ΝΙΚΟΛΑΟΣ Α. ΑΓΓΕΛΙΔΑΚΗΣ

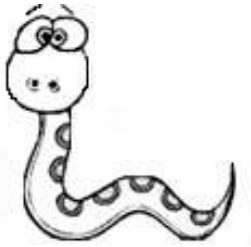
Εισαγωγή στον προγραμματισμό με την Python

>>> print('Hello, World!')

Εισαγωγή

στον προγραμματισμό

με την Python



Νικόλαος Α. Αγγελιδάκης

Εκπαιδευτικός Πληροφορικής,
Μ.Δ.Ε. (M.Sc.) στην Επιστήμη Υπολογιστών

Α' ΕΚΔΟΣΗ

Ηράκλειο, Αύγουστος 2015

Εισαγωγή στον προγραμματισμό με την Python

Συγγραφέας: Νικόλαος Α. Αγγελιδάκης, εκπαιδευτικός Πληροφορικής

Γλωσσική επιμέλεια: Ασημένια Χαρκιωτάκη, φιλόλογος

1η Έκδοση

Ηράκλειο, Αύγουστος 2015

ISBN: 978-960-93-7364-7

Δικτυακός τόπος βιβλίου: http://aggelid.mysch.gr/pythonbook

Πρόλογος

Το βιβλίο αυτό απευθύνεται σε αρχάριους που θέλουν να μάθουν προγραμματισμό εύκολα και σχετικά γρήγορα. Η **Python** είναι μια γλώσσα προγραμματισμού γενικής χρήσης, πάρα πολύ υψηλού επιπέδου, απλή και εύκολη στην εκμάθηση, ισχυρή, δυναμική, αποδοτική, παραγωγική και επεκτάσιμη. Είναι κατάλληλη και για αρχάριους και για έμπειρους προγραμματιστές. Μπορεί να χρησιμοποιηθεί τόσο για εκπαιδευτικούς σκοπούς όσο και για την ανάπτυξη ολοκληρωμένων εφαρμογών.

Τα παραδείγματα αναπτύχθηκαν στην έκδοση **3.4.3** της Python και σε περιβάλλον **Windows**. Γενικότερα, η Python μπορεί να χρησιμοποιηθεί για τη γρήγορη ανάπτυξη ολοκληρωμένων εφαρμογών σε διάφορες περιοχές ενδιαφέροντος (διαχείριση συστήματος υπολογιστή, ανάπτυξη εφαρμογών Διαδικτύου, επεξεργασία αρχείων κειμένου, επιστημονικές εφαρμογές, εκπαίδευση, ανάπτυξη παιχνιδιών, κ.λπ.) και στις περισσότερες πλατφόρμες υλικού υπολογιστών και Λειτουργικών Συστημάτων (Windows, Unix, Linux, Mac OS X, κ.λπ.). Διαθέτει πληθώρα έτοιμων βιβλιοθηκών που είναι δυνατό να χρησιμοποιηθούν εύκολα και άμεσα. Οι βιβλιοθήκες μπορούν να επεκταθούν με νέα τμήματα γραμμένα σε C ή C++. Τα προγράμματα σε Python είναι συμπαγή, ευανάγνωστα, και γράφονται και συντηρούνται γρηγορότερα σε σχέση με άλλες δημοφιλείς γλώσσες προγραμματισμού όπως οι C, C++ και Java.

Περιεχόμενα σελ.

Κεφάλαιο 1

Εισαγωγή στον προγραμματισμό

Ο υπολογιστής αποτελεί μια εξαιρετική μηχανή επεξεργασίας δεδομένων και αντιμετώπισης υπολογιστικών προβλημάτων. Για να επιτευχθεί αυτό, θα πρέπει να του δώσουμε σαφείς εντολές με τη μορφή των προγραμμάτων. Η διαδικασία συγγραφής προγραμμάτων ονομάζεται προγραμματισμός και περιλαμβάνει τη διατύπωση των κατάλληλων εντολών προς τον υπολογιστή με τη χρήση τεχνητών γλωσσών, των γλωσσών προγραμματισμού.

1.1 Πρόβλημα και υπολογιστής

Ως **πρόβλημα** θεωρούμε κάθε ζήτημα που τίθεται προς επίλυση, κάθε κατάσταση που μας απασχολεί και πρέπει να αντιμετωπιστεί. Καθημερινά αντιμετωπίζουμε διαφόρων ειδών προβλήματα, σε κάποια μπορούμε να βρούμε εύκολα τη λύση, ενώ σε άλλα συνθετότερα δυσκολευόμαστε ή και μερικές φορές αδυνατούμε να προσδιορίσουμε κάποια λύση.

Ο **υπολογιστής** (επιτραπέζιος, φορητός, ταμπλέτα, έξυπνο κινητό, έξυπνη τηλεόραση, κ.ο.κ.) αποτελεί πλέον αναπόσπαστο κομμάτι κάθε καθημερινής ανθρώπινης δραστηριότητας (εργασία, επικοινωνία, ενημέρωση, ψυχαγωγία, κ.ά.) και στην ουσία μάς βοηθάει στην επίλυση προβλημάτων. Η χρήση του υπολογιστή είναι ιδιαίτερα σημαντική στις περιπτώσεις κατά τις οποίες τα προβλήματα που αντιμετωπίζουμε έχουν πολλά δεδομένα και ζητούμενα, ή η μέθοδος επίλυσης είναι πολύπλοκη ή επαναλαμβάνεται πολλές φορές.

Ο υπολογιστής μάς προσφέρει μεγάλο αποθηκευτικό χώρο δεδομένων (αριθμών, κειμένων, εικόνων, ήχου, βίντεο), και εκτελεί υπολογισμούς και επεξεργάζεται δεδομένα ταχύτερα από τον άνθρωπο. Μπορεί να εκτελέσει μια

λογική σειρά εντολών με συνέπεια, πειθαρχία και για όσες φορές χρειαστεί. Οι εντολές δίνονται στον υπολογιστή με τη μορφή προγραμμάτων.

Ο υπολογιστής αποτελεί μια μηχανή επεξεργασίας δεδομένων και αντιμετώπισης **υπολογιστικών προβλημάτων**. Τα υπολογιστικά προβλήματα απαιτούν για την επίλυσή τους λογικές σκέψεις και μαθηματικές πράξεις. Εισάγουμε στον υπολογιστή τα απαραίτητα δεδομένα με τη βοήθεια διάφορων συσκευών εισόδου, τα οποία και επεξεργάζεται σύμφωνα με τις εντολές που του δίνουμε. Μετά την επεξεργασία, στις συσκευές εξόδου, παίρνουμε χρήσιμες πληροφορίες και απαντήσεις σχετικές με τη λύση των προβλημάτων μας.

Για να επιλύσουμε με αποτελεσματικότητα ένα δύσκολο και σύνθετο πρόβλημα, με ή χωρίς τη βοήθεια υπολογιστή, καλό είναι να ακολουθήσουμε τα παρακάτω βήματα:

- Να το κατανοήσουμε πλήρως: το περιεχόμενό του, τα δεδομένα που δίνονται, τα ζητούμενα που απαιτούνται.
- Να το αναλύσουμε σε επιμέρους απλούστερα υπο-προβλήματα που επιλύονται ευκολότερα.
- Να εκτελέσουμε οργανωμένα βήματα επίλυσης.

Η Επιστήμη των Υπολογιστών (Computer Science) αφορά κυρίως στην υπολογιστική επίλυση προβλημάτων.

1.2 Αλγόριθμος και πρόγραμμα

Αλγόριθμο ονομάζουμε κάθε πεπερασμένη και αυστηρά καθορισμένη σειρά βημάτων (οδηγιών) για την επίλυση ενός προβλήματος. Τα βήματα που ακολουθούμε, για παράδειγμα, κατά την εκτέλεση μιας μαγειρικής συνταγής ή την επίλυση ενός μαθηματικού προβλήματος αποτελούν έναν αλγόριθμο. Όταν ακολουθήσουμε τα βήματα ενός αλγορίθμου, θα πρέπει στο τέλος να προκύψει ένα αποτέλεσμα. Επίσης, ένας αλγόριθμος θα πρέπει κάποτε να τελειώνει.

Η αναπαράσταση ενός αλγορίθμου μπορεί να γίνει με χρήση της φυσικής γλώσσας, με διαγραμματικές τεχνικές ή με κωδικοποίηση σε ειδική (τεχνητή) γλώσσα. Ένα **πρόγραμμα** αποτελεί έναν αλγόριθμο γραμμένο σε γλώσσα

κατανοητή για τον υπολογιστή και περιέχει εντολές (οδηγίες) που κατευθύνουν με κάθε λεπτομέρεια τον υπολογιστή, για να εκτελέσει μια συγκεκριμένη εργασία και να επιλύσει ένα πρόβλημα.

1.3 Γλώσσες προγραμματισμού

Οι εντολές των προγραμμάτων γράφονται από τους προγραμματιστές σε τεχνητές γλώσσες που ονομάζονται **γλώσσες προγραμματισμού**. Μια γλώσσα προγραμματισμού θα πρέπει να έχει αυστηρά ορισμένη σύνταξη και σημασιολογία. Η σύνταξη καθορίζει αν μια σειρά από σύμβολα αποτελούν «νόμιμες» εντολές ενός προγράμματος γραμμένου σε μια συγκεκριμένη γλώσσα προγραμματισμού και η σημασιολογία καθορίζει τη σημασία του προγράμματος, δηλαδή τις υπολογιστικές διαδικασίες που υλοποιεί.

Οι γλώσσες προγραμματισμού θα πρέπει να είναι ταυτόχρονα κατανοητές τόσο από τους ανθρώπους που θα πρέπει να διατυπώσουν σε αυτές έναν αλγόριθμο, όσο και από τους υπολογιστές που θα πρέπει να εκτελέσουν τις εντολές των προγραμμάτων που είναι γραμμένα σε αυτές. Αν χρησιμοποιούσαμε τη φυσική γλώσσα, τότε αυτή θα ήταν κατανοητή μόνο από τους ανθρώπους. Αν χρησιμοποιούσαμε τη γλώσσα μηχανής (το πρόγραμμα γράφεται ως ακολουθία δυαδικών ψηφίων 0,1), τότε αυτή θα ήταν κατανοητή μόνο από τους υπολογιστές. Η προφανής λύση του παραπάνω προβλήματος είναι η δημιουργία γλωσσών προγραμματισμού με συμβολισμούς κατανοητούς από έναν άνθρωπο και ειδικών μεταφραστικών προγραμμάτων που μετατρέπουν τις εντολές των προγραμμάτων σε γλώσσα κατανοητή από έναν υπολογιστή.

Η ανάγκη για ευκολότερη συγγραφή, διόρθωση και συντήρηση προγραμμάτων οδήγησε στη δημιουργία των **γλωσσών υψηλού επιπέδου**. Οι γλώσσες υψηλού επιπέδου μοιάζουν με τη φυσική μας γλώσσα και διαθέτουν δικό τους αλφάβητο, λεξιλόγιο και συντακτικό. Μερικές από τις πιο δημοφιλείς γλώσσες είναι η C, C++, C#, Java, PHP, Javascript, Perl, Lisp, Basic, Ruby και η **Python**. Ανάλογα με το είδος του προγράμματος που θέλουμε να αναπτύξουμε (για γενική χρήση, για επιστημονική χρήση, για χρήση στον Παγκόσμιο Ιστό, κ.λπ.) επιλέγουμε και την κατάλληλη γλώσσα προγραμματισμού.

1.4 Εργαλεία προγραμματισμού

Προγραμματισμό ονομάζουμε την εργασία σύνταξης ενός προγράμματος. Πρόκειται για μια εξαιρετικά δημιουργική δραστηριότητα, για την οποία απαιτούνται τα παρακάτω εργαλεία:

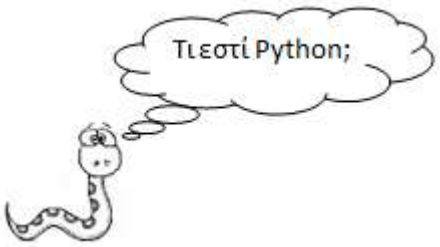
- ένας συντάκτης κειμένων (**editor**) με τον οποίο γράφεται το αρχικό πρόγραμμα, που ονομάζεται **πηγαίο πρόγραμμα** ή κώδικας (**source code**).
- ένα μεταφραστικό πρόγραμμα, **μεταγλωττιστή** ή **διερμηνευτή**, το οποίο μεταφράζει το πηγαίο πρόγραμμα σε **αντικείμενο πρόγραμμα** ή κώδικα (**object code**). Το μεταφραστικό πρόγραμμα ελέγχει το πηγαίο πρόγραμμα για συντακτικά λάθη (λάθη στο αλφάβητο, στο λεξιλόγιο, στο συντακτικό), εμφανίζει κατάλληλα διαγνωστικά μηνύματα, εάν βρεθούν λάθη, και μόνο αν δεν υπάρχουν λάθη παράγεται το αντικείμενο πρόγραμμα. Το αντικείμενο πρόγραμμα είναι σε γλώσσα μηχανής, αλλά δεν είναι ακόμη εκτελέσιμο από τον υπολογιστή, και πρέπει να περάσει από κάποιες άλλες διαδικασίες. Ο μεταγλωττιστής ελέγχει όλο το πρόγραμμα συνολικά, ενώ ο διερμηνευτής μία εντολή κάθε φορά, την εκτελεί και έπειτα ελέγχει την επόμενη εντολή.
- ένα ειδικό πρόγραμμα που ονομάζεται **συνδέτης** (**linker**), το οποίο πολλές φορές συνδέει το αντικείμενο πρόγραμμα ή ένα σύνολο από αντικείμενα προγράμματα με έτοιμα υποπρογράμματα της βιβλιοθήκης της γλώσσας προγραμματισμού ή του προγραμματιστή. Το τελικό πρόγραμμα που παράγεται είναι το **εκτελέσιμο πρόγραμμα** ή κώδικας (**executable code**), είναι διατυπωμένο σε γλώσσα μηχανής και μπορεί να εκτελεστεί άμεσα από τον επεξεργαστή του υπολογιστή.
- **εργαλεία εντοπισμού λαθών** (**debuggers**) με τα οποία ο προγραμματιστής παρακολουθεί τι ακριβώς συμβαίνει στο παρασκήνιο κατά την εκτέλεση ενός προγράμματος, ώστε να βρει και να διορθώσει τυχόν λάθη.

Οι προγραμματιστές χρησιμοποιούν συνήθως ένα περιβάλλον λογισμικού που περιλαμβάνει όλα τα παραπάνω εργαλεία και γι' αυτό ονομάζεται **ολοκληρωμένο περιβάλλον ανάπτυξης εφαρμογών** (προγραμμάτων) (Integrated Development Environment, **IDE**).

Κεφάλαιο 2

Εισαγωγή στην Python

Η Python είναι μια γλώσσα προγραμματισμού γενικής χρήσης, πάρα πολύ υψηλού επιπέδου, απλή και εύκολη στην εκμάθηση, ισχυρή, δυναμική, αποδοτική, παραγωγική και επεκτάσιμη. Είναι κατάλληλη και για αρχάριους και για έμπειρους προγραμματιστές. Μπορεί να χρησιμοποιηθεί τόσο για εκπαιδευτικούς σκοπούς όσο και για την ανάπτυξη ολοκληρωμένων εφαρμογών.



Η Python διαθέτει αποδοτικές δομές δεδομένων υψηλού επιπέδου και υποστηρίζει, χωρίς να αναγκάζει, μια απλή αλλά συνάμα αρκετά αποτελεσματική προσέγγιση στον αντικειμενοστρεφή προγραμματισμό. Υποστηρίζει και άλλες γνωστές προγραμματιστικές προσεγγίσεις, όπως είναι ο διαδικαστικός και ο συναρτησιακός προγραμματισμός. Η σύνταξή της είναι κομψή και οι τύποι της δυναμικοί. Είναι διερμηνευόμενη (interpreted) γλώσσα προγραμματισμού και μπορεί να χρησιμοποιηθεί τόσο για τη δημιουργία σεναρίων εντολών όσο και για τη γρήγορη ανάπτυξη ολοκληρωμένων εφαρμογών σε διάφορες περιοχές ενδιαφέροντος (διαχείριση συστήματος υπολογιστή, ανάπτυξη εφαρμογών Διαδικτύου, επεξεργασία αρχείων κειμένου, επιστημονικές εφαρμογές, εκπαίδευση, ανάπτυξη παιχνιδιών, κ.λπ.) και στις περισσότερες πλατφόρμες υλικού υπολογιστών και Λειτουργικών Συστημάτων (Windows, Unix, Linux, Mac OS X, κ.λπ.). Διαθέτει πληθώρα έτοιμων βιβλιοθηκών που μπορούν να χρησιμοποιηθούν εύκολα και άμεσα. Οι βιβλιοθήκες μπορούν να επεκταθούν με νέα τμήματα γραμμένα σε C ή C++. Τα προγράμματα σε Python είναι συμπαγή, ευανάγνωστα, και γράφονται και συντηρούνται γρηγορότερα σε σχέση με άλλες δημοφιλείς γλώσσες προγραμματισμού όπως οι C, C++ και Java. Ο κώδικας

μπορεί να ομαδοποιηθεί σε αρθρώματα (modules) και πακέτα (packages). Οι οπαδοί της Python χρησιμοποιούν μάλιστα γι' αυτή τη γλώσσα τη φράση «οι μπαταρίες περιλαμβάνονται».

Αναπτύσσεται ως λογισμικό ανοικτού κώδικα με βάση το μοντέλο της κοινότητας προγραμματιστών που εργάζονται για την ανάπτυξή της, και ο συντονισμός και η διαχείριση γίνεται από τον μη κερδοσκοπικό οργανισμό Python Software Foundation (PSF) (`www.python.org/psf`). Ως αποστολή του PSF αναφέρεται η ανάδειξη, η προστασία και η προαγωγή της Python, καθώς και η υποστήριξη της ανάπτυξης μιας ποικίλης και διεθνούς κλίμακας κοινότητας προγραμματιστών της Python. Η κοινότητα αυτή αυξάνεται καθημερινά και η Python βρίσκεται σε υψηλή θέση στη λίστα των δημοφιλέστερων γλωσσών προγραμματισμού. Επίσης, χρησιμοποιείται από μεγάλες εταιρείες και οργανισμούς του χώρου της Πληροφορικής και όχι μόνο.

Πέρα όμως από την πληθώρα των θετικών χαρακτηριστικών της Python, οφείλουμε να αναφέρουμε και μειονεκτήματα. Ο χρόνος εκτέλεσης των προγραμμάτων της Python μπορεί να μην είναι πάντα τόσο γρήγορος όσο είναι στις μεταγλωττιζόμενες (compiled) γλώσσες όπως η C και η C++. Αυτό οφείλεται στο ότι ένα πρόγραμμα σε Python δεν μεταγλωττίζεται σε δυαδικό κώδικα μηχανής που εκτελείται άμεσα από τον επεξεργαστή του υπολογιστή. Το μειονέκτημα αυτό αντισταθμίζεται από το ότι πολλές φορές είναι σημαντικότερη η εξοικονόμηση χρόνου που έχουμε κατά την ανάπτυξη μιας εφαρμογής σε Python. Θα μιλήσουνε αργότερα για το πώς η Python εκτελεί ένα πρόγραμμα. Επιπλέον, η Python δεν είναι και τόσο αποδοτική, ώστε να δημιουργήσουμε, για παράδειγμα, ένα νέο Λειτουργικό Σύστημα.

Εικόνα 2.1: Το λογότυπο της Python (http://www.python.org)

2.1 Ιστορία της Python

Η δημιουργία της Python ξεκίνησε το 1989 από τον Ολλανδό Guido van Rossum στο ερευνητικό κέντρο Centrum Wiskunde & Informatica (CWI), το οποίο επιτελεί βασική έρευνα στο πεδίο των Μαθηματικών και της Επιστήμης Υπολογιστών. Η Python θεωρείται διάδοχος της γλώσσας προγραμματισμού ABC, μια και αυτή υπήρξε η βασική πηγή έμπνευσης για τη δημιουργία της. Αρχικά χρησιμοποιήθηκε ως γλώσσα σεναρίων για το κατανεμημένο λειτουργικό σύστημα Amoeba. Όπως αναφέρει ο ίδιος ο Guido van Rossum, η πιο καινοτόμα συνεισφορά του στην επιτυχία της Python ήταν το ότι αυτή είχε τη δυνατότητα να επεκτείνεται εύκολα. Το μοντέλο της Python σχεδιάστηκε με τέτοιο τρόπο ώστε να είναι ευέλικτα επεκτάσιμο, να παρέχει ενσωματωμένα στοιχεία (εντολές, τύπους αντικειμένων, κ.λπ.) αλλά και να δίνει τη δυνατότητα στους προγραμματιστές να προσθέτουν τα δικά τους στοιχεία ανάλογα με τις ανάγκες τους και το σύστημα που χρησιμοποιούν. Η πρώτη έκδοση κυκλοφόρησε το 1991. Η ονομασία της οφείλεται στη δημοφιλή κωμική σειρά Monty Python's Flying Circus του BBC της Μεγάλης Βρετανίας (δεκαετία του '70). Ο Guido van Rossum ήθελε ένα όνομα για τη νέα γλώσσα προγραμματισμού, το οποίο να είναι μικρό, μοναδικό και ελαφρώς μυστήριο. Μάλιστα ενθαρρύνεται η χρήση στην τεκμηρίωση της Python αναφορών στη σάτιρα της παραπάνω κωμικής σειράς. Δεν υπάρχει συνεπώς καμία συσχέτιση με ερπετά και το γνωστό φίδι πύθωνας.

Η Python εξελίσσεται γρήγορα μέσω νέων εκδόσεων και με βασικό εργαλείο τα Python Enhancement Proposals ("PEPs"), τα οποία είναι τυποποιημένα κείμενα που περιέχουν γενικές πληροφορίες, οδηγίες, προτάσεις και περιγραφές για τυχόν νέα χαρακτηριστικά της γλώσσας. Τη χρονική στιγμή που γραφόταν το παρόν βιβλίο υπήρχαν δύο σειρές εκδόσεων της Python, η σειρά 2.χ με πιο πρόσφατη έκδοση παραγωγής την 2.7.9 και η νεότερη σειρά 3.χ με πιο πρόσφατη έκδοση παραγωγής την 3.4.3. Η Python είναι κατά το μεγαλύτερο μέρος η ίδια και στις δύο σειρές εκδόσεων, αλλά με διαφορές σε πολλές λεπτομέρειες, κυρίως στη λειτουργία των ενσωματωμένων αντικειμένων όπως είναι τα λεξικά και οι συμβολοσειρές. Επίσης, στη νεότερη έκδοση έχουν αφαιρεθεί τα απαξιωμένα χαρακτηριστικά και έχει αναδιοργανωθεί η πρότυπη βιβλιοθήκη. Πολλές εφαρμογές τρίτων κατασκευαστών είναι γραμμένες σε εκδόσεις της σειράς 2.χ. Αν ξεκινάτε τώρα με την Python και δεν σας ενδιαφέρει

ιδιαίτερα κάποια από αυτές τις εφαρμογές, τότε καλύτερα να ασχοληθείτε με την πιο πρόσφατη έκδοση της σειράς 3.χ.

2.2 Φιλοσοφία της Python

Η φιλοσοφία της Python συνοψίζεται κυρίως στην απλότητα και την αναγνωσιμότητα του κώδικα. Οι πεποιθήσεις που χαρακτηρίζουν την Python παρατίθενται στο "The Zen of Python" από τον Tim Peters (PEP 20, The Zen of Python) και μπορούν να εμφανιστούν στην οθόνη με την παρακάτω εντολή (θα δούμε αργότερα πού και πώς γράφουμε τις εντολές):

```
>>> import this

The Zen of Python, by Tim Peters


Beautiful is better than ugly.

(Το όμορφο είναι καλύτερο από το άσχημο.)

Explicit is better than implicit.

(Το σαφές είναι καλύτερο από το υπονοούμενο.)

Simple is better than complex.

(Το απλό είναι καλύτερο από το πολύπλοκο.)

Complex is better than complicated.

(Το πολύπλοκο είναι καλύτερο από το περίπλοκο.)

Flat is better than nested.

(Το επίπεδο είναι καλύτερο από το εμφωλευμένο.)

Sparse is better than dense.

(Το αραιό είναι καλύτερο από το πυκνό.)
```

Readability counts.

(Η αναγνωσιμότητα μετράει.)

Special cases aren't special enough to break the rules.

(Οι ειδικές περιπτώσεις δεν είναι αρκετά ειδικές για να σπάσουν τους κανόνες.)

Although practicality beats purity.

(Αν και η χρηστικότητα νικάει τη γνησιότητα.)

Errors should never pass silently.

(Τα λάθη δεν πρέπει να περνάνε σιωπηλά.)

Unless explicitly silenced.

(Εκτός και αν ρητά αποσιωπούνται.)

In the face of ambiguity, refuse the temptation to guess.

(Μπροστά στην αμφιβολία, αρνηθείτε τον πειρασμό να μαντέψετε.)

There should be one-- and preferably only one --obvious way to do it.

(Πρέπει να υπάρχει ένας-- και κατά προτίμηση μόνο ένας -- προφανής τρόπος για να το κάνετε.)

Although that way may not be obvious at first unless you're Dutch.

(Αν και αυτός ο τρόπος μπορεί να μην είναι προφανής εξαρχής, εκτός και αν είστε Ολλανδός.)

Now is better than never.

(Τώρα είναι καλύτερα από ποτέ.)

Although never is often better than *right* now.

(Αν και το ποτέ είναι συχνά καλύτερο από το *πρέπον* τώρα.)

If the implementation is hard to explain, it's a bad idea.

(Αν η υλοποίηση εξηγείται δύσκολα, τότε είναι μια κακή ιδέα.)

If the implementation is easy to explain, it may be a good idea.

(Αν η υλοποίηση εξηγείται εύκολα, τότε ίσως είναι μια καλή ιδέα.)

Namespaces are one honking great idea -- let's do more of those!

(Οι χώροι ονομάτων είναι μια σπουδαία ιδέα -- ας φτιάξουμε περισσότερους απ΄ αυτούς!)

2.3 Εγκατάσταση της Python

Η Python μπορεί να χρησιμοποιηθεί σε 21 διαφορετικά Λειτουργικά Συστήματα και περιβάλλοντα. Η διαδικασία εγκατάστασης είναι απλή:

- Windows:
  1. Επισκεπτόμαστε τον ιστότοπο της Python: `www.python.org`
  2. Επιλέγουμε τον σύνδεσμο Downloads και κατεβάζουμε το νεότερο αρχείο εγκατάστασης της σειράς 3.χ για Windows. Μπορούμε να διαλέξουμε μεταξύ 32-bit και 64-bit εγκατάστασης.
  3. Εκτελούμε το αρχείο της εγκατάστασης (εικόνα 2.2).

Εικόνα 2.2: Εγκατάσταση της Python σε Windows

- Linux: Η Python πρέπει να είναι ήδη εγκατεστημένη, μάλλον όμως μια παλαιότερη έκδοση. Για να εγκαταστήσετε τη νεότερη έκδοση, μπορείτε είτε να κατεβάσετε τον πηγαίο κώδικα από τον ιστότοπο της Python και να τον μεταγλωττίσετε ή να την εγκαταστήσετε χρησιμοποιώντας τον διαχειριστή πακέτων λογισμικού (package manager) που συνοδεύει το λειτουργικό σας σύστημα.

- Mac OS X: Πρέπει να υπάρχει ήδη προεγκατεστημένη μια έκδοση της Python. Επειδή το πιθανότερο είναι να πρόκειται για μια παλαιότερη, κατεβάστε τη νεότερη έκδοση που θα βρείτε στον ιστότοπο της Python: `www.python.org/downloads/mac-osx`

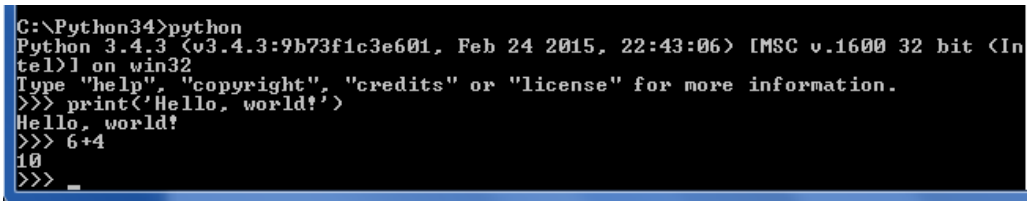
2.4 Συγγραφή και εκτέλεση προγράμματος σε Python

Η Python, όπως αναφέρθηκε και σε προηγούμενη ενότητα, είναι διερμηνευόμενη γλώσσα προγραμματισμού. Η εγκατάστασή της στον υπολογιστή μας περιλαμβάνει το λογισμικό του διερμηνευτή (interpreter) και την πρότυπη βιβλιοθήκη (library). Ο διερμηνευτής διαβάζει το πρόγραμμά μας και εκτελεί τις εντολές που περιέχει. Η βιβλιοθήκη διαθέτει έτοιμο κώδικα, με τη μορφή συλλογής αρθρωμάτων (modules), ο οποίος μπορεί να χρησιμοποιηθεί για τη διεκπεραίωση πολλών συνηθισμένων διαδικασιών που απαιτούνται στα προγράμματά μας.

Στο παρόν βιβλίο θα γράψουμε προγράμματα στο περιβάλλον των Windows και θα ασχοληθούμε με τον πρότυπο τρόπο συγγραφής και εκτέλεσης αυτών των προγραμμάτων.

Από τη σκοπιά του προγραμματιστή

Για να αρχίσουμε να γράφουμε τις πρώτες μας εντολές σε Python, μπορούμε να ανοίξουμε τη γραμμή εντολών των Windows και πληκτρολογώντας την εντολή `python` να εκκινήσουμε τον λεγόμενο φλοιό ή κονσόλα του διερμηνευτή της Python (**Python Shell**). Θα πρέπει ο φάκελος στον οποίο έχει εγκατασταθεί η Python (από προεπιλογή είναι ο `C:\Python34`) να έχει οριστεί κατάλληλα στη μεταβλητή περιβάλλοντος `Path` των Windows ή να τον έχουμε ως τρέχοντα φάκελο. Στη συνέχεια με διαδραστικό τρόπο (interactive mode) μπορούμε να δίνουμε μία μία τις εντολές μας και αυτές να εκτελούνται άμεσα από τον διερμηνευτή (εικόνα 2.3). Η κονσόλα κλείνει πατώντας `Ctrl-Z` και `Enter` ή δίνοντας την εντολή `exit()`.



Εικόνα 2.3: Python Shell

Στην προτροπή (prompt) >>> της Python γράφουμε την εντολή μας και, μόλις πατήσουμε Enter, η εντολή εκτελείται από τον διερμηνευτή και στην αμέσως παρακάτω γραμμή εμφανίζεται το αποτέλεσμα (υπάρχουν και εντολές, όπως θα δούμε σε επόμενο κεφάλαιο, που δεν εμφανίζουν αποτέλεσμα κατά την εκτέλεσή τους). Παρακάτω εμφανίζεται πάλι η προτροπή >>> και ο διερμηνευτής περιμένει την επόμενη εντολή μας. Δεν είναι δύσκολο να αντιληφθούμε ότι με την εντολή `print('Hello, world!')` τυπώνεται η φράση `Hello, world!` και με την εντολή `6+4` υπολογίζεται το άθροισμα των δύο αριθμών και τυπώνεται το αποτέλεσμα `10` (λειτουργεί ως αριθμομηχανή).

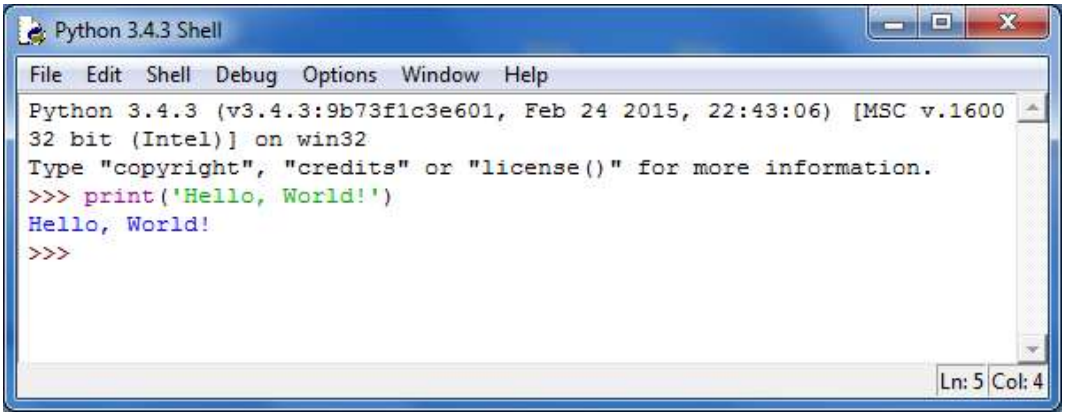
Για τη διευκόλυνση του έργου μας ως προγραμματιστές, η Python μας προσφέρει και ένα ολοκληρωμένο περιβάλλον ανάπτυξης εφαρμογών (IDE). Το περιβάλλον αυτό ονομάζεται **IDLE** (**I**ntegrated **D**eve**L**opment **E**nvironment) και μάλλον προήλθε τιμητικά από το όνομα ενός μέλους της ομάδας της κωμικής σειράς Monty Python's Flying Circus, τον Eric Idle. Το IDLE είναι ένα απλό περιβάλλον, ειδικά για αρχάριους προγραμματιστές, στο οποίο μπορούμε να γράψουμε και να εκτελέσουμε τα προγράμματά μας. Διαθέτει τα παρακάτω μέρη και χαρακτηριστικά:

- **Παράθυρο Python Shell** (κονσόλα του διερμηνευτή της Python) για διαδραστική και άμεση εκτέλεση εντολών, όπως ακριβώς αναφέρθηκε πρωτύτερα, δηλαδή με την εμφάνιση της προτροπής >>> για πληκτρολόγηση εντολής και εκτέλεσής της από τον διερμηνευτή. Τα διάφορα τμήματα μιας εντολής χρωματίζονται κατάλληλα (χρωματική συντακτική επισήμανση). Η επισήμανση αυτή βοηθάει πάρα πολύ στη σωστή συγγραφή των εντολών. Το παράθυρο αυτό υποστηρίζει και επανάκληση των προηγούμενων εντολών που έχουμε δώσει. Αυτό γίνεται, αν κάνουμε απλά κλικ με το ποντίκι σε προηγούμενη εντολή και πατήσουμε το Enter.
- **Παράθυρο συντάκτη προγράμματος (editor)**, το οποίο υποστηρίζει πολλές δυνατότητες, όπως χρωματική επισήμανση των εντολών, πολλαπλές αναιρέσεις, κατάλληλη στοίχιση των εντολών και αυτόματη συμπλήρωση εντολών. Σε ένα τέτοιο παράθυρο γράφουμε πρώτα ολόκληρο το πρόγραμμά μας (ο τρόπος αυτός ονομάζεται program mode), στη συνέχεια το αποθηκεύουμε

σε αρχείο και στο τέλος το εκτελούμε. Τα αποτελέσματα της εκτέλεσης τα βλέπουμε στο Python Shell.

- **Αποσφαλματωτή (debugger)** για παρακολούθηση της εκτέλεσης των προγραμμάτων μας με σκοπό την εύρεση σφαλμάτων.

Το IDLE εγκαθίσταται εξαρχής με την εγκατάσταση της Python στα Windows και το βρίσκουμε στη λίστα των προγραμμάτων (εφαρμογών) του υπολογιστή μας. Υπάρχουν και ολοκληρωμένα περιβάλλοντα ανάπτυξης εφαρμογών Python τρίτων κατασκευαστών, όμως η αναφορά τους είναι εκτός των στόχων του παρόντος βιβλίου. Προτείνουμε ανεπιφύλακτα να ξεκινήσετε το ταξίδι σας στην Python με το IDLE.

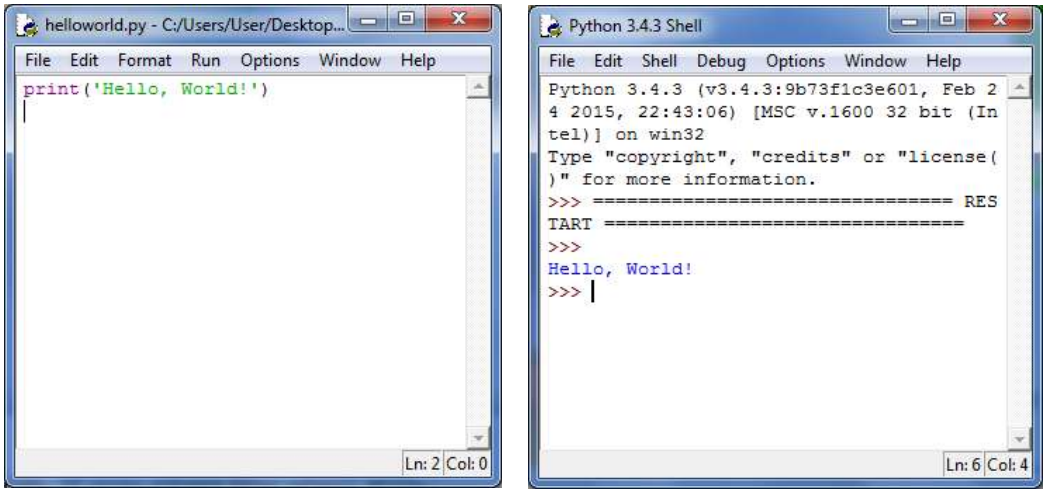


Εικόνα 2.4: Το IDLE

Στην εικόνα 2.4 βλέπουμε το IDLE και συγκεκριμένα το παράθυρο με το Python Shell. Στο πάνω μέρος βλέπουμε την έκδοση της Python που είναι εγκατεστημένη στον υπολογιστή μας. Στην προτροπή >>> δώσαμε την παραδοσιακή εντολή που γράφει ως πρώτη κάποιος που μαθαίνει μια νέα γλώσσα προγραμματισμού. Η εντολή αυτή απλά τυπώνει στην οθόνη μας τη φράση `Hello, World!`. Σύμφωνα με τον Simon Cozens η συγκεκριμένη εντολή είναι «το παραδοσιακό ξόρκι προς τους θεούς του προγραμματισμού για να μας βοηθήσουν να μάθουμε τη γλώσσα καλύτερα». Η προτροπή >>> εμφανίζεται ξανά δηλώνοντάς μας ότι η Python περιμένει την επόμενη εντολή μας.

Όμως τα προγράμματα σε Python δεν θα μας ήταν χρήσιμα, εάν έπρεπε να τα ξαναγράφουμε κάθε φορά που τα χρειαζόμασταν. Οι «σοβαρές» εφαρμογές μπορεί να αποτελούνται από χιλιάδες γραμμές εντολών. Για να γράψουμε ένα ολοκληρωμένο πρόγραμμα, επιλέγουμε στη γραμμή μενού του IDLE `File→New File`. Ένα άδειο παράθυρο εμφανίζεται με τίτλο `Untitled`. Το παράθυρο αυτό είναι ένας συντάκτης προγράμματος (editor) στον οποίο μπορούμε να γράψουμε όλο μας το πρόγραμμα, να το αποθηκεύσουμε σε αρχείο πηγαίου κώδικα και στο τέλος να το εκτελέσουμε.

Για εξάσκηση μπορούμε να γράψουμε το πρώτο μας πρόγραμμα με μοναδική εντολή την παραδοσιακή `print('Hello, World!')` και μετά να το αποθηκεύσουμε επιλέγοντας `File→Save`. Δίνουμε ένα όνομα αρχείου, για παράδειγμα `helloworld.py`, και αποθηκεύουμε σε φάκελο της αρεσκείας μας. Έπειτα, μπορούμε να το εκτελέσουμε επιλέγοντας `Run→Run Module`. Το αποτέλεσμα της εκτέλεσης εμφανίζεται στο Python Shell (εικόνα 2.5).



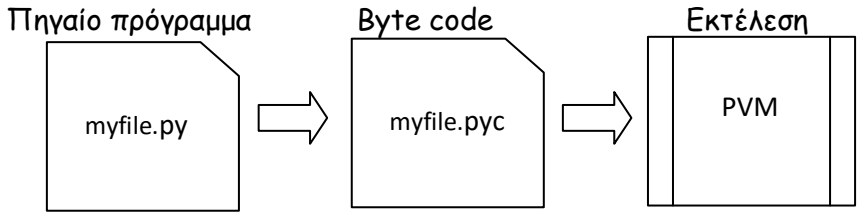
Εικόνα 2.5: Συγγραφή και εκτέλεση προγράμματος

Συχνά χρησιμοποιείται για την Python ο όρος «σενάριο» ("script") αντί «πρόγραμμα» ("program") για να περιγράψει ένα αρχείο πηγαίου κώδικα. Μάλιστα η Python χαρακτηρίζεται και ως γλώσσα σεναρίων (scripting language), κυρίως γιατί υποστηρίζει γρήγορη και ευέλικτη ανάπτυξη προγραμμάτων.

Από τη σκοπιά της

Όταν εκτελούμε ένα πρόγραμμα, η Python εσωτερικά και σχεδόν εντελώς «κρυφά» από εμάς πρώτα μεταφράζει τον πηγαίο κώδικά μας σε μία μορφή που ονομάζεται `byte code` (σχήμα 2.1). Ο `byte code` είναι μια «χαμηλού επιπέδου» και ανεξάρτητη πλατφόρμας αναπαράσταση του κώδικά μας. Δεν προορίζεται για ανάγνωση από άνθρωπο και μπορεί να «τρέξει» σε διαφορετικά συστήματα υπολογιστών χωρίς καμία αλλαγή.

Ο `byte code` εκτελείται από την PVM (Python Virtual Machine, Εικονική Μηχανή της Python), που αποτελεί αναπόσπαστο κομμάτι του συστήματος της Python. Η PVM είναι ένα ειδικό λογισμικό το οποίο προσομοιώνει έναν υπολογιστή που έχει σχεδιαστεί για εκτέλεση προγραμμάτων της Python.



Σχήμα 2.1: Το μοντέλο εκτέλεσης προγράμματος σε Python

Αν η Python έχει δικαιώματα εγγραφής στο σύστημα του υπολογιστή μας, τότε αποθηκεύει τον `byte code` σε αρχεία με κατάληξη `.pyc` (σημαίνει compiled `.py`). Κάθε φορά που εκτελούμε ένα συγκεκριμένο πρόγραμμά μας, η Python φορτώνει το αντίστοιχο αρχείο `.pyc` και έτσι επιτυγχάνεται ταχύτερη εκτέλεση. Στην περίπτωση που η Python δεν έχει δικαιώματα εγγραφής στο σύστημα του υπολογιστή μας, τότε ο `byte code` αποθηκεύεται προσωρινά στη μνήμη μέχρι την έξοδο από το πρόγραμμά μας.

Έτσι, μπορούμε να πούμε ότι η Python αποτελείται από τρία βασικά τμήματα: έναν διερμηνευτή (interpreter) για εκτέλεση ξεχωριστών εντολών, έναν μεταφραστή (compiler) για μετατροπή `.py` αρχείων σε `.pyc` αρχεία και μια εικονική μηχανή (PVM) για εκτέλεση `.pyc` αρχείων.

Κεφάλαιο 3

Τιμές και τύποι, μεταβλητές

Ένα πρώτο σημαντικό βήμα για να μάθουμε να προγραμματίζουμε είναι να κατανοήσουμε τους βασικούς τύπους δεδομένων της Python: τους ακέραιους αριθμούς, τους αριθμούς κινητής υποδιαστολής (δεκαδικούς) και τις συμβολοσειρές (ακολουθίες χαρακτήρων). Ειδικότερα, οι συμβολοσειρές χρησιμοποιούνται σε πολλές κατηγορίες προγραμμάτων και γι' αυτό η Python μάς παρέχει σημαντική υποστήριξη για τη διαχείρισή τους.

Η έννοια της μεταβλητής είναι σημαντική για τον προγραμματισμό. Οι μεταβλητές χρησιμοποιούνται για την αποθήκευση και διαχείριση δεδομένων σε ένα πρόγραμμα και είναι απαραίτητες για τη συγγραφή χρήσιμων προγραμμάτων.

Στο κεφάλαιο αυτό θα γράψουμε εντολές στο Python Shell και θα τις εκτελέσουμε άμεσα. Απαιτείται εξάσκηση, για να αρχίσουμε σιγά-σιγά να μαθαίνουμε να προγραμματίζουμε. Γράψτε και εκτελέστε τις εντολές των παραδειγμάτων.

3.1 Τιμές και τύποι δεδομένων

Ένα από τα βασικά στοιχεία που διαχειρίζεται ένα πρόγραμμα σε Python είναι οι **τιμές** (values) ή **κυριολεκτικές σταθερές** (literal constants). Οι τιμές παραμένουν αμετάβλητες σε όλη τη διάρκεια εκτέλεσης ενός προγράμματος.

Για παράδειγμα:

```
3
2.5
'Hello, World!'
```

Αυτές οι τιμές ανήκουν σε διαφορετικούς **τύπους δεδομένων** (**data types**) ή **κλάσεις** (**classes**). Το `3` είναι ένας **ακέραιος αριθμός** (**integer**), το `2.5` είναι ένας **αριθμός κινητής υποδιαστολής** (**floating point**) και το `'Hello, World!'` μια **συμβολοσειρά** (**string**). Μια συμβολοσειρά είναι μια ακολουθία από χαρακτήρες που περικλείονται σε μονά ή διπλά εισαγωγικά. Για να τυπώσουμε τιμές, μπορούμε να χρησιμοποιήσουμε την εντολή `print`.

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

Αν δεν είμαστε σίγουροι για τον τύπο στον οποίο ανήκει μια τιμή, μπορούμε να «ζητήσουμε βοήθεια» από τον διερμηνευτή χρησιμοποιώντας την εντολή `type`. Βλέπουμε ότι οι ακέραιοι αριθμοί ανήκουν στην κλάση `int`, οι αριθμοί κινητής υποδιαστολής (δεκαδικοί αριθμοί) στην κλάση `float` και οι συμβολοσειρές (ακολουθίες χαρακτήρων) στην κλάση `str`. Προς το παρόν, μπορούμε να χρησιμοποιήσουμε εναλλάξ τις έννοιες τύπος και κλάση. Θα εμβαθύνουμε στην έννοια της κλάσης σε επόμενο κεφάλαιο.

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

Η κλάση `float` περιέχει τους αριθμούς με υποδιαστολή (ως σύβολο χρησιμοποιούμε την τελεία και όχι το κόμμα) και ονομάζεται έτσι λόγω της μορφής αναπαράστασης των αριθμών (κινητή υποδιαστολή, floating point). Ας δοκιμάσουμε να εκτελέσουμε την εντολή `print(2,5)`. Βλέπουμε ότι δεν τυπώνεται ο δεκαδικός αριθμός 2,5 αλλά η λίστα των αριθμών 2 και 5 (θα μιλήσουμε για λίστες σε επόμενο κεφάλαιο). Για να τυπώσουμε τον δεκαδικό αριθμό 2,5 θα πρέπει να χρησιμοποιήσουμε την εντολή `print(2.5)`.

```
>>> print(2,5)
2 5
>>> print(2.5)
2.5
>>>
```

Τι τύπου είναι οι τιμές '3' και '2.5'; Φαίνεται να μοιάζουν με αριθμούς αλλά βρίσκονται μέσα σε εισαγωγικά όπως τις συμβολοσειρές. Η απάντηση είναι

```
>>> type('3')
<class 'str'>
>>> type('2.5')
<class 'str'>
>>>
```

αναμενόμενη, οι τιμές '3' και '2.5' είναι συμβολοσειρές.

Οι συμβολοσειρές στην Python περικλείονται σε μονά (') ή διπλά (") ή ακόμα και τριπλά εισαγωγικά (''' ή """). Μπορείτε να χρησιμοποιείτε ελεύθερα μονά εισαγωγικά μέσα σε διπλά εισαγωγικά. Χρησιμοποιώντας τριπλά εισαγωγικά μπορείτε να ορίσετε συμβολοσειρές πολλαπλών γραμμών. Επίσης, μπορείτε να χρησιμοποιείτε ελεύθερα μονά και διπλά εισαγωγικά μέσα σε τριπλά εισαγωγικά.

```
>>> type('Αυτή είναι μια συμβολοσειρά.')
<class 'str'>
>>> type("Κι αυτή είναι μια συμβολοσειρά.")
<class 'str'>
>>> type('''Ακόμα κι αυτή είναι μια συμβολοσειρά.''')
<class 'str'>
>>> print("Let's go.")
Let's go.
>>> print('''Hello,
World!''')
Hello,
World!
>>>
```

Δοκιμάστε να τυπώσετε τη φράση `Let's go` *με χρήση μόνο μονών εισαγωγικών. Τι τυπώνεται; Γιατί συμβαίνει αυτό;*

Στην Python δεν δηλώνουμε ρητά τους τύπους δεδομένων που χρησιμοποιούμε. Η ιδιότητα αυτή ονομάζεται δυναμική απόδοση τύπων (dynamic typing). Ο έλεγχος γίνεται κατά την εκτέλεση του προγράμματος. Σε επόμενα κεφάλαια θα δούμε και άλλους τύπους δεδομένων, καθώς επίσης και τη δυνατότητα να δημιουργούμε δικούς μας τύπους ορίζοντας νέες κλάσεις.

3.2 Μεταβλητές

Ένα από τα πιο δυνατά χαρακτηριστικά μιας γλώσσας προγραμματισμού είναι η δυνατότητα διαχείρισης μεταβλητών. Μια **μεταβλητή** (**variable**) είναι ένα όνομα που αναφέρεται σε μία τιμή. Η τιμή αυτή μπορεί να αλλάξει όσες φορές θέλουμε ή απαιτείται κατά την εκτέλεση του προγράμματός μας, εξ ου και ο όρος μεταβλητή. Οι μεταβλητές απλά είναι τμήματα της μνήμης του υπολογιστή μας όπου μπορούμε να αποθηκεύσουμε πληροφορία.

Η **εντολή εκχώρησης** (**assignment statement**) δημιουργεί νέες μεταβλητές και τους δίνει τιμές. Οι μεταβλητές έχουν επίσης τύπους και μπορούμε να ρωτήσουμε την Python ποιοι είναι. Ο τύπος μιας μεταβλητής είναι ο τύπος της τιμής στην οποία αναφέρεται. Για να χρησιμοποιήσουμε μια μεταβλητή, χρειάζεται απλά να γνωρίζουμε το όνομά της. Ας δούμε ένα παράδειγμα:

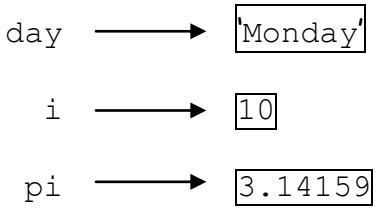
```
>>> day = 'Monday'
>>> print(day)
Monday
>>> type(day)
<class 'str'>
>>> i = 10
>>> print(i)
10
>>> type(i)
<class 'int'>
>>> pi = 3.14159
>>> print(pi)
3.14159
>>> type(pi)
<class 'float'>
>>>
```

Στο παράδειγμα αυτό έχουμε τρεις εντολές εκχώρησης:

- Η συμβολοσειρά 'Monday' εκχωρείται στη νέα μεταβλητή που ονομάζεται `day`.
- Ο ακέραιος αριθμός `10` εκχωρείται στη μεταβλητή `i`.
- Ο δεκαδικός αριθμός `3.14159` εκχωρείται στη μεταβλητή `pi`.

Μία εντολή εκχώρησης αποτελείται από τρία μέρη: το αριστερό που υποχρεωτικά πρέπει να είναι μία μεταβλητή, το ίσον = (τελεστής εκχώρησης) και το δεξί μέρος που μπορεί να είναι μία τιμή, μία άλλη μεταβλητή ή μία έκφραση που αποτιμάται (υπολογίζεται) σε μία τιμή. Θα μιλήσουμε περαιτέρω για εκφράσεις και τελεστές στο επόμενο κεφάλαιο. Να τονίσουμε ότι το ίσον = είναι τελεστής εκχώρησης και όχι ισότητας, και ότι δεν μπορούμε να εκχωρήσουμε μία τιμή σε μία άλλη τιμή.

Ένας κοινός τρόπος για να αναπαραστήσουμε στο χαρτί τις μεταβλητές για μια δεδομένη χρονική στιγμή είναι να γράψουμε το όνομά τους ακολουθούμενο από ένα βελάκι και την τιμή στην οποία δείχνουν. Για παράδειγμα:



Καλό είναι γενικά να διαλέγουμε ονόματα μεταβλητών που έχουν νόημα, εξηγούν για ποιο σκοπό χρησιμοποιούνται. Θα ήταν παράλογο, για παράδειγμα, να αποθηκεύσουμε μία τιμή που αντιστοιχεί σε ημέρα σε μία μεταβλητή με όνομα `colour`.

Κανόνες δημιουργίας ονομάτων μεταβλητών:

- Τα ονόματα μπορούν να είναι όσο μεγάλα θέλουμε, μπορούν να περιέχουν γράμματα και αριθμούς, αλλά πρέπει να ξεκινούν με ένα γράμμα ή τον χαρακτήρα _ (underscore/κάτω παύλα). Αν και μπορούμε να χρησιμοποιήσουμε ελληνικά γράμματα, καλό είναι να τα αποφεύγουμε, επειδή είναι εύκολο να προκληθεί σύγχυση.
- Τα πεζά διακρίνονται από τα κεφαλαία γράμματα (case sensitive), για παράδειγμα οι `day`, `Day`, `DAy`, `DAY`, `dAy`, `daY`, `DaY`, `dAY` είναι διαφορετικές μεταβλητές. Οι έμπειροι προγραμματιστές αποφεύγουν τα κεφαλαία.
- Δεν επιτρέπονται κενά, εισαγωγικά, τελείες, κόμματα και άλλοι παρόμοιοι χαρακτήρες. Επιτρέπεται μόνο ο χαρακτήρας _ (underscore/κάτω παύλα), ο οποίος είναι χρήσιμος κυρίως σε ονόματα που αποτελούνται από πολλές λέξεις.

Αν δοθεί σε μία μεταβλητή ένα «παράνομο» όνομα, θα έχουμε συντακτικό λάθος. Δείτε δίπλα μερικά παραδείγματα:

```
>>> _my_name_is = 'George'
>>> print(_my_name_is)
George
>>> day = 'Monday'
>>> print(day)
Monday
>>> print(Day)
Traceback (most recent call last):
  File "<pyshell#119>", line 1, in <module>
    print(Day)
NameError: name 'Day' is not defined
>>> day1 = 'Monday'
>>> print(day1)
Monday
>>> 1day = 'Monday'
SyntaxError: invalid syntax
>>> day@ = 'Monday'
SyntaxError: invalid syntax
>>> from = 'Monday'
SyntaxError: invalid syntax
>>>
```

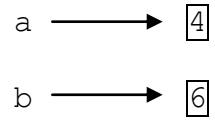
Κατά την εκτέλεση της εντολής `print(Day)` εμφανίζεται μήνυμα λάθους, διότι δεν έχουμε εκχωρήσει τιμή στη μεταβλητή `Day`, η `day` είναι διαφορετική μεταβλητή (διάκριση πεζών και κεφαλαίων στα ονόματα μεταβλητών). Το όνομα `1day` είναι «παράνομο», επειδή ξεκινάει με αριθμό, το όνομα `day@` είναι παράνομο, αφού περιέχει τον μη επιτρεπόμενο χαρακτήρα `@`. Ποιο είναι όμως το λάθος με το όνομα `from`; Η λέξη `from` είναι μία από τις **λέξεις κλειδιά** (**keywords**) της Python. Οι λέξεις κλειδιά συνδέονται με τους κανόνες και τη δομή της γλώσσας, και δεν μπορούν να χρησιμοποιηθούν ως ονόματα μεταβλητών. Η Python 3.4.3 έχει 33 λέξεις κλειδιά, τις παρακάτω:

```
['False', 'None', 'True', 'and', 'as', 'assert',
'break', 'class', 'continue', 'def', 'del', 'elif',
'else', 'except', 'finally', 'for', 'from', 'global',
'if', 'import', 'in', 'is', 'lambda', 'nonlocal',
'not', 'or', 'pass', 'raise', 'return', 'try', 'while',
'with', 'yield']
```

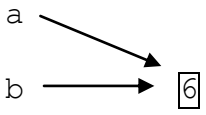
Η Python παρακολουθεί όλες τις τιμές και αυτόματα τις διαγράφει από τη μνήμη, όταν πάψει να γίνεται αναφορά σε αυτές από κάποια μεταβλητή. Η διαδικασία αυτή ονομάζεται **garbage collection**. Ας δούμε το παρακάτω παράδειγμα:

```
>>> a = 4
>>> b = 6
>>> a = b
>>> print(a)
6
>>> print(b)
6
>>>
```

Μετά την εκτέλεση των εντολών `a = 4` και `b = 6` η μεταβλητή `a` δείχνει στην τιμή `4` και η μεταβλητή `b` στην τιμή `6`:



αλλά στη συνέχεια μετά την εκτέλεση της εντολής `a = b` και οι δύο μεταβλητές δείχνουν στην τιμή `6` και η τιμή `4` διαγράφεται αυτόματα από τη μνήμη:



Τι σημαίνει η εντολή εκχώρησης `num = num + 1`; Στα Μαθηματικά δεν έχει νόημα, αλλά στην Python χρησιμοποιείται για να αυξήσουμε την τιμή μιας μεταβλητής κατά ένα:

```
>>> num = 17
>>> num = num + 1
>>> print(num)
18
>>>
```

Αν σας βολεύει, μπορείτε να σκέφτεστε τον τελεστή εκχώρησης = σαν βελάκι:

num ← num + 1

Αρχικά: `num` ⟶ 17

Μετά την εντολή εκχώρησης: `num` ⟶ 18

Κεφάλαιο 4

Εκφράσεις, τελεστές, σχόλια

Οι περισσότερες εντολές ενός προγράμματος περιέχουν εκφράσεις. Μία έκφραση είναι ένας συνδυασμός συμβόλων, συνήθως τελεστών και τελεστέων, που αποτιμώνται σε μία τιμή. Ένας τελεστής είναι ένα σύμβολο που αναπαριστά μία λειτουργία. Ένα απλό παράδειγμα έκφρασης είναι η `3 + 2` , όπου το σύμβολο `+` είναι ο τελεστής που αναπαριστά τη λειτουργία της πρόσθεσης, οι αριθμοί `3` και `2` είναι οι τελεστέοι της έκφρασης και η αποτίμησή της θα δώσει την τιμή `5`.

Μία καλή πρακτική στον προγραμματισμό είναι η τοποθέτηση σχολίων στα προγράμματά μας, ειδικά στα μεγαλύτερα, για καλύτερη κατανόηση του κώδικα.

Στο κεφάλαιο αυτό θα γράψουμε εντολές και στο Python Shell για άμεση εκτέλεση και μικρά ολοκληρωμένα σενάρια (προγράμματα).

4.1 Εκφράσεις και τελεστές

Μία **έκφραση** (**expression**) είναι ένας συνδυασμός από τιμές, μεταβλητές, τελεστές και κλήσεις σε συναρτήσεις (θα μιλήσουμε για συναρτήσεις σε επόμενο κεφάλαιο). Οι **τελεστές** (**operators**) είναι λειτουργίες που κάνουν κάτι και μπορούν να αναπαρασταθούν με σύμβολα όπως το `+` ή με λέξεις κλειδιά όπως το `and`. Η αποτίμηση μιας έκφρασης παράγει μία τιμή και αυτός είναι και ο λόγος που μία έκφραση μπορεί να βρίσκεται στο δεξί μέρος μια εντολής εκχώρησης. Όταν μία μεταβλητή εμφανίζεται σε έκφραση, αντικαθίσταται από την τιμή της, προτού αποτιμηθεί η έκφραση. Εάν εισάγετε μία έκφραση στον διερμηνευτή, αυτός την υπολογίζει και δείχνει το αποτέλεσμα:

```
>>> 3 + 2
5
>>> x = 3
>>> x + 2
5
>>>
```

Δεν απαιτείται μία έκφραση να περιέχει ταυτόχρονα και τιμές και μεταβλητές και τελεστές. Μία τιμή, όπως και μία μεταβλητή, από μόνες τους είναι επίσης εκφράσεις:

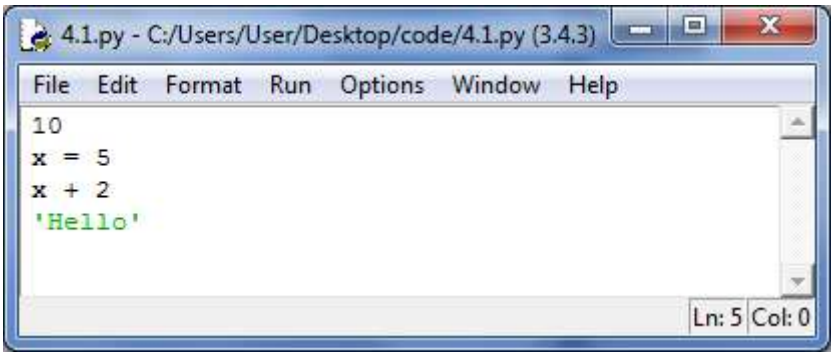
```
>>> 10
10
>>> x = 8
>>> x
8
>>>
```

Παρατηρήστε τη διαφορά μεταξύ του υπολογισμού μιας έκφρασης και της εκτύπωσής της:

```
>>> message = 'Hello, World!'
>>> message
'Hello, World!'
>>> print(message)
Hello, World!
>>>
```

Σε ένα σενάριο (script), μία έκφραση είναι από μόνη της μια νόμιμη εντολή, αλλά δεν παράγει έξοδο:

Κώδικας 4.1




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

Πώς θα παραχθεί έξοδος στο παραπάνω παράδειγμα

Μελετήστε τον παρακάτω πίνακα, ο οποίος περιέχει την εξήγηση και παραδείγματα για μερικούς τελεστές της Python:

Πίνακας 4.1 Τελεστές και η χρήση τους

Τελεστής	Όνομα	Εξήγηση	Παραδείγματα
+	Συν	Πρόσθεση αριθμών ή αλληλουχία συμβολοσειρών.	Το `5 + 3` δίνει `8` Το `'a' + 'b'` δίνει `'ab'`
-	Μείον	Αφαίρεση ενός αριθμού από έναν άλλο.	Το `50 - 26` δίνει `24`
*	Επί	Γινόμενο δύο αριθμών ή επανάληψη μιας συμβολοσειράς τόσες φορές.	Το `2 * 3` δίνει `6` Το `'la' * 3` δίνει `'lalala'`
**	Δύναμη	Ύψωση αριθμού σε δύναμη.	Το `3 ** 4` δίνει `81` Το `10 ** 2` δίνει `100` Το `2.0 ** 3` δίνει `8.0`

/	Διά	Διαίρεση δύο αριθμών. Το αποτέλεσμα είναι πάντα δεκαδικός αριθμός.	Το `4 / 3` δίνει `1.3333333333333333` Το `10 / 2` δίνει `5.0` Το `1 / 2` δίνει `0.5`
//	Ακέραια διαίρεση	Διαίρεση δύο αριθμών στρογγυλοποιημένη προς τα κάτω (floor division).	Το `4 // 3` δίνει `1` Το `1 // 2` δίνει `0` Το `17 // 3` δίνει `5` Το `17.1 // 3` δίνει `5.0`
%	Υπόλοιπο	Υπόλοιπο διαίρεσης δύο αριθμών.	Το `11 % 3` δίνει `2`

Στο επόμενο κεφάλαιο θα δούμε και άλλους τελεστές.

