Κ. ΜΑΓΚΟΥΤΗΣ, Χ. ΝΙΚΟΛΑΟΥ

# ΕΙΣΑΓΩΓΗ ΣΤΟΝ ΑΝΤΙΚΕΙΜΕΝΟΣΤΡΑΦΗ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟ ΜΕ PYTHON

Μια προσέγγιση από την πλευρά
των υπολογιστικών συστημάτων

Ελληνικά Ακαδημαϊκά Ηλεκτρονικά Συγγράμματα και Βοηθήματα
www.kallipos.gr

HEALLINK
Σύνδεσμος Ελληνικών Ακαδημαϊκών Βιβλιοθηκών

Ευρωπαϊκή Ένωση
Ευρωπαϊκό Κοινωνικό Ταμείο

ΕΠΙΧΕΙΡΗΣΙΑΚΟ ΠΡΟΓΡΑΜΜΑ
ΕΚΠΑΙΔΕΥΣΗ ΚΑΙ ΔΙΑ ΒΙΟΥ ΜΑΘΗΣΗ
επένδυση στην κοινωνία της γνώσης
ΥΠΟΥΡΓΕΙΟ ΠΑΙΔΕΙΑΣ ΚΑΙ ΘΡΗΣΚΕΥΜΑΤΩΝ
ΕΙΔΙΚΗ ΥΠΗΡΕΣΙΑ ΔΙΑΧΕΙΡΙΣΗΣ

ΕΣΠΑ 2007-2013
πρόγραμμα για την ανάπτυξη
ΕΥΡΩΠΑΪΚΟ ΚΟΙΝΩΝΙΚΟ ΤΑΜΕΙΟ

Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης

**Κ. ΜΑΓΚΟΥΤΗΣ**

Τμήμα Μηχανικών Η/Υ και Πληροφορικής

Πανεπιστήμιο Ιωαννίνων

**Χ. ΝΙΚΟΛΑΟΥ**

Τμήμα Επιστήμης Υπολογιστών

Πανεπιστήμιο Κρήτης

# ΕΙΣΑΓΩΓΗ ΣΤΟΝ ΑΝΤΙΚΕΙΜΕΝΟΣΤΡΑΦΗ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟ ΜΕ PYTHON

## Μια Προσέγγιση από την Πλευρά των Υπολογιστικών Συστημάτων

Ελληνικά Ακαδημαϊκά Ηλεκτρονικά Συγγράμματα και Βοηθήματα
www.kallipos.gr

# ΕΙΣΑΓΩΓΗ ΣΤΟΝ ΑΝΤΙΚΕΙΜΕΝΟΣΤΡΑΦΗ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟ ΜΕ PYTHON


***Συγγραφή***

Κώστας Μαγκούτης (κύριος συγγραφέας), Πανεπιστήμιο Ιωαννίνων

Χρήστος Νικολάου, Πανεπιστήμιο Κρήτης

***Κριτικός αναγνώστης***

Αθηνά Βακάλη, Αριστοτέλειο Πανεπιστήμιο Θεσσαλονίκης

***Συντελεστές έκδοσης***

Γλωσσική επιμέλεια: Σοφία Σέγκουλη

Γραφιστική επιμέλεια: Κλειώ Καπαντώνη

Τεχνική επεξεργασία: Ιωάννης Παππάς





ΣΥΝΔΕΣΜΟΣ ΕΛΛΗΝΙΚΩΝ ΑΚΑΔΗΜΑΪΚΩΝ ΒΙΒΛΙΟΘΗΚΩΝ

Εθνικό Μετσόβιο Πολυτεχνείο

Ηρώων Πολυτεχνείου 9, 15780 Ζωγράφου

www.kallipos.gr

ISBN: 978-960-603-101-4

*Το σύγγραμμα αυτό είναι αφιερωμένο στη μνήμη του Χρήστου Νικολάου (1954-2015)*

# Περιεχόμενα

# ΕΙΣΑΓΩΓΗ ΣΤΟΝ ΑΝΤΙΚΕΙΜΕΝΟΣΤΡΑΦΗ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟ ΜΕ PYTHON

# ΚΕΦΑΛΑΙΟ 3

## Εκφράσεις, τελεστές, σχόλια

**3.1 Εισαγωγή**

**3.2 Τελεστές**

**3.3 Σχόλια**

**3.4 Επίλογος**

**Βιβλιογραφία/Αναφορές**

**Κριτήρια αξιολόγησης**

# ΚΕΦΑΛΑΙΟ 4

## Συναρτήσεις και εκτέλεση υπό συνθήκη

**4.1 Εισαγωγή**

**4.2 Εκτέλεση υπό συνθήκη, αναδρομή, είσοδος από το πληκτρολόγιο**

**4.3 Συναρτήσεις επιστροφής τιμής**

**4.4 Ανάπτυξη προγράμματος**

**4.5 Σύνθεση συναρτήσεων**

**4.6 Λογικές συναρτήσεις**

**4.7 Περισσότερη αναδρομή**

**4.8 Έλεγχος τύπων**

**4.9 Επίλογος**

**Βιβλιογραφία/Αναφορές**

**Κριτήρια αξιολόγησης**

# ΚΕΦΑΛΑΙΟ 5

## Επανάληψη με την εντολή while

**5.1 Εισαγωγή**

**5.2 Η εντολή while**

**5.3 Ενθυλάκωση και γενίκευση**

**5.4 Τοπικές μεταβλητές**

**5.5 Η εντολή break**

**5.6 Επίλογος**

**Βιβλιογραφία/Αναφορές**

**Κριτήρια αξιολόγησης**

# ΚΕΦΑΛΑΙΟ 6

## Συμβολοσειρές, λίστες, πλειάδες, λεξικά



# ΚΕΦΑΛΑΙΟ 7

## Αρχεία, εξαιρέσεις & εκσφαλμάτωση



# ΚΕΦΑΛΑΙΟ 8

## Κλάσεις και αντικείμενα

# ΚΕΦΑΛΑΙΟ II

## Εισαγωγή στον προγραμματισμό κατανεμημένων συστημάτων λογισμικού



# ΚΕΦΑΛΑΙΟ I2

## Τεχνολογικές τάσεις και μελλοντικές εξελίξεις

# Γλωσσάρι τεχνικών όρων

| | |
|---|---|
| Ακολουθία διαφυγής | Escape sequence |
| Αλληλουχία | Concatenation |
| Άλμευση | Pickling |
| Αμετάβλητο ή αμετάτρεπτο ή αμετάλλακτο | Immutable |
| Αμοιβαίος αποκλεισμός | Mutual exclusion |
| Αναδρομή | Recursion |
| Ανοικτός πηγαίος κώδικας | Open source |
| Αντικειμενοστρέφεια | Object orientation |
| Αποθετήριο | Repository |
| Ατέρμων βρόγχος | Infinite loop |
| Γνωρίσματα, ιδιότητες | Attributes |
| Διάγραμμα στοίβας | Stack trace |
| Εκσφαλμάτωση | Debugging |
| Έκφραση | Expression |
| Εκχώρηση | Assignment |
| Εμφωλευμένες συνθήκες | Nested conditionals |
| Εμφωλευμένη λίστα | Nested list |
| Ενθυλάκωση | Encapsulation |
| Ενσωματωμένη συνάρτηση | Built-in function |
| Εντολή | Statement |
| Εξαίρεση | Exception |
| Εσοχή | Indentation |
| Ευρετήριο | Dictionary |
| Ίχνος | Traceback |

| | |
|---|---|
| Καρποφόρα συνάρτηση | Fruitful function |
| Κληρονομικότητα | Inheritance |
| Κρίσιμη περιοχή | Critical section |
| Λάθος σημαντικής | Semantic error |
| Λέξη-κλειδί | Keyword |
| Μεταβλητή συνθήκης | Condition variable |
| Οδηγούμενο από τα γεγονότα | Event driven |
| Οριοθέτης | Delimiter |
| Όρισμα | Argument |
| Πακέτο | Module |
| Παραγοντικό | Factorial |
| Πλαίσιο | Frame |
| Πλαίσιο | Framework |
| Πλειάδα | Tuple |
| Πολυεκπομπή | Multicast |
| Ρηχή/βαθιά ισότητα | Shallow/deep equality |
| Σημαιοφόρος | Semaphore |
| Συμβολοσειρά | String |
| Ταυτότητα | Identifier |
| Τελεστής | Operator |
| Τροποποιητής | Modifier |
| Υπερφόρτωση τελεστή | Operator overloading |
| Υπόσταση, υποστασιασμός | Instance, instantiation |
| Φέτα | Slice |
| Ψευδώνυμο | Alias |

# Πρόλογος

Ο σκοπός αυτού του συγγράμματος είναι να διδάξει σε προπτυχιακούς φοιτητές Μηχανικών Η/Υ, Επιστήμης Υπολογιστών και συναφών σχολών τις βασικές αρχές του προγραμματισμού λογισμικού με γλώσσες υψηλού επιπέδου, χρησιμοποιώντας τη γλώσσα Python. Το σύγγραμμα εισάγει τους φοιτητές στις αρχές του αντικειμενοστραφούς προγραμματισμού αλλά και σε θέματα πιο προχωρημένου επιπέδου, όπως ο προγραμματισμός συστημάτων με έμφαση σε τεχνικές συγχρονισμού πολλαπλών διεργασιών/νημάτων, τον δικτυακό προγραμματισμό και εισαγωγικά θέματα προγραμματισμού κατανεμημένων συστημάτων.

Η γλώσσα Python είναι μια νέα γλώσσα – πρωτοεμφανίστηκε στις αρχές της δεκαετίας του 1990 – η οποία, όπως και οι Java και C++, είναι αντικειμενοστραφής. Μια σημαντική της διαφορά από άλλες γλώσσες υψηλού επιπέδου είναι η *συνοπτικότητά* της, στην οποία οφείλεται η σημαντική *αναγνωσιμότητα* των προγραμμάτων της. Αν και δεν έχει χρησιμοποιηθεί σε ευρεία κλίμακα για την παραγωγή κώδικα συστημάτων μεγάλης κλίμακας στο παρελθόν, η Python παρέχει όλες τις απαραίτητες προϋποθέσεις για κάτι τέτοιο. Λόγω των προτερημάτων της σε σχέση με άλλες γλώσσες υψηλού επιπέδου, προσφέρεται ιδιαίτερα και για τη διδασκαλία αρχών προγραμματισμού συστημάτων.

Τα κεφάλαια 1-8 αυτού του συγγράμματος συγκεντρώνουν διδακτική εμπειρία του μαθήματος *ΗΥ-100 Εισαγωγή στην Επιστήμη Υπολογιστών*, το οποίο διδάχθηκε από τον Καθηγητή Χρήστο Νικολάου στο Τμήμα Επιστήμης Υπολογιστών του Πανεπιστημίου Κρήτης από το 2005 ως το 2010. Η διδασκαλία αυτού του μαθήματος σε περίπου 150 πρωτοετείς φοιτητές, ανά έτος, παρήγαγε πλούσιο διδακτικό υλικό (διαφάνειες, απομαγνητοφωνημένες σημειώσεις, διαδραστικές εφαρμογές κ.λπ.) και αποτέλεσε τη βάση για τη συγγραφή των κεφαλαίων 1-8. Κατά τον αρχικό σχεδιασμό των διαλέξεων αυτών χρησιμοποιήθηκε η οργάνωση καθώς και υλικό από το βιβλίο *Think Python* του Allen B. Downey[1]. Η τελική μορφή των κεφαλαίων 1-8 λαμβάνει υπόψη στοιχεία αυτού του βιβλίου, αλλά διαφοροποιείται με βάση τις

---

1 Allen B. Downey, “Think Python: How to Think Like a Computer Scientist”, δωρεάν διαθέσιμο σύγγραμμα στο http://greenteapress.com/thinkpython, και από τον εκδοτικό οίκο O’Reilly στην αγγλική γλώσσα, υπό την άδεια Creative Commons Attribution-NonCommercial 3.0 Unported License.

ανάγκες και τους στόχους του παρόντος συγγράμματος· βασίστηκε εξ ολοκλήρου στην απομαγνητοφώνηση διαλέξεων, ενώ διαμορφώθηκε με βάση την εμπειρία της διδασκαλίας του ΗΥ-100, επί σειρά πέντε ετών.

Τα κεφάλαια 9-12 συγκεντρώνουν διδακτική εμπειρία του Επίκ. Καθηγητή, Κώστα Μαγκούτη, από τη διδασκαλία των μαθημάτων ΜΥΕ-017 *Κατανεμημένα Συστήματα* και ΜΥΥ-801 *Δίκτυα Υπολογιστών ΙΙ* στο Πανεπιστήμιο Ιωαννίνων (2014-2015) και ΗΥ-335 Δίκτυα Υπολογιστών (2009-2013), ΗΥ-559 *Τεχνολογίες Υποδομής για Υπηρεσιοκεντρικά Συστήματα Μεγάλης Κλίμακας* και ΗΥ-590.45 *Προχωρημένα Θέματα σε Κλιμακώσιμα Συστήματα Αποθήκευσης* (2008-2011) στο Πανεπιστήμιο Κρήτης. Η εμπειρία διδασκαλίας με βάση την ύλη αυτού του συγγράμματος δείχνει οτι, δύσκολες έννοιες που παραδοσιακά διδάσκονταν κυρίως θεωρητικά, μπορούν με το κατάλληλο προγραμματιστικό περιβάλλον να διδαχθούν και εργαστηριακά στα χρονικά περιθώρια ενός εξαμηνιαίου μαθήματος.

Ο στόχος των συγγραφέων ξεκινώντας την προσπάθεια συγγραφής ήταν να παρουσιάσουν με απλό τρόπο βασικές έννοιες της Python, του αντικειμενοστραφούς προγραμματισμού, και του προγραμματισμού συστημάτων, προσφέροντας ένα εργαλείο μάθησης στην ελληνόφωνη ακαδημαϊκή κοινότητα. Όπως κάθε συγγραφική προσπάθεια που υλοποιείται σε συγκεκριμένα χρονικά περιθώρια, η έκδοση αυτή θα χρειαστεί να ενημερωθεί και βελτιωθεί εν ευθέτω χρόνω. Κάθε επικοινωνία από τους αναγνώστες σχετικά με τυχόν λάθη, παραλείψεις ή άλλες προτάσεις είναι ευπρόσδεκτη και θα ληφθεί υπόψη σε μελλοντικές εκδόσεις.

Κώστας Μαγκούτης[2]

Ιωάννινα, Οκτώβριος 2015

---

2 Στοιχεία επικοινωνίας: Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, 45110 Ιωάννινα, Email: magoutis@cse.uoi.gr, Τηλ. 2651008871

# ΚΕΦΑΛΑΙΟ Ι

# Εισαγωγή στην Python

*Σύνοψη*

*Σε αυτό το κεφάλαιο κάνουμε μια σύντομη εισαγωγή στην Python και στα εργαλεία λογισμικού που θα χρησιμοποιήσουμε στη συνέχεια του συγγράμματος.*

*Προαπαιτούμενη γνώση*

*Ενδείκνυται (ωστόσο δεν απαιτείται) προηγούμενη εμπειρία σε οποιαδήποτε άλλη γλώσσα υψηλού επιπέδου και εξοικείωση με περιβάλλοντα προγραμματισμού.*

## Ι.Ι Εισαγωγή

Η Python (Python Software Foundation, 2011) είναι γλώσσα προγραμματισμού υψηλού επιπέδου, ανοικτού πηγαίου κώδικα (open source) και γενικής χρήσης. Είναι εύκολη στην εκμάθηση (απλούστερη από τις C, C++, Java) και παρέχει ισχυρές δυνατότητες σε αρχάριους και έμπειρους προγραμματιστές. Ένα από τα κύρια χαρακτηριστικά της είναι η αντικειμενοστρέφεια, στην οποία αναφερόμαστε εκτενώς σε αυτό το σύγγραμμα και ιδιαίτερα στο Κεφάλαιο 8. Αναπτύχθηκε από τον Guido van Rossum, στις αρχές της δεκαετίας '90 ως διάδοχος της γλώσσας ABC και το όνομά της προέρχεται από την ομάδα κωμικών Monty Python. Η βασική ιστοσελίδα της κοινότητας γύρω από τη γλώσσα βρίσκεται στην ηλεκτρονική διεύθυνση http://python.org (Python Software Foundation).

**Εικόνα Ι.Ι** *Λογότυπο της Python*

Η Python χρησιμοποιείται, διεθνώς, ως εκπαιδευτική γλώσσα αλλά και στην ανάπτυξη σημαντικών εφαρμογών με την ίδια επιτυχία. Εκδόσεις της είναι διαθέσιμες για μια ευρεία γκάμα απο λειτουργικά συστήματα, συμπεριλαμβανομένων των Windows, Unix/Linux, Mac OS X, iPod, κλπ. Είναι επεκτάσιμη μέσω ενσωματωμένων βιβλιοθηκών, δηλαδή, μπορεί να υλοποιηθεί σημαντική λειτουργικότητα χωρίς να χρειαστεί να αναζητηθούν επεκτάσεις, όπως μεταφορικά λέγεται στα αγγλικά, 'batteries are included'.

Ας εστιάσουμε καταρχήν στη συνοπτικότητά της. Στις Εικόνες Ι.2-Ι.5 βλέπετε τέσσερα παραδείγματα του προγράμματος που παραδοσιακά γράφουμε ως πρώτο, όταν μαθαίνουμε οποιαδήποτε γλώσσα προγραμματισμού, απο τις δημοφιλείς γλώσσες προγραμματισμού C, C++, Java, και Python. Είναι εμφανές ότι η Python προσφέρει τη συνοπτικότερη, απλούστερη στην κατανόηση έκφραση.

C

```
#include <stdio.h>

main()
{
  printf("Hello world!");
}
```

**Εικόνα Ι.2** *Πρώτο πρόγραμμα στη γλώσσα προγραμματισμού C*

C++

```
#include <iostream.h>
void main( )
{
  cout << "Hello world!";
}
```

**Εικόνα Ι.3** *Πρώτο πρόγραμμα στη γλώσσα προγραμματισμού C++*

Java

```
public class HelloWorld {
  public static void main(String[ ] args) {
    System.out.println("Hello World!");
  }
}
```

**Εικόνα Ι.4** *Πρώτο πρόγραμμα στη γλώσσα προγραμματισμού Java*

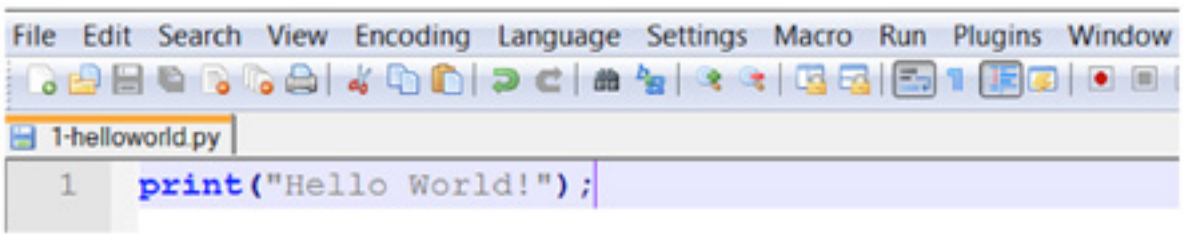
Python

```
print("Hello World!")
```

**Εικόνα Ι.5** *Πρώτο πρόγραμμα στη γλώσσα προγραμματισμού Python*

## I.2 Προκαταρτικές πληροφορίες

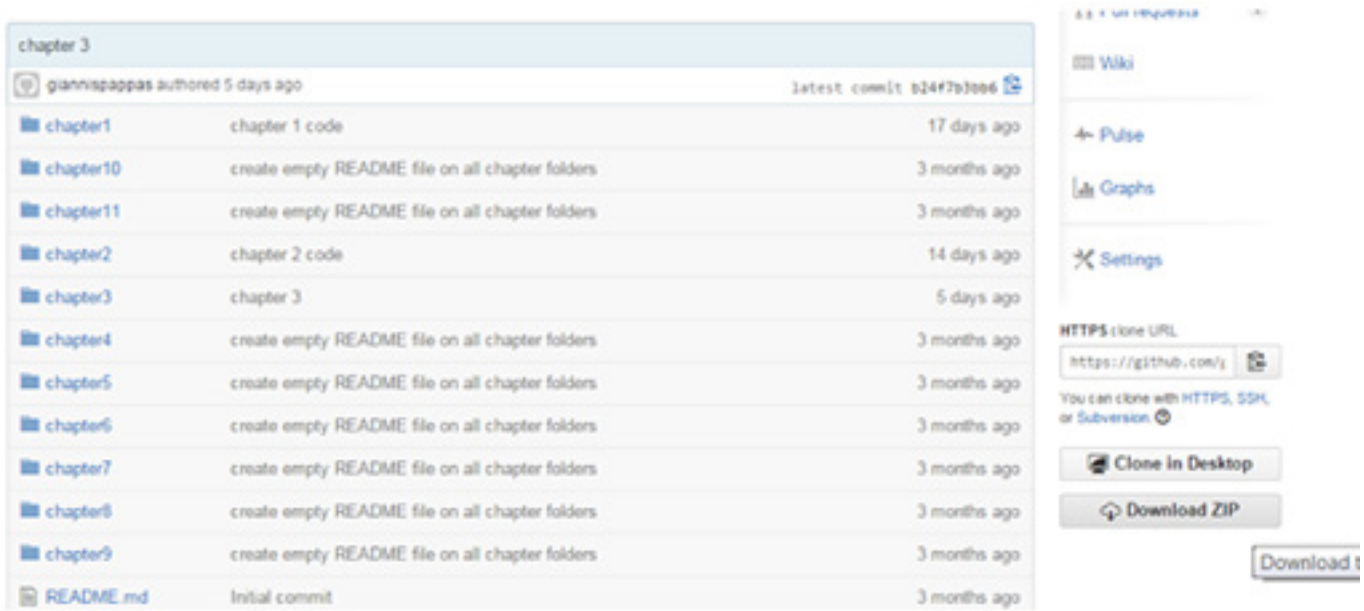
Σ' αυτό το σύγγραμμα θα χρησιμοποιήσουμε τον συντάκτη κειμένου (editor) Notepad++ (https://notepad-plus-plus.org) ο οποίος μπορεί να αναγνωρίσει τη σύνταξη και δομή των προγραμμάτων Python, με τη ρύθμιση Language, Python στο κύριο μενού του. Στην Εικόνα I.6 βλέπουμε ένα παράδειγμα της χρήσης του Notepad++, για να γράψουμε το πρώτο μας πρόγραμμα σε Python.




**Εικόνα I.6** *Πρώτο πρόγραμμα στη γλώσσα προγραμματισμού Python με χρήση του Notepad++*

Ο πηγαίος κώδικας ο οποίος χρησιμοποιείται στα παραδείγματα αυτού του συγγράμματος είναι διαθέσιμος στο αποθετήριο που βρίσκεται στην ηλεκτρονική διεύθυνση https://github.com/giannispappas/K_book.git.

Για κατεβάσετε τον πηγαίο κώδικα των παραδειγμάτων του συγγράμματος προτείνονται δύο τρόποι. Ο πρώτος είναι να πάτε στη σελίδα https://github.com/giannispappas/K_book και να επιλέξετε το εικονίδιο που βρίσκεται δεξιά στο κάτω μέρος της σελίδας και γράφει: "Download ZIP", όπως φαίνεται και στην Εικόνα I.7. Με τον τρόπο αυτό θα έχετε όλο τον κώδικα του συγγράμματος ως ένα συμπιεσμένο αρχείο.




**Εικόνα Ι.7** *Κατέβασμα κώδικα βιβλίου ως συμπιεσμένο αρχείο*

Ο δεύτερος τρόπος είναι να προσπελάσετε το αποθετήριο της υπηρεσίας GitHub (http://github.com) με κάποιο πρόγραμμα διαχείρισης κώδικα ανοικτού λογισμικού που χρησιμοποιεί το πρωτόκολλο git. Αρχικά πρέπει να εγκαταστήσετε ένα πρόγραμμα git στον υπολογιστή σας ακολουθώντας τις οδηγίες αυτού του συνδέσμου. Μετά την επιτυχή εγκατάσταση του προγράμματος πρέπει να ανοίξετε μια γραμμή εντολών (command prompt), να πληκτρολογήσετε "git clone", και στη συνέχεια το σύνδεσμο όπου βρίσκεται ο κώδικας του βιβλίου, πατώντας Enter. Η εντολή και το αποτέλεσμα φαίνονται στην Εικόνα Ι.8.

```
C:\Users\IP\Documents\GitHub> git clone http://github.com/giannispappas/K_book
Cloning into 'K_book'...
remote: Counting objects: 51, done.
remote: Total 51 (delta 0), reused 0 (delta 0), pack-reused 51
Unpacking objects: 100% (51/51), done.
Checking connectivity... done.
C:\Users\IP\Documents\GitHub>
```

**Εικόνα Ι.8** *Κατέβασμα του κώδικα που συνοδεύει αυτό το σύγγραμμα ως συμπιεσμένο αρχείο*

Με τον τρόπο αυτό έχετε ένα τοπικό αντίγραφο του κώδικα του συγγράμματος στον υπολογιστή σας. Περισσότερες πληροφορίες σχετικά με την εγκατάσταση

του προγράμματος git, και τη δημιουργία και χρήση αποθετηρίων στη διαδικτυακή υπηρεσία GitHub θα βρείτε στον εξής σύνδεσμο.

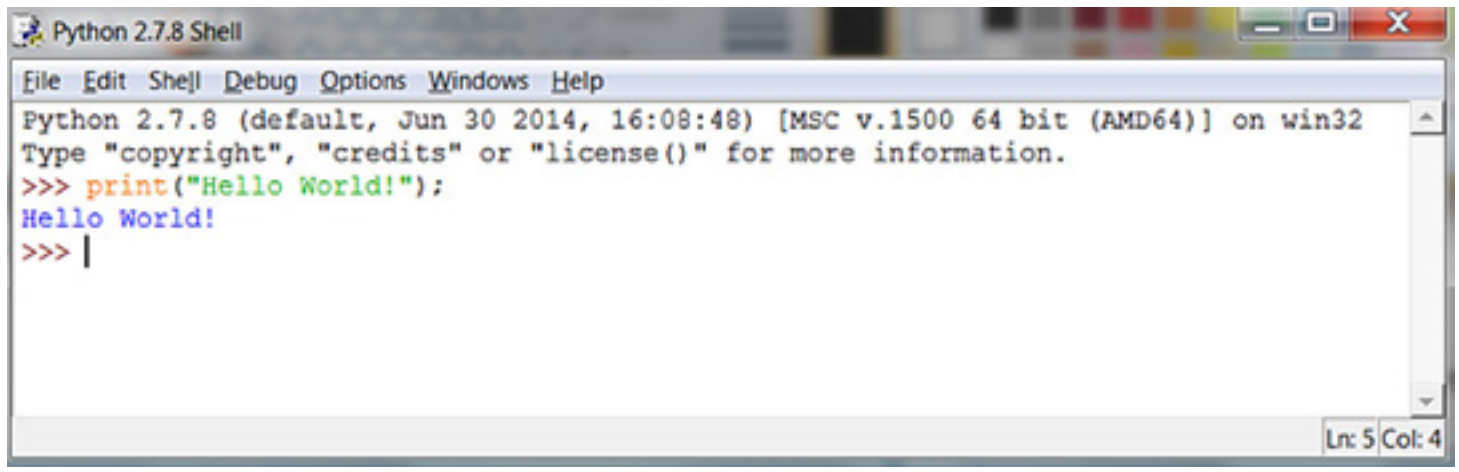
## I.3 Ο διερμηνέας της Python

Η έκδοση Python που επιλέξαμε να χρησιμοποιήσουμε σε αυτό το σύγγραμμα είναι η 2.7.8 την οποία μπορείτε να βρείτε στον εξής σύνδεσμο: https://www.python.org/download/releases/2.7.8/ . Στην Εικόνα I.9 βλέπετε ένα στιγμιότυπο αυτής της ιστοσελίδας.


**Εικόνα I.9** *Ιστοσελίδα της έκδοσης 2.7.8 της Python*

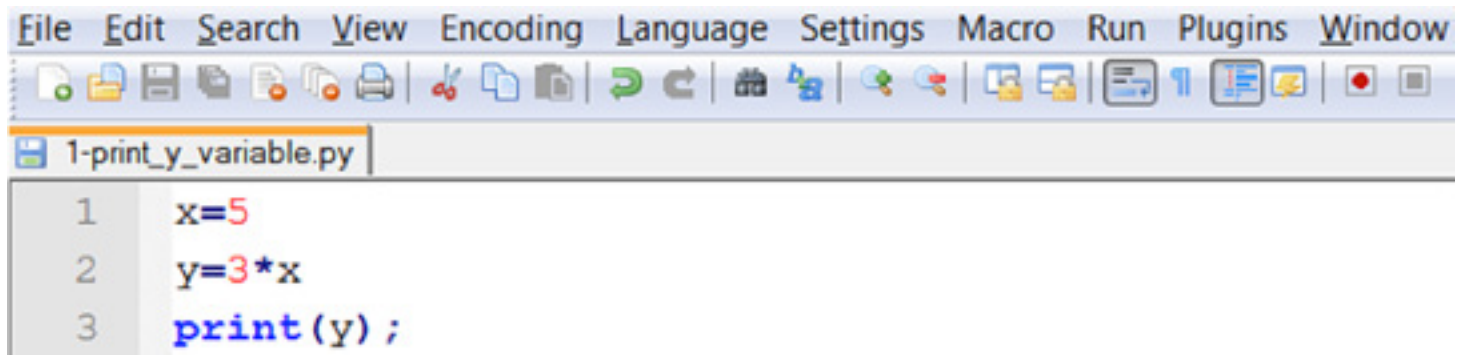
Το περιβάλλον ανάπτυξης και εκτέλεσης προγραμμάτων περιλαμβάνει το διερμηνέα (interpreter) της Python ο οποίος διαβάζει και εκτελεί προγράμματα Python. Ενώ ο διερμηνέας μπορεί να καλεστεί απευθείας σε μια γραμμή εντολής (command line), συνήθως χρησιμοποιούμε ολοκληρωμένα περιβάλλοντα ανάπτυξης (integrated

development environments ή IDEs) τα οποία μας προσφέρουν γραφικό περιβάλλον και υποστήριξη στον προγραμματισμό, εκτέλεση, και εκσφαλμάτωση (debugging) προγραμμάτων. Σε αυτό το σύγγραμμα θα χρησιμοποιήσουμε ένα απλό αλλά αρκετά διαδεδομένο τέτοιο περιβάλλον το οποίο συμπεριλαμβάνεται στη βασική διανομή της Python και ονομάζεται IDLE (Integrated DeveLopment Environment). Στην Εικόνα I.10 βλέπουμε το παράθυρο του IDLE της έκδοσης 2.7.8. Στα Windows μπορούμε να βρούμε το IDLE ακολουθώντας Έναρξη → Προγράμματα → Python 2.7 → IDLE (Python GUI).




**Εικόνα I.10** *IDLE για τη γλώσσα Python σε περιβάλλον Windows*

Στην Εικόνα I.11 ακολουθεί το δεύτερο πρόγραμμά μας στην Python. Το αρχείο που δημιουργούμε και στο οποίο γράφουμε τον κώδικα Python, το αποθηκεύουμε με κατάληξη .py.

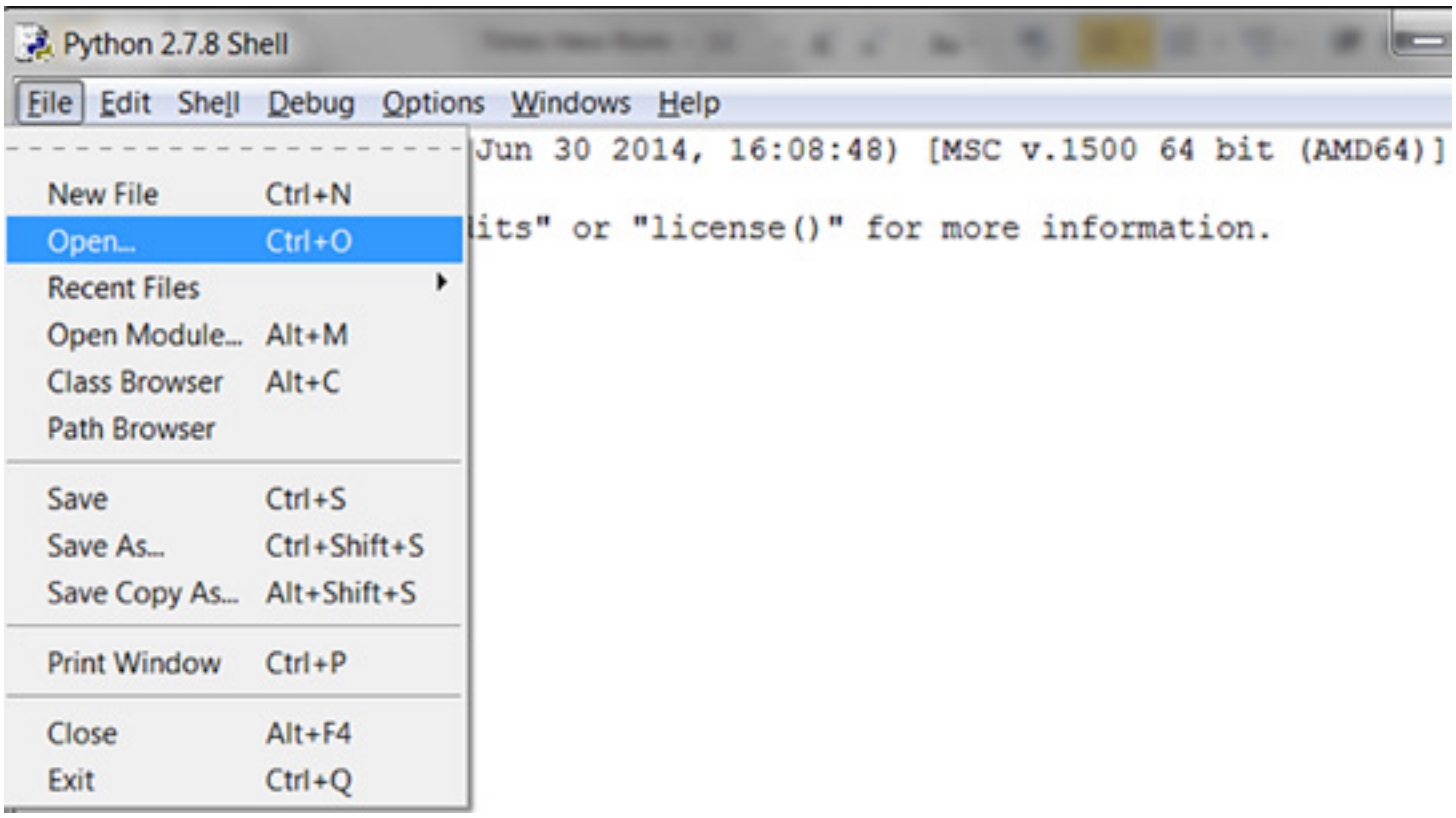



**Εικόνα I.11** *Το δεύτερό μας πρόγραμμα στην Python*

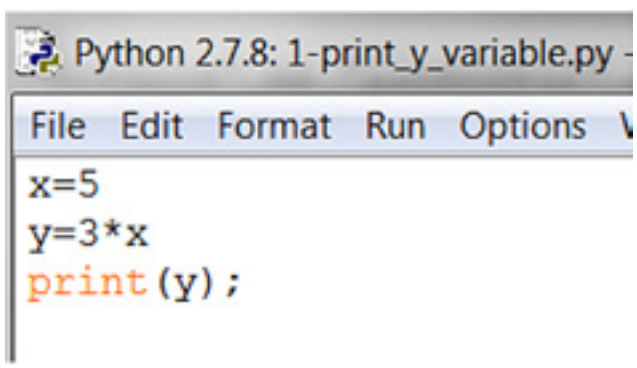
Για την εκτέλεση (ή όπως λέμε συνήθως, το «τρέξιμο») του προγράμματος που μόλις γράψαμε, ακολουθούμε τα εξής βήματα στο μενού του IDLE:

- Ανοίγουμε το αρχείο και το «φορτώνουμε» στο περιβάλλον επιλέγοντας File → Open
- Στη συνέχεια επιλέγουμε στο μενού Run → Run Module

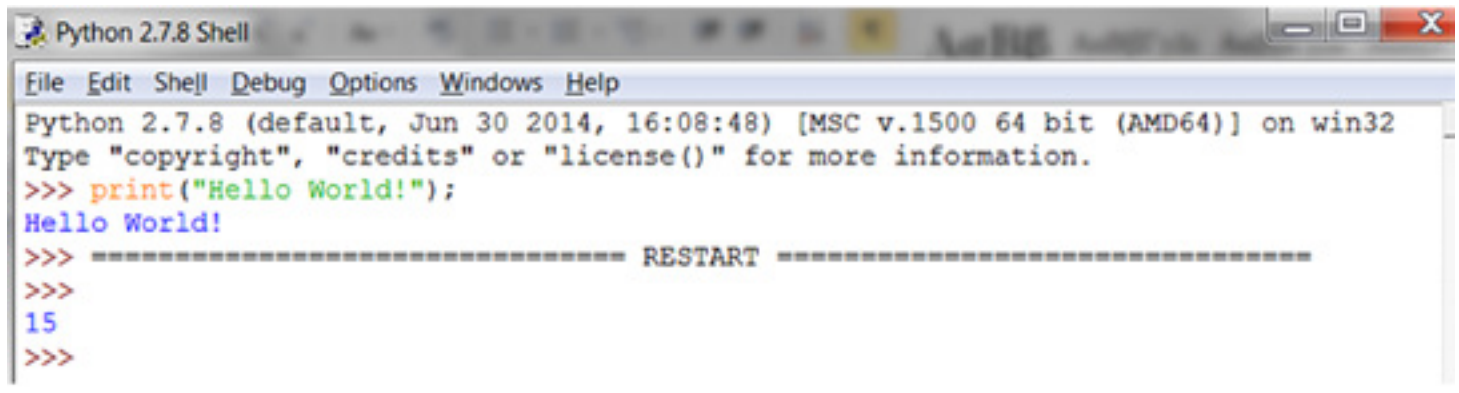
Το αποτέλεσμα φαίνεται στην Εικόνα I.14:




**Εικόνα I.12** *Άνοιγμα αρχείου προγράμματος*




**Εικόνα I.13** *Εκτέλεση προγράμματος*




**Εικόνα Ι.14** *Αποτέλεσμα εκτέλεσης προγράμματος*

## Ι.4 Επίλογος

Αυτό το κεφάλαιο αποτελεί εισαγωγή στα βασικά χαρακτηριστικά της γλώσσας Python και στα εργαλεία λογισμικού τα οποία θα χρησιμοποιήσουμε στη συνέχεια του παρόντος συγγράμματος. Στο Κεφάλαιο 2 θα δούμε βασικές έννοιες, απαραίτητες στα πρώτα βήματά μας στον προγραμματισμό σε Python: τιμές, τύπους, μεταβλητές, και πρώτες εντολές.

## Βιβλιογραφία/Αναφορές

Python Software Foundation. (2011). *Python*. Retrieved from: http://python.org

# Κριτήρια αξιολόγησης

## Κριτήριο αξιολόγησης Ι (Βαθμός δυσκολίας[1]: ●)

**Για να εξασφαλίσουμε ότι εργάζεστε σωστά με τον Python interpreter, δοκιμάστε τα εξής βήματα στον υπολογιστή σας:**

**Ξεκινήστε τον Python interpreter σε διαδραστική μορφή (IDLE)**

**Στην προτροπή χρήστη >>> γράψτε την ακόλουθη δήλωση και μετά πατήστε Enter:**

```
print('Αυτό είναι ένα τεστ για τον Python interpreter.')
```

**Απάντηση/Λύση**

1 Ο βαθμός δυσκολίας των ασκήσεων κυμαίνεται μεταξύ των: χαμηλός (●), μέτριος (●●), και υψηλός (●●●).

## Κριτήριο αξιολόγησης 2 (Βαθμός δυσκολίας: ●)

**Γράψτε το εξής κείμενο σε ένα αρχείο με όνομα ex1.py και εκτελέστε το.**

```
print “Hello World!”
print “Hello”
print “World!”
print “Hello Hello!”
```

**Απάντηση/Λύση**

# ΚΕΦΑΛΑΙΟ 2

## Τιμές, τύποι, μεταβλητές, λέξεις-κλειδιά, εντολές

*Σύνοψη*

*Το κεφάλαιο αυτό εισάγει τον αναγνώστη σε βασικές έννοιες που είναι χρήσιμες για την απόκτηση μιας πρώτης εμπειρίας με την Python.*

*Προαπαιτούμενη γνώση*

*Βασικές αρχές προγραμματισμού τις οποίες ο αναγνώστης μπορεί να έχει αποκτήσει από κάποιο εισαγωγικό βιβλίο σε γλώσσα προγραμματισμού ή/και από προσωπική εμπειρία και κατανόηση της ύλης του Κεφαλαίου Ι.*

### 2.1 Εισαγωγή

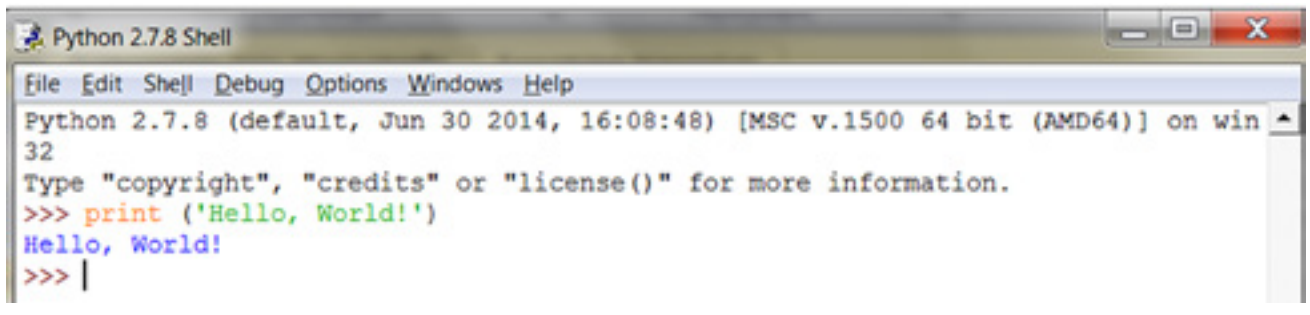
Η παραγωγή εκτελέσιμων προγραμμάτων από πηγαίο κώδικα (source code) σε οποιαδήποτε γλώσσα προγραμματισμού γίνεται μέσω μεταφραστών (compilers) ή διερμηνέων (interpreters). Ο μεταφραστής παίρνει τον πηγαίο κώδικα και τον μεταφράζει σε αντικειμενικό κώδικα (object code) ή κώδικα μηχανής (machine code) και μετά τρέχει το πρόγραμμα με τις συγκεκριμένες εισόδους, το οποίο, βεβαίως, παράγεται στις συγκεκριμένες εξόδους. Έχουμε, λοιπόν, δύο φάσεις: τη φάση της μετάφρασης, η οποία παράγει κώδικα σε γλώσσα μηχανής, ειδικότερα σε αντικειμενικό κώδικα, και τη φάση η οποία τρέχει το πρόγραμμα.


**Εικόνα 2.1** *Παραγωγή εκτελέσιμου προγράμματος μέσω διερμηνέα*

Η γλώσσα Python διαφέρει διότι χρησιμοποιεί διερμηνέα, δηλαδή, παίρνει κατευθείαν το πρόγραμμα που έχουμε γράψει και αμέσως το μεταφράζει και το τρέχει σε ένα στάδιο. Ακόμη βγάζει τις εισόδους όπου δίνονται τα αποτελέσματα. Οπότε στην Python συμπιέζονται οι δύο φάσεις σε μία.

Ας θυμηθούμε το πρώτο μας πρόγραμμα, το "Hello World", και το περιβάλλον ανάπτυξης προγραμμάτων (το οποίο περιέχει το διερμηνευτή) IDLE. Υποθέτουμε ότι ο αναγνώστης έχει ήδη κατεβάσει το IDLE της έκδοσης 2.7.8. Στη συνέχεια ας πειραματιστούμε με αυτό:

- Ανοίγουμε το IDLE
- Γράφουμε την εντολή print ('Hello, World!') και πατάμε το πλήκτρο Enter




**Εικόνα 2.2** *Το πρώτο μας πρόγραμμα*

## 2.2 Κυριολεκτικές σταθερές: Τιμές και τύποι

Η έννοια της τιμής αναφέρεται σε μεταβλητές αλλά και σε σταθερές τιμές. Επίσης, τιμές μπορεί να είναι οι αριθμοί αλλά και οι αριθμητικές εκφράσεις, όπως αν γράψετε 1+1. Ωστόσο, τιμή είναι και η έκφραση 'hello world'. Κάθε τιμή ανήκει σε κάποιον τύπο (ακέραιοι αριθμοί, ομάδες χαρακτήρων ή strings, κλπ.). Η ομάδα χαρακτήρων είναι ένας από τους θεμελιώδεις τύπους της Python και σταθερές του τύπου ομάδα χαρακτήρων (ή συμβολοσειρά) τις οποίες περικλείουμε με εισαγωγικά (μονά (') ή διπλά (")) αρχής και τέλους. Η εντολή εκτύπωσης δουλεύει, βεβαίως, με ακέραιους αριθμούς και με πολλούς άλλους τύπους, όπως θα δούμε αργότερα. Με τον όρο κυριολεκτικές σταθερές (literal constants) αναφερόμαστε σε προκαθορισμένες τιμές που παραμένουν αμετάβλητες σε όλη τη διάρκεια εκτέλεσης ενός προγράμματος.

**Παραδείγματα τιμών:**

- 3,
- 2.5,
- 'Hello, World!'

**Αυτές οι τιμές ανήκουν σε διαφορετικούς τύπους (types):**

- 3 είναι ένας ακέραιος αριθμός (integer).
- 2.5 είναι ένας αριθμός κινητής υποδιαστολής (float).
- 'Hello, World!' είναι μια συμβολοσειρά (string). Πρόκειται για μια ακολουθία από χαρακτήρες οι οποίοι περικλείονται σε μονά ή διπλά εισαγωγικά.
- Εκτύπωση με χρήση της εντολής print.

```
>>> print (3)
3
>>> print (2.5)
2.5
>>> print ('Hello, World!')
Hello, World!
```

**Εικόνα 2.3** *Εκτύπωση με χρήση της εντολής print*

Αν δεν είναι γνωστός ο τύπος μιας τιμής, μπορείτε να χρησιμοποιήσετε μια ενσωματωμένη (built-in) συνάρτηση του διερμηνευτή της Python, την type, ώστε να τον μάθετε. Ο interpreter βοηθά, επίσης, γιατί δείχνει τη σύνταξη του υπολοίπου· δηλαδή, αν αρχίσετε να τυπώνετε type (όπως στην Εικόνα 2.4), ο interpreter καταλαβαίνει ότι είναι η συνάρτηση type της Python και θα σας πει ποιες είναι οι δυνατότητες για τη συνέχεια αυτής της έκφρασης. Έπειτα, μετά το κλείσιμο της παρένθεσης αναφέρει τον τύπο του object. Για παράδειγμα, ας πούμε type (3). Όταν ο διερμηνέας καταλάβει ότι έκλεισε η παρένθεση, του δίνει γκρι χρώμα και απαντά ότι το type είναι 'int', δηλαδή, ακέραιος. Παρατηρούμε, λοιπόν, ότι ο διερμηνέας είναι αρκετά βοηθητικός.

```
>>> type (3)
<type 'int'>
>>> type (2.5)
<type 'float'>
>>> type ('Hello, World!')
<type 'str'>
```

**Εικόνα 2.4** *Ο διερμηνευτής επιστρέφει στο χρήστη τον τύπο μιας τιμής*

Εάν του λέγατε type (2.5), το type είναι 'float', ο τύπος των πραγματικών αριθμών που προέρχεται από την έκφραση floating point. Τι σημαίνει, όμως, floating point; Το point είναι η τέλεια που χρησιμοποιείται ως υποδιαστολή στις αγγλοσαξονικές χώρες. Ονομάζεται floating («επιπλέει»), επειδή ένας πραγματικός αριθμός (π.χ. το 2.5) μπορεί να εκφραστεί με πολλούς και διαφορετικούς τρόπους. Μπορεί για παράδειγμα να εκφραστεί σαν $0.25*10$, ή σαν $25*10^{-1}$, με δυνάμεις του 10. Επομένως, η υποδιαστολή μπορεί να μετακινηθεί ελεύθερα αριστερά ή δεξιά, δηλαδή να «επιπλέει», ανάλογα με τι δυνάμεις του 10 βάζουμε. Γι' αυτό οι τύποι των πραγματικών αριθμών λέγονται floats.

Είναι σημαντικό εδώ ο αναγνώστης να προσέξει ένα λεπτό σημείο: Ότι από τη μια υπάρχει ο ακέραιος 15, και από την άλλη υπάρχει μια ομάδα χαρακτήρων που είναι: το ψηφίο 1 και το ψηφίο 5 και περικλείονται από δύο απλά εισαγωγικά (‘), δηλαδή '15'. Η δεύτερη περίπτωση έχει τελείως διαφορετικό τύπο, είναι άλλη τιμή, διαφορετική από τον αριθμό 15. Γι' αυτό και όταν λέμε type '15', η απάντηση είναι string

και όχι int, όπως θα ήταν αν δε βάζαμε τα εισαγωγικά. Αντίστοιχα το ίδιο ισχύει και με τις τιμές 3.7 και '3.7'. Αν δε βάζαμε τα εισαγωγικά, θα έδινε ως αποτέλεσμα τον τύπο float. Παρακάτω δίνεται η εξήγηση των ακόλουθων τύπων:

Τύπος **int** → ακέραιοι αριθμοί
Τύπος **float** → αριθμοί κινητής υποδιαστολής (δεκαδικοί αριθμοί)
Τύπος **str** → συμβολοσειρές (ακολουθίες χαρακτήρων)

**Παράδειγμα:**
Τι τύπου είναι οι τιμές '15' και '3.7';
Φαίνεται σαν να είναι αριθμοί, αλλά επειδή βρίσκονται μέσα σε εισαγωγικά, είναι συμβολοσειρές:

```
>>> type ('15')
<type 'str'>
>>> type ('3.7')
<type 'str'>
```

**Εικόνα 2.5** *Ο διερμηνευτής διαβάζει τους αριθμούς μέσα σε εισαγωγικά ως συμβολοσειρές*

Τέλος, υπάρχει ακόμα ένα λεπτό σημείο που πρέπει να προσέξουμε. Επειδή στο αγγλοσαξονικό σύστημα χρησιμοποιείται η τελεία ως υποδιαστολή (αντί του κόμματος), αν πούμε στην Python να τυπώσει το 3,7 δεν καταλαβαίνει τον πραγματικό αριθμό 3,7 *αλλά μια λίστα αριθμών*, από όπου το πρώτο στοιχείο είναι ο αριθμός 3 και το δεύτερο στοιχείο είναι ο αριθμός 7. Έτσι, λοιπόν, τυπώνει τη λίστα αφήνοντας ένα κενό (space) ανάμεσα στους χαρακτήρες, όπως φαίνεται και στην Εικόνα 2.6.

```
>>> print (3,7)
(3, 7)
```

**Εικόνα 2.6** *Εκτύπωση λίστας αριθμών 3 και 7 και όχι του δεκαδικού αριθμού 3,7*

Όπως φαίνεται από το παραπάνω παράδειγμα εμείς εννοούσαμε το δεκαδικό αριθμό 3,7 αλλά ο διερμηνευτής της Python εννοούσε τη λίστα αριθμών 3 και 7.

Αυτό είναι το πρώτο παράδειγμα **«λάθους σημαντικής» (semantic error):** το πρόγραμμα τρέχει χωρίς να βγάζει συντακτικό λάθος, αλλά δεν κάνει αυτό που περιμέναμε να κάνει, το «σωστό».

## 2.3 Μεταβλητές

Τώρα, λοιπόν, γνωρίζουμε ότι έχουμε σταθερές τιμές, αλλά έχουμε και μεταβλητές τιμές. Από τα πιο δυνατά χαρακτηριστικά μιας γλώσσας προγραμματισμού είναι η δυνατότητα διαχείρισης μεταβλητών. Η **μεταβλητή (variable)** είναι ένα όνομα που αναφέρεται σε μια τιμή.

Πώς, όμως, ορίζουμε μεταβλητές τιμές με ονόματα και πώς δίνουμε συγκεκριμένες σταθερές τιμές σε μια μεταβλητή; Η πρώτη εντολή την οποία, συνήθως, μαθαίνει κανείς σε μια γλώσσα προγραμματισμού είναι η *εντολή εκχώρησης*. Άρα, αυτό που πρέπει να θυμόμαστε είναι ότι η μεταβλητή είναι ένα όνομα που αναφέρεται σε μια τιμή. Πρέπει να θυμόμαστε, δηλαδή, ότι πρόκειται για δυαδική αναφορά. Είναι ένα όνομα που αναφέρεται σε μια τιμή. Αυτήν την τιμή κάπου την έχουμε αποθηκεύσει στη μνήμη που έχει ο υπολογιστής. Μπορούμε, όμως, ακόμη να σκεφτούμε ότι η μεταβλητή είναι ένα όνομα που δείχνει τη θέση στην οποία βρίσκεται η τιμή. Άρα το όνομα είναι ένας δείκτης του σημείου όπου βρίσκεται η τιμή.

Ας υποθέσουμε ότι έχουμε μια μεταβλητή που έχει όνομα «course». Θα μπορούσε να είναι οποιοδήποτε το όνομα, ωστόσο εδώ χρησιμοποιούμε το όνομα course (μάθημα). Όπως θα δείτε είναι πιο χρήσιμο ως ονόματα μεταβλητών μας να χρησιμοποιούμε ονόματα τα οποία δείχνουν ότι αυτή η μεταβλητή σημαίνει κάτι στον πραγματικό κόσμο. Σημαίνει, δηλαδή, κάτι για μας που γράφουμε το πρόγραμμα. Η Python δίνει τη δυνατότητα να έχουμε τεράστιες μεταβλητές, όσο μεγάλες θέλουμε. Για παράδειγμα μπορούμε να γράψουμε σαν μεταβλητές ολόκληρες προτάσεις για να θυμόμαστε τη σημασία τους. Πιο συγκεκριμένα, έχουμε τη μεταβλητή course όπου εκχωρούμε την τιμή 'Python book', η οποία είναι μια ομάδα χαρακτήρων, ένα string αλλά και μια άλλη μεταβλητή που λέγεται *i* στην οποία εκχωρούμε

την τιμή 5. Επίσης, ένα ακόμη παράδειγμα μεταβλητής μπορεί να είναι η μεταβλητή pi και της εκχωρούμε την τιμή 3.14159.

Για να εκφράσει την εκχώρηση, η Python χρησιμοποιεί το απλό σύμβολο: = (π.χ., i = 5).

```
>>> course='Python book'
>>> i=5
>>> pi=3.14159
>>>
```

**Εικόνα 2.7** *Τρεις εντολές εκχώρησης*

Εδώ έχουμε τρεις εκχωρήσεις:

- Στην πρώτη εκχώρηση η συμβολοσειρά 'Python book' εκχωρείται στη νέα μεταβλητή που ονομάζεται course.
- Στη δεύτερη ο ακέραιος 5 εκχωρείται στην i.
- Στην τρίτη ο δεκαδικός 3.14159 στην pi.

Θα πρέπει να γνωρίζουμε ότι η print δουλεύει και με μεταβλητές:

```
>>> print(course)
Python book
>>> print(i)
5
>>> print(pi)
3.14159
>>>
```

**Εικόνα 2.8** *Η εντολή print με όρισμα μεταβλητή*

Οι μεταβλητές έχουν, επίσης, τύπους και μπορούμε να ρωτήσουμε την Python ποιοι είναι:

```
>>> type(course)
<type 'str'>
>>> type(i)
<type 'int'>
>>> type(pi)
<type 'float'>
```

**Εικόνα 2.9** *Η εντολή type με όρισμα μεταβλητή*

Ο τύπος μιας μεταβλητής είναι ο τύπος της τιμής στην οποία αναφέρεται.

## 2.4 Πολλαπλές εκχωρήσεις

Όπως ίσως θα έχετε διαπιστώσει, είναι νόμιμο να γίνουν παραπάνω από μία εκχωρήσεις σε μια μεταβλητή. Μια νέα εκχώρηση κάνει μια μεταβλητή να αναφέρεται σε μία νέα τιμή και να σταματήσει να αναφέρεται στην παλιά τιμή, όπως στο παράδειγμα της Εικόνας 2.10.

```
>>> value=1
>>> print(value)
1
>>> value=2
>>> print(value)
2
```

**Εικόνα 2.10** *Παράδειγμα πολλαπλών εκχωρήσεων*

Είναι σημαντικό να μπορείτε να ξεχωρίσετε μεταξύ δύο συμβολισμών, του x=3 και του x==3. Στο πρώτο, η μεταβλητή x παίρνει την τιμή 3 (εντολή εκχώρησης), ενώ στο άλλο το x==3 είναι δήλωση ισότητας. Με το πρώτο, η μεταβλητή x παραπέμπει σε μια θέση μνήμης όπου υπάρχει η τιμή 3, στο δεύτερο λέμε στο διερμηνευτή να ελέγξει κατά πόσο η τιμή στην οποία δείχνει το x είναι 3. Άρα επιστρέφει η Python true ή false (1 ή 0) 1 για το true, 0 για το false, και όπως θα δούμε και αργότερα, είναι μια λογική έκφραση. Άρα έχει πάρα πολύ σημασία να μην μπερδεύουμε αυτές τις δύο τιμές. Όταν θέλουμε να ελέγξουμε κατά πόσο μια μεταβλητή έχει μια συγκεκριμένη τιμή, χρησιμοποιούμε το "==", ενώ όταν θέλουμε να εκχωρήσουμε σε μια μεταβλητή μια τιμή, χρησιμοποιούμε το "=". Στην Python, ενώ η έκφραση "a=7" είναι αποδεκτή, αντίθετα η έκφραση "7=a" δεν είναι αποδεκτή, διότι δεν έχει νόημα να πούμε ότι η σταθερά 7 παίρνει την τιμή a. Και, βεβαίως, έχει νόη-

μα να λέμε ότι το x γίνεται "x+1" δηλαδή ότι η μεταβλητή x δείχνει σε μια θέση μνήμης όπου ήταν η παλιά τιμή, στην οποία όμως, έχει προστεθεί 1. Η ίδια θέση μνήμης δείχνει απλώς ότι στην παλιά τιμή έχει προστεθεί η τιμή 1.

## 2.5 Ονόματα μεταβλητών και λέξεις-κλειδιά

Τα ονόματα μεταβλητών μπορούν να περιέχουν αριθμούς και γράμματα, αλλά **πρέπει να ξεκινούν με ένα γράμμα.**

Τα πεζά διακρίνονται από τα κεφαλαία:

- Course και course είναι διαφορετικές μεταβλητές.
- myname και myName είναι διαφορετικές μεταβλητές.

Ο χαρακτήρας _ (underscore/κάτω παύλα) μπορεί να χρησιμοποιηθεί σε ένα όνομα και χρησιμοποιείται συχνά σε ονόματα με πολλές λέξεις, όπως π.χ.:

- my_name
- _my_name
- the_first_lesson

Οι μεταβλητές δεν μπορούν να αρχίζουν με αριθμούς, αλλά μπορούν να έχουν αριθμούς ενδιάμεσα ή στο τέλος. Το δολάριο δεν μπορεί να αντιπροσωπεύει ονόματα μεταβλητών και το class δεν μπορεί να είναι όνομα μεταβλητής, γιατί είναι από τα *δεσμευμένα ονόματα* (ή λέξεις-κλειδιά, από την αγγλική λέξη keywords). Η λίστα των δεσμευμένων ονομάτων φαίνεται στην Εικόνα 2.12.

Αυτές τις λέξεις με άλλα λόγια δεν μπορούμε να τις χρησιμοποιήσουμε για να ορίσουμε μεταβλητές. Είναι καλή πρακτική να έχουμε αυτή τη λίστα σε μια γρήγορη αναφορά, ώστε να αποφεύγουμε περίεργα συντακτικά λάθη. Για παράδειγμα, αν δώσουμε σε μια μεταβλητή ένα μη αποδεκτό όνομα (δηλαδή ένα όνομα του οποίου η χρήση είναι μη ορθή ή αντιστοιχεί σε δεσμευμένο όνομα) θα έχουμε συντακτικό λάθος (syntax error):

```
>>> 1course='Python book'
SyntaxError: invalid syntax
>>> price$=20
SyntaxError: invalid syntax
>>> class='Python book'
SyntaxError: invalid syntax
```

**Εικόνα 2.11** *Συντακτικό λάθος ορισμού μεταβλητής*

Στην Εικόνα 2.11 έχουμε συντακτικά λάθη διότι:

- 1course είναι μη αποδεκτό, γιατί δεν αρχίζει με γράμμα.
- price$ είναι μη αποδεκτό, γιατί περιέχει μη επιτρεπόμενο χαρακτήρα ($).

Ποιο είναι, όμως, το λάθος με το class;

Η λέξη **class,** λοιπόν, είναι μία από τις λέξεις-κλειδιά της Python. Οι λέξεις-κλειδιά συνδέονται με τους κανόνες και τη δομή της γλώσσας και δεν μπορούν να χρησιμοποιηθούν για ονόματα μεταβλητών.

Για να βρούμε τις λέξεις κλειδιά εκτελούμε τις εξής εντολές:

```
>>> import keyword
>>> keyword.kwlist
['and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'els
e', 'except', 'exec', 'finally', 'for', 'from', 'global', 'if', 'import', 'in',
'is', 'lambda', 'not', 'or', 'pass', 'print', 'raise', 'return', 'try', 'while',
'with', 'yield']
```

**Εικόνα 2.12** *Λέξεις κλειδιά της Python*

Είναι καλή πρακτική να κρατήσετε την παραπάνω λίστα σε μέρος όπου μπορείτε να ανατρέξετε εύκολα, έτσι ώστε, αν ο διερμηνευτής παραπονεθεί για συντακτικό λάθος, να μπορείτε εύκολα να το ελέγξετε.

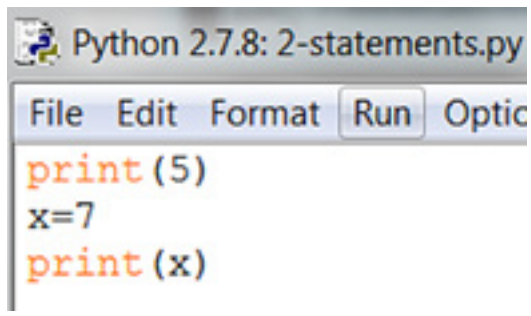
## 2.6 Εντολές

Έχουμε δει, ήδη, δύο είδη εντολών: print (εκτύπωση) και assignment (εκχώρηση). Οι εντολές γενικότερα εκτελούνται από το διερμηνευτή της Python. Όταν γράφετε μια εντολή στη γραμμή εντολών (command line), ο διερμηνευτής την εκτελεί και δείχνει στην οθόνη το αποτέλεσμα, αν η εντολή είναι εντολή εξόδου. Στην περίπτωση του print είναι, ενώ στην περίπτωση της εκχώρησης δεν είναι. Ένα πρόγραμμα ή ένα σενάριο (script) όπως ονομάζεται στην Python, συνήθως περιέχει περισσότερες της μίας εντολές. Στην περίπτωση αυτή τα αποτελέσματα παρουσιάζονται ένα-ένα καθώς εκτελούνται οι εντολές.

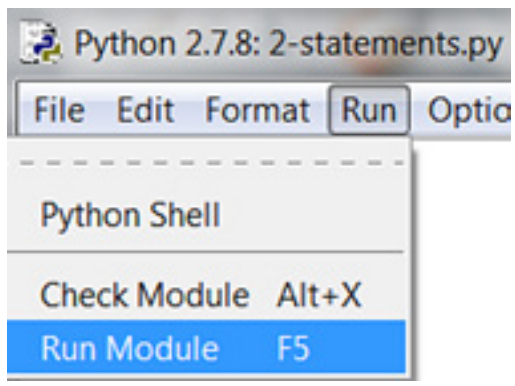
Για να γράψουμε και να τρέξουμε ένα σενάριο (script), ακολουθούμε τα εξής βήματα:

- Ανοίγουμε ένα νέο παράθυρο (File → New File).
- Γράφουμε το script, όπως στην Εικόνα 2.13.
- Αποθηκεύουμε σε αρχείο με κατάληξη .py.
- Καταρχήν, μπορούμε να κάνουμε check module (Run → Check Module).
- Τρέχουμε το script (Run → Run Module), όπως στην Εικόνα 2.14.

Ας φτιάξουμε, λοιπόν, ένα directory (ας το πούμε Python source) και ας ονομάσουμε το αρχείο test3. Το test δεν έβγαλε κανένα λάθος και άρα μπορούμε να το τρέξουμε. Όταν επιλέξουμε run, το πρόγραμμα τρέχει στο πρώτο παράθυρο και βλέπουμε το αποτέλεσμα, όπως στην Εικόνα 2.15. Λέμε print 5 και έτσι τυπώνει 5, μετά λέμε το x να πάρει την τιμή 7, γι' αυτό δεν έχει να τυπώσει τίποτα, και μετά λέμε print x και έτσι τυπώνει την τιμή 7. Βλέπετε, λοιπόν, πως πολύ εύκολα μπορείτε να φτιάξετε ένα μικρό πρόγραμμα σε Python, να το αποθηκεύσετε σε ένα αρχείο και να το τσεκάρετε. Αν λέγαμε σε αυτό το σημείο κάτι που θα είναι λάθος, π.χ. print y, που δεν είναι ορισμένο, πρώτα θα μας έδινε την οδηγία «σώσε το κάπου», και επειδή δεν έχει συντακτικό λάθος, θα το είχε αποθηκεύσει στο ίδιο σημείο με πριν. Στην περίπτωση που το τρέχαμε με run, θα είχε run time error.




**Εικόνα 2.13** *Σύνταξη ενός Python script*




**Εικόνα 2.14** *Εκτέλεση ενός Python script*

```
>>> ========================== RESTART ==========================
>>>
5
7
>>>
```

**Εικόνα 2.15** *Εμφάνιση του αποτελέσματος του Python script*

Ας δούμε τη διαφορά συντακτικών λαθών και λαθών στο χρόνο εκτέλεσης (run-time errors). Στο παράδειγμα της Εικόνας 2.16 στο οποίο ελέγχουμε για ισότητα της μεταβλητής x με το 2 (η σύγκριση για ισότητα εκφράζεται με δύο ίσον (==) σε αντίθεση με την εκχώρηση (=)). Αν το 3 είναι ίσον με το 2, ακολουθεί η άνω-κάτω-τελεία (:), που σημαίνει ότι τελείωσε το κομμάτι της συνθήκης και τώρα πάμε να εκτελέσουμε αν ισχύει η συνθήκη. Αν το 3 ήταν ίσο με 2, που δεν είναι, τότε θα εκτυπώναμε την τιμή του x, διαφορετικά (κάτι που ισχύει εδώ) το x παίρνει την τιμή 3 και μετά τυπώνουμε το x.

```
>>> x=3
>>> if x == 2:
        print x
else: x=3

>>> print x
3
```

**Εικόνα 2.16** *Ένα ακόμα απλό παράδειγμα κώδικα Python*

Αν κάνουμε κάποιο συντακτικό λάθος (π.χ. παραλείψουμε το :) η λειτουργία check module («έλεγξε το πρόγραμμα») θα μας πει ότι δεν είναι σωστή η σύνταξη και θα κοκκινίσει το μέρος όπου βλέπει κάποιο λάθος. Έτσι θα το διορθώσουμε, θα επιλέξουμε check module πάλι, το οποίο θα μας πει ότι δεν υπάρχει συντακτικό λάθος στο πρόγραμμα. Επιλέγοντας Run module θα τρέξουμε το πρόγραμμα και θα τυπώσει 3, όπως αναμένουμε.

Αν υποθέσουμε ότι για κάποιο λόγο στην Εικόνα 2.16 είχαμε βάλει print y, το οποίο δεν είναι ορισμένο, αυτό δεν είναι συντακτικό λάθος, γιατί αντίθετα με άλλες γλώσσες προγραμματισμού, στην Python δε χρειάζεται να δηλώνουμε καινούριες μεταβλητές. Οι μεταβλητές δηλώνονται την πρώτη φορά που τις χρησιμοποιούμε. Αλλά αν προσπαθήσουμε να το τρέξουμε, τότε βλέπετε ότι έχουμε το λεγόμενο run time error, λάθος κατά το χρόνο εκτέλεσης γιατί, προφανώς, δεν ήταν ορισμένο, επομένως δεν μπορεί να τυπωθεί κάτι, αφού δεν ξέρει την τιμή.

Είδαμε, λοιπόν, δύο διαφορετικές πολύ μεγάλες κατηγορίες λαθών, τα πρώτα, σχετικά πιο εύκολα, τα λεγόμενα συντακτικά λάθη, δηλαδή, στη σύνταξη της γλώσσας και τα άλλα πιο δύσκολα να αντιληφθεί κανείς, τα λεγόμενα λάθη εκτέλεσης, run time error. Είναι πολύ σημαντικό να γίνει κατανοητή αυτή η σημαντική διαφορά μεταξύ αυτών των δύο κατηγοριών λαθών.

## 2.7 Επίλογος

Σε αυτό το κεφάλαιο έγινε μια πρώτη εισαγωγή σε βασικές έννοιες που πρέπει να ξέρουμε για να γράψουμε το πρώτο μας πρόγραμμα σε Python. Είδαμε τις δεσμευμένες λέξεις-κλειδιά (keywords) της Python, βασικές εντολές, και τη διαφορά μεταξύ συντακτικών λαθών και λαθών εκτέλεσης, την οποία θα δούμε σε περισσότερη λεπτομέρεια στο Κεφάλαιο 7. Στο Κεφάλαιο 3 θα δούμε τις έννοιες της έκφρασης, των τελεστών, και των σχολίων του προγραμματιστή μέσα στο πρόγραμμα Python.

## Βιβλιογραφία/Αναφορές

Gaddis, T. (2012). *Starting out with Python* (Second Edition). Addison-Wesley.

## Κριτήρια αξιολόγησης

**Κριτήριο αξιολόγησης I (Βαθμός δυσκολίας[1]:●)**

**Ποια από τα παρακάτω είναι μη επιτρεπόμενα ονόματα στην Python; Σημειώστε την απάντησή σας (επιτρεπόμενο / μη επιτρεπόμενο) δίπλα στο κάθε όνομα (Gaddis, 2012);**

**units_per_day**

**dayOfWeek**

**3dGraph**

**June1997**

**Mixture#3**

1 Ο βαθμός δυσκολίας των ασκήσεων κυμαίνεται μεταξύ των: χαμηλός (●), μέτριος (●●), και υψηλός (●●●)

**Απάντηση/Λύση**

**Κριτήριο αξιολόγησης 2 (Βαθμός δυσκολίας:●)**

**Το όνομα της μεταβλητής Book και book είναι ίδιο;**

**Απάντηση/Λύση**

**Κριτήριο αξιολόγησης 3 (Βαθμός δυσκολίας:●)**

**Τι θα εκτυπωθεί από το ακόλουθο πρόγραμμα:**

```
my_book = 'Java book'
my_book = 'Python book'
print(my_book)
```

**Απάντηση/Λύση**

# ΚΕΦΑΛΑΙΟ 3

# Εκφράσεις, τελεστές, σχόλια

*Σύνοψη*

*Στο κεφάλαιο αυτό εισάγουμε τον τρόπο τέλεσης πράξεων μεταξύ μεταβλητών και σταθερών, εκφράσεις μεταξύ αυτών καθώς και το σχολιασμό της λογικής ενός προγράμματος.*

*Προαπαιτούμενη γνώση*

*Κεφάλαια 1, 2 του παρόντος συγγράμματος.*

## 3.1 Εισαγωγή

Σε αυτό το κεφάλαιο θα ασχοληθούμε με την τέλεση πράξεων μεταξύ μεταβλητών και σταθερών, τη σύνθεσή τους σε εκφράσεις, και το σχολιασμό της λογικής ενός προγράμματος. Μια **έκφραση** (expression) είναι ένας συνδυασμός τιμών, μεταβλητών και τελεστών. Οι **τελεστές** (operators) είναι λειτουργίες που επιτελούν πράξεις και μπορούν να αναπαρασταθούν με σύμβολα όπως το + ή με λέξεις- κλειδιά, όπως το **and**. Οι τιμές και οι μεταβλητές πάνω στις οποίες εφαρμόζονται οι τελεστές, ονομάζονται **τελεστέοι** (operands).

Εάν εισάγετε μια έκφραση στο διερμηνευτή, αυτός την υπολογίζει και δείχνει το αποτέλεσμα.

```
>>> 2+1
3
>>> x=4
>>> x+2
6
>>>
```

**Εικόνα 3.1** *Υπολογισμός εκφράσεων*

Δε χρειάζεται μια έκφραση να περιέχει ταυτόχρονα και τιμές και μεταβλητές και τελεστές. Μια τιμή, όπως και μια μεταβλητή, από μόνες τους αποτελούν εκφράσεις. Στην Εικόνα 3.2 αυτό που πρέπει να προσέξουμε είναι ότι στη μεταβλητή x θα πρέπει να έχει ανατεθεί νωρίτερα κάποια τιμή.

```
>>> 9
9
>>> x
4
>>>
```

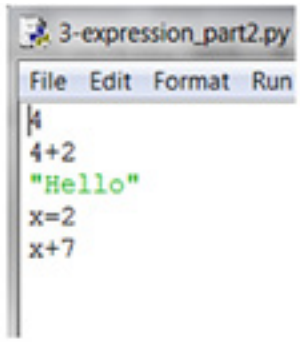
**Εικόνα 3.2** *Τιμές και μεταβλητές ως εκφράσεις*

Στην Εικόνα 3.3 παρατηρούμε τη διαφορά μεταξύ του υπολογισμού μιας έκφρασης και της εκτύπωσής της. Αρχικοποιούμε, λοιπόν, μια μεταβλητή που τυχαίνει να είναι συμβολοσειρά (ομάδα χαρακτήρων), και μετά ζητούμε από το διερμηνευτή της Python την τιμή της, οπότε τυπώνει τη συμβολοσειρά με τα εισαγωγικά. Είναι, επομένως, διαφορετικό να τυπώνει κανείς στην έξοδο κάτι, από το να ρωτάει τον interpreter της Python, τον IDLE, για την τιμή αυτής της μεταβλητής. Η τιμή μιας μεταβλητής που είναι τύπου συμβολοσειράς πρέπει να έχει και τα εισαγωγικά. Είναι ακόμη σημαντικό να τονίσουμε ότι δεν παράγουν οι εντολές έξοδο, εκτός από την εντολή εκτύπωσης.

```
>>> message="Τι γίνεται;
>>> message
'Τι γίνεται;'
>>> print(message)
Τι γίνεται;
>>>
```

**Εικόνα 3.3** *Υπολογισμός έκφρασης και η εκτύπωσή της*

Σε ένα σενάριο (script), μια έκφραση είναι από μόνη της μια νόμιμη εντολή, αλλά δεν παράγει έξοδο (Εικόνα 3.4). Για να παραχθεί έξοδος χρησιμοποιούμε την εντολή print.





```
>>> ========== RESTART ==========
>>>
>>>
```

**Εικόνα 3.4** *Έκφραση ως μέρος ενός script*

## 3.2 Τελεστές

Μια ομάδα σημαντικών τελεστών παρουσιάζονται στην Εικόνα 3.5. Το * σημαίνει πολλαπλασιασμός, τα ** είναι ύψωση σε δύναμη (δείτε π.χ. το 5**2 ή $5^2$ στην Εικόνα 3.6). Στη διαίρεση, όταν ο διαιρετέος και ο διαιρέτης είναι ακέραιοι, το αποτέλεσμα είναι επίσης ακέραιος, στρογγυλοποιημένος προς τον πλησιέστερο (προς τα κάτω) ακέραιο. Γι' αυτό το 59/60 βγάζει 0, επειδή το ακέραιο μέρος του 59/60 είναι 0. Αυτό δε βοηθάει στην περίπτωση που θα θέλαμε να έχουμε κάποια δεκαδικά ψηφία. Θα μπορούσαμε να τους μετατρέψουμε σε πραγματικούς, αλλά μια άλλη

εναλλακτική λύση την οποία θα μπορούσαμε να επιλέξουμε είναι να πολλαπλασιάσουμε επί 100 ή επί τόσες δυνάμεις του 10, όσα είναι και τα δεκαδικά ψηφία που θέλουμε να βγάλουμε.

| | |
|---|---|
| + | Πρόσθεση αριθμών ή αλληλουχία συμβολοσειρών. |
| - | Αφαίρεση ενός αριθμού από έναν άλλο. |
| * | Γινόμενο δύο αριθμών ή επανάληψη μιας συμβολοσειράς τόσε φορές. |
| ** | ΄Υψωση αριθμού σε δύναμη. |
| / | Διαίρεση δύο αριθμών. |
| // | Διαίρεση δύο αριθμών στρογγυλοποιημένη προς τα κάτω (floor division). |
| % | Υπόλοιπο διαίρεσης δύο αριθμών. |

**Εικόνα 3.5** *Τελεστές*

Στην Εικόνα 3.6 παρουσιάζονται παραδείγματα πράξεων με αριθμούς.

```
>>> 2+3
5
>>> 7-4
3
>>> 2*5
10
>>> 5**2
25
>>> 10/2
5.0
>>> 10//2
5
>>> 10/3
3.3333333333333335
```

**Εικόνα 3.6** *Παραδείγματα πράξεων με αριθμούς.*

Όταν μια μεταβλητή εμφανίζεται σε έκφραση, αντικαθίσταται από την τιμή της πριν υπολογιστεί η έκφραση (Εικόνα 3.7).

```
>>> x=5
>>> x+2
7
```

**Εικόνα 3.7** *Υπολογισμός έκφρασης με μεταβλητή*

Κάποια παραδείγματα εκφράσεων είναι τα εξής:

l2+24*hour-6

hour*60+minute

minute/60

6**3

(6+9)*(3l-7)

Ένα άλλο σημαντικό θέμα έχει να κάνει με την προτεραιότητα με την οποία εφαρμόζονται οι τελεστές. Όταν έχουμε μια έκφραση, *με τι σειρά υπολογίζει η Python την έκφραση;* Αρχικά υπολογίζει τα εντός παρενθέσεων ευρισκόμενα, ξεκινώντας από τις εσωτερικότερες παρενθέσεις και κλείνοντας προς τις εξωτερικότερες παρεν-

θέσεις. Στο πρώτο παράδειγμα της Εικόνας 3.8, πρώτα θα υπολογίσει το 3-1 και μετά θα πολλαπλασιάσει επί 2. Στη συνέχεια, πρώτα θα υπολογίσει το 2+1 και το 5-2, και μετά θα γίνει η ύψωση στη δύναμη. Οι παρενθέσεις είναι ένας τρόπος να επηρεάσουμε τη σειρά με την οποία υπολογίζονται οι εκφράσεις. Αν στο παράδειγμα minute*100/60, θέλουμε πρώτα να κάνει τη διαίρεση και μετά τον πολλαπλασιασμό, πρέπει να πούμε minute*(100/60), ώστε να επιτευχθεί αυτό.

Μετά τις παρενθέσεις, αυτό που έχει αμέσως προτεραιότητα είναι η ύψωση σε δύναμη. Γι' αυτό και στην Εικόνα 3.8, η Python στην έκφραση 2+1**5-2 υπολογίζει πρώτα την τιμή της πράξης 1**5 και μετά τις υπόλοιπες. Οι κανόνες είναι χρήσιμοι και για να καταλαβαίνουμε τι γίνεται. Όπου δεν είμαστε σίγουροι ή δε θυμόμαστε τι ισχύει, είναι καλό να βάζουμε παρενθέσεις για να μην μπερδευόμαστε.

```
>>> 2*(3-1)
4
>>> (2+1)**(5-2)
27
>>> 2*3-1
5
>>> 2+1**5-2
1
>>> 5*2+3
13
>>> (5*2)+3
13
```

**Εικόνα 3.8** *Προτεραιότητα τελεστών*

Το ακρωνύμιο **PEMDAS** (Purplemath, 2015) μας βοηθάει να θυμόμαστε τους κανόνες προτεραιότητας. Οι **παρενθέσεις** (**P**arenthesis) έχουν τη μεγαλύτερη προτεραιότητα και χρησιμοποιούνται για να αναγκάσουν την Python να υπολογίσει μια έκφραση σύμφωνα με τη σειρά που θέλουμε. Εκφράσεις σε παρενθέσεις υπολογίζονται πρώτες. Παρενθέσεις χρησιμοποιούνται, επίσης, για να κάνουν τις εκφράσεις πιο αναγνώσιμες, χωρίς να αλλάζουν το αποτέλεσμα.

Η **ύψωση σε δύναμη** (**E**xponentiation) όπως προείπαμε, έχει την επόμενη μεγαλύτερη προτεραιότητα (Εικόνα 3.9).

```
>>> 2**1+1
3
>>> 3*1**3
3
```

**Εικόνα 3.9** *Ύψωση σε δύναμη*

**Πολλαπλασιασμός** (**M**ultiplication) και **διαίρεση** (**D**ivision) έχουν την ίδια προτεραιότητα, που είναι μεγαλύτερη από την **πρόσθεση** (**A**ddition) και την **αφαίρεση** (**S**ubtraction) οι οποίες έχουν, επίσης, την ίδια προτεραιότητα (Εικόνα 3.10). Τελεστές με την ίδια προτεραιότητα υπολογίζονται από τα αριστερά προς τα δεξιά.

```
>>> 3*4+2
14
>>> 15/3-2
3.0
```

**Εικόνα 3.10** *Προτεραιότητα τελεστών*

Είδαμε, προηγουμένως, ότι μπορούμε να κάνουμε πράξεις με ακέραιους αριθμούς. Μπορούμε, επίσης, να κάνουμε και κάποιες πράξεις στις ομάδες χαρακτήρων, ωστόσο όχι όλες τις πράξεις. Για παράδειγμα, τα παρακάτω **δεν είναι σωστά** (υποθέτοντας ότι η μεταβλητή message είναι τύπου string):

**message-1  “Hello”/2  “Hello”/ “H”  message*“Hello”  “15”+2**

Ωστόσο, ο τελεστής + έχει νόημα για συμβολοσειρές. Δημιουργεί μια αλληλουχία (concatenation) συμβολοσειρών, δηλαδή, συνδέει τους δύο «προσθετέους» φτιάχνοντας νέα συμβολοσειρά, όπου η αρχή της δεύτερης βρίσκεται μετά το τέλος της πρώτης (Εικόνα 3.11).

```
>>> first_name="Python"
>>> last_name=" book"
>>> my_name=first_name+last_name
>>> print(my_name)
Python book
```

**Εικόνα 3.11** *Χρήση τελεστή + για συμβολοσειρές*

Επιπλέον ο τελεστής * έχει νόημα στις συμβολοσειρές, εκτελεί επανάληψη (Εικόνα 3.12).

```
>>> 3*"Hello"
'HelloHelloHello'
>>> print(3*"Hello")
HelloHelloHello
>>>
```

**Εικόνα 3.12** *Χρήση τελεστή * για συμβολοσειρές*

Ο ένας από τους πολλαπλασιαστέους πρέπει να είναι συμβολοσειρά και ο άλλος ακέραιος. Υπάρχουν ομοιότητες και διαφορές με τις αντίστοιχες πράξεις της αριθμητικής. Όπως το 3*4 είναι ισοδύναμο με το 4+4+4, έτσι και το 3*“Hello” είναι ισοδύναμο με το “Hello”+“Hello”+“Hello”.

Στην Python υπάρχει η δυνατότητα σύνθεσης μεταβλητών, εκφράσεων και εντολών (Εικόνα 3.13). Μπορούμε να τυπώνουμε όχι μόνο τιμές, όχι μόνο μεταβλητές, αλλά και εκφράσεις. Στην Εικόνα 3.13 βλέπετε μια έκφραση αριθμητική, ενώ από κάτω έχουμε έκφραση με ομάδες χαρακτήρων.

```
>>> hours=2

>>> minutes=50
>>> print("Αριθμός λεπτών μέχρι τα μεσάνυχτα:", hours*60+minutes)
Αριθμός λεπτών μέχρι τα μεσάνυχτα: 170
```

**Εικόνα 3.13** *Σύνθεση μεταβλητών, εκφράσεων και εντολών*

## 3.3 Σχόλια

Οι γλώσσες προγραμματισμού συμπυκνώνουν νοήματα, γι’ αυτό και είναι δύσκολο να τις διαβάσει κανείς, να καταλάβει **τι** θέλει να επιτελέσει ένα κομμάτι κώδικα και

**γιατί**. Γι' αυτό και χρειάζεται να προσθέτουμε σχόλια στα προγράμματα που γράφουμε. **Τα σχόλια στην Python αρχίζουν πάντα με το χαρακτήρα #** (Εικόνα 3.14). Οτιδήποτε ακολουθεί μετά το # αγνοείται από την Python μέχρι το τέλος της γραμμής.

```
#Υπολογισμός εμβαδού ενός ορθογωνίου

length = 4   #μήκος
breadth = 3 #πλάτος
area = length * breadth #εμβαδόν
print("Εμβαδόν =", area)
```

Εκτέλεση

```
>>> ================================ RESTART ================================
>>>
Εμβαδόν = 12
```

**Εικόνα 3.14** *Σχόλια στην Python*

Τέλος, ας δούμε πώς αναγνωρίζει ο interpreter πού τελειώνει το σχόλιο και πού αρχίζει η επόμενη εκτελέσιμη γραμμή. Η απάντηση είναι ένας ειδικός χαρακτήρας ο οποίος εισέρχεται, όταν διαβάζει αυτά που του δίνουμε από την κονσόλα (console ή command line) ή στο αρχείο που γράφουμε. Υπάρχει πάντα ένας ειδικός χαρακτήρας που χαρακτηρίζει το τέλος γραμμής και ότι μετά από αυτό αλλάζει η γραμμή. Άρα, όταν ο διερμηνευτής δει αυτόν τον ειδικό χαρακτήρα (ο οποίος δεν τυπώνεται), καταλαβαίνει ότι τέλειωσε το σχόλιό μας, βλέποντας το τέλος γραμμής. Καθώς, λοιπόν, διαβάζει σειρά-σειρά το πρόγραμμα, βλέπει τον ειδικό χαρακτήρα ο οποίος δηλώνει το τέλος της γραμμής.

## 3.4 Επίλογος

Σε αυτό το κεφάλαιο κάναμε μια εισαγωγή στις έννοιες των εκφράσεων, των τελεστών και της προτεραιότητάς τους, και των σχολίων του προγραμματιστή μέσα σε ένα πρόγραμμα Python. Στο επόμενο κεφάλαιο θα μελετήσουμε την έννοια της συνάρτησης και της εκτέλεσης υπό συνθήκη.

## Βιβλιογραφία/Αναφορές

Purplemath. (2015). *The Order of Operations:* PEMDAS. Retrieved from http://www.purplemath.com/modules/orderops.htm

## Κριτήρια αξιολόγησης

### Κριτήριο αξιολόγησης 1 (Βαθμός δυσκολίας: ●●)

**Υπολογίστε το αποτέλεσμα της εξής έκφρασης:**

**print 3 + 2 + 1 - 5 + 4 % 2 - 1 / 8 + 4**

**Απάντηση/Λύση**

### Κριτήριο αξιολόγησης 2 (Βαθμός δυσκολίας: ●●)

**Ποια θα είναι η τιμή της μεταβλητής c στο εξής παράδειγμα;**

**a=50**

**b=15**

**c=0**

**c=a&b**

**print c**

**Απάντηση/Λύση**

## Κριτήριο αξιολόγησης 3 (Βαθμός δυσκολίας:●)

**Η εκτέλεση του συγκεκριμένου κώδικα, τι θα έχει ως αποτέλεσμα;**

```
File  Edit  Format  Run  Options  Windows  Help
# Σχόλιο 1
# print 1
print 2 # Η συνέχεια του παραδείγματος

# print 3
# print 4
```

**Απάντηση/Λύση**

## Κριτήριο αξιολόγησης 4 (Βαθμός δυσκολίας:●)

**Μπορείτε να σκεφτείτε μια ιδιότητα που έχει η πρόσθεση (addition) αριθμών, ενώ η αλληλουχία (concatenation) συμβολοσειρών δεν την έχει;**

**Απάντηση/Λύση**

# ΚΕΦΑΛΑΙΟ 4

# Συναρτήσεις και εκτέλεση υπό συνθήκη

*Σύνοψη*

*Στο κεφάλαιο αυτό εισάγουμε την έννοια της συνάρτησης και της εκτέλεσης υπό συνθήκη.*

*Προαπαιτούμενη γνώση*

*Κεφάλαια 1-3 του παρόντος συγγράμματος.*

## 4.1 Εισαγωγή

Σε αυτό το κεφάλαιο πραγματευόμαστε την έννοια των συναρτήσεων στην Python. Έχουμε, ήδη, δει κάποιες συναρτήσεις, όπως την print, με την οποία μπορούμε να τυπώνουμε τις τιμές μεταβλητών, σταθερών, ομάδων χαρακτήρων, δηλαδή, string κτλ. και την type, η οποία μας επιστρέφει τον τύπο κάποιας μεταβλητής. Οι **συναρτήσεις** είναι επαναχρησιμοποιήσιμα μέρη προγραμμάτων. Η έννοια της συνάρτησης μας επιτρέπει να δίνουμε ένα όνομα (το όνομα της συνάρτησης) σε ένα σύνολο εντολών, το οποίο αποτελεί το σώμα της συνάρτησης, αλλά και να εκτελούμε τη συνάρτηση οπουδήποτε στο πρόγραμμά μας και όσες φορές θέλουμε. Αυτό είναι γνωστό σαν **κλήση (calling)** της συνάρτησης. Κατά τον **ορισμό** μιας συνάρτησης πρέπει να καθορίζουμε το όνομά της, τις παραμέτρους εισόδου της, και το σύνολο των εντολών που περιέχει. Οι τιμές των παραμέτρων κατά την κλήση των συναρτήσεων λέγονται και **ορίσματα** (arguments) της συνάρτησης και το αποτέλεσμα της συνάρτησης λέγεται **«επιστρεφόμενη τιμή»** (return value).

Ας πάρουμε σαν πρώτο παράδειγμα τη συνάρτηση type (Εικόνα 4.1) η οποία μάς επιστρέφει τον τύπο μιας τιμής ή μιας μεταβλητής την οποία περνάμε ως όρισμα της συνάρτησης, όπως φαίνεται στην Εικόνα 4.1. Μπορούμε να περάσουμε σαν ορίσματα της type μεταβλητές, όπως για παράδειγμα την i, στην οποία προηγουμένως έχουμε δώσει μια ακέραια τιμή. Σε αυτό το παράδειγμα επιστρέφεται ο τύπος της μεταβλητής, που είναι 'int'. Η δεύτερη κλήση της type επιστρέφει τον τύπο της σταθεράς 3, η οποία μάς λέει ότι ανήκει στην κλάση των ακεραίων, ή αντίστοιχα τον τύπο του 2.5 που είναι 'float', δηλαδή πραγματικός αριθμός. Τέλος, ο τύπος της ομάδας χαρακτήρων 'Hello World!' είναι 'str' ή String.

```
>>> i=3
>>> type(i)
<type 'int'>
>>> type(3)
<type 'int'>
>>> type(2.5)
<type 'float'>
>>> type('Hello World!'
<type 'str'>
>>> type(i)
<type 'int'>
>>> type(pi)
<type 'float'>
>>> type(course)
<type 'str'>
```

**Εικόνα 4.1** *Παραδείγματα κλήσης συνάρτησης*

Στο παράδειγμα της Εικόνας 4.2 φαίνεται ότι η επιστρεφόμενη αυτή τιμή μπορεί να εκχωρηθεί, να δοθεί σαν τιμή μιας μεταβλητής, την οποία ορίζουμε μέσω αυτής της εντολής εκχώρησης. Βλέπετε π.χ. ότι το t παίρνει σαν τιμή τον τύπο της ομάδας χαρακτήρων "Hello", που είναι string.

```
>>> t=type("Hello")
>>> print(t)
<type 'str'>
```

**Εικόνα 4.2** *Εκχώρηση επιστρεφόμενης τιμής σε μεταβλητή*

Κάθε τιμή έχει και μια «ταυτότητα», η οποία έχει σχέση με τη θέση στη μνήμη του υπολογιστή όπου είναι αποθηκευμένη. Η «ταυτότητα» μιας μεταβλητής είναι

η ταυτότητα της τιμής στην οποία αναφέρεται. Για παράδειγμα στην Εικόνα 4.3, ο διερμηνευτής της Python μέσω της συνάρτησης id μάς επιστρέφει την ταυτότητα της σταθεράς 7. Η «ταυτότητα» αναφέρεται στη θέση της μνήμης που έχει αποθηκεύσει ο διερμηνευτής της Python τη σταθερά 7. Εν προκειμένω στη θέση 30509816.

Για τον προγραμματιστή δεν έχει ιδιαίτερη σημασία ο ακριβής αριθμός. Σημασία έχει οτι η «ταυτότητα» αυτή είναι μοναδική. Ας προχωρήσουμε, λοιπόν, και ας δώσουμε ένα όνομα, ας πούμε n, στην σταθερά 7. Δηλαδή, με άλλα λόγια το n γίνεται 7. Τότε το όνομα αυτό δείχνει την ίδια θέση μνήμης και έτσι όταν λέμε id(n), δίνει ακριβώς το ίδιο νούμερο. Εκεί, δηλαδή, που είναι αποθηκευμένη η σταθερά 7.

```
>>> id(7)
30509816L
>>> n=7
>>> id(n)
30509816L
```

**Εικόνα 4.3** *Παράδειγμα κλήσης συνάρτησης*

Εκτός από τις προαναφερόμενες συναρτήσεις: την print, την type, και την id, υπάρχουν και συναρτήσεις οι οποίες μετατρέπουν τύπους μεταβλητών, όπως για παράδειγμα η int, η οποία μπορεί να πάρει σαν όρισμα μια ομάδα χαρακτήρων, όπως για παράδειγμα την "19", την οποία και μετατρέπει στον ακέραιο 19 (Εικόνα 4.4). Άρα κάποιες συναρτήσεις μπορούν να παίρνουν ομάδες χαρακτήρων ή και πραγματικούς αριθμούς και τους κάνουν ακεραίους αν μπορούν. Όμως στο επόμενο παράδειγμα της Εικόνας 4.4, στο οποίο παίρνει σαν όρισμα την λέξη "Hello", δεν μπορεί να την μετατρέψει σε ακέραιο και επιστρέφει μήνυμα λάθους.

```
>>> int("19")
19
>>> int("Hello")

Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    int("Hello")
ValueError: invalid literal for int() with base 10: 'Hello'
```

**Εικόνα 4.4** *Η συνάρτηση int*

Η int μετατρέπει, επίσης, πραγματικούς σε ακέραιους, αφαιρώντας το δεκαδικό τους μέρος, όπως φαίνεται στα παραδείγματα της Εικόνας 4.5:

```
>>> int(3.14159)
3
>>> int(-2.7)
-2
```

**Εικόνα 4.5** *Η συνάρτηση int με όρισμα πραγματικό αριθμό*

Η **συνάρτηση float** μετατρέπει ακεραίους και συμβολοσειρές σε πραγματικούς αριθμούς (Εικόνα 4.6).

```
>>> float(15)
15.0
>>> float("3.14159")
3.14159
```

**Εικόνα 4.6** *Η συνάρτηση float*

Τέλος, η **συνάρτηση str** μετατρέπει άλλους τύπους (όπως αριθμούς) σε συμβολοσειρές (Εικόνα 4.7).

```
>>> str(15)
'15'
>>> str(3.14159)
'3.14159'
```

**Εικόνα 4.7** *Η συνάρτηση str.*

Η Python διαθέτει ένα «πακέτο» (module) το οποίο περιλαμβάνει πολλές από τις γνωστές μαθηματικές συναρτήσεις. Εκτός από αυτές στις οποίες ήδη αναφερθήκαμε, μαθηματικές συναρτήσεις είναι αυτές που υπολογίζουν ημίτονα, συνημίτονα κλπ. Για να τις χρησιμοποιήσουμε, πρέπει να εισαγάγουμε τις συναρτήσεις αυτές στο διερμηνευτή, εκεί που τρέχουν τα προγράμματα. Αυτό το κάνουμε με το import, όπως φαίνεται στην Εικόνα 4.8.

```
>>> import math
```

**Εικόνα 4.8** *Εισαγωγή μαθηματικών συναρτήσεων*

Για περισσότερες πληροφορίες για τη χρήση του module math μπορούμε να χρησιμοποιήσουμε τη συνάρτηση help. Για να δούμε ποιες συναρτήσεις υπάρχουν και πώς ορίζονται στο αρχείο αυτό γράφουμε:

```
help(math)
```

**Εικόνα 4.9** *Χρήση της συνάρτησης help*

Παράδειγμα τέτοιων μαθηματικών συναρτήσεων είναι ο λογάριθμος. Για να καλέσετε αυτές τις συναρτήσεις βάζετε πρώτα το όνομα του πακέτου, δηλαδή το όνομα του αρχείου που περιέχει τις μαθηματικές συναρτήσεις math, και μετά τελεία. Πρόκειται για το λεγόμενο "dot notation" με το οποίο γίνεται η επιλογή. Η τελεία είναι ένας τελεστής, όπως θα δούμε αργότερα, όπως το +, το –, το / κ.λπ. η λειτουργία του οποίου είναι να κάνει επιλογή.

Στην Εικόνα 4.10 όταν λέμε "math. ", ορίζουμε στο διερμηνευτή της Python να ανατρέξει στο πακέτο math και να διαλέξει από εκεί τη μαθηματική συνάρτηση, λογάριθμο με βάση το 10 ($\log_{10}$), μια λέξη. Ο λογάριθμος με βάση το 10 του 100 είναι 2, αντικαθιστά λοιπόν το 2, προσθέτει 3 και άρα το x=5, το οποίο φαίνεται και μετά αν πούμε print(x).

Μια άλλη μαθηματική συνάρτηση είναι ο αριθμός π ή pi στα αγγλικά, "math.pi", που επιστρέφει με κάποια ακρίβεια τα πρώτα ψηφία του αριθμού π.

```
>>> x=3+math.log10(100)
>>> print(x)
5.0
>>> print(math.pi)
3.14159265359
```

**Εικόνα 4.10** *Χρήση μαθηματικών συναρτήσεων*

Στις τριγωνομετρικές συναρτήσεις τα ορίσματα είναι οι γωνίες. Η τιμή της γωνίας πρέπει να είναι σε radians και όχι σε μοίρες. Θυμηθείτε ότι ο κύκλος έχει 2π radians. Οπότε, πρέπει να μετατρέψουμε τις μοίρες σε radians. Διαιρούμε, λοιπόν, με 360 και πολλαπλασιάζουμε με 2π. Αυτό κάνουμε στην Εικόνα 4.11, όπου μετατρέπουμε τις 45 μοίρες σε radians με τη διαίρεση 45/360*2π. Βλέπετε ότι για να βάλουμε το

π, επιστρέφουμε στο πακέτο math και επιλέγουμε τη μαθηματική συνάρτηση pi. Εκεί επιλέγουμε τη συνάρτηση ημίτονο και έτσι υπολογίζουμε το ημίτονο της γωνίας των 45°. Και, προφανώς, μπορούμε να κάνουμε το ίδιο για το συνημίτονο, την εφαπτομένη, τη συνεφαπτομένη και άλλες σχετικές συναρτήσεις. Όπως και στις μαθηματικές συναρτήσεις, η παράμετρος που περνάμε σε μια συνάρτηση μπορεί να είναι η ίδια αριθμητική έκφραση. Δηλαδή, αντί να ζητήσουμε το ημίτονο του π, μπορούμε να δώσουμε μια έκφραση, όπως στην Εικόνα 4.12. Σημειώστε ότι η συνάρτηση exp σημαίνει εκθέτης (exponent) με βάση το e (φυσικοί λογάριθμοι).

```
>>> degrees = 45
>>> radians = degrees / 360 * 2 * math.pi
>>> math.sin(radians)
0.7071067811865475
```

**Εικόνα 4.11** *Χρήση τριγωνομετρικών συναρτήσεων*

Όπως και στις μαθηματικές συναρτήσεις, έτσι και στην Python μπορούμε να γράψουμε:

```
>>> x = math.sin(degrees / 360 * 2 * math.pi)
>>> x = math.exp(math.log(x+1))
```

**Εικόνα 4.12** *Σύνθεση συναρτήσεων*

### 4.1.1 Ορισμός συνάρτησης

Μέχρι τώρα χρησιμοποιούσαμε συναρτήσεις που μάς έδινε η Python. Ας δούμε τώρα πώς μπορούμε να ορίσουμε, λοιπόν, μια δική μας συνάρτηση. Όπως φαίνεται στην Εικόνα 4.13, αρχίζουμε με τη λέξη def που προέρχεται από τη λέξη define, η οποία σημαίνει «ορίζω», αφήνουμε κενό, και μετά γράφουμε το όνομα της συνάρτησης (αντί του NAME). Μετά ακολουθεί η λίστα των παραμέτρων που θα ορίσουμε για τη συνάρτηση αυτή. Έπειτα βάζουμε ":" για να ορίσουμε ότι μετά ακολουθούν οι παράμετροι / ορίσματα.

Οι εντολές κάτω από τον ορισμό είναι πιο μέσα από το αριστερό περιθώριο του ορισμού (def)˙ κάνουμε, λοιπόν, μια εσοχή (indentation). Όταν τελειώσει ο ορισμός της συνάρτησης ξαναγυρίζουμε πάλι στο αριστερό περιθώριο. Έτσι «καταλαβαίνει» η Python ότι τελείωσε ο ορισμός της συνάρτησης. Σε άλλες γλώσσες ο συνηθισμένος τρόπος να γίνει αυτό είναι, για παράδειγμα, η χρήση αγκύλων ({}) στη C ή begin και end στην Pascal. Η Python στοχεύει στην οικονομία γραφής και έτσι ο διερμηνευτής «καταλαβαίνει» ότι όταν κάνουμε εσοχή κώδικα, ορίζουμε κάτι και όταν τελειώνει η εσοχή, τελειώνει ο ορισμός.

Η σύνταξη του ορισμού μιας συνάρτησης γίνεται, λοιπόν, ως εξής:

```
def NAME (λίστα παραμέτρων):
    εντολές
```

**Εικόνα 4.13** *Σύνταξη ορισμού συνάρτησης*

Ένα παράδειγμα μιας πολύ απλής συνάρτησης της οποίας το όνομα είναι sayHello, χωρίς παραμέτρους, φαίνεται στην Εικόνα 4.14. Εφόσον δεν έχουμε παραμέτρους, βλέπετε ότι ανοίγουμε και κλείνουμε την παρένθεση στον ορισμό. Ο ορισμός της συνάρτησης είναι «τύπωσε τη λέξη "Hello" και τερμάτισε».

```
>>> def sayHello():
        print("Hello")


>>> sayHello()
Hello
>>> sayHello()
```

**Εικόνα 4.14** *Πρώτο παράδειγμα ορισμού συνάρτησης*

Στην Εικόνα 4.15 φαίνεται ο ορισμός μιας συνάρτησης που τυπώνει Hello τρεις φορές καθώς και η εκτέλεσή της.

```
>>> def say3Hello():
        sayHello()
        sayHello()
        sayHello()


>>> say3Hello()
Hello
Hello
Hello
```

**Εικόνα 4.15** *Δεύτερο παράδειγμα ορισμού συνάρτησης*

Στο παραπάνω παράδειγμα παρατηρήστε τα εξής:

- Μπορεί κανείς να καλεί την ίδια συνάρτηση πολλές φορές, μάλιστα, κάτι τέτοιο είναι πολύ συνηθισμένο και χρήσιμο.
- Στον ορισμό μιας συνάρτησης μπορεί να γίνεται κλήση μιας άλλης συνάρτησης. Στην περίπτωσή μας η say3Hello καλεί την sayHello.

Στην Εικόνα 4.16 φαίνεται το αποτέλεσμα που παράγεται, όταν καλούμε print(sayHello). Τα ονόματα των συναρτήσεων για την print αντιστοιχούν στη θέση της μνήμης όπου είναι η κάθε συνάρτηση.

```
>>> print(sayHello)
<function sayHello at 0x000000000331A128>
```

**Εικόνα 4.16** *Εκτύπωση της θέσης μνήμης που αντιστοιχεί στη συνάρτηση "sayHello"*

Οι λόγοι, λοιπόν, για τους οποίους θέλουμε να έχουμε μια συνάρτηση είναι ότι: Πρώτα, ομαδοποιεί ένα σύνολο εντολών, έπειτα, κάνει το πρόγραμμα μικρότερο γιατί δε γράφουμε πολλές φορές τις ίδιες ομάδες εντολών, άρα είναι πιο εύκολο στην ανάγνωση και, επομένως, διευκολύνεται η κατανόηση. Τέλος, οι καλοσχεδιασμένες συναρτήσεις, οι οποίες είναι αρκετά γενικές, μπορούν να χρησιμοποιηθούν και από άλλους. Σαν παράδειγμα μπορούμε να αναφέρουμε τις μαθηματικές συναρτήσεις που είδαμε ότι αναπτύχθηκαν στο πακέτο math. Στην περίπτωση του πακέτου math, προγραμματιστές διάφοροι ανά τον κόσμο συντονίστηκαν για να

γράψουν συναρτήσεις που υπολογίζουν ημίτονο, συνημίτονο, κλπ. Όταν ο προγραμματισμός τους τελείωσε, δοκιμάστηκε από πολλούς άλλους προγραμματιστές και όταν οι συναρτήσεις κρίθηκαν ικανοποιητικές, αποφασίστηκε να ενταχθούν στο πακέτο των μαθηματικών συναρτήσεων της Python.

Συνοπτικά, λοιπόν, ισχύουν τα παρακάτω αναφορικά με τις συναρτήσεις:

- Μια συνάρτηση ομαδοποιεί ένα σύνολο εντολών, με αποτέλεσμα οι συναρτήσεις να απλοποιούν ένα πρόγραμμα, κρύβοντας πολύπλοκους υπολογισμούς πίσω από μια απλή εντολή κλήσης συνάρτησης.
- Οι συναρτήσεις διευκολύνουν την ανάγνωση, κατανόηση και διόρθωση ενός προγράμματος.
- Ο ορισμός μιας νέας συνάρτησης μπορεί να κάνει ένα πρόγραμμα μικρότερο, απομακρύνοντας τμήματα κώδικα που επαναλαμβάνονται. Οι τυχόν διορθώσεις ή αλλαγές γίνονται σε ένα και μόνο μέρος του προγράμματος.
- Οι καλοσχεδιασμένες συναρτήσεις είναι, συνήθως, χρήσιμες για πολλά προγράμματα (επαναχρησιμοποίηση).

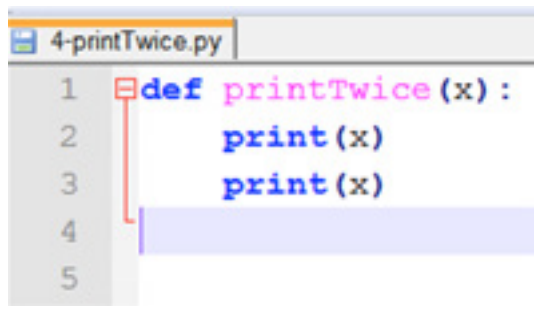
Υπάρχει, λοιπόν, και μια έννοια κοινωνική, μια έννοια συλλογικότητας κοινότητας γύρω από τον ορισμό συναρτήσεων. Αν οι συναρτήσεις είναι καλά ορισμένες, αν είναι γενικές αρκετά, αν είναι χρήσιμες για πολλούς χρήστες και έχουν τεσταριστεί και ελεγχθεί, τότε μια συνάρτηση που, ενδεχομένως, να γράψουμε, μπορεί να γίνει κομμάτι όλης της κοινότητας και πολύ άλλοι να την χρησιμοποιούν. Ουσιαστικά η συνάρτηση γίνεται κομμάτι ενός παζλ και πολύ άλλοι προγραμματιστές μπορούν μετά να την χρησιμοποιήσουν και να «χτίσουν» μεγαλύτερα προγράμματα.

### 4.1.2 Παράμετροι συναρτήσεων και τοπικές μεταβλητές

Όπως ήδη είδαμε, μερικές συναρτήσεις για να είναι αποτελεσματικές χρειάζονται ορίσματα π.χ. για τη συνάρτηση του ημιτόνου (math.sin) χρειαζόμαστε μια αριθ-

μητική τιμή ως όρισμα. Μερικές συναρτήσεις χρειάζονται περισσότερα ορίσματα, π.χ. η math.pow (ύψωση σε δύναμη) παίρνει δύο, τη βάση και τον εκθέτη.

Όταν καλείται μια συνάρτηση, στα ορίσματα αντιστοιχίζονται μεταβλητές που λέγονται **παράμετροι,** και χρησιμοποιούνται στο εσωτερικό της συνάρτησης. Παράδειγμα συνάρτησης ορισμένης από το χρήστη που χρησιμοποιεί ένα όρισμα παρατίθεται αμέσως μετά:




**Εικόνα 4.17** *Η συνάρτηση printTwice ορισμένη από το χρήστη*

Η συνάρτηση **printTwice** παίρνει ένα μοναδικό όρισμα και το εκχωρεί σε μια παράμετρο με όνομα x. Η τιμή της παραμέτρου (που στη φάση του ορισμού της συνάρτησης δε γνωρίζουμε ποια είναι) τυπώνεται δύο φορές κατά την κλήση της συνάρτησης. Αν της δώσουμε ως όρισμα το 15, θα τυπώσει δύο φορές . Αν της δώσουμε την ομάδα χαρακτήρων "Hello", θα τυπώσει δύο φορές το Hello και άμα της δώσουμε το x θα τυπώσει δύο φορές το x.

Η συνάρτηση printTwice δουλεύει με κάθε τύπο που μπορεί να τυπωθεί (Εικόνα 4.18).

```
>>> import math
>>> printTwice(15)
15
15
>>> printTwice("Hello")
Hello
Hello
>>> printTwice(math.pi)
3.14159265359
3.14159265359
```

**Εικόνα 4.18** *Η συνάρτηση printTwice σε χρήση*

Μπορούμε, επίσης, να έχουμε μεταβλητή για όρισμα (Εικόνα 4.19).

```
>>> message="Hello, World!"
>>> printTwice(message)
Hello, World!
Hello, World!
```

**Εικόνα 4.19** *Η συνάρτηση printTwice με μεταβλητή ως όρισμα*

Παρατηρήστε κάτι ιδιαίτερα σημαντικό εδώ: το όνομα της μεταβλητής που περνάμε ως όρισμα (message) δεν έχει καμία απολύτως σχέση με το όνομα της παραμέτρου (x). Από την άλλη μεριά, μέσα στη συνάρτηση printTwice το μόνο που ξέρουμε είναι η παράμετρος x.

Όταν ορίζουμε μια συνάρτηση, μπορούμε να ορίζουμε στο σώμα της συνάρτησης τοπικές μεταβλητές. Οι τοπικές αυτές μεταβλητές έχουν νόημα μόνο μέσα στον ορισμό της συνάρτησης και όχι έξω από αυτήν. Παράδειγμα στην Εικόνα 4.20, η μεταβλητή cat (από το concatenation) σημαίνει: να πάρουμε 2 ομάδες χαρακτήρων και να βάλουμε τη μία δίπλα στην άλλη. Στο παράδειγμα με το "Hello", "World" βάζουμε τα δύο μαζί και γίνεται "HelloWorld".

Οι παράμετροι είναι, επίσης, τοπικές. Για παράδειγμα, έξω από τη συνάρτηση printTwice δεν ορίζεται η μεταβλητή x. Η τοπική μεταβλητή «υπάρχει» μόνο μέσα στη συνάρτηση και δεν μπορούμε να την χρησιμοποιήσουμε έξω από αυτή. Έτσι, όταν η catTwice τερματιστεί, η μεταβλητή cat καταστρέφεται. Εάν προσπαθήσουμε να την τυπώσουμε, παίρνουμε λάθος (Εικόνα 4.21).

```
>>> def catTwice(part1, part2):
    cat= part1 + part2
    printTwice(cat)


>>> catTwice("Hello","World")
HelloWorld
HelloWorld
```

**Εικόνα 4.20** *Παράδειγμα τοπικής μεταβλητής μέσα σε μια συνάρτηση*

```
>>> print(cat)

Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    print(cat)
NameError: name 'cat' is not defined
```

**Εικόνα 4.21** *Σφάλμα κλήσης τοπικής μεταβλητής*

### 4.1.3 Διαγράμματα στοίβας

Κάθε κλήση μιας συνάρτησης αντιστοιχεί σε ένα **πλαίσιο**, το οποίο περιλαμβάνει το όνομα της συνάρτησης, τις παραμέτρους και τις μεταβλητές της συνάρτησης αυτής, όπως χρησιμοποιούνται κατά τη διάρκεια της συγκεκριμένης κλήσης. Το πλαίσιο που αναπαριστά την κλήση μιας συνάρτησης αντιστοιχεί, επίσης, και σε ένα «κομμάτι» μνήμης που καταλαμβάνει ο διερμηνευτής της Python, για να αποθηκεύσει τις παραμέτρους και τις μεταβλητές της συνάρτησης. Στη συνέχεια θα αναπαριστούμε ένα πλαίσιο ως παραλληλόγραμμο με αυτήν την πληροφορία εντός του.

Για να μπορούμε να παρακολουθούμε ποιες μεταβλητές και πού χρησιμοποιούνται κατά τη διάρκεια εκτέλεσης ενός προγράμματος, χρησιμοποιούμε το **διάγραμμα στοίβας,** το οποίο περιλαμβάνει μια ακολουθία πλαισίων κλήσης συναρτήσεων. Τα διαγράμματα στοίβας δείχνουν την τιμή της κάθε μεταβλητής, αλλά και τη συνάρτηση στην οποία ανήκει αυτή η μεταβλητή.

Στις Εικόνες 4.22 και 4.23 παρουσιάζουμε ένα ολοκληρωμένο παράδειγμα σε μορφή σεναρίου και το αποτέλεσμα της εκτέλεσής του, ενώ στην Εικόνα 4.24 τα διαγράμματα στοίβας μιας εκτέλεσής του.

4-twice.py

```
1  def printTwice(x):
2      print(x)
3      print(x)
4
5  def catTwice(part1, part2):
6      cat= part1 + part2
7      printTwice(cat)
8
9  message1="Hello"
10 message2="World"
11 catTwice(message1, message2)
```

**Εικόνα 4.22** *Ολοκληρωμένο παράδειγμα σε μορφή σεναρίου*

```
>>>
HelloWorld
HelloWorld
```

**Εικόνα 4.23** *Αποτέλεσμα εκτέλεσης του σεναρίου*

| | | | |
|---|---|---|---|
| ‹module› | message1 | -› | “Hello” |
| _main_ | message2 | -› | “World” |
| catTwice | part1 | -› | “Hello” |
| | part2 | -› | “World” |
| | cat | -› | “HelloWorld” |
| printTwice | x | -› | “HelloWorld” |

**Εικόνα 4.24** *Διαγράμματα στοίβας του σεναρίου*

Αν υπάρξει λάθος κατά την εκτέλεση μιας κλήσης συνάρτησης, η Python τυπώνει το όνομα της συνάρτησης, το όνομα της συνάρτησης που την κάλεσε, κ.ο.κ. μέχρι το κυρίως πρόγραμμα. Αυτή η λίστα ονομάζεται και **ίχνος (traceback)**. Παρατηρήστε πόσο κοντινές είναι οι έννοιες του ίχνους και του διαγράμματος στοίβας. Για παράδειγμα ας προσπαθήσουμε να «δούμε» την τιμή της cat μέσα από την **printTwice**:

4-twice.py

```
1  def printTwice(x):
2      print(cat)
3      print(x)
4      print(x)
5
6  def catTwice(part1, part2):
7      cat= part1 + part2
8      printTwice(cat)
9
10 message1="Hello"
11 message2="World"
12 catTwice(message1, message2)
```

```
Traceback (most recent call last):
  File "C:\Python_book\4-twice.py", line 12, in <module>
    catTwice(message1, message2)
  File "C:\Python_book\4-twice.py", line 8, in catTwice
    printTwice(cat)
  File "C:\Python_book\4-twice.py", line 2, in printTwice
    print(cat)
NameError: global name 'cat' is not defined
```

**Εικόνα 4.25** *Παράδειγμα ίχνους (traceback)*

## 4.2 Εκτέλεση υπό συνθήκη, αναδρομή, είσοδος από το πληκτρολόγιο

Από τα πιο βασικά στοιχεία ενός προγράμματος είναι η εκτέλεση υπό συνθήκη. Πριν εισέλθουμε σε αυτό ωστόσο, ας μιλήσουμε για το πώς φτιάχνουμε τις συνθήκες μέσω λογικών εκφράσεων. Ένα χρήσιμο στοιχείο στην κατασκευή λογικών εκφράσεων είναι ο τελεστής modulus ή το υπόλοιπο, η εύρεση υπολοίπου από διαίρεση.

### 4.2.1 Ο τελεστής modulus

Ο τελεστής **modulus** εφαρμόζεται σε ακεραίους (και εκφράσεις ακεραίων) και έχει ως αποτέλεσμα το υπόλοιπο της διαίρεσης του πρώτου ακέραιου από το δεύτερο. Στην Python το σύμβολό του είναι το **%**. Στην Εικόνα 4.26 παίρνουμε πρώτα το πηλίκο της διαίρεσης του 7/3, το οποίο είναι 2, και μετά το υπόλοιπο του ιδίου, το οποίο είναι 1.

```
>>> quotient=7//3
>>> print(quotient)
2
>>> remainder=7%3
>>> print(remainder)
1
```

**Εικόνα 4.26** *Παραδείγματα τελεστή modulus*

Ο τελεστής modulus είναι εξαιρετικά χρήσιμος. Για παράδειγμα, για να ελέγξουμε κατά πόσο ο αριθμός x είναι διαιρετός από τον αριθμό y, εκτελούμε x % y και αν το αποτέλεσμα είναι μηδέν, τότε ο y διαιρεί τον x. Επίσης, μπορούμε να εξαγάγουμε το πιο δεξιό ψηφίο ενός ακεραίου με x % 10, ή τα δύο τελευταία ψηφία ενός ακεραίου με x % 100. (Εικόνα 4.27).

```
>>> print(37%10)
7
>>> print(375%100)
75
```

**Εικόνα 4.27** *Παραδείγματα τελεστή modulus*

### 4.2.2 Λογικές εκφράσεις και τελεστές

Μια λογική έκφραση είναι είτε αληθής (true) είτε ψευδής (false). Ένας τρόπος να γράψουμε λογική έκφραση είναι με χρήση του τελεστή == που συγκρίνει δύο τιμές και παράγει μια λογική (boolean) τιμή (Εικόνα 4.28).

```
>>> 5==5
True
>>> 5==6
False
```

**Εικόνα 4.28** *Παραδείγματα τελεστή ==*

Στην πρώτη εντολή οι δύο αριθμοί είναι ίσοι, άρα η τιμή της λογικής έκφρασης είναι **True**, ενώ στη δεύτερη εντολή οι δύο αριθμοί δεν είναι ίσοι, άρα παίρνουμε **False**. Οι **True** και **False** είναι ειδικές τιμές που αποτελούν μέρος της Python (λέξεις-κλειδιά) και ανήκουν στον τύπο **bool** (Εικόνα 4.29).

```
>>> type(True)
<type 'bool'>
>>> type(False)
<type 'bool'>
```

**Εικόνα 4.29** *Οι ειδικές τιμές True, False*

Βλέπετε ότι εκτός από το == έχουμε το != που σημαίνει διαφορετικό, μεγαλύτερο, μικρότερο. Αν θέλουμε να εκφράσουμε την σχέση «μεγαλύτερο ή ίσο» ο αντίστοιχος τελεστής αποτελείται από τα σύμβολα μεγαλύτερο (>) ακολουθούμενο

από το ίσο (=). Αν βάζαμε τα σύμβολα αυτά με την αντίθετη σειρά (πρώτα το ίσο και μετά το μεγαλύτερο) ο διερμηνέας της Python θα μας έβγαζε λάθος.

Ο τελεστής == είναι ένας από τους τελεστές σύγκρισης, οι υπόλοιποι παρουσιάζονται στην Εικόνα 4.30.

| | |
|---|---|
| x!=y | # x δεν είναι ίσος με y |
| x>y | # x είναι μεγαλύτερος από y |
| x<y | # x είναι μικρότερος από y |
| x>=y | # x είναι μεγαλύτερος ή ίσος με y |
| x<=y | # x είναι μικρότερος ή ίσος με y |

**Εικόνα 4.30** *Τελεστές σύγκρισης*

Κάποια στοιχεία για τους τελεστές σύγκρισης είναι τα εξής:

- Προσοχή να μη γίνεται σύγχυση μεταξύ των συμβόλων της Python και ίδιων ή παρόμοιων μαθηματικών συμβόλων.
- Συνηθισμένο λάθος: χρήση του απλού μαθηματικού συμβόλου ισότητας = αντί του συμβόλου ισότητας στην Python που είναι ==.
- = σημαίνει εκχώρηση στη Python, ενώ == είναι τελεστής ισότητας.
- Επίσης δεν υπάρχουν τα =< και =>.

Μπορούμε να συνδυάζουμε τέτοιους ελέγχους ισότητας σε λογικές εκφράσεις με χρήση του λογικού «and», του «or», και του «not». Το «and» σημαίνει ότι, εάν έχω δύο λογικές εκφράσεις με ένα «and» ανάμεσά τους, τότε ο συνδυασμός είναι αληθής μόνον όταν και οι δύο εκφράσεις είναι αληθείς. Όταν βάλω «or» ανάμεσά τους, τότε αρκεί η μία από τις δύο εκφράσεις να είναι αληθής, για να είναι ο συνδυασμός αληθής. Και το «not» κάνει αντιστροφή, αν η έκφραση ήταν αληθής γίνεται ψευδής και εάν ήταν ψευδής γίνεται αληθής.

Για παράδειγμα η έκφραση x > 0 and x < 10 είναι αληθής μόνον όταν το x είναι μεγαλύτερο του 0 και μικρότερο του 10. Επίσης, n%2 == 0 or n%3 == 0, είναι αλη-

θής, όταν, τουλάχιστον, μία από τις συνθήκες είναι αληθής, δηλαδή, αν ο αριθμός n διαιρείται από το 2 ή το 3. Τέλος, η τιμή της λογικής έκφρασης not(x > y) είναι αληθής αν (x > y) είναι ψευδής, δηλαδή, αν το x είναι μικρότερο ή ίσο του y. Ειδικότερα, οι λογικοί τελεστές πρέπει να εφαρμόζονται πάνω σε λογικές εκφράσεις. Όμως, η Python δεν είναι αυστηρή γλώσσα και ερμηνεύει (σε λογικές εκφράσεις) κάθε μη μηδενικό αριθμό ως True.

Σε μια έκφραση με το λογικό τελεστή and, όπως η x and y, ο διερμηνέας της Python αποτιμά πρώτα το x και αν είναι ψευδές, επιστρέφει ψευδές (0). Στη συνέχεια αποτιμά το y και επιστρέφει το αποτέλεσμα (y ή 0). Στην Εικόνα 4.31 βλέπουμε στο πρώτο παράδειγμα μια αληθή έκφραση στην οποία ο διερμηνέας επιστρέφει τη δεύτερη τιμή και στο δεύτερο παράδειγμα μια ψευδή πρόταση η οποία επιστρέφει 0.

```
>>> x=1
>>> x and 2
2
>>> y = 0
>>> y and 2
0
```

**Εικόνα 4.31** *Λογικοί τελεστές*

### 4.2.3 Εκτέλεση υπό συνθήκη

Για να μπορούμε να δημιουργούμε χρήσιμα προγράμματα, σχεδόν πάντα χρειαζόμαστε τη δυνατότητα να ελέγχουμε την ισχύ συνθηκών και να αλλάζουμε τη συμπεριφορά του προγράμματος ανάλογα με το αποτέλεσμα του ελέγχου. Οι εντολές συνθήκης μας δίνουν αυτήν τη δυνατότητα. Στην απλούστερή τους μορφή έχουμε την εντολή if:

```
>>> x=5

>>> if x > 0:
        print("x is positive")


x is positive
```

**Εικόνα 4.32** *Η εντολή if*

Η λογική έκφραση μετά το if ονομάζεται συνθήκη. Αν είναι αληθής, τότε εκτελούνται οι εντολές που είναι μετατοπισμένες προς τα δεξιά. Αν είναι ψευδής, δεν εκτελείται τίποτε. Η εντολή if έχει γενικά τη μορφή μιας σύνθετης εντολής (compound statement), όπως παρουσιάζεται στην Εικόνα 4.33.

```
HEADER:
    FIRST STATEMENT
    ...
    LAST STATEMENT
```

**Εικόνα 4.33** *Μορφή σύνθετης εντολής*

**Η εντολή επικεφαλίδα (header)** ξεκινά πάντοτε σε νέα γραμμή και τελειώνει με άνω και κάτω τελεία (:). Οι προς τα δεξιά μετατοπισμένες εντολές που ακολουθούν, αποτελούν ένα μπλοκ εντολών. Η πρώτη εντολή που ευθυγραμμίζεται με το αριστερό περιθώριο είναι η πρώτη εντολή έξω από το μπλοκ. Ένα μπλοκ εντολών μέσα σε μια σύνθετη εντολή ονομάζεται το «σώμα» (body) της εντολής. Δεν υπάρχει όριο στο πλήθος των εντολών που μπορούν να εμφανιστούν στο «σώμα» μιας εντολής if, αλλά πρέπει να υπάρχει τουλάχιστον μία εντολή. Αν παρόλα αυτά χρειαζόμαστε ένα «σώμα» χωρίς εντολές (π.χ. για να θυμηθούμε αργότερα να πάμε και να συμπληρώσουμε εντολές), τότε μπορούμε να χρησιμοποιήσουμε την εντολή pass που δεν κάνει τίποτε.

Μια δεύτερη μορφή της εντολής if είναι η εναλλακτική εξέταση, κατά την οποία υπάρχουν δύο δυνατότητες (ή κλάδοι εκτέλεσης) και η συνθήκη καθορίζει ποια εκτελείται. Η σύνταξη παρουσιάζεται στην Εικόνα 4.34.

```
>>> x=4
>>> if x % 2== 0:
        print(x, "is even")
else:
        print(x, "is odd")

(4, 'is even')
```

```
>>> x =5
>>> if x % 2== 0:
        print(x, "is even")
else:
        print(x, "is odd")

(5, 'is odd')
```

**Εικόνα 4.34** *Εναλλακτική εκτέλεση*

Επίσης, μπορούμε να έχουμε αλυσιδωτές συνθήκες στο πρόγραμμά μας. Μπορούμε να χρησιμοποιήσουμε την εντολή elif, που αποτελεί συντόμευση της εντολής else if, χωρίς να υπάρχει κάποιο όριο στο πλήθος των εντολών elif τις οποίες μπορούμε να χρησιμοποιήσουμε. Ένα παράδειγμα παρουσιάζεται στην Εικόνα 4.35, όπου ένας μόνον κλάδος εκτελείται.

4-if_elif.py

```
1  x=7
2  y=4
3
4  if x<y:
5      print(x, "is less than", y)
6  elif x>y:
7      print(x, "is greater than", y)
8  else:
9      print(x, "and", y, "are equal")
```

Εκτέλεση

```
>>>
(7, 'is greater than', 4)
```

**Εικόνα 4.35** *Εντολή elif*

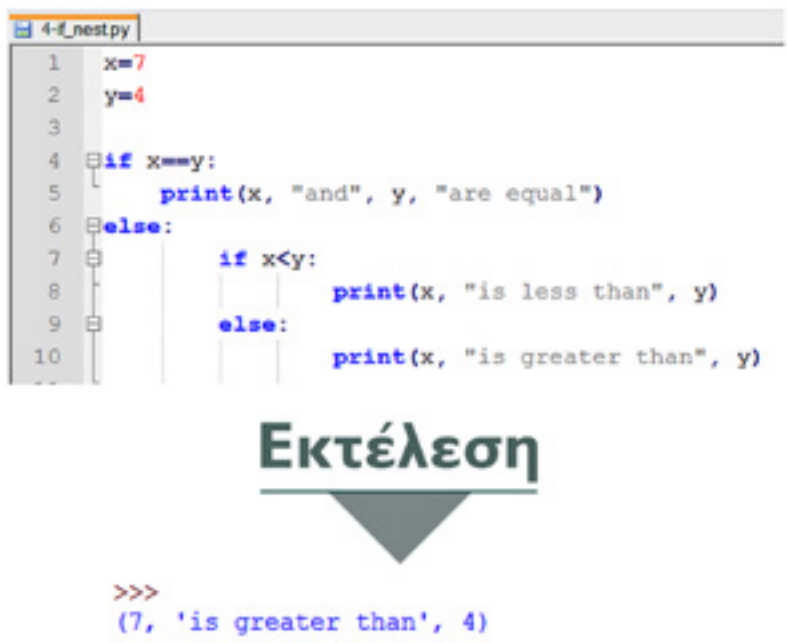
Ένα ακόμη παράδειγμα αλυσιδωτής συνθήκης παρουσιάζεται στην Εικόνα 4.36. Κάθε συνθήκη εξετάζεται με τη σειρά, αν η πρώτη είναι ψευδής πάμε στη δεύτερη, κ.ο.κ. Αν κάποια συνθήκη είναι αληθής, τότε ο αντίστοιχος κλάδος εκτελείται και η εντολή τελειώνει, ακόμη και αν άλλες συνθήκες (παρακάτω) είναι αληθείς.

4-if_elif_chain.py

```
1  if choice == 'A':
2      functionA()
3  elif choice == 'B':
4      functionB()
5  elif choice == 'C':
6      functionC()
7  else:
8      print('Invalid choice.')
```

**Εικόνα 4.36** *Παράδειγμα αλυσιδωτής συνθήκης με την εντολή elif*

Ένας άλλος τρόπος γραφής συνθηκών είναι οι εμφωλευμένες (nested conditionals). Παράδειγμα εμφωλευμένων συνθηκών φαίνεται στην Εικόνα 4.37. Οι **εμφωλευμένες συνθήκες** κάνουν το πρόγραμμα δυσανάγνωστο και γι' αυτό **συνίσταται να αποφεύγονται.**




**Εικόνα 4.37** *Παράδειγμα εμφωλευμένων συνθηκών*

Στο παράδειγμα της Εικόνας 4.38 βλέπετε ένα παράδειγμα προγραμματισμού όπου αν και οι εκφράσεις είναι ορθές, υπάρχει συνοπτικότερος τρόπος να εκφραστεί το ίδιο αποτέλεσμα. Αυτό γιατί οι δύο συνθήκες μπορούν να συνδυαστούν έτσι ώστε να ελέγχεται αν συμβαίνουν και τα δυο ταυτόχρονα, δηλαδή και το x να είναι μεγαλύτερο του 0, και το x μικρότερο του 10, όπως φαίνεται στην Εικόνα 4.39.


**Εικόνα 4.38** *Παράδειγμα εμφωλευμένων συνθηκών*

Οι λογικοί τελεστές, λοιπόν, βοηθούν στην απλοποίηση των εμφωλευμένων συνθηκών. Η εντολή εκτύπωσης της Εικόνας 4.39 εκτελείται μόνον αν περάσουμε επιτυχώς και τους δύο ελέγχους, χρησιμοποιώντας το λογικό τελεστή and.

```
if 0 < x and x < 10 :
    print("x is a positive single digit.")
```

**Εικόνα 4.39** *Παράδειγμα εμφωλευμένων συνθηκών ξαναγραμμένο με χρήση λογικών τελεστών*

Η Python δέχεται, επίσης, και το εξής:

```
if 0 < x < 10 :
    print("x is a positive single digit.")
```

**Εικόνα 4.40** *Δεύτερο παράδειγμα εμφωλευμένων συνθηκών ξαναγραμμένο με χρήση λογικών τελεστών*

Μία χρήσιμη εντολή είναι η εντολή return. Μάς επιτρέπει τον τερματισμό της εκτέλεσης μιας συνάρτησης πριν φθάσουμε το τέλος του κώδικα της συνάρτησης αυτής, π.χ. διότι ανακαλύψαμε λάθος (Εικόνα 4.41).

4-return_function.py

```
1  import math
2
3  def printLogarithm(x):
4      if x<=0:
5          print("Positive numbers only, please.")
6          return
7      result=math.log(x)
8      print("The log of x is", result)
9
10 printLogarithm(-2)
11 printLogarithm(2)
```

Εκτέλεση

```
>>>
Positive numbers only, please.
('The log of x is', 0.6931471805599453)
```

**Εικόνα 4.41** *Η εντολή return*

Μάθαμε, λοιπόν, και την εκτέλεση υπό συνθήκη, οπότε μπορούμε να προχωρήσουμε σε μια ιδιαιτέρως ενδιαφέρουσα έννοια, αυτή της αναδρομής (recursion).

### 4.2.4 Αναδρομή

Έχουμε πει, ήδη, ότι είναι νόμιμο μια συνάρτηση να καλεί μια άλλη. Αλλά είναι, επίσης, νόμιμο μια συνάρτηση να καλεί τον εαυτό της. Μπορεί να μην είναι προφανές γιατί αυτό μπορεί να είναι κάτι καλό, αλλά σίγουρα είναι ένα από τα πιο «κομψά» πράγματα που μπορούμε να κάνουμε με μια γλώσσα προγραμματισμού. Η διαδικασία όπου μια συνάρτηση καλεί τον εαυτό της λέγεται **αναδρομή** και οι συναρτήσεις αυτές λέγονται **αναδρομικές**.

Παράδειγμα αναδρομής είναι η συνάρτηση countdown η οποία παρουσιάζεται στην Εικόνα 4.42. Η συνάρτηση αυτή μετρά αντίστροφα, όπως γίνεται για παράδειγμα πριν την εκτόξευση ενός διαστημόπλοιου: 3,2,1,0, Blastoff!

Η countdown έχει μια παράμετρο, την n, που είναι θετικός ακέραιος. Αν το n είναι 0 τότε τυπώνει "Blastoff!". Αλλιώς, τυπώνει n και μετά καλεί τη συνάρτηση countdown (τον εαυτό της, δηλαδή) περνώντας το n-1 ως όρισμα.

4-countdown.py

```
1  def countdown(n):
2      if n == 0:
3          print ("Blastoff!")
4      else:
5          print (n)
6          countdown(n-1)
```

**Εικόνα 4.42** *Παράδειγμα αναδρομής*

Τι συμβαίνει εάν την τρέξουμε με όρισμα τον αριθμό 3; countdown(3):

- Η εκτέλεση της countdown ξεκινά με n=3, και επειδή το n δεν είναι 0, τυπώνει την τιμή 3 και μετά καλεί τον εαυτό της.
- Η εκτέλεση της countdown ξεκινά με n=2, και επειδή το n δεν είναι 0, τυπώνει την τιμή 2 και μετά καλεί τον εαυτό της.
- Η εκτέλεση της countdown ξεκινά με n=1, και επειδή το n δεν είναι 0, τυπώνει την τιμή 1 και μετά καλεί τον εαυτό της.

- Η εκτέλεση της countdown ξεκινά με n=0, και επειδή το n είναι 0, τυπώνει τη λέξη "Blastoff!" και μετά επιστρέφει.
- Η countdown με n=1 επιστρέφει.
- Η countdown με n=2 επιστρέφει.
- Η countdown με n=3 επιστρέφει.
- Και τώρα είμαστε πίσω στο main.

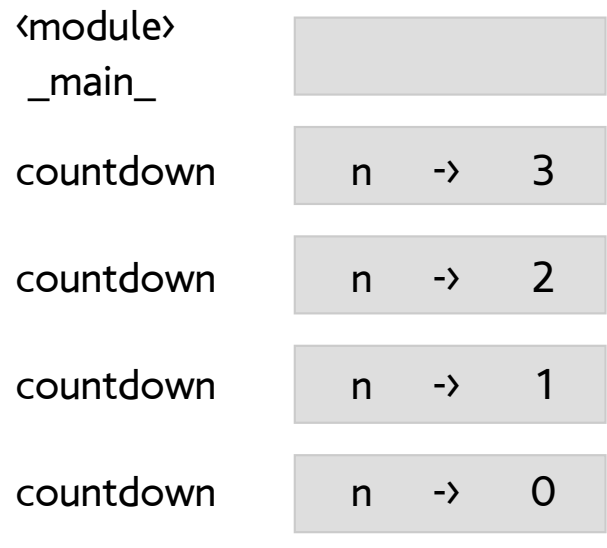
Άρα συνολικά η έξοδος φαίνεται ως εξής:

```
>>> countdown(3)
3
2
1
Blastoff!
```

**Εικόνα 4.43** *Έξοδος συνάρτησης countdown με όρισμα n=3*

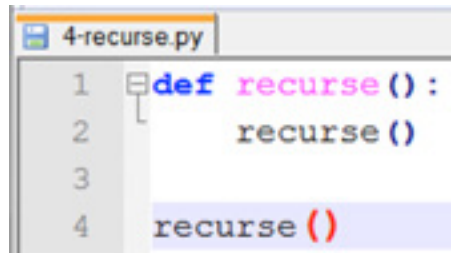
Κάθε κλήση αντιστοιχεί και σε διαφορετικό πλαίσιο στη στοίβα. Σε κάθε κλήση της συνάρτησης παράγεται και ένα διαφορετικό πλαίσιο. Άρα κάνοντας αυτήν την αντιστοίχιση μπορούμε πολύ καθαρά σε κάθε πλαίσιο να κρατάμε τις τιμές των τοπικών μεταβλητών σε αυτήν την κλήση. Και φτάνουμε μέχρι το τελευταίο σημείο που δεν καλεί πια τον εαυτό της, γιατί έχει φτάσει στη συνθήκη που σταματάει η διαδρομή οπότε και αφαιρούμε τα πλαίσια ένα-ένα, με τη σειρά με την οποία τα εισαγάγαμε (το τελευταίο πρώτο ή last-in first-out (LIFO)).

Στην Εικόνα 4.44 μπορεί να δει κανείς πώς ξεκινάμε από την αρχή, από το κυρίως πρόγραμμα. Το πρώτο πιάτο είναι το countdown με n=3, επόμενη κλωνοποίηση countdown με n=2, επόμενη κλωνοποίηση με n=1, επόμενη κλωνοποίηση με n=0. Εκεί φτάνουμε στο τέλος της αναδρομής. Τυπώνουμε το blastoff και μετά το countdown με n=0 επιστρέφει στο countdown με n=1, το countdown με n=1 επιστρέφει στο countdown με n=0 κτλ.




**Εικόνα 4.44** *Διάγραμμα στοίβας της αναδρομικής συνάρτησης countdown με όρισμα n=3*

Εάν μια αναδρομή δε φτάνει ποτέ τη λεγόμενη «περίπτωση βάσης», π.χ. το n==0 στο παράδειγμά μας, συνεχίζει να κάνει αναδρομικές κλήσεις επ' άπειρον και το πρόγραμμα δεν τελειώνει ποτέ. Αυτό λέγεται άπειρη αναδρομή και εν γένει δεν είναι καθόλου καλή ιδέα! Παράδειγμα άπειρης αναδρομής είναι η εξής:




**Εικόνα 4.45** *Παράδειγμα άπειρης αναδρομής*

Στις περισσότερες γλώσσες προγραμματισμού η άπειρη αναδρομή δεν επιτρέπεται, γιατί απλούστατα δεν μπορεί να επιτραπεί άπειρα μεγάλη στοίβα (κάποτε θα τελειώσει η μνήμη του υπολογιστή). Στην Python, ένα παράδειγμα σφάλματος είναι το εξής:

```
  File "C:\Python_book\4-recurse.py", line 2, in recurse
    recurse()
  File "C:\Python_book\4-recurse.py", line 2, in recurse
    recurse()
  File "C:\Python_book\4-recurse.py", line 2, in recurse
    recurse()
  File "C:\Python_book\4-recurse.py", line 2, in recurse
    recurse()
  File "C:\Python_book\4-recurse.py", line 2, in recurse
    recurse()
  File "C:\Python_book\4-recurse.py", line 2, in recurse
    recurse()
  File "C:\Python_book\4-recurse.py", line 2, in recurse
    recurse()
  File "C:\Python_book\4-recurse.py", line 2, in recurse
    recurse()
  File "C:\Python_book\4-recurse.py", line 2, in recurse
    recurse()
  File "C:\Python_book\4-recurse.py", line 2, in recurse
    recurse()
  File "C:\Python_book\4-recurse.py", line 2, in recurse
    recurse()
RuntimeError: maximum recursion depth exceeded
```

**Εικόνα 4.46** *Παράδειγμα σφάλματος άπειρης αναδρομής*

### 4.2.5 Είσοδος από το πληκτρολόγιο

Τα προγράμματα που έχουμε γράψει μέχρι τώρα δε δέχονται είσοδο από το χρήστη και άρα κάνουν το ίδιο πράγμα κάθε φορά, κάτι που είναι μάλλον περιορισμένο και πληκτικό. Η Python διαθέτει τη συνάρτηση input για αυτό. Όταν καλείται η συνάρτηση αυτή, το πρόγραμμα σταματάει και περιμένει από το χρήστη να πληκτρολογήσει κάτι. Όταν ο χρήστης πατήσει τα πλήκτρα Return ή Enter, το πρόγραμμα συνεχίζει και η input επιστρέφει αυτό που πληκτρολόγησε ο χρήστης σαν συμβολοσειρά.

```
>>> message=input()
'Hello'
>>> message
'Hello'
>>> n=input()
5
>>> n
5
```

**Εικόνα 4.47** *Η συνάρτηση input*

Πριν καλέσετε την input, είναι καλή ιδέα να τυπώσετε ένα μήνυμα το οποίο θα πληροφορεί το χρήστη τι είσοδο περιμένετε. Το μήνυμα αυτό λέγεται «prompt» και μπορούμε να το δώσουμε ως όρισμα στην input (Εικόνα 4.48).

```
>>> n=input("How old are you?")
How old are you?20
```

**Εικόνα 4.48** *Χρήση μηνύματος (prompt) στη συνάρτηση input*

Επειδή η input επιστρέφει μία συμβολοσειρά, μπορείτε να χρησιμοποιήσετε μία από τις συναρτήσεις μετατροπής τύπων, εάν η είσοδός σας είναι ακέραιος ή πραγματικός αριθμός (Εικόνα 4.49).

```
>>> x=input("Give the first number:")
Give the first number:5
>>> y=input("Give the second number:")
Give the second number:7
>>> print x,"+",y,"=",int(x)+int(y)
5 + 7 = 12
```

**Εικόνα 4.49** *Η συνάρτηση input επιστρέφει συμβολοσειρά*

## 4.3 Συναρτήσεις επιστροφής τιμής

Έχουμε δει ενσωματωμένες (built-in) συναρτήσεις της Python οι οποίες επιστρέφουν μία τιμή όταν κληθούν (π.χ. μαθηματικές συναρτήσεις). Μπορούμε, ωστόσο, να γράψουμε και δικές μας συναρτήσεις οι οποίες να επιστρέφουν τιμή. Ας θυμηθούμε αυτό που είχαμε πει για την εντολή "return" την οποία προσθέτει κανείς, συνήθως, σε μια συνάρτηση για να δείξει πού επιστρέφει ο έλεγχος στο σημείο που βρίσκεται το "return". Από το σημείο που είναι το return επιστρέφουμε στο σημείο από το οποίο κλήθηκε η συνάρτηση, και δεν εκτελούμε καμία εντολή πέραν του return. Συναρτήσεις οι οποίες επιστρέφουν τιμή ονομάζονται καρποφόρες (fruitful). Η τιμή που επιστρέφεται μπορεί να εκχωρηθεί σε μία μεταβλητή ή να χρησιμοποιηθεί ως μέρος μιας έκφρασης.

Χαρακτηριστικό παράδειγμα καρποφόρας συνάρτησης είναι αυτό της Εικόνας 4.50, στο οποίο φαίνεται η συνάρτηση area η οποία υπολογίζει το εμβαδόν ενός κύκλου. Αρχικά εισάγουμε τη βιβλιοθήκη math. Στη συνέχεια ορίζουμε τη συνάρτηση και το όρισμά της, radius, το οποίο είναι η ακτίνα σε εκατοστά. Έτσι υπολογίζουμε το π*$R^2$, ο γνωστός αριθμός π επί radius στο τετράγωνο. Αφού γίνει αυτός ο υπολογισμός και αυτό αποθηκευτεί στην τοπική μεταβλητή temp, επιστρέφουμε την τιμή που έχει η temp. Αν για παράδειγμα το radius είναι 10, η συνάρτηση επιστρέφει π*100.

4-fruitful.py

```
1  import math
2
3  def area(radius):
4      temp = math.pi*radius**2
5      return temp
```

**Εικόνα 4.50** *Παράδειγμα καρποφόρας συνάρτησης*

Τώρα αντί να έχουμε αυτήν την προσωρινή μεταβλητή, την temp, στην οποία αποθηκεύουμε το αποτέλεσμα του υπολογισμού και μετά επιστρέφουμε την τιμή της temp, μπορούμε να επιστρέψουμε κατευθείαν την τιμή της έκφρασης (Εικόνα 4.51)

και αυτή η έκφραση μπορεί να είναι όσο περίπλοκη θέλουμε. Αυτό έχει ως αποτέλεσμα ένα πιο συνοπτικό ορισμό της συνάρτησης.

4-fruitful_2.py

```
1  import math
2
3  def area(radius):
4      return math.pi*radius**2
```

**Εικόνα 4.51** *Δεύτερο παράδειγμα καρποφόρας συνάρτησης*

Από την άλλη μεριά, προσωρινές μεταβλητές, όπως η temp, κάνουν το debugging ευκολότερο, γιατί ο προγραμματιστής μπορεί να εξετάζει ενδιάμεσα αποτελέσματα μιας μεγαλύτερης και πιο περίπλοκης έκφρασης.

Μερικές φορές είναι χρήσιμο να έχουμε πολλαπλές εντολές return, μία για κάθε κλάδο συνθήκης. Μια περίπτωση συνάρτησης που ορίζουμε εμείς οι ίδιοι και στην οποία χρησιμοποιούμε πολλαπλές εντολές return, είναι η συνάρτηση absolute value, η απόλυτη τιμή (Εικόνα 4.52). Αν το x είναι μικρότερο του 0 τότε επιστρέφουμε -x. Διαφορετικά, αν το x δηλαδή είναι 0 ή μεγαλύτερο του 0, επιστρέφουμε x. Εδώ πρέπει να προσέξουμε ότι μόνον ένα από τα δύο return μπορεί να ισχύει. Αν ισχύει ότι x<0 τότε επιστρέφουμε το –x, άρα εκτελούμε τον πρώτο κλάδο. Διαφορετικά, εκτελούμε το δεύτερο κλάδο, όμως ποτέ δεν εκτελούμε και τους δύο. Και όταν επιστρέφουμε την τιμή, επιστρέφουμε και σε αυτόν που μας κάλεσε.

4-absoluteValue.py

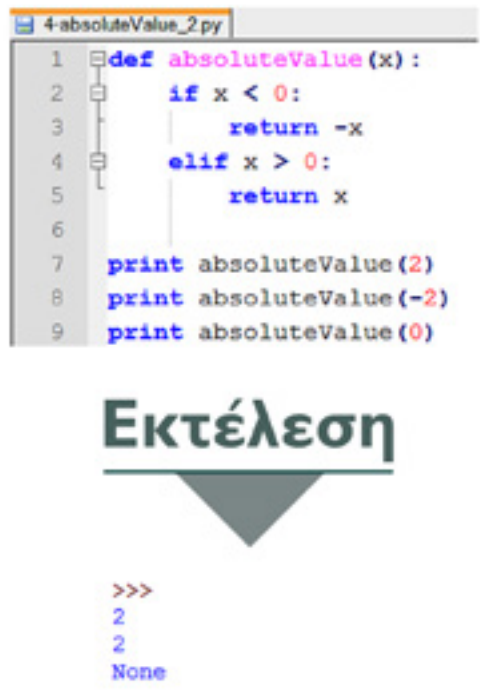
```
1  def absoluteValue(x):
2      if x<0:
3          return -x
4      else:
5          return x
```

**Εικόνα 4.52** *Πολλαπλές εντολές return*

Αν με κάποιο τρόπο ξέραμε ότι το x είναι πάντοτε θετικός ακέραιος αριθμός, τότε το πρώτο κομμάτι του if….then…else δε θα εκτελούνταν ποτέ, δε θα πήγαινε ποτέ εκεί το πρόγραμμα, αφού και το «x» θα ήταν πάντα θετικός αριθμός και

επομένως θα εκτελούσε το else. Κομμάτια κώδικα τα οποία δεν εκτελούνται ποτέ για τον έναν ή τον άλλο λόγο, λέγονται **«νεκροί κώδικες».**

Σε μια καρποφόρα συνάρτηση καλό είναι να είμαστε σίγουροι ότι κάθε δυνατό μονοπάτι εκτέλεσης μέσα από τον κώδικα της συνάρτησης καταλήγει σε εντολή return. Τι γίνεται, ωστόσο, αν δεν προβλέψουμε κάποια περίπτωση και τότε η συνάρτηση δεν έχει πού να πάει. Για παράδειγμα αν δεν είμαστε προσεχτικοί και γράψουμε τον κώδικα της Εικόνας 4.53, τι γίνεται όταν το x είναι 0; Στην περίπτωση αυτή το πρόγραμμα θα έλεγχε ότι το x δεν είναι μικρότερο του 0 αλλά ούτε μεγαλύτερο του 0, οπότε δε θα έφτανε σε κάποιο return. Στην περίπτωση αυτή η Python έχει μια ειδική τιμή που επιστρέφει για συναρτήσεις οι οποίες υποτίθεται πως πρέπει να επιστρέψουν κάτι αλλά δεν επιστρέφουν τίποτα. Η ειδική αυτή τιμή είναι η None (τίποτα). Προσέξτε ότι στην Python η τιμή *τίποτα* είναι διαφορετική από την τιμή μηδέν.




**Εικόνα 4.53** *Έλεγχος επιστροφής μιας καρποφόρας συνάρτησης*

Αυτές οι μικρές συναρτήσεις μάς δίνουν την αφορμή να μιλήσουμε για κάτι πάρα πολύ σημαντικό στον προγραμματισμό που λέγεται ανάπτυξη προγράμματος κατά στάδιο (incremental program development).

## 4.4 Ανάπτυξη προγράμματος

Για να μπορούμε να φτιάχνουμε ολοένα και πιο πολύπλοκα προγράμματα, μια τεχνική που βοηθάει, λέγεται **ανάπτυξη προγράμματος κατά στάδια** (**incremental development)**. Η ιδέα είναι να αποφύγει κανείς χρονοβόρο debugging με το να προσθέτει και να επαληθεύει μικρά κομμάτια κώδικα κάθε φορά. Η βασική ιδέα είναι απλή: ότι γράφουμε απλά προγράμματα τα οποία είναι σωστά συντακτικά, μπορεί να μην κάνουν όλα όσα θέλαμε και είχαμε κατά νου να κάνουν, αλλά επειδή είναι σωστά συντακτικά, εκτελούνται, τρέχουν, βγάζουν κάποιο αποτέλεσμα, και σιγά σιγά τα αυξάνουμε, ώστε να φτάσουμε στο επιθυμητό αποτέλεσμα. Προσθέτουμε μικρά κομμάτια προγραμματισμού, τα οποία κάθε φορά ελέγχουμε και βλέπουμε αν είναι σωστά. Και το μεγάλο πλεονέκτημα είναι ότι ακριβώς επειδή ακολουθούμε μικρά βήματα, ξέρουμε, αν προκύψει κάποιο λάθος, ποια είναι η γραμμή που προσθέσαμε η οποία προκάλεσε αυτό το λάθος: είναι η προηγούμενη ακριβώς. Γιατί αν συμπεριφερόταν μέχρι τώρα σωστά και με τη μία γραμμή παραπάνω που προσθέσαμε, άρχισε να έχει περίεργη, λάθος συμπεριφορά, τότε ξέρουμε πού θα πάμε να ψάξουμε για να βρούμε το λάθος. Αυτό είναι πάρα πολύ βοηθητικό και θα μπορούσε να είναι μια πάρα πολύ σωστή αρχή, ωστόσο δε δουλεύει πάντα.

Όλη η μεθοδολογία, όλος ο τρόπος σκέψης μας για τη λύση του προβλήματος μπορεί να είναι λάθος και να ανακαλύψουμε ότι ξεχάσαμε κάποιες περιπτώσεις, ότι ακολουθήσαμε τελείως λάθος μέθοδο καθώς θα προσθέτουμε σιγά σιγά κομματάκια. Αν, όμως, έχουμε το βασικό τρόπο σκέψης, και τη βασική λύση κατά νου, τότε αυτός ο κατά στάδια τρόπος ανάπτυξης του προγράμματος βοηθάει πάρα πολύ. Είναι κάτι το οποίο πραγματικά αν το μάθει κανείς, μπορεί να προλάβει πολλά λάθη στον προγραμματισμό.

Ας δούμε ένα σχετικά απλό παράδειγμα στο οποίο θέλουμε να υπολογίσουμε την απόσταση μεταξύ δυο σημείων (Πυθαγόρειο θεώρημα).

**Παράδειγμα:** Η απόσταση μεταξύ δύο σημείων από τις καρτεσιανές τους συντεταγμένες (x1,y1) και (x2,y2), με βάση το Πυθαγόρειο θεώρημα υπολογίζεται ως:

$$\text{distance} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

**Εικόνα 4.54** *Η απόσταση κατά το Πυθαγόρειο θεώρημα*

Στην Python αναφορικά με τη συνάρτηση distance πρέπει να λάβουμε υπόψη ποια είναι η είσοδος (παράμετροι) και ποια η έξοδος (επιστρεφόμενη τιμή). Σε αυτήν την περίπτωση οι δύο συντεταγμένες x1, x2 και y1, y2 είναι η **είσοδος** και αναπαριστώνται από τις τέσσερις παραμέτρους. **Η επιστρεφόμενη τιμή** είναι η απόσταση (distance) και πρόκειται για έναν πραγματικό αριθμό. Άρα, ήδη, μπορούμε να γράψουμε το «περίγραμμα» της συνάρτησης:

4-distance.py

```
1  def distance(x1,y1,x2,y2):
2      return 0.0
```

**Εικόνα 4.55** *"Περίγραμμα" της συνάρτησης*

Προφανώς και η μορφή αυτή της συνάρτησης δεν υπολογίζει αποστάσεις, πάντα επιστρέφει μηδέν. Αλλά είναι συντακτικά ορθή και θα τρέξει, άρα μας βοηθάει να ελέγξουμε το γενικό περίγραμμα του προγράμματός μας πριν ακόμη γίνει πολύπλοκο.

Για να ελέγξουμε τη συνάρτησή μας, την καλούμε με τιμές-δείγματα. Ας υποθέσουμε οτι οι συντεταγμένες που θα χρησιμοποιήσουμε σαν τιμές-δείγματα είναι οι (x1, y1) = (1, 2) και (x2, y2) = (3, 6).

```
4-distance.py
1  def distance(x1,y1,x2,y2):
2      return 0.0
3
4  print(distance(1,2,3,6))
```

Εκτέλεση

```
>>>
0.0
```

**Εικόνα 4.56** *Έλεγχος της συνάρτησής μας*

Ένα λογικό πρώτο βήμα στον υπολογισμό είναι να βρει κανείς τις αποστάσεις:

$$x_2 - x_1 \quad \text{και} \quad y_2 - y_1$$

**Εικόνα 4.57** *Υπολογισμός αποστάσεων*

Αποθηκεύουμε αυτές τις ενδιάμεσες τιμές στις μεταβλητές dx και dy και τις τυπώνουμε. Αυτές οι εκτυπώσεις μπορεί να μη χρειάζονται αλλά τις βάζουμε εκεί καθώς αναπτύσσουμε το πρόγραμμα, για να είμαστε σίγουροι ότι οι τοπικές αυτές μεταβλητές παίρνουν τιμές και οι τιμές αυτές είναι σωστές.

```
4-distance.py
1  def distance(x1,y1,x2,y2):
2      dx = x2 - x1
3      dy = y2 - y1
4      print("dx=",dx)
5      print("dy=",dy)
6      return 0.0
7
```

**Εικόνα 4.58** *Υπολογισμός και εκτύπωση ενδιάμεσων τιμών*

Έτσι μπορούμε να βεβαιωθούμε ότι τα dx και τα dy τα ορίσαμε σωστά. Αυτές οι επιπλέον εντολές, τα prints, λέγονται scaffolding («σκαλωσιά») γιατί όπως και στις οικοδομές βοηθούν στην κατασκευή, αλλά δεν αποτελούν τμήμα του τελικού προϊόντος. Αυτή η «σκαλωσιά», προφανώς, όταν τελειώσει η κατασκευή, φεύγει. Κατά

απολύτως αντίστοιχο τρόπο χρησιμοποιούμε τα print statements, για να μπορούμε να είμαστε βέβαιοι ότι ο κώδικας δουλεύει σωστά.

Στη συνέχεια προσθέτουμε τα δύο τετράγωνα, $dx^2+dy^2=d^2$, και τυπώνουμε το $d^2$ για να δούμε ότι όντως υπολογίζουμε σωστά το άθροισμα των δύο αυτών τετραγώνων. Τρέχουμε το πρόγραμμα που υπολογίζει το άθροισμα των τετραγώνων των dx και dy, και ελέγχουμε την έξοδο, που πρέπει να είναι 25.

4-distance.py

```
1  def distance(x1,y1,x2,y2):
2      dx = x2 - x1
3      dy = y2 - y1
4      dsquared = dx**2 + dy**2
5      print("dsquared=",dsquared)
6      return 0.0
```

**Εικόνα 4.59** *Υπολογισμός του αθροίσματος των τετραγώνων*

Τέλος, εφόσον έχουμε εισαγάγει τη μαθηματική βιβλιοθήκη, μπορούμε να υπολογίσουμε την τετραγωνική ρίζα και να επιστρέψουμε το σωστό αποτέλεσμα που είναι το result (5.0) και όχι το 0.0:

4-distance.py

```
1  import math
2
3  def distance(x1,y1,x2,y2):
4      dx = x2 - x1
5      dy = y2 - y1
6      dsquared = dx**2 + dy**2
7      result = math.sqrt(dsquared)
8      return result
9
10 print(distance(1,2,4,6))
```

Εκτέλεση

```
>>>
5.0
```

**Εικόνα 4.60** *Υπολογισμός τετραγωνικής ρίζας*

Τα κλειδιά της διαδικασίας ανάπτυξης κώδικα κατά στάδια είναι:

1. Αρχίζετε πάντα με ένα πρόγραμμα που δουλεύει και κάνετε μικρές αυξητικές αλλαγές. Αν βρείτε λάθος, ξέρετε ακριβώς πού να το διορθώσετε.
2. Χρησιμοποιείτε προσωρινές (ενδιάμεσες) μεταβλητές για ενδιάμεσες τιμές, έτσι ώστε να μπορείτε να τις τυπώνετε και να τις ελέγχετε.
3. Μόλις βεβαιωθείτε ότι το τελικό πρόγραμμα δουλεύει, μπορείτε να αφαιρέσετε τις «σκαλωσιές» και να συγκεντρώσετε πολλαπλές εντολές σε σύνθετες εκφράσεις, μόνο, βέβαια, εφόσον δεν κάνουν δύσκολη την ανάγνωση του προγράμματος.

Είναι σημαντικό να προσέξουμε και κάτι άλλο. Όταν χτίζουμε κατά αυτόν τον τρόπο, είναι σαν πρώτη προσέγγιση καλό είναι να ξέρουμε με τις εισόδους που δίνουμε στη συνάρτησή μας τι έξοδο πρέπει να έχουμε. Εδώ είναι απλό, γιατί με συντεταγμένες (x1, y1) και (x2, y2), είναι x2-x1 η μία απόσταση, δηλαδή 4-1=3, και 6-2=4, η άλλη . Έτσι 3 στο τετράγωνο 9, 4 στο τετράγωνο 16, και 9+16=25. Η ρίζα του 25 είναι το 5, άρα ξέρουμε ότι πρέπει να περιμένουμε 5 σαν απάντηση.

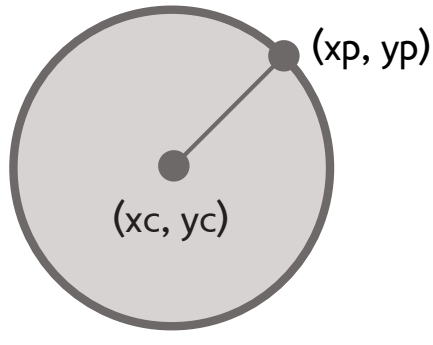
Ύστερα από κάθε αυξητική αλλαγή, ελέγχουμε και πάλι τη συνάρτηση. Αν σε κάποιο σημείο έχουμε λάθος, ξέρουμε πού πρέπει να βρίσκεται: στην τελευταία γραμμή κώδικα που προσθέσαμε!

## 4.5 Σύνθεση συναρτήσεων

Αν έχουμε, λοιπόν, δύο σημεία τα οποία είναι η αρχή και το τέλος μιας ακτίνας, μπορούμε να υπολογίσουμε πρώτα την απόσταση μεταξύ των σημείων. Στη συνέχεια λέμε ότι, αν αυτή η απόσταση είναι ακτίνα, ας βρούμε το εμβαδόν του κύκλου που ορίζεται από την τετμημένη. Αυτό μπορούμε να το πετύχουμε βάζοντας τη μία συνάρτηση να καλεί μια άλλη (πετυχαίνοντας τη σύνθεσή τους). Για παράδειγμα, ας γράψουμε συνάρτηση που λαμβάνει ως παραμέτρους δύο σημεία: το κέντρο ενός κύκλου και ένα σημείο στην περιφέρειά του και επιστρέφει το εμβαδόν του κύκλου. Αν υποθέσουμε ότι το κέντρο του κύκλου είναι αποθηκευμένο στις μεταβλητές xc και yc, και ότι το σημείο στην περιφέρεια είναι αποθηκευμένο στις xp και yp, τότε το πρώτο βήμα είναι να βρούμε την ακτίνα του κύκλου που είναι

η απόσταση μεταξύ των δύο σημείων. Χρησιμοποιούμε, λοιπόν, τη συνάρτηση distance, που κάνει ακριβώς αυτό:

```
radius = distance(xc, yc, xp, yp)
```




**Εικόνα 4.61** *Υπολογισμός εμβαδού κύκλου.*

Το δεύτερο βήμα είναι να βρούμε το εμβαδόν με αυτήν την ακτίνα και να το επιστρέψουμε:

```
result = area(radius)
return result
```

Άρα έχουμε τη συνάρτηση circle_area (Εικόνα 4.62), η οποία δε λαμβάνει πια σαν όρισμα το μήκος μιας ακτίνας αλλά τις συντεταγμένες δύο σημείων. Τα δύο αυτά σημεία τα θεωρεί, αρχή και τέλος μιας ακτίνας και υπολογίζει το εμβαδόν του κύκλου χωρίς να το δίνουμε. Η circle_area μπορεί, λοιπόν, να οριστεί χωρίς τις ενδιάμεσες μεταβλητές. Η distance παίρνει 4 συντεταγμένες και υπολογίζει το μήκος της ακτίνας ενώ η area παίρνει το μήκος της ακτίνας, υπολογίζει το εμβαδόν του κύκλου και το επιστρέφει. Αν και δεν αρχίσαμε γράφοντάς τα έτσι, με τις δοκιμές, επιμέρους εκτυπώσεις κτλ. τελικά τα συνδυάζουμε σε μια συνάρτηση:

4-circle_area.py

```
1  def circle_area(xc, yc, xp, yp):
2      radius = distance(xc, yc, xp, yp)
3      result = area(radius)
4      return result
```

**Εικόνα 4.62** *Αρχική συνάρτηση εμβαδού κύκλου*

Αφαιρούμε τώρα τις προσωρινές μεταβλητές, χρήσιμες για debugging, και έτσι κάνουμε τη συνάρτηση πιο συνοπτική:

4-circle_area.py

```
1  def circle_area(xc, yc, xp, yp):
2      return area(distance(xc, yc, xp, yp))
```

**Εικόνα 4.63** *Τελική συνάρτηση εμβαδού κύκλου*

Το πρόγραμμά μας ολοκληρωμένο έχει ως εξής:

4-circle_area.py

```
1  import math
2
3  def area(radius):
4      return math.pi*radius**2
5
6  def distance(x1, y1, x2, y2):
7      dx = x2 - x1
8      dy = y2 - y1
9      dsquared = dx**2 + dy**2
10     result = math.sqrt(dsquared)
11     return result
12
13 def circle_area(xc, yc, xp, yp):
14     return area(distance(xc, yc, xp, yp))
```

**Εικόνα 4.64** *Ολοκληρωμένο πρόγραμμα υπολογισμού εμβαδού κύκλου*

## 4.6 Λογικές συναρτήσεις

Μια συνάρτηση μπορεί να επιστρέφει οποιονδήποτε τύπο: πραγματικό, ακέραιο, ή και λογική τιμή, δηλαδή αληθές ή ψευδές. Στην Εικόνα 4.65 βλέπουμε τη συνάρτηση isDivisible η οποία εξακριβώνει αν ένας αριθμός διαιρεί κάποιον άλλο. Αυτή η συνάρτηση λαμβάνει σαν είσοδο δύο αριθμούς x και y και αν το υπόλοιπο της διαίρεσης του x με το y είναι 0 (δηλαδή αν το y διαιρεί το x), επιστρέφει True διαφορετικά επιστρέφει False.

4-isDivisible.py

```
1  def isDivisible(x,y):
2      if x % y == 0:
3          return True
4      else:
5          return False
```

**Εικόνα 4.65** *Συνάρτηση που επιστρέφει λογική τιμή*

Καθώς παρατηρούμε ότι η συνθήκη στην εντολή if είναι η ίδια λογική έκφραση, μπορούμε να κάνουμε τη συνάρτηση πιο λιτή:

4-isDivisible.py

```
1  def isDivisible(x,y):
2      return x % y == 0
```

**Εικόνα 4.66** *Η συνάρτηση isDivisible*

Στην Εικόνα 4.67 φαίνονται τα αποτελέσματα σε κάποιες περιπτώσεις εισόδων.

4-isDivisible.py

```
1  def isDivisible(x,y):
2      return x % y == 0
3
4  print(isDivisible(10, 3))
5  print(isDivisible(10, 2))
6
7  x=10
8  y=2
9
10 if isDivisible(x,y):
11     print 'Το ',x,' διαιρείται από το ',y
12 else:
13     print 'Το ',x,' δεν διαιρείται από το ',y
```

Εκτέλεση

```
False
True
Το  10  διαιρείται από το  2
```

**Εικόνα 4.67** *Η συνάρτηση isDivisible σε δράση*

## 4.7 Περισσότερη αναδρομή

Μέχρι τώρα είδαμε μόνο ένα μικρό υποσύνολο της Python, αλλά το ενδιαφέρον σημείο είναι ότι αυτό το μικρό υποσύνολο είναι μια **πλήρης** γλώσσα προγραμματισμού, υπό την έννοια ότι οτιδήποτε μπορεί να υπολογιστεί, μπορεί να εκφραστεί σε αυτό το μικρό υποσύνολο της Python! Οποιοδήποτε πρόγραμμα που γράφτηκε ποτέ, σε οποιαδήποτε γλώσσα, μπορεί να ξαναγραφεί με τη χρήση μόνον των στοιχείων της γλώσσας που έχετε διδαχτεί μέχρι τώρα (και ενδεχομένως κάποιες επιπλέον εντολές για τον έλεγχο συσκευών όπως το πληκτρολόγιο, το ποντίκι, ο σκληρός δίσκος, κλπ.). Η απόδειξη αυτού του ισχυρισμού κάθε άλλο παρά τετριμμένη είναι και έγινε για πρώτη φορά από τον Alan Turing (Lewis & Papadimitriou, 1997), έναν από τους πρώτους επιστήμονες των υπολογιστών.

Για να εκτιμήσουμε ακόμη περισσότερο τι μπορούμε να κάνουμε με αυτό το μικρό υποσύνολο της Python, θα υπολογίσουμε μερικές αναδρομικές μαθηματικές συναρτήσεις. Οι αναδρομικοί ορισμοί μοιάζουν λίγο με κυκλικούς ορισμούς, υπό την έννοια ότι ο ορισμός περιέχει αναφορά στην ίδια την έννοια που υποτίθεται ότι ορίζει (Cormen, Leiserson, Rivest, & Stein, 2009).

Οι πραγματικά κυκλικοί ορισμοί δεν είναι και πολύ χρήσιμοι (ορισμός δια του οριζομένου), όπως για παράδειγμα ένας ορισμός σε λεξικό, που παρατίθεται αμέσως μετά:

- *Καλλίμορφος: επίθετο που περιγράφει κάτι που είναι καλλίμορφο.*

Κάτι τέτοιο σε ένα λεξικό είναι ενοχλητικό αλλά από την άλλη μεριά δείτε τον ορισμό του παραγοντικού στα μαθηματικά:

$$n! = \begin{cases} 1 & \text{αν } n=0 \\ n(n-1)! & \text{αν } n>0 \end{cases}$$

**Εικόνα 4.68** *Ο ορισμός του παραγοντικού στα μαθηματικά*

Ο ορισμός λέει ότι το παραγοντικό του μηδενός είναι 1 και ότι το παραγοντικό κάθε άλλου n είναι το n πολλαπλασιαζόμενο με το παραγοντικό του n-1. Έτσι

- το 3! είναι 3 φορές το 2!
- το 2! είναι 2 φορές το 1!
- το 1! είναι 1 φορά το 0! (αλλά το 0! είναι ίσο 1, απο τον ορισμό).

Άρα: το 3! ισούται με 3 επί 2 επί 1 επί 1, δηλαδή 6.

## Παράδειγμα: παραγοντικό

Αν μπορείτε να γράψετε έναν αναδρομικό ορισμό, συνήθως, μπορείτε να γράψετε και ένα πρόγραμμα σε Python, για να το υπολογίζει. Το πρώτο βήμα είναι να βρούμε τι παραμέτρους χρειαζόμαστε. Εύκολα αποφασίζουμε:

4-factorial.py

```
1  def factorial(n):
```

**Εικόνα 4.69** *Ορισμός συνάρτησης παραγοντικού*

Εάν το όρισμα είναι μηδέν, επιστρέφουμε 1:

4-factorial.py

```
1  def factorial(n):
2      if n == 0:
3          return 1
```

**Εικόνα 4.70** *Όρισμα μηδέν στη συνάρτηση παραγοντικού*

Διαφορετικά κάνουμε αναδρομική κλήση στο factorial(n-1) και πολλαπλασιάζουμε με το n:

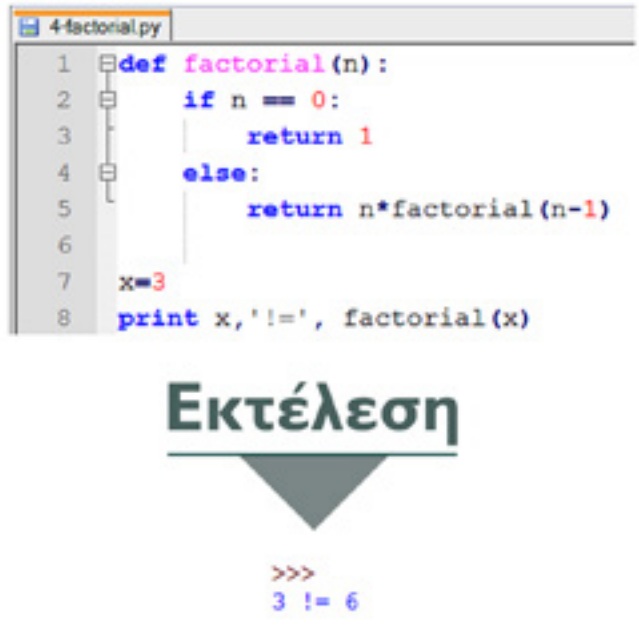
4-factorial.py

```
1  def factorial(n):
2      if n == 0:
3          return 1
4      else:
5          return n*factorial(n-1)
```

**Εικόνα 4.71** *Ολοκληρωμένος ορισμός συνάρτησης παραγοντικού*

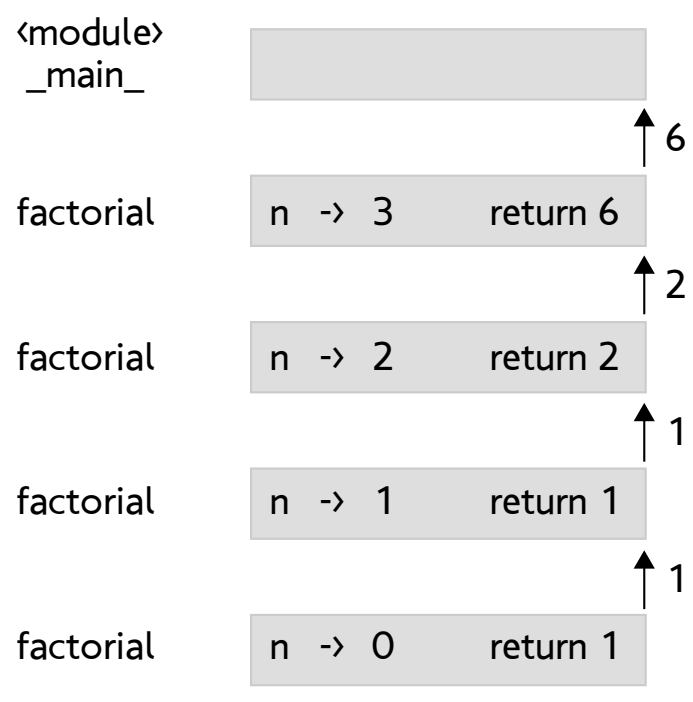
Η ροή εκτέλεσης για το 3!  (Εικόνα 4.72) είναι η εξής:

- Επειδή το 3 δεν είναι 0, πάρε το δεύτερο κλάδο και υπολόγισε το factorial του n-1...
- Επειδή το 2 δεν είναι 0, πάρε το δεύτερο κλάδο και υπολόγισε το factorial του n-1...
- Επειδή το 1 δεν είναι 0, πάρε το δεύτερο κλάδο και υπολόγισε το factorial του n-1...
- Επειδή το 0 είναι 0, πάρε τον πρώτο κλάδο και επίστρεψε 1, χωρίς άλλη αναδρομική κλήση.
- Η επιστρεφόμενη τιμή (1) πολλαπλασιάζεται με n, που είναι 1, και το αποτέλεσμα 1 επιστρέφεται.
- Η επιστρεφόμενη τιμή (1) πολλαπλασιάζεται με n, που είναι 2, και το αποτέλεσμα 2 επιστρέφεται.
- Η επιστρεφόμενη τιμή (2) πολλαπλασιάζεται με n, που είναι 3, και το αποτέλεσμα 6 γίνεται η επιστρεφόμενη τιμή της συνάρτησης που ξεκίνησε όλη τη διαδικασία.




**Εικόνα 4.72** *Παράδειγμα εκτέλεσης συνάρτησης παραγοντικού*

Στην Εικόνα 4.73 φαίνεται η γνωστή μας στοίβα η οποία αντιστοιχεί σε αυτό το παράδειγμα. Αυτό υπάρχει, επίσης, σαν διαδραστικό υλικό στο οποίο μπορείτε να δείτε πώς εξελίσσεται η στοίβα βήμα-βήμα και πώς αντιστοιχίζει με τις τιμές που παίρνει εσωτερικά στη συνάρτηση.




**Εικόνα 4.73** *Διάγραμμα στοίβας για το 3!*

## Παράδειγμα: η αναδρομή Fibonacci

Μετά το παραγοντικό, το συνηθέστερο παράδειγμα μαθηματικής συνάρτησης αναδρομικά ορισμένης, είναι η fibonacci:

fibonacci(0) = 1
fibonacci(1) = 1
fibonacci (n) = fibonacci (n -1) + fibonacci (n - 2)

Σε Python:

4-fibonacci.py

```
1  def fibonacci(n):
2      if n == 0 or n == 1:
3          return 1
4      else:
5          return fibonacci(n-1) + fibonacci(n-2)
```

**Εικόνα 4.74** *Η μαθηματική συνάρτηση Fibonacci ορισμένη στη γλώσσα Python*

## 4.8 Έλεγχος τύπων

Τι γίνεται όταν καλέσουμε την factorial με όρισμα 1.5; Αν το δοκιμάσουμε, βλέπουμε το μήνυμα της Εικόνας 4.75:

```
RuntimeError: maximum recursion depth exceeded in cmp
```

**Εικόνα 4.75** *Μήνυμα σφάλματος κατά την εκτέλεση*

Φαίνεται σαν να ξεκινά άπειρη αναδρομή. Αυτό συμβαίνει γιατί η εκτέλεση δε σταματά στην περίπτωση βάσης n == 0, αλλά την προσπερνάει! Στην πρώτη αναδρομική κλήση το n είναι 0.5, στη δεύτερη -0.5, κ.ο.κ. Το n γίνεται μικρότερο και μικρότερο αλλά ποτέ 0.

Για να λύσουμε το πρόβλημα έχουμε δύο επιλογές:

- Μπορούμε να προσπαθήσουμε να γενικεύσουμε τη συνάρτηση του παραγοντικού, ώστε να δουλεύει και με πραγματικούς αριθμούς· η συνάρτηση αυτή λέγεται συνάρτηση gamma αλλά η χρήση της υπερβαίνει τους στόχους αυτού του συγγράμματος, ή
- Να βάλουμε το factorial να ελέγχει τον τύπο του ορίσματος· αυτό θα κάνουμε εδώ.

Προσθέτουμε, λοιπόν, στον κώδικα της συνάρτησης factorial, δύο ελέγχους. Θυμηθείτε εδώ τη συνάρτηση isDivisible. Στο σημείο αυτό θέλουμε να ελέγξουμε κατά πόσο ο n είναι ακέραιος (άρα χρειαζόμαστε μια συνάρτηση "is instance", η οποία σημαίνει «είναι στιγμιότυπο από»). Μπορούμε να χρησιμοποιήσουμε την built-in συνάρτηση isinstance, για να επαληθεύσουμε τον τύπο του ορίσματος, και επιπλέον, να βεβαιωθούμε ότι το όρισμα είναι θετικό. Αυτό το παράδειγμα δείχνει ένα σχήμα (δομή, μόρφωμα, pattern) που μερικές φορές ονομάζεται «φύλακας» (guardian). Οι δύο πρώτες συνθήκες λειτουργούν ως «φύλακες» που προστατεύουν τον κώδικα ο οποίος ακολουθεί, από τιμές που μπορεί να οδηγήσουν σε λάθος. Έτσι και η ορθότητα του κώδικα μπορεί να αποδειχθεί ευκολότερα.

4-factorial_2.py

```
1  def factorial(n):
2      if not isinstance(n, int):
3          print('Το παραγοντικό ορίζεται μόνο για ακεραίους.')
4          return -1
5      elif n<0:
6          print('Το παραγοντικό ορίζεται μόνο για θετικούς ακεραίους.')
7          return- 1
8      elif n == 0:
9          return 1
10     else:
11         return n*factorial(n-1)
12
```

**Εικόνα 4.76** *Χρήση της built-in συνάρτησης isinstance*

## 4.9 Επίλογος

Σε αυτό το κεφάλαιο κάναμε μια εισαγωγή στις έννοιες της συνάρτησης, της εκτέλεσης υπό συνθήκη, και μια εκτενή μελέτη της αναδρομής. Στο επόμενο κεφάλαιο θα εμβαθύνουμε στην έννοια της επανάληψης.

## Βιβλιογραφία/Αναφορές


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

Lewis, H. R., & Papadimitriou, C. H. (1997). *Elements of the Theory of Computation*. Prentice Hall.

# Κριτήρια αξιολόγησης

## Κριτήριο αξιολόγησης Ι (Βαθμός δυσκολίας:●)

**Εκτελέστε τον εξής κώδικα:**

```
def message():
    print('This is a book')
    print(' on Python')

message()
```

**Απάντηση/Λύση**

### Κριτήριο αξιολόγησης 2 (Βαθμός δυσκολίας:●●)

**Κατασκευάστε μία συνάρτηση που θα ζητάει από το χρήστη να γράψει έναν ακέραιο αριθμό και στη συνέχεια να τον εκτυπώνει**

**Απάντηση/Λύση**

### Κριτήριο αξιολόγησης 3 (Βαθμός δυσκολίας: ●●●)

**Γράψτε ένα πρόγραμμα που ζητάει από το χρήστη να γράψει ένα κωδικό. Ο κωδικός που θα γράψει ο χρήστης συγκρίνεται με ένα συγκεκριμένο string. Αν το string είναι το σωστό στέλνεται το μήνυμα στο χρήστη ότι ο κωδικός είναι αποδεκτός. Διαφορετικά το πρόγραμμά του δείχνει μήνυμα ότι ο κωδικός είναι λάθος.**

**Απάντηση/Λύση**

# ΚΕΦΑΛΑΙΟ 5

# Επανάληψη με την εντολή while

*Σύνοψη*

*Σε αυτό το κεφάλαιο θα μελετήσουμε την επανάληψη μέσω της εντολής while και άλλες έννοιες, όπως η ενθυλάκωση και η γενίκευση.*

*Προαπαιτούμενη γνώση*

*Κεφάλαια 1-4 του παρόντος συγγράμματος.*

## 5.1 Εισαγωγή

Οι υπολογιστές, συχνά, χρησιμοποιούνται για να κάνουν επαναλαμβανόμενες (ανιαρές) εργασίες. Η επανάληψη ίδιων ή παραπλήσιων εργασιών χωρίς λάθη είναι κάτι που οι υπολογιστές κάνουν καλά, σε αντίθεση με τους ανθρώπους. Είδαμε, ήδη τη συνάρτηση countdown (Εικόνα 4.42) η οποία χρησιμοποιεί την αναδρομή για να εκτελέσει μια επαναληπτική εργασία. Επειδή η επανάληψη είναι κάτι πολύ συνηθισμένο, οι γλώσσες προγραμματισμού περιλαμβάνουν, συνήθως, ειδικές εντολές για το σκοπό αυτό.

## 5.2 Η εντολή while

Στο παράδειγμα της Εικόνας 5.2, βλέπουμε ένα παράδειγμα επανάληψης. Ορίζουμε μια συνάρτηση που την ονομάζουμε available_tomatoes που έχει μια παράμετρο, το n. Η εντολή έχει ως εξής: «Καθόσο ("while") το n είναι μεγαλύτερο του 0, συνέχισε να εκτυπώνεις την τιμή του n και μετά να μειώνεις το n κατά μία μονάδα. Όταν φτάσεις σε ένα σημείο που το n δε θα είναι μεγαλύτερο του 0, τότε βγες από το while loop και τύπωσε τη φράση 'end of tomatoes!'. Αυτή είναι η πιο απλή κατασκευή για να κάνει κανείς επαναληπτικές συναρτήσεις.

### Παράδειγμα: countdown με χρήση του while:

```
def available_tomatoes(n):
        while n>0:
         print('available tomatoes',n)
         n=n-1
         print('one tomato removed')
        print('end of tomatoes')

available_tomatoes(6)
```

**Εκτέλεση**

```
('available tomatoes', 6)
one tomato removed
('available tomatoes', 5)
one tomato removed
('available tomatoes', 4)
one tomato removed
('available tomatoes', 3)
one tomato removed
('available tomatoes', 2)
one tomato removed
('available tomatoes', 1)
one tomato removed
end of tomatoes
```

**Εικόνα 5.1** *Η συνάρτηση available_tomatoes*

Στο σημείο αυτό προσέξτε το indentation (εσοχή), δηλαδή, το ότι ο κώδικας εισέρχεται ενδότερα του σώματος του ορισμού και ακόμα πιο μέσα στο σώμα του while. Κάθε επίπεδο εσοχής αντιστοιχεί σε *4 κενά (spaces)*, όπως ορίζεται και στον οδηγό ορθής μορφοποίησης κώδικα της Python (Python Software Foundation, 2013). Συντάκτες κειμένου (editors) με ρύθμιση για την παραγωγή κώδικα Python

(όπως ο Notepad++, Ενότητα Ι.Ι) προσθέτουν αυτόματα 4 κενά σε κάθε χτύπημα του πλήκτρου tab.

Η ένδειξη ότι τελείωσε το σώμα του while είναι ότι έχει σταματήσει το indentation, έχει τελειώσει δηλαδή, η εσοχή κώδικα του while και έχουμε ξαναγυρίσει στα προηγούμενα, στην εσοχή κώδικα του countdown. Με αυτόν τον τρόπο «καταλαβαίνει» η Python ότι τελείωσε το σώμα του while. Όλες οι δηλώσεις που έχουν ίδια απόσταση προς τα δεξιά ανήκουν στο ίδιο μπλοκ κώδικα. Εάν κάποιο μπλοκ κώδικα είναι πιο βαθιά εμφωλευμένο, θα μπει η εσοχή του πιο δεξιά.

Ας εξηγήσουμε τώρα λίγο πιο αυστηρά τι σημαίνει η εντολή while: «Υπολόγισε πρώτα τη συνθήκη που είναι δίπλα στο while, η οποία θα είναι είτε αληθής είτε ψευδής. Αν η συνθήκη είναι ψευδής (FALSE, 0) τότε αυτό σημαίνει ότι τελείωσε το while loop, βγες έξω από το σώμα while και προχώρησε στην εκτέλεση της επόμενης εντολής που ακολουθεί το σώμα. Αν είναι αληθής (TRUE, Ι) η συνθήκη, τότε εκτέλεσε όλες τις εντολές που είναι μέσα στο σώμα του while και ξαναπήγαινε πίσω στο βήμα Ι, δηλαδή, υπολόγισε ξανά τη συνθήκη του while». Ο τύπος αυτός ροής ελέγχου ονομάζεται *βρόγχος*. Παρατηρήστε, βεβαίως, ότι, αν η συνθήκη είναι ψευδής από την πρώτη φορά που ελέγχεται, το «σώμα» δεν εκτελείται ποτέ, απλώς το παρακάμπτει.

Το «σώμα» του βρόγχου πρέπει να αλλάζει τις τιμές μιας ή περισσότερων μεταβλητών, τις οποίες ελέγχει η συνθήκη στην επικεφαλίδα του βρόγχου while, ώστε η συνθήκη να γίνεται τελικά ψευδής και ο βρόγχος να τελειώνει. Διαφορετικά, ο βρόγχος θα επαναλαμβάνεται επ' άπειρον, γι' αυτό και στην περίπτωση αυτή ονομάζεται **ατέρμων βρόγχος (infinite loop)**. Αναφορικά με τη συνάρτηση countdown μπορούμε να αποδείξουμε ότι ο βρόγχος τελειώνει, επειδή ξέρουμε ότι η τιμή του n είναι θετική και πεπερασμένη. Ειδικότερα, βλέπουμε ότι η τιμή του n μειώνεται κάθε φορά που εκτελείται ο βρόγχος, άρα τελικά θα γίνει 0.

Υπάρχουν κάποιες περιπτώσεις που θέλουμε να έχουμε ένα "infinite loop". Τέτοιες περιπτώσεις εμφανίζονται για παράδειγμα εντός του λειτουργικού συστήματος, το οποίο συνεχώς περιμένει για κάποια είσοδο από εμάς ή από τις διάφορες συσκευές. Αυτό αποτελεί παράδειγμα ενός επιθυμητού "infinite loop". Από την άλλη μεριά, βεβαίως, τα περισσότερα προγράμματα που θα φτιάχνουμε εμείς, θέλουμε να τελειώνουν κάποτε. Αναφορικά με το πρόγραμμα που είδαμε νωρίτερα (Εικόνα

5.2), μπορούμε άραγε να κάνουμε έναν συλλογισμό για το αν το αυτό κάποτε θα τελειώσει; Με την προϋπόθεση, βεβαίως, ότι το n αρχίζει με τιμή πάνω από το 0, γιατί αν αρχίζει με τιμή κάτω από το 0, έχουμε πρόβλημα (ο αναγνώστης καλείται να σκεφτεί μόνος του την εξήγηση). Θα ήταν πολύ καλό αν μπορούσαμε να το διαγνώσουμε αυτόματα αυτό και να πούμε: «Λάθος, εδώ περιμένω θετική και ακέραια τιμή και μου έδωσε αρνητική ή μη ακέραια» (γιατί και με αρχική τιμή του n 0.25, τότε πάλι το n δε γίνεται ποτέ 0). Ένα τέτοιου είδους επιχείρημα μπορεί να πάρει τη μορφή μαθηματικού θεωρήματος και απόδειξης, που αποδεικνύει την ορθότητα του προγράμματος.

Στην Εικόνα 5.3 βλέπουμε ένα παράδειγμα προγράμματος που δεν ξέρουμε αν θα τερματίσει κάποτε. Σε αυτό το παράδειγμα για όσο το n≠1 (n!=1), δηλαδή, για οτιδήποτε θετικό, αρνητικό, ακέραιο ή πραγματικό, τύπωσε το n. Εάν το υπόλοιπο της διαίρεσης του n διά του 2 είναι 0 (n%2==0), (με άλλα λόγια ο n είναι άρτιος) τότε διαίρεσε το n δια 2. Επειδή το n είναι άρτιος αριθμός, το αποτέλεσμα θα είναι ακέραιο. Διαφορετικά αν το n δεν είναι άρτιος (δηλαδή, το αποτέλεσμα της διαίρεσης με το 2 δεν είναι 0), τότε πολλαπλασίασε επί 3 το n και πρόσθεσε 1. Εδώ λέμε ότι, εάν το n είναι άρτιος μείωσέ το διαιρώντας διά 2, άρα, εάν είναι δύναμη του 2 και το διαιρούμε, συνεχώς, με το 2, κάποια στιγμή θα γίνει μονάδα. Έτσι θα βγούμε από εδώ. Αν, όμως, δεν είναι δύναμη του 2, κάποια στιγμή αυτό που θα προκύψει δε θα είναι διαιρετός με το 2, θα είναι περιττός και τότε θα πάρουμε το δεύτερο κλάδο του if και θα το πολλαπλασιάσουμε επί 3 και θα προσθέσουμε 1.

Αν είναι άρτιος θα ξαναρχίσει να μειώνεται, αλλά αν δεν είναι δύναμη του 2, κάποια στιγμή θα είναι περιττός. Έτσι, λοιπόν, βλέπουμε ότι το πρόγραμμα μπορεί να μην τελειώσει ποτέ.

```
def unpredictable_function(n):
        while n!=1:
         print(n)
         if n%2 == 0: #n άρτιος
          n=n//2
         else:        #n περιττός
          n=n*3+1

unpredictable_function(6)
```

**Εκτέλεση**

```
6
3
10
5
16
8
4
2
```

**Εικόνα 5.2** *Συνάρτηση με δυσκολία απόδειξης ότι θα τερματίσει ο βρόχος*

Είναι σημαντικό να τονίσουμε ότι η απόδειξη τερματισμού δεν είναι απλή. Στη γενική περίπτωση, ας αναρωτηθούμε αν θα μπορούσαμε να φτιάξουμε ένα πρόγραμμα που θα έχει σαν είσοδο ένα οποιοδήποτε άλλο πρόγραμμα, ίσως και τον εαυτό του, και σαν έξοδο μια απάντηση *ναι* ή *όχι*: *ναι* εάν τερματίζει το πρόγραμμα εισόδου, *όχι* αν δεν τερματίζει το πρόγραμμα εισόδου. Το πρόβλημα, το λεγόμενο *"halting problem"* ή το *πρόβλημα τερματισμού* στη γενική του έκφραση, αποδεικνύεται ότι δε λύνεται. Αυτό δε σημαίνει ότι αν πάρουμε ένα συγκεκριμένο πρόγραμμα δεν μπορούμε να αποδείξουμε αν τερματίζεται ή όχι. Αλλά αποδεικνύεται ότι δεν υπάρχει ένα πρόγραμμα που παίρνει σαν είσοδό του ένα οποιοδήποτε άλλο πρόγραμμα και να απαντάει *ναι* αν τερματίζει το πρόγραμμα αυτό ή *όχι* αν δεν τερματίζει. Η απόδειξη γι' αυτό ανήκει στο διάσημο μαθηματικό Alan Turing (Lewis & Papadimitriou, 1997).

### 5.2.1 Εκτύπωση πινάκων

Ένα από τα πράγματα που μπορούμε να κάνουμε καλά με βρόγχους είναι η εκτύπωση πινάκων δεδομένων. Υποθέστε ότι θέλουμε να τυπώσουμε μια λίστα λογα-

ρίθμων. Αυτή είναι μια άλλη επαναληπτική δουλειά που κάποτε αναγκαστικά έκαναν με το χέρι (ενίοτε με λάθη) άνθρωποι που δεν είχαν τίποτα άλλο να κάνουν στη ζωή τους: έγραφαν, έκαναν πράξεις, συνέγραφαν κάτι τεράστια βιβλία με πίνακες λογαριθμικούς και έλεγαν για παράδειγμα 5,34789 λογάριθμος *τόσο*, 5,345810 λογάριθμος τόσο κ.ο.κ. Όλα αυτά δε χρειάζονται σήμερα γιατί, αν θέλαμε να φτιάξουμε έναν τέτοιο πίνακα, θα μπορούσαμε πολύ εύκολα να χρησιμοποιήσουμε το πακέτο μαθηματικών της Python και να βγάλουμε τους λογαρίθμους.

Ας δούμε, λοιπόν, πώς μπορούμε να εκτυπώσουμε έναν πίνακα λογαρίθμων με τη χρήση βρόγχων. Στην Εικόνα 5.3 υπολογίζουμε το φυσικό λογάριθμο με βάση το e. Βλέπετε ένα πολύ απλό while loop πριν από το οποίο δίνουμε σε μια μεταβλητή την τιμή 1. Μετά και για όσο ισχύει x<=6.0 τυπώνουμε το λογάριθμό του. Ωστόσο ο τρόπος που τον τυπώνουμε έχει σημασία, για να αρχίσουμε να καταλαβαίνουμε πώς επιτυγχάνουμε την κατάλληλη μορφοποίηση. Ο χαρακτήρας '\t' είναι το «περιβόητο» tab ή *στηλογνώμονας* στη γραφομηχανή. Θα εξηγήσουμε τη χρήση του tab λίγο πιο κάτω.

Το επόμενο πρόγραμμα τυπώνει μια σειρά τιμών στην αριστερή κολόνα και τους λογαρίθμους τους στη δεξιά:

```
import math

x = 1.0
while x<= 6.0:
 print (x, '\t', math.log(x))
 x = x +1.0
```

**Εκτέλεση**

```
(1.0, '\t', 0.0)
(2.0, '\t', 0.6931471805599453)
(3.0, '\t', 1.0986122886681098)
(4.0, '\t', 1.3862943611198906)
(5.0, '\t', 1.6094379124341003)
(6.0, '\t', 1.791759469228055)
```

**Εικόνα 5.3** *Εκτύπωση σειράς τιμών και των λογαρίθμων*

Αλλά ας εξηγήσουμε εδώ τι κάνει το πλήκτρο tab: μετακινεί τον κέρσορα κατά μία ομάδα χαρακτήρων, μέχρι το επόμενο tab stop, όπως λέγεται, συνήθως, σ' αυτήν την ορολογία. Όλα αυτά τα δανειζόμαστε από την εποχή της γραφομηχανής κατά

την οποία υπήρχε και πάλι αυτό το tab και όταν το πατούσε ο δακτυλογράφος πήγαινε στο επόμενο tab stop. Οπότε, μπορούσες να χωρίσεις το χαρτί σου ας πούμε σε 4 tab stops, όπου δινόντουσαν οι εξής εντολές: «Με το πρώτο πήγαινε εδώ, δεύτερο πήγαινε εδώ, τρίτο πήγαινε εδώ και τέταρτο πήγαινε στο τέλος». Κατά αυτόν τον τρόπο μπορούσαν να χωρίσουν το χαρτί σε στήλες.

Ας υποθέσουμε ότι θέλουμε να υπολογίσουμε τους λογαρίθμους του 1, 2, 3, 4, 5 όχι με βάση το e, αλλά με βάση το 2. Δεδομένου ότι έχουμε τη συνάρτηση που μας δίνει η Python, βάσει του e, για να υπολογίσουμε το λογάριθμο με βάση το 2 χρησιμοποιούμε τον ακόλουθο τύπο:

$$\log_2 x = \frac{\log_e x}{\log_e 2}$$

**Εικόνα 5.4** *Τύπος του λογαρίθμου με βάση τον αριθμό 2*

Σαν εξάσκηση, δοκιμάστε να αλλάξτε το πρόγραμμα της προηγούμενης διαφάνειας, ώστε να τυπώνει λογαρίθμους με βάση το 2 (το αποτέλεσμα φαίνεται στην Εικόνα 5.5).

5-log2.py

```
1  import math
2
3  x = 1.0
4  while x<= 10.0:
5      print  x, '\t', math.log(x)/math.log(2.0)
6      x = x +1.0
7
```

Εκτέλεση

```
>>>
1.0     0.0
2.0     1.0
3.0     1.58496250072
4.0     2.0
5.0     2.32192809489
6.0     2.58496250072
7.0     2.80735492206
8.0     3.0
9.0     3.16992500144
10.0    3.32192809489
```

**Εικόνα 5.5** *Εκτύπωση λογαρίθμων με βάση τον αριθμό 2*

Στην Εικόνα 5.6 βλέπουμε ένα πρόγραμμα που τυπώνει τα πολλαπλάσια του 2:


**Εικόνα 5.6** *Εκτύπωση των πολλαπλασίων του αριθμού 2*

## 5.3 Ενθυλάκωση και γενίκευση

**Ενθυλάκωση** είναι το να «περιτυλίξουμε» ένα κομμάτι κώδικα μέσα σε μια συνάρτηση, για να εκμεταλλευτούμε όλα όσα προσφέρει μια συνάρτηση. Θυμηθείτε την isDivisible που είδαμε στο Κεφάλαιο 4 (Εικόνα 4.66). **Γενίκευση** είναι η διαδικασία κατά την οποία παίρνουμε κάτι συγκεκριμένο, όπως η εκτύπωση των πολλαπλασίων του 2, και το κάνουμε πιο γενικό, όπως π.χ. η εκτύπωση των πολλαπλασίων κάθε ακεραίου.

Η παρακάτω συνάρτηση ενθυλακώνει το βρόγχο της Εικόνας 5.7 και τυπώνει πολλαπλάσια του n:

```
def printMultiplies(n):
 i=1
 while i<=6:
  print(n*i)
  i=i+1
```

**Εικόνα 5.7** *Παράδειγμα ενθυλάκωσης*

Ωστόσο εδώ έχουμε ταυτόχρονα και ένα παράδειγμα γενίκευσης, γιατί η συνάρτηση printMultiplies εκτυπώνει τα πολλαπλάσια οποιουδήποτε ακεραίου (ως και το 6°) και όχι μόνον του 2, όπως γίνεται στην Εικόνα 5.7. Θα μπορούσε το 6 να γενικευτεί και αυτό και να γίνει μια ακόμα παράμετρος αν θέλαμε κάτι τέτοιο. Εάν καλέσουμε την printMultiplies με όρισμα 2, παίρνουμε ακριβώς το ίδιο αποτέλεσμα όπως πριν.

Καλώντας την printMultiplies με όρισμα 3, παίρνουμε στην έξοδο:

```
>>>
3
6
9
12
15
18
```

**Εικόνα 5.8** *Η συνάρτηση printMultiplies με όρισμα 3.*

Με όρισμα 4, η έξοδος γίνεται:

```
4
8
12
16
20
24
>>>
```

**Εικόνα 5.9** *Η συνάρτηση printMultiplies με όρισμα 4.*

Στη συνέχεια ας δούμε πώς μπορούμε να τυπώσουμε δισδιάστατους πίνακες. Ένας δισδιάστατος πίνακας είναι ένας πίνακας που έχει τιμές σε κάθε σημείο διασταύρωσης *ν* σειρών επί *μ* στήλες. Ένα καλό παράδειγμα είναι ένας πίνακας πολλαπλασιασμού (η γνωστή μας *προπαίδεια*), π.χ. των τιμών από 1 έως 6. Σε έναν τέτοιο πίνακα η πρώτη γραμμή είναι 1, 2, 3, 4, 5, 6, η δεύτερη γραμμή ίση με την πρώτη πολλαπλασιασμένη επί 2, η τρίτη γραμμή ίση με την πρώτη πολλαπλασιασμένη επί 3, κλπ. έως και την έκτη γραμμή. Στο παράδειγμα της Εικόνας 5.10 βλέπουμε τις πρώτες δύο γραμμές ενός τέτοιου πίνακα:

```
>>>
1       2       3       4       5       6
2       4       6       8       10      12
```

**Εικόνα 5.10** *Δισδιάστατος πίνακας.*

Στην Εικόνα 5.11 βλέπουμε ένα πρόγραμμα που τυπώνει έναν πίνακα πολλαπλασιασμού καλώντας την printMultiplies κατ' επανάληψη και με διαφορετικά ορίσματα μέσα από ένα νέο βρόγχο. Προσέξτε την ομοιότητα αυτού του βρόγχου με το βρόγχο εντός της printMultiplies. Η μόνη τους διαφορά είναι η αντικατάσταση της εντολής print με μια κλήση συνάρτησης.

Εξετάζοντας εντός της printMultiplies την εντολή print (κάτω από το while) βλέπουμε στο τέλος ένα κόμμα (',') το οποίο σημαίνει «μην αλλάξεις γραμμή στο τέλος του print». Το δεύτερο print, έξω από το while loop, είναι αυτό το οποίο μας πάει σε νέα γραμμή. Κατά την κλήση printMultiplies(1) θα εκτυπωθεί 1, 2, 3, 4, 5, 6 (με tab μεταξύ των αριθμών). Κατά τη δεύτερη κλήση printMultiplies(2) θα εκτυπωθεί 2, 4, 6, 8, 10, 12, κλπ. ώς και την κλήση του printMultiplies(6), όπως φαίνεται στην Εικόνα 5.11.

5-printMultiplies.py

```
1  def printMultiplies(n):
2      i=1
3      while i<=6:
4          print n*i,'\t',
5          i=i+1
6      print
7
8  i=1
9  while i<=6:
10     printMultiplies(i)
11     i=i+1
```

**Εκτέλεση**

```
>>>
1       2       3       4       5       6
2       4       6       8       10      12
3       6       9       12      15      18
4       8       12      16      20      24
5       10      15      20      25      30
6       12      18      24      30      36
```

**Εικόνα 5.11** *Πρόγραμμα που καλεί τη συνάρτηση printMultiplies με διαφορετικά ορίσματα*

Μπορούμε να πάρουμε πάλι τον κώδικα των γραμμών 8-11 της Εικόνας 5.11 και να τον περιτυλίξουμε και αυτόν μέσα σε μια νέα συνάρτηση:

```
def printMultTable():
    i=1
    while i<=6:
        printMultiples(i)
        i = i + 1
```

**Εικόνα 5.12** *Ακόμα ένα παράδειγμα ενθυλάκωσης*

Η παραπάνω διαδικασία χρησιμοποιείται συχνά για την ανάπτυξη κώδικα. Αναπτύσσουμε λοιπόν πρώτα κώδικα έξω από συνάρτηση, π.χ. δακτυλογραφώντας τον στο διερμηνευτή. Όταν σιγουρευτούμε ότι ο κώδικας δουλεύει, τον περιτυλίσσουμε με μια συνάρτηση. Μια τέτοια μέθοδος είναι ιδιαίτερα χρήσιμη όταν δε γνωρίζουμε από την αρχή πώς να χωρίσουμε ένα μεγάλο πρόγραμμα σε συναρτή-

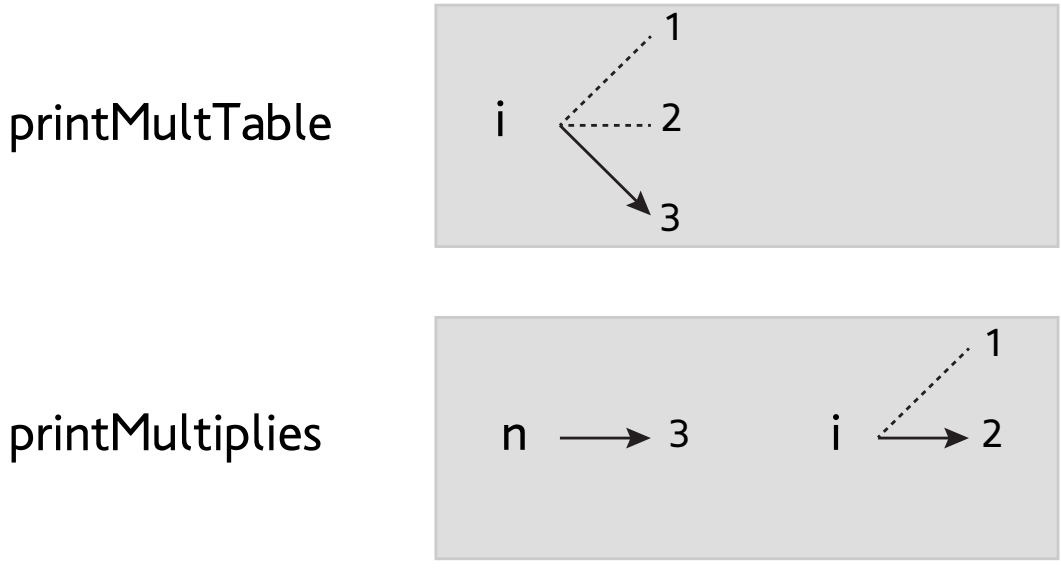
σεις. Μια καλή πρακτική προγραμματισμού είναι, λοιπόν, να διαιρούμε ένα μεγάλο πρόβλημα σε μικρότερα υπο-προβλήματα και να προγραμματίζουμε συναρτήσεις για το κάθε υπο-πρόβλημα, γενικεύοντας, ενθυλακώνοντας κλπ.

## 5.4 Τοπικές μεταβλητές

Ένα ερώτημα το οποίο, ίσως, θα έχετε, αφορά τη χρήση της μεταβλητής i στο πρόγραμμα της Εικόνας 5.12. Εκεί το i χρησιμοποιείται και στην εσωτερική συνάρτηση (printMultiplies), μεταβλητή του βρόγχου while, αλλά και στην εξωτερική συνάρτηση (printMultTable) σαν μεταβλητή του εκεί βρόγχου. Το ερώτημα που προκύπτει είναι: *το i της εσωτερικής συνάρτησης είναι το ίδιο με αυτό της εξωτερικής συνάρτησης;*

Το i της εξωτερικής συνάρτησης, λοιπόν, δεν έχει *καμία απολύτως σχέση* με το i της εσωτερικής. Το ένα i ζει μέσα στην εξωτερική συνάρτηση και μόνο, ενώ το άλλο i ζει μόνο μέσα στην εσωτερική συνάρτηση και πουθενά αλλού. Μεταβλητές που δημιουργούνται μέσα σε μια συνάρτηση είναι **τοπικές και δεν μπορεί να τις προσπελάσει κανείς έξω από τη συνάρτηση που είναι το «σπίτι» της!** Αυτό σημαίνει ότι είναι κανείς ελεύθερος να χρησιμοποιεί μεταβλητές με το ίδιο όνομα, εφόσον δε χρησιμοποιούνται μέσα στην ίδια συνάρτηση! Η εξήγηση έχει να κάνει με τα διαγράμματα στοίβας που είδαμε στο Κεφάλαιο 4.

Στην Εικόνα 5.13, λοιπόν, βλέπουμε τα διαγράμματα στοίβας που αντιστοιχούν στην κλήση της συνάρτησης printMultTable (επάνω πλαίσιο) και printMultiplies (κάτω πλαίσιο). Παρατηρούμε ότι κάθε πλαίσιο προσφέρει το δικό του σπίτι στην αντίστοιχη μεταβλητή $\mathrm{i}$ και σε ποια αναφερόμαστε κάθε φορά συσχετίζεται με την εξής παράμετρο: ποια συνάρτηση εκτελούμε εκείνη τη στιγμή.




**Εικόνα 5.13** *Διαγράμματα στοίβας τοπικών μεταβλητών*

## 5.5 Η εντολή break

Με την εντολή break τερματίζονται άμεσα οι επαναλήψεις μιας εντολής while. Βλέπουμε τη χρήση του break στο παράδειγμα της Εικόνας 5.14, το οποίο περιλαμβάνει ένα βρόγχο και ο οποίος δέχεται, συνεχώς, είσοδο από το πληκτρολόγιο, τυπώνει στην οθόνη την είσοδο που διάβασε, και τερματίζει όταν διαβάσει τη λέξη 'end':

```
while True:
    s = input('Type something:')
    print (s)
    if s == 'end':
    break
```

**Εικόνα 5.14** *Παράδειγμα χρήσης της εντολής break*

## 5.6 Επίλογος

Στο κεφάλαιο αυτό εξετάσαμε την έννοια της επανάληψης με εφαρμογή της στην περίπτωση του βρόγχου while σε σειρά παραδειγμάτων. Είδαμε, επίσης, τις έννοιες της ενθυλάκωσης και της γενίκευσης. Στο Κεφάλαιο 6 θα εξετάσουμε σημαντικές δομές δεδομένων, όπως οι συμβολοσειρές (strings), οι λίστες (lists), οι πλειάδες (tuples) και τα λεξικά (dictionaries), καθώς και τη διαχείρισή τους.

## Βιβλιογραφία/Αναφορές


Lewis, H., & Papadimitriou, C. (1997). *Elements of the Theory of Computation* (Second Edition). Prentice Hall.

Python Software Foundation. (2013). Style guide for Python code. Retrieved from https://www.python.org/dev/peps/pep-0008/

# Κριτήρια αξιολόγησης

## Κριτήριο αξιολόγησης Ι (Βαθμός δυσκολίας: ●)

**Πόσες γραμμές θα έχει ως έξοδο το εξής πρόγραμμα:**

```
x = 0
while (x < 12):
    print 'Το μέτρημα είναι:', x
    x = x + 1
print "Γεια σας!"
```

**Απάντηση/Λύση**

## Κριτήριο αξιολόγησης 2 (Βαθμός δυσκολίας:●●)

**Ποια τιμή πρέπει να εισαγάγει ο χρήστης για να τερματίσει το πρόγραμμα που ακολουθεί:**

```
x = 1
while x == 1 :
    y = raw_input("Εισάγετε ένα αριθμό :")
    print "You entered: ", y
print "Γειά σας!"
```

**Απάντηση/Λύση**

# 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 ώς 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[I] διαλέγει το χαρακτήρα στη θέση I 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[I] 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 "<pyshell#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
```

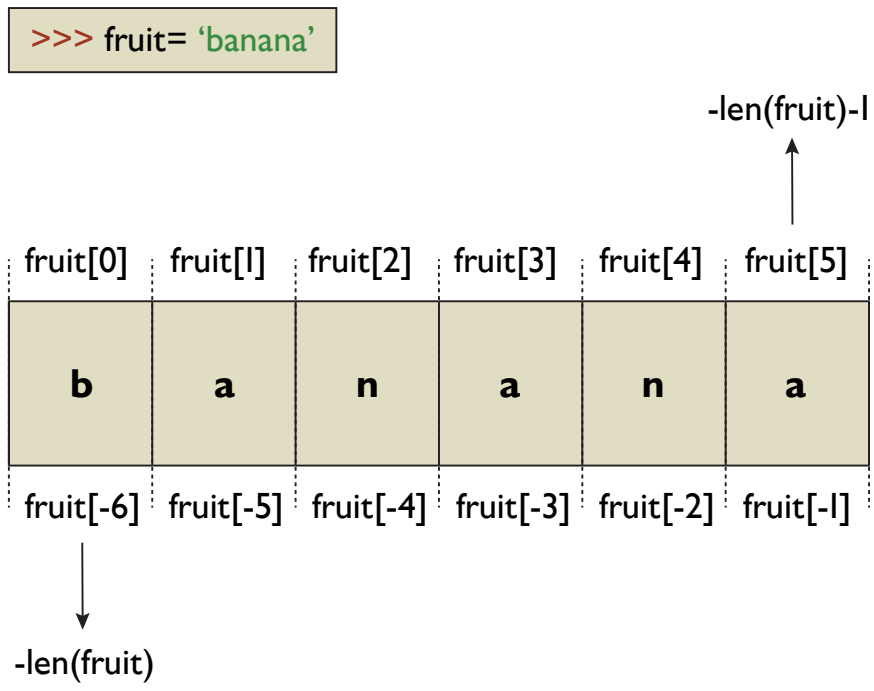
**Εικόνα 6.10** *Σωστός τρόπος επιλογής χαρακτήρα συμβολοσειράς*

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
```

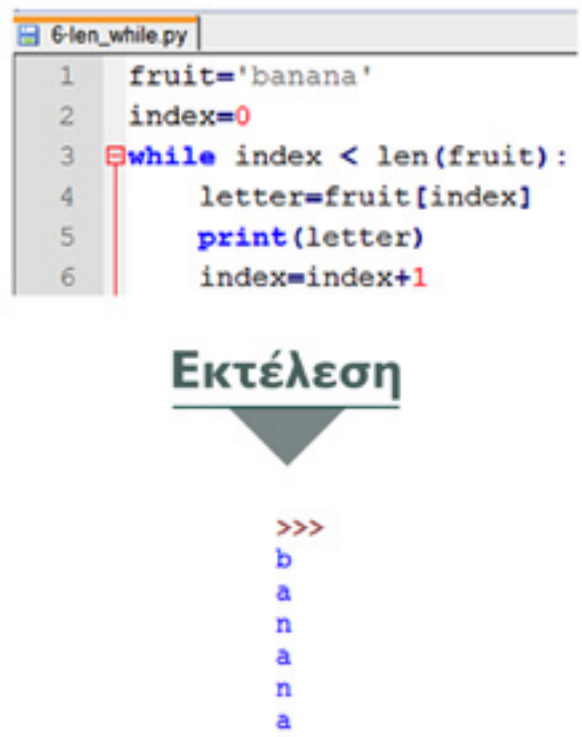
**Εικόνα 6.11** *Χρήση αρνητικών δεικτών για επιλογή χαρακτήρα*




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

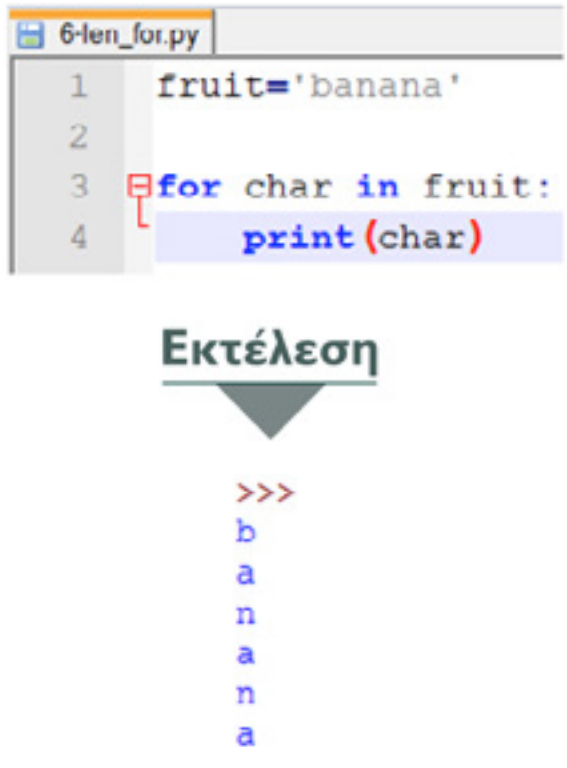
### 6.2.4 Πέρασμα συμβολοσειράς και βρόχος for

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 βρόγχος συνεχίζει, μέχρι να τελειώσουν οι χαρακτήρες.



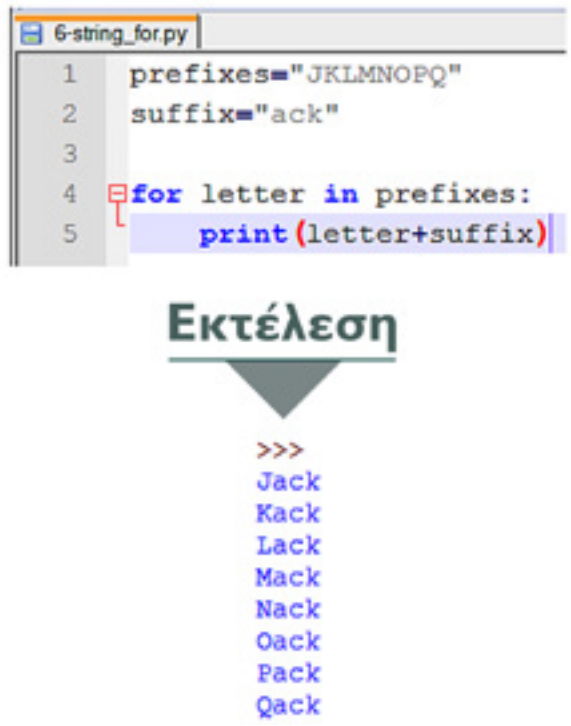

**Εικόνα 6.14** *Πέρασμα μιας συμβολοσειράς με χρήση της εντολής for*

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) ώς και το 4ο δηλαδή το r, αλλά δε φτάνει ώς τον 5ο 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 ώς 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.

6-compare_string.py

```
1  word=input('Give a word:')
2
3  if word < 'banana':
4          print 'Your word,', word,', comes before banana.'
5  elif word > 'banana':
6          print 'Your word,', word,', comes after banana.'
7  else:
8          print('All right, bananas.')
9
```

**Εκτέλεση**

```
Give a word:'apple'
Your word, apple , comes before banana.
```

**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

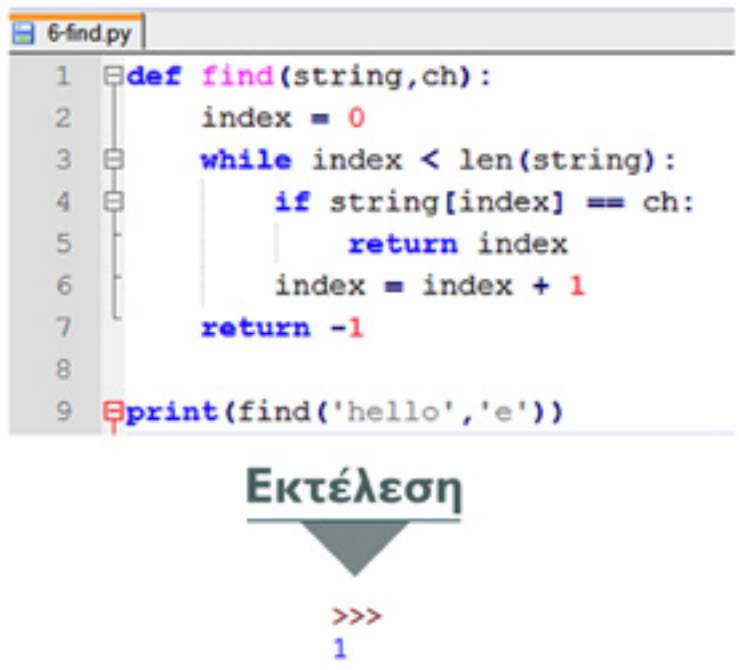
Γράψτε τον ορισμό μιας συνάρτησης με όνομα find, η οποία να έχει δύο παραμέτρους: μια συμβολοσειρά string και ένα χαρακτήρα ch. Η συνάρτηση αυτή να επιστρέφει τον πρώτο δείκτη της συμβολοσειράς string στον οποίο εμφανίζεται χαρακτήρας ίσος με τον ch, αλλιώς να επιστρέφει -1.

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*

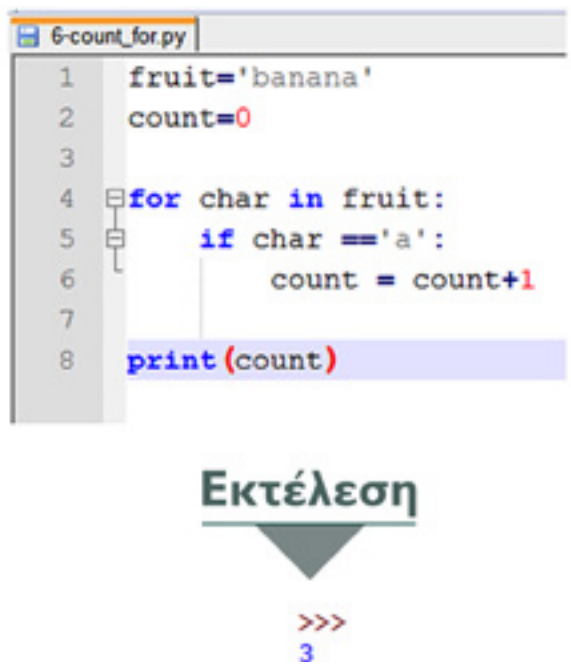
Κατά μια έννοια, η συνάρτηση find είναι η αντίστροφη του τελεστή [ ]. Αντί να παίρνει ως όρισμα ένα δείκτη και να επιστρέφει τον αντίστοιχο χαρακτήρα, παίρνει ένα χαρακτήρα και επιστρέφει τη θέση όπου βρίσκεται ο χαρακτήρας αυτός μέσα στη συμβολοσειρά. Αν ο χαρακτήρας δε βρεθεί, επιστρέφει -1.

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*

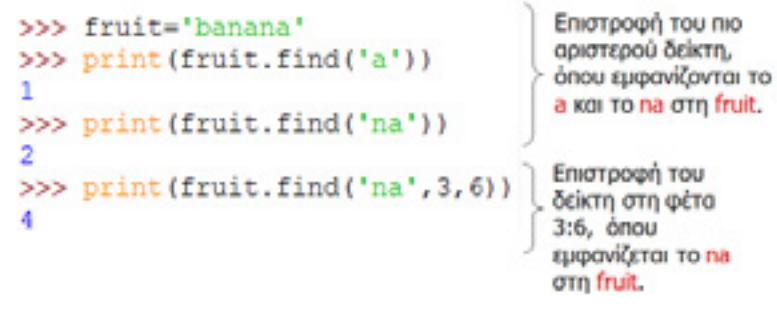
Στη συνέχεια θα αναφερθούμε σε μια πολύ σημαντική έννοια της Python, αυτή της *μεθόδου*. Σε αυτό το Κεφάλαιο θα κατανοήσουμε την έννοια της μεθόδου στα πλαίσια των συμβολοσειρών, στη συνέχεια του συγγράμματος, ωστόσο, θα δούμε ότι η έννοια αυτή είναι γενικότερη και κεντρική στην Python.

### 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.

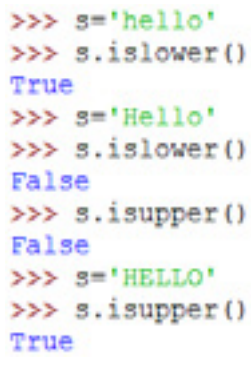
Ένα παράδειγμα μεθόδου συμβολοσειράς είναι η find, (σημειώστε ότι είναι διαφορετική find από τη συνάρτηση με το ίδιο όνομα που ορίσαμε νωρίτερα (πρβ. Εικόνα 6.22), ωστόσο έχει παρόμοια συμπεριφορά. Για να επικαλεστούμε τη μέθοδο find, χρησιμοποιούμε το λεγόμενο dot notation, το οποίο τονίζει ότι η μέθοδος 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.




**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.




**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 {I}, 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 {I} 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 ζητήστε 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 *rang*e, 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 6.41** *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 βρόγχο 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 βρόγχος for έχει πιο συνοπτική και απλή έκφραση, επειδή μπορούμε να καταργήσουμε τη μεταβλητή βρόγχου i (και άρα να αποφύγουμε τη διαχείρισή της, όπως αρχικοποίηση, αύξηση, κλπ.).

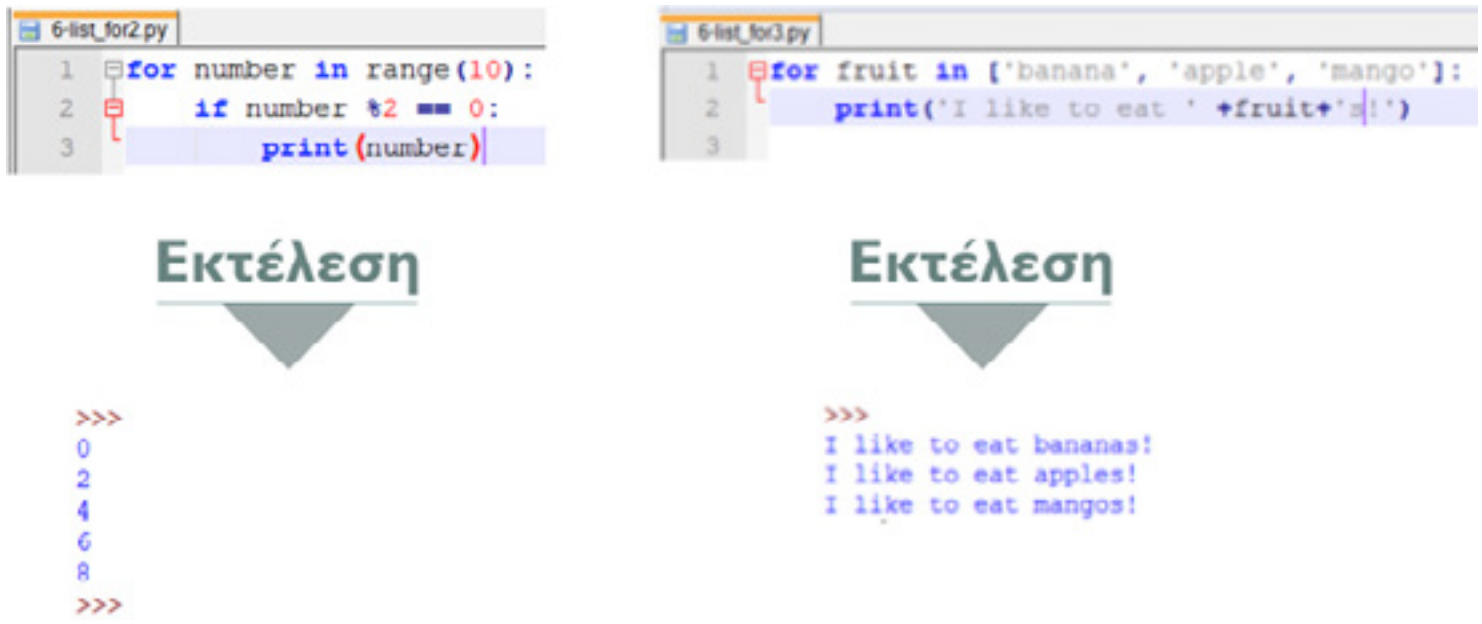
Ο προηγούμενος βρόγχος (Εικόνα 6.42) μπορεί, λοιπόν, να γραφεί ως εξής:


**Εικόνα 6.45** *Ο βρόχος 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, ανοιχτό από δεξιά), άρα παίρνει το 2ο στοιχείο ('b') και το 3ο στοιχείο ('c') της λίστας, αλλά όχι το 4ο. 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 ώς 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 «στριμώχνοντάς» 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*

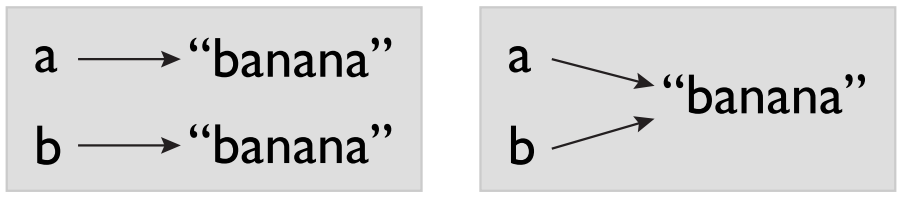
### 6.3.10 Αντικείμενα και τιμές

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

Δηλαδή:

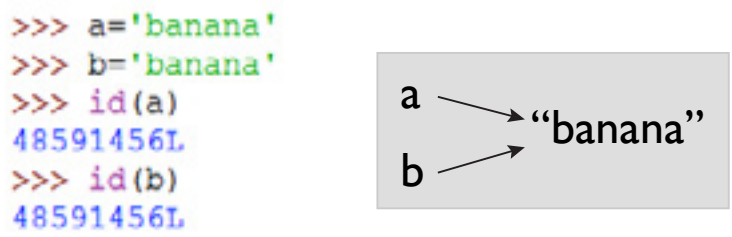
a = "banana"
b = "banana"

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.

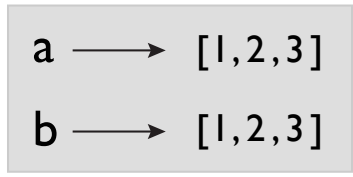



**Εικόνα 6.77** *Δύο μεταβλητές που αναφέρονται στο ίδιο αντικείμενο*

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*




**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.

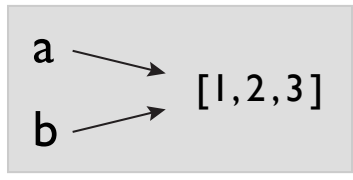
### 6.3.11 Ψευδώνυμα

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** *Ψευδώνυμα*

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 ψευδώνυμα*

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.

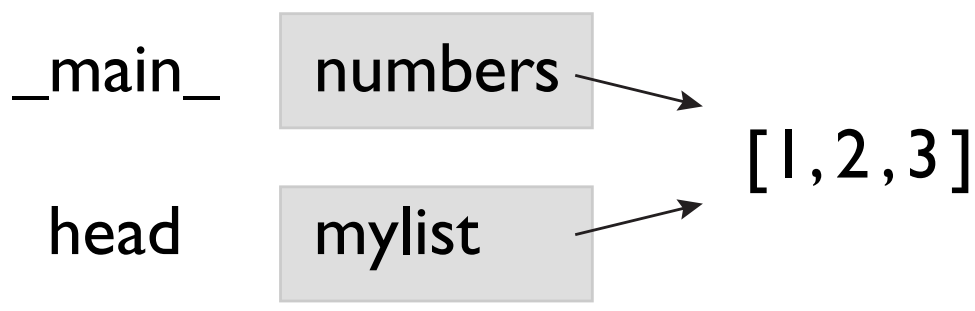
### 6.3.13 Η λίστα ως παράμετρος

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.


**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:*

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

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 εξάγουμε 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 ακεραίους 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'.

```
>>> songwrords = ['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:

```
fruits = ('apple', 'banana', 'mango', 'melon')
i = 0

while i < len(fruits):
    print(fruits[i])
    i = i + 1
```

Εκτέλεση

```
apple
banana
mango
melon
```

```
fruits = ('apple', 'banana', 'mango', 'melon')

for fruit in fruits:
    print(fruit)
```

**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.II1** *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.

6-return_tuples.py

```
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 καταρχήν 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.

6-swap_err_tuples.py

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

Εκτέλεση

```
>>>
a= 2 b= 4
```

**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 ώς 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 1ο, 2ο, 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 επιλεχθεί 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.values()
>>> print(items_view)
['tres', 'dos', '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 μάς 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.

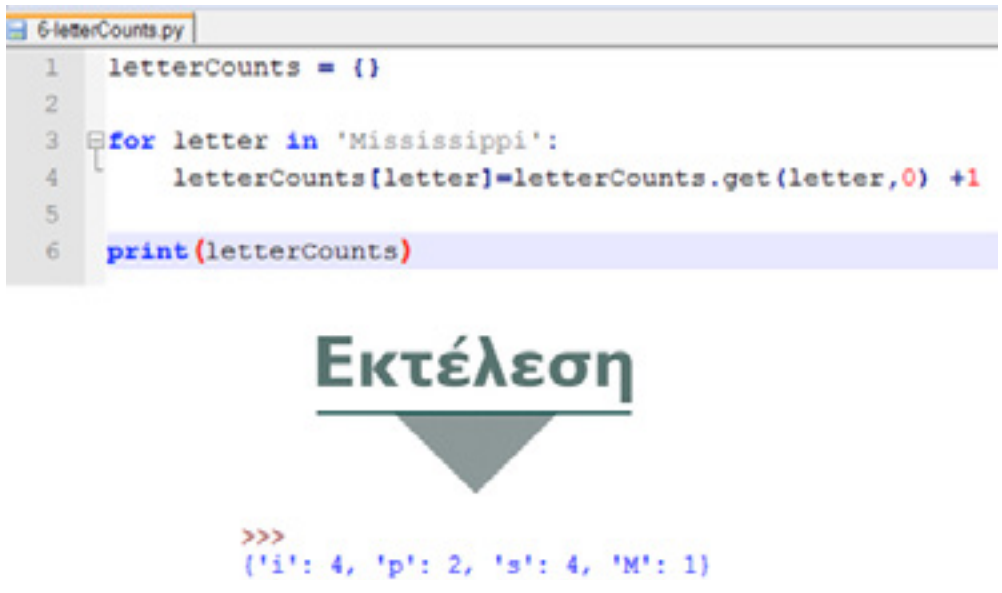
```
>>> matrix.get((1,3),0)
0
```

**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.




**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 1 (Βαθμός δυσκολίας: ●)

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

**Απάντηση/Λύση**

### Κριτήριο αξιολόγησης 2 (Βαθμός δυσκολίας: ●)

**Τι θα εκτυπώσει ο παρακάτω κώδικας:**

```
names = ['Jo','Yo','Lo']
names[0] = 100
print names
```

**Απάντηση/Λύση**

### Κριτήριο αξιολόγησης 3 (Βαθμός δυσκολίας:●●)

**Πότε θα επιλέξουμε να χρησιμοποιήσουμε μία πλειάδα;**

**Απάντηση/Λύση**

### 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**

**Απάντηση/Λύση**

### Κριτήριο αξιολόγησης 5 (Βαθμός δυσκολίας: ●●●)

**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

**Απάντηση/Λύση**

# ΚΕΦΑΛΑΙΟ 7

# Αρχεία, εξαιρέσεις & εκσφαλμάτωση

*Σύνοψη*

*Στο κεφάλαιο αυτό ο αναγνώστης εισάγεται στην έννοια και χρήση των αρχείων, στις εξαιρέσεις και τη διαχείρισή τους, καθώς και στις έννοιες των τεχνικών εκσφαλμάτωσης.*

*Προαπαιτούμενη γνώση*

*Κεφάλαια 1-6 του παρόντος συγγράμματος.*

## 7.1 Εισαγωγή

Κατά τη διάρκεια της εκτέλεσης ενός προγράμματος Python, η κύρια μνήμη χρησιμοποιείται σαν ένα εφήμερο μέσο αποθήκευσης δεδομένων, τα οποία σβήνονται όταν τελειώσει το πρόγραμμα ή όταν ο υπολογιστής κλείσει. Τα δεδομένα αποθηκεύονται μόνιμα όταν τοποθετηθούν σε *αρχεία*, τα οποία, συνήθως, βρίσκονται σε σκληρό δίσκο (ή CD, DVD, flash memory, κλπ.). Ομάδες αρχείων, συνήθως, οργανώνονται σε φακέλους (folders, directories). Κάθε αρχείο ταυτοποιείται με ένα μοναδικό όνομα ή ένα συνδυασμό ονόματος αρχείου και ονόματος φακέλου (ή φακέλων). Διαβάζοντας από αρχεία και γράφοντας σε αυτά, τα προγράμματα μπορούν να επικοινωνούν μεταξύ τους και να δημιουργούν αρχεία τα οποία μπορούν να εκτυπωθούν.

## 7.2 Αρχεία

Τα αρχεία τα μεταχειριζόμαστε λίγο πολύ σαν τετράδια. Ένα τετράδιο πρώτα το ανοίγεις, όταν τελειώσεις το κλείνεις. Όταν είναι ανοικτό μπορούμε να γράψουμε σε αυτό ή να διαβάσουμε από αυτό. Σε κάθε περίπτωση ξέρουμε πού ακριβώς είμαστε μέσα στο τετράδιο. Τις πιο πολλές φορές διαβάζουμε το τετράδιο με τη φυσική του σειρά, αλλά μερικές φορές πάμε και από τη μια σελίδα στην άλλη. Όλα αυτά εφαρμόζονται και στα αρχεία. Για να ανοίξουμε ένα αρχείο, προσδιορίζουμε ένα όνομα και ορίζουμε αν θέλουμε να το διαβάσουμε (read) ή να το γράψουμε (write). Όταν τελειώσουμε την εργασία μας με ένα αρχείο, θα πρέπει να το κλείσουμε (close). Ένα αρχείο κειμένου είναι μια σειρά από χαρακτήρες οι οποίοι είναι αποθηκευμένοι σε ένα μόνιμο μέσο αποθήκευσης (σκληρό δίσκο, κλπ.).

Η βασική συνάρτηση που έχει η Python για να ανοίξουμε ένα αρχείο, είναι η συνάρτηση open η οποία επιστρέφει ένα αρχείο (Εικόνα 7.1) και έχει το αντικείμενο το οποίο φτιάχνεται στη μνήμη. Πρόκειται για μια αναπαράσταση του αρχείου, η οποία έχει τη δυνατότητα να δίνει ορισμένες πληροφορίες για το όνομά του και το περιεχόμενο που έχει και ενδεχομένως κάποιο κομμάτι του αρχείου αντιγράφεται στην κυρίως μνήμη από εκεί όπου ήταν, δηλαδή από το σκληρό δίσκο.

Σαν παράδειγμα αναφέρεται το εξής: Αν τυπώσετε το αντικείμενο *f* (Εικόνα 7.2) θα σας δείξει σαν τιμή πληροφορίες για το αρχείο. Ειδικότερα, όμως, στο παράδειγμα δηλώνεται ότι είναι ανοιχτό, ότι καλείται myfile.txt, είναι ανοιχτό για γραφή και ανάγνωση (αυτό ακριβώς σημαίνει το w). Στην περίπτωση που είναι ανοιχτό για γραφή, μπορείτε και να διαβάζετε. Στην περίπτωση που είναι ανοιχτό μόνο για ανάγνωση, δεν μπορείτε να γράφετε σε αυτό.

Η συνάρτηση open παίρνει βασικά δύο ορίσματα: το πρώτο είναι το όνομα του αρχείου και το άλλο είναι ο τρόπος (mode) 'w', που σημαίνει ότι ανοίγουμε το αρχείο για γράψιμο. Αν δεν υπάρχει αρχείο με όνομα myfile.txt, τότε αυτό θα δημιουργηθεί, ενώ αν υπάρχει ήδη, τότε τα δεδομένα που περιέχονται σε αυτό σβήνονται και αντικαθίστανται από τα νέα που θα γράψουμε. Γι' αυτό πρέπει να είμαστε πολύ προσεκτικοί: Αν το έχουμε δημιουργήσει και θέλουμε να το ξανανοίξουμε, να το ανοίξουμε για ανάγνωση. Πρέπει να τονίσουμε ότι το f είναι μια μεταβλητή

του τύπου «αρχείο» και έχει μεθόδους. Μία μέθοδος είναι η μέθοδος write. Η λειτουργία της μεθόδου write είναι να γράφει στο αρχείο.

## Παράδειγμα:

```
f=open('myfile.txt','w')
```

**Εικόνα 7.1** *Άνοιγμα αρχείου*

Στα Windows, το αρχείο δημιουργείται στο φάκελο που είναι εγκατεστημένη η Python. Αν θέλουμε το αρχείο να δημιουργηθεί σε άλλο φάκελο, πρέπει να γράψουμε και το μονοπάτι που οδηγεί σε αυτόν:

```
f=open('C:\Python_book\myfile.txt','w')

>>> print(f)
<open file 'C:\\Python_book\\myfile.txt', mode 'w' at 0x0000000002727420>
```

**Εικόνα 7.2** *Ορισμός δημιουργίας αρχείου σε άλλο φάκελο πέρα από τον καθορισμένο*

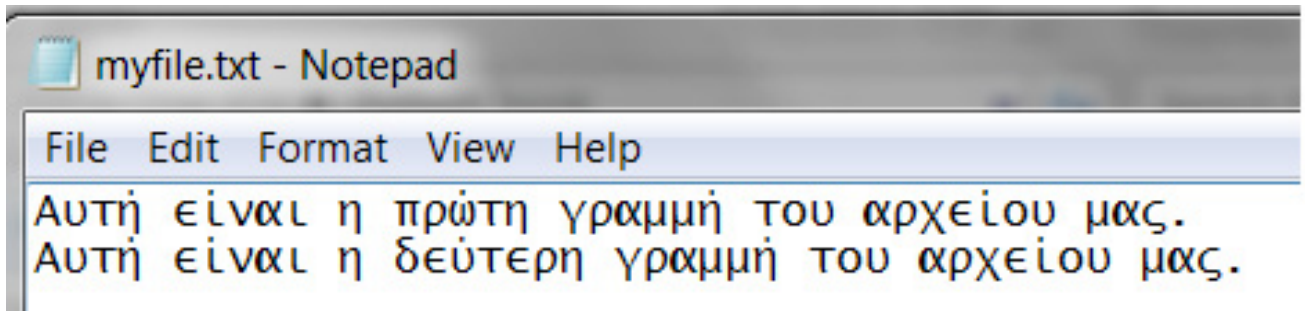
Για να βάλουμε δεδομένα σε ένα αρχείο, επικαλούμαστε τη μέθοδο *write* πάνω στο αντικείμενο του αρχείου. Εμφανίζεται και ο αριθμός των χαρακτήρων που γράφονται στο αρχείο.

```
>>> f=open('C:\Python_book\myfile.txt','w')
>>> grammi1='Αυτή είναι η πρώτη γραμμή του αρχείου μας. \n'
>>> f.write(grammi1)
>>> grammi2='Αυτή είναι η δεύτερη γραμμή του αρχείου μας. \n'
>>> f.write(grammi2)
>>> f.close()
```

**Εικόνα 7.3** *Μέθοδος write και close*

Μπορούμε να κλείσουμε το αρχείο με άλλη μέθοδο και, βεβαίως, μετά μπορούμε να το ανοίξουμε για ανάγνωση.

Το κλείσιμο αρχείου γίνεται με επίκληση της μεθόδου close πάνω στο αντικείμενο του αρχείου.



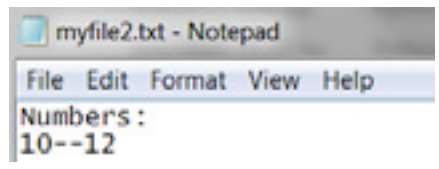

**Εικόνα 7.4** *Το περιεχόμενο του αρχείου μας*

Το όρισμα της μεθόδου *write* πρέπει να είναι μια συμβολοσειρά. Έτσι, αν θέλουμε να βάλουμε άλλες τιμές θα πρέπει πρώτα να τις μετατρέψουμε σε συμβολοσειρές και ο ευκολότερος τρόπος να το πετύχουμε είναι με τη χρήση της συνάρτησης str.

```
>>> f=open('C:\Python_book\myfile2.txt','w')
>>> f.write('Numbers:\n')
>>> numbers=[10,12]
>>> f.write(str(numbers[0]))
>>> f.write('--')
>>> f.write(str(numbers[1]))
>>> f.close()
```

**Εικόνα 7.5** *Η συνάρτηση str*

Προσοχή στις αλλαγές γραμμής (\n).




**Εικόνα 7.6** *Προσοχή στις αλλαγές γραμμής*

Μπορούμε να ανοίξουμε ένα αποθηκευμένο αρχείο για ανάγνωση και να διαβάσουμε το περιεχόμενό του σε μια συμβολοσειρά. Αυτήν τη φορά ο τρόπος (mode) είναι 'r' (ανάγνωση):

```
f = open ('myfile.txt','r')
```

**Εικόνα 7.7 *Mode*** *ανοίγματος αρχείου*

Αν δοκιμάσουμε να ανοίξουμε για ανάγνωση ένα αρχείο που δεν υπάρχει, παίρνουμε μήνυμα λάθους:

```
>>> f=open('C:\Python_book\myfile3.txt','r')

Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    f=open('C:\Python_book\myfile3.txt','r')
IOError: [Errno 2] No such file or directory: 'C:\\Python_book\\myfile3.txt'
```

**Εικόνα 7.8** *Σφάλμα ανοίγματος αρχείου που δεν υπάρχει*

Η μέθοδος read διαβάζει δεδομένα από ένα αρχείο. Χωρίς ορίσματα διαβάζει όλο το περιεχόμενο του αρχείου:

```
>>> f=open('C:\Python_book\myfile2.txt','r')
>>> text=f.read()
>>> print(text)
Numbers:
10--12
>>>
```

**Εικόνα 7.9** *Η μέθοδος read*

Μπορούμε, επίσης, όταν διαβάζουμε, να μη διαβάσουμε ολόκληρο το αρχείο, αλλά να διαβάσουμε ως τον 6ο χαρακτήρα του αρχείου.

Η *read* μπορεί, επίσης, να πάρει όρισμα που καθορίζει πόσοι χαρακτήρες πρέπει να διαβαστούν:

```
>>> f=open('C:\Python_book\myfile2.txt','r')
>>> text=f.read(6)
>>> print(text)
Number
```

**Εικόνα 7.10** *Καθορισμός αριθμού χαρακτήρων που θα διαβαστούν*

Τι γίνεται αν συνεχίσουμε να επικαλούμαστε τη read;

```
>>> text=f.read(100)
>>> print(text)
s:
10--12
```

**Εικόνα 7.11** *Συνέχεια κλήσης της read*

Προσοχή: στο τέλος πρέπει να κλείσουμε το αρχείο:

```
>>> f.close()
```

**Εικόνα 7.12** *Κλείσιμο αρχείου*

Στο παράδειγμα της Εικόνας 7.13 αντιγράφουμε ένα αρχείο, διαβάζοντας και αντιγράφοντας 5 χαρακτήρες τη φορά. Η συνάρτησή μας είναι η copyFile με παραμέτρους oldFile και newFile (το παλιό αρχείο που διαβάζουμε και το καινούριο που γραφούμε αντίστοιχα). Η συνάρτηση while True είναι ένας άπειρος βρόγχος, από όπου πρέπει να έχουμε τρόπο να βγούμε. Με ποιον τρόπο θα γίνει αυτό; Εφόσον διαβάζουμε 5 χαρακτήρες τη φορά, όταν φτάσουμε στο τέλος, η *read* θα μας επιστρέψει κενό, θα κάνουμε break, και θα αποχωρήσουμε από το loop. Διαφορετικά γράφουμε το text στο f2 και πάλι από την αρχή:

7copyFile.py

```
1  def copyFile(oldFile, newFile):
2      f1 = open(oldFile, 'r')
3      f2 = open(newFile, 'w')
4
5      while True:
6          text = f1.read(5)
7      if text == '':
8          break
9      f2.write(text)
10
11     f1.close()
12     f2.close()
13
```

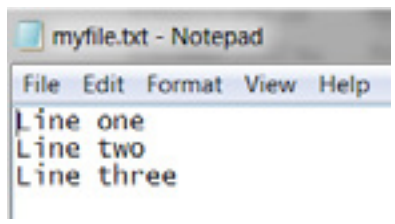
**Εικόνα 7.13** *Κατασκευή συνάρτησης αντιγραφής αρχείου*

## 7.3 Αρχεία κειμένου

Ένα αρχείο κειμένου περιέχει εκτυπώσιμους χαρακτήρες και whitespaces, οργανωμένους σε γραμμές που χωρίζονται από χαρακτήρες newline. Επειδή η Python είναι ειδικά σχεδιασμένη να επεξεργάζεται αρχεία κειμένου, προσφέρει μεθόδους ειδικά για το σκοπό αυτό. Για να το δούμε, ας δημιουργήσουμε ένα αρχείο με τρεις γραμμές κειμένου που χωρίζονται με newlines:

```
>>> f=open('C:\Python_book\myfile.txt','w')
>>> f.write('Line one\nLine two\nLine three\n')
>>> f.close()
```

**Εικόνα 7.14** *Διαχείριση αρχείων κειμένου*




**Εικόνα 7.15** *Έξοδος*

Η μέθοδος *readline* διαβάζει όλους τους χαρακτήρες μέχρι και τον επόμενο χαρακτήρα *newline*:

```
>>> f=open('C:\Python_book\myfile.txt','r')
>>> line=f.readline()
>>> line
'Line one\n'
>>> print(line)
Line one

>>>
```

**Εικόνα 7.16** *Η μέθοδος readline*

Η μέθοδος *readlines* εφαρμόζεται πάνω σε τύπους files και επιστρέφει όλες τις εναπομένουσες τιμές ως λίστα από συμβολοσειρές. Στο παράδειγμα της Εικόνας 7.17 καλούμε *readlines* μια φορά, και αυτό μας επιστρέφει την 2η και 3η γραμμή του αρχείου (επειδή έχουμε, ήδη, διαβάσει την πρώτη πριν λίγο). Τις επιστρέφει,

λοιπόν, σαν ομάδες χαρακτήρων συν τον χαρακτήρα αλλαγής γραμμής. Αυτό είναι πολύ χρήσιμο, διότι τώρα μπορούμε να πάρουμε αυτές τις συμβολοσειρές, να τις χωρίσουμε σε λέξεις για τη λεξικογραφική ανάλυση, κλπ. Αν είμαστε στο τέλος του αρχείου, η *readlines* επιστρέφει άδεια λίστα. Έτσι, λοιπόν, μπορούμε να παίρνουμε κείμενο σε αρχείο και να το χωρίζουμε σε γραμμές.

```
>>> print(f.readlines())
['Line two\n', 'Line three\n']
```

**Εικόνα 7.17** *Η μέθοδος readlines*

Ας θυμηθούμε ότι ο διερμηνέας της Python αγνοεί τα σχόλια τα οποία μπαίνουν μετά το χαρακτήρα της δίεσης (#) μέχρι να τελειώσει η γραμμή. Όταν, λοιπόν, ο διερμηνέας της Python θέλει να αφαιρέσει τα σχόλια από ένα αρχείο, χρησιμοποιεί κώδικα όπως αυτός του παραδείγματος της Εικόνας 7.18, όπου η συνάρτηση filterFile φτιάχνει αντίγραφο του oldFile, παραλείποντας όλες τις γραμμές που αρχίζουν με #.

Προσέξτε τη χρήση της εντολής continue, η οποία τερματίζει την τρέχουσα επανάληψη του βρόγχου και συνεχίζει με την επόμενη επανάληψη. Η ροή εκτέλεσης μεταφέρεται στην κορυφή του βρόγχου όπου ελέγχει ξανά τη συνθήκη και προχωράει αντίστοιχα. Αν το text είναι η άδεια συμβολοσειρά βγαίνουμε από το βρόγχο. Αν ο πρώτος χαρακτήρας του text είναι ο #, η ροή εκτέλεσης πάει στην κορυφή του βρόγχου (παραλείπουμε να γράψουμε τη γραμμή στην έξοδο). Μόνον αν και οι δύο συνθήκες είναι ψευδείς, αντιγράφουμε το text στο νέο αρχείο.

```
7-filterFile.py
1  def filterFile(oldFile, newFile):
2      f1 = open(oldFile, 'r')
3      f2 = open(newFile, 'w')
4
5      while True:
6          text = f1.readline()
7      if text == '':
8          break
9      if text[0]=='#'
10         continue
11     f2.write(text)
12
13     f1.close()
14     f2.close()
15
```

**Εικόνα 7.18** *Η συνάρτηση filterFlile*

## 7.4 Τελεστής διάταξης

Όπως αναφέρθηκε και προηγουμένως, το όρισμα της μεθόδου write πρέπει να είναι συμβολοσειρά, άρα αν πρέπει να βάλουμε άλλου τύπου τιμές σε ένα αρχείο, πρέπει να τις μετατρέψουμε σε συμβολοσειρές. Ο ευκολότερος τρόπος είναι μέσω της συνάρτησης str. Εναλλακτικά τώρα για να έχουμε ακέραιους, πραγματικούς κτλ. μπορούμε να χρησιμοποιήσουμε τον τελεστή διάταξης (format) %. Στους ακεραίους ο τελεστής % είναι ο modulus και επιστρέφει υπόλοιπο διαίρεσης. Όταν, όμως, ο πρώτος τελεστέος είναι συμβολοσειρά, τότε ο τελεστής % είναι τελεστής διάταξης. Ο δεύτερος τελεστέος είναι πλειάδα εκφράσεων. Γίνεται, λοιπόν, μια αντιστοίχιση μεταξύ κομματιών ομάδας χαρακτήρων και στοιχείων της πλειάδας.

Στο παράδειγμα της Εικόνας 7.19 δημιουργούμε νέες συμβολοσειρές που περιέχουν τιμές μεταβλητών. Αρχικά δίνουμε στην ακέραια μεταβλητή cars την τιμή 52. Στη συνέχεια χρησιμοποιούμε τον τελεστή διάταξης, αριστερά του οποίου είναι μια ομάδα χαρακτήρων, ενώ δεξιά του είναι μια πλειάδα που έχει σαν μόνο στοιχείο τη μεταβλητή cars. Η σημασιολογία του τελεστή %d είναι: «Τύπωσε τη μεταβλητή σαν ακέραιο» (d σημαίνει decimal ή ακέραιος). Το αποτέλεσμα είναι μια

ομάδα χαρακτήρων που συμπεριλαμβάνει στην αρχική συμβολοσειρά τα ψηφία του ακεραίου cars.

Θα μπορούσαμε να βάλουμε δύο ή περισσότερους αριθμούς, ή πλειάδες. Στη δεύτερη εντολή, λοιπόν, έχουμε πάλι τον τελεστή διάταξης με μια ομάδα χαρακτήρων αριστερά και μια πλειάδα δεξιά με 3 στοιχεία: το 34, το 6.1 και το 'dollars'. Αυτά τα τρία στοιχεία θα πρέπει να αντιστοιχηθούν σε 3 χαρακτήρες με το σύμβολο % στην αριστερή μεριά του τελεστή διάταξης. Το πρώτο % (**%d** το οποίο σημαίνει ακέραιος) θα αντιστοιχηθεί με το πρώτο στοιχείο της πλειάδας, το δεύτερο % (**%f** το οποίο σημαίνει floating-point ή πραγματικός) με το δεύτερο στοιχείο της πλειάδας, και το τρίτο % (**%s** δηλαδή string) με το τρίτο στοιχείο της πλειάδας. Στην εκτύπωση του πραγματικού 6.1 με το %f, ο διερμηνέας βάζει 6 δεκαδικά ψηφία (6.100000). Σύμφωνα με τον κανόνα της Python που αποτελεί σύμβαση, αν δεν ορίσουμε πόσα δεκαδικά θέλουμε, τότε βάζει 6 δεκαδικά (συμπληρώνοντας 0 αν χρειαστεί). Μπορεί, ωστόσο, κανείς να ορίσει τον αριθμό των δεκαδικών βάζοντας δύο νούμερα μπροστά από το f. Το δεύτερο νούμερο ορίζει το πλήθος των δεκαδικών και κάνει στρογγυλοποίηση αν έχει περισσότερα δεκαδικά. Το πρώτο νούμερο είναι το συνολικό πλήθος των ψηφίων ή θέσεων στην εκτύπωση ή θέσεων (χαρακτήρων) στο αρχείο που θα πιάσει αυτός ο αριθμός.

Στο παράδειγμα το "%12.2f", λέει ότι αυτός είναι ένας πραγματικός αριθμός με μέγιστο εύρος 12 θέσεις (χαρακτήρες), από τις οποίες οι δύο είναι δεκαδικά ψηφία. Η υποδιαστολή λογίζεται και αυτή σαν μια θέση, άρα μένουν 9 ψηφία (χαρακτήρες) για το ακέραιο μέρος. Αν ο αριθμός ο ίδιος δεν αρκεί να καταλάβει 12 ψηφία, τότε αφήνονται κενές θέσεις. Αν αντίθετα το ακέραιο μέρος υπερβαίνει το 12, ο διερμηνέας βγάζει λάθος (υπέρβαση ή overflow). Αν τα δεκαδικά δε χωρούν, τότε κάνει στρογγυλοποίηση. Με το "%12f" δεν ορίζουμε πόσα δεκαδικά πρέπει να βάλει, άρα εφαρμόζει το δικό του κανόνα: 6 δεκαδικά ψηφία. Με τα 6 δεκαδικά ψηφία συν την υποδιαστολή και το δεκαδικό, έχουμε 4 θέσεις κενές. Με το "%12.2f" ισχύουν τα παραπάνω, αλλά καθορίζουμε ότι κρατάμε μόνο δύο δεκαδικά.

```
>>> cars = 52
>>> 'In July we sold %d cars.'% cars
'In July we sold 52 cars.'
>>> 'In %d days we made %f million %s.' %(34, 6.1, 'dollars')
'In 34 days we made 6.100000 million dollars.'
>>> '%6d'% 62
'    62'
>>> '%12f'% 6.1
'    6.100000'
>>> '%-6d'% 62
'62    '
>>> '%12.2f'% 6.1
'        6.10'
```

**Εικόνα 7.19** *Ο τελεστής διάταξης %*

Ένα πιθανό λάθος το οποίο χρειάζεται προσοχή είναι να έχουμε λιγότερα στοιχεία στην πλειάδα στα δεξιά του % από όσα χρειάζεται. Επίσης, ενδέχεται από παραδρομή να ζητήσουμε να τυπωθεί π.χ. μια ομάδα χαρακτήρων σαν ακέραιος, το οποίο δεν επιτρέπεται. Ακολουθεί ένα παράδειγμα.

```
>>> '%d' % 'python_book'

Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    '%d' % 'python_book'
TypeError: %d format: a number is required, not str
```

## 7.5 Άλμευση (pickling)

Πρέπει να θυμόμαστε ότι για να βάλουμε τιμές μέσα σε ένα αρχείο πρέπει να τις κάνουμε ομάδες χαρακτήρων. Το θέμα είναι ότι, όταν διαβάζουμε πίσω τη συμβολοσειρά, δεν ξέρουμε πού τελειώνει η μία τιμή και πού αρχίζει η επόμενη, και άρα η αρχική πληροφορία για τον τύπο της τιμής έχει χαθεί. Εκφράζοντας το πρόβλημα με διαφορετικό τρόπο, θέλουμε να μπορούμε να αποθηκεύουμε στα αρχεία μας, όχι μόνον ομάδες χαρακτήρων και χαρακτήρες, αλλά να μπορούμε να αποθηκεύουμε και την πιο πολύπλοκη δομή των τύπων δεδομένων καθώς και να θυμόμαστε τη δομή αυτή. Πώς γίνεται αυτό, θα εξηγηθεί με το παράδειγμα που φαίνεται στην Εικόνα 7.20.

Θεωρήστε, λοιπόν, ότι έχουμε τον πραγματικό αριθμό 5.2 και τη λίστα 1,2,3. Και τα δύο θα αποθηκευτούν. Πιο συγκεκριμένα το πρώτο θα αποθηκευτεί σαν η ομάδα χαρακτήρων 5,2 και το δεύτερο θα αποθηκευτεί, επίσης, σαν η ομάδα χαρακτήρων

με αριστερή τετραγωνική παρένθεση 1,2,3 δεξιά τετραγωνική παρένθεση ([1,2,3]). Ωστόσο θα χαθεί, τελείως, η πληροφορία ότι αυτό ήταν λίστα, ότι αυτό ήταν πραγματικός αριθμός. Διότι, θα είναι μια ομάδα χαρακτήρων και όταν διαβάσουμε το αρχείο αυτό μετά, παίρνουμε μια ομάδα χαρακτήρων. Η Python, όμως, δεν μπορεί να καταλάβει ότι αυτός ήταν πραγματικός αριθμός κάποτε ή ότι αυτό ήταν λίστα κάποτε. Το πρόβλημα είναι ότι χάθηκε η πληροφορία ότι αυτός ήταν πραγματικός αριθμός κάποτε ή αυτός ήταν λίστα κάποτε. Τώρα όλα αυτά αποτελούν μια ομάδα χαρακτήρων.

```
>>> f = open('myfile.txt','w')
>>> f.write(str(5.2))
>>> f.write(str([1,2,3]))
>>> f.close()
>>> f=open('myfile.txt','r')
>>> f.readline()
'5.2[1, 2, 3]'
>>> f.close()
```

**Εικόνα 7.20** *Τοποθέτηση τιμών σε ένα αρχείο*

Πρέπει, λοιπόν, να δώσουμε πιο πολύ δομή στην έξοδό μας στο αρχείο. Η λύση ονομάζεται «άλμευση» (βάζω κάτι στην άλμη ή άρμη ή αρμύρα), που «συντηρεί» τις δομές δεδομένων. Υπάρχει ένα πακέτο που ονομάζεται pickle, το οποίο κάνουμε import, και μετά ανοίγουμε το αρχείο ως συνήθως. Για αποθήκευση μιας δομής δεδομένων, χρησιμοποιούμε τη συνάρτηση dump. Ορίζουμε: «Αυτό θα το αποθηκεύσεις σαν πραγματικό αριθμό» γιατί βάζουμε το float (f) σαν δεύτερη παράμετρο, ομοίως για τη λίστα, και μετά κλείνουμε το αρχείο με το συνηθισμένο τρόπο. Σημειώστε ότι στο open πρέπει να χρησιμοποιήσουμε το mode 'wb' (write binary).

```
>>> import pickle
>>> f = open('myfile.txt','wb')
>>> pickle.dump(5.2,f)
>>> pickle.dump([1,2,3],f)
>>> f.close()
```

**Εικόνα 7.21** *Το module pickle και η συνάρτηση dump*

Τώρα μπορούμε να ανοίξουμε το αρχείο για διάβασμα και να φορτώσουμε τις δομές δεδομένων που αδειάσαμε (dumped) στο αρχείο (με τη συνάρτηση load).

Σημειώστε ότι στο open πρέπει να χρησιμοποιήσουμε το mode 'rb' (read binary). Έτσι, μπορούμε να διαβάσουμε και τον πραγματικό αριθμό και τη λίστα διατηρώντας τη δομή τους. Συνοψίζοντας, οι σημαντικές μέθοδοι εδώ είναι dump για να διατηρήσουμε τη δομή και load για να διαβάσουμε ένα αρχείο σαν δομημένο αντικείμενο. Κάθε load διαβάζει το επόμενο δομημένο αντικείμενο, όσο μεγάλο κι αν είναι αυτό.

```
>>> f = open('myfile.txt','rb')
>>> x = pickle.load(f)
>>> type(x)
<type 'float'>
>>> print(x)
5.2
>>> l=pickle.load(f)
>>> type(l)
<type 'list'>
>>> print(l)
[1, 2, 3]
```

**Εικόνα 7.22** *Η συνάρτηση load*

## 7.6 Εξαιρέσεις

Όταν συμβαίνει λάθος κατά τη διάρκεια της εκτέλεσης ενός προγράμματος (runtime error), δημιουργείται μια **εξαίρεση (exception)**· το πρόγραμμα σταματά και η Python τυπώνει μήνυμα λάθους. Παραδείγματα:

```
>>> print 5/0

Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    print 5/0
ZeroDivisionError: integer division or modulo by zero
>>> a=[]
>>> print(a[2])

Traceback (most recent call last):
  File "<pyshell#28>", line 1, in <module>
    print(a[2])
IndexError: list index out of range
>>> b={}
>>> print(b['what'])

Traceback (most recent call last):
  File "<pyshell#30>", line 1, in <module>
    print(b['what'])
KeyError: 'what'
>>> f = open('newfile.txt','r')

Traceback (most recent call last):
  File "<pyshell#31>", line 1, in <module>
    f = open('newfile.txt','r')
IOError: [Errno 2] No such file or directory: 'newfile.txt'
```

**Εικόνα 7.23** *Μήνυμα λάθους της Python*

Σε κάθε περίπτωση, το μήνυμα λάθους έχει δύο μέρη: τον τύπο του λάθους πριν από την άνω και κάτω τελεία, και λεπτομέρειες για το λάθος μετά την άνω και κάτω τελεία. Μερικές φορές θέλουμε να εκτελέσουμε μια πράξη που μπορεί να προκαλέσει εξαίρεση, αλλά δε θέλουμε το πρόγραμμά μας να σταματήσει. Το χειριζόμαστε αυτό με τις εντολές try και except.

Για παράδειγμα, ζητάμε από το χρήστη «όνομα αρχείου» και μετά προσπαθούμε να το ανοίξουμε. Αν το αρχείο δεν υπάρχει, δε θέλουμε το πρόγραμμα να αποτύχει και να τερματίσει. Θέλουμε αντ' αυτού να χειριστούμε την εξαίρεση:

C:\Python_book\7-filename.py - Notepad++

File Edit Search View Encoding Language Settings Macro Run TextFX F

7-filename.py

```
1  filename = input ('Enter a file name:')
2  try:
3      f = open(filename, 'r')
4  except IOError:
5      print('There is no file named', filename)
```

**Εικόνα 7.24** *Οι συντελεστές try και except*

Η εντολή try εκτελεί τις εντολές στο πρώτο μπλοκ. Αν δεν υπάρξουν εξαιρέσεις αγνοεί την εντολή except. Αν συμβεί εξαίρεση τύπου IOError, τότε εκτελεί τις εντολές στον κλάδο except και μετά συνεχίζει.

Μπορούμε να ενθυλακώσουμε τον παραπάνω κώδικα σε συνάρτηση:

File Edit Search View Encoding Language Settings M

7-exists.py

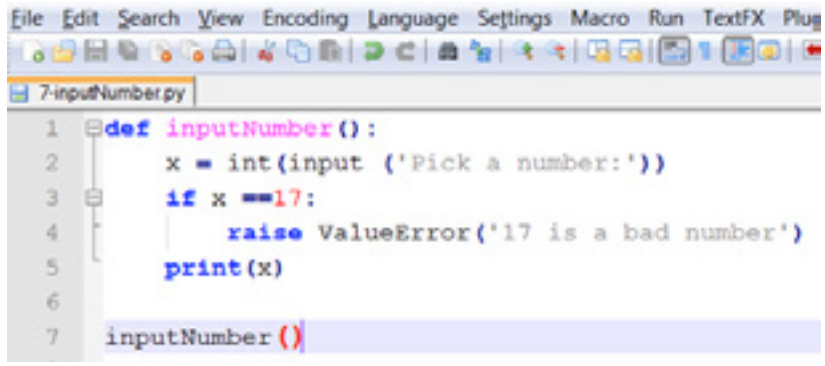
```
1  def exists(filename):
2      try:
3          f = open(filename, 'r')
4          f.close()
5          return True
6      except IOError:
7          return False
```

**Εικόνα 7.25** *Παράδειγμα: Συνάρτηση exists*

Εάν το πρόγραμμά σας ανακαλύψει ένα λάθος, μπορείτε να το κάνετε να «σηκώσει» (από την αγγλική έκφραση, το raise) μια εξαίρεση.

## Παράδειγμα:

Παίρνουμε είσοδο από το χρήστη και ελέγχουμε αν η τιμή είναι 17. Υποθέτοντας ότι για κάποιο λόγο το 17 δεν είναι σωστή είσοδος, σηκώνουμε εξαίρεση με την εντολή raise:




**Εικόνα 7.26** *Η εντολή raise*

Εάν καλέσουμε τη συνάρτηση inputNumber και δώσουμε ως είσοδο τον αριθμό 17, τότε θα εμφανιστεί το παρακάτω:

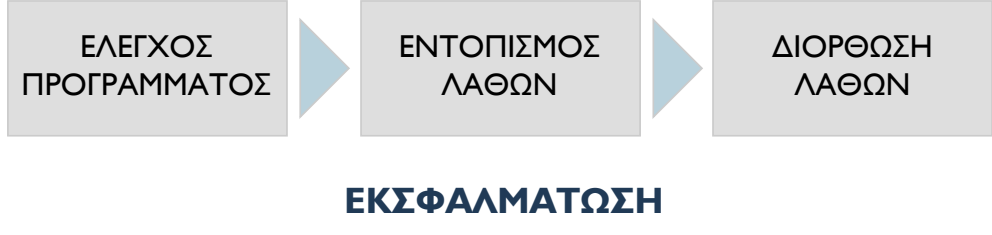
```
>>>
Pick a number:17

Traceback (most recent call last):
  File "C:\Python_book\7-inputNumber.py", line 7, in <module>
    inputNumber()
  File "C:\Python_book\7-inputNumber.py", line 4, in inputNumber
    raise ValueError('17 is a bad number')
ValueError: 17 is a bad number
```

**Εικόνα 7.27** *Μήνυμα σφάλματος*

Ο τύπος ValueError είναι ένας από τους τύπους εξαιρέσεων που δίνει η Python. Άλλοι τύποι είναι οι: TypeError, KeyError, IOError, NameError, MemoryError, OSError, SyntaxError NotImplementedError, κ.λπ.

## 7.7 Εκσφαλμάτωση (debugging)

Η διαδικασία του προγραμματισμού είναι μια πολύπλοκη διαδικασία που συχνά οδηγεί σε λάθη, άλλωστε αυτό είναι αναμενόμενο καθώς πραγματοποιείται από ανθρώπους. Τα προγραμματιστικά λάθη λέγονται bugs και η διαδικασία εντοπισμού και διόρθωσής τους ονομάζεται debugging ή εκσφαλμάτωση (Zeller, 2005).




**Εικόνα 7.28** *Εκσφαλμάτωση*

Πιθανά προγραμματιστικά λάθη μπορεί να είναι τα παρακάτω:

- συντακτικά λάθη (Syntax errors),
- λάθη κατά την εκτέλεση (Runtime errors),
- σημασιολογικά λάθη (Semantic errors).

### Συντακτικά λάθη

Παράγονται κατά τη μετάφραση του πηγαίου κώδικα και, συνήθως, δείχνουν ότι υπάρχει κάποιο λάθος στη σύνταξη του προγράμματος. Οφείλονται σε παραβίαση των κανόνων της γραμματικής και του συντακτικού της γλώσσας προγραμματισμού. Παραδείγματα λαθών:

```
SyntaxError: invalid syntax
>>> def printMessage()
        print('Hello World')

SyntaxError: invalid syntax
>>> deef printMessage()
        print('Hello World')

SyntaxError: invalid syntax
```

**Εικόνα 7.29** *Συντακτικά λάθη*

Τα συντακτικά λάθη είναι εύκολα στη διόρθωση, αρκεί να εντοπιστούν. Συνήθως, όμως, τα μηνύματα λάθους (error messages) δεν είναι κατατοπιστικά. Τα πιο συνηθισμένα μηνύματα είναι τα εξής:

- SyntaxError: invalid syntax
- SyntaxError: invalid token

Το μήνυμα λάθους μας πληροφορεί και για το πού βρίσκεται το λάθος. Αν ένα πρόγραμμα γράφεται αυξητικά (incrementally), τότε το λάθος θα βρίσκεται στις χρονικά πιο πρόσφατες γραμμές που προστέθηκαν.

Μερικοί τρόποι αποφυγής κοινότοπων λαθών:

- Μη χρησιμοποιείτε τις δεσμευμένες λέξεις της Python (keyword) ως ονόματα μεταβλητών.
  Λάθος: def = 5
- Επιβεβαιώστε ότι υπάρχει ένα σύμβολο colon (:) στο τέλος της επικεφαλίδας κάθε σύνθετης εντολής (for, while, if, def).
  Λάθος: while x > 0
- Ελέγξτε τη συνέπεια των εσοχών (indentation).
  Λάθος:
  def printMessage():
  print(‘Hello world’)
- Σιγουρευτείτε ότι τα τυχόν strings στον κώδικα έχουν quotation marks που ταιριάζουν.

- Λάθος: print(“Hello world’)

  - single quote: ‘
  - double quote: ”

- Να κλείνετε σωστά τα brackets (, {, [
- Μην μπερδεύετε την εκχώρηση = με την ισότητα ==

## Λάθη κατά την εκτέλεση

Πολλές φορές ένα πρόγραμμα, απαλλαγμένο από συντακτικά λάθη, μπορεί να εμφανίσει λάθη κατά την εκτέλεσή του. Συνήθως, τα λάθη αυτά οφείλονται σε απαιτήσεις του προγράμματος που ο υπολογιστής δεν μπορεί να ικανοποιήσει και ο προγραμματιστής δεν είχε προβλέψει στη διαδικασία ανάλυσης του προβλήματος και κωδικοποίησης του προγράμματος. Θα μπορούσαν να αναφερθούν ως παραδείγματα τα εξής: διαίρεση με το μηδέν, χρήση αρχείου για ανάγνωση όταν δεν υπάρχει, ατέρμονη αναδρομή, κλπ. Τα περισσότερα runtime error messages περιλαμβάνουν πληροφορία για το πού συνέβη το λάθος και ποιες συναρτήσεις ήταν σε εκτέλεση.

Το πρόγραμμα μπορεί να περιέχει συναρτήσεις και κλάσεις, αλλά σταματάει και φαίνεται να μην κάνει τίποτα. Η αιτία μπορεί να είναι μια από τις παρακάτω:

- ατέρμονας βρόγχος (infinite loop) ή
- ατέρμονη αναδρομή (infinite recursion).

## Ατέρμονας βρόγχος:

- Αν νομίζετε ότι ένας συγκεκριμένος βρόγχος είναι ατέρμονας, τότε προσθέστε στο τέλος της επανάληψης εντολές εκτύπωσης που τυπώνουν τις τιμές των μεταβλητών στη συνθήκη και την τιμή της συνθήκης.

## Για παράδειγμα:

## Ατέρμονη αναδρομή:

- Τις περισσότερες φορές το πρόγραμμα τρέχει για λίγο και έπειτα παράγει ένα λάθος:
- Maximum recursion depth exceeded.
- Εάν κρίνετε ότι μια συνάρτηση ή μέθοδος προκαλεί ατέρμονη αναδρομή, τότε ελέγχετε αν υπάρχει μια *base case*, δηλαδή, κάποια συνθήκη που θα αναγκάσει τη συνάρτηση να επιστρέψει χωρίς να παράγει επαναλαμβανόμενη κλήση.
- Εάν δεν υπάρχει κάποια *base case*, πρέπει να ξανασκεφτείτε τον αλγόριθμο και να προσδιορίσετε κάποια.
- Εάν υπάρχει κάποια *base case* και ο αλγόριθμος έχει την ίδια συμπεριφορά, προσθέστε στην αρχή της συνάρτησης εντολές εκτύπωσης που τυπώνουν τις τιμές των παραμέτρων. Εάν οι παράμετροι δεν κινούνται προς την*base case*, θα πρέπει να βρείτε την αιτία γι' αυτό.

## Exception:

- Εάν κάτι πηγαίνει στραβά κατά την εκτέλεση, η Python τυπώνει ένα μήνυμα που περιλαμβάνει το όνομα του exception, τη γραμμή του προγράμματος που εμφανίστηκε καθώς και ένα traceback.
- Το πρώτο βήμα είναι να εξεταστεί η θέση του προγράμματος όπου εμφανίστηκε το λάθος.

Κάποια από τα πιο κοινά λάθη κατά την εκτέλεση (runtime errors) είναι τα παρακάτω:

- NameError
- TypeError
- KeyError
- AttributeError
- IndexError

Προβλήματα με πολλές εντολές εκτύπωσης στο debugging:

- Μπορεί να καταλήξουν κρυμμένες στην έξοδο.

Λύσεις:

- Απλοποίηση της εξόδου
- Απλοποίηση του προγράμματος
- Καθαρισμός του προγράμματος:
  - > Αφαιρέστε το «νεκρό» κώδικα
  - > Αναδιοργανώστε το πρόγραμμα.

## Σημασιολογικά Λάθη

Σημασιολογικά λάθη μπορούν να συμβούν όταν το πρόγραμμα τρέχει, αλλά δεν εκτελεί τις σωστές λειτουργίες. Είναι τα πιο δύσκολα λάθη για να διορθωθούν, καθώς ο compiler και το runtime σύστημα δεν μπορούν να τα διαγνώσουν και άρα δεν παρέχουν κάποια πληροφορία για αυτά.

Λύσεις για τα λάθη αυτά είναι οι παρακάτω:

- Να συνδέσετε το κείμενο του προγράμματος με τη συμπεριφορά που βλέπετε.
- Να βρείτε μια υπόθεση για αυτό που πραγματικά κάνει το πρόγραμμα.
- Να μπορείτε να επιβραδύνετε το πρόγραμμά σας.
- Να χρησιμοποιείτε τον debugger και να “περπατήσετε” το πρόγραμμα ελέγχοντας καλύτερα τη λογική του.

Ακόμη ένα πρόβλημα είναι οι μεγάλες δύσκολες εκφράσεις που δε δουλεύουν. Μια λύση είναι να σπάσουμε τη σύνθετη έκφραση σε μία σειρά αναθέσεων σε προσωρινές μεταβλητές.

Για παράδειγμα:

Αντί

```
self.hands[i].addCard(self.hands[self.findNeighbor(i)].popCard())
```

μπορείτε να γράψετε:

```
neighbor = self.findNeighbor(i)
pickedCard = self.hands[neighbor].popCard()
self.hands[i].addCard(pickedCard)
```

Αυτό είναι ευκολότερο να διορθωθεί καθώς μπορείτε να ελέγξετε τους τύπους των ενδιάμεσων μεταβλητών και τις τιμές τους.

Ένα άλλο πρόβλημα που μπορεί να εμφανιστεί στις μεγάλες εκφράσεις είναι η μη αναμενόμενη σειρά αποτίμησης.

Για παράδειγμα, η έκφραση μπορεί να γραφτεί στην Python ως:

```
y = x / 2 * math.pi
```

Όμως, επειδή ο πολλαπλασιασμός και η διαίρεση έχουν την ίδια προτεραιότητα, οι πράξεις θα εκτελεστούν από τα αριστερά προς τα δεξιά. Η λύση είναι η χρήση παρενθέσεων:

```
y = x / (2 * math.pi)
```

Επίσης, μπορεί να έχουμε μια συνάρτηση που δεν επιστρέφει αυτό που θέλουμε:

- Εάν έχετε μία σύνθετη έκφραση επιστροφής μπορείτε να τυπώσετε την επιστρεφόμενη τιμή πριν την επιστρέψετε. Επίσης, μπορείτε να χρησιμοποιήσετε μια προσωρινή μεταβλητή.

Για παράδειγμα:

Αντί

```
return self.hands[i].removeMatches()
```

μπορείτε να γράψετε:

```
count = self.hands[i].removeMatches()
return count
```

## 7.8 Επίλογος

Σε αυτό το κεφάλαιο μελετήσαμε τη χρήση των αρχείων, τις εξαιρέσεις κατά τη διάρκεια εκτέλεσης ενός προγράμματος και τη διαχείρισή τους, και την έννοια της εκσφαλμάτωσης και των τεχνικών που χρησιμοποιούνται για την εύρεση σφαλμάτων αλλά και την αποφυγή τους. Στο Κεφάλαιο 8 θα μελετήσουμε τις βασικές έννοιες του αντικειμενοστραφούς προγραμματισμού και την εφαρμογή τους στην Python.

## Βιβλιογραφία/Αναφορές

Zeller, A. (2005). Why Programs Fail: *A Guide to Systematic Debugging*. Morgan Kaufmann.

### Κριτήριο αξιολόγησης Ι (Βαθμός δυσκολίας:●●)

**Γράψτε ένα πρόγραμμα που παίρνει το όνομα από το χρήστη και το γράφει στο αρχείο myname.txt**

### Κριτήριο αξιολόγησης 2 (Βαθμός δυσκολίας: ●●)

**Γράψτε ένα πρόγραμμα όπου ζητάμε από το χρήστη «όνομα αρχείου» και μετά προσπαθούμε να το ανοίξουμε. Αν το αρχείο δεν υπάρχει, θέλουμε το πρόγραμμά μας να χειριστεί την εξαίρεση.**

### Κριτήριο αξιολόγησης 3 (Βαθμός δυσκολίας:●)

**Γράψτε δύο τρόπους αποφυγής λαθών στον προγραμματισμό.**

# ΚΕΦΑΛΑΙΟ 8

## Κλάσεις και αντικείμενα

*Σύνοψη*

*Σ' αυτό το κεφάλαιο εμβαθύνουμε σε έννοιες του αντικειμενοστραφούς προγραμματισμού όπως οι κλάσεις, τα αντικείμενα, ο πολυμορφισμός, και η κληρονομικότητα.*

*Προαπαιτούμενη γνώση*

*Κεφάλαια Ι-7 του παρόντος συγγράμματος.*

### 8.1 Εισαγωγή

Θα ξεκινήσουμε την περιήγησή μας στις έννοιες του αντικειμενοστραφούς προγραμματισμού εξετάζοντας πώς ένας χρήστης μπορεί να ορίσει έναν νέο τύπο. Θα χρησιμοποιήσουμε σαν παράδειγμα το γεωμετρικό σημείο (point), το οποίο ορίζεται στις δύο διαστάσεις με δύο αριθμούς (συντεταγμένες) μέσα σε παρενθέσεις, όπως για παράδειγμα τα σημεία (0,0) και (x,y).

Ένας τρόπος να παρασταθεί το σημείο στην Python είναι με τις τιμές δύο πραγματικών αριθμών. Το ερώτημα είναι πώς να ομαδοποιήσουμε τις τιμές αυτές σε ένα σύνθετο αντικείμενο. Μια προφανής απάντηση είναι η χρήση μιας λίστας ή μιας πλειάδας. Μια εναλλακτική λύση είναι να ορίσουμε έναν καινούργιο σύνθετο τύπο ο οποίος *εκτός από δεδομένα μπορεί να περικλείει και λειτουργικότητα*. Η μέθοδος συγγραφής προγραμμάτων, η οποία συνδυάζει δεδομένα και λειτουργικότητα εμπερικλείοντάς τα σε ένα αντικείμενο, ονομάζεται *αντικειμενοστραφής προγραμματισμός (object-oriented programming)* (Booch, 2007).

Ένας σύνθετος τύπος με τα παραπάνω χαρακτηριστικά ονομάζεται **κλάση (class).** Ένα παράδειγμα ορισμού μιας πολύ απλής κλάσης η οποία υλοποιεί το σύνθετο τύπο Point είναι το παρακάτω:

8-class_definition.py
```
1 class Point:
2     pass
```

**Εικόνα 8.1** *Ορισμός κλάσης*

Οι ορισμοί κλάσεων είναι δυνατό να βρίσκονται παντού σε ένα πρόγραμμα, αλλά, συνήθως, βρίσκονται κοντά στην αρχή (μετά τις εντολές import). Στο παραπάνω παράδειγμα η εντολή pass δεν έχει κάποιο αποτέλεσμα, είναι εκεί γιατί η σύνθετη εντολή πρέπει να έχει κάτι στο σώμα της. Δημιουργώντας την κλάση Point, δημιουργούμε έναν καινούργιο τύπο που, επίσης, ονομάζεται Point. Τα μέλη ενός καινούργιου τύπου ονομάζονται *υποστάσεις (instances)* ή *αντικείμενα (objects)* και η δημιουργία μιας υπόστασης ονομάζεται *υποστασιασμός (instantiation)*.

Για να δημιουργήσουμε ένα αντικείμενο τύπου Point, καλούμε την Point σαν να ήταν συνάρτηση:

```
>>> blank = Point()
```

**Εικόνα 8.2** *Δημιουργία αντικειμένου*

Η επιστρεφόμενη τιμή είναι μια αναφορά σε ένα Point αντικείμενο, την οποία και εκχωρούμε στη μεταβλητή blank. Με την παρακάτω εντολή μπορούμε να εκτυπώσουμε μια υπόσταση κλάσης:

```
>>> print(blank)
```

**Εικόνα 8.3** *Εκτύπωση μιας υπόστασης*

```
<__main__.Point instance at 0x0000000002E64E48>
```

**Εικόνα 8.4** *Έξοδος εκτύπωσης μιας υπόστασης*

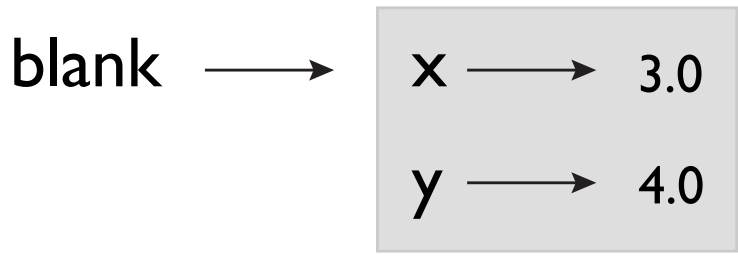
## 8.2 Γνωρίσματα ή ιδιότητες (attributes)

Μπορούμε να προσθέσουμε δεδομένα στην υπόσταση, όπως στο παρακάτω παράδειγμα, στο οποίο ορίζουμε και αναθέτουμε τιμές σε μεταβλητές x και y στην υπόσταση της κλάσης Point, που δείχνει η αναφορά blank. Ο τρόπος αυτός αναφοράς σε μια μεταβλητή κλάσης ονομάζεται *dot notation*.

```
>>> blank.x = 3.0
>>> blank.y = 4.0
```

**Εικόνα 8.5** *Προσθήκη δεδομένων στην υπόσταση*

Οι μεταβλητές που επιλέγουμε από μια υπόσταση (όπως οι x και y παραπάνω) ονομάζονται *γνωρίσματα ή ιδιότητες (attributes)*. Το παρακάτω διάγραμμα δείχνει τα γνωρίσματα του αντικειμένου blank. Κάθε γνώρισμα αναφέρεται σε έναν πραγματικό αριθμό:




**Εικόνα 8.6** *Διάγραμμα αντικειμένου*

Μπορούμε να δούμε το περιεχόμενο ενός γνωρίσματος βάσει της ίδιας σύνταξης, όπως και στην ανάθεση. Στην Εικόνα 8.7 η έκφραση blank.x σημαίνει: «Πήγαινε στο αντικείμενο στο οποίο αναφέρεται η μεταβλητή blank και πάρε την τιμή του x». Δεν υπάρχει «σύγκρουση» μεταξύ του γνωρίσματος blank.x και της μεταβλητής x:

```
>>> print(blank.y)
4.0
>>> x = blank.x
>>> print(x)
3.0
```

**Εικόνα 8.7** *Εκτύπωση περιεχομένου ενός γνωρίσματος αντικειμένου*

Μπορούμε να χρησιμοποιούμε *dot notation* ως μέρος οποιασδήποτε έκφρασης:

```
>>> print('(' +str(blank.x) +',' +str(blank.y) +')')
(3.0,4.0)
>>> distanceSquared = blank.x * blank.x + blank.y*blank.y
>>> print(distanceSquared)
25.0
```

**Εικόνα 8.8** *dot notation ως μέρος οποιασδήποτε έκφρασης*

## 8.3 Υποστάσεις ως ορίσματα

Αναφορές σε υποστάσεις κλάσεων (όπως η blank στην Εικόνα 8.9) μπορούν να περαστούν σαν ορίσματα σε συναρτήσεις της Python. Παράδειγμα:

```
>>> def printPoint(p):
    print('(' + str(p.x)+ ',' +str(p.y) + ')')


>>> printPoint(blank)
(3.0,4.0)
```

**Εικόνα 8.9** *Οι υποστάσεις ως γνωρίσματα*

## 8.4 Ισότητα

Η έννοια της «ισότητας» στα αντικείμενα μπορεί να οριστεί με δύο τρόπους. Δύο (αντικείμενα) σημεία (points) μπορούμε να πούμε ότι είναι ίδια, είτε αν περιέχουν τα ίδια δεδομένα (συντεταγμένες) ή αν είναι στην πραγματικότητα το ίδιο αντικείμενο. Ο γνωστός μας τελεστής == ελέγχει αν δύο αναφορές δείχνουν στο ίδιο ακριβώς αντικείμενο. Για παράδειγμα, στην Εικόνα 8.10 κάνουμε τη σύγκριση δύο διαφορετικών αντικειμένων p1 και p2, τα οποία έχουν, όμως, τα ίδια περιεχόμενα.

Παρόλο που τα p1 και p2 περιέχουν γνωρίσματα με τις ίδιες τιμές, δεν είναι το ίδιο αντικείμενο και άρα η σύγκριση μέσω του == επιστρέφει «ψευδές».

8-instances_equality.py

```
1  p1 = Point()
2  p1.x = 3.0
3  p1.y = 4.0
4  p2 = Point()
5  p2.x = 3.0
6  p2.y = 4.0
7  print 'Ισότητα', p1 == p2
```

Εκτέλεση

```
>>> print'Ισότητα', p1 == p2
Ισότητα False
```

**Εικόνα 8.10** *Έλεγχος αναφορικά με το αν δύο αναφορές γίνονται στο ίδιο αντικείμενο*

Αν, ωστόσο, εκχωρήσουμε το p1 στο p2, τότε οι δύο μεταβλητές είναι ψευδώνυμα του ίδιου αντικειμένου και άρα η σύγκρισή τους μέσω του τελεστή == επιστρέφει «αληθές»:

```
p2 = p1
print 'Ισότητα', p1 == p2
```

Εκτέλεση

```
>>> print 'Ισότητα', p1 == p2
Ισότητα True
```

**Εικόνα 8.10** *Εκχώρηση μιας αναφοράς αντικειμένου σε μια άλλη*

Αυτός ο τύπος ισότητας ονομάζεται **ρηχός (shallow equality),** γιατί συγκρίνει μόνον τις αναφορές και όχι τα περιεχόμενα των αντικειμένων. Αντικείμενα τα οποία έχουν το ίδιο περιεχόμενο σε πρώτο επίπεδο μπορεί να φαίνονται ως ίσα

με βάση το κριτήριο της ρηχής ισότητας, αλλά μπορεί να διαφέρουν βάσει του συνολικού περιεχομένου τους, ακολουθώντας όλες τις αναφορές σε εμφωλευμένα αντικείμενα, σε όλα τα επίπεδα. Για να ελέγξουμε, λοιπόν, δύο αντικείμενα για **βαθιά ισότητα (deep equality),** πρέπει να συγκρίνουμε, πλήρως, τα περιεχόμενα των αντικειμένων αυτών.

8-samePoint.py

```
1  def samePoint(p1, p2):
2      return (p1.x == p2.x) and (p1.y == p2.y)
3
```

**Εικόνα 8.11** *Βαθιά ισότητα*

Στο παράδειγμα της Εικόνας 8.12 ελέγχουμε αν δύο αντικείμενα-σημεία αναπαριστούν το ίδιο σημείο:

8-samePoint.py

```
1  def samePoint(p1, p2):
2      return (p1.x == p2.x) and (p1.y == p2.y)
3
4  p1 = Point()
5  p1.x = 3.0
6  p1.y = 4.0
7  p2 = Point()
8  p2.x = 3.0
9  p2.y = 4.0
10
11 print 'Ισότητα', samePoint(p1,p2)
```

Εκτέλεση

```
Ισότητα True
```

**Εικόνα 8.12** *Βαθιά ισότητα*

## 8.5 Ορθογώνια

Στη συνέχεια θα ορίσουμε μια νέα κλάση η οποία να αναπαριστά ένα ορθογώνιο παραλληλόγραμμο. Για απλότητα ας υποθέσουμε ότι το ορθογώνιο δε βρίσκεται σε γωνία σε σχέση με το σύστημα συντεταγμένων.

Υπάρχουν διάφοροι τρόποι με τους οποίους μπορούμε να αναπαραστήσουμε την πληροφορία που ορίζει ένα ορθογώνιο παραλληλόγραμμο:

- Το κέντρο του ορθογωνίου (δύο συντεταγμένες) και μέγεθος (ύψος και μήκος),
- Η κορυφή μιας γωνίας και μέγεθος,
- δύο αντίθετες γωνίες,
- Ή άνω αριστερή γωνία του ορθογωνίου και μέγεθος.

Ορίζουμε, λοιπόν, μια νέα κλάση (Rectangle) για το παραλληλόγραμμο:

8-Rectangle_class.py

```
1 class Rectangle:
2     pass
3
```

**Εικόνα 8.13** *Ορισμός κλάσης ορθογωνίου*

και δημιουργούμε μια υπόστασή της με το όνομα box:

```
box = Rectangle()
box.width = 100.0
box.height = 200.0
```

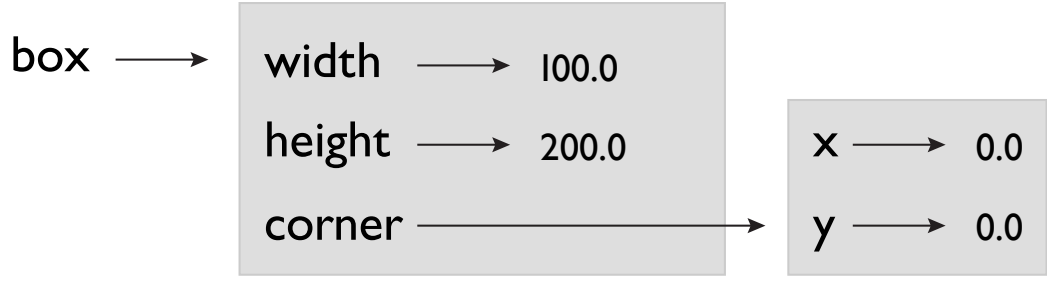
**Εικόνα 8.14** *Δημιουργία αντικειμένου ορθογωνίου*

Για να ορίσουμε μια γωνία του ορθογωνίου (π.χ. άνω αριστερή), μπορούμε να «εμφυτεύσουμε» ένα αντικείμενο μέσα σε ένα άλλο:

```
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0
```

**Εικόνα 8.15** *Ορισμός γωνίας του ορθογωνίου*

Ένα αντικείμενο το οποίο είναι γνώρισμα (attribute) σε ένα άλλο αντικείμενο, είναι εμφωλευμένο. Το επόμενο σχήμα δείχνει την κατάσταση του αντικειμένου box το οποίο ορίσαμε στις Εικόνες 8.14 και 8.15:




**Εικόνα 8.16** *Εμφωλευμένο αντικείμενο*

## 8.6 Υποστάσεις ως επιστρεφόμενες τιμές

Οι συναρτήσεις μπορούν να επιστρέφουν υποστάσεις. Για παράδειγμα στην Εικόνα 8.17 βλέπουμε την περίπτωση της συνάρτησης findCenter που επιστρέφει την υπόσταση (αναφορά) p της κλάσης Point:

```
>>> def findCenter(box):
    p = Point()
    p.x = box.corner.x + box.width/2.0
    p.y = box.corner.y - box.height/2.0
    return p

>>> center =findCenter(box)
>>> printPoint(center)
(50.0,-100.0)
```

**Εικόνα 8.17** *Η υπόσταση ως επιστρεφόμενη τιμή*

## 8.7 Τα αντικείμενα είναι μετατρέψιμα (mutable)

Μπορούμε να αλλάξουμε την κατάσταση ενός αντικειμένου εκχωρώντας νέες τιμές στα γνωρίσματά του:

```
box.width = box.width + 50.0
box.height = box.height + 100.0
```

**Εικόνα 8.18** *Αλλαγή κατάστασης ενός αντικειμένου με εκχώρηση νέων τιμών στα γνωρίσματά του*

Ο κώδικας της Εικόνας 8.18 μπορεί να ενθυλακωθεί σε συνάρτηση, ώστε να μπορούμε να εκτελέσουμε την αλλαγή κατάστασης του αντικειμένου σε κάθε κλήση της growRect:

8-growRect.py

```
1  def growRect(box, dwidth, dheight):
2      box.width = box.width + dwidth
3      box.height = box.height + dheight
```

**Εικόνα 8.19** *Ενθυλάκωση κώδικα σε συνάρτηση*

Παράδειγμα χρήσης:

```
bob=Rectangle()
bob.width = 100.0
bob.height = 200.0
bob.corner = Point()
bob.corner.x = 0.0
bob.corner.y = 0.0
growRect(bob, 50.0 , 100.0)
```

**Εικόνα 8.20** *Παράδειγμα χρήσης της growRect*

Στο παράδειγμα της Εικόνας 8.20 περνάμε την παράμετρο bob (αναφορά σε υπόσταση της κλάσης Rectangle) σαν πρώτο όρισμα στην growRect. Προσέξτε ότι για όσο τρέχει η growRect, η παράμετρος box είναι ψευδώνυμο (alias) της bob.

## 8.8 Αντιγραφή

Στην Εικόνα 8.10 είδαμε πόσο εύκολα μπορούμε να παράγουμε ψευδώνυμα για μια υπόσταση κλάσης (μέσω μιας απλής ανάθεσης). Επειδή πολλές φορές επιθυμούμε να παράγουμε αντίγραφα μιας υπόστασης, η Python μάς παρέχει το αρχείο (module) copy, που περιέχει μια συνάρτηση με το όνομα copy και η οποία μπορεί να αντιγράψει κάθε αντικείμενο, όπως φαίνεται στο παράδειγμα της Εικόνας 8.21:

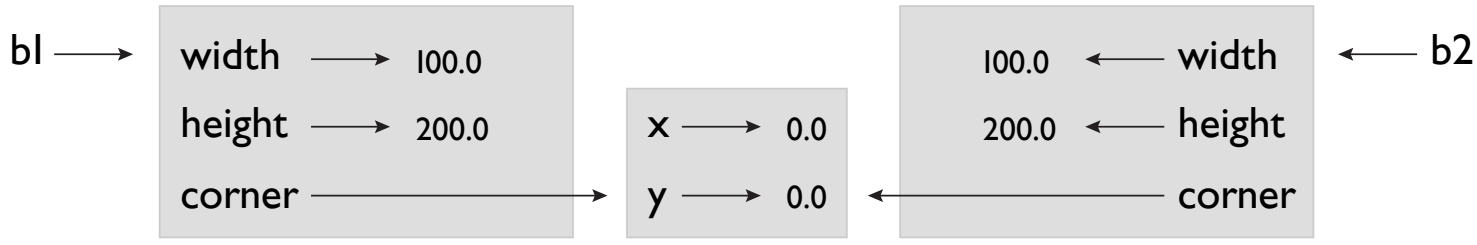
```
>>> import copy
>>> p1 = Point()
>>> p1.x = 3.0
>>> p1.y = 4.0
>>> p2 = copy.copy(p1)
>>> print(p1 == p2)
False
>>> print(samePoint(p1,p2))
True
```

**Εικόνα 8.21** *Η συνάρτηση copy για αντικείμενα*

Δυστυχώς, η αντιγραφή μέσω της κλήσης του copy.copy() γίνεται μόνο σε ένα επίπεδο, δηλαδή είναι ρηχή. Στην Εικόνα 8.22 ορίζουμε μια υπόσταση bl της Rectangle, την οποία αντιγράφουμε στην b2. Ενώ οι δύο υποστάσεις δε φαίνονται ίσες με βάση τον τελεστή == (επειδή είναι διαφορετικές αναφορές), εμπεριέχουν την ίδια αναφορά στο σημείο corner. Όπως φαίνεται στην Εικόνα 8.23, η αντιγραφή γίνεται μόνο σε ένα επίπεδο (δεν έχει αντιγραφεί το corner):

```
>>> import copy
>>> b1 = Rectangle()
>>> b1.width = 100.0
>>> b1.height = 200.0
>>> b1.corner = Point()
>>> b1.corner.x = 0.0
>>> b1.corner.y = 0.0
>>> b2 = copy.copy(b1)
>>> print(b1 == b2)
False
>>> print(b1.corner == b2.corner)
True
```

**Εικόνα 8.22** *Ρηχή αντιγραφή*




**Εικόνα 8.22** *Η αντιγραφή γίνεται μόνο σε ένα επίπεδο*

Αν θέλουμε ένα πλήρες αντίγραφο (σε όσα επίπεδα χρειάζεται), υπάρχει η συνάρτηση *deepcopy* η οποία το κάνει αυτό σωστά. Στην Εικόνα 8.23 φαίνεται ότι η *deepcopy* παράγει δύο διαφορετικά αντικείμενα bl και b2:

```
>>> import copy
>>> b1 = Rectangle()
>>> b1.width = 100.0
>>> b1.height = 200.0
>>> b1.corner = Point()
>>> b1.corner.x = 0.0
>>> b1.corner.y = 0.0
>>> b2 = copy.deepcopy(b1)
>>> print(b1 == b2)
False
>>> print(b1.corner == b2.corner)
False
```

**Εικόνα 8.23** *Η συνάρτηση deepcopy*

Ο αναγνώστης καλείται να παρακολουθήσει και το διαδραστικό υλικό που συνοδεύει αυτήν την ενότητα.

## 8.9 Κλάσεις & συναρτήσεις

Στη συνέχεια θα δούμε την αλληλεπίδραση των κλάσεων με τις συναρτήσεις και θα αναφέρουμε κάποια χαρακτηριστικά των τελευταίων, όπως οι καθαρές συναρτήσεις και οι τροποποιητές.

### 8.9.1 Χρόνος - Time

Για τις ανάγκες αυτής της ενότητας θα ορίσουμε μια νέα κλάση που θα ονομάσουμε Time. Η Time αποτελεί ένα νέο τύπο για την καταγραφή του χρόνου:

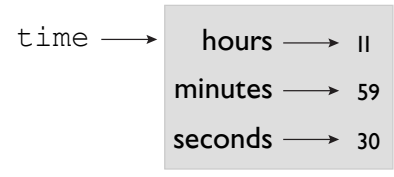
8-class_Time.py

```
1 class Time:
2     pass
```

**Εικόνα 8.24** *Ορισμός κλάσης.*

Στη συνέχεια δημιουργούμε ένα νέο αντικείμενο της κλάσης Time και ορίζουμε τα τρία ορίσματά του: ώρες, λεπτά, και δευτερόλεπτα:

```
time = Time ()
time.hours = 11
time.minutes = 59
time.seconds = 30
```




**Εικόνα 8.25** *Δημιουργία αντικειμένου*

### 8.9.2 Καθαρές συναρτήσεις και τροποποιητές

Στη συνέχεια θα γράψουμε μια συνάρτηση (addTime) η οποία υπολογίζει το άθροισμα δύο αντικειμένων της κλάσης Time και θα εξετάσουμε δύο παραλλαγές της. Μια παραλλαγή σε μορφή **καθαρής συνάρτησης (pure function)** και μια άλλη σε μορφή **τροποποιητή (modifier).**

Στην Εικόνα 8.26 βλέπουμε την πρώτη παραλλαγή της addTime. Η συνάρτηση αυτή είναι καθαρή συνάρτηση, γιατί δεν τροποποιεί κανένα από τα αντικείμενα που της δίνονται ως ορίσματα και δεν έχει άλλες «παρενέργειες» (side effects) όπως η επίδειξη (εκτύπωση) τιμής ή η είσοδος από χρήστη:

8-addTime.py

```
1  def addTime(t1, t2):
2      sumt = Time()
3      sumt.hours = t1.hours + t2.hours
4      sumt.minutes = t1.minutes + t2.minutes
5      sumt.seconds = t1.seconds + t2.seconds
6      return sumt
```

**Εικόνα 8.26** *Ορισμός της συνάρτησης addTime*

Παρακάτω βλέπουμε ένα παράδειγμα χρήσης της addTime. Προσέξτε ότι ορίζουμε και τη συνάρτηση printTime, η οποία εκτυπώνει μορφοποιημένα τα γνωρίσματα της time:

```
def printTime(time):
    print time.hours, ':', time.minutes, ':', time.seconds

start = Time()
start.hours = 9
start.minutes = 45
start.seconds = 5
duration = Time()
duration.hours = 1
duration.minutes = 35
duration.seconds = 10
done = addTime(start,duration)
printTime done
```

Εκτέλεση

```
10 : 80 : 15
```

**Εικόνα 8.27** *Παράδειγμα χρήσης της addTime*

Η συνάρτηση addTime δε λειτουργεί ορθά, όταν το άθροισμα των λεπτών ή των δευτερολέπτων υπερβαίνει το 60. Η λύση είναι να «μεταφέρουμε» τα έξτρα δευτερόλεπτα στα λεπτά και τα έξτρα λεπτά στις ώρες. Η δεύτερη και βελτιωμένη παραλλαγή της addTime φαίνεται στην Εικόνα 8.28:

8-addTime2.py

```
1  def addTime(t1, t2):
2      sumt = Time()
3      sumt.hours = t1.hours + t2.hours
4      sumt.minutes = t1.minutes + t2.minutes
5      sumt.seconds = t1.seconds + t2.seconds
6
7      if sumt.seconds >=60:
8          sumt.seconds = sumt.seconds - 60
9          sumt.minutes = sumt.minutes + 1
10
11     if sumt.minutes >=60:
12         sumt.minutes = sumt.minutes -60
13         sumt.hours = sumt.hours +1
14
15
16     return sumt
```

**Εικόνα 8.28** *Βελτιωμένη παραλλαγή της addTime*

Είναι μεν μια σωστή υλοποίηση αλλά αρκετά εκτενής. Στο προηγούμενο παράδειγμα χρήσης αυτή η υλοποίηση θα τύπωνε: 11 : 20 : 15.

### 8.9.3 Τροποποιητές

Σε αντίθεση με τις καθαρές συναρτήσεις, υπάρχουν περιπτώσεις όπου είναι χρήσιμο μια συνάρτηση να τροποποιεί ένα ή περισσότερα από τα αντικείμενα που παίρνει ως ορίσματα. Συνήθως, αυτός ο οποίος καλεί τη συνάρτηση, κρατάει μια αναφορά στο αντικείμενο που περνάει ως όρισμα, για να μπορεί να δει αργότερα τις αλλαγές. Συναρτήσεις αυτού του είδους λέγονται τροποποιητές.

Παράδειγμα τροποποιητή ο οποίος παίρνει ως όρισμα μια αναφορά στο αντικείμενο time το οποίο τροποποιεί, παρατίθεται αμέσως μετά:

8-increment.py

```
1  def increment(time, seconds):
2      time.seconds = time.seconds + seconds
3      if time.seconds >= 60:
4          time.seconds = time.seconds - 60
5          time.minutes = time.minutes + 1
6      if time.minutes >= 60:
7          time.minutes = time.minutes - 60
8          time.hours = time.hours +1
9
```

**Εικόνα 8.29** *Συνάρτηση της μορφής τροποποιητή*

Αναφορικά με την προαναφερόμενη συνάρτηση πρέπει να παρατηρήσουμε αν είναι σωστή. Επίσης, πρέπει να δούμε: *τι γίνεται όταν το seconds γίνει πολύ μεγαλύτερο από 60;* Λύση στο παραπάνω πρόβλημα δίνει η παρακάτω υλοποίηση:

8-increment2.py

```
1  def increment(time, seconds):
2      time.seconds = time.seconds + seconds
3      while time.seconds >= 60:
4          time.seconds = time.seconds - 60
5          time.minutes = time.minutes + 1
6
7      while time.minutes >= 60:
8          time.minutes = time.minutes - 60
9          time.hours = time.hours +1
```

**Εικόνα 8.30** *Τροποποιητής*

Τώρα η συνάρτηση είναι σωστή αλλά όχι και η πιο αποτελεσματική. Θεωρητικά, ό,τι μπορεί να υλοποιηθεί με τροποποιητές, μπορεί να υλοποιηθεί και με καθαρές συναρτήσεις. Μάλιστα, μερικές γλώσσες προγραμματισμού επιτρέπουν μόνον καθαρές συναρτήσεις, και η εμπειρία δείχνει ότι προγράμματα που χρησιμοποιούν μόνον καθαρές συναρτήσεις, τείνουν να έχουν λιγότερα λάθη. Ο προγραμματισμός με καθαρές συναρτήσεις λέγεται συναρτησιακό στυλ προγραμματισμού (functional programming style). Ωστόσο, δεν είναι πάντα πρακτικό να χρησιμοποιούμε καθαρές συναρτήσεις, ιδίως όταν τα αντικείμενα είναι πάρα πολύ μεγάλα λόγω του κόστους της δημιουργίας αντιγράφων των ορισμάτων, σε κάθε κλήση αυτών των συναρτήσεων.

Ο αναγνώστης καλείται να παρακολουθήσει και το διαδραστικό υλικό που συνοδεύει αυτήν την ενότητα.

## 8.10 Ανάπτυξη πρωτοτύπου και σχεδιασμός

Σε αυτήν την ενότητα δείξαμε πώς αναπτύσσουμε προγράμματα αρχίζοντας με πρόχειρες παραλλαγές (πρωτότυπα), τα οποία στη συνέχεια τελειοποιήσαμε. Παρόλο που αυτή είναι μια μέθοδος η οποία παράγει αποτελέσματα, τείνει να παράγει κώδικα που είναι πολύπλοκος, γιατί προσπαθεί να χειριστεί ειδικές περιπτώσεις και ποτέ δεν ξέρουμε αν έχουμε βρει όλα τα λάθη.

Εναλλακτικά, μπορούμε να σχεδιάσουμε την ανάπτυξη του προγράμματος στοχεύοντας εξαρχής στην τελική του λύση. Στην περίπτωσή μας η τελική λύση θα μπορούσε να είναι ένα αντικείμενο Time σχεδιασμένο σαν ένας τριψήφιος αριθμός εκφραζόμενος σε αριθμητικό σύστημα με βάση το 60! Όταν γράψαμε τις addTime and increment, ουσιαστικά κάναμε πρόσθεση με βάση το 60, και γι' αυτό μεταφέραμε το κρατούμενο από τη μια ιδιότητα (στοιχείο του αντικειμένου) στην άλλη. Αυτή η παρατήρηση οδηγεί σε μια διαφορετική προσέγγιση του προβλήματος. Μετατρέπουμε, λοιπόν, πρώτα το αντικείμενο Time σε ακέραιο (δευτερόλεπτα):

8-convert_Time.py

```
1  def convertToSeconds(t):
2      minutes = t.hours*60 + t.minutes
3      seconds = minutes*60 + t.seconds
4      return  seconds
```

**Εικόνα 8.31** *Συνάρτηση μετατροπής του χρόνου σε δευτερόλεπτα*

Και τώρα το αντίστροφο:

```
def makeTime(seconds):
    time = Time()
    time.hours = seconds // 3600
    time.minutes = (seconds%3600)  // 60
    time.seconds = seconds%60
    return time
```

**Εικόνα 8.32** *Συνάρτηση μετατροπής από δευτερόλεπτα σε χρόνο*

Έτσι μπορούμε να ξαναγράψουμε την addTime:

```
def addTime(t1,t2):
    seconds = convertToSeconds(t1) + convertToSeconds(t2)
    return makeTime(seconds)
```

**Εικόνα 8.33** *Γράφουμε ξανά την addTime*

# 8.II Κλάσεις & μέθοδοι

## 8.II.I Αντικειμενοστραφή χαρακτηριστικά της Python

Μερικά από τα χαρακτηριστικά του αντικειμενοστραφούς προγραμματισμού τα οποία μας προσφέρει η Python, είναι τα εξής:

- Κάθε ορισμός αντικειμένου αντιστοιχεί σε κάποιο αντικείμενο ή μια έννοια του πραγματικού κόσμου και οι συναρτήσεις που κάνουν πράξεις πάνω σ' ένα αντικείμενο, αντιστοιχούν σε τρόπους με τους οποίους τα αντικείμενα του πραγματικού κόσμου αλληλεπιδρούν.
    - π.χ. κλάση Point και μαθηματικός ορισμός σημείου

- Δεδομένα και λειτουργικότητα περικλείονται σε ένα αντικείμενο.

Τα προγράμματα αποτελούνται, κυρίως, από ορισμούς αντικειμένων και ορισμούς συναρτήσεων και οι υπολογισμοί εκφράζονται, συνήθως, ως πράξεις πάνω σε αντικείμενα.

Ως τώρα, σ' αυτό το κεφάλαιο ορίσαμε τις κλάσεις Point, Rectangle, και Time. Μέχρι στιγμής δεν έχουμε εκμεταλλευτεί τα παραπάνω ειδικά χαρακτηριστικά της Python για αντικειμενοστραφή προγραμματισμό. Ενώ αυτά τα χαρακτηριστικά δεν είναι απαραίτητα, όταν τα χρησιμοποιούμε, φτιάχνουμε πιο συνοπτικά προγράμματα και με καλύτερη δομή. Για παράδειγμα, στην κλάση Time δεν υπάρχει προφανής συσχέτιση μεταξύ του *ορισμού της κλάσης* και των *ορισμών των συναρτήσεων* που ακολουθούν, παρόλο που οι συναρτήσεις αυτές παίρνουν, τουλάχιστον, ένα όρισμα που είναι αντικείμενο της κλάσης Time.

Η παρατήρηση αυτή είναι το κίνητρο για τη χρήση των μεθόδων. Πήραμε, ήδη, μια ιδέα της χρήσης των μεθόδων στις συμβολοσειρές, στις λίστες, στις πλειάδες και στα λεξικά. Κάθε μέθοδος συσχετίζεται με μια κλάση και ο σκοπός είναι να γίνεται επίκληση της μεθόδου για αντικείμενα αυτής της κλάσης.

Οι μέθοδοι είναι όπως οι συναρτήσεις, με δύο διαφορές:

- Οι μέθοδοι ορίζονται μέσα σε έναν ορισμό κλάσης, δημιουργώντας άμεση σχέση μεταξύ της κλάσης και των μεθόδων της.
- Η σύνταξη για την επίκληση μιας μεθόδου είναι διαφορετική από τη σύνταξη για την κλήση μιας συνάρτησης.

Στη συνέχεια κάποιες συναρτήσεις τις οποίες έχουμε, ήδη, ορίσει θα τις μετατρέψουμε με εύκολο τρόπο σε μεθόδους.

### 8.II.2 Μετατρέποντας τη συνάρτηση printTime σε μέθοδο

Νωρίτερα χρησιμοποιήσαμε τη συνάρτηση printTime (Εικόνα 8.27), η οποία ορίζεται με τον παρακάτω τρόπο:

8-printTime.py

```
1  class Time:
2      pass
3  def printTime(time):
4      print(str(time.hours)+':'+str(time.minutes)+':'+str(time.seconds))
```

**Εικόνα 8.34** *Η printTime ως συνάρτηση*

Ένα παράδειγμα για τον τρόπο που καλούμε τη συνάρτηση printTime είναι το παρακάτω:

```
>>> currentTime = Time()
>>> currentTime.hours = 9
>>> currentTime.minutes = 14
>>> currentTime.seconds = 30
>>> printTime(currentTime)
9:14:30
```

**Εικόνα 8.35** *Παράδειγμα κλήσης της συνάρτησης printTime*

Η συνάρτηση printTime παίρνει σαν μοναδικό όρισμα μια υπόσταση της κλάσης Time και δρα σε αυτήν την υπόσταση, οπότε είναι μια εξαιρετική επιλογή συνάρτησης η οποία να μπορεί να εξελιχθεί σε μέθοδο της κλάσης Time. Αυτό μπορεί να συμβεί με την απλή μετακίνηση του ορισμού της συνάρτησης μέσα στον ορισμό της κλάσης. Προσέξτε τις εσοχές (indentations) του κώδικα:

```
class Time:
    def printTime(time):
        print(str(time.hours)+':'+str(time.minutes)+':'+str(time.seconds))
```

**Εικόνα 8.36** *Η printTime γίνεται μέθοδος*

Μπορούμε να επικαλεστούμε τη μέθοδο printTime με χρήση του *dot notation* («όνομα υπόστασης-τελεία-όνομα μεθόδου»):

```
>>> currentTime = Time()
>>> currentTime.hours = 9
>>> currentTime.minutes = 14
>>> currentTime.seconds = 30
>>> currentTime.printTime()
9:14:30
```

**Εικόνα 8.37** *Επίκληση της printTime με χρήση του dot notation*

Το αντικείμενο για το οποίο γίνεται επίκληση της μεθόδου είναι η πρώτη παράμετρος, άρα εν προκειμένω, το αντικείμενο currentTime εκχωρείται στην παράμετρο time. Κατά σύμβαση, η πρώτη παράμετρος μιας μεθόδου ονομάζεται self (εαυτός). Ο λόγος βασίζεται στην εξής αλληγορία:

- Η σύνταξη της κλήσης -συνάρτησης printTime(currentTime), υποβάλει την άποψη ότι *η συνάρτηση είναι το ενεργό υποκείμενο.* Είναι σαν να ορίζει:

  *«printTime! πάρε αυτό το αντικείμενο και τύπωσέ το!».*

- Στον αντικειμενοστραφή προγραμματισμό ωστόσο, *τα αντικείμενα είναι τα ενεργά υποκείμενα.* Η επίκληση currentTime.printTime() είναι σαν να ορίζει:

  *«currentTime! τύπωσε τον εαυτό σου!».*

Έτσι η printTime μπορεί, τελικά, να γραφεί ως εξής:

```
class Time:
    def printTime(self):
        print(str(self.hours)+':'+str(self.minutes)+':'+str(self.seconds))
```

**Εικόνα 8.38** *Διαφορετική συγγραφή της printTime*

### 8.II.3 Ένα ακόμα παράδειγμα

Θα δούμε στη συνέχεια πώς μπορούμε, με αντίστοιχο τρόπο, να μετατρέψουμε και τη συνάρτηση increment σε μέθοδο της Time:

8-increment_class_function.py

```
1  class Time:
2      def printTime(self):
3          print(str(self.hours)+':'+str(self.minutes)+':'+str(self.seconds))
4
5      def increment(self, seconds):
6          self.seconds = self.seconds + seconds
7          while self.seconds >= 60:
8              self.seconds = self.seconds - 60
9              self.minutes = self.minutes + 1
10
11         while self.minutes >= 60:
12             self.minutes = self.minutes - 60
13             self.hours = self.hours +1
14
15
16
17 currentTime = Time()
18 currentTime.hours = 9
19 currentTime.minutes = 14
20 currentTime.seconds = 30
21 currentTime.increment(100)
22 currentTime.printTime()
```

Εκτέλεση

```
9:16:10
```

**Εικόνα 8.39** *Μετατροπή της increment σε μέθοδο*

### 8.II.4 Ένα πιο περίπλοκο παράδειγμα

Στη συνέχεια θα ορίσουμε τη μέθοδο after, η οποία παίρνει δύο αντικείμενα Time ως ορίσματα και επιστρέφει «ορθό» (true), αν το πρώτο είναι μετά το δεύτερο, «λάθος» (false) σε διαφορετική περίπτωση. Εφόσον το πρώτο όρισμα αναφέρεται στο αντικείμενο, στα πλαίσια του οποίου καλείται η after, αυτό αναπαρίσταται ως το self, και άρα το δεύτερο όρισμα είναι μια αναφορά (time2) στο δεύτερο αντικείμενο της κλάσης Time:

8-after_class_function.py

```
1  class Time:
2      def printTime(self):
3          print(str(self.hours)+':'+str(self.minutes)+':'+str(self.seconds))
4
5      def increment(self, seconds):
6          self.seconds = self.seconds + seconds
7          while self.seconds >= 60:
8              self.seconds = self.seconds - 60
9              self.minutes = self.minutes + 1
10
11         while self.minutes >= 60:
12             self.minutes = self.minutes - 60
13             self.hours = self.hours +1
14
15     def after(self,time2):
16         if self.hours > time2.hours: return 1
17         if self.hours < time2.hours: return 0
18         if self.minutes > time2.minutes: return 1
19         if self.minutes > time2.minutes: return 0
20         if self.seconds > time2.seconds: return 1
21         return 0
```

**Εικόνα 8.40** *Η μέθοδος after*

Μπορούμε να επικαλεστούμε τη μέθοδο after ως εξής:

```
t1 = Time()
t1.hours = 9
t1.minutes = 14
t1.seconds = 30
t2 = Time()
t2.hours = 11
t2.minutes = 14
t2.seconds = 30

print(t1.after(t2))
```

Εκτέλεση

```
>>>
0
```

**Εικόνα 8.41** *Επίκληση της μεθόδου after*

## 8.12 Προαιρετικά ορίσματα

Μπορούμε να ορίσουμε συναρτήσεις με προαιρετική λίστα ορισμάτων. Για παράδειγμα, ας βελτιώσουμε τη συνάρτηση find, την οποία είδαμε στις συμβολοσειρές (Κεφάλαιο 6). Η αρχική παραλλαγή της find είναι:

8-find.py

```
1  def find(string,ch):
2      index = 0
3      while index < len(string):
4          if string[index] == ch:
5              return index
6          index = index +1
7      return -1
```

**Εικόνα 8.42** *Η συνάρτηση find*

Ενδεχομένως να θέλουμε να ξεκινήσουμε την αναζήτηση για το χαρακτήρα *ch* από μια διαφορετική θέση στη συμβολοσειρά (μετά τη θέση 0), χωρίς, ωστόσο, αυτό να είναι αναγκαίο σε κάθε κλήση της find. Μια βελτιωμένη έκδοση της find, λοιπόν (Εικόνα 8.43), μπορεί να πάρει μια τρίτη παράμετρο (start), η οποία, ωστόσο, είναι προαιρετική. Αν καλέσουμε τη find χωρίς τρίτο όρισμα, τότε το start θα πάρει την τιμή 0 («αντί άλλης τιμής»). Αυτό εκφράζεται γράφοντας start=0 στον ορισμό της παραμέτρου start. Διαφορετικά (εάν δώσουμε τρίτο όρισμα), θα περαστεί η τιμή του τρίτου ορίσματος:

8-find2.py

```
1  def find(string,ch,start=0):
2      index = start
3      while index < len(string):
4          if string[index] == ch:
5              return index
6          index = index +1
7      return -1
8
9  print(find('hello','e'))
10 print(find('hello','e',2))
```

Εκτέλεση

```
>>>
1
-1
```

**Εικόνα 8.43** *Η βελτιωμένη έκδοση της find*

## 8.13 Μέθοδος αρχικοποίησης

Η μέθοδος αρχικοποίησης είναι μια ειδική μέθοδος της οποίας η επίκληση γίνεται αυτόματα, όταν ένα αντικείμενο δημιουργείται. Το όνομα της μεθόδου είναι __init__ (δύο underscore χαρακτήρες, ακολουθούμενοι από το init, και από δύο ακόμα underscore χαρακτήρες).

Για παράδειγμα, για την κλάση Time, η μέθοδος αρχικοποίησης είναι:

8-class_Time_init.py

```
1  class Time:
2      def __init__(self, hours=0, minutes=0, seconds=0):
3          self.hours = hours
4          self.minutes = minutes
5          self.seconds = seconds
6
```

**Εικόνα 8.44** *Μέθοδος αρχικοποίησης μιας κλάσης*

Όταν δημιουργούμε ένα αντικείμενο της κλάσης Time, τα ορίσματα που δίνουμε, δίνονται στην __init__ (Εικόνα 8.45). Επειδή τα ορίσματα είναι προαιρετικά, μπορούμε να παραλείψουμε μερικά από αυτά ή και όλα. Παραδείγματα:

```
time1 = Time(9,20,10)
time1.printTime()

time2 = Time(9,20)
time2.printTime()

time3 = Time(9)
time3.printTime()

time4 = Time()
time4.printTime()
```

**Εκτέλεση**

```
>>>
9:20:10
9:20:0
9:0:0
0:0:0
```

**Εικόνα 8.45** *Διαφορετικές αρχικοποιήσεις αντικειμένων της κλάσης Time*

### 8.13.1 Επιστροφή στο "σημείο"

Επιστρέφουμε στο παράδειγμα της κλάσης σημείο (Point) για να την γράψουμε με αντικειμενοστραφή τρόπο. Αρχικά, ορίζουμε μια μέθοδο αρχικοποίησης και μια μέθοδο με την οποία τυποποιούμε τον τρόπο εκτύπωσης των αντικειμένων της κλάσης, όπως φαίνεται στην Εικόνα 8.46:

8-class_Point_init.py

```
1  class Point:
2      def __init__(self, x=0, y=0):
3          self.x = x
4          self.y = y
5      def __str__(self):
6          return '(' + str(self.x) + ',' + str(self.y) + ')'
7
```

**Εικόνα 8.46** *Η κλάση Point, γραμμένη με αντικειμενοστραφή τρόπο*

Με χρήση προαιρετικών ορισμάτων, όταν κατά την αρχικοποίηση της Point δεν παρέχονται συντεταγμένες, η __init__ τοποθετεί το σημείο στην αρχή των συντεταγμένων (0, 0) «αντί άλλης τιμής».

Όταν οποιαδήποτε κλάση ορίζει μια μέθοδο __str __ (όπως κάνει η Point), τότε αγνοείται η γενικής χρήσης συνάρτηση str της Python. Η εντολή print επικαλείται από μόνη της τη μέθοδο __str__ (αν υπάρχει) για το αντικείμενο, η οποία επιστρέφει μια αναπαράσταση του σημείου ως συμβολοσειρά. Άρα ο ορισμός της __str__ αλλάζει και τη συμπεριφορά της print:

```
>>> p  = Point(2,3)
>>> print(str(p))
(2,3)
>>> print(p)
(2,3)
```

**Εικόνα 8.47** *Αρχικοποίηση και εκτύπωση υπόστασης της κλάσης Point*

Οι __init__ και __str__ είναι χρήσιμες μέθοδοι τις οποίες καλό είναι να ορίζουμε κάθε φορά που δημιουργούμε μια νέα κλάση. Η πρώτη κάνει ευκολότερη τη δουλειά της αρχικοποίησης αντικειμένων, ενώ η δεύτερη μπορεί να βοηθήσει στην εκσφαλμάτωση (debugging).

## 8.14 Υπερφόρτωση τελεστών

Στην Python επιτρέπεται η αλλαγή του ορισμού των τελεστών (όπως ο +, *, κλπ.), όταν αυτοί εφαρμόζονται σε νέους σύνθετους τύπους, ορισμένους από τους χρήστες. Αυτή η λειτουργικότητα ονομάζεται *υπερφόρτωση (overloading)* των τελεστών και είναι ιδιαίτερα χρήσιμη όταν ορίζουμε νέους μαθηματικούς τύπους. Για παράδειγμα, μπορούμε να *υπερφορτώσουμε* τον τελεστή: πρόσθεση +, με τη γνώση που αφορά στον τρόπο πρόσθεσης δύο γεωμετρικών σημείων (ο ορισμός για τον τρόπο με τον οποίο το κάνουμε αυτό σε κάθε περίπτωση, εξαρτάται από τον προγραμματιστή). Έτσι, μπορούμε να ορίζουμε τη μέθοδο __add__ εντός της κλάσης Point, σε αντικείμενα της οποίας θα δράσει ο τελεστής +:

8-class_Point_add.py

```
1  class Point:
2      def __init__(self, x=0, y=0):
3          self.x = x
4          self.y = y
5      def __str__(self):
6          return '(' + str(self.x) + ',' + str(self.y) + ')'
7      def __add__(self, other):
8          return Point(self.x + other.x, self.y + other.y)
9
10 p1 = Point(3, 4)
11 p2 = Point(5, 7)
12 p3 = p1 + p2
13 print(p3)
```

Εκτέλεση

```
>>>
(8,11)
```

**Εικόνα 8.48** *Υπερφόρτωση του τελεστή πρόσθεσης*

Η έκφραση p1 + p2 είναι ισοδύναμη της έκφρασης p1.__add__ (p2), ωστόσο η πρώτη είναι, προφανώς, πιο κομψή.

Υπάρχουν διάφοροι τρόποι να υπερφορτώσει κανείς τον τελεστή πολλαπλασιασμού *: με τη μέθοδο __mul__ , τη μέθοδο __rmul__ ή και τις δύο. Αν το αριστερό όρισμα του * είναι ένα Point, τότε η Python επικαλείται τη __mul__, το οποίο υπο-

θέτει οτι και το άλλο όρισμα είναι επίσης ένα Point, και υπολογίζει το εσωτερικό γινόμενο των δύο σημείων:

```
def __mul__(self, other):
    return self.x * other.x + self.y * other.y
```

**Εικόνα 8.49** *Υπερφόρτωση του τελεστή* *

Αν το αριστερό όρισμα του * είναι ένας βασικός τύπος (δηλαδή, ακέραιος, πραγματικός, κ.λπ.) και το δεξί όρισμά του * είναι ένα Point, τότε η Python επικαλείται τη __rmul__ , η οποία ορίζεται ως εξής:

```
def __rmul__(self, other):
    return Point(other * self.x , other * self.y)
```

**Εικόνα 8.50** *Κλήση της μεθόδου __rmul__*

Προσέξτε ότι το δεξί όρισμα του + πρέπει σε κάθε περίπτωση να είναι ένα Point. Παραδείγματα:

```
p1 = Point(3, 4)
p2 = Point(5, 7)
print(p1 * p2)
print(2 * p2)
```

Εκτέλεση

```
>>>
43
(10,14)
```

**Εικόνα 8.5Ι** *Παράδειγμα υπερφόρτωσης του τελεστή* *

Τι γίνεται, ωστόσο, αν προσπαθήσουμε να χρησιμοποιήσουμε τον τελεστή * βάζοντας σαν αριστερό όρισμα το Point και δεξί έναν ακέραιο, όπως πχ. στο p2 * 2; Στην περίπτωση αυτή παίρνουμε ένα (όχι ιδιαίτερα διαφωτιστικό) μήνυμα λά-

θους, το οποίο μας λέει ότι προσπάθησε να εκτελέσει τη μέθοδο __mult__ της κλάσης Point θέτοντας ως δεύτερο όρισμα τον ακέραιο 2, και διαπίστωσε ότι δεν μπόρεσε να βρει γνώρισμα x στην κλάση των ακεραίων (όπως ήταν αναμενόμενο). Το πρόβλημα οφείλεται στο ότι δε χρησιμοποιήσαμε τα ορίσματα του τελεστή * στην ορθή σειρά (το δεξί να είναι οπωσδήποτε ένα Point).

```
>>> print(p2*2)

Traceback (most recent call last):
  File "<pyshell#136>", line 1, in <module>
    print(p2*2)
  File "C:\Users\IP\Desktop\M now\8-class_Point_mul.py", line 10, in __mul__
    return self.x * other.x + self.y * other.y
AttributeError: 'int' object has no attribute 'x'
```

**Εικόνα 8.52** *Σφάλμα εξόδου*

## 8.15 Πολυμορφισμός

Οι περισσότερες μέθοδοι που είδαμε, δουλεύουν με ένα συγκεκριμένο τύπο. Ωστόσο, υπάρχουν πράξεις που θα θέλαμε να εφαρμόζονται σε πολλούς τύπους, όπως οι αριθμητικές πράξεις τις οποίες ορίσαμε στα προηγούμενα. Μια συνάρτηση η οποία μπορεί να έχει ως ορίσματα διάφορους τύπους ονομάζεται *πολυμορφική*. Ο πολυμορφισμός διευκολύνει την επαναχρησιμοποίηση κώδικα, ώστε να μη χρειάζεται να γράφουμε διαφορετικές εκδόσεις συναρτήσεων για διαφορετικούς τύπους εισόδου.

Για παράδειγμα, η πράξη multadd παίρνει τρία ορίσματα, πολλαπλασιάζει τα δύο πρώτα και προσθέτει στο γινόμενο το τρίτο. Η multadd θα δουλέψει για όλες τις τιμές x και y τις οποίες μπορούμε να πολλαπλασιάσουμε και για κάθε z, το οποίο μπορεί να προστεθεί στο γινόμενο. Μπορούμε να γράψουμε τη multadd ως συνάρτηση στην Python ως εξής:

8-multadd.py

```
1 def multadd(x,y,z):
2     return x * y + z
```

**Εικόνα 8.53** *Ορισμός πολυμορφικής συνάρτησης multadd*

Βλέπουμε ένα παράδειγμα εκτέλεσης της multadd με αριθμούς:

```
>>> def multadd(x,y,z):
        return x * y + z

>>> print(multadd(2,4,6))
14
```

**Εικόνα 8.54** *Παράδειγμα χρήσης της multadd με αριθμούς*

Ένα ακόμα παράδειγμα με σημεία:

```
>>> p1 = Point(3,4)
>>> p2 = Point(5,7)
>>> print(multadd(2,p1,p2))
(11,15)
>>> print(multadd(p1,p2,1))
44
```

**Εικόνα 8.55** *Παράδειγμα χρήσης της multadd με σημεία*

Ο αναγνώστης καλείται να παρακολουθήσει και το διαδραστικό υλικό που συνοδεύει αυτήν την ενότητα.

## 8.16 Κληρονομικότητα

Το χαρακτηριστικό το οποίο συνδέεται συχνότερα με τις αντικειμενοστραφείς γλώσσες προγραμματισμού, είναι η κληρονομικότητα (inheritance). Κληρονομικότητα είναι η δυνατότητα να ορίσουμε μια νέα κλάση που είναι μια παραλλαγή μιας ήδη υπάρχουσας κλάσης. Το κύριο πλεονέκτημα αυτού του χαρακτηριστικού είναι ότι μπορούμε να ορίσουμε νέες μεθόδους στη νέα κλάση, χωρίς να μεταβάλουμε την ήδη υπάρχουσα. Η κληρονομικότητα έγκειται στο γεγονός ότι η νέα κλάση κληρονομεί όλες τις μεθόδους της υπάρχουσας κλάσης. Γι' αυτό και η υπάρχουσα κλάση ονομάζεται μερικές φορές και γονική κλάση (parent class), ενώ η νέα κλάση ονομάζεται κλάση-παιδί (child class) ή και υποκλάση (subclass). Σημειώστε ότι μία κλάση μπορεί να κληρονομεί από περισσότερες της μιας γονικές κλάσεις.

Η σύνταξη είναι σχετικά απλή, το μόνο πράγμα που αλλάζει για την κλάση-παιδί είναι ότι βάζουμε εντός παρενθέσεως σαν όρισμα την κλάση-γονιός (το ParentClass είναι σε παρένθεση στον ορισμό του ChildClass, όπως φαίνεται στην Εικόνα 8.56). Αν η κλάση ChildClass είναι παιδί της κλάσης ParentClass, τότε η πρώτη κληρονομεί όλες τις μεθόδους (εδώ δεν έχουμε ορίσει καμία μέθοδο ακόμα βέβαια) που έχουν ήδη οριστεί για την κλάση ParentClass:

8-inheritance_definition.py

```
1  class ParentClass:
2      pass
3
4  class ChildClass(ParentClass):
5      pass
6
7  class ChildClass(ParentClass1, ParentClass2):
8      pass
```

**Εικόνα 8.56** *Ορισμός υποκλάσης*

Σε περίπτωση που μια κλάση κληρονομεί από περισσότερες της μιας γονικής κλάσης (*πολλαπλή κληρονομικότητα*), προσθέτουμε στην παρένθεση μια λίστα από κλάσεις (ParentClass1, ParentClass2):

```
class ChildClass(ParentClass1, ParentClass2):
    pass
```

**Εικόνα 8.57** *Πολλαπλή κληρονομικότητα*

### 8.16.1 Πρώτο παράδειγμα

Ας δούμε ένα πρώτο απλό παράδειγμα. Στην Εικόνα 8.58, η κλάση SecondClass ορίζεται ως υποκλάση της γονικής κλάσης FirstClass και κληρονομεί από αυτήν τη μέθοδο αρχικοποίησης __init__. Η κλάση SecondClass έχει μια μέθοδο με όνομα display, η οποία διαφέρει από τη μέθοδο display της γονικής κλάσης. Προσέξτε ότι στην εντολή a.display() έχουμε επίκληση της μεθόδου display, που έχει οριστεί στην SecondClass και η οποία υπερτερεί (overrides) της γονικής με το ίδιο όνομα.

8-inheritance_first_example.py

```
1  class FirstClass:
2      def __init__(self,value):
3          self.data = value
4      def display(self):
5          print(self.data)
6
7  class SecondClass(FirstClass):
8      def display(self):
9          print 'Current value =', self.data
10
11 a = SecondClass(15)
12 a.display()
```

**Εκτέλεση**

```
Current value = 15
```

**Εικόνα 8.58** *Πρώτο παράδειγμα κληρονομικότητας*

### 8.16.2 Δεύτερο παράδειγμα

Στη συνέχεια θα δούμε ένα ακόμα, λίγο πιο περίπλοκο, παράδειγμα, εμπνευσμένο από την ακαδημαϊκή κοινότητα. Στο παράδειγμα αυτό, έχουμε μια στοιχειώδη καταγραφή δεδομένων που αφορούν σε καθηγητές και φοιτητές ενός Πανεπιστημίου. Οι καθηγητές και οι φοιτητές έχουν κάποια κοινά χαρακτηριστικά (ονοματεπώνυμο, ηλικία) αλλά και κάποια ιδιαίτερα χαρακτηριστικά (π.χ., μισθός για τους καθηγητές, αριθμός μητρώου (ΑΜ) για τους φοιτητές). Θα μπορούσαμε να

δημιουργήσουμε δύο ανεξάρτητες κλάσεις: μία για τους καθηγητές και μία για τους φοιτητές, αλλά η προσθήκη ενός νέου *κοινού* χαρακτηριστικού θα απαιτούσε την προσθήκη του και στις δύο αυτές ανεξάρτητες κλάσεις. Η προσέγγιση αυτή είναι δύσχρηστη. Καλύτερη λύση είναι να δημιουργήσουμε μια κοινή γονική κλάση (UniversityMember) και οι κλάσεις καθηγητής (Professor) και φοιτητής (Student) να *κληρονομήσουν* από αυτήν.

Τα πλεονεκτήματα αυτής της σχεδίασης έχουν ως εξής:

- Εάν προσθέσουμε/αλλάξουμε κάποια λειτουργία στην κλάση UniversityMember, αυτομάτως ενημερώνονται και οι υποκλάσεις.
- Εάν μπορούμε να αναφερθούμε σ' ένα αντικείμενο καθηγητή ή φοιτητή ως αντικείμενο UniversityMember, θα μας φανεί χρήσιμο σε περιπτώσεις όπως ο υπολογισμός του συνολικού αριθμού των μελών του Πανεπιστημίου.
- Επαναχρησιμοποιούμε τον κώδικα της γονικής κλάσης και δε χρειάζεται να τον επαναλαμβάνουμε.

Στην Εικόνα 8.59 ορίζουμε έναν αντικείμενο της κλάσης Professor. Στη μέθοδο αρχικοποίησης (__init__) αυτής της κλάσης δίνουμε τα τρία ορίσματα: name, age και salary.

Μπορούμε να επικαλεστούμε μεθόδους της γονικής κλάσης βάζοντας ως πρόθεμα το όνομά της. Για παράδειγμα:

```
UniversityMember.__init__(self, name, age)

UniversityMember.tell(self)
```

Με την εντολή member.tell() γίνεται επίκληση της μεθόδου tell της υποκλάσης. Αν δεν υπήρχε η μέθοδος tell της υποκλάσης, τότε η Python θα αναζητούσε τη μέθοδο αυτή στη γονική της κλάση.

```
class UniversityMember:
    def __init__(self, name, age):
        self.name = name
        self.age = age
        print('->Αρχικοποίηση μέλους του Πανεπιστημίου:', self.name)
    def tell(self):
        print('Ονοματεπώνυμο:"{0}}" Ηλικία:"{l}"'.format(self.name, self.age), end=' ')

class Professor(UniversityMember):
    def __init__(self, name, age, salary):
        UniversityMember.__init__(self, name, age)
        self.salary = salary
        print('->Αρχικοποίηση καθηγητή:', self.name)
    def tell(self):
        UniversityMember.tell(self)
        print('Μισθός:', self.salary)

class Student(UniversityMember):
    def __init__(self, name, age, am):
        UniversityMember.__init__(self, name, age)
        self.am = am
        print('->Αρχικοποίηση φοιτητή:', self.name)
    def tell(self):
        UniversityMember.tell(self)
        print('AM:', self.am)
```

**Εικόνα 8.59** *Δεύτερο παράδειγμα κληρονομικότητας*

Στην Εικόνα 8.60 βλέπουμε το κυρίως πρόγραμμα στο οποίο κατασκευάζουμε ένα αντικείμενο t της θυγατρικής κλάσης Professor με τα γνωρίσματα: name "Αναστασίου Αναστάσιος", ηλικία 45, και salary 1500 και ένα αντικείμενο s της θυγατρικής κλάσης Student, με τα γνωρίσματα: "Αγγελίδης Αντώνιος", ηλικία 20, αριθμός μητρώου 2075. Στη συνέχεια δημιουργούμε τη λίστα members, με στοιχεία τα δύο αντικείμενα t και s και εισερχόμαστε στο βρόγχο for, εντός του οποίου καλούμε τη μέθοδο tell() του κάθε αντικειμένου:

```
t = Professor('Αναστασίου Αναστάσιος', 45, 1500)
s = Student('Αγγελίδης Αντώνιος', 20, 2075)

members = [t, s]
for member in members:
    member.tell()
```

Run:

```
>>>
->Αρχικοποίηση μέλους του Πανεπιστημίου: Αναστασίου Αναστάσιος
->Αρχικοποίηση καθηγητή: Αναστασίου Αναστάσιος
->Αρχικοποίηση μέλους του Πανεπιστημίου: Αγγελίδης Αντώνιος
->Αρχικοποίηση φοιτητή: Αγγελίδης Αντώνιος
Ονοματεπώνυμο:"Αναστασίου Αναστάσιος" Ηλικία:"45" Μισθός: 1500
Ονοματεπώνυμο:"Αγγελίδης Αντώνιος" Ηλικία:"20" ΑΜ: 2075
>>>
```

**Εικόνα 8.60** *Τυπική εκτέλεση δεύτερου παραδείγματος*

## 8.16.3 Τρίτο παράδειγμα

Στο τρίτο και τελευταίο παράδειγμα αυτού του κεφαλαίου ορίζουμε μια μητρική κλάση για το αυτοκίνητο (Car), και μια θυγατρική κλάση για τα υβριδικά (οικολογικά) αυτοκίνητα (Hybrid). Στην Εικόνα 8.61 βλέπουμε ότι στη θυγατρική κλάση ορίζουμε τις γεννήτριες μεθόδους και τη μέθοδο εκτύπωσης, επαναχρησιμοποιώντας τις αντίστοιχες μητρικές μεθόδους.

Έχουμε κάποιες μεθόδους της μητρικής κλάσης Car, οι οποίες κληρονομούνται από την Hybrid: set_color (θέτουμε το χρώμα του αυτοκινήτου), get_color (βλέπουμε τι χρώμα έχει το αυτοκίνητο), set_km_per_liter (ορίζουμε χιλιόμετρα ανά λίτρο), get_km_per_liter (βλέπουμε τι χιλιόμετρα έχει ανά λίτρο ένα συγκεκριμένο αντικείμενο που το έχουμε ορίσει). Αυτά κληρονομούνται από την Hybrid, και επιπλέον ορίζουμε την set_battery_life (πόσο διαρκεί η ζωή της μπαταρίας) και get_battery_life (ποια είναι η ζωή της μπαταρίας για ένα συγκεκριμένο αυτοκίνητο).

8-inheritance_third_example.py

```
1  class Car:
2      def __init__(self):
3          self.color = 'White'
4          self.km_per_liter = 15
5      def __str__(self):
6          return 'Car:->Color:' + self.color+'->Km/l:' + str(self.km_per_liter)
7      def set_color(self,color):
8          self.color = color
9      def get_color(self):
10         return  self.color
11     def set_km_per_liter(self,kpl):
12         self.km_per_liter = kpl
13     def get_km_per_liter(self):
14         return self.km_per_liter
15
16 class Hybrid(Car):
17     def __init__(self):
18         Car.__init__(self)
19         self.battery_life=20
20     def __str__(self):
21         s=Car.__str__(self)
22         s = s+'->Battery Life:'+str(self.battery_life)
23         return s
24     def set_battery_life(self,bl=20):
25         self.battery_life=bl
26     def get_battery_life(self):
27         return self.battery_life
```

**Εικόνα 8.61** *Τρίτο παράδειγμα κληρονομικότητας*

Στην Εικόνα 8.62 έχουμε ένα κυρίως πρόγραμμα στο οποίο ορίζουμε υποστάσεις για τις κλάσεις Car και Hybrid, θέτουμε το χρώμα της υπόστασης του Hybrid, και τέλος επικαλούμαστε τις μεθόδους print των δύο κλάσεων:

```
c = Car()
h = Hybrid()
h.set_color('Red')
print(c)
print(h)
```

Εκτέλεση

```
>>>
Car:->Color:White->Km/l:15
Car:->Color:Red->Km/l:15->Battery Life:20
```

**Εικόνα 8.62** *Τυπική εκτέλεση τρίτου παραδείγματος*

## 8.17 Επίλογος

Σε αυτό το κεφάλαιο μελετήσαμε βασικές έννοιες του αντικειμενοστραφούς προγραμματισμού όπως οι κλάσεις, τα αντικείμενα, ο πολυμορφισμός, και η κληρονομικότητα. Με το κεφάλαιο αυτό κλείνει η μελέτη των βασικών χαρακτηριστικών της γλώσσας Python. Στα επόμενα κεφάλαια θα μελετήσουμε έννοιες όπως τα νήματα και ο συγχρονισμός τους, ο δικτυακός προγραμματισμός, και ο προγραμματισμός κατανεμημένων συστημάτων.

## Βιβλιογραφία/Αναφορές

Booch, G., Maksimchuk, R., Engle, M., Young, B., Conallen, J. & Houston, K. (2007). *Object-Oriented Analysis and Design with Applications (Third Edition)*. Addison-Wesley.

## Κριτήρια αξιολόγησης

### Κριτήριο αξιολόγησης I (Βαθμός δυσκολίας: ●)

**Ποια η διαφορά των μεθόδων με τις συναρτήσεις;**

**Απάντηση/Λύση**

### Κριτήριο αξιολόγησης 2 (Βαθμός δυσκολίας: ●)

**Τι είναι η μέθοδος αρχικοποίησης;**

**Απάντηση/Λύση**

### Κριτήριο αξιολόγησης 3 (Βαθμός δυσκολίας: ●)

**Τι είναι η υπερφόρτωση τελεστών και πότε είναι χρήσιμη;**

**Απάντηση/Λύση**

### Κριτήριο αξιολόγησης 4 (Βαθμός δυσκολίας: ●)

**Πότε ονομάζουμε μια συνάρτηση πολυμορφική;**

**Απάντηση/Λύση**

**Τι είναι κληρονομικότητα;**

**Απάντηση/Λύση**

# ΚΕΦΑΛΑΙΟ 9

# Ταυτόχρονος προγραμματισμός και νήματα

*Σύνοψη*

*Σε αυτό το κεφάλαιο πραγματευόμαστε τον ταυτόχρονο προγραμματισμό με τη χρήση νημάτων. Η έμφαση είναι στην κατανόηση βασικών λειτουργιών των νημάτων και του συγχρονισμού τους και στην παρουσίαση τεχνικών «καλής πρακτικής» (best practice) προγραμματισμού, οι οποίες θα φανούν ιδιαίτερα χρήσιμες στους αναγνώστες.*

*Προαπαιτούμενη γνώση*

*Η κατανόηση των εννοιών και παραδειγμάτων σε αυτό το κεφάλαιο απαιτεί βασική γνώση της γλώσσας Python, όπως παρουσιάστηκε στα Κεφάλαια 1-8 του παρόντος συγγράμματος.*

## 9.1 Εισαγωγή

Ο προγραμματισμός ταυτόχρονων διεργασιών ή νημάτων είναι σήμερα περισσότερο επίκαιρος παρά ποτέ λόγω της ανάγκης των εφαρμογών να αξιοποιήσουν τον παραλληλισμό (πολλαπλοί πυρήνες, επεξεργαστές, κάρτες δικτύου, κλπ.) σε εξυπηρετητές, προσωπικούς υπολογιστές αλλά και σε καταναλωτικές συσκευές όπως οι προσωπικές ταμπλέτες και τα κινητά τηλέφωνα. Κάθε εκτέλεση ενός προγράμματος Python αντιστοιχεί σε μια ξεχωριστή διεργασία (process), εντός της οποίας ο προγραμματιστής μπορεί να δημιουργήσει ένα ή περισσότερα νήματα (threads). Σε αυτό το σύγγραμμα θα υποθέσουμε ότι όλα τα νήματα μοιράζονται το χώρο διευθύνσεων (μεταβλητές, δομές δεδομένων, κλπ.) της διεργασίας (εφαρμογής),

χωρίς να υπεισέλθουμε σε περισσότερες λεπτομέρειες για τον τρόπο υλοποίησης των νημάτων και των διεργασιών σε επίπεδο λειτουργικού συστήματος.

Η γλώσσα προγραμματισμού Python επιτρέπει τη δημιουργία ενός νήματος σε λίγες μόνο γραμμές κώδικα, όπως φαίνεται στο παράδειγμα της Εικόνας 9.1. Σε αυτό το παράδειγμα εισάγουμε (import) το module threading, το οποίο απαιτείται και σε όλα τα άλλα προγράμματα αυτού του κεφαλαίου. Στη συνέχεια ορίζουμε τη συνάρτηση func, η οποία θα αποτελέσει το αρχικό σημείο εκτέλεσης του νήματος. Η μέθοδος δημιουργίας (constructor) του νήματος είναι η Thread, η οποία σε αυτό το παράδειγμα παίρνει ως ορίσματα τη συνάρτηση που θα εκτελέσει το νήμα και πιθανά ορίσματα σε αυτό (στη συνέχεια θα δούμε να περνάει ως όρισμα και το όνομα του νήματος σε μορφή συμβολοσειράς). Η μέθοδος *Thread* επιστρέφει ένα αντικείμενο τύπου *Thread,* το οποίο χρησιμοποιούμε στη συνέχεια για να εκτελέσουμε διάφορες λειτουργίες επί του νήματος. Η μέθοδος *start* ξεκινά την εκτέλεση του νήματος. Θα δούμε στη συνέχεια και άλλες λειτουργίες των νημάτων.

1-threadstart.py

```
1  import threading
2
3  def func(i):
4      print "Hello world, thread %d" % i
5
6
7  t = threading.Thread(target=func, args=(0,))
8  t.start()
```

**Εικόνα 9.1** *Βασική χρήση (δημιουργία, τερματισμός) ενός νήματος*

Ο ταυτόχρονος προγραμματισμός με νήματα συχνά κρύβει δυσκολίες, ιδιαίτερα στην ανάγκη αλληλεπίδρασης και συγχρονισμού μεταξύ των νημάτων. Ο συγχρονισμός είναι πολλές φορές απαραίτητος για τη διατήρηση της συνέπειας δομών δεδομένων, τις οποίες τα νήματα μοιράζονται μέσω μνήμης κοινής πρόσβασης.

Το πρόβλημα του συγχρονισμού μεταξύ ταυτόχρονων διεργασιών ή νημάτων μελετάται, τουλάχιστον, από το 1965, όταν ο Edsger Dijkstra δημοσίευσε την εργασία του με θέμα: *«Λύση σε ένα πρόβλημα ελέγχου ταυτόχρονου προγραμματισμού»* (E.

W. Dijkstra, 1965), στην οποία εισήγαγε την έννοια της *κρίσιμης περιοχής* (critical section). Ως κρίσιμη περιοχή ορίζεται μια ακολουθία εντολών, εντός της οποίας δεν μπορούν να βρίσκονται ανά πάσα στιγμή, ταυτόχρονα, περισσότερα του ενός νήματα. Το πρόβλημα της επίτευξης κρίσιμων περιοχών ονομάζεται ενίοτε και *αμοιβαίος αποκλεισμός* (mutual exclusion), δηλαδή ο συγχρονισμός μεταξύ νημάτων, ώστε η παρουσία ενός από αυτά εντός μιας κρίσιμης περιοχής να αποκλείει την παρουσία των υπολοίπων.

Έκτοτε, ο συγχρονισμός μεταξύ διεργασιών ή νημάτων έχει μελετηθεί διεξοδικά και αρκετοί μηχανισμοί διευκόλυνσης της επίλυσής του έχουν προταθεί, τυποποιηθεί και ενσωματωθεί σε επίπεδο συστήματος αλλά και γλωσσών προγραμματισμού. Η γλώσσα Python παρέχει αρκετές τέτοιες μεθόδους συγχρονισμού, οι κυριότερες από τις οποίες είναι οι παρακάτω:

- κλειδώματα (locks),
- μεταβλητές συνθήκης (condition variables),
- σημαιοφόροι (semaphores).

Στη συνέχεια του κεφαλαίου αυτού θα συζητήσουμε τα χαρακτηριστικά και την ορθή χρήση των παραπάνω μεθόδων.

## 9.2 Κλειδώματα

Τα κλειδώματα (locks) είναι ένας μηχανισμός για την επίτευξη αμοιβαίου αποκλεισμού. Η γλώσσα Python προσφέρει το μηχανισμό κλειδωμάτων μέσω του module threading, και ιδιαίτερα τις μεθόδους και μηχανισμούς που αναφέρονται αμέσως μετά:

- threading.Lock() : μέθοδος δημιουργίας lock. Μια τέτοια μέθοδος αναφέρεται συχνά σαν μέθοδος *εργοστάσιο (factory)*.

- lock.acquire(): μέθοδος με την οποία ένα νήμα προσπαθεί να αποκτήσει ένα lock. Προσέξτε ωστόσο, ότι αν το νήμα έχει, ήδη, το lock το οποίο ζητά, καλώντας την acquire μπορεί να μπλοκάρει για πάντα. Πιθανές λύσεις σε αυτό το πρόβλημα είναι οι εξής: (1) η acquire να καλεστεί με μια παράμετρο, η οποία καθορίζει αν η μέθοδος μπλοκάρει (αν δεν μπορεί να πάρει το lock (όρισμα True)) ή επιστρέφει αμέσως (όρισμα False). Στην περίπτωση που το νήμα δεν μπορεί να πάρει το lock (για οποιονδήποτε λόγο) η acquire επιστρέφει την τιμή False. (2) Να ελεγχθεί αν το νήμα, ήδη, κρατά αυτό το lock, μέσω της μεθόδου locked() του lock. (3) Να χρησιμοποιηθεί ο reentrant τύπος του lock, ο οποίος περιγράφεται παρακάτω.
- threading.RLock() ή *reentrant* lock : μέθοδος (εργοστάσιο) δημιουργίας lock με την ιδιότητα ότι ένα τέτοιου είδους lock δεν μπλοκάρει στο acquire, όταν το νήμα που το καλεί έχει, ήδη, στην κατοχή του αυτό το lock. Αν ένα τέτοιο νήμα επιτρεπόταν να μπλοκάρει, αυτό θα δημιουργούσε μια συνθήκη αδιεξόδου (deadlock), επειδή το νήμα το οποίο θα ήταν υπεύθυνο να ελευθερώσει το lock, περιμένει την απελευθέρωσή του!
- lock.release() : αποδέσμευση του lock. Ο προγραμματιστής πρέπει να είναι σίγουρος ότι κάθε λήψη ενός lock συνοδεύεται πάντα από την απελευθέρωσή του, ώστε και άλλα νήματα που το χρειάζονται να μπορούν να έχουν την ευκαιρία να το αποκτήσουν. Ένας τρόπος να επιτευχθεί αυτό είναι η έκφραση *with <lock>*, η οποία περιγράφεται παρακάτω.
- Με την έκφραση *with <lock>*, ένα νήμα εισέρχεται στο επακόλουθο κομμάτι (block) κώδικα *μόνον* αν κρατάει το lock (αν είναι ελεύθερο, το αποκτά). Μετά το πέρας της εκτέλεσης του κώδικα, το lock *ελευθερώνεται αυτόματα*.

Το παράδειγμα της Εικόνας 9.2 υποδεικνύει τη χρήση των κλειδωμάτων στην περίπτωση κοινής πρόσβασης δύο νημάτων στη μεταβλητή sharedctr, βάσει της μεθόδου *with <lock>*.

Το πρόγραμμα της Εικόνας 9.2 ξεκινά αρχικά δύο νήματα, T1 και T2, τα οποία εκτελούν τη συνάρτηση inc(). Παρατηρήστε εντός της συνάρτησης main τη χρήση της μεθόδου *join,* με την οποία το κύριο νήμα ελέγχου περιμένει την ολοκλήρωση των T1 και T2. Χωρίς τη χρήση της join το κύριο νήμα θα οδηγούνταν στην έξοδο, τερματίζοντας τη συνολική διεργασία. Σημειώστε, επίσης, τη χρήση της μεθόδου current_thread του threading, η οποία επιστρέφει μια αναφορά στο νήμα που τρέχει αυτήν τη στιγμή. Ένα γνώρισμα του αντικειμένου Thread, το οποίο χρησιμοποιείται στις εντολές print είναι το όνομα του κάθε νήματος, το οποίο δώσαμε κατά τη δημιουργία του.

Εντός της inc() τα νήματα αυξάνουν την τιμή της sharedctr κατά ένα, μέχρι να φτάσει την τιμή 5. Αν ο κώδικας ελέγχου (if sharedctr < 5) και αύξησης (sharedctr += 1) της μεταβλητής δεν ενταχθεί στην ίδια κρίσιμη περιοχή, τότε ελλοχεύει ο κίνδυνος να μπορέσουν να επιβεβαιώσουν και τα δύο νήματα ότι η sharedctr είναι μικρότερη του 5 και άρα να προσθέσουν 1 (δυο φορές), φτάνοντάς την στο 6, ενώ κανονικά δεν πρέπει να ξεπεράσει το 5. Σε αυτήν την περίπτωση το αποτέλεσμα θα ήταν ανώδυνο, ωστόσο σε άλλες περιπτώσεις μια τέτοια ασυνέπεια θα μπορούσε να έχει δυσάρεστες επιπτώσεις.

Προτρέπουμε τον αναγνώστη να επιβεβαιώσει ότι το πρόγραμμα της Εικόνας 9.2 με τη χρήση του *with <lock>* αποτρέπει αυτόν τον κίνδυνο. Αυτό που πετύχαμε είναι γνωστό και σαν *σειριοποίηση (serialization)* των νημάτων εντός της κρίσιμης περιοχής.

Σε αυτό το σημείο ανοίγουμε παρένθεση για να εξηγήσουμε δύο νέες ιδέες στην Python οι οποίες χρησιμοποιούνται στο πρόγραμμα της Εικόνας 9.2. Η πρώτη είναι η χρήση της δεσμευμένης λέξης (keyword) global, η οποία χρησιμοποιείται στη συνάρτηση inc(). Το global χρησιμοποιείται όταν μια συνάρτηση (όπως η inc()) θέλει να αναθέσει τιμή σε μεταβλητή η οποία ορίζεται έξω από αυτήν. Αν δεν είχαμε χρησιμοποιήσει το global σε αυτήν την περίπτωση, ο διερμηνέας της Python θα θεωρούσε ότι η ανάθεση sharedctr +=1 αναφέρεται σε μια νέα τοπική μεταβλητή sharedctr ορισμένη εντός της συνάρτησης inc().

Στη συνέχεια θα εξηγήσουμε τη χρήση της εντολής if __name__ == '__main__' στο τέλος του προγράμματος της Εικόνας 9.2. Ένα αρχείο πηγαίου κώδικα στην Python είναι δυνατόν να εκτελεστεί είτε δίνοντάς το ως κύρια είσοδο στο διερ-

μηνέα (κύριο πρόγραμμα), ή να χρησιμοποιηθεί εμμέσως ως module από κάποιο άλλο πρόγραμμα μέσω του import. Στην πρώτη περίπτωση θα θέλαμε εντολές που βρίσκονται στο κύριο μέρος του προγράμματος (εκτός συναρτήσεων και κλάσεων) να εκτελεστούν. Στη δεύτερη περίπτωση θέλουμε τέτοιες εντολές να μην εκτελεστούν. Η εντολή if __name__ == '__main__' πετυχαίνει ακριβώς αυτό. Όταν ο διερμηνέας της Python εκτελεί κάποιο αρχείο ως κύριο πρόγραμμα, δίνει στη μεταβλητή __name__ την τιμή '__main__', ενώ όταν το ίδιο αρχείο εισάγεται (import) ως module σε κάποιο άλλο, η ίδια μεταβλητή παίρνει ως τιμή το όνομα του αρχείου. Άρα με την εντολή if __name__ == '__main__' το κομμάτι κώδικα που εσωκλείει, εκτελείται μόνον όταν το αρχείο είναι το κύριο πρόγραμμα.

2-lock.py

```python
import threading
import time

sharedctr = 0
lock = threading.Lock()

def inc():
    global sharedctr
    while True:
        print "Thread %s trying to acquire lock..." % threading.current_thread().name
        with lock:
            if sharedctr < 5:
                sharedctr += 1
            else:
                break;
            print "Thread %s acquired lock, sharedctr = %d" % (threading.current_thread().name, sharedctr)

def thread1(p):
    print "Thread %s started" % threading.current_thread().name
    inc()

def thread2(p):
    print "Thread %s started" % threading.current_thread().name
    inc()

def main():
    t1 = threading.Thread(target=thread1, name="T1", args=(0,))
    t1.start()
    t2 = threading.Thread(target=thread2, name="T2", args=(0,))
    t2.start()
    t1.join()
    t2.join()

if __name__ == '__main__':
    main()
```

**Εικόνα 9.2** *Κλείδωμα κοινού πόρου (μεταβλητή sharedctr)*

## 9.3 Μεταβλητές συνθήκης

Οι *μεταβλητές συνθήκης* (condition variables) προτάθηκαν από τον Tony Hoare (Hoare, 1974) και στη συνέχεια τυποποιήθηκαν στα πλαίσια του προτύπου POSIX στα πλαίσια πολλών γλωσσών προγραμματισμού.

Ο λόγος ύπαρξης των μεταβλητών συνθήκης είναι ο συνδυασμός των κλειδωμάτων (locks) με την ανταλλαγή σημάτων (signals) μεταξύ νημάτων, με τρόπο ατομικό (αδιαίρετο). Για να γίνει κατανοητή αυτή η ανάγκη, ας δούμε ένα παράδειγμα: Ας υποθέσουμε ότι δύο νήματα έχουν σχέση παραγωγού (producer)-καταναλωτή (consumer), μοιραζόμενα μια κοινή δομή ουράς. Τα νήματα μπορούν να προστατεύσουν την πρόσβαση στην ουρά χρησιμοποιώντας ένα κλείδωμα m, όπως είδαμε στην προηγούμενη ενότητα. Ωστόσο, ενίοτε η ουρά αδειάζει, και έτσι εγείρεται το ερώτημα: *τι κάνει το νήμα-καταναλωτής μέχρι να μπει κάτι στην ουρά;* Άλλες φορές η ουρά γεμίζει, και αντίστοιχα εγείρεται το ερώτημα: *τι κάνει το νήμα-παραγωγός μέχρι να αδειάσει χώρος στην ουρά;* Μια απάντηση και στις δύο περιπτώσεις είναι ότι: *τα νήματα θα μπορούσαν να εισέλθουν σε ένα βρόγχο στον οποίο, συνεχώς, ελέγχουν αν ικανοποιούνται οι συνθήκες τις οποίες αναμένουν.* Αυτή η λύση είναι δαπανηρή σε πόρους CPU. Μια εναλλακτική λύση είναι τα νήματα να περιμένουν (ελευθερώνοντας την CPU) έως ότου ειδοποιηθούν από κάποιο άλλο νήμα ότι η συνθήκη που αναμένουν έχει ικανοποιηθεί. Αυτή η λύση, ωστόσο, μπορεί να οδηγήσει σε λάθος ως εξής: ένα νήμα-καταναλωτής Κ βλέπει ότι η ουρά έχει αδειάσει, ελευθερώνει το lock m, και οδεύει προς την κατάσταση αναμονής (ωστόσο, ας υποθέσουμε ότι θα καθυστερήσει να φτάσει εκεί). Στην πορεία προς την κατάσταση αναμονής, ένα νήμα-παραγωγός Π αποκτά το m, προσθέτει ένα αντικείμενο στην ουρά, και στέλνει ένα σήμα για αυτήν τη συνθήκη σε όσα νήματα είναι σε κατάσταση αναμονής. Επειδή το Κ, όμως, είναι ακόμα στο δρόμο προς αυτήν την κατάσταση, χάνει αυτό το σήμα και άρα μπορεί δυνητικά να μείνει στην ίδια κατάσταση για πάντα, ενώ υπάρχει ένα αντικείμενο στην ουρά για να καταναλωθεί. Η πηγή του προβλήματος είναι ότι το κλείδωμα και το σήμα από το ένα νήμα στο άλλο είναι διακριτές διαδικασίες, οι οποίες δεν εκτελούνται ατομικά. Οι μεταβλητές συνθήκης λύνουν, ακριβώς, αυτό το πρόβλημα.

Οι μεταβλητές συνθήκης ακολουθούν τις παρακάτω αρχές: Μια μεταβλητή συνθήκης c σχετίζεται με ένα lock m. Όταν ένα νήμα καλεί τη μέθοδο c.wait() μπαίνει σε κατάσταση αναμονής (ελευθερώνοντας την CPU) και ξεκλειδώνει το m, *σε μια ατομική δράση*. Το νήμα μπορεί να ξυπνήσει όταν κάποιο άλλο νήμα καλέσει μια από τις μεθόδους c.notify() ή c.notifyAll(). Το νήμα ξυπνά και ανακτά το κλείδωμα m, *πάλι σε μια ατομική δράση*.

Η κλήση της μεθόδου c.notify() εξετάζει εσωτερικά αν υπάρχει, τουλάχιστον, ένα νήμα που περιμένει στο c (δηλαδή, έχει καλέσει c.wait() στο παρελθόν). Αν υπάρχει ένα τέτοιο νήμα, αυτό γίνεται εκτελέσιμο και του επιτρέπεται να ανακτήσει τον έλεγχο. Η μέθοδος c.notifyAll() δρα όπως και η c.notify(), αλλά ξυπνά περισσότερα του ενός νήματα τα οποία μπορεί να περιμένουν στο c. Η c.notifyAll(), λοιπόν, εξετάζει το c, και αν υπάρχει τουλάχιστον ένα νήμα το οποίο περιμένει, τότε όλα αυτά τα νήματα γίνονται εκτελέσιμα και τους επιτρέπεται να ανακτήσουν τον έλεγχο. Αυτή η λειτουργία συμβαίνει σαν μια ατομική δράση: τα νήματα που εγείρονται είναι, ακριβώς, αυτά που είχαν καλέσει c.wait() πριν από αυτήν την κλήση της c.notifyAll· τα νήματα αυτά πρέπει να περιμένουν στη σειρά για να αποκτήσουν το m.

Το παράδειγμα της Εικόνας 9.3 υποδεικνύει τη χρήση των μεταβλητών συνθήκης (condition variables) για το συγχρονισμό παραγωγού (producer)-καταναλωτή (consumer) πάνω από μια κοινή δομή λίστας με δύο άκρες (double-ended queue ή *deque*). Η ιδιαίτερη αυτή μορφή λίστας (deque) προσφέρει λειτουργίες append («πρόσθεσε στην ουρά») και pop («αφαίρεσε από την ουρά») που εφαρμόζονται και στις δύο άκρες της λίστας (οι μέθοδοι που εφαρμόζονται δεξιά της λίστας είναι οι append() και pop() και αυτές που εφαρμόζονται στα αριστερά της είναι οι appendleft() και popleft()). Το module dequeue περιέχεται στη βιβλιοθήκη collections.

3-cv.py

```
import threading
import time
from collections import deque

lock = threading.Lock()
cond = threading.Condition(lock)

queue = deque(["1", "2", "3"])

def producer():
    while True:
        cond.acquire()
        # produce here
        s = "something"
        queue.append(s)
        print "produced %s" % s
        cond.notify() # or notifyAll to wake everyone up
        cond.release()
        time.sleep(1)

def consumer():
    while True:
        cond.acquire()
        # consume here
        s = queue.popleft()
        print "consumed %s" % s
        cond.wait() # sleep until item becomes available
        cond.release()
        time.sleep(1)

def thread1(p):
    print "Thread %s started" % threading.current_thread().name
    producer()

def thread2(p):
    print "Thread %s started" % threading.current_thread().name
    consumer()

def main():
    t1 = threading.Thread(target=thread1, name="T1", args=(0,))
    t1.start()
    t2 = threading.Thread(target=thread2, name="T2", args=(0,))
    t2.start()
    t1.join()
    t2.join()

if __name__ == '__main__':
    main()
```

**Εικόνα 9.3** *Χρήση μεταβλητής συνθήκης*

## 9.4 Σημαιοφόροι

Οι *σημαιοφόροι* (semaphores) προτάθηκαν από τον Edsger Dijkstra (E.W. Dijkstra, 2002), σαν μια γενική λύση σε προβλήματα συγχρονισμού, μια λύση ισοδύναμη (η οποία, επομένως, μπορεί να υποκαταστήσει) τα κλειδώματα και τις μεταβλητές συνθήκης. Σε αυτό το σύγγραμμα δε θα εξετάσουμε τις λεπτομέρειες των σχέσεων μεταξύ μηχανισμών συγχρονισμού· περισσότερες πληροφορίες μπορούν να αναζητηθούν σε κάποιο προχωρημένο σύγγραμμα λειτουργικών συστημάτων.

Οι σημαιοφόροι είναι αντικείμενα συγχρονισμού τα οποία εμπεριέχουν μια ακέραια μεταβλητή (την ονομάζουμε N) και προσφέρουν δύο μεθόδους: τις acquire() και release(). Ένας σημαιοφόρος μπορεί να δημιουργηθεί μέσω της μεθόδου εργοστασίου threading.BoundedSemaphore(*N*), όπου *N* η αρχική τιμή της ακέραιας μεταβλητής. Η σημασία των μεθόδων acquire() και release() αναλύεται ως εξής:

- Η acquire() όταν καλείται χωρίς όρισμα ή με όρισμα True, ελέγχει ατομικά αν η τρέχουσα τιμή της ακέραιας μεταβλητής N είναι μεγαλύτερη του μηδενός, αν ισχύει αυτό, την μειώνει κατά ένα και επιστρέφει με την τιμή True. Αν η τρέχουσα τιμή του N είναι μηδέν ή αρνητική, η acquire την μειώνει κατά ένα και μπλοκάρει. Αν επιπρόσθετα το όρισμά της acquire είναι False (δηλαδή: «μην μπλοκάρεις») τότε η acquire επιστρέφει, ανεπιτυχώς, με την τιμή False.
- Η release() αυξάνει το N κατά ένα. Αν το N αρχικά ήταν αρνητικό, η release() ξυπνά ένα από τα νήματα που περίμεναν. Η σειρά με την οποία ξυπνά τα νήματα είναι τυχαία, και όχι απαραίτητα η σειρά με την οποία τα νήματα είχαν μπλοκάρει.

Όταν η εσωτερική ακέραια μεταβλητή του σημαιοφόρου έχει αρνητική τιμή, ο απόλυτος αριθμός υποδεικνύει τον αριθμό των νημάτων τα οποία περιμένουν.

Στην Εικόνα 9.4 βλέπουμε μια εναλλακτική υλοποίηση του προγράμματος της Εικόνας 9.2, η οποία χρησιμοποιεί σημαιοφόρο αντί για κλείδωμα. Προσέξτε ότι η μεταβλητή semaphore αρχικοποιείται στην εσωτερική τιμή 1. Αν θέλαμε να επι-

τρέψουμε την είσοδο περισσοτέρων νημάτων, για παράδειγμα Μ στον αριθμό, ταυτόχρονα στην κρίσιμη περιοχή, τότε θα θέταμε την εσωτερική μεταβλητή του σημαιοφόρου στην τιμή Μ.

4-semaphore.py

```
import threading
import time

sharedctr = 0
lock = threading.Lock()
semaphore = threading.BoundedSemaphore(1)

def inc():
    global sharedctr
    while True:
        print "Thread %s trying to acquire semaphore...\n" % threading.current_thread().name
        semaphore.acquire()
        print "Thread %s acquired semaphore, sharedctr = %d" % (threading.current_thread().name, sharedctr)
        try:
            if sharedctr < 5:
                sharedctr += 1
            else:
                # finally: clause will release semaphore
                break;
        finally:
            semaphore.release()

def thread1(p):
    print "Thread %s started" % threading.current_thread().name
    inc()

def thread2(p):
    print "Thread %s started" % threading.current_thread().name
    inc()

def main():
    t1 = threading.Thread(target=thread1, name="T1", args=(0,))
    t1.start()
    t2 = threading.Thread(target=thread2, name="T2", args=(0,))
    t2.start()
    t1.join()
    t2.join()

if __name__ == '__main__':
    main()
```

**Εικόνα 9.4** *Χρήση σημαιοφόρου*

## 9.5 Επίλογος

Σε αυτό το κεφάλαιο μελετήσαμε βασικές λειτουργίες των νημάτων και τρεις σημαντικούς μηχανισμούς συγχρονισμού τους: τα κλειδώματα, τις μεταβλητές συνθήκης, και τους σημαιοφόρους. Με αυτές τις γνώσεις ο προγραμματιστής της Python μπορεί να υλοποιήσει αμοιβαίο αποκλεισμό μεταξύ νημάτων για πρόσβαση σε κρίσιμες περιοχές, προστατεύοντας τη συνέπεια δομών δεδομένων κοινής πρόσβασης.

## Βιβλιογραφία/Αναφορές


Dijkstra, E. W. (1965). Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9), 569.

Dijkstra, E. W. (2002). Cooperating sequential processes. In *The origin of concurrent programming* (pp. 65 - 138). New York, NY: Springer-Verlag Inc.

Hoare, C. A. R. (1974). Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10), 549-557.

### Κριτήριο αξιολόγησης Ι (Βαθμός δυσκολίας: ●●)

**Εξηγήστε γιατί στο πρόγραμμα της Εικόνας 9.2 η χρήση του with <lock> αποτρέπει την πιθανότητα η κοινή μεταβλητή sharedctr να υπερβεί τον αριθμό 5.**

**Απάντηση/Λύση**

### Κριτήριο αξιολόγησης 2 (Βαθμός δυσκολίας: ●●)

**Μετασχηματίστε το πρόγραμμα της Εικόνας 9.2, ώστε να χρησιμοποιεί τις μεθόδους lock.acquire και lock.release αντί του with <lock>.**

**Απάντηση/Λύση**

### Κριτήριο αξιολόγησης 3 (Βαθμός δυσκολίας: ●●)

**Μετασχηματίστε το πρόγραμμα της Εικόνας 9.3, για να χρησιμοποιεί σημαιοφόρο αντί μεταβλητής συνθήκης.**

**Απάντηση/Λύση**

# ΚΕΦΑΛΑΙΟ 10

# Δικτυακός προγραμματισμός

*Σύνοψη*

*Σε αυτό το κεφάλαιο πραγματευόμαστε το δικτυακό προγραμματισμό μέσω της αφαίρεσης των sockets.*

*Προαπαιτούμενη γνώση*

*Η κατανόηση των εννοιών και παραδειγμάτων σε αυτό το κεφάλαιο απαιτεί βασική γνώση της γλώσσας Python, όπως παρουσιάστηκε στα Κεφάλαια 1-8 του παρόντος συγγράμματος.*

## 10.1 Εισαγωγή

Ο δικτυακός προγραμματισμός στηρίζεται στην αφαίρεση των sockets, η οποία επινοήθηκε στα πλαίσια του 4.2BSD Unix operating system, στις αρχές της δεκαετίας του `80 (McKusick, Bostic, Karels, & Quarterman, 1996). Η διεπαφή εφαρμογής-χρήστη (application-programmer interface ή API) των sockets τυποποιήθηκε μεταγενέστερα, εντός του προτύπου POSIX. Ένα socket ορίζει μια «θύρα» μέσω της οποίας μια διεργασία επικοινωνεί με μια άλλη (μέσω της αντίστοιχης «θύρας» σε εκείνη τη διεργασία) πάνω από ένα δίκτυο (Kurose & Ross, 2005). Σε αυτό το σύγγραμμα θα ασχοληθούμε με τρία είδη επικοινωνίας:

I. Επικοινωνία «ένας-προς-έναν» μεταξύ δύο sockets σε σύνδεση: Τα δύο sockets είναι σε σύνδεση από τη στιγμή που η σύνδεση δημιουργείται μέχρι τη στιγμή που τερματίζεται. Αυτού του είδους η επικοινωνία υλοποιείται με το πρωτόκολ-

λο TCP και κληρονομεί τις ιδιότητές του: αξιόπιστη ροή bytes (δε διατηρεί όρια μεταξύ μηνυμάτων) και εν σειρά.

2. Επικοινωνία «ένας-προς-έναν» μεταξύ δύο sockets χωρίς σύνδεση μεταξύ τους: Ένα socket μπορεί να στείλει και λάβει δεδομένα από οποιοδήποτε άλλο socket. Αυτού του είδους η επικοινωνία υλοποιείται από το πρωτόκολλο UDP και κληρονομεί τις ιδιότητές του: αναξιόπιστη, βάσει μηνυμάτων, χωρίς να εγγυάται ότι τα μηνύματα παραδίδονται με τη σειρά με την οποία στέλνονται.

3. Επικοινωνία «ένας-προς-πολλούς» μέσα σε μια ομάδα sockets: Ένα socket μπορεί να στείλει δεδομένα προς μια ομάδα από sockets, και να λάβει δεδομένα από οποιοδήποτε άλλο socket στην ομάδα. Αυτού του είδους η επικοινωνία υλοποιείται από το πρωτόκολλο IP multicast. Σε προγραμματιστικό επίπεδο προσομοιάζει την επικοινωνία τύπου (2) –επικοινωνία χωρίς σύνδεση- με πρόσθετες διεπαφές για τη διαχείριση της ομάδας.

## I0.2 Το module socket

Το module socket παρέχει συναρτήσεις για τη βασική επικοινωνία μεταξύ δύο διεργασιών. Στην έναρξη μιας σύνδεσης οι δύο διεργασίες έχουν τους διακριτούς ρόλους του πελάτη (client) και του διακομιστή (server), με βάση το ποια ξεκινά ενεργά τη σύνδεση και ποια την αποδέχεται. Μετά το στήσιμο της σύνδεσης, το ρόλο του πελάτη έχει αυτός ο οποίος στέλνει αιτήσεις, ενώ ο διακομιστής είναι αυτός που επεξεργάζεται τις αιτήσεις στέλνοντας πίσω απαντήσεις σε αυτές (ωστόσο, οι διεργασίες μπορεί να αλλάζουν ρόλους κατά περίπτωση).

Η χρήση του API των sockets τυπικά ακολουθεί την παρακάτω ροή: Οι δύο πλευρές δημιουργούν (η καθεμία από την πλευρά της) ένα αντικείμενο τύπου socket s. Ο διακομιστής προετοιμάζεται να δεχθεί συνδέσεις, καλώντας s.bind (συσχετίζοντας το socket του με κάποια πόρτα και προαιρετικά με κάποια IP διεύθυνση) και s.listen. Ο πελάτης στη συνέχεια προσπαθεί να συνδέσει το τοπικό socket με το απομακρυσμένο (σε διεργασία στο ίδιο ή άλλο μηχάνημα) καλώντας s.connect. Μια εισερχόμενη αίτηση για σύνδεση γίνεται δεκτή και ένα νέο socket δημιουργείται, όταν ο διακομιστής καλέσει s.accept. Σημειώστε ότι το αρχικό socket του

διακομιστή (υπό μια έννοια το «μητρικό» του συνδεμένου socket) συνεχίζει να είναι σε κατάσταση αναμονής και ακρόασης για νέες συνδέσεις και δε συνδέεται το ίδιο ποτέ με άλλα sockets. Όταν ένα socket συνδεθεί, μπορεί να χρησιμοποιηθεί για αποστολή και παραλαβή δεδομένων μέσω των συναρτήσεων **send** και **recv** αντίστοιχα. Η επικοινωνία πάνω από σύνδεση τερματίζεται με κλήση της συνάρτησης **close** και από τις δύο διεργασίες. Ο σκοπός αυτού του κεφαλαίου είναι να εισαγάγει τον αναγνώστη στη χρήση ενός βασικού συνόλου κλήσεων μέσω παραδειγμάτων.

## I0.3 Πελάτης και διακομιστής Echo με TCP sockets

Σε αυτήν την ενότητα θα μελετήσουμε το παράδειγμα ενός απλού πελάτη ο οποίος στέλνει ένα string και ενός διακομιστή ο οποίος το επιστρέφει αυτούσιο (υλοποιώντας την «ηχώ» ή echo του) πάνω από μια σύνδεση TCP. Η Εικόνα I0.1 υποδεικνύει το διακομιστή που περιμένει («ακούει») για συνδέσεις στην πόρτα 8888. Η επιλογή του TCP γίνεται μέσω της σταθεράς `SOCK_STREAM` στην κλήση δημιουργίας του socket. Όταν συνδεθεί το socket, ο διακομιστής μπαίνει σε ατέρμονο βρόχο επιστρέφοντας ό,τι λαμβάνει πίσω στον πελάτη, τερματίζοντας τη σύνδεση από την πλευρά του, όταν και ο πελάτης τερματίσει τη σύνδεση από τη δική του (η γραμμή `if not receivedData: break`, ικανοποιείται όταν κλείσιμο της σύνδεσης από τον πελάτη επιστρέφει null (end of stream) αποτέλεσμα στην κλήση του `recv`). Το όρισμα στο newSocket.recv είναι 8192, το οποίο είναι το πάνω όριο στο πόσα δεδομένα θα ληφθούν στην κλήση, ωστόσο η κλήση μπορεί να επιστρέψει με λιγότερα δεδομένα. Ο διακομιστής επιστρέφει για να δεχθεί την επόμενη σύνδεση κ.ο.κ. έως ότου ο πελάτης διακόψει τη ροή μέσω Ctrl-C από την κονσόλα.

```python
import socket
import time

# Create socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.bind(('', 8888))
sock.listen(5)

try:
    # Echo loop
    while True:
        try:
            newSocket, address = sock.accept()
        except socket.error, (value, message):
            print "Socket error while waiting to accept socket: " + message
        except:
            print "Unexpected error while waiting to accept"
            break
        print "Connected from echo client ", address
        while True:
            try:
                receivedData = newSocket.recv(1024)
            except socket.error as (value, message):
                # Client may reset connection (terminate
                # abnormally) while server waits to
                # receive next echo line. Print message,
                # then go up to accept
                print "Socket error while waiting to receive: (value:", value, ") " + message
                break
            except:
                print "Unexpected error while waiting to receive"
                break

            if not receivedData: break

            newSocket.sendall(receivedData)

        # Done transacting with client
        newSocket.close()
        print "Disconnected from", address

except:
    print "Unexpected error in echo loop"

finally:
    sock.close()
```

**Εικόνα 10.1** *Διακομιστής TCP echo*

Το παράδειγμα της Εικόνας 10.2 υποδεικνύει έναν απλό TCP πελάτη που συνδέεται στην πόρτα 8888 στο μηχάνημα με IP διεύθυνση 192.168.168.11 (θα μπορούσε να βρίσκεται και στο τοπικό μηχάνημα ή localhost), στέλνει μια ακολουθία από γραμμές δεδομένων (η κάθε γραμμή τερματίζεται με \n), και τυπώνει ό,τι λαμβάνει πίσω από το διακομιστή.

```python
import time
import socket

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(('192.168.168.11', 8888))
print "Connected to echo server"

echo = """testing\n1\n2\n3"""

try:
    for line in echo.splitlines():
        sock.sendall(line)
        print "Sent:", line
        try:
            # Uncomment delay to crash client before receiving server response
            #time.sleep(5) # delay here
            response = sock.recv(1024)
        except socket.error as (value, message):
            # Server may reset connection (terminate
            # abnormally) while client waits to
            # receive next echo line. Print message,
            # then go up to send next line (if any)
            print "Socket error while waiting to receive: (value:", value, ") " + message
        except:
            print "Unexpected error"
            break
        print "Received:", response

        if not response: break

except:
    print "Unexpected error in echo loop"

sock.close()
```

**Εικόνα Ι0.2** *Πελάτης TCP echo*

Στο εργαστήριο ανοίγετε δύο παράθυρα, ένα στο τοπικό μηχάνημα και ένα σε κάποιο απομακρυσμένο (π.χ. το γειτονικό μηχάνημα στο υπολογιστικό περιβάλλον ενός εργαστηρίου). Στο ένα από τα δύο τερματικά εκτελέστε το διακομιστή της Εικόνας Ι0.Ι. Η πόρτα που θα χρησιμοποιήσουμε για το διακομιστή είναι η 8888 για τις ανάγκες αυτού του παραδείγματος, ωστόσο σε εργαστηριακό περιβάλλον υποθέτουμε ότι ο κάθε φοιτητής θέτει την πόρτα σε κάποιο προσωπικό αριθμό όπως π.χ. ο Αριθμός Μητρώου του. Για να ανακαλύψουμε τις IP διευθύνσεις των σταθμών εργασίας σε συστήματα Unix/Linux εκτελούμε την εντολή ifconfig (η οποία βρίσκεται, συνήθως, στο directory /sbin). Στο δεύτερο παράθυρο δοκιμάστε μερικές εκτελέσεις του παραδείγματος της Εικόνας Ι0.2 προς το διακομιστή.

Πολλές φορές είναι χρήσιμο να μπορούμε να έχουμε μια άποψη της επικοινωνίας στο χαμηλότερο επίπεδο, όπως για παράδειγμα της ανταλλαγής TCP μηνυμάτων που υλοποιούν τη διεπαφή socket (Fall & Stevens, 1994). Για αυτόν το σκοπό μπορούμε να χρησιμοποιήσουμε ένα εργαλείο επισκόπησης της δικτυακής επικοινωνίας, όπως το wireshark (Wireshark, 2015). Σ' αυτό το σύγγραμμα θα υποθέσουμε ότι ο αναγνώστης έχει εισαγωγική γνώστη της χρήσης του wireshark και θα εστι-

άσουμε στην εφαρμογή του στην επικοινωνία, όταν χρησιμοποιούμε τα παραδείγματα των Εικόνων I0.1 και I0.2. Ξεκινούμε θέτοντας τις παρακάτω ρυθμίσεις:

- Φίλτρο tcp,
- Name resolution for transport layer OFF,
- Packet bytes OFF,
- Colorize packet list OFF.

Το output του wireshark σε μια τυπική εκτέλεση των παραδειγμάτων των Εικόνων I0.1 και I0.2, φαίνεται στην Εικόνα I0.3. Αρχικά παρατηρούμε την «τριπλή χειραψία» (3-way handshake) του TCP και τις παραμέτρους της διαπραγμάτευσης μεταξύ των δύο πλευρών (π.χ. το μέγιστο μήκος πακέτου ή maximum segment size (MSS) I460 bytes, κλπ.). Στη συνέχεια παρατηρούμε την ανταλλαγή μηνυμάτων μεταφοράς δεδομένων της εφαρμογής και επιβεβαιώσεων (ACK), και στο τέλος έχουμε το κλείσιμο των συνδέσεων (FIN) και από τις δύο πλευρές.

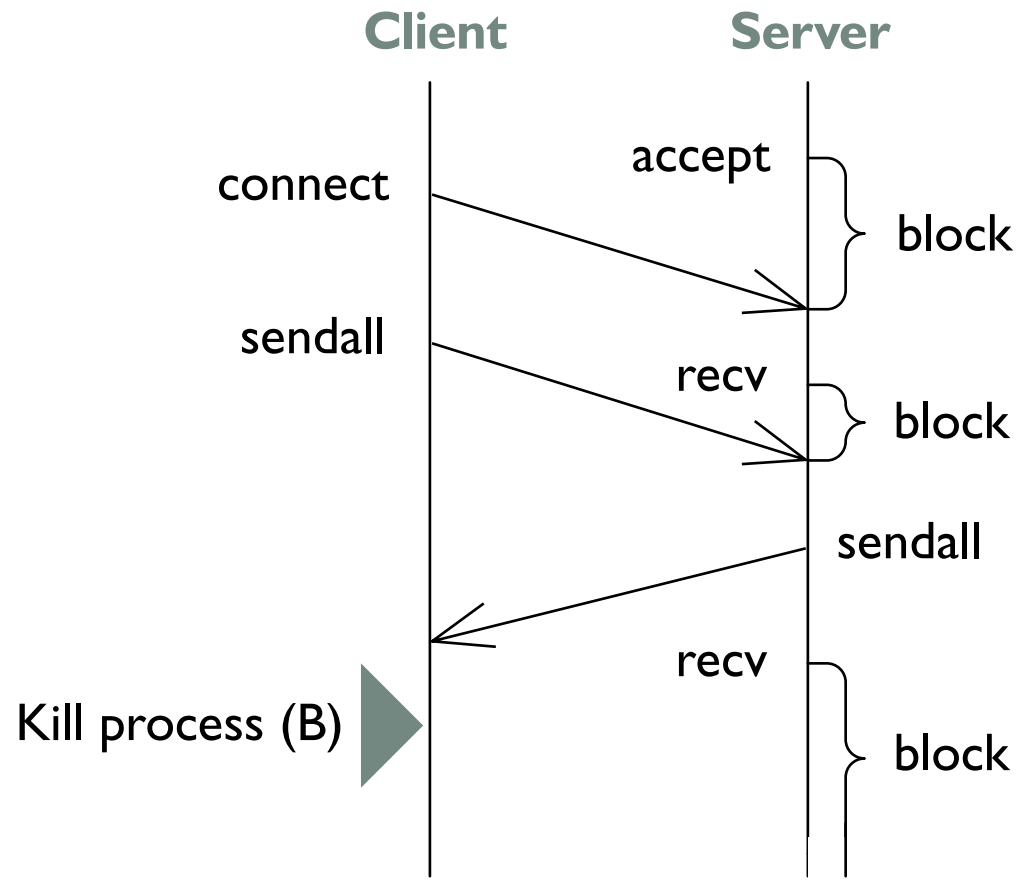
| | | | |
|---|---|---|---|
| 192.168.168.118 | 192.168.168.11 | TCP | 66 49563→8888 [SYN] Seq=0 Win=65535 Len= |
| 192.168.168.11 | 192.168.168.118 | TCP | 66 8888→49563 [SYN, ACK] Seq=0 Ack=1 Win |
| 192.168.168.118 | 192.168.168.11 | TCP | 54 49563→8888 [ACK] Seq=1 Ack=1 Win=2129 |
| 192.168.168.118 | 192.168.168.11 | TCP | 61 49563→8888 [PSH, ACK] Seq=1 Ack=1 Win |
| 192.168.168.11 | 192.168.168.118 | TCP | 60 8888→49563 [ACK] Seq=1 Ack=8 Win=2931 |
| 192.168.168.11 | 192.168.168.118 | TCP | 61 8888→49563 [PSH, ACK] Seq=1 Ack=8 Win |
| 192.168.168.118 | 192.168.168.11 | TCP | 55 49563→8888 [PSH, ACK] Seq=8 Ack=8 Win |
| 192.168.168.11 | 192.168.168.118 | TCP | 60 8888→49563 [ACK] Seq=8 Ack=9 Win=2931 |
| 192.168.168.11 | 192.168.168.118 | TCP | 60 8888→49563 [PSH, ACK] Seq=8 Ack=9 Win |
| 192.168.168.118 | 192.168.168.11 | TCP | 55 49563→8888 [PSH, ACK] Seq=9 Ack=9 Win |
| 192.168.168.11 | 192.168.168.118 | TCP | 60 8888→49563 [ACK] Seq=9 Ack=10 Win=293 |
| 192.168.168.11 | 192.168.168.118 | TCP | 60 8888→49563 [PSH, ACK] Seq=9 Ack=10 Wi |
| 192.168.168.118 | 192.168.168.11 | TCP | 55 49563→8888 [PSH, ACK] Seq=10 Ack=10 W |
| 192.168.168.11 | 192.168.168.118 | TCP | 60 8888→49563 [ACK] Seq=10 Ack=11 Win=29 |
| 192.168.168.11 | 192.168.168.118 | TCP | 60 8888→49563 [PSH, ACK] Seq=10 Ack=11 W |
| 192.168.168.118 | 192.168.168.11 | TCP | 54 49563→8888 [FIN, ACK] Seq=11 Ack=11 W |
| 192.168.168.11 | 192.168.168.118 | TCP | 60 8888→49563 [FIN, ACK] Seq=11 Ack=12 W |
| 192.168.168.118 | 192.168.168.11 | TCP | 54 49563→8888 [ACK] Seq=12 Ack=12 Win=21 |

**Εικόνα I0.3** *Ανταλλαγή TCP πακέτων κατά τη διάρκεια μιας τυπικής επιτυχούς εκτέλεσης*

Αν η μία από τις δύο πλευρές τερματίσει (π.χ. με ctrl-C σε κάποια χρονική στιγμή) την εκτέλεση του client ή του server, τις περισσότερες φορές αυτό οδηγεί σε «καθαρό» τερματισμό και της άλλης πλευράς (π.χ. διαβάζοντας 0 bytes σε κάποιο recv και οδηγώντας τη ροή εκτέλεσης σε κάποιο close).

Εξαιρέσεις μπορούν να δημιουργηθούν υπό ειδικές συνθήκες, όπως το κλείσιμο της σύνδεσης, όταν δεν έχουν ακόμα διαβαστεί όλα τα εισερχόμενα bytes σε αυτήν την σύνδεση. Για να πειραματιστούμε με αυτήν την περίπτωση, κλείνουμε τη σύνδεση (ctrl-C) στον client, ενώ:

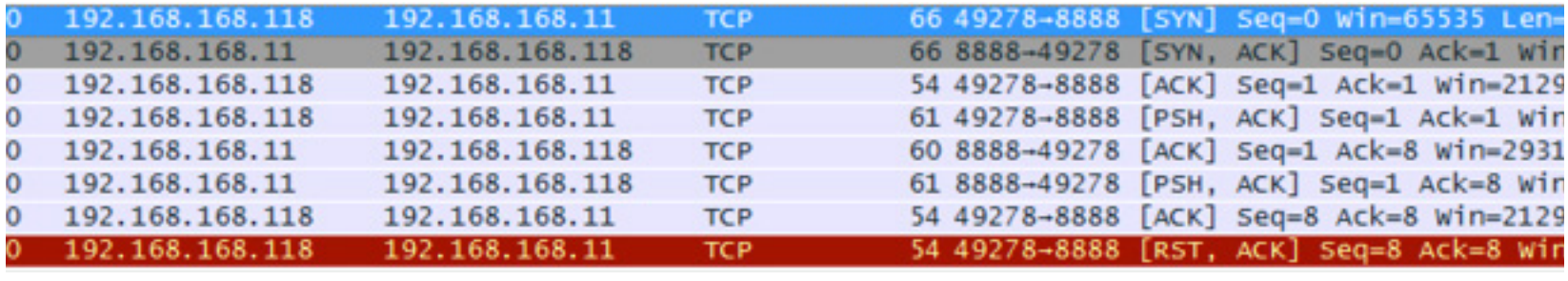
- Ο client έχει στείλει δεδομένα, ο server τα έχει λάβει και έχει στείλει την απάντησή του.
- Ο client δεν έχει διαβάσει ακόμα την απάντηση (υπάρχουν bytes στο socket buffer του client).
- Ο server μπλοκάρει στο recv περιμένοντας την επόμενη αίτηση από τον client.




**Εικόνα 10.4** *Περίπτωση παραγωγής εξαίρεσης. Η κατάρρευση του client παράγει reset, όταν υπάρχουν bytes προερχόμενα από τον server στο socket buffer, διαφορετικά κλείνει καθαρά η σύνδεση*

Ο server παίρνει εξαίρεση "Connection reset by peer" (104) βγάζοντάς τον από το recv και τερματίζοντας το πρόγραμμα. Στο παρακάτω wireshark dump παρατηρούμε (α) το στήσιμο της πρώτης σύνδεσης, (β) την αποστολή της πρώτης αίτησης (7 bytes) και το ACK του server, (γ) την απάντηση του server και το ACK του client, (δ) το RST πακέτο από τον client προς τον server, το οποίο μεταφέρει την

πληροφορία ότι τα τελευταία bytes από τον server προς τον client δε θα διαβαστούν ποτέ (αντικαθιστά το FIN που κανονικά θα έστελνε ο client). Το RST εξαφανίζεται και αντικαθίσταται από ένα καθαρό κλείσιμο της σύνδεσης αν το Ctrl-C συμβεί, αφού διαβαστούν τα δεδομένα στον client.




**Εικόνα 10.5** *Ανταλλαγή TCP πακέτων κατά τη διάρκεια μιας ανεπιτυχούς εκτέλεσης (ctrl-C στον client)*

## I0.4 Πελάτης και διακομιστής Echo με UDP sockets

Τα παραδείγματα των Εικόνων I0.6 και I0.7 υποδεικνύουν έναν πελάτη και διακομιστή echo που χρησιμοποιούν UDP (datagram) αντί για TCP (stream) sockets.

```
1  import socket
2  sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
3  sock.bind(('', 0000))
4
5  # loop waiting for datagrams
6  try:
7      while True:
8          data, address = sock.recvfrom(8192)
9          print "Datagram from", address
10         sock.sendto(data, address)
11 finally:
12     sock.close( )
13
```

**Εικόνα I0.6** *Διακομιστής echo με UDP*

```
1  import socket
2  sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
3  data = """Test1\n2\n3."""
4  for line in data.splitlines( ):
5      sock.sendto(line, ('localhost', 8888))
6      print "Sent:", line
7      response = sock.recv(1024)
8      print "Received:", response
9  sock.close( )
```

**Εικόνα I0.7** *Πελάτης echo με UDP*

Δοκιμάστε μερικές εκτελέσεις των παραδειγμάτων των Εικόνων I0.6 και I0.7 σε διαφορετικά τερματικά, όπως κάναμε και στην περίπτωση του TCP. Σημειώστε μια σημαντική διαφορά στα χαρακτηριστικά αξιοπιστίας μεταξύ του UDP και του TCP· σε περίπτωση που το datagram το οποίο στέλνει ο πελάτης ή αυτό που επιστρέφει ο διακομιστής χαθούν, ο πελάτης δε λαμβάνει καμία ειδοποίηση.

Τα UDP sockets έχουν μια συνάφεια από άποψη χαρακτηριστικών αξιοπιστίας και διεπαφής εφαρμογής (API) με την *πολυεκπομπή IP* (multicast), η οποία είναι ένας μηχανισμός επικοινωνίας μεταξύ ενός αποστολέα και περισσοτέρων του ενός παραληπτών, με χρήση του πρωτοκόλλου IP. Η πολυεκπομπή, γενικότερα, είναι μια μορφή *ομαδικής επικοινωνίας* κατά την οποία κάθε μήνυμα που στέλνεται από τον αποστολέα, λαμβάνεται από όλα τα μέλη της ομάδας των παραληπτών. Η ομάδα των παραληπτών ταυτοποιείται μέσω ειδικών IP διευθύνσεων μεταξύ 220.0.0.0 και 239.255.255.255 (220.0.0.0/4). Μορφές πολυεκπομπής που προσφέρουν πρόσθετα χαρακτηριστικά όπως αξιοπιστία, σειριοποίηση μηνυμάτων (από έναν ή διαφορετικούς αποστολείς) κλπ. αποτελούν αντικείμενο μελέτης στην περιοχή των κατανεμημένων συστημάτων και μέρος της ύλης συγγραμμάτων, όπως το (Coulouris, Dollimore, & Kindberg, 2005).

Τα Παραδείγματα των Εικόνων I0.8 και I0.9 υποδεικνύουν την προγραμματιστική χρήση της πολυεκπομπής IP μέσω της διεπαφής εφαρμογής sockets.

```python
import socket
import struct

MCAST_GRP = '224.1.1.1'
MCAST_PORT = 5007

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, socket.IPPROTO_UDP)
sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

sock.bind(('', MCAST_PORT))

mreq = struct.pack("4sl", socket.inet_aton(MCAST_GRP), socket.INADDR_ANY)

sock.setsockopt(socket.IPPROTO_IP, socket.IP_ADD_MEMBERSHIP, mreq)

while True:
  print sock.recv(10240)

```

**Εικόνα I0.8** *Παραλήπτης πολυεκπομπής UDP/IP*

```python
import socket

MCAST_GRP = '224.1.1.1'
MCAST_PORT = 5007

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, socket.IPPROTO_UDP)
sock.setsockopt(socket.IPPROTO_IP, socket.IP_MULTICAST_TTL, 2)
sock.sendto("robot", (MCAST_GRP, MCAST_PORT))

```

**Εικόνα Ι0.9** *Αποστολέας πολυεκπομπής UDP/IP*

Ένα πρόβλημα που, συνήθως, αντιμετωπίζουν διακομιστές οι οποίοι χειρίζονται, ταυτόχρονα, ένα μεγάλο αριθμό συνδέσεων, είναι το γεγονός ότι πολλές κλήσεις όπως accept, recv, send κλπ. ενδέχεται να μπλοκάρουν αν δεν είναι άμεσα ικανοποιήσιμες οι συνθήκες ή διαθέσιμοι οι πόροι που απαιτούνται για την επιτυχή λειτουργία τους· κατά αυτόν τον τρόπο εμποδίζεται και η συνολική πρόοδος του συστήματος, αν ο διακομιστής χρησιμοποιεί ένα και μοναδικό νήμα.

Μια τέτοια κατάσταση μπορεί να αποφευχθεί με χρήση πολλαπλών νημάτων, ένα ανά socket, όπως περιγράφεται στο Κεφάλαιο 9. Παρά το πλεονέκτημα της απλότητας μιας τέτοιας προσέγγισης, καθώς αυξάνονται οι συνδέσεις, ανακαλύπτουμε ένα όριο στην κλιμακωσιμότητα, λόγω του αυξανόμενου κόστους διαχείρισης μεγάλου αριθμού νημάτων. Μια άλλη λύση είναι να θέτουμε τα sockets σε κατάσταση μη-μπλοκαρίσματος (χρησιμοποιώντας την κλήση setblocking()), ωστόσο αυτό προϋποθέτει ότι ο μικρός αριθμός νημάτων αναλώνεται στο να ελέγχει, συνεχώς, πότε οι συνθήκες θα επιτρέπουν την επιτυχή κλήση των συναρτήσεων. Ένας τρόπος να αποφύγουμε αυτό το κόστος είναι η χρήση της συνάρτησης *select*, η οποία επιτρέπει να προσαρμόσουμε τη ροή ελέγχου του προγράμματος στην εξέλιξη των γεγονότων δραστηριότητας των sockets.

## Ι0.5 Προγραμματισμός sockets με ροή ελέγχου οδηγούμενη από τα γεγονότα

Η μέθοδος προγραμματισμού συστημάτων, σύμφωνα με την οποία η ροή ελέγχου ορίζεται από τα γεγονότα, ακολουθεί τη χρονική τους εξέλιξη, και η οποία εστιάζει

στο χειρισμό τους όταν αυτά συμβούν, έχει μακρά ιστορία. Ο προγραμματισμός συστημάτων σε χαμηλό επίπεδο (π.χ. στο όριο υλικού-λογισμικού) ακολουθεί παραδοσιακά αυτήν την πρακτική. Ο προγραμματισμός με ροή ελέγχου οδηγούμενη απο τα γεγονότα σε υψηλότερο επίπεδο έχει γνωρίσει επιτυχία σε αρκετές περιπτώσεις (Ousterhout, 1996).

Ένα event-driven πρόγραμμα αποτελείται από έναν βρόχο γεγονότων (event loop) όπου το νήμα ελέγχου αναμένει ειδοποιήσεις για γεγονότα και τα χειρίζεται όταν συμβούν. Στο δικτυακό προγραμματισμό, τυπικά γεγονότα είναι *«ένας πελάτης επιθυμεί τη δημιουργία σύνδεσης»*, *«δεδομένα είναι διαθέσιμα για ανάγνωση σε κάποιο socket»*, ή *«ένα socket έχει τώρα διαθέσιμους πόρους για την αποστολή δεδομένων»*. Το πρόγραμμα χειρίζεται κάθε γεγονός εκτελώντας ένα σχετικά σύντομο κομμάτι κώδικα. Στην πιο καθαρή μορφή τους, οι ρουτίνες χειρισμού γεγονότων πρέπει να εκτελούνται χωρίς διακοπή (non-preemptive) και γι' αυτόν τον λόγο πρέπει να είναι σύντομες.

Ο προγραμματισμός με βάση τα γεγονότα, απαιτεί τη χρήση εντολών/συναρτήσεων οι οποίες δεν μπλοκάρουν ώς το πέρας της λειτουργίας τους, αλλά αντ' αυτού επιστρέφουν, αφού ξεκινήσουν τη διαδικασία εκτέλεσής τους και ενημερώνουν σε μεταγενέστερο χρόνο για το πέρας της εκτέλεσης της λειτουργίας. Αυτός ο τρόπος λειτουργίας αναφέρεται ως *ασύγχρονος*. Σε αυτό το κεφάλαιο θα εστιάσουμε στη ροή ελέγχου η οποία οδηγείται από τα γεγονότα, χωρίς την αυστηρή χρήση ασύγχρονων συναρτήσεων. Στο Κεφάλαιο II θα μελετήσουμε τον πλήρως ασύγχρονο τρόπο λειτουργίας αυτού του τρόπου προγραμματισμού, μέσω του πλαισίου Twisted.

## I0.6 Το module select

Το module select παρέχει έναν μηχανισμό διαχείρισης γεγονότων τα οποία αφορούν μια ομάδα από sockets με τη γενική χρήση:

*select(inputs,outputs,excepts,timeout=None)*

Οι λίστες inputs, outputs, και excepts περιέχουν αντικείμενα socket τα οποία αναμένουν γεγονότα εισόδου, εξόδου, και συνθήκες εξαίρεσης αντίστοιχα. Το timeout (τύπος float) είναι ο μέγιστος χρόνος αναμονής σε δευτερόλεπτα. Όταν το timeout είναι None, η κλήση μπλοκάρει, μέχρι να συμβεί ένα ή περισσότερα γεγονότα. Όταν το timeout είναι 0, το select επιστρέφει αμέσως, επιστρέφοντας τα γεγονότα που μπορούν να αναφερθούν τη στιγμή εκείνη.

Το select επιστρέφει την πλειάδα (i,o,e). Το i είναι μια λίστα (ενδεχομένως, κενή) αντικειμένων που αναφέρουν γεγονότα εισόδου. Το o αντίστοιχα είναι μια λίστα (ενδεχομένως, κενή), αντικειμένων που αναφέρουν γεγονότα εξόδου. Τέλος e είναι μια λίστα (ενδεχομένως, κενή) αντικειμένων που αναφέρουν συνθήκες εξαίρεσης. Αν το timeout είναι None και η κλήση select επιστρέψει, τότε τουλάχιστον ένα από τα i, o, και e δεν είναι άδειο.

Το παράδειγμα της Εικόνας 10.10 χρησιμοποιεί το select για να υλοποιήσει τη λειτουργικότητα του διακομιστή του παραδείγματος της Εικόνας 10.1, με την πρόσθετη ικανότητα να μπορεί να εξυπηρετήσει ταυτόχρονα οποιοδήποτε αριθμό από πελάτες. Εκτελέστε το παράδειγμα της Εικόνας 10.10 σε ένα από τα τερματικά και δοκιμάστε να συνδεθείτε στο διακομιστή από περισσότερους του ενός πελάτες (θα χρειαστείτε, ενδεχομένως, περισσότερα των δύο τερματικών σε αυτήν την εξάσκηση).

Ο προγραμματισμός σε αυτό το επίπεδο είναι αρκετά περίπλοκος, όπως φαίνεται και από τη δομή του κώδικα της Εικόνας 10.10.

```python
import socket
import select
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.bind(('', 8881))
sock.listen(5)

# lists of sockets to watch for input and output events
ins = [sock]
ous = []
# mapping socket -> data to send on that socket when feasible
data = {}
# mapping socket -> (host, port) on which the client is running
adrs = {}

try:
    while True:
        i, o, e = select.select(ins, ous, [])  # no excepts nor timeout
        for x in i:
            if x is sock:
                # input event on sock means client trying to connect
                newSocket, address = sock.accept( )
                print "Connected from", address
                ins.append(newSocket)
                adrs[newSocket] = address
            else:
                # other input events mean data arrived, or disconnections
                newdata = x.recv(8192)
                if newdata:
                    # data arrived, prepare and queue the response to it
                    print "%d bytes from %s" % (len(newdata), adrs[x])
                    data[x] = data.get(x, '') + newdata
                    if x not in ous: ous.append(x)
                else:
                    # a disconnect, give a message and clean up
                    print "disconnected from", adrs[x]
                    del adrs[x]
                    try: ins.remove(x)
                    except ValueError: pass
                    x.close( )
        for x in o:
            # output events always mean we can send some data
            tosend = data.get(x)
            if tosend:
                nsent = x.send(tosend)
                print "%d bytes to %s" % (nsent, adrs[x])
                # remember data still to be sent, if any
                tosend = tosend[nsent:]
            if tosend:
                print "%d bytes remain for %s" % (len(tosend), adrs[x])
                data[x] = tosend
            else:
                try: del data[x]
                except KeyError: pass
                ous.remove(x)
                print "No data currently remain for", adrs[x]
finally:
    sock.close( )
```

**Εικόνα 10.10** *Διακομιστής TCP echo που χρησιμοποιεί select*

## I0.7 Επίλογος

Σε αυτό το κεφάλαιο μελετήσαμε την αφαίρεση των sockets και την υλοποίησή της στο module select της Python. Είδαμε παραδείγματα προγραμματισμού πελάτη-διακομιστή με χρήση sockets πάνω απο TCP, UDP, και IP multicast και τρόπους με τους οποίους μπορούμε να ενδοσκοπήσουμε την επικοινωνία σε επίπεδο πρωτοκόλλου μεταφοράς με χρήση εργαλείων όπως το Wireshark. Στη συνέχεια μελετήσαμε το δικτυακό προγραμματισμό με βάση τα γεγονότα και τη διεπαφή select. Αυτή η γνώση επιτρέπει στον προγραμματιστή της Python να υλοποιήσει σημαντική λειτουργικότητα δικτυακού διακομιστή με περιορισμένη χρήση νημάτων, επιτυγχάνοντας καλύτερη κλιμακωσιμότητα συγκριτικά με τη χρήση πολλαπλών νημάτων, ένα ανά socket. Ωστόσο ο παραγόμενος κώδικας σε αυτήν την περίπτωση είναι αρκετά πολύπλοκος. Στο επόμενο κεφάλαιο θα μελετήσουμε το πλαίσιο προγραμματισμού (Twisted) το οποίο απλοποιεί σημαντικά τον ασύγχρονο δικτυακό προγραμματισμό, με βάση τη ροή γεγονότων.

## Βιβλιογραφία/Αναφορές


Coulouris, G., Dollimore, J., & Kindberg, T. (2005). *Distributed systems: Concepts and Design* (4th Edition). Addison-Wesley Longman, Inc.

Fall, K., & Stevens, R. (1994). *TCP/IP Illustrated, Volume 1*: The Protocols (Second Edition). Addison-Welsey Publishing Company.

Kurose, J. F., & Ross, K. W. (2005). Computer Networking *A Top-Down Approach Featuring the Internet*. Pearson Education, Inc.

McKusick, K., Bostic, K., Karels, M. J., & Quarterman, J. S. (1996). *The Design and Implementation of the 4.4 BSD Operating System*. Addison-Wesley Longman, Inc.

Ousterhout, J. (1996). Why threads are a bad idea (for most purposes). In *USENIX Winter Technical Conference*. San Diego, CA.

Wireshark. (2015). *Wireshark Network Protocol Analyzer*. Retrieved from https://www.wireshark.org/

### Κριτήριο αξιολόγησης 1 (Βαθμός δυσκολίας: ●●)

**Ο αναγνώστης καλείται να πειραματιστεί με τα προγράμματα των Εικόνων 10.1 και 10.2 (TCP server, client) και με τη χρήση του wireshark στην πλευρά του client, αναπαράγοντας το output της Εικόνας 10.3. Στη συνέχεια ο αναγνώστης καλείται να πειραματιστεί με περιπτώσεις διακοπής της διεργασίας (ctrl-C) στην πλευρά του client, αναπαράγοντας το output της Εικόνας 10.5.**

**Απάντηση/Λύση**

### Κριτήριο αξιολόγησης 2 (Βαθμός δυσκολίας: ●●●)

**Η άσκηση αυτή σας ζητά να υλοποιήσετε ένα μηχανισμό πολυεκπομπής (multicasting) μεταξύ ενός αποστολέα και περισσοτέρων του ενός παραληπτών πάνω απο συνδέσεις TCP. Η πολυεκπομπή είναι μια μορφή ομαδικής επικοινωνίας κατά την οποία κάθε μήνυμα που στέλνεται απο τον αποστολέα λαμβάνεται απο όλα τα μέλη της ομάδας των παραληπτών. Ο αποστολέας γνωρίζει εκ των προτέρων τον αριθμό των παραληπτών (έστω N) και περιμένει τις αιτήσεις συνδέσεών τους. Μετά τη δημιουργία των N συνδέσεων προχωρά στην αποστολή ενός αριθμού μηνυμάτων, ενδεχομένως με κάποια χρονική καθυστέρηση μεταξύ τους, προς την ομάδα των παραληπτών. Ο αποστολέας θεωρεί ότι ένα μήνυμα το οποίο πολυεκπέμπει, έχει ληφθεί όταν λάβει επιβεβαιώσεις απο όλους τους παραλήπτες. Σε αυτήν την άσκηση υποθέστε ότι ο αποστολέας πολυεκπέμπει την ώρα της**

**ημέρας, ανά δευτερόλεπτο, προς τρεις (3) παραλήπτες. Εκτός της ώρας της ημέρας, κάθε μήνυμα του αποστολέα πρέπει να περιέχει και έναν αύξοντα αριθμό σειράς. Για τις υλοποιήσεις μπορείτε να επεκτείνετε τον κώδικα των Εικόνων 10.1 και 10.2. Χρησιμοποιήστε τουλάχιστον δύο μηχανήματα για να εγκαταστήσετε και εκτελέσετε τα μέρη του συστήματός σας.**

**Απάντηση/Λύση**

### Κριτήριο αξιολόγησης 3 (Βαθμός δυσκολίας: ●●●)

**Η άσκηση αυτή σας ζητά να υλοποιήσετε ένα μηχανισμό πολυεκπομπής (multicasting) μεταξύ ενός αποστολέα και περισσοτέρων του ενός παραληπτών πάνω απο συνδέσεις TCP με χρήση του API select στον αποστολέα. Ο αποστολέας δε γνωρίζει εκ των προτέρων τον αριθμό των παραληπτών και έτσι θα πρέπει να χειρίζεται αιτήσεις συνδέσεων ταυτόχρονα με το χειρισμό των υπαρχουσών συνδέσεων. Ο αποστολέας θεωρεί ότι ένα μήνυμα το οποίο πολυεκπέμπει έχει ληφθεί, όταν λάβει επιβεβαιώσεις από όλους τους παραλήπτες. Υποθέστε ότι ο αποστολέας πολυεκπέμπει την ώρα της ημέρας, ανά 5 δευτερόλεπτα προς όσους παραλήπτες είναι διαθέσιμοι. Εκτός της ώρας της ημέρας, κάθε μήνυμα του αποστολέα πρέπει να περιέχει και έναν αύξοντα αριθμό σειράς. Για τις υλοποιήσεις μπορείτε να επεκτείνετε τον κώδικα των Εικόνων 10.2 και 10.10. Χρησιμοποιήστε τουλάχιστον δύο μηχανήματα για να εγκαταστήσετε και εκτελέσετε τα μέρη του συστήματός σας.**

## Απάντηση/Λύση

# ΚΕΦΑΛΑΙΟ II

# Εισαγωγή στον προγραμματισμό κατανεμημένων συστημάτων λογισμικού

*Σύνοψη*

*Το κεφάλαιο εισάγει τον αναγνώστη στον προγραμματισμό κατανεμημένων συστημάτων λογισμικού. Στο κεφάλαιο αυτό πραγματευόμαστε το μοντέλο πελάτη-διακομιστή (client-server), το προγραμματιστικό πλαίσιο Twisted το οποίο διευκολύνει σημαντικά τον προγραμματισμό κατανεμημένων συστημάτων, και ένα παράδειγμα προγραμματισμού της πλευράς του πελάτη στο κατανεμημένο σύστημα αποθήκευσης Cassandra.*

*Προαπαιτούμενη γνώση*

*Η κατανόηση των εννοιών και παραδειγμάτων σε αυτό το κεφάλαιο απαιτεί βασική γνώση της γλώσσας Python και του δικτυακού προγραμματισμού, όπως αντίστοιχα παρουσιάστηκαν στα Κεφάλαια I-8 και I0 του παρόντος συγγράμματος.*

## II.I Εισαγωγή

Ένα *κατανεμημένο σύστημα* ορίζεται ως ένα σύστημα στο οποίο μέρη λογισμικού που βρίσκονται σε διαφορετικούς, δικτυωμένους υπολογιστές, επικοινωνούν και συντονίζουν τις δράσεις τους μέσω ανταλλαγής μηνυμάτων (Coulouris, Dollimore, & Kindberg, 2005). Στο Κεφάλαιο I0 μελετήσαμε τρόπους επικοινωνίας πάνω από δίκτυο μέσω της διεπαφής των sockets. Ο προγραμματισμός κατανεμημένων συστημάτων με κατευθείαν χρήση sockets, ωστόσο, οδηγεί σε προγράμματα υψηλής

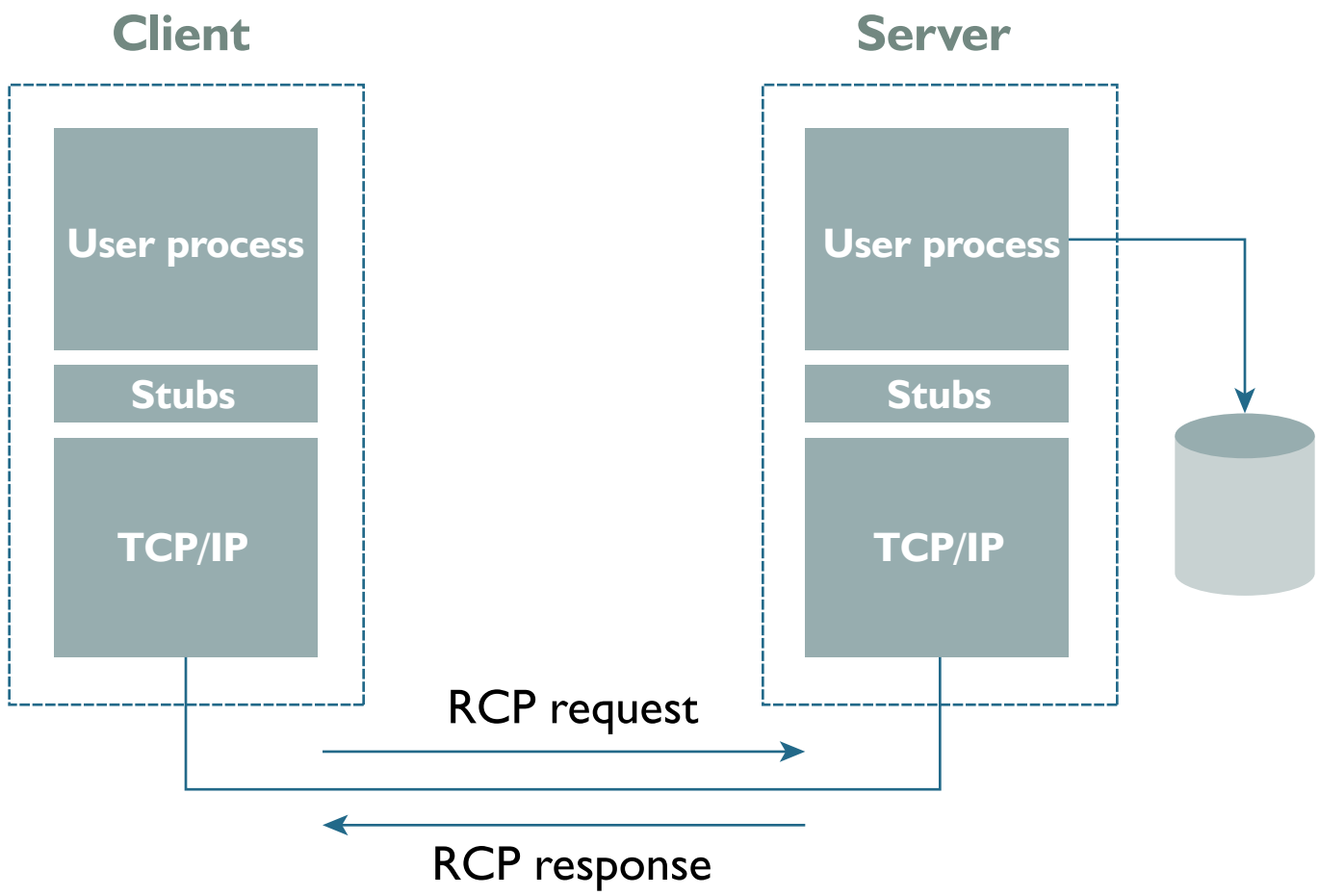
πολυπλοκότητας (όπως της Εικόνας 10.10), και άρα χρειαζόμαστε έναν ευκολότερο τρόπο.

Ένα διαδεδομένο μοντέλο κατανεμημένου συστήματος είναι αυτό του πελάτη-διακομιστή (client-server). Ο προγραμματισμός αυτού του μοντέλου κατανεμημένου συστήματος γίνεται συχνά με μια διεπαφή εφαρμογής-χρήστη υψηλότερου επιπέδου από τα sockets, αυτή των *απομακρυσμένων κλήσεων διαδικασιών* (remote procedure calls ή RPCs) (Birrell & Nelson, 1983). Ένας σχεδιαστικός στόχος των RPCs είναι να δώσουν την αίσθηση στον προγραμματιστή ότι εκτελεί μια *τοπική κλήση συνάρτησης*, αφαιρώντας την ανάγκη να χειριστεί ο ίδιος την πολυπλοκότητα της αποστολής και λήψης μηνυμάτων. Αυτό το χειρίζεται εσωτερικά η υλοποίηση των RPCs, αποστέλλοντας μηνύματα αίτησης (request) και απάντησης (response) πάνω από (συνήθως) TCP/IP συνδέσεις (Εικόνα 11.1). Οι υλοποιήσεις RPC παράγουν, αυτόματα, κώδικα ο οποίος χειρίζεται την εισαγωγή/εξαγωγή παραμέτρων και αποτελεσμάτων από τα μηνύματα αυτά, ώστε να μη χρειαστεί να το κάνει αυτό ο προγραμματιστής. Ο κώδικας αυτός συχνά αναφέρεται και ως stubs (Εικόνα 11.1).

Ο αρχικός σκοπός, λοιπόν, των RPCs ήταν να μπορεί να εστιάσει ο προγραμματιστής, κυρίως, στη λογική της εφαρμογής του, σαν να προγραμμάτιζε μία μη-κατανεμημένη εφαρμογή. Η φύση, όμως, ενός κατανεμημένου συστήματος καθιστά δύσκολο αυτό το έργο: Εφόσον οι υπολογιστές που τρέχουν τα client και server μέρη του συστήματος, μπορεί να καταρρεύσουν ανεξάρτητα (δηλαδή, να καταρρεύσει ο server και όχι ο client ή αντίστροφα), είναι δυνατόν η εκτέλεση ενός RPC να τερματίσει με μια *εξαίρεση* που μπορεί να οφείλεται σε αστοχία στο δίκτυο ή στο διακομιστή πριν ή μετά την εκτέλεση της απομακρυσμένης διαδικασίας.

Στην περίπτωση κατά την οποία ο πελάτης λάβει μια εξαίρεση κατά την εκτέλεση του RPC, δεν μπορεί να είναι σίγουρος αν η αστοχία έχει συμβεί *πριν* ή *μετά* την εκτέλεση της απομακρυσμένης διαδικασίας. Οι επιλογές του πελάτη είναι είτε να υποθέσει ότι δεν εκτελέστηκε και άρα να επαναλάβει την αίτηση μέχρι να λάβει επιτυχή απάντηση ή να θεωρήσει ότι εκτελέστηκε και άρα να μην επαναλάβει την αίτηση. Στην πρώτη περίπτωση λέμε ότι η υλοποίηση RPC εκτελεί την απομακρυσμένη διαδικασία «τουλάχιστον μια φορά» (τη φορά που έλαβε την επιτυχή απάντηση αλλά πιθανώς και σε κάποιες άλλες από τις προσπάθειες που έλαβε εξαί-

ρεση) ενώ στη δεύτερη περίπτωση «το μέγιστο μια φορά» (αν η απομακρυσμένη συνάρτηση εκτελέστηκε στη μία αυτή προσπάθεια).




**Εικόνα II.1** *Λειτουργία απομακρυσμένης κλήσης διαδικασίας (RPC) στο μοντέλο πελάτη-εξυπηρετητή*

Υλοποιήσεις των RPCs έχουν χρησιμοποιηθεί σε πολλά και σημαντικά συστήματα, όπως το Network File System (NFS), και χρησιμοποιείται σήμερα για την υλοποίηση πολλών κατανεμημένων συστημάτων, όπως το σύστημα Cassandra που πραγματευόμαστε στην Ενότητα II.3.

Παρά τα σημαντικά οφέλη του μοντέλου των RPCs, εφαρμογές που απαιτούν κυρίως τη μεταφορά δεδομένων και όχι την κλήση απομακρυσμένων διαδικασιών, ιδιαίτερα αυτές που εμπλέκουν επικοινωνία εντός μιας ομάδας διεργασιών (όπως η πολυεκπομπή που είδαμε στο Κεφάλαιο I0), ταιριάζουν περισσότερο σε ένα μοντέλο ανταλλαγής μηνυμάτων. Μια τέτοια διεπαφή η οποία μπορεί να χρησιμοποιηθεί για την ανταλλαγή μηνυμάτων, με χαμηλότερη πολυπλοκότητα κώδικα από αυτήν των sockets, προσφέρεται από το *πλαίσιο προγραμματισμού (programming framework)* Twisted, το οποίο θα δούμε στην Ενότητα II.2.

Η έννοια του *προγραμματιστικού πλαισίου* γενικότερα στοχεύει στην απλοποίηση του προγραμματισμού συστημάτων και της γρήγορης παραγωγής πρότυπου κώδι-

κα (Wayner, 2015). Ένα προγραμματιστικό πλαίσιο παρέχει υποστηρικτικό κώδικα ο οποίος μειώνει αρκετά τον κώδικα που πρέπει να παρέχει ο χρήστης, επιτρέποντας στον τελευταίο να εστιάσει στη λογική της εφαρμογής. Το προγραμματιστικό πλαίσιο, συνήθως, διαχειρίζεται τον έλεγχο ροής του προγράμματος και παρέχει μια όσο το δυνατόν απλούστερη διεπαφή χρήστη-πλαισίου.

Ο τρόπος επικοινωνίας με χρήση του πλαισίου Twisted είναι υψηλότερου επιπέδου από τα sockets. Μοιάζει με το μοντέλο RPC στο ότι η κλήση μιας συνάρτησης αποστολής μηνύματος στην πλευρά του πελάτη, προκαλεί *αυτόματα* την κλήση μιας συνάρτησης λήψης μηνύματος στην πλευρά του διακομιστή (χωρίς ο τελευταίος να χρειαστεί να καλέσει socket.recv). Διαφέρει, ωστόσο, από τα RPCs στο ότι η ανταλλαγή μηνυμάτων υλοποιείται από την εφαρμογή. Με χρήση του πλαισίου Twisted μπορεί να υλοποιηθούν σημαντικές κατανεμημένες εφαρμογές, όπως διάφορες μορφές πολυεκπομπής (Coulouris et al., 2005).

## II.2 Το πλαίσιο δικτυακού προγραμματισμού Twisted

Το πλαίσιο προγραμματισμού Twisted υποστηρίζει το δικτυακό προγραμματισμό, με βάση τη διαχείριση γεγονότων και την ασύγχρονη κλήση μεθόδων επικοινωνίας. Κεντρικές έννοιες στο Twisted είναι ο *βρόγχος γεγονότων* (event loop), εντός του οποίου εκτελεί, διαρκώς, το μοναδικό νήμα εκτέλεσης, και η βασική κλάση Protocol, την οποία επεκτείνει κάθε πρωτόκολλο επιπέδου εφαρμογής (όπως το Peer το οποίο θα αναλύσουμε παρακάτω).

Στην Εικόνα II.2 βλέπουμε μια γραφική απεικόνιση των βασικών χαρακτηριστικών της χρήσης του Twisted. Η απεικόνιση αυτή εστιάζει σε χαρακτηριστικά που θα χρησιμοποιήσουμε στη συνέχεια αυτού του κεφαλαίου και όχι σε μια πλήρη παρουσίαση του Twisted (ο ενδιαφερόμενος αναγνώστης μπορεί να συμβουλευτεί τον σύνδεσμο (Twisted Matrix Labs, 2015)). Ο χαρακτηρισμός ενός μέρους ως client ή server έχει να κάνει με το ποιος ξεκίνησε πρώτος το στήσιμο της σύνδεσης καλώντας connect και ποιος αποδέχθηκε την αίτηση σύνδεσης αντίστοιχα. Μετά το στήσιμο της σύνδεσης, οι δύο πλευρές εκτελούν το βρόγχο γεγονότων.

Ο βρόγχος αναφέρεται και ως reactor (το αντίστοιχο module ονομάζεται twisted.internet.reactor), λόγω της φύσης λειτουργίας του, η οποία είναι *να αντιδρά* (react) σε εισερχόμενα γεγονότα.

Ο βρόγχος γεγονότων λαμβάνει ως είσοδο γεγονότα όπως *εισερχόμενα δεδομένα, αιτήσεις για στήσιμο σύνδεσης*, ή *ειδοποιήσεις τερματισμού σύνδεσης*. Κάθε γεγονός αντιστοιχίζεται στη σύνδεση στην οποία αναφέρεται, η οποία με τη σειρά της αντιστοιχεί σε ένα αντικείμενο τύπου Protocol (ένα σε κάθε πλευρά της σύνδεσης), που με τη σειρά του συνδέεται με το αντίστοιχο αντικείμενο τύπου Protocol στην επικοινωνούσα διεργασία. Το αντικείμενο δημιουργείται κατά το στήσιμο της σύνδεσης από την κλάση-εργοστάσιο ProtocolFactory. Κάθε δικτυακή εφαρμογή (όπως οι Echo και Peer που θα δούμε στη συνέχεια) δημιουργεί μια υποκλάση της Protocol, οι κύριες μέθοδοι της οποίας είναι οι παρακάτω χειριστές (handlers) των γεγονότων:

connectionMade(self): Καλείται, όταν η σύνδεση έχει στηθεί επιτυχώς.

dataReceived(self, data): Καλείται, όταν υπάρχουν εισερχόμενα δεδομένα.

connectionLost(self, reason): Καλείται, όταν διακοπεί η σύνδεση.

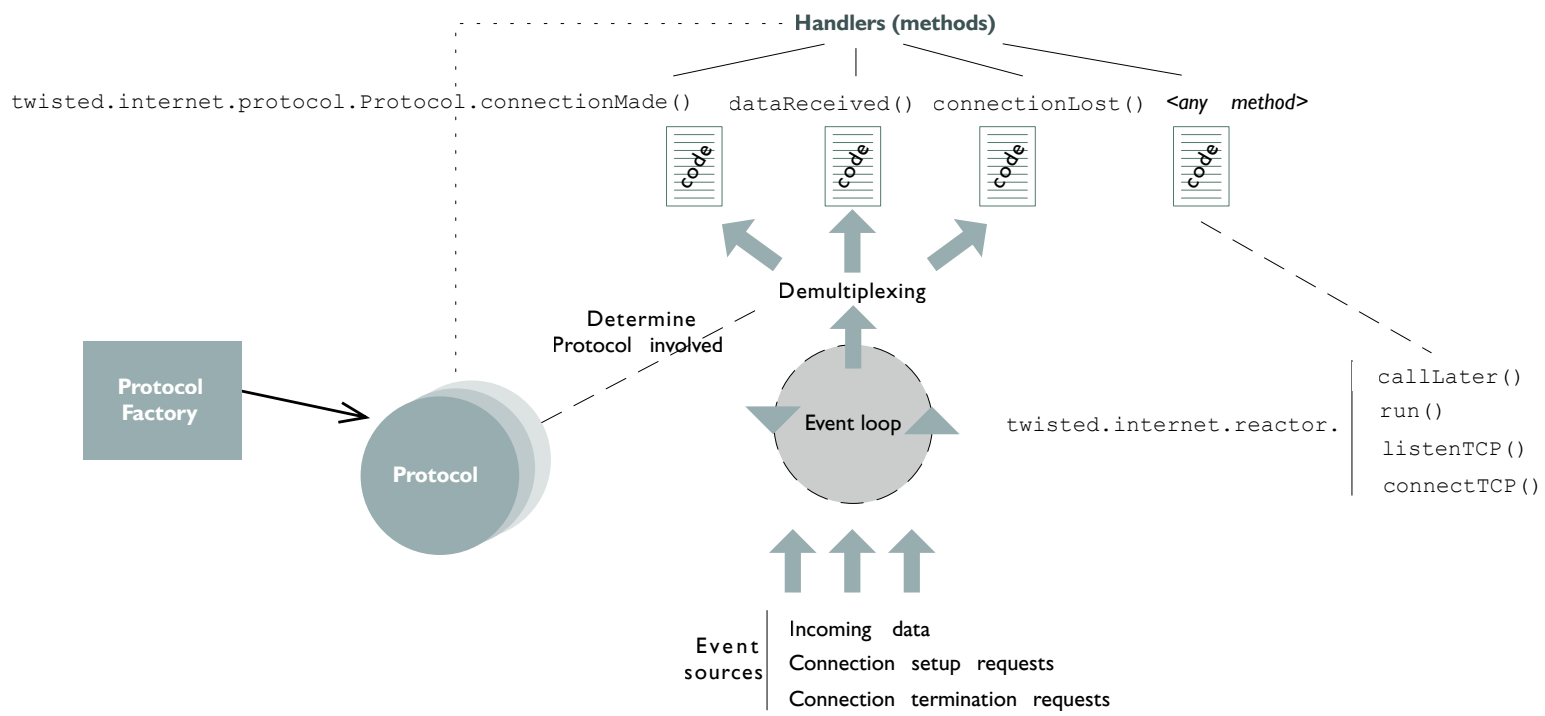
Επιπρόσθετα, κάθε υποκλάση της Protocol ορίζει τη μέθοδο __init__(self, factory, p) η οποία καλείται από την ProtocolFactory κατά τη δημιουργία του αντικειμένου.

Εκτός από τις μεθόδους-χειριστές (handlers), άλλες χρήσιμες μέθοδοι που προσφέρει ο βρόγχος γεγονότων (reactor) του Twisted, και οι οποίες καλούνται απευθείας από τον κώδικα είναι:

reactor.callLater(sec, method): Καλεί τη μέθοδο method (ορισμένη από το χρήστη) εντός συγκεκριμένου χρονικού διαστήματος (sec). Αυτή η συνάρτηση είναι χρήσιμη για την περιοδική εκτέλεση λειτουργιών, όπως η περιοδική αποστολή μηνυμάτων τύπου “heartbeat”.

reactor.listenTCP(port, factory): Κατ’ αντιστοιχία με το socket.listen που είδαμε στο Κεφάλαιο I0, η listenTCP ζητά από το βρόγχο να ακούει στην πόρτα port. Όταν ολοκληρωθεί κάποια σύνδεση, δημιουργείται νέο αντικείμενο τύπου Protocol (από την κλάση ProtocolFactory (όρισμα factory)).

reactor.connectTCP(host, port, factory): Κατ' αντιστοιχία με το socket.connect που είδαμε στο Κεφάλαιο 10, η connectTCP ζητά από το βρόγχο να ξεκινήσει μια σύνδεση προς τη διεύθυνση host και πόρτα port. Όταν ολοκληρωθεί η σύνδεση, δημιουργείται νέο αντικείμενο τύπου Protocol (από την κλάση ProtocolFactory (όρισμα factory)).




**Εικόνα 11.2** *Λειτουργία του βρόγχου γεγονότων του Twisted*

Στην Εικόνα 11.3 θα δούμε μια πιο εκτεταμένη γραφική απεικόνιση των βασικών χαρακτηριστικών και της τυπικής λειτουργίας του client μέρους με πολλαπλές συνδέσεις.

Η αποστολή δεδομένων πάνω από μια σύνδεση την οποία ενθυλακώνει αντικείμενο τύπου Protocol γίνεται μέσω της συνάρτησης transport.write(data) η οποία προσφέρεται από τη γονική κλάση Protocol.

Αν ο προγραμματιστής θέλει να εξειδικεύσει τη δημιουργία αντικειμένων Protocol, τότε πρέπει να δημιουργήσει μια υποκλάση της ProtocolFactory η οποία να κάνει ακριβώς αυτό. Οι μέθοδοι που, συνήθως, χρειάζεται να υλοποιηθούν σε αυτήν την περίπτωση είναι οι:

- buildProtocol(self, addr): Καλείται, για να δημιουργήσει ένα εξειδικευμένο αντικείμενο τύπου Protocol.

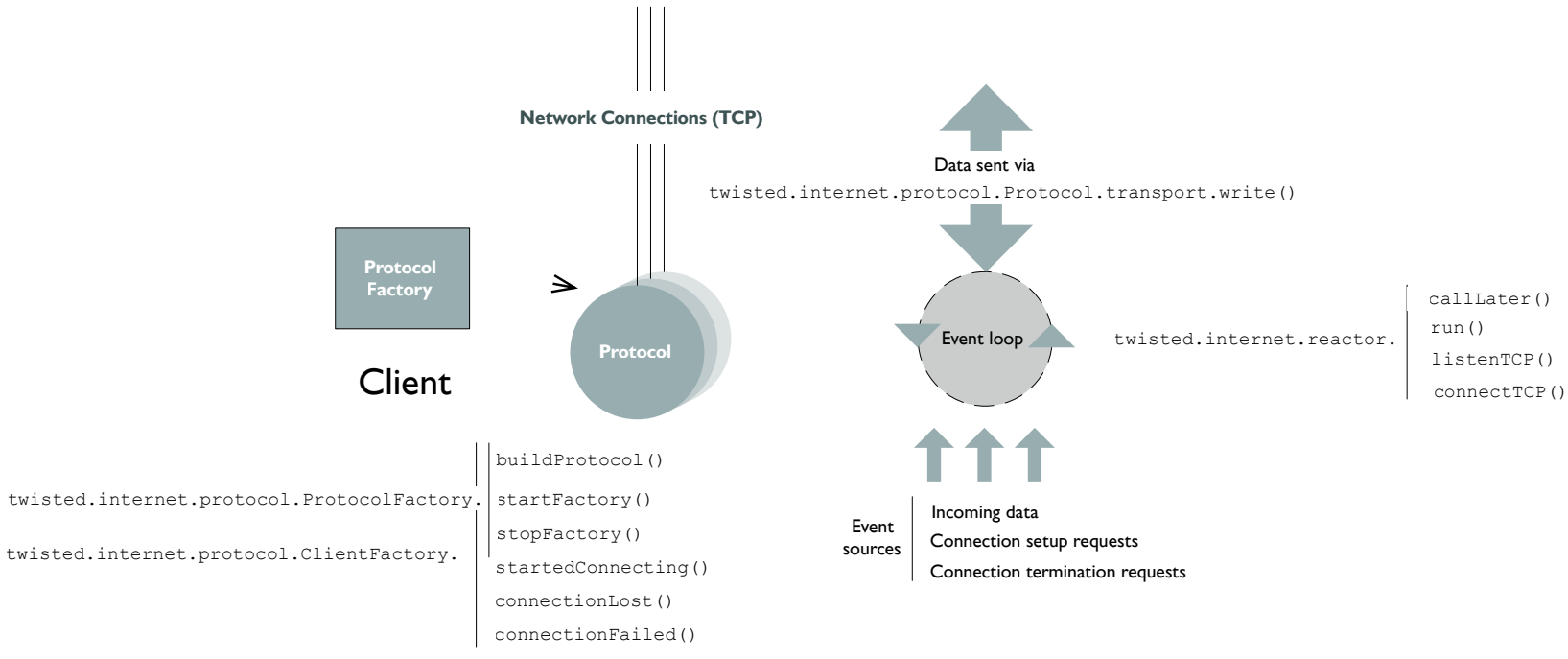
- startFactory(self): Καλείται, πριν ζητηθεί από το βρόγχο να ακούει σε κάποια πόρτα. Αυτή η μέθοδος καλείται μόνο μία φορά για να εκτελέσει αρχικές δράσεις, ακόμα και αν ο βρόγχος ακούει σε περισσότερες της μίας πόρτες.
- stopFactory(self): Καλείται, πριν ζητηθεί από το βρόγχο να σταματήσει να ακούει σε κάποια πόρτα. Αυτή η μέθοδος καλείται, συνήθως, για να απελευθερώσει πόρους.

Μια υποκλάση της ProtocolFactory, την οποία το Twisted προσφέρει έτοιμη, είναι η ClientFactory η οποία παράγει αντικείμενα τύπου ClientProtocol, εξειδικευμένα για λειτουργίες τύπου client. Μέθοδοι της ClientFactory (επιπρόσθετες των παραπάνω της ProtocolFactory) είναι οι:

- startedConnecting(self, connector): Καλείται, όταν η διαδικασία του στησίματος μιας σύνδεσης έχει ξεκινήσει. Το πλαίσιο Twisted δεν ενθαρρύνει τη χρήση του αντικειμένου connector από τους προγραμματιστές εφαρμογών και έτσι δε θα ασχοληθούμε περαιτέρω με αυτό.
- clientConnectionFailed(self, connector, reason): Καλείται, όταν μια προσπάθεια για σύνδεση ήταν ανεπιτυχής.
- clientConnectionLost(self, connector, reason): Καλείται, όταν κάποια υπάρχουσα σύνδεση διακοπεί.

Η ClientFactory συνεργάζεται με τη μέθοδο reactor.connectTCP(host,port,factory) του βρόγχου γεγονότων. Μόλις μια σύνδεση έχει στηθεί μέσω της connectTCP, καλείται η μέθοδος startedConnecting.

Στη συνέχεια θα δούμε δύο παραδείγματα προγραμματισμού με χρήση του Twisted. Θα ξεκινήσουμε με ένα απλό παράδειγμα (κλάση Echo) και θα συνεχίσουμε με ένα λίγο πιο σύνθετο (κλάση Peer).




**Εικόνα II.3** *Λειτουργία του client μέρους στο Twisted με πολλαπλές συνδέσεις*

## II.2.1 Echo class

Η πρώτη μας εμπειρία με το πλαίσιο προγραμματισμού Twisted θα γίνει μέσω ενός απλού παραδείγματος, του γνωστού μας Echo (McKellar, 2013). Το παράδειγμα αυτό (Εικόνα II.4) υποδεικνύει τη χρήση των μεθόδων των Protocol και reactor για την υλοποίηση ενός διακομιστή echo[1]. Ο αναγνώστης καλείται να αντιπαραβάλει τη συνοπτικότητα και απλότητα του παραδείγματος αυτού με την υλοποίηση του ίδιου διακομιστή με χρήση του select (Εικόνα I0.II) ή των βασικών μεθόδων του socket API (Εικόνα I0.I).

[1] Ο κώδικας αυτού του κεφαλαίου έχει δοκιμαστεί με την έκδοση 14.0.2 του Twisted.

```python
import twisted.internet.protocol
import twisted.internet.reactor

class EchoProtocol(twisted.internet.protocol.Protocol):
    def connectionMade(self):
        self.peer = self.transport.getPeer( )[1:]
        print "Connected from", self.peer
    def dataReceived(self, data):
        self.transport.write(data)
    def connectionLost(self, reason):
        print "Disconnected from", self.peer, reason.value

factory = twisted.internet.protocol.Factory( )
factory.protocol = EchoProtocol

twisted.internet.reactor.listenTCP(8881, factory)
twisted.internet.reactor.run( )
```

**Εικόνα ΙΙ.4** *Echo server*

Το EchoProtocol υλοποιεί τις τρεις μεθόδους-χειριστές (handlers) που περιγράφηκαν νωρίτερα. Στο κυρίως σώμα του προγράμματος καλούμε τον reactor σε δύο περιπτώσεις. Κατά την πρώτη για να του ζητήσουμε να ακούει στην πόρτα 8881, περνώντας του το ProtocolFactory που θα χρησιμοποιήσουμε για τη δημιουργία αντικειμένων Protocol σε κάθε σύνδεση. Σε αυτήν την περίπτωση χρησιμοποιούμε την standard υλοποίηση του ProtocolFactory, η οποία θα παράγει αντικείμενα τύπου EchoProtocol (factory.protocol = EchoProtocol). Στη δεύτερη κλήση (twisted.internet.reactor.run()) ξεκινάμε την αέναη εκτέλεση του βρόγχου γεγονότων. Σε αυτό το παράδειγμα, εστιάσαμε στην υλοποίηση ενός απλού server. Ο αναγνώστης μπορεί να δοκιμάσει την υλοποίηση του server χρησιμοποιώντας το πρόγραμμα telnet ή ssh προς την πόρτα 8881 (McKellar, 2013). Στο επόμενο παράδειγμα θα δούμε μια πληρέστερη υλοποίηση των μερών του client και server.

### II.2.2 Peer class

Σε αυτήν την ενότητα θα μελετήσουμε την κλάση Peer, η οποία μπορεί να αποτελέσει δομικό λίθο για την υλοποίηση γενικότερων λειτουργιών συστημάτων κατανεμημένου λογισμικού. Η κλάση Peer μαζί με τις υποστηρικτικές κλάσεις και συναρτήσεις μπορεί να συμπεριφερθεί είτε ως πελάτης (συνδεόμενη σε ένα διακομιστή και, περιοδικά, στέλνοντας ένα μήνυμα ενημέρωσης προς αυτόν) ή ως διακομιστής (δεχόμενος τα περιοδικά μηνύματα από συνδεδεμένους πελάτες και απαντώντας σε καθένα από αυτά με μια επιβεβαίωση).

Το παράδειγμα της Εικόνας II.5 υποδεικνύει το χειρισμό των παραμέτρων εισόδου, οι οποίες είναι (α) ο τρόπος λειτουργίας της συγκεκριμένης εκτέλεσης (πελάτης (client) ή διακομιστής (server)), και (β) η IP διεύθυνση του μηχανήματος και πόρτα στην οποία θα συνδεθεί ο πελάτης. Χρησιμοποιώντας το module optparse, παίρνουμε τις παραμέτρους εισόδου σαν μια πλειάδα (args). Στη συνέχεια εξάγουμε τον τρόπο λειτουργίας (peer_type), και την IP διεύθυνση και πόρτα μέσω της συνάρτησης parse_address.

10-peer-twisted-server.py

```
11  import optparse
12
13  from twisted.internet.protocol import Protocol, ClientFactory
14  from twisted.internet import reactor
15  import time
16
17  def parse_args():
18      usage = """usage: %prog [options] [client|server] [hostname]:port
19
20      python peer.py server 127.0.0.1:port """
21
22      parser = optparse.OptionParser(usage)
23
24      _, args = parser.parse_args()
25
26      if len(args) != 2:
27          print parser.format_help()
28          parser.exit()
29
30      peertype, addresses = args
31
32      def parse_address(addr):
33          if ':' not in addr:
34              host = '127.0.0.1'
35              port = addr
36              else:
37              host, port = addr.split(':', 1)
38
39          if not port.isdigit():
40              parser.error('Ports must be integers.')
41
42          return host, int(port)
43
44      return peertype, parse_address(addresses)
```

**Εικόνα II.5** *Ορισμός παραμέτρων εισόδου*

Το παράδειγμα της Εικόνας II.6 υποδεικνύει την υλοποίηση της κλάσης Peer η οποία κληρονομεί από την κλάση Protocol. Αν η εκτέλεση συμπεριφέρεται ως πελάτης, με το στήσιμο της σύνδεσης καλεί κάθε 5 λεπτά τη μέθοδο `sendUpdate`, η οποία στέλνει το string '<update>' στο διακομιστή. Αν η εκτέλεση συμπεριφέρεται ως διακομιστής, σε κάθε παραλαβή του '<update>' απαντά (μέθοδος `sendAck`) με μια επιβεβαίωση '<Ack>'.

```
47  class Peer(Protocol):
48
49      acks = 0
50      connected = False
51
52      def __init__(self, factory, peer_type):
53          self.pt = peer_type
54          self.factory = factory
55
56      def connectionMade(self):
57          if self.pt == 'client':
58              self.connected = True
59              reactor.callLater(5, self.sendUpdate)
60          else:
61              print "Connected from", self.transport.client
62              try:
63                  self.transport.write('<connection up>')
64              except Exception, e:
65                  print e.args[0]
66              self.ts = time.time()
67
68      def sendUpdate(self):
69          print "Sending update"
70          try:
71              self.transport.write('<update>')
72          except Exception, ex1:
73              print "Exception trying to send: ", ex1.args[0]
74          if self.connected == True:
75              reactor.callLater(5, self.sendUpdate)
76
77      def sendAck(self):
78          print "sendAck"
79          self.ts = time.time()
80          try:
81              self.transport.write('<Ack>')
82          except Exception, e:
83              print e.args[0]
84
85      def dataReceived(self, data):
86          if self.pt == 'client':
87              print 'Client received ' + data
88              self.acks += 1
89          else:
90              print 'Server received ' + data
91              self.sendAck()
92
93      def connectionLost(self, reason):
94          print "Disconnected"
95          if self.pt == 'client':
96              self.connected = False
97              self.done()
98
99      def done(self):
100         self.factory.finished(self.acks)
101
```

**Εικόνα ΙΙ.6** *Κλάση Peer*

Το παράδειγμα της Εικόνας II.7 υποδεικνύει την υλοποίηση της κλάσης PeerFactory.

```
103 class PeerFactory(ClientFactory):
104
105     def __init__(self, peertype, fname):
106         print '@__init__'
107         self.pt = peertype
108         self.acks = 0
109         self.fname = fname
110         self.records = []
111
112     def finished(self, arg):
113         self.acks = arg
114         self.report()
115
116     def report(self):
117         print 'Received %d acks' % self.acks
118
119     def clientConnectionFailed(self, connector, reason):
120         print 'Failed to connect to:', connector.getDestination()
121         self.finished(0)
122
123     def clientConnectionLost(self, connector, reason):
124         print 'Lost connection.  Reason:', reason
125
126     def startFactory(self):
127         print "@startFactory"
128         if self.pt == 'server':
129             self.fp = open(self.fname, 'w+')
130
131     def stopFactory(self):
132         print "@stopFactory"
133         if self.pt == 'server':
134             self.fp.close()
135
136     def buildProtocol(self, addr):
137         print "@buildProtocol"
138         protocol = Peer(self, self.pt)
139         return protocol
140
```

**Εικόνα II.7** *Κλάση PeerFactory*

Το παράδειγμα της Εικόνας II.8 αποτελεί το κυρίως πρόγραμμα, το οποίο διαχωρίζει τις περιπτώσεις κατά τις οποίες η εκτέλεση προορίζεται να λειτουργήσει ως πελάτης ή ως διακομιστής. Η λειτουργία πελάτη ξεκινά τη σύνδεση προς το διακομιστή και εισέρχεται στο βρόχο γεγονότων (event loop). Η λειτουργία διακομιστή ξεκινά να ακούει σε συγκεκριμένη πόρτα (στο παράδειγμα αυτό, στην

πόρτα 8888) και εισέρχεται στο βρόχο γεγονότων. Από το σημείο αυτό και μετά ο χειρισμός οποιουδήποτε γεγονότος εκτελείται ως η κλήση κάποιας συνάρτησης της κλάσης Peer.

```
__name__ == '__main__':
 peer_type, address = parse_args()


 if peer_type == 'server':
     factory = PeerFactory('server', 'log')
     reactor.listenTCP(8888, factory)
     print "Starting server @" + address[0] + " port " + str(add
 else:
     factory = PeerFactory('client', '')
     host, port = address
     print "Connecting to host " + host + " port " + str(port)
     reactor.connectTCP(host, port, factory)

 reactor.run()
```

**Εικόνα II.8** *Κυρίως πρόγραμμα*

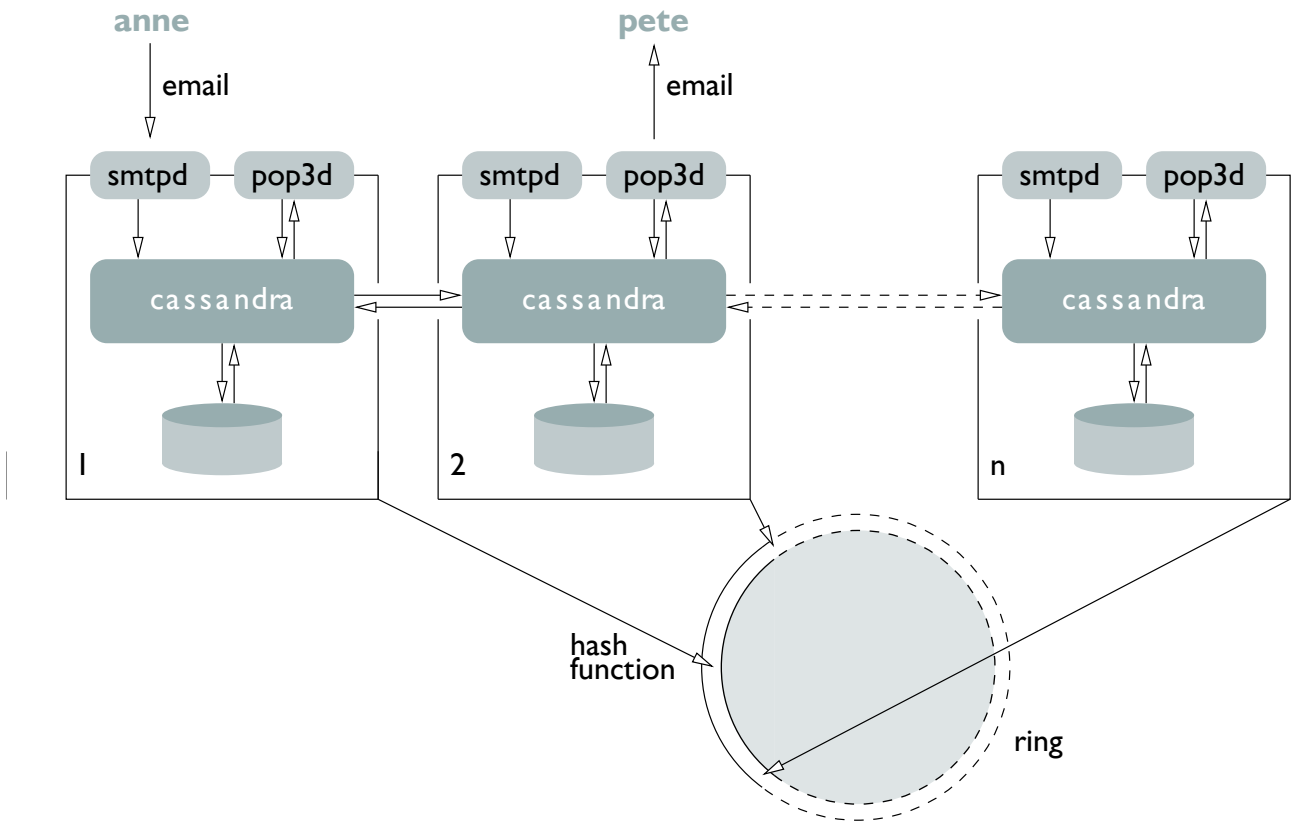
Ο αναγνώστης καλείται να πειραματιστεί με το πρόγραμμα των Εικόνων II.3-II.7, ξεκινώντας μια εκτέλεση διακομιστή σε ένα από τα τερματικά παράθυρα και μια εκτέλεση πελάτη σε ένα άλλο. Στη συνέχεια, η εκκίνηση ενός ακόμα πελάτη θα αναδείξει το πρόβλημα της διαχείρισης περισσότερης της μιας συνδέσεων στην πλευρά του διακομιστή (δείτε σχετική άσκηση στο τέλος του κεφαλαίου).

## II.3 Προγραμματιστική πρόσβαση στο κατανεμημένο σύστημα Cassandra

Σε αυτήν την ενότητα εισάγουμε τον αναγνώστη στην προγραμματιστική πρόσβαση στο κατανεμημένο συστήματος αποθήκευσης Cassandra (Lakshman, A., & Malik, 2010). Η περιγραφή ακολουθεί την παρουσίαση επιστημονικής εργασίας των συγγραφέων (Koromilas & Magoutis, 2011) και εστιάζει στη συνοπτικότητα και ταχεία παραγωγή πρωτότυπου συστήματος που καθιστά δυνατή η χρήση της Python.

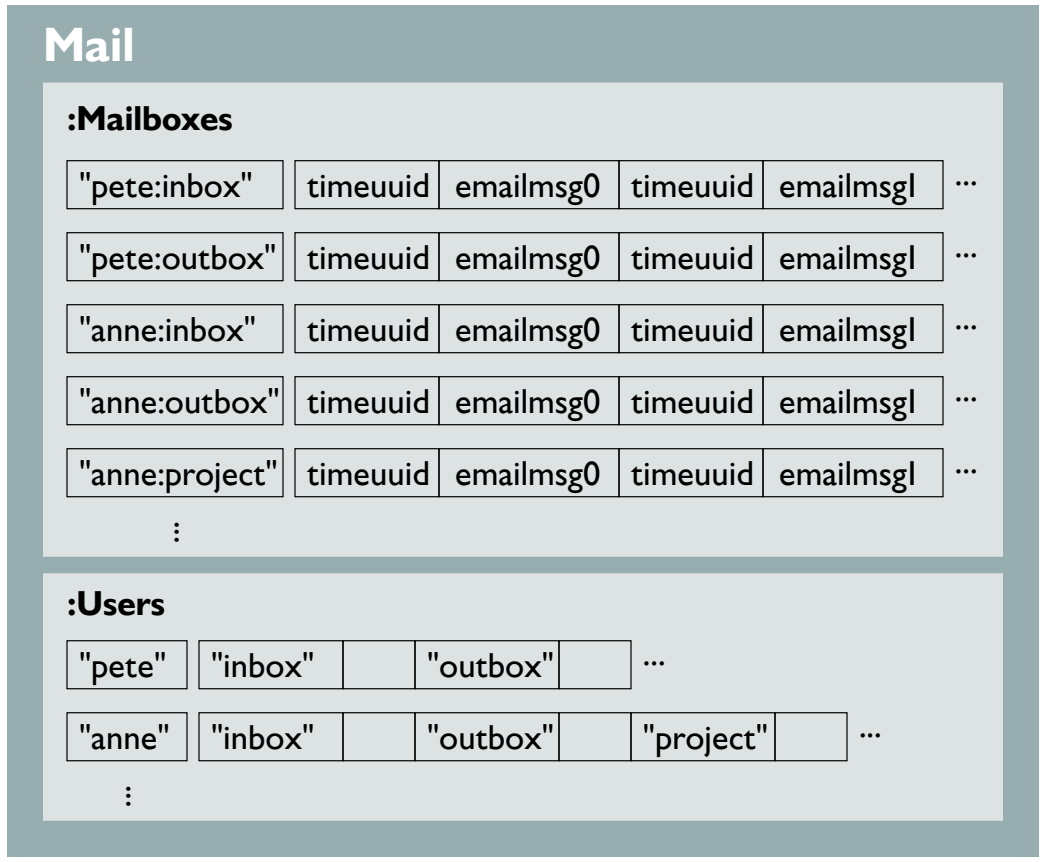
Το κατανεμημένο σύστημα αποθήκευσης Apache Cassandra ανήκει σε μια σχετικά νέα κατηγορία συστημάτων (γνωστή με το όνομα NoSQL) τα οποία προσφέρουν δομημένη οργάνωση δεδομένων, ταχεία αναζήτηση μέσω ευρετηρίων, χωρίς ωστόσο να υποστηρίζουν τις πλήρεις λειτουργίες διαχείρισης δεδομένων οι οποίες είναι διαθέσιμες στις παραδοσιακές σχεσιακές βάσεις δεδομένων SQL. Η νέα αυτή κατηγορία συστημάτων δημιουργήθηκε από την ανάγκη να επιτευχθεί καλύτερη κλιμακωσιμότητα από όση είναι εφικτή στις παραδοσιακές SQL παράλληλες/κατανεμημένες βάσεις δεδομένων, μειώνοντας τη λειτουργικότητα η οποία, ωστόσο, δεν είναι πάντα απαραίτητη στις εφαρμογές τις οποίες στοχεύουν τα NoSQL συστήματα.

Το σύστημα Cassandra, λοιπόν, αποτελείται από μια ομάδα κόμβων, λειτουργικά ισοδύναμων, που προσδίδουν αποθηκευτική ικανότητα στο σύστημα (Εικόνα II.9). Οι πελάτες του συστήματος (smtpd και pop3d στην Εικόνα II.9) μπορούν να συνδεθούν σε οποιοδήποτε κόμβο για να προσπελάσουν δεδομένα μέσω ενός απλού put/get API. Το ευρετήριο δεδομένων το οποίο χρησιμοποιεί το Cassandra, βασίζεται σε μια δομή hash η οποία υλοποιείται με μια συνάρτηση τύπου MD5 ή SHA-1 και απεικονίζει το κύριο κλειδί κάθε γραμμής δεδομένων ενός πίνακα σε έναν ή περισσότερους από τους κόμβους του συστήματος (περισσότερους του ενός όταν έχει επιλεχθεί η χρήση αντιγράφων για λόγους υψηλής απόδοσης και διαθεσιμότητας).




**Εικόνα II.9** *Δομή κατανεμημένου συστήματος Cassandra και εφαρμογής SMTP server πάνω από αυτό*

Στη συνέχεια θα εστιάζουμε στο μοντέλο δεδομένων του Cassandra και το σχήμα δεδομένων που σχεδιάστηκε για την εφαρμογή διακομιστή E-Mail (πρωτόκολλο SMTP) πάνω από το Cassandra. Η συγκεκριμένη εφαρμογή θα αναφέρεται στη συνέχεια ως CassMail. Η βασική μονάδα δεδομένων του Cassandra είναι η στήλη (column) την οποία στη συνέχεια θα αναφέρουμε και ως block. Ένα block αποτελείται από ένα κλειδί (key) και μια τιμή (value). Μια ακολουθία από blocks (οποιοσδήποτε αριθμός από αυτά) μαζί αποτελούν μια γραμμή (row). Τα blocks σε μια γραμμή μπορούν να ταξινομηθούν με οποιοδήποτε τρόπο επιθυμεί ο χρήστης, με βάση τον τύπο του κλειδιού (για παράδειγμα, με χρονική σειρά). Κάθε γραμμή ταυτοποιείται από το δικό της ιδιαίτερο κλειδί. Μια γραμμή αντιστοιχίζεται σε έναν ή περισσότερους Cassandra κόμβους, βάσει της δομής hashing που αναφέραμε νωρίτερα. Οι γραμμές ομαδοποιούνται σε οικογένειες στηλών (column families), οι οποίες είναι έννοιες που μοιάζουν με σχεσιακούς πίνακες. Οι οικογένειες στηλών ομαδοποιούνται σε χώρους κλειδιών (keyspaces). Στην Εικόνα II.10 βλέπουμε το μοντέλο δεδομένων που σχεδιάστηκε για την αποθήκευση και αναζήτηση E-Mail από ένα διακομιστή του πρωτοκόλλου SMTP.




**Εικόνα II.10** *Μοντέλο δεδομένων Cassandra για τον E-Mail (SMTP) εξυπηρετητή*

Το Cassandra, όπως τα περισσότερα αντίστοιχα κατανεμημένα συστήματα, συνοδεύονται από κάποιον πελάτη/οδηγό γραμμένο σε Python και ο οποίος προσφέρει μια προγραμματιστική διεπαφή (API) προς τις εφαρμογές. Σε αυτήν την ενότητα θα μελετήσουμε έναν τέτοιο πελάτη/οδηγό, το pycassa (pycassa 1.11.0 Documentation, 2015). Το παράδειγμα της Εικόνας II.10 υποδεικνύει τη χρήση, μεταξύ άλλων, του smtpd module, του re (regular expression), και του pycassa. Το smtpd υλοποιεί έναν SMTP server ως μια κλάση με τις παρακάτω μεθόδους:

- SMTPServer(localaddr, remoteaddr): Δημιουργεί ένα αντικείμενο τύπου SMTPServer στην τοπική πόρτα localaddr, σε σύνδεση με έναν upstream SMTP relay, στην πόρτα remoteaddr. Αυτή η κλάση κληρονομεί από το asyncore.dispatcher, ένα προγενέστερο του Twisted πλαίσιο ασύγχρονου δικτυακού προγραμματισμού, με αντίστοιχο βρόχο γεγονότων (asyncore.loop()).
- process_message(peer, mailfrom, rcpttos, data): Αυτή η μέθοδος πρέπει να γίνει override για να επεξεργαστεί ένα μήνυμα, διαφορετικά προκαλείται εξαίρεση τύπου NotImplementedError. Η διεύθυνση του απομακρυσμένου υπολογιστή είναι peer, η διεύθυνση του αποστολέα και πα-

ραλήπτη είναι mailfrom και rcpttos αντίστοιχα, και data είναι ένα string που περιέχει το μήνυμα του ηλεκτρονικού ταχυδρομείου (σε RFC 2822 format).

Το παράδειγμα της Εικόνας II.10 δημιουργεί έναν SMTP διακομιστή (CassSMTPServer) ο οποίος αποθηκεύει μηνύματα ηλεκτρονικού ταχυδρομείου στο Cassandra.

```
import smtpd
import asyncore
import time
import uuid
import pycassa
import re
import sys

class CassSMTPServer(smtpd.SMTPServer):
    def __init__(*args, **kwargs):
        smtpd.SMTPServer.__init__(*args, **kwargs)

    def process_message(self, peer, mailfrom, rcpttos, data):
        now = time.strftime('%a %b %d %T %Y', time.gmtime())
        head = 'From {0} {1}\n'.format(mailfrom, now)
        email = head + data
        user = re.search('(.*)@(.*)', rcpttos[0]).group(1)
        mbox = 'inbox'
        c = pycassa.ConnectionPool('Mail')
        cl = pycassa.cassandra.ttypes.ConsistencyLevel.ONE
        cf = pycassa.ColumnFamily(c, 'Mailboxes', write_consistency_level=c
        cf.insert('{0}:{1}'.format(user, mbox),
            {uuid.uuid1() : email})
        cf = pycassa.ColumnFamily(c, 'Users', write_consistency_level=cl)
        cf.insert(user, {mbox : ''})
        return


if __name__ == '__main__':
    port = int(sys.argv[1]) if len(sys.argv) > 1 else 8025
    s = CassSMTPServer(('0.0.0.0', port), None)
    asyncore.loop()
```

**Εικόνα II.11** *Προγραμματιστική πρόσβαση πελάτη στο κατανεμημένο σύστημα Cassandra*

## II.4 Επίλογος

Σε αυτό το κεφάλαιο εξετάσαμε παραδείγματα προγραμματισμού κατανεμημένων συστημάτων που ακολουθούν το μοντέλο πελάτη-διακομιστή (client-server). Ξεκινήσαμε με μια εισαγωγή στο προγραμματιστικό πλαίσιο Twisted, εξετάζοντας αρχικά τη δομή της κλάσης Echo και στη συνέχεια την κλάση Peer, η οποία μπορεί να ενσωματώσει λειτουργικότητα πελάτη ή/και διακομιστή, επιλέγοντας κατά περίπτωση δυναμικά με βάση τις παραμέτρους εισόδου. Επεκτείνοντας το παράδειγμα της κλάσης Peer, μπορεί κανείς εύκολα να δημιουργήσει πιο πλούσια λειτουργικότητα, όπως μηχανισμούς πολυεκπομπής (Άσκηση 2). Είδαμε, τέλος, ένα παράδειγμα προγραμματισμού της πλευράς του πελάτη στο κατανεμημένο σύστημα Cassandra, ένα δημοφιλές κατανεμημένο σύστημα αποθήκευσης ανοικτού πηγαίου κώδικα.

## Βιβλιογραφία/Αναφορές


Birrell, A. D., & Nelson, B. J. (1983). Implementing Remote procedure calls. *ACM SIGOPS Operating Systems Review*, 17(5), 3.

Coulouris, G., Dollimore, J., & Kindberg, T. (2005). *Distributed systems: Concepts and Design* (4th Edition). Addison-Wesley Longman, Inc.

Koromilas, L., & Magoutis, K. (2011). CassMail: A scalable, highly-available, and rapidly-prototyped E-mail service. In *IFIP Distributed Applications and Interoperable Systems (DAIS 2011)* (pp. 278-291). Heidelberg: Springer.

Lakshman, A., & Malik, P. (2010). Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, *44*(2), 35–40.

McKellar, J. (2013). Twisted Python: The engine of your Internet. Retrieved from http://radar.oreilly.com/2013/04/twisted-python-the-engine-of-your-internet.html

pycassa 1.11.0 Documentation. (2015). Retrieved from https://pycassa.github.io/pycassa/

Twisted Matrix Labs. (2015). Retrieved from https://twistedmatrix.com

Wayner, P. (2015, March 30). 7 reasons why frameworks are the new programming

languages. *InfoWorld*. Retrieved from http://www.infoworld.com/article/2902242

## Κριτήρια αξιολόγησης

### Κριτήριο αξιολόγησης 1 (Βαθμός δυσκολίας: ●●●)

**Ο αναγνώστης καλείται να ξεκινήσει μια εκτέλεση εξυπηρετητή Peer σε ένα από τα τερματικά παράθυρα και μια εκτέλεση πελάτη Peer σε ένα άλλο. Η εκκίνηση ενός ακόμα πελάτη αναδεικνύει την ανάγκη διαχείρισης περισσότερων της μιας συνδέσεων στην πλευρά του εξυπηρετητή, πρόβλημα το οποίο καλείστε να επιλύσετε.**

**Απάντηση/Λύση**

### Κριτήριο αξιολόγησης 2 (Βαθμός δυσκολίας: ●●●)

**Η άσκηση αυτή σας ζητά να υλοποιήσετε ένα μηχανισμό πολυεκπομπής (multicasting) μεταξύ ενός αποστολέα και περισσοτέρων του ενός παραληπτών με χρήση του twisted. Η πολυεκπομπή είναι μια μορφή ομαδικής επικοινωνίας κατά την οποία κάθε μήνυμα που στέλνεται από τον αποστολέα λαμβάνεται από όλα τα μέλη της ομάδας των παραληπτών. Ο αποστολέας γνωρίζει εκ των προτέρων τον αριθμό των παραληπτών (έστω N) και περιμένει τις αιτήσεις συνδέσεών τους. Μετά τη δημιουργία των N συνδέσεων**

**προχωρά στην αποστολή ενός αριθμού μηνυμάτων, ενδεχομένως με κάποια χρονική καθυστέρηση μεταξύ τους, προς την ομάδα των παραληπτών.**

**Απάντηση/Λύση**

### Κριτήριο αξιολόγησης 3 (Βαθμός δυσκολίας: ●●●)

**Εγκαταστήστε το Cassandra στο μηχάνημά σας, εκκινήστε το στην ελάχιστή του μορφή (με έναν κόμβο) και σε διαφορετικό τερματικό εκτελέστε το παράδειγμα της Εικόνας II.10. Έχοντας ξεκινήσει τον SMTP server, πειραματιστείτε με το συνολικό σύστημα, για παράδειγμα χρησιμοποιώντας ένα benchmark, όπως το Postal** (http://doc.coker.com.au/projects/postal/).

**Απάντηση/Λύση**

# ΚΕΦΑΛΑΙΟ 12

# Τεχνολογικές τάσεις και μελλοντικές εξελίξεις

*Σύνοψη*

*Στο κεφάλαιο αυτό εξετάζουμε τις τεχνολογικές τάσεις και επερχόμενες εξελίξεις στο χώρο του προγραμματισμού (κατανεμημένων και μη) συστημάτων με γλώσσες υψηλού επιπέδου.*

*Προαπαιτούμενη γνώση*

*Η κατανόηση των εννοιών και παραδειγμάτων αυτού του κεφαλαίου απαιτεί κατανόηση της ύλης που παρουσιάστηκε στα Κεφάλαια I-II του παρόντος συγγράμματος.*

## 12.1 Εισαγωγή

Σε αυτό το κεφάλαιο εξετάζουμε τις τεχνολογικές τάσεις και τις επερχόμενες εξελίξεις στο χώρο του προγραμματισμού (κατανεμημένων και μη) συστημάτων με γλώσσες υψηλού επιπέδου, επικεντρώνοντας σε τρία κεντρικά θέματα: Την αυξητική τάση στη χρήση γλωσσών υψηλού επιπέδου (όπως η Java, C++, και Python, αντί των γλωσσών χαμηλού επιπέδου, όπως η C) για τον προγραμματισμό συστημάτων, την τάση της προγραμματιστικής κοινότητας να χρησιμοποιεί προγραμματιστικά πλαίσια (frameworks) για τη γρήγορη παραγωγή πρωτότυπου κώδικα, και τέλος τις επερχόμενες εξελίξεις στη γλώσσα Python.

## I2.2 Η χρήση γλωσσών υψηλού επιπέδου στον προγραμματισμό συστημάτων

Εδώ και περίπου δύο δεκαετίες, η χρήση γλωσσών υψηλού επιπέδου για τις ανάγκες προγραμματισμού επιπέδου συστημάτων (κατανεμημένη επεξεργασία, δικτυακή επικοινωνία, αποθήκευση δεδομένων σε μεγάλη κλίμακα, κλπ.) έχει γίνει αποδεκτή από την παγκόσμια κοινότητα. Δειγματολειπτική μελέτη των projects ανοικτού πηγαίου κώδικα (open source) της κοινότητας Apache (Apache Software Foundation, 2015) (ίσως της σημαντικότερης στο χώρο) καθώς και η εμπειρία μεγάλων εταιριών λογισμικού όπως η Google και η Yahoo, αποδεικνύουν ότι γλώσσες υψηλού επιπέδου έχουν τύχει σημαντικής αποδοχής και χρήσης. Τα συγκριτικά πλεονεκτήματα των γλωσσών αυτών σε σχέση με γλώσσες χαμηλού επιπέδου, όπως η C ή η γλώσσα μηχανής, εστιάζονται, κυρίως, στην ευκολία παραγωγής πρωτότυπου κώδικα καθώς και τη διαχειρισιμότητα κατά τον πλήρη κύκλο ζωής ενός project λογισμικού.

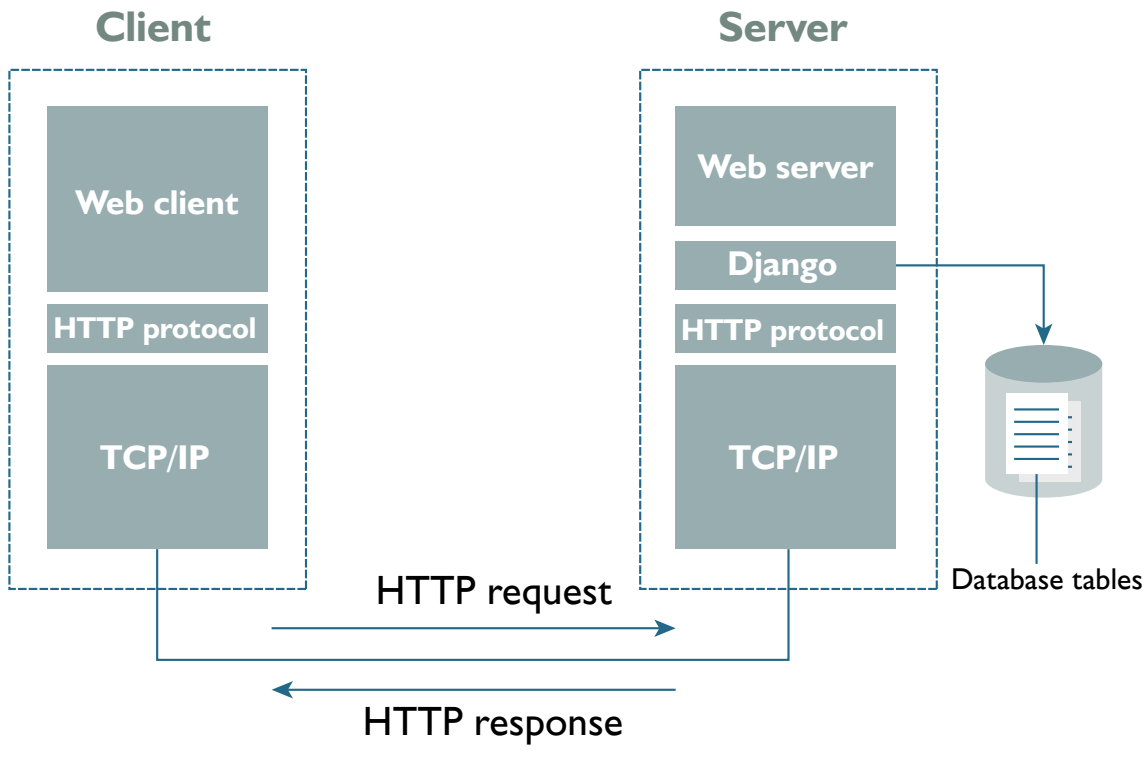
Ο αρχικός σκεπτικισμός της κοινότητας των προγραμματιστών συστημάτων να χρησιμοποιήσουν γλώσσες υψηλού επιπέδου λόγω ελλιπούς υποστήριξης μηχανισμών υψηλής απόδοσης, έχει προοδευτικά αντιμετωπιστεί με την εισαγωγή νέων και εξειδικευμένων διεπαφών προγραμματιστή-εφαρμογής (APIs), όπως το πλαίσιο Twisted που είδαμε στο Κεφάλαιο II, βελτιωμένων εικονικών μηχανών εκτέλεσης προγραμμάτων, καλύτερης ολοκλήρωσης με το λειτουργικό σύστημα, κλπ.

Πρόσφατα, η χρήση της Python έχει επεκταθεί και στη δημιουργία εξελιγμένων κατανεμημένων συστημάτων, όπως η υλοποίηση του συστήματος κατανεμημένης συμφωνίας (consensus), ομοιοτυπίας (replication) δεδομένων, και κατανεμημένου συντονισμού (distributed coordination), OpenReplica (Altinbuken & Sirer, 2012). Το OpenReplica υλοποιεί κατανεμημένα αντικείμενα συντονισμού (coordination objects), ανεκτικά στα σφάλματα (fault tolerant), μέσω των οποίων οι κατανεμημένες εφαρμογές μπορούν να συγχρονίσουν τις δραστηριότητές τους. Η τάση της υλοποίησης προχωρημένης λειτουργικότητας, όπως στην περίπτωση του

OpenReplica, αναμένεται να συνεχιστεί και να επεκταθεί με νέα παραδείγματα τέτοιων συστημάτων στο μέλλον.

## 12.3 Προγραμματιστικά πλαίσια

Μια σημαντική τάση στο μοντέρνο προγραμματισμό με γλώσσες υψηλού επιπέδου είναι η έννοια του *προγραμματιστικού πλαισίου* που είδαμε στο Κεφάλαιο 11. Όπως αναφέραμε εκεί, ένα προγραμματιστικό πλαίσιο στοχεύει στην απλοποίηση του προγραμματισμού και της γρήγορης παραγωγής πρότυπου κώδικα. Τα προγραμματιστικά πλαίσια έχουν τύχει σημαντικής αποδοχής, με αποτέλεσμα έμπειροι προγραμματιστές και ερευνητές να αναρωτιούνται αν *«τα προγραμματιστικά πλαίσια είναι οι νέες γλώσσες προγραμματισμού»* (Wayner, 2015). Εκτός του Twisted που είδαμε σε λεπτομέρεια στο Κεφάλαιο 11, στη συνέχεια, ως ένδειξη των τάσεων που επικρατούν σε αυτόν το χώρο, θα εξετάσουμε συνοπτικά ένα ακόμα παράδειγμα προγραμματισμού κατανεμημένων συστημάτων με χρήση προγραμματιστικών πλαισίων, τον προγραμματισμό *εφαρμογών παγκοσμίου ιστού* (World Wide Web ή εν συντομία Web).



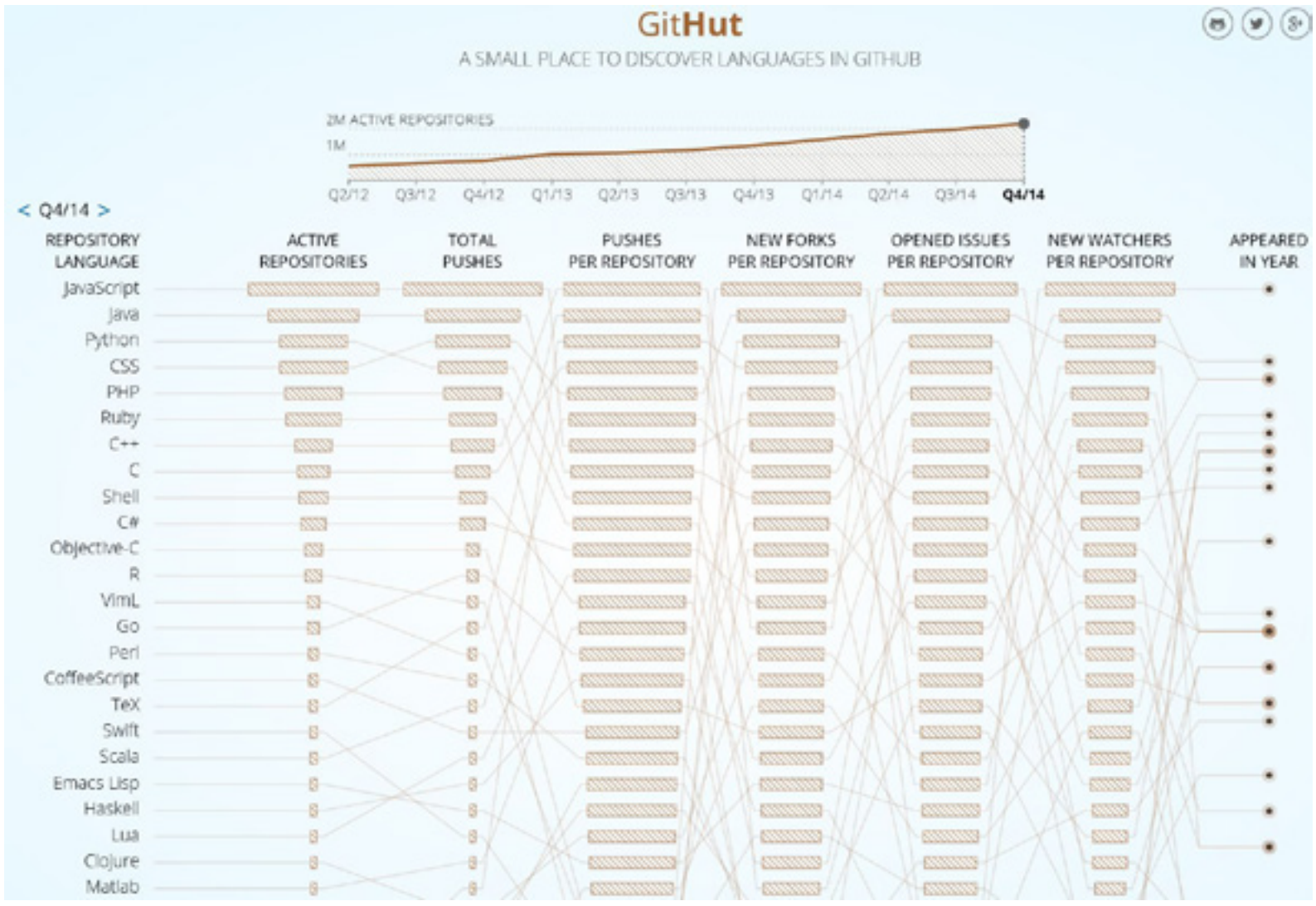

**Εικόνα 12.1** *Αλληλεπίδραση μεταξύ Web client και Web server υλοποιημένου με το πλαίσιο Django*

Ο παγκόσμιος ιστός (World Wide Web ή Web) αποτελεί ένα κατανεμημένο σύστημα πλανητικής κλίμακας του οποίου η δημιουργία και εξέλιξη βασίστηκε στο πρωτόκολλο επικοινωνίας Hypertext Transfer Protocol ή HTTP (Kurose & Ross, 2005). Ο προγραμματισμός της επιθυμητής λειτουργικότητας εφαρμογών Web είναι γενικά μια περίπλοκη διαδικασία, η οποία μπορεί να απλοποιηθεί σημαντικά με χρήση του πλαίσιου προγραμματισμού Django (Django framework, 2015). Το πλαίσιο Django υποθέτει ότι η Web εφαρμογή χρησιμοποιεί και αποθηκεύει δεδομένα σε μια σχεσιακή βάση δεδομένων στην πλευρά του διακομιστή (Εικόνα 12.1). Αναλαμβάνει, λοιπόν, πλήρως την αλληλεπίδραση μεταξύ της εφαρμογής και της βάσης δεδομένων καθώς και τη διαχείριση συνδέσεων πάνω από το πρωτόκολλο HTTP. Ο προγραμματιστής αρκεί να ορίσει τα δεδομένα που χρησιμοποιεί, τη λογική με την οποία τα διαχειρίζεται, και το πώς τα δεδομένα αυτά θα απεικονιστούν στο σελιδομετρητή (browser) του πελάτη, επιτυγχάνοντας την παραγωγή ολοκληρωμένων Web εφαρμογών σε σύντομο χρονικό διάστημα. Το Django, μαζί με το Twisted, δίνουν μια καλή εικόνα των μοντέρνων τάσεων στη χρήση προγραμματιστικών πλαισίων στο χώρο της Python.

## 12.4 Η εξέλιξη της Python

Η Python, όπως και κάθε γλώσσα προγραμματισμού, εξελίσσεται συνεχώς από την εμφάνισή της το 1991. Ποσοτική ανάλυση 2,2 εκατομμυρίων projects ανοικτού λογισμικού στο αποθετήριο GitHub (GitHut, 2015) αναφέρει την Python ως μια από τις τρείς πιο δημοφιλείς γλώσσες προγραμματισμού (Εικόνα 12.2). Η έκδοση 2, στην οποία βασίζεται το παρόν σύγγραμμα (για την ακρίβεια η έκδοση 2.7.8), είναι επί του παρόντος το de facto standard της γλώσσας, κυρίως λόγω της χρήσης της στη μεγάλη πλειοψηφία των πακέτων, βιβλιοθηκών, και πλαισίων, συμπεριλαμβανομένων και των Twisted και Django που εξετάσαμε σε αυτό το σύγγραμμα. Ωστόσο από το 2008, η κοινότητα της Python έχει ξεκινήσει την ανάπτυξη της επόμενης κύριας έκδοσης της γλώσσας, της Python 3. Αν και η Python 3 εισάγει μικρές αλλαγές στη γλώσσα (Van Rossum, 2015), έχει συναντήσει αργή αποδοχή από την

κοινότητα λόγω της μη-συμβατότητας προγραμμάτων Python 3 με πακέτα και βιβλιοθήκες ανεπτυγμένων σε Python 2. Η έκδοση 2 της Python είναι, λοιπόν, ακόμα, και θα είναι μεσοπρόθεσμα, σε ευρεία χρήση, ωστόσο το γεγονός ότι η μελλοντική ανάπτυξη της γλώσσας εστιάζει στην έκδοση 3, σημαίνει ότι εν καιρώ –ιδιαίτερα όταν η μεγάλη πλειοψηφία των δημοφιλών πακέτων την υποστηρίξουν– η έκδοση 3 θα γίνει η κύρια επιλογή των προγραμματιστών της Python.




**Εικόνα I2.2** *Ανάλυση 2,2 εκατομμυρίων projects ανοικτού λογισμικού στο αποθετήριο GitHub (GitHut, 2015)*

## I2.4 Επίλογος

Στο παρόν κεφάλαιο εξετάσαμε τρεις μοντέρνες τάσεις στο χώρο του προγραμματισμού με γλώσσες υψηλού επιπέδου όπως η Python. Στα επόμενα χρόνια αναμένουμε ότι η χρήση τέτοιων γλωσσών, ιδιαίτερα για την παραγωγή (κατανεμημένου και μη) λογισμικού, θα αυξηθεί, όπως και η χρήση προγραμματιστικών πλαισίων για τη γρήγορη παραγωγή πρωτότυπου κώδικα. Αν και η έκδοση 2 της γλώσσας Python είναι σήμερα το de facto standard, λόγω της μεγάλης έκτασης του λογισμικού που έχει παραχθεί σε αυτήν, είναι αναμενόμενο ότι μεσοπρόθεσμα η έκδοση 3 θα γίνει αποδεκτή ως η κύρια επιλογή των προγραμματιστών της Python.

## Βιβλιογραφία/Αναφορές


Altinbuken, D., & Sirer, E. G. (2012). *Commodifying Replicated State Machines with OpenReplica* (Computing and Information Science Technical Report 1813-29009). Ithaca, NY: Cornell University.

Apache Software Foundation. (2015). Retrieved January 1, 2015, from http://www.apache.org/

Django framework. (2015). Retrieved from https://www.djangoproject.com/

GitHut. (2015). GitHut: A small place to discover languages in GitHub. Retrieved from http://githut.info/

Kurose, J. F., & Ross, K. W. (2005). *Computer Networking A Top-Down Approach Featuring the Internet* (Third Edition). Pearson Education, Inc.

Van Rossum, G. (2015). What's new in Python 3.0. Retrieved from https://docs.python.org/3/whatsnew/3.0.html

Wayner, P. (2015, March 30). 7 reasons why frameworks are the new programming languages. *InfoWorld*. Retrieved from http://www.infoworld.com/article/2902242