quick Sort](#)
- [Selection Sort](#)