Η Python δεν βάζει όρια στο μέγεθος των ακέραιων αριθμών. Για παράδειγμα:

```
>>> 37 ** 100
6609557828843866774348296857793615320986068325257944996730965130260195627493490637048004105256563742994070037769599882399012397170569200279466412758131334001
>>>
```

Πολύ μικροί ή πολύ μεγάλοι δεκαδικοί αριθμοί γράφονται σε επιστημονική μορφή. Για παράδειγμα:

```
>>> 2348.5 ** 17
2.011127464213224e+57
>>>
```

Το `e+57` σημαίνει πολλαπλασιασμό του προηγηθέντος αριθμού με το 10^{57}.

Προτεραιότητα τελεστών

Όταν περισσότερα από ένα σύμβολα τελεστών εμφανίζονται σε μία έκφραση, η σειρά υπολογισμού εξαρτάται από τους κανόνες προτεραιότητας. Οι κανόνες αυτοί είναι σύμφωνοι με τη σειρά εκτέλεσης των αριθμητικών πράξεων που ξέρουμε από τα Μαθηματικά.

Οι **παρενθέσεις** έχουν τη μεγαλύτερη προτεραιότητα και έτσι χρησιμοποιούνται, για να «αναγκάσουν» την Python να αποτιμήσει μία έκφραση σύμφωνα με τη σειρά που θέλουμε. Εκφράσεις σε παρενθέσεις αποτιμώνται πρώτες. Επίσης, παρενθέσεις χρησιμοποιούνται, για να κάνουν τις εκφράσεις πιο αναγνώσιμες, χωρίς να αλλάξουν το τελικό αποτέλεσμα. Η **ύψωση σε δύναμη** έχει την επόμενη μεγαλύτερη προτεραιότητα. Για παράδειγμα:

```
>>> 2 * (4 - 1)
6
>>> 2 * 4 - 1
7
>>> (2 + 1) ** (5 - 2)
27
>>> 2 ** 1 + 1
3
>>> 2 ** (1 + 1)
4
>>>
```

Πολλαπλασιασμός και **διαίρεση** έχουν την ίδια προτεραιότητα, η οποία είναι μεγαλύτερη από την **πρόσθεση** και την **αφαίρεση** που μεταξύ τους έχουν επίσης την ίδια προτεραιότητα. Για παράδειγμα:

```
>>> 3 * 5 + 2
17
>>> 15 / 3 - 2
3.0
>>>
```

Τελεστές με την ίδια προτεραιότητα αποτιμώνται από τα αριστερά προς τα δεξιά.

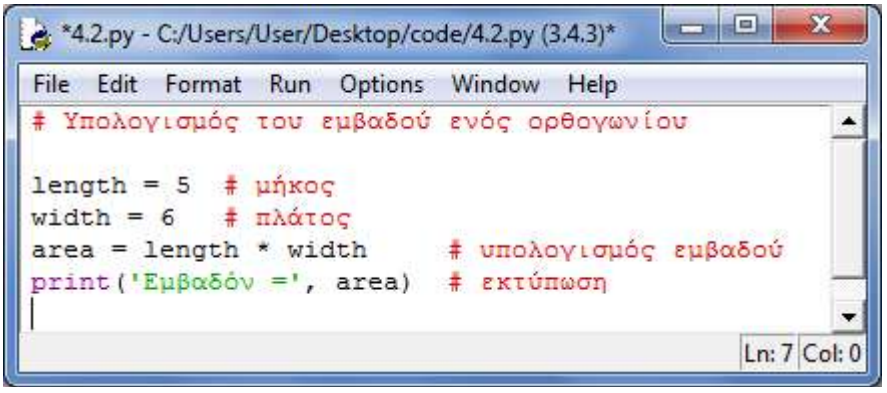
Πειραματιστείτε, δώστε τις δικές σας εκφράσεις στον διερμηνευτή και δείτε τις τιμές που παράγονται.

4.2 Σχόλια

Καθώς γράφουμε όλο και μεγαλύτερα προγράμματα, μεγαλώνει και η δυσκολία να καταλαβαίνουμε αυτά που γράφουμε. Πολύ περισσότερο όμως, αν προσπαθήσει να τα διαβάσει κάποιος τρίτος. Γενικά, οι γλώσσες προγραμματισμού συμπυκνώνουν νοήματα και γι' αυτό είναι πολλές φορές δύσκολο να τις διαβάσει κάποιος και να καταλάβει τι θέλει να κάνει ένα κομμάτι κώδικα. Χρειάζεται λοιπόν να προσθέτουμε **σχόλια (comments)** στα προγράμματα που γράφουμε.

Τα σχόλια στην Python αρχίζουν πάντα με τον χαρακτήρα #. Οτιδήποτε ακολουθεί μετά το # αγνοείται από την Python μέχρι το τέλος της γραμμής (προορίζεται για ανάγνωση μόνο από άνθρωπο). Ας δούμε ένα παράδειγμα:

Κώδικας 4.2




```
>>> ================================ RESTART ====
>>>
Εμβαδόν = 30
>>>
```

Κεφάλαιο 5

Έλεγχος ροής εκτέλεσης

Τα προγράμματα που γράψαμε μέχρι τώρα αποτελούνταν από μια σειρά από εντολές που εκτελούνταν από την Python πιστά η μία μετά την άλλη. Η ακολουθιακή (σειριακή) αυτή δομή εντολών χρησιμοποιείται πρακτικά για την αντιμετώπιση πολύ απλών προβλημάτων. Σε πιο σύνθετα προβλήματα απαιτείται η χρήση κατάλληλων εντολών για έλεγχο συνθηκών, με τελικό σκοπό την επιλογή των απαραίτητων ομάδων εντολών που θα εκτελούνται κάθε φορά που θα «τρέχει» ένα πρόγραμμα (δομή επιλογής). Ο έλεγχος μιας συνθήκης, στις περισσότερες γλώσσες προγραμματισμού, βασίζεται στη λογική `Boolean`, όπου υπάρχουν μόνο δύο τιμές (`True` και `False`) και αξιοποιούνται οι λογικοί τελεστές. Επίσης, σε πολλές περιπτώσεις χρειάζεται η πολλαπλή επανάληψη μιας συγκεκριμένης ομάδας εντολών (δομή επανάληψης).

Προτού όμως γράψουμε προγράμματα που να περιέχουν δομές επιλογής ή/και επανάληψης, απαραίτητο είναι να δούμε πώς μπορούμε να κάνουμε είσοδο δεδομένων από το πληκτρολόγιο καθώς και κάποιες χρήσιμες ενσωματωμένες (built-in) συναρτήσεις της Python.

5.1 Είσοδος δεδομένων από το πληκτρολόγιο

Στα περισσότερα προγράμματα απαιτείται η είσοδος δεδομένων από τον χρήστη. Για τον σκοπό αυτό η Python διαθέτει την ενσωματωμένη συνάρτηση `input`. Κάθε φορά που καλείται η συνάρτηση αυτή από ένα πρόγραμμα, αυτό σταματάει και περιμένει από τον χρήστη να πληκτρολογήσει κάτι. Όταν ο χρήσης πληκτρολογήσει την είσοδο και πατήσει το πλήκτρο `Enter`, το πρόγραμμα συνεχίζει τη ροή εκτέλεσης και η συνάρτηση `input` επιστρέφει την είσοδο σαν συμβολοσειρά. Χρήσιμο είναι να εμφανίζεται και ένα μήνυμα (prompt) που να πληροφορεί τον χρήστη για το είδος της εισόδου που αναμένεται από το πρόγραμμα.

Για παράδειγμα:

```
>>> message = input()
Hello
>>> print(message)
Hello
>>> name = input('Πώς σε λένε;')
Πώς σε λένε;Αντώνη
>>> print(name)
Αντώνη
>>> age = input('Πόσο χρονών είσαι;')
Πόσο χρονών είσαι;13
>>> print(age)
13
>>> type(age)
<class 'str'>
>>>
```

Στο παράδειγμα αυτό βλέπουμε ξεκάθαρα ότι η συνάρτηση `input` επιστρέφει πάντα συμβολοσειρά και έτσι η μεταβλητή `age` δεν δείχνει σε αριθμό.

Η Python μάς προσφέρει ενσωματωμένες συναρτήσεις για μετατροπή από έναν τύπο σε έναν άλλο:

- `float(x)`: μετατρέπει ακέραιους αριθμούς και συμβολοσειρές σε αριθμούς κινητής υποδιαστολής.
- `int(x)`: μετατρέπει αριθμούς κινητής υποδιαστολής («κόβει» τα δεκαδικά ψηφία) και συμβολοσειρές σε ακέραιους αριθμούς.
- `str(n)`: μετατρέπει αριθμούς σε συμβολοσειρές.

Για παράδειγμα:

```
>>> float(5)
5.0
>>> float('6.8')
6.8
>>> int(6.8)
6
>>> int('13')
13
>>> str(13)
'13'
>>>
```

Σε αρκετές περιπτώσεις είναι χρήσιμη και η ενσωματωμένη συνάρτηση `round` η οποία στρογγυλοποιεί δεκαδικούς αριθμούς. Για παράδειγμα:

```
>>> round(5.3)
5
>>> round(5.5)
6
>>> round(5.7)
6
>>>
```

Ας δούμε τώρα σε παράδειγμα έναν συνδυασμό εισόδου δεδομένων από το πληκτρολόγιο και αξιοποίησης των συναρτήσεων μετατροπής:

```
>>> age = input('Πόσο χρονών είσαι;')
Πόσο χρονών είσαι;13
>>> print('Το επόμενο έτος θα είσαι', int(age)+1)
Το επόμενο έτος θα είσαι 14
>>> x = float(input('Δώστε έναν αριθμό:'))
Δώστε έναν αριθμό:2.5
>>> print('Το διπλάσιο του αριθμού που δώσατε είναι:', 2*x)
Το διπλάσιο του αριθμού που δώσατε είναι: 5.0
>>>
```

Μελετήστε και το παρακάτω παράδειγμα:

```
>>> x = input('Δώστε τον πρώτο αριθμό:')
Δώστε τον πρώτο αριθμό:6
>>> y = input('Δώστε τον δεύτερο αριθμό:')
Δώστε τον δεύτερο αριθμό:4
>>> print(x, '+', y, '=', int(x)+int(y))
6 + 4 = 10
>>>
```

Στο παραπάνω παράδειγμα τι θα τυπώσει και γιατί η ακόλουθη εντολή

```
>>> print(x, '+', y, '=', x+y)
```

Υπάρχει λογική στη χρήση μιας τέτοιας εντολής;

Δώστε προσοχή στις παρακάτω επισημάνσεις:

- Οι συναρτήσεις `int` και `float` επιστρέφουν μία τιμή, έναν ακέραιο αριθμό ή έναν αριθμό κινητής υποδιαστολής αντίστοιχα, δεν αλλάζουν τον τύπο μιας μεταβλητής και την τιμή στην οποία αυτή δείχνει. Αν θέλουμε, μπορούμε να εκχωρήσουμε την επιστρεφόμενη τιμή σε μια νέα μεταβλητή. Για παράδειγμα:

```
>>> x = input('Δώστε έναν ακέραιο αριθμό:')
Δώστε έναν ακέραιο αριθμό:5
>>> print(2*int(x))
10
>>> print(2*x)
55
>>> type(x)
<class 'str'>
>>> y = int(x)
>>> print(y)
5
>>> type(y)
<class 'int'>
>>>
```

- Στις συναρτήσεις `int` και `float` θα πρέπει οι συμβολοσειρές που θέλουμε να μετατρέψουμε σε αριθμούς να «μοιάζουν» πράγματι με αριθμούς. Διαφορετικά θα πάρουμε μήνυμα σφάλματος. Για παράδειγμα, η συμβολοσειρά '`8.25`' μπορεί να μετατραπεί σε αριθμό, ενώ η συμβολοσειρά '`hello`' είναι φυσικό ότι δεν μπορεί. Τι γίνεται όμως με τη συμβολοσειρά '`8,25`'; Μπορεί να μετατραπεί σε αριθμό; Δοκιμάστε.
- Όταν αναμειγνύουμε ακέραιους με δεκαδικούς αριθμούς σε μία έκφραση, η Python αυτόματα μετατρέπει τους ακέραιους σε δεκαδικούς.
- Όταν έχουμε παρενθέσεις στις εντολές μας, θα πρέπει κάθε παρένθεση που ανοίγει να κλείνει κιόλας ή αλλιώς ο αριθμός των αριστερών παρενθέσεων θα πρέπει να ισούται με τον αριθμό των δεξιών παρενθέσεων.
- Αρκετά λάθη προκύπτουν, αν μπερδεύουμε γράμματα του Ελληνικού με γράμματα του Λατινικού αλφάβητου.
- Δεν πρέπει να ξεχνάμε τη διάκριση μεταξύ πεζών και κεφαλαίων γραμμάτων στα ονόματα των μεταβλητών.

5.2 Λογική Boolean

Στα προγράμματα που γράφουμε γίνεται συχνά έλεγχος συνθηκών και ανάλογα με το αποτέλεσμα αλλάζει η συμπεριφορά τους κατά την εκτέλεση. Για να γίνει αυτό, χρησιμοποιείται η **λογική Boolean**. Το όνομα προέρχεται από τον Βρετανό Μαθηματικό George Boole, ο οποίος εισήγαγε την ομώνυμη άλγεβρα που αφορά σε λογικούς κανόνες συνδυασμού των δύο τιμών `True` (`Αληθής`) και `False` (`Ψευδής`). Οι τιμές αυτές ονομάζονται λογικές ή Boolean και ο τύπος τους στην Python είναι ο `bool`. Μία μεταβλητή μπορεί να δείχνει και σε μία τιμή τύπου `bool`. Για παράδειγμα:

```
>>> type(False)
<class 'bool'>
>>> raining = True
>>> type(raining)
<class 'bool'>
>>>
```

Ο συνδυασμός των Boolean τιμών γίνεται με τη χρήση τριών βασικών **λογικών τελεστών**: `not`, `and` και `or`. Οι τελεστές αυτοί συμβολίζουν λογικές πράξεις (`όχι`, `και`, `ή`) και βοηθούν στη λήψη αποφάσεων σε ένα πρόγραμμα. Η σημαντική τους είναι πολύ παρόμοια με την καθημερινή τους σημασία. Ας υποθέσουμε ότι `p` και `q` είναι δύο μεταβλητές που δείχνουν σε Boolean τιμές ή δύο εκφράσεις (ονομάζονται λογικές ή Boolean) που αποτιμώνται σε Boolean τιμές. Ο επόμενος πίνακας, ο οποίος ονομάζεται **πίνακας αληθείας**, περιέχει τις τιμές που επιστρέφουν οι τρεις λογικές πράξεις για όλους τους συνδυασμούς τιμών των `p` και `q`.

Πίνακας 5.1 Πίνακας αληθείας για τους βασικούς λογικούς τελεστές

p	q	not p	p and q	p or q
False	False	True	False	False
False	True	True	False	True
True	False	False	False	True
True	True	False	True	True

Η λογική έκφραση `not p` είναι αληθής, όταν η `p` ψευδής, και ψευδής, όταν η `p` είναι αληθής. Η λογική έκφραση `p and q` είναι αληθής μόνο όταν και η `p` και η `q` είναι αληθείς, σε κάθε άλλη περίπτωση είναι ψευδής. Η λογική έκφραση `p or q` είναι αληθής όταν η `p` είναι αληθής ή η `q` είναι αληθής, ή όταν και οι δύο είναι αληθείς. Σε μία λογική έκφραση μπορούμε να έχουμε και **τελεστές σύγκρισης**:

Πίνακας 5.2 Τελεστές σύγκρισης

`x == y`	x είναι ίσος με y
`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

Οι λογικές εκφράσεις χρησιμοποιούν παρενθέσεις και κανόνες προτεραιότητας, για να καθορίσουν τη σειρά αποτίμησης των τμημάτων από τα οποία αποτελούνται. Οι εκφράσεις μέσα σε παρενθέσεις αποτιμώνται πρώτες. Η προτεραιότητα των τελεστών, από τη μεγαλύτερη στη μικρότερη, είναι η εξής:

`Αριθμητικοί τελεστές` (με τη γνωστή προτεραιότητα)

`<, >, <=, >=, !=, ==`

`not`

`and`

`or`

Για παράδειγμα:

```
>>> not (True and (False or True))
False
>>> False and not False or True
True
>>> (6 == 6) and (7 >= 2)
True
>>> (4 > 6) or (4 != 6)
True
>>>
```

Δείτε και αυτό το παράδειγμα:

```
>>> 5 == (3+2)
True
>>> name = 'James'
>>> name + ' ' + 'Bond' == 'James Bond'
True
>>>
```

Δώστε προσοχή στις παρακάτω επισημάνσεις:

- Ο τελεστής ισότητας είναι ο ==. Ο τελεστής = είναι, όπως έχουμε ξαναδεί, ο τελεστής εκχώρησης.
- Δεν υπάρχουν τελεστές σύγκρισης με σύμβολα όπως τα =< και =>.
- Σε πολύπλοκες λογικές εκφράσεις είναι χρήσιμο να χρησιμοποιούμε παρενθέσεις για αποφυγή σύγχυσης στη σειρά αποτίμησης.

Ας δούμε μερικά επιπλέον παραδείγματα για τη σειρά αποτίμησης σε μια λογική έκφραση:

`3 + 2 < 7 + 3` → `5 < 10` → `True`

`3 > 2 and 7 <= 3` → `True and False` → `False`

`10 < 20 and 30 < 20 or 30 < 40` → `True and False or True`

→ `False or True` → `True`

`not 10 < 20 or 30 < 20` → `not True or False`

→ `False or False` → `False`

Μελετήστε τις παρακάτω «ιδιαίτερες» λογικές εκφράσεις. Προσπαθήστε να καταγράψετε κανόνες για την άλγεβρα του

```
(x and False) == False
(False and x) == False
(y and x) == (x and y)
(x and True) == x
(True and x) == x
(x and x) == x
(x or False) == x
(False or x) == x
(y or x) == (x or y)
(x or True) == True
(True or x) == True
(x or x) == x
(not (not x)) == x
```

5.3 Εκτέλεση υπό συνθήκη, η εντολή `if`

Η εντολή `if` χρησιμοποείται για έλεγχο της ροής εκτέλεσης ενός προγράμματος. Ελέγχεται μία συνθήκη και ανάλογα με το αποτέλεσμα (Αληθής ή Ψευδής) εκτελείται ή δεν εκτελείται μία ή κάποια άλλη ομάδα (μπλοκ) εντολών. Η εντολή `if` συντάσσεται ως εξής:

```
if συνθήκη:
    μπλοκ_εντολών_1     (true_block)
else:
    μπλοκ_εντολών_2     (false_block)
```

Βλέπουμε ότι ξεκινάει με μία **επικεφαλίδα** (**header**) που αποτελείται από τη δεσμευμένη λέξη `if`, μία συνθήκη και τελειώνει με μία άνω κάτω τελεία (`:`). Η συνθήκη μπορεί να είναι μία οποιαδήποτε λογική έκφραση που αποτιμάται σε `True` ή `False`. Αν η συνθήκη είναι αληθής (`True`), τότε εκτελείται το πρώτο μπλοκ εντολών (true_block), αλλιώς, αν είναι ψευδής (`False`), εκτελείται το δεύτερο μπλοκ εντολών (false_block). Ακριβώς πριν το δεύτερο μπλοκ εντολών τοποθετείται η δεσμευμένη λέξη `else` ακολουθούμενη από άνω και κάτω τελεία.

Ένα μπλοκ εντολών μέσα σε μία σύνθετη εντολή ονομάζεται το **σώμα** (**body**) της εντολής. Δεν υπάρχει όριο στο πλήθος των εντολών που μπορούν να εμφανισθούν στο σώμα μιας εντολής `if`, αρκεί να υπάρχει τουλάχιστον μία. Αν παρόλα αυτά χρειαζόμαστε ένα σώμα χωρίς εντολές (π.χ. για να θυμηθούμε αργότερα να επιστρέψουμε και να προσθέσουμε κάποιες), τότε μπορούμε να χρησιμοποιήσουμε την εντολή `pass` που ουσιαστικά δεν κάνει τίποτε.

Οι εντολές ενός μπλοκ πρέπει να είναι μετατοπισμένες προς τα δεξιά. Η τυπική μετατόπιση ή **εσοχή** (**indentation**) των εντολών είναι τέσσερα κενά. Η Python μάς βοηθάει σε αυτό ρυθμίζοντας αυτόματα τις εσοχές για μας, απλά πατώντας `Enter` μετά την πληκτρολόγηση της άνω κάτω τελείας. Η έλλειψη ενός κενού ή η ύπαρξη επιπλέον κενών μπορεί να οδηγήσει σε λάθος ή απρόσμενη συμπεριφορά σε ένα πρόγραμμα. Η πρώτη εντολή που ευθυγραμμίζεται με το αριστερό περιθώριο είναι η πρώτη εντολή έξω από το μπλοκ. Σε άλλες γλώσσες προγραμματισμού η οριοθέτηση των μπλοκ εντολών γίνεται με σύβολα ή λέξεις κλειδιά, π.χ. `{ }` στη *C* ή `begin end` στην Pascal.

Ας δούμε ένα απλό παράδειγμα:

Κώδικας 5.1

```
x = int(input('Δώστε έναν ακέραιο αριθμό:'))
if x % 2 == 0:
    print(x, 'είναι άρτιος')
else:
    print(x, 'είναι περιττός')
```

1η εκτέλεση (εκτελείται το πρώτο μπλοκ εντολών):

```
>>> ================================ RESTART ===
>>>
Δώστε έναν ακέραιο αριθμό:4
4 είναι άρτιος
```

2η εκτέλεση (εκτελείται το δεύτερο μπλοκ εντολών):

```
>>> ================================ RESTART ===
>>>
Δώστε έναν ακέραιο αριθμό:7
7 είναι περιττός
```

Η συνθήκη `x % 2 == 0` ελέγχει εάν ο αριθμός που δείχνει η μεταβλητή `x` είναι άρτιος ή περιττός (ένας αριθμός είναι άρτιος αν το υπόλοιπο της διαίρεσής του με το `2` είναι `0`). Αν είναι άρτιος, εκτελείται το πρώτο μπλοκ εντολών (μετά το `if`), αλλιώς, αν είναι περιττός, εκτελείται το δεύτερο μπλοκ εντολών (μετά το `else`).

Μέχρι τώρα είδαμε ότι στην εντολή `if` έχουμε δύο δυνατότητες εκτέλεσης (κλάδους εκτέλεσης). Μπορούμε να έχουμε και μόνο έναν κλάδο εκτέλεσης (χωρίς `else`) και να εκτελείται ένα μπλοκ εντολών στην περίπτωση που η συνθήκη είναι αληθής. Με άλλα λόγια ο κλάδος `else` είναι προαιρετικός. Για παράδειγμα:

Κώδικας 5.2

```
x = int(input('Δώστε έναν ακέραιο αριθμό:'))
if x > 0:
    print(x, 'είναι θετικός αριθμός')
print('Τέλος προγράμματος')
```

1η εκτέλεση (η συνθήκη είναι αληθής, ο αριθμός είναι θετικός):

```
>>> ================================ RESTART ===
>>>
Δώστε έναν ακέραιο αριθμό:10
10 είναι θετικός αριθμός
Τέλος προγράμματος
```

2η εκτέλεση (η συνθήκη είναι ψευδής, ο αριθμός δεν είναι θετικός):

```
>>> ================================ RESTART ===
>>>
Δώστε έναν ακέραιο αριθμό:-5
Τέλος προγράμματος
```

Προσέξτε ότι το μήνυμα `Τέλος προγράμματος` εμφανίζεται σε όλες τις εκτελέσεις του παραπάνω προγράμματος. Αυτό γίνεται επειδή η εντολή `print` που το εμφανίζει δεν βρίσκεται στο μπλοκ της `if` και απλά είναι στοιχισμένη αριστερά.

Στην Python μπορούμε να έχουμε και **αλυσιδωτές συνθήκες** με πολλούς κλάδους εκτέλεσης (όσους θέλουμε). Σε κάθε εκτέλεση ένας μόνο κλάδος εκτελείται. Κάθε συνθήκη ελέγχεται με τη σειρά, αν η πρώτη είναι ψευδής, πάμε στη δεύτερη, κ.ο.κ. Αν κάποια συνθήκη είναι αληθής, τότε ο αντίστοιχος κλάδος (μπλοκ εντολών) εκτελείται και η εντολή `if` ολοκληρώνεται, ακόμη και αν ακολουθούν και άλλες συνθήκες που αληθεύουν. Για αλυσιδωτές συνθήκες χρησιμοποιούμε τη δεσμευμένη λέξη `elif` (συντόμευση του `else if`). Για παράδειγμα:

Κώδικας 5.3

```
x = 20
y = 10

if x < y:
    print(x, '<', y)
elif x > y:
    print(x, '>', y)
else:
    print(x, '=', y)
```


```
>>> ================================ RESTART ===
>>>
20 > 10
```

Γράψτε και εκτελέστε το παραπάνω πρόγραμμα για διάφορες τιμές των `x` *και* `y`

Επίσης, η Python μας δίνει τη δυνατότητα να έχουμε και **εμφωλευμένες συνθήκες** (μία συνθήκη μέσα σε άλλη συνθήκη). Για παράδειγμα:

Κώδικας 5.4

```
x = 20
y = 10

if x == y:
    print(x, '=', y)
else:
    if x < y:
        print(x, '<', y)
    else:
        print(x, '>', y)
```


```
>>> ================================ RESTART ===
>>>
20 > 10
```

Γράψτε και εκτελέστε το παραπάνω πρόγραμμα για διάφορες τιμές των x *και* y

Οι αρχάριοι προγραμματιστές καλό είναι γενικά να αποφεύγουν τις εμφωλευμένες συνθήκες γιατί κάνουν τα προγράμματα δυσανάγνωστα. Σε πολλές περιπτώσεις, για να τις απλοποιήσουμε, μπορούμε να χρησιμοποιήσουμε κατάλληλους λογικούς τελεστές.

Μελετήστε τις παρακάτω εντολές. Τι μας προσφέρει αυτή η σύνταξη εντολών;

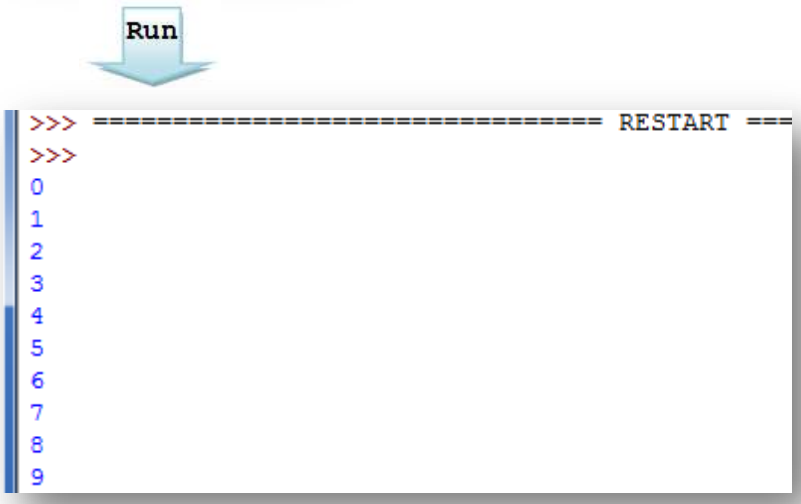
```
>>> x = int(input('Δώστε έναν ακέραιο αριθμό:'))
Δώστε έναν ακέραιο αριθμό:8
>>> what_is_x = 'άρτιος' if x % 2 == 0 else 'περιττός'
>>> print(x, what_is_x)
8 άρτιος
>>>
```

5.4 Η εντολή επανάληψης `for`

Πλήθος προβλημάτων μπορούν να επιλυθούν με συγγραφή προγραμμάτων στα οποία, κατά την εκτέλεσή τους, συγκεκριμένες ομάδες (μπλοκ) εντολών επαναλαμβάνονται (εκτελούνται) πολλές φορές. Μία τέτοια ομάδα εντολών που επαναλαμβάνεται ονομάζεται **βρόγχος (loop)**. Η εντολή `for` επαναλαμβάνει ένα δεδομένο μπλοκ εντολών για συγκεκριμένο πλήθος φορών. Για παράδειγμα, ο παρακάτω κώδικας τυπώνει μια ακολουθία αριθμών, τους ακέραιους αριθμούς από το `0` έως και το `9`, με τη βοήθεια και της ενσωματωμένης συνάρτησης `range`:

Κώδικας 5.5

```
for i in range(10):
    print(i)
```



Η επικεφαλίδα ενός βρόγχου `for` ξεκινάει πάντα με τη δεσμευμένη λέξη `for`, έπειτα ακολουθεί μία μεταβλητή, η λέξη κλειδί `in`, συνήθως η συνάρτηση `range` και τελειώνει με άνω κάτω τελεία. Το μπλοκ των εντολών του βρόγχου πρέπει να είναι σε εσοχή, όπως είδαμε και στην εντολή `if`. Στο παραπάνω παράδειγμα η εντολή `print` εκτελείται 10 φορές. Αρχικά η μεταβλητή `i`

παίρνει την τιμή `0` και στην τελευταία επανάληψη την τιμή `9` (προσοχή: όχι την τιμή `10`). Σε κάθε επανάληψη η τιμή της μεταβλητής `i` αυξάνει κατά `1`.

Τι γίνεται στην περίπτωση που θέλουμε η μεταβλητή του βρόγχου `for` να μην ξεκινάει από το `0` ή/και να μην αυξάνει κατά `1` σε κάθε επανάληψη; Απλά συντάσσουμε με κατάλληλο τρόπο τη συνάρτηση `range`. Για παράδειγμα:

Κώδικας 5.6

```
for i in range(5, 10):
    print(i)
```


```
>>> ================================ RESTART ===
>>>
5
6
7
8
9
```

Κώδικας 5.7

```
for i in range(1, 20, 5):
    print(i)
```


```
>>> ================================ RESTART ===
>>>
1
6
11
16
```

Στον κώδικα 5.6 η μεταβλητή `i` παίρνει ως αρχική τιμή το `5`, τελική τιμή το `9` και σε κάθε επανάληψη αυξάνει κατά `1` (βήμα επανάληψης). Στον κώδικα 5.7 η μεταβλητή `i` παίρνει ως αρχική τιμή το `1`, σε κάθε επανάληψη αυξάνει κατά `5` (βήμα επανάληψης) και ως τελική τιμή παίρνει την τιμή `16` που πλησιάζει κοντύτερα και δεν ξεπερνάει το `20`.

Βλέπουμε ότι το βήμα επανάληψης είναι προαιρετικό. Αν δεν το ορίσουμε ρητά, τότε εξ ορισμού από την Python είναι `1`. Ακόμη, το βήμα μπορεί να είναι και αρνητικός αριθμός. Στην περίπτωση αυτή θα πρέπει η αρχική τιμή να είναι μεγαλύτερη από την τελική τιμή, π.χ. `range(100, 1, -10)`:

```
>>> for i in range(100, 1, -10):
        print(i)


100
90
80
70
60
50
40
30
20
10
>>>
```

Η εντολή `for` μπορεί να πάρει και διαφορετικές μορφές και να χρησιμοποιηθεί και για άλλους σκοπούς από αυτούς που είδαμε στα προηγούμενα παραδείγματα. Θα επανέλθουμε στη `for` με περισσότερα παραδείγματα σε επόμενα κεφάλαια.

Γράψτε κώδικα ο οποίος να περιέχει έναν βρόγχο `for` για εκτύπωση των αριθμών από το `1` έως και το `10`
Γράψτε κώδικα ο οποίος να περιέχει έναν βρόγχο `for` για αντίστροφη εκτύπωση των προηγούμενων αριθμών από το `10` έως και το `1`
Γράψτε κώδικα ο οποίος να διαβάζει την ηλικία σας και στη συνέχεια να τυπώνει τους άρτιους αριθμούς μέχρι να φτάσει την ηλικία.

5.5 Η εντολή επανάληψης `while`

Οι υπολογιστές χρησιμοποιούνται συχνά, για να κάνουν ανιαρές και επαναλαμβανόμενες εργασίες. Η επανάληψη ίδιων ή παραπλήσιων εργασιών χωρίς λάθη είναι κάτι που οι υπολογιστές κάνουν καλά και όχι οι άνθρωποι. Η Python μάς προσφέρει την εντολή `while`, η οποία επαναλαμβανόμενα εκτελεί μία ομάδα (μπλοκ) εντολών, όσο μια συνθήκη (προϋπόθεση) παραμένει αληθής.

Ας δούμε το παράδειγμα της εκτύπωσης των αριθμών από το `0` έως και το `9`:

Κώδικας 5.8

```
i = 0
while i < 10:
    print(i)
    i = i + 1
```

Run

```
>>> ================================ RESTART ===
>>>
0
1
2
3
4
5
6
7
8
9
```

Ένας βρόγχος `while` ξεκινάει με μία επικεφαλίδα που περιλαμβάνει τη δεσμευμένη λέξη `while`, μία συνθήκη (οποιαδήποτε Boolean έκφραση) και τελειώνει με μία άνω κάτω τελεία. Στη συνέχεια ακολουθεί σε εσοχή ένα μπλοκ εντολών («σώμα» του βρόγχου `while`) που επαναλαμβανόμενα εκτελούνται, όσο η συνθήκη είναι αληθής. Όταν η συνθήκη γίνει ψευδής, το «σώμα» δεν εκτελείται και η ροή εκτέλεσης των εντολών μεταφέρεται στην πρώτη εντολή μετά το βρόγχο `while`. Αν η συνθήκη είναι ψευδής από την πρώτη φορά που ελέγχεται, το «σώμα» δεν εκτελείται ποτέ. Το «σώμα» του βρόγχου θα πρέπει να αλλάζει τις τιμές μιας ή περισσότερων μεταβλητών, τις οποίες ελέγχει η συνθήκη στην επικεφαλίδα, ώστε η συνθήκη να γίνεται κάποια στιγμή ψευδής και ο βρόγχος να τερματίζεται. Διαφορετικά, ο βρόγχος θα επαναλαμβάνεται επ' άπειρον, γι΄αυτό και στην περίπτωση αυτή ονομάζεται **ατέρμων βρόγχος** (**infinite loop**). Κατά τη συγγραφή ενός προγράμματος δεν είναι πάντα εύκολο να αποδείξουμε ότι ένας βρόγχος `while` δεν θα είναι ατέρμων. Επίσης, η μεταβλητή ή οι μεταβλητές που αναφέρονται στη συνθήκη θα πρέπει να έχουν πάρει νωρίτερα στον κώδικά μας αρχικές τιμές με τις λεγόμενες εντολές αρχικοποίησης.

Οι εντολές `break` και `continue`

Η εντολή `break` χρησιμοποιείται για άμεση έξοδο από οποιοδήποτε σημείο ενός βρόγχου. Ας δούμε ένα παράδειγμα στο οποίο αθροίζουμε ένα άγνωστο πλήθος θετικών αριθμών:

Κώδικας 5.9

```
sum = 0
while True:
    n = int(input('Δώστε έναν θετικό αριθμό (-1 για τέλος):'))
    if n == -1:
        break
    sum = sum + n
print('Άθροισμα =', sum)
```

```
>>> ================================ RESTART ===
>>>
Δώστε έναν θετικό αριθμό (-1 για τέλος):4
Δώστε έναν θετικό αριθμό (-1 για τέλος):3
Δώστε έναν θετικό αριθμό (-1 για τέλος):10
Δώστε έναν θετικό αριθμό (-1 για τέλος):3
Δώστε έναν θετικό αριθμό (-1 για τέλος):-1
Άθροισμα = 20
```

Στον κώδικα 5.9 η συνθήκη του βρόγχου `while` είναι `True` και αυτό σημαίνει ότι είναι πάντα αληθής και ο βρόγχος θα επαναλαμβάνεται διαρκώς εκτός και εάν εκτελεστεί η εντολή `break`. Ο μόνος τρόπος να εκτελεστεί η `break` είναι να δώσουμε ως είσοδο τον αριθμό `-1`.

Γενικά δεν είναι καλή πρακτική προγραμματισμού η χρήση της εντολής `break`. Αν κάποια στιγμή τη χρησιμοποιήσετε, καλό είναι ο κώδικας που θα προκύψει να είναι ευανάγνωστος και ξεκάθαρος για το πώς δουλεύει.

Η Python διαθέτει και μία άλλη σχετική εντολή, την `continue`. Όταν εκτελεστεί η `continue` στο «σώμα» ενός βρόγχου, ο έλεγχος ροής μεταφέρεται στη συνθήκη του βρόγχου. Κι αυτή η εντολή καλό είναι να αποφεύγεται, ειδικά από αρχάριους προγραμματιστές.

5.6 Σύγκριση της `for` με τη `while`

Στην εντολή `for` ο αριθμός των επαναλήψεων είναι προκαθορισμένος, ενώ στην εντολή `while` δεν είναι πάντα. Επίσης, στην εντολή `while` υπάρχουν περιπτώσεις που δεν μπορούμε να αποδείξουμε διαβάζοντας απλά τον κώδικα αν θα έχουμε ή όχι ατέρμονα βρόγχο κατά την εκτέλεση. Συνήθως η `while` μας δίνει πιο πολύπλοκο κώδικα.

Ας δούμε ένα παράδειγμα στο οποίο θα λύσουμε το πρόβλημα του υπολογισμού του παραγοντικού ενός αριθμού, πρώτα με τη `for` και στη συνέχεια εναλλακτικά με τη `while`. Το παραγοντικό ενός αριθμού `n` (συμβολίζεται `n!`) εκφράζει με πόσους τρόπους μπορούν να διαταχθούν `n` αντικείμενα σε μια γραμμή και ισούται με `1·2·3· … ·n`. Για παράδειγμα, τα τέσσερα γράμματα ΑΒΓΔ μπορούν να διαταχθούν με `1·2·3·4 = 24` διαφορετικούς τρόπους.

Κώδικας 5.10

```
# υπολογισμός παραγοντικού με for
n = int(input('Δώστε έναν θετικό αριθμό:'))
f = 1
for i in range(2, n+1):
    f = f * i
print(str(n) + ' παραγοντικό = ' + str(f))

# υπολογισμός παραγοντικού με while
n = int(input('Δώστε έναν θετικό αριθμό:'))
f = 1
i = 2
while i <= n:
    f = f * i
    i = i + 1
print(str(n) + ' παραγοντικό = ' + str(f))
```


```
>>> ================================ RESTART ===
>>>
Δώστε έναν θετικό αριθμό:4
4 παραγοντικό = 24
Δώστε έναν θετικό αριθμό:4
4 παραγοντικό = 24
```

Γράψτε κώδικα για τον υπολογισμό του αθροίσματος ενός προκαθορισμένου πλήθους αριθμών που δίνονται από το πληκτρολόγιο. Δώστε μία εκδοχή με χρήση της `for` *και μία εκδοχή με χρήση της* `while`

Μπορούμε να έχουμε και έναν βρόγχο μέσα σε έναν άλλο βρόγχο (**εμφωλευμένος βρόγχος**). Η δομή αυτή αποτελεί συνηθισμένη πρακτική σε αρκετά προγράμματα. Για παράδειγμα, ας τυπώσουμε την προπαίδεια του 8 και του 9:

Κώδικας 5.11

```
for i in range(8,10):
    print('-----')
    for j in range (1, 11):
        print(i, '*', j, '=', i*j)
```


```
>>> ================================ RESTART ===
>>>
-----
8 * 1 = 8
8 * 2 = 16
8 * 3 = 24
8 * 4 = 32
8 * 5 = 40
8 * 6 = 48
8 * 7 = 56
8 * 8 = 64
8 * 9 = 72
8 * 10 = 80
-----
9 * 1 = 9
9 * 2 = 18
9 * 3 = 27
9 * 4 = 36
9 * 5 = 45
9 * 6 = 54
9 * 7 = 63
9 * 8 = 72
9 * 9 = 81
9 * 10 = 90
```

Ας δούμε ένα επιπλέον παράδειγμα στο οποίο καλούμαστε να γράψουμε κώδικα για την καταμέτρηση του πλήθους των δεκαδικών ψηφίων ενός θετικού ακέραιου αριθμού:

Κώδικας 5.12

```
n = int(input('Δώστε έναν θετικό ακέραιο αριθμό:'))
count = 0
while n != 0:
    count = count + 1
    n = n // 10
print('Πλήθος ψηφίων =', count)
```


```
>>> ================================ RESTART ===
>>>
Δώστε έναν θετικό ακέραιο αριθμό:14500
Πλήθος ψηφίων = 5
```

Τροποποιήστε κατάλληλα τον παραπάνω κώδικα ώστε να μετράει μόνο το πλήθος των μηδενικών ψηφίων σε έναν θετικό ακέραιο αριθμό

Να σημειώσουμε στο σημείο αυτό ότι η καλή γνώση Μαθηματικών είναι πάντα χρήσιμη στους προγραμματιστές!

Κεφάλαιο 6

Συναρτήσεις

Οι **συναρτήσεις** στον προγραμματισμό είναι επαναχρησιμοποιήσιμα μέρη προγραμμάτων. Μια συνάρτηση μας επιτρέπει να δίνουμε ένα όνομα σε μια ομάδα εντολών και να την εκτελούμε χρησιμοποιώντας το όνομά της οπουδήποτε στο πρόγραμμά μας και για όσες φορές θέλουμε. Επίσης, μια συνάρτηση μπορεί να δεχθεί είσοδο και να παραγάγει έξοδο (επιστρεφόμενη τιμή). 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 συναρτήσεις γενικά διευκολύνουν τη συγγραφή, ανάγνωση, κατανόηση και διόρθωση ενός προγράμματος.

Οι συναρτήσεις μπορούν να αποθηκευτούν και σε αρχεία για μελλοντική χρήση. Γενικότερα, χρήσιμες ομάδες εντολών «αποθηκεύονται» σε κατάλληλες συναρτήσεις. Η Python μάς προσφέρει άριστη υποστήριξη στις συναρτήσεις που γράφουμε και επιπλέον μάς παρέχει και πολλές έτοιμες (ενσωματωμένες συναρτήσεις). Η πρακτική εξάσκηση στις συναρτήσεις θα σας βοηθήσει να βάλετε ένα ακόμα γερό λιθαράκι στην κατάκτηση της Python. Κατά τη συγγραφή μεγάλων και πολύπλοκων προγραμμάτων η χρήση κατάλληλων συναρτήσεων είναι θεμελιώδης για την επιτυχή έκβαση της προσπάθειας του προγραμματιστή.

6.1 Ορισμός και κλήση συνάρτησης

Η πρώτη γραμμή του ορισμού μιας συνάρτησης είναι η επικεφαλίδα της συνάρτησης. Η επικεφαλίδα ξεκινάει με τη δεσμευμένη λέξη `def` ακολουθούμενη από το όνομα της συνάρτησης, ένα ζευγάρι παρενθέσεων που προαιρετικά περικλείει μια διαχωριζόμενη με κόμματα λίστα παραμέτρων (parameters) και τελειώνει με μια άνω κάτω τελεία. Κάτω από την επικεφαλίδα ακολουθεί το σώμα της συνάρτησης (οι εντολές της συνάρτησης) σε εσοχή, όπως είδαμε και στις εντολές `if`, `while` και `for`.

Τα ονόματα των συναρτήσεων δίνονται με βάση τους ίδιους κανόνες που είδαμε και για τα ονόματα των μεταβλητών. Καλό είναι να δίνουμε ονόματα σχετικά με τη λειτουργία που επιτελούν οι συναρτήσεις. Οι συναρτήσεις ορίζονται συνήθως στην αρχή των προγραμμάτων. Είναι προφανές όμως ότι μια συνάρτηση πρέπει να οριστεί πρώτα πριν χρησιμοποιηθεί (κληθεί). Κατά την **κλήση** μιας συνάρτησης εκτελούνται οι εντολές που περιέχονται στο σώμα της. Μπορούμε να καλούμε μία συνάρτηση όσες φορές απαιτούνται για τη λύση του προβλήματος που αντιμετωπίζει το πρόγραμμά μας. Μάλιστα κάτι τέτοιο είναι πολύ συνηθισμένο και χρήσιμο.

Ας δούμε ένα απλό παράδειγμα στο οποίο δεν έχουμε παραμέτρους και επιστρεφόμενη τιμή. Η παρακάτω συνάρτηση περιέχει μόνο μία εντολή για την εκτύπωση ενός μηνύματος:

Κώδικας 6.1

```
def sayHello():      # ορισμός συνάρτησης
    print('Hello')

sayHello()           # κλήση συνάρτησης
```

Run

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

Μία συνάρτηση μπορεί να δεχθεί **παραμέτρους (formal parameters** ή απλά **parameters)**, οι οποίες βοηθούν να δοθούν τιμές στη συνάρτηση ως είσοδος, έτσι ώστε αυτή να μπορεί να κάνει κάτι αξιοποιώντας αυτές τις τιμές. Οι παράμετροι μοιάζουν με τις μεταβλητές και οι τιμές τους ορίζονται, όταν καλούμε μία συνάρτηση. Οι παράμετροι υπάρχουν μόνο κατά τη διάρκεια εκτέλεσης της συνάρτησης. Κατά την κλήση μιας συνάρτησης οι τιμές που παίρνουν οι παράμετροι ονομάζονται **ορίσματα (actual arguments** ή απλά **arguments)**. Με απλά λόγια, για να μην μπερδευόμαστε, στον ορισμό μιας συνάρτησης έχουμε παραμέτρους (ονομασίες μεταβλητών) και στην κλήση μιας συνάρτησης έχουμε ορίσματα (τιμές ή μεταβλητές στις οποίες έχουν εκχωρηθεί τιμές). Ας δούμε ένα παράδειγμα στο οποίο ορίζουμε και καλούμε μία συνάρτηση η οποία συγκρίνει δύο αριθμούς:

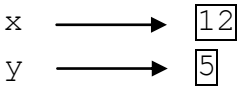
Κώδικας 6.2

```
def compare(a, b): # ορισμός της συνάρτησης compare με παραμέτρους a, b
    if a > b:
        print(a, '>', b)
    elif a < b:
        print(a, '<', b)
    else:
        print(a, '=', b)

compare(7,9)    # κλήση συνάρτησης με ορίσματα απευθείας τιμές
x = 12
y = 5
compare(x,y)    # κλήση συνάρτησης με ορίσματα μεταβλητές
compare(x,12)   # κλήση συνάρτησης με ορίσματα μεταβλητή και τιμή
```

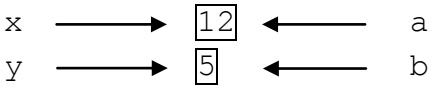

```
>>> ================================ RESTART ===
>>>
7 < 9
12 > 5
12 = 12
```

Στο παραπάνω παράδειγμα (κώδικας 6.2) με την κλήση `compare(7,9)` τα ορίσματα είναι οι τιμές `7` και `9`, και δίνονται ως τιμές στις παραμέτρους `a` και `b` αντίστοιχα. Με την κλήση `compare(x,y)` τα ορίσματα είναι οι μεταβλητές `x` και `y`, και οι τιμές στις οποίες δείχνουν δίνονται ως τιμές στις παραμέτρους `a` και `b` αντίστοιχα. Η τεχνική αυτή περάσματος παραμέτρων σε μία συνάρτηση ονομάζεται **pass by reference** (πέρασμα μέσω αναφοράς). Σχηματικά για τον κώδικα 6.2 έχουμε:

- πριν την κλήση `compare(x,y)`



- κατά την κλήση `compare(x,y)`



Βλέπουμε ότι οι τιμές δεν αντιγράφονται, απλά παίρνουν νέα ονόματα που χρησιμοποιούνται από τη συνάρτηση για αναφορά σε αυτές. Μετά την κλήση της συνάρτησης οι αναφορές των παραμέτρων διαγράφονται αυτόματα. Θα μπορούσαμε να δώσουμε ως ονόματα παραμέτρων αντί των `a` και `b` τα `x` και `y`. Για να μη δημιουργείται όμως σύγχυση, καλό είναι να μη δίνουμε ως ονόματα παραμέτρων ονόματα που έχουμε δώσει νωρίτερα σε μεταβλητές.

Τι θα τυπωθεί και γιατί όταν εκτελεστεί το παρακάτω πρόγραμμα;

```
def set10(x):
    x = 10

y = 0
set10(y)
print(y)
```

Μία συνάρτηση μπορεί να επιστρέφει και τιμή, η οποία μπορεί να εκχωρηθεί σε μία μεταβλητή ή ακόμη και να χρησιμοποιηθεί ως μέρος μιας έκφρασης. Η επιστροφή τιμής γίνεται με την εντολή `return` η οποία σημαίνει: «επίστρεψε αμέσως τον έλεγχο ροής από τη συνάρτηση στο σημείο του προγράμματος που αυτή κλήθηκε και χρησιμοποίησε την τιμή της έκφρασης που ακολουθεί ως επιστρεφόμενη τιμή». Ας δούμε το παράδειγμα μιας συνάρτησης που δέχεται ως όρισμα έναν αριθμό και επιστρέφει την απόλυτη τιμή του:

Κώδικας 6.3

```
def absoluteValue(x):
    if x < 0:
        return -x
    else:
        return x

print(absoluteValue(5))
print(absoluteValue(-5))
```


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

Ας δούμε και το παράδειγμα μιας συνάρτησης που δέχεται ως ορίσματα δύο αριθμούς ή δύο συμβολοσειρές και επιστρέφει ως τιμή το άθροισμα των αριθμών ή τη συνένωση των συμβολοσειρών αντίστοιχα:

Κώδικας 6.4

```
def add(x, y):
    return x + y

print(add(7,5))
print(add('Hello ','World'))
```


```
>>> ================================ RESTART ===
>>>
12
Hello World
```

Ας αλλάξουμε τον κώδικα 6.4 αφαιρώντας την εντολή `return`, ώστε η συνάρτησή μας να μην επιστρέφει τιμή. Τι θα γίνει, αν προσπαθήσουμε να εκχωρήσουμε την κλήση της συνάρτησης σε μία μεταβλητή; Θα εμφανιστεί μήνυμα λάθους; Ας δοκιμάσουμε:

Κώδικας 6.5

```
def add2(x, y):
    print(x + y)

add2(7,5)
sum = add2(7,5)
print(sum)
```


```
>>> ================================ RESTART ===
>>>
12
12
None
```

Βλέπουμε ότι στη μεταβλητή `sum` εκχωρήθηκε μία ειδική τιμή, η `None`. Η `None` υποδεικνύει ότι η συνάρτηση `add2` δεν επιστρέφει τιμή. Δεν είναι ούτε αριθμός ούτε συμβολοσειρά, ώστε να μπορούμε να κάνουμε κάποιον υπολογισμό με αυτή.

6.2 Εμβέλεια μεταβλητών

Η εμβέλεια μιας μεταβλητής ή μιας συνάρτησης αφορά στην περιοχή του προγράμματος στην οποία μπορεί να προσπελαστεί ή με άλλα λόγια να είναι ορατή. Όταν δημιουργούμε μία μεταβλητή μέσα σε μία συνάρτηση, αυτή ονομάζεται **τοπική μεταβλητή** (**local variable**) και «υπάρχει» μόνο μέσα στη συνάρτηση, δεν μπορούμε να τη χρησιμοποιήσουμε έξω από αυτή. Οι τοπικές μεταβλητές «ζουν» όσο διαρκεί η εκτέλεση της συνάρτησης στην οποία έχουν οριστεί, και καταστρέφονται, όταν ολοκληρωθεί η εκτέλεση της συνάρτησης. Κάθε νέα κλήση μιας συνάρτησης δημιουργεί νέες τοπικές μεταβλητές. Οι παράμετροι μιας συνάρτησης θεωρούνται επίσης τοπικές μεταβλητές. Ας δούμε ένα παράδειγμα:

Κώδικας 6.6

```
def add3(x, y):
    s = x + y
    return s

print(add3(7,3))
print(s)
```


```
>>> ================================ RESTART ================================
>>>
10
Traceback (most recent call last):
  File "C:/Users/User/Desktop/code/6.6.py", line 7, in <module>
    print(s)
NameError: name 's' is not defined
```

Στον κώδικα 6.6 η μεταβλητή `s` είναι τοπική στη συνάρτηση `add3`. Αν προσπαθήσουμε να την προσπελάσουμε έξω από αυτή, τότε θα εμφανιστεί μήνυμα λάθους (`NameError: name 's' is not defined`). Επίσης, οι παράμετροι `x` και `y` είναι τοπικές μεταβλητές στη συνάρτηση `add3`.

Οι μεταβλητές που δηλώνονται έξω από τις συναρτήσεις ενός προγράμματος ονομάζονται **καθολικές μεταβλητές (global variables)** και μπορούν να προσπελαστούν από οποιοδήποτε σημείο μέσα στο πρόγραμμα συμπεριλαμβανομένων και των συναρτήσεων. Τι γίνεται όμως στην περίπτωση που έχουμε δώσει το ίδιο όνομα σε καθολική και τοπική μεταβλητή; Ας δούμε ένα παράδειγμα:

Κώδικας 6.7

```
name = 'Mariza'
def sayHello():
    print('Hello ' + name)

def changeName(new_name):
    name = new_name

sayHello()
changeName('Katerina')
sayHello()
```

Run

```
>>> ================================ RESTART ===
>>>
Hello Mariza
Hello Mariza
```

Στον κώδικα 6.7 μέσα στη συνάρτηση `changeName` έχουμε την τοπική μεταβλητή `name`, η οποία δεν έχει καμία σχέση με την καθολική μεταβλητή `name` που δηλώθηκε στην αρχή του προγράμματος. Αν θέλουμε να προσπελάσουμε την καθολική μεταβλητή `name` μέσα στη συνάρτηση `changeName`, θα πρέπει να χρησιμοποιήσουμε την εντολή `global` ως εξής:

Κώδικας 6.8

```
name = 'Mariza'
def sayHello():
    print('Hello ' + name)

def changeName(new_name):
    global name
    name = new_name

sayHello()
changeName('Katerina')
sayHello()
```


```
>>> ================================ RESTART ===
>>>
Hello Mariza
Hello Katerina
```

6.3 Συμβολοσειρές τεκμηρίωσης (doc strings)

Με τις **συμβολοσειρές τεκμηρίωσης** (**doc strings**) μπορούμε προαιρετικά να τεκμηριώσουμε μία συνάρτηση: τι υποτίθεται ότι θα κάνει κατά την κλήση της και οποιαδήποτε άλλη χρήσιμη πληροφορία. Ένα doc string γράφεται στην αρχή μιας συνάρτησης και οριοθετείται με τριπλά εισαγωγικά. Για παράδειγμα:

```
>>> def sum(x, y):
        """Αθροίζει δύο αριθμούς ή συνενώνει δύο συμβολοσειρές"""
        return(x+y)

>>> sum(12,8)
20
>>> print(sum.__doc__)
Αθροίζει δύο αριθμούς ή συνενώνει δύο συμβολοσειρές
>>>
```

6.4 Προεπιλεγμένα ορίσματα

Μπορούμε, αν θέλουμε, σε μία συνάρτηση να έχουμε μερικές παραμέτρους προαιρετικές και να χρησιμοποιούμε προεπιλεγμένες τιμές, εάν ο χρήστης δεν θέλει να δώσει τιμές σε αυτές τις παραμέτρους. Ο καθορισμός προεπιλεγμένων τιμών ορισμάτων για παραμέτρους γίνεται τοποθετώντας μετά το όνομα της παραμέτρου στον ορισμό της συνάρτησης τον τελεστή εκχώρησης (=) ακολουθούμενο από την προεπιλεγμένη τιμή. Για παράδειγμα:

Κώδικας 6.9

```
def greet(name, greeting = 'Hello'):
    print(greeting, name)

greet('Antonis')
greet('Antonis', 'Good bye')
```


```
>>> ================================ RESTART ===
>>>
Hello Antonis
Good bye Antonis
```

6.5 Ορίσματα με λέξεις κλειδιά

Μπορούμε επίσης, αν θέλουμε, σε μία συνάρτηση να ορίσουμε μερικές παραμέτρους χρησιμοποιώντας λέξεις κλειδιά. Τα βασικά πλεονεκτήματα της προσέγγισης αυτής είναι η δημιουργία ευανάγνωστων συναρτήσεων και η ελευθερία στη διάταξη των ορισμάτων. Δίνουμε τιμές μόνο στις παραμέτρους που θέλουμε, οι υπόλοιπες θα παίρνουν τις προεπιλεγμένες τιμές. Για παράδειγμα:

Κώδικας 6.10

```
def func(a, b=1, c=2):
    print('a =', a, 'και b =', b, 'και c =', c)

func(5, 10)
func(2, c=20)
func(c=50, a=10)
```


```
>>> ================================ RESTART ===
>>>
a = 5 και b = 10 και c = 2
a = 2 και b = 1 και c = 20
a = 10 και b = 1 και c = 50
```

6.6 Αναδρομή

Μία συνάρτηση μπορεί εκτός από το να καλεί μία άλλη συνάρτηση, να καλεί και τον εαυτό της. Ίσως να μην είναι προφανές για έναν αρχάριο προγραμματιστή, επειδή αυτό μπορεί να είναι κάτι καλό, αλλά σίγουρα είναι ένα από τα πιο κομψά πράγματα που μπορούμε να κάνουμε σε μια γλώσσα προγραμματισμού. Η διαδικασία κατά την οποία μια συνάρτηση καλεί τον εαυτό της ονομάζεται **αναδρομή,** και η συνάρτηση λέγεται **αναδρομική συνάρτηση**.

Ας δούμε το παράδειγμα του υπολογισμού του **παραγοντικού** ενός αριθμού. Το παραγοντικό μπορεί να οριστεί με τον παρακάτω αναδρομικό ορισμό:

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

Ο ορισμός αυτός μας λέει ότι το παραγοντικό του μηδενός είναι `1` και το παραγοντικό κάθε άλλου `n` είναι το `n` πολλαπλασιαζόμενο με το παραγοντικό του

n-1. Με βάση τον παραπάνω αναδρομικό ορισμό μπορούμε να γράψουμε εύκολα την αναδρομική συνάρτηση που υπολογίζει το παραγοντικό:

Κώδικας 6.11

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)

x = 4
print(x,'! =',factorial(x))
```

Run

```
>>> ================================ RESTART ===
>>>
4 ! = 24
```

Μετά το παραγοντικό το συνηθέστερο παράδειγμα μαθηματικής συνάρτησης αναδρομικά ορισμένης είναι η **Fibonacci**:

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

Η Fibonacci παράγει μια σειρά αριθμών όπου ο καθένας είναι το άθροισμα των δύο προηγούμενων: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89,…

Γράψτε τον κώδικα αναδρομικής συνάρτησης για τον υπολογισμό της σειράς
Γράψτε τον κώδικα αναδρομικής συνάρτησης για την αντίστροφη εκτύπωση αριθμών, από έναν θετικό ακέραιο n *έως και το* μηδέν

6.7 Συνάρτηση `main`

Μία καλή πρακτική προγραμματισμού είναι ο ορισμός και η χρήση μιας τουλάχιστον συνάρτησης σε κάθε πρόγραμμα που γράφουμε και με όνομα το `main`. Σε άλλες γλώσσες προγραμματισμού, όπως η C και η Java, η συνάρτηση `main` είναι μέρος της γλώσσας. Η `main` δηλώνει την αρχή ενός προγράμματος και είναι προαιρετική στην Python. Ας δούμε ένα παράδειγμα:

Κώδικας 6.12

```
def main():
    pwd = input('Δώστε τον κωδικό πρόσβασης:')
    if pwd == 'heraklion':
        print('Είσοδος...')
    else:
        print('Λανθασμένος κωδικός προσβασης')

main()
```


```
>>> ================================ RESTART ===
>>>
Δώστε τον κωδικό πρόσβασης:heraklion
Είσοδος...
```

6.8 Αρθρώματα (Modules)

Ένα **άρθρωμα** (**module**) είναι μία συλλογή σχετιζόμενων συναρτήσεων. Ένα άρθρωμα αποθηκεύεται σε ένα αρχείο με κατάληξη `.py`. Η Python διαθέτει πάρα πολλά ενσωματωμένα αρθρώματα. Μπορούμε να κατεβάσουμε και αρκετά αρθρώματα από το Διαδίκτυο, τα οποία μπορούν να μας βοηθήσουν, για παράδειγμα, στην ανάπτυξη παιχνιδιών ή στη δημιουργία γραφικών. Μπορούμε, αν θέλουμε, να γράψουμε και τα δικά μας αρθρώματα, απλά αποθηκεύοντας τις σχετιζόμενες συναρτήσεις μας σε ένα αρχείο με κατάληξη `.py`.

Για να χρησιμοποιήσουμε ένα άρθρωμα σε ένα πρόγραμμα, θα πρέπει να το εισαγάγουμε με χρήση της εντολής `import`. Ένα πάρα πολύ χρήσιμο παράδειγμα αρθρώματος είναι το `math` που περιέχει μαθηματικές συναρτήσεις. Αφού το εισαγάγουμε με την εντολή `import math`, μπορούμε να δούμε τις συναρτήσεις που περιέχει με την εντολή `help(math)`. Για παράδειγμα:

```
>>> import math
>>> print(math.pi)
3.141592653589793
>>> x = 3 + math.log10(100)
>>> print(x)
5.0
>>> degrees = 45
>>> radians = degrees / 360 * 2 * math.pi
>>> math.sin(radians)
0.7071067811865475
>>>
```

Μπορούμε να εισαγάγουμε μεμιάς όλες τις συναρτήσεις ενός αρθρώματος. Για παράδειγμα:

```
>>> from math import *
>>> pi
3.141592653589793
>>> 3 + log10(100)
5.0
```

Με την ενσωματωμένη συνάρτηση `dir` μπορούμε να δούμε όλα τα ονόματα που περιέχονται σε ένα άρθρωμα, για παράδειγμα:

```
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'a
cosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos'
, 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial
', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'hypot', 'isfinite', 'isinf',
'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'pi',
'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
>>>
```

6.9 Επιπλέον θέματα

Μέχρι τώρα έχουμε καλύψει ένα μικρό υποσύνολο της Python, αλλά το ενδιαφέρον είναι ότι αυτό το μικρό υποσύνολο είναι μια πλήρης γλώσσα προγραμματισμού, υπό την έννοια ότι οτιδήποτε που μπορεί να υπολογιστεί, μπορεί να εκφραστεί σε αυτό το υποσύνολο. Η απόδειξη ενός τέτοιου ισχυρισμού κάθε άλλο παρά τετριμμένη είναι, και έγινε για πρώτη φορά από τον **Alan Turing**, ένα από τους πρώτους επιστήμονες των υπολογιστών. Γι' αυτό και ονομάζεται **Turing Thesis**.

Επανεκχώρηση ονομάτων συναρτήσεων

Πρέπει να προσέχουμε, ώστε να μην εκχωρήσουμε κατά λάθος τιμή σε μια ενσωματωμένη συνάρτηση, διότι μετέπειτα δεν θα μπορέσουμε να τη χρησιμοποιήσουμε όπως αυτή έχει οριστεί. Για παράδειγμα, θα ήταν λάθος να γράψουμε τον παρακάτω κώδικα:

```
>>> dir = 10
>>> print(dir)
10
>>> dir(math)
Traceback (most recent call last):
  File "<pyshell#73>", line 1, in <module>
    dir(math)
TypeError: 'int' object is not callable
>>>
```

Άπειρη αναδρομή

Όταν γράφουμε μια αναδρομική συνάρτηση, θα πρέπει να προσέχουμε το θέμα του τερματισμού της. Εάν μια αναδρομή δεν φτάνει ποτέ τη λεγόμενη «περίπτωση βάσης», θα συνεχίζει να κάνει αναδρομικές κλήσεις επ' άπειρον και το πρόγραμμα δεν θα τερματίζει ποτέ. Στην Python δεν επιτρέπεται η άπειρη αναδρομή, επειδή απλά κάποτε εξαντλείται η μνήμη του υπολογιστή:

```
>>> def recurse():
        recurse()


>>> recurse()
Traceback (most recent call last):
  File "<pyshell#80>", line 1, in <module>
    recurse()
  File "<pyshell#79>", line 2, in recurse
    recurse()
  File "<pyshell#79>", line 2, in recurse
    recurse()
RuntimeError: maximum recursion depth exceeded
>>>
```

Πολλαπλή εκχώρηση τιμών

Η Python υποστηρίζει πολλαπλή εκχώρηση τιμών. Για παράδειγμα:

```
>>> x, y = 5, 10
>>> print(x)
5
>>> print(y)
10
>>>
```

Εκτύπωση πινάκων

Ας δούμε ένα παράδειγμα εκτύπωσης πίνακα πολλαπλασιασμού:

Κώδικας 6.13

```
def printMultiples(n):
    i = 1
    while i <= 6:
        print(n*i, '\t', end='')
        i = i + 1
    print()

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


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

Η συμβολοσειρά `'\t'` παριστάνει τον χαρακτήρα «tab» (από τη λέξη tabulation, που σημαίνει εκτύπωση πίνακα δεδομένων). Το όρισμα `end=' '` στην `print` χρησιμοποιείται ώστε σε κάθε επανάληψη να διατηρείται η εκτύπωση στην ίδια γραμμή. Με την `print()` γίνεται αλλαγή γραμμής.

Έλεγχος τύπων

Η Python μάς διαθέτει την ενσωματωμένη συνάρτηση `isinstance` για έλεγχο του τύπου μιας μεταβλητής. Για παράδειγμα:

```
>>> i = 2
>>> s = 'Hello'
>>> isinstance(i, int)
True
>>> isinstance(i, str)
False
>>> isinstance(s, str)
True
>>> isinstance(s, int)
False
>>>
```

Ανάπτυξη προγράμματος κατά στάδια

Στην κατασκευή ολοένα και πιο πολύπλοκων προγραμμάτων βοηθάει μια τεχνική που ονομάζεται **ανάπτυξη προγράμματος κατά στάδια** (**incremental development**). Η ιδέα στην τεχνική αυτή είναι η αποφυγή χρονοβόρων διαδικασιών αποσφαλμάτωσης χάρη στη σταδιακή προσθήκη και επαλήθευση μικρών κομματιών κώδικα.

Στην ανάπτυξη προγράμματος κατά στάδια αρχίζουμε πάντα με ένα μικρό πρόγραμμα που δουλεύει, και στη συνέχεια κάνουμε μικρές αυξητικές αλλαγές. Χρησιμοποιούμε προσωρινές (ενδιάμεσες) μεταβλητές και εντολές έτσι ώστε να μπορούμε να τυπώνουμε ενδιάμεσες τιμές και να ελέγχουμε την ορθότητα και τη λειτουργικότητα του προγράμματός μας. Ο ενδιάμεσος κώδικας («σκαλωσιά») βοηθάει στην κατασκευή, όπως οι σκαλωσιές σε μια οικοδομή, αλλά δεν αποτελεί τμήμα του τελικού προϊόντος. Μόλις βεβαιωθούμε ότι το τελικό πρόγραμμα δουλεύει σωστά, αφαιρούμε τις «σκαλωσιές» και συγκεντρώνουμε πολλαπλές εντολές σε σύνθετες εκφράσεις, μόνο βέβαια εφόσον δεν κάνουν δυσανάγνωστο το πρόγραμμά μας.

Κεφάλαιο 7

Συμβολοσειρές

Μια **συμβολοσειρά** (**string**) είναι μια ακολουθία από χαρακτήρες (αριθμητικά ψηφία, γράμματα, σύμβολα) και ανήκει στον τύπο `str`. Μπορούμε να ορίσουμε συμβολοσειρές με μονά, διπλά ή και τριπλά εισαγωγικά. Για παράδειγμα:

```
>>> s1 = 'Hello'
>>> s2 = "Hello"
>>> print(s1)
Hello
>>> print(s2)
Hello
>>> type('Hello')
<class 'str'>
>>> s3 = '''Hello,
World!'''
>>> print(s3)
Hello,
World!
>>>
```

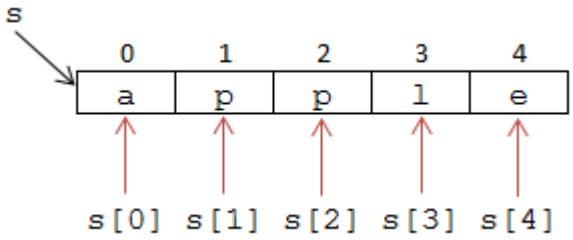
Σε αντίθεση με τους τύπους `int` και `float`, ο τύπος `str` αποτελείται από μικρότερα κομμάτια, τους χαρακτήρες, γι' αυτό και λέγεται **σύνθετος**. Ανάλογα με το τι θέλουμε να κάνουμε στο πρόγραμμά μας, μεταχειριζόμαστε έναν σύνθετο τύπο είτε ως ενιαίο σύνολο είτε ως αποτελούμενο από διακριτά μέρη. Τις συμβολοσειρές μπορούμε να τις διαβάσουμε από το πληκτρολόγιο ως είσοδο, μπορούμε να τις τυπώσουμε και στην οθόνη του υπολογιστή μας. Θα δούμε σε επόμενο κεφάλαιο ότι τα αρχεία κειμένων τα χειριζόμαστε συνήθως ως μεγάλες συμβολοσειρές.

7.1 Προσπέλαση με δείκτες

Κατά την εργασία μας με συμβολοσειρές απαιτείται πολλές φορές να προσπελάσουμε ατομικά τους χαρακτήρες από τους οποίους αποτελούνται. Για να γίνει αυτό, χρησιμοποιούμε **δείκτες**. Γενικά ένας δείκτης (index) αναφέρεται σε μέλος ενός διατεταγμένου συνόλου, στη δική μας περίπτωση του συνόλου χαρακτήρων μιας συμβολοσειράς. Δείκτης μπορεί να είναι κάθε ακέραιος αριθμός ή έκφραση. Ας δούμε ένα παράδειγμα:

```
>>> s = 'apple'
>>> s[0]
'a'
>>> s[1]
'p'
>>> s[2]
'p'
>>> s[3]
'l'
>>> s[4]
'e'
>>>
```

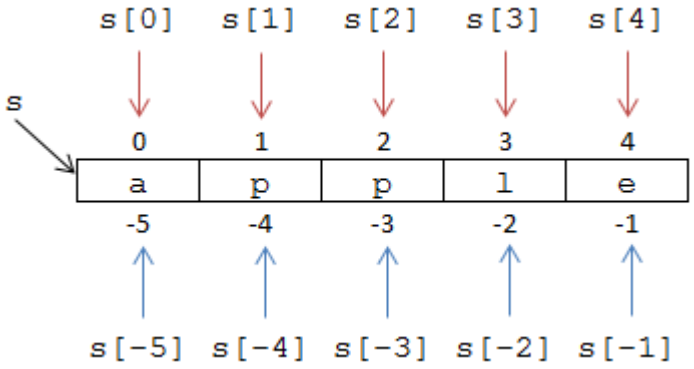
Βλέπουμε ότι, για να προσπελάσουμε έναν χαρακτήρα μιας συμβολοσειράς, χρησιμοποιούμε το όνομα της συμβολοσειράς ακολουθούμενο από ένα ζευγάρι αγκυλών που περιέχουν έναν δείκτη. Ο δείκτης υποδεικνύει τον χαρακτήρα που θέλουμε να προσπελάσουμε κάθε φορά. Η αρίθμηση των δεικτών ξεκινάει από το μηδέν. Υποθέστε ότι ένας δείκτης μετράει την «απόσταση» από τον πρώτο χαρακτήρα μιας συμβολοσειράς, όπως ακριβώς και ένας χάρακας ξεκινάει από το μηδέν. Για καλύτερη κατανόηση ας δούμε το παραπάνω παράδειγμα με μορφή διαγράμματος:



Για μια συμβολοσειρά με όνομα `s` και μήκος `n` (πλήθος χαρακτήρων), `s[0]` είναι ο πρώτος χαρακτήρας, `s[1]` είναι ο δεύτερος χαρακτήρας, `s[2]` είναι ο τρίτος χαρακτήρας, κ.ο.κ. μέχρι τον τελευταίο χαρακτήρα που είναι ο `s[n-1]`. Αν προσπαθήσουμε να χρησιμοποιήσουμε δείκτη έξω από τα όρια, θα πάρουμε μήνυμα σφάλματος:

```
>>> s[5]
Traceback (most recent call last):
  File "<pyshell#35>", line 1, in <module>
    s[5]
IndexError: string index out of range
>>>
```

Εναλλακτικά, μπορούμε να χρησιμοποιήσουμε και αρνητικούς δείκτες, οι οποίοι μετράνε από το τέλος της συμβολοσειράς προς την αρχή. Ο δείκτης `-1` δίνει τον τελευταίο χαρακτήρα μιας συμβολοσειράς, ο `-2` τον προτελευταίο, κ.ο.κ. μέχρι τον `-n` (πλήθος των χαρακτήρων με αρνητικό πρόσημο) που δίνει τον πρώτο χαρακτήρα. Ας δούμε σε διάγραμμα το παράδειγμα της συμβολοσειράς '`apple`':




```
>>> s[-1]
'e'
>>> s[-2]
'l'
>>> s[-3]
'p'
>>> s[-4]
'p'
>>> s[-5]
'a'
>>>
```

7.2 Πέρασμα συμβολοσειράς με βρόγχο `while` και `for`

Η Python μάς προσφέρει την ενσωματωμένη συνάρτηση `len`, η οποία επιστρέφει το πλήθος των χαρακτήρων μιας συμβολοσειράς ή αλλιώς το μήκος μιας συμβολοσειράς. Για παράδειγμα:

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

Σε πολλά προγράμματα απαιτείται η επεξεργασία συμβολοσειρών χαρακτήρα-χαρακτήρα. Συνήθως η επεξεργασία αυτή ξεκινάει από τον πρώτο χαρακτήρα, επιλέγει έναν χαρακτήρα τη φορά, κάτι υπολογίζει με αυτόν, μετά πηγαίνει στον επόμενο και συνεχίζει μέχρι το τέλος της συμβολοσειράς. Αυτός ο τρόπος υπολογισμού ονομάζεται **πέρασμα** (**traversal**).

Ένας τρόπος για να γράψουμε πρόγραμμα για το πέρασμα μιας συμβολοσειράς είναι με χρήση της εντολής `while`. Ας δούμε ένα παράδειγμα:

Κώδικας 7.1

```
fruit = 'banana'
index = 0

while index < len(fruit):
    letter = fruit[index]
    print(letter)
    index = index + 1
```


```
>>> ================================ RESTART ===
>>>
b
a
n
a
n
a
```

Στον κώδικα 7.1 ο βρόγχος `while` διατρέχει τη συμβολοσειρά `fruit` και δείχνει έναν χαρακτήρα σε διαφορετική γραμμή κάθε φορά. Ο τελευταίος χαρακτήρας που τυπώνεται έχει δείκτη `len(fruit)-1`, και αυτός είναι ο τελευταίος χαρακτήρας της συμβολοσειράς.

Εναλλακτικά, για το πέρασμα μιας συμβολοσειράς μπορούμε να χρησιμοποιήσουμε την εντολή `for`. Η προσέγγιση αυτή μας προσφέρει μικρότερο και πιο «κομψό» κώδικα. Για παράδειγμα:

Κώδικας 7.2

```
fruit = 'banana'

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


```
>>> ================================ RESTART ===
>>>
b
a
n
a
n
a
```

Στον κώδικα 7.2 σε κάθε πέρασμα του βρόγχου `for` ο επόμενος χαρακτήρας της συμβολοσειράς `fruit` εκχωρείται στη μεταβλητή `char`. Ο βρόγχος συνεχίζει μέχρι να εξαντληθούν οι χαρακτήρες.

7.3 Χαρακτήρες

Κάθε χαρακτήρας αντιστοιχεί σε έναν κωδικό χαρακτήρα, τον οποίο μπορούμε να τυπώσουμε με χρήση της συνάρτησης `ord`:

```
>>> ord('a')
97
>>> ord('b')
98
>>> ord('z')
122
>>> ord('α')
945
>>> ord('β')
946
>>> ord('ω')
969
>>>
```

Με δεδομένο έναν κωδικό χαρακτήρα μπορούμε να τυπώσουμε τον αντίστοιχο χαρακτήρα με χρήση της συνάρτησης `chr`:

```
>>> chr(97)
'a'
>>> chr(946)
'β'
>>>
```

Το διασημότερο σχήμα κωδικοποίησης χαρακτήρων είναι ο κώδικας `ASCII` (American Standard Code for Information Interchange) με τον οποίο μπορούν να αναπαρασταθούν 256 διαφορετικοί χαρακτήρες. Με τους 256 χαρακτήρες μπορούμε να αναπαραστήσουμε όλα τα γράμματα του Ελληνικού και του Λατινικού αλφάβητου. Επειδή όμως θέλουμε να έχουμε στον υπολογιστή μας γράμματα από όλα τα γνωστά αλφάβητα του κόσμου, χρησιμοποιούμε το σχήμα κωδικοποίησης `Unicode` με το οποίο μπορούμε να αναπαραστήσουμε 65.536 διαφορετικούς χαρακτήρες.

7.4 Χαρακτήρες διαφυγής

Υπάρχουν χαρακτήρες οι οποίοι δεν έχουν οπτική απεικόνιση με κάποιο σύμβολο, όπως για παράδειγμα ο χαρακτήρας νέας γραμμής. Για να χειριστεί τέτοιους χαρακτήρες σε συμβολοσειρές, η Python διαθέτει τους **χαρακτήρες διαφυγής (escape characters)**:

\\	backslash
\'	μονά εισαγωγικά
\"	διπλά εισαγωγικά
\n	νέα γραμμή
\t	tab

Ας δούμε μερικά παραδείγματα χρήσης των χαρακτήρων διαφυγής:

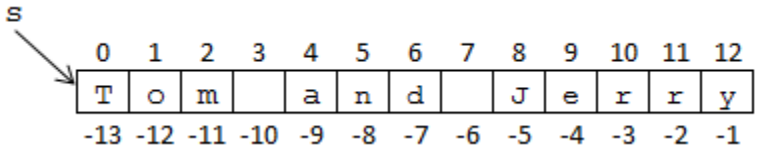
```
>>> print('\\Hello\\')
\Hello\
>>> print('It\'s')
It's
>>> print('one\ntwo\nthree')
one
two
three
>>> print('one\ttwo')
one     two
>>>
```

Κάθε χαρακτήρας διαφυγής είναι ένας μονός χαρακτήρας, το σύμβολο \ δεν μετράει ως χαρακτήρας, απλά δηλώνει ότι πρόκειται για ειδικό χαρακτήρα:

```
>>> len('It\'s')
4
>>>
```

7.5 Φέτα συμβολοσειράς

Για να λάβουμε μια υπο-συμβολοσειρά μιας συμβολοσειράς παίρνουμε, όπως λέμε, μία **φέτα** (**slice**) της. Για να γίνει αυτό, χρησιμοποιούμε τον τελεστή `[n:m]`, ο οποίος επιστρέφει το τμήμα της συμβολοσειράς από τον `n`-οστό χαρακτήρα μέχρι τον `m`-στό χαρακτήρα, περιλαμβάνοντας τον πρώτο αλλά αποκλείοντας τον τελευταίο. Η χρήση των δεικτών στον παραπάνω τελεστή είναι όμοια με αυτή της μεμονωμένης προσπέλασης χαρακτήρων σε μια συμβολοσειρά. Ας δούμε, για παράδειγμα, τη συμβολοσειρά `'Tom and Jerry'`:




```
>>> s = 'Tom and Jerry'
>>> s[0:3]
'Tom'
>>> s[4:7]
'and'
>>> s[8:13]
'Jerry'
>>> s[0:1]
'T'
>>>
```

Αν αφήσουμε κενό τον πρώτο δείκτη μιας φέτας, η Python υποθέτει ότι εννοούμε τον δείκτη `0` και, αν αφήσουμε κενό τον δεύτερο δείκτη, η Python υποθέτει ότι εννοούμε όλο το υπόλοιπο τμήμα μέχρι το τέλος της συμβολοσειράς. Για παράδειγμα (συνέχεια του προηγούμενου):

```
>>> s[:3]
'Tom'
>>> s[4:]
'and Jerry'
>>>
```

Μπορούμε να χρησιμοποιήσουμε και αρνητικούς δείκτες σε μία φέτα. Για παράδειγμα (συνέχεια του προηγούμενου):

```
>>> s[-13:-10]
'Tom'
>>>
```

Ποιες υπο συμβολοσειρές θα επιστρέψουν στο παραπάνω παράδειγμα οι φέτες `s[:]` *και* `s[4:4]`

7.6 Σύγκριση συμβολοσειρών

Μπορούμε να χρησιμοποιήσουμε τους τελεστές σύγκρισης, για να συγκρίνουμε συμβολοσειρές. Η σύγκριση βασίζεται στη διάταξη των χαρακτήρων στο σχήμα κωδικοποίησης του υπολογιστή μας. Για παράδειγμα:

```
>>> 'a' < 'b'
True
>>> 'aa' < 'ab'
True
>>> 'aa' == 'aa'
True
>>> 'apple' < 'banana'
True
>>> 'Apple' > 'apple'
False
>>> 'Αντώνης' < 'Εμμανουέλα'
True
```

Τα κεφαλαία προηγούνται των πεζών γραμμάτων και τα Λατινικά προηγούνται των Ελληνικών γραμμάτων. Μια σημαντική χρήση της σύγκρισης συμβολοσειρών είναι η αλφαβητική ταξινόμηση ονομάτων.

7.7 Οι συμβολοσειρές είναι αμετάβλητες

Αν δοκιμάσουμε να χρησιμοποιήσουμε τον τελεστή `[]` στην αριστερή πλευρά μιας εκχώρησης και με στόχο να αλλάξουμε έναν χαρακτήρα σε μια συμβολοσειρά, η Python δεν θα μας αφήσει και θα εμφανιστεί μήνυμα λάθους. Για παράδειγμα:

```
>>> greeting = 'Hello, World!'
>>> greeting[0] = 'J'
Traceback (most recent call last):
  File "<pyshell#43>", line 1, in <module>
    greeting[0] = 'J'
TypeError: 'str' object does not support item assignment
>>>
```

Αυτό συμβαίνει γιατί οι συμβολοσειρές είναι **αμετάβλητες** (**immutable**). Το μόνο που μπορούμε να κάνουμε είναι να δημιουργήσουμε μια νέα συμβολοσειρά που θα είναι παραλλαγή της αρχικής. Για παράδειγμα:

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

Στο παραπάνω παράδειγμα βλέπουμε ότι η δημιουργία της νέας συμβολοσειράς μπορεί να γίνει σχετικά εύκολα με συνένωση και χρήση φέτας συμβολοσειράς.

7.8 Ο τελεστής `in`

Ο τελεστής `in` είναι ένας λογικός τελεστής που εφαρμόζεται σε δύο συμβολοσειρές και ελέγχει αν η μία εμφανίζεται μέσα στην άλλη. Για παράδειγμα:

```
>>> 'a' in 'banana'
True
>>> 'an' in 'banana'
True
>>> 'c' in 'banana'
False
>>>
```

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

7.9 Μέθοδοι για συμβολοσειρές

Οι εργασίες που καλούμαστε να κάνουμε σε συμβολοσειρές είναι αρκετές, όπως εργασίες ελέγχου (π.χ. αν μία συμβολοσειρά περιέχει μόνο γράμματα ή ψηφία), εργασίες αναζήτησης (π.χ. αναζήτηση χαρακτήρα σε μια συμβολοσειρά), εργασίες μορφοποίησης, κ.λπ. Η Python μάς διευκολύνει με αυτές τις εργασίες προσφέροντάς μας ένα σύνολο από χρήσιμες μεθόδους. Οι μέθοδοι είναι όμοιες με τις συναρτήσεις (παίρνουν ορίσματα και επιστρέφουν κάποια τιμή), αλλά διαφέρουν στον τρόπο με τον οποίο συντάσσονται (dot notation, χρήση του συμβόλου της τελείας). Θα μιλήσουμε αναλυτικά για τις μεθόδους σε επόμενο κεφάλαιο.

Στον παρακάτω πίνακα μπορείτε να δείτε μια λίστα με μερικές μεθόδους ελέγχου συμβολοσειράς. Οι μέθοδοι αυτοί επιστρέφουν `True` ή `False` ανάλογα με το αποτέλεσμα του ελέγχου.

Μέθοδοι ελέγχου μίας συμβολοσειράς s	
`s.isalpha()`	Η s περιέχει μόνο γράμματα.
`s.isdigit()`	Η s περιέχει μόνο ψηφία.
`s.islower()`	Η s περιέχει μόνο πεζά γράμματα.
`s.isupper()`	Η s περιέχει μόνο κεφαλαία γράμματα
`s.isprintable()`	Η s περιέχει μόνο εκτυπώσιμους χαρακτήρες.
`s.isspace()`	Η s περιέχει μόνο whitespaces (κενά, tab, αλλαγές γραμμής).
`s.startswith(t)`	Η s ξεκινάει με την t.
`s.endswith(t)`	Η s τελειώνει με την t.

Για παράδειγμα:

```
>>> s = 'hello'
>>> s.isalpha()
True
>>> s = '1209'
>>> s.isdigit()
True
>>> s = 'python'
>>> s.islower()
True
>>> s = 'PYTHON'
>>> s.isupper()
True
>>> s.isprintable()
True
>>> s = '\t \n'
>>> s.isspace()
True
>>> s = 'Tom and Jerry'
>>> s.startswith('Tom')
True
>>> s.endswith('Jerry')
True
>>>
```

Για να αναζητήσουμε υπο-συμβολοσειρές μέσα σε μία συμβολοσειρά, μπορούμε να χρησιμοποιήσουμε τις παρακάτω μεθόδους:

Μέθοδοι αναζήτησης μέσα σε μία συμβολοσειρά `s`	
`s.find(t)`	Επιστρέφει τον πρώτο δείκτη, στον οποίο ξεκινάει η `t` μέσα στην `s`, αλλιώς επιστρέφει `-1`.
`s.rfind(t)`	Όμοια με τη `find`, αλλά αναζητάει από τα δεξιά προς τα αριστερά.

Για παράδειγμα:

```
>>> s = 'apple'
>>> s.find('le')
3
>>> s.find('w')
-1
>>> s.find('p')
1
>>> s.rfind('p')
2
>>>
```

Με τη μέθοδο `count` μπορούμε να απαριθμήσουμε την εμφάνιση μιας συμβολοσειράς μέσα σε μία άλλη. Για παράδειγμα:

```
>>> fruit = 'banana'
>>> fruit.count('a')
3
>>> fruit.count('na')
2
>>>
```

Για κάνουμε μετατροπές μεταξύ κεφαλαίων και πεζών γραμμάτων, μπορούμε να χρησιμοποιήσουμε τις παρακάτω μεθόδους:

Μέθοδοι μετατροπών μεταξύ κεφαλαίων και πεζών γραμμάτων σε μία συμβολοσειρά s	
`s.capitalize()`	Το `s[0]` μετατρέπεται σε κεφαλαίο.
`s.lower()`	Όλα τα γράμματα της `s` μετατρέπονται σε πεζά.
`s.upper()`	Όλα τα γράμματα της `s` μετατρέπονται σε κεφαλαία.
`s.swapcase()`	Τα πεζά μετατρέπονται σε κεφαλαία και τα κεφαλαία σε πεζά.

Για παράδειγμα:

```
>>> s = 'hello'
>>> s.capitalize()
'Hello'
>>> s.lower()
'hello'
>>> s.upper()
'HELLO'
>>> s = 'hElLo'
>>> s.swapcase()
'HeLlO'
>>>
```

Για να αφαιρέσουμε ανεπιθύμητους χαρακτήρες, κυρίως whitespaces, από την αρχή ή το τέλος μιας συμβολοσειράς, χρησιμοποιούμε τις παρακάτω μεθόδους:

Μέθοδοι αφαίρεσης ανεπιθύμητων χαρακτήρων σε μία συμβολοσειρά s	
`s.strip(ch)`	Αφαιρούνται όλοι οι χαρακτήρες `ch` που εμφανίζονται στην αρχή ή το τέλος της `s`.
`s.lstrip(ch)`	Αφαιρούνται όλοι οι χαρακτήρες `ch` που εμφανίζονται στην αρχή (αριστερά) της `s`.
`s.rtrip(ch)`	Αφαιρούνται όλοι οι χαρακτήρες `ch` που εμφανίζονται στο τέλος (δεξιά) της `s`.

Για παράδειγμα:

```
>>> fruit = '--banana--'
>>> fruit.strip('-')
'banana'
>>> fruit.lstrip('-')
'banana--'
>>> fruit.rstrip('-')
'--banana'
>>>
```

Αν παραλείψουμε να δώσουμε όρισμα στις παραπάνω μεθόδους, τότε εξ ορισμού αφαιρούνται οι χαρακτήρες whitespace. Για παράδειγμα:

```
>>> fruit = ' banana  '
>>> fruit.strip()
'banana'
>>>
```

Για να μορφοποιήσουμε μια συμβολοσειρά, μπορούμε να χρησιμοποιήσουμε τη μέθοδο `format`. Για παράδειγμα:

```
>>> name = 'Αντώνης'
>>> age = 13
>>> print('Ο {0} είναι {1} ετών.'.format(name, age))
Ο Αντώνης είναι 13 ετών.
>>>
```

Η Python διαθέτει πάρα πολλές ακόμη έτοιμες προς χρήση μεθόδους για εργασίες σε συμβολοσειρές. Ζητήστε αναλυτική βοήθεια από τον διερμηνευτή πληκτρολογώντας την εντολή `help(str)`. Επίσης, διαθέτει και το άρθρωμα `string`. Για παράδειγμα:

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

Κεφάλαιο 8

Δομές Δεδομένων

Μία **δομή δεδομένων** (**data structure**) είναι σύνολο δεδομένων (τιμών) μαζί με ένα σύνολο επιτρεπτών λειτουργιών (πράξεων) επί αυτών. Οι πιο βασικές λειτουργίες είναι οι ακόλουθες: προσπέλαση, εισαγωγή, διαγραφή, αναζήτηση, ταξινόμηση, συγχώνευση και διαχωρισμός. Τα δεδομένα που χειρίζονται τα διάφορα προγράμματα που αναπτύσσουμε θα πρέπει να είναι οργανωμένα σε δομές δεδομένων. Ουσιαστικά ένα πρόγραμμα είναι ένα σύνολο αλγορίθμων και δομών δεδομένων.

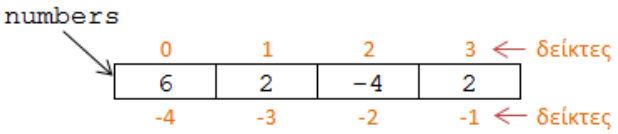
Η Python διαθέτει αρκετές δομές δεδομένων και παρέχει επαρκή υποστήριξη αυτών. Οι συμβολοσειρές που είδαμε στο προηγούμενο κεφάλαιο μπορούν να θεωρηθούν δομές δεδομένων που αποθηκεύουν αυστηρά μόνο χαρακτήρες. Άλλες χρήσιμες δομές δεδομένων που μπορούν να αποθηκεύουν όλων των ειδών δεδομένα είναι οι **λίστες**, οι **πλειάδες** και τα **λεξικά**.

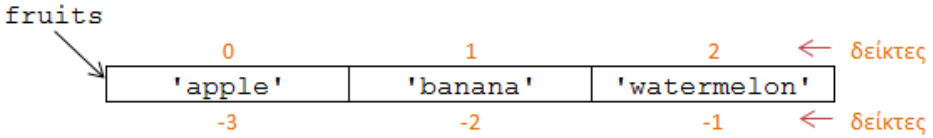
8.1 Λίστες

Μία **λίστα** (**list**) είναι μια διατεταγμένη συλλογή τιμών, οι οποίες αντιστοιχίζονται σε δείκτες. Οι τιμές που είναι μέλη μιας λίστας ονομάζονται **στοιχεία** (**elements**). Τα στοιχεία μιας λίστας δεν χρειάζεται να είναι ίδιου τύπου και ένα στοιχείο σε μία λίστα μπορεί να υπάρχει περισσότερες από μία φορές. Μία λίστα μέσα σε μία άλλη λίστα ονομάζεται **εμφωλευμένη λίστα** (**nested list**). Επιπρόσθετα, τόσο οι λίστες όσο και οι συμβολοσειρές, που συμπεριφέρονται ως διατεταγμένες συλλογές τιμών, ονομάζονται **ακολουθίες** (**sequences**).

Τα στοιχεία μιας λίστας διαχωρίζονται με κόμμα και περικλείονται σε τετράγωνες αγκύλες (`[` και `]`). Μία λίστα που δεν περιέχει στοιχεία ονομάζεται **άδεια λίστα** και συμβολίζεται με `[]`. Για παράδειγμα:

```
>>> numbers = [6, 2, -4, 2]
>>> fruits = ['apple', 'banana', 'watermelon']
>>> lst = [1, 6.5, 'cherry', [5, 7, 9]]
>>> empty_list = []
>>> print(numbers, '\n', fruits, '\n', lst, '\n', empty_list)
[6, 2, -4, 2]
 ['apple', 'banana', 'watermelon']
 [1, 6.5, 'cherry', [5, 7, 9]]
 []
>>>
```





Οι λίστες που περιέχουν συνεχόμενους ακέραιους αριθμούς είναι συνηθισμένες, και έτσι η Python μάς προσφέρει έναν εύκολο τρόπο, για να τις δημιουργούμε:

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

Η συνάρτηση `range` δέχεται δύο ακεραίους ως ορίσματα και επιστρέφει την ακολουθία των ακέραιων αριθμών στο διάστημα από το πρώτο έως το δεύτερό της όρισμα, συμπεριλαμβάνοντας το πρώτο αλλά αποκλείοντας το

δεύτερο. Με ένα μόνο όρισμα η `range` επιστρέφει μία ακολουθία που ξεκινάει από το μηδέν. Αν υπάρχει και τρίτο όρισμα, τότε αυτό ορίζει τον «βηματισμό» της ακολουθίας τιμών. Αν παραλειφθεί το τρίτο όρισμα, τότε εξ ορισμού το βήμα είναι `1`. Η συνάρτηση `list` αρχικοποιεί μία νέα λίστα που περιέχει τους ακέραιους αριθμούς της ακολουθίας που δίνεται ως όρισμα σε αυτήν. Χωρίς όρισμα η `list` επιστρέφει άδεια λίστα. Για παράδειγμα:

```
>>> numbers1 = list(range(10))
>>> print(numbers1)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> numbers2 = list(range(1,10,2))
>>> print(numbers2)
[1, 3, 5, 7, 9]
>>>
```

Προσπέλαση στοιχείων μιας λίστας

Για να προσπελάσουμε τα στοιχεία μιας λίστας, χρησιμοποιούμε παρόμοια σύνταξη με αυτή της προσπέλασης των χαρακτήρων μιας συμβολοσειράς. Κάθε ακέραια έκφραση μπορεί να χρησιμοποιηθεί ως δείκτης. Αν προσπαθήσουμε να προσπελάσουμε στοιχείο που δεν υπάρχει, τότε θα εμφανιστεί μήνυμα λάθους. Για παράδειγμα:

```
>>> numbers = [2, 5, 9, 15]
>>> print(numbers[0])
2
>>> print(numbers[1])
5
>>> print(numbers[3])
15
>>> print(numbers[3-1])
9
>>> print(numbers[5])
Traceback (most recent call last):
  File "<pyshell#20>", line 1, in <module>
    print(numbers[5])
IndexError: list index out of range
>>>
```

Μπορούμε να χρησιμοποιήσουμε και δείκτες με αρνητική τιμή. Για παράδειγμα:

```
>>> numbers = [2, 5, 9, 15]
>>> print(numbers[-1])
15
>>> print(numbers[-2])
9
>>> print(numbers[-3])
5
>>> print(numbers[-4])
2
>>>
```

Συχνά μία μεταβλητή βρόγχου χρησιμοποιείται για την προσπέλαση όλων των στοιχείων μιας λίστας (πέρασμα λίστας). Για παράδειγμα:

Κώδικας 8.1

```
fruits = ['apple', 'cherry', 'banana', 'mango']
i = 0
while i < len(fruits):
    print(fruits[i])
    i = i + 1
```


```
>>> ================================ RESTART ===
>>>
apple
cherry
banana
mango
```

Η συνάρτηση `len` επιστρέφει το μήκος μιας λίστας (το πλήθος των στοιχείων της). Αν μία λίστα περιέχει άλλη λίστα ως στοιχείο, τότε η εμφωλευμένη λίστα μετράει ως απλό στοιχείο. Για παράδειγμα:

```
>>> mylist = [2, 5, [8, 12, 20]]
>>> print(len(mylist))
3
>>>
```

Εναλλακτικά, για το πέρασμα μιας λίστας, μπορούμε να χρησιμοποιήσουμε βρόγχο `for`. Για παράδειγμα:

Κώδικας 8.2

```
fruits = ['apple', 'cherry', 'banana', 'mango']
for fruit in fruits:
    print(fruit)
```


```
>>> ================================ RESTART ===
>>>
apple
cherry
banana
mango
```

Με τον λογικό τελεστή `in` μπορούμε να ελέγξουμε αν μια τιμή ανήκει σε μια λίστα και δουλεύει όπως και στις συμβολοσειρές. Μπορούμε επίσης να χρησιμοποιήσουμε την έκφραση `not in`. Για παράδειγμα:

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

Κάθε έκφραση ακολουθίας τιμών μπορεί να χρησιμοποιηθεί στην επικεφαλίδα ενός βρόγχου `for`:

Κώδικας 8.3

```
for fruit in ['apple', 'banana', 'mango']:
    print('I like to eat ' + fruit + 's.')
```


```
>>> ================================ RESTART ===
>>>
I like to eat apples.
I like to eat bananas.
I like to eat mangos.
```

Πράξεις σε λίστες

Με τον τελεστή + μπορούμε να συνενώσουμε λίστες (αλληλουχία) και με τον τελεστή * να επαναλάβουμε μια λίστα. Για παράδειγμα:

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

Φέτες από λίστες

Ο τελεστής φέτας δουλεύει και με λίστες. Για παράδειγμα:

```
>>> 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']
>>>
```

Οι λίστες είναι μετατρέψιμες (

Μπορούμε εύκολα να αλλάξουμε στοιχεία μιας λίστας. Για παράδειγμα:

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

Με τον τελεστή φέτας μπορούμε να αλλάξουμε μεμιάς αρκετά στοιχεία:

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

Μπορούμε να αφαιρέσουμε στοιχεία από μια λίστα εκχωρώντας τους την άδεια λίστα:

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

Επίσης, μπορούμε να προσθέσουμε στοιχεία σε μια λίστα στριμώχνοντάς τα σε μια άδεια φέτα:

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

Διαγραφή στοιχείων μιας λίστας

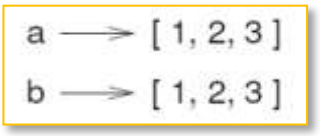
Ο ευκολότερος τρόπος για να διαγράψουμε στοιχεία από μια λίστα είναι να χρησιμοποιήσουμε τον τελεστή `del`. Ο `del` χειρίζεται και αρνητικούς δείκτες και φέτες. Για παράδειγμα:

```
>>> mylist = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del mylist[5]
>>> print(mylist)
['a', 'b', 'c', 'd', 'e']
>>> del mylist[-1]
>>> print(mylist)
['a', 'b', 'c', 'd']
>>> del mylist[1:4]
>>> print(mylist)
['a']
>>>
```

Αντικείμενα και αναφορές, ψευδώνυμα

Μπορούμε να πούμε με απλά λόγια ότι όλες οι τιμές στην Python αποθηκεύονται στη μνήμη του υπολογιστή με τη μορφή **αντικειμένων**. Γενικά όλα στην Python είναι αντικείμενα και θα μιλήσουμε γι' αυτό σε επόμενο κεφάλαιο. Ας δούμε ένα παράδειγμα:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
>>>
```



Στο παραπάνω παράδειγμα οι μεταβλητές `a` και `b` έχουν την ίδια τιμή, αλλά αναφέρονται σε διαφορετικά αντικείμενα. Με άλλα λόγια είναι ισότιμες αλλά όχι ταυτόσημες. Ο τελεστής `is` μάς λέει αν δύο μεταβλητές αναφέρονται στο ίδιο αντικείμενο.

Εφόσον οι μεταβλητές αναφέρονται σε αντικείμενα, αν εκχωρήσουμε μια μεταβλητή σε μία άλλη, και οι δύο θα αναφέρονται στο ίδιο αντικείμενο:

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

Εάν μία λίστα έχει δύο ονόματα, λέμε ότι έχει **ψευδώνυμα**. Αλλαγές που γίνονται στο ένα ψευδώνυμο επηρεάζουν το άλλο:

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

Λίστες κλώνοι

Ο ευκολότερος τρόπος για να κλωνοποιήσουμε μια λίστα είναι με χρήση του τελεστή φέτας. Η κλωνοποίηση δημιουργεί καινούρια λίστα, στην οποία μπορούμε να κάνουμε αλλαγές χωρίς να επηρεάζεται η αρχική. Για παράδειγμα:

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

Η λίστα ως όρισμα

Όταν περνάμε μια λίστα ως όρισμα σε μια συνάρτηση, στην πραγματικότητα περνάμε μια αναφορά σε αυτή και όχι ένα αντίγραφό της. Για παράδειγμα:

Κώδικας 8.4

```
def deleteHead(mylist):
    del mylist[0]

numbers = [1, 2, 3]
deleteHead(numbers)
print(numbers)
```


```
>>> ================================ RESTART ===
>>>
[2, 3]
```

Εμφωλευμένες λίστες, πίνακες

Όπως έχουμε δει, μια εμφωλευμένη λίστα είναι μια λίστα που εμφανίζεται ως στοιχείο άλλης λίστας. Για παράδειγμα:

```
>>> mylist = ['hello', 3, [10, 20]]
>>> inlist = mylist[2]
>>> print(inlist)
[10, 20]
>>> print(mylist[2][1])
20
>>>
```

Οι τελεστές `[]` υπολογίζονται από αριστερά προς τα δεξιά.

Με εμφωλευμένες λίστες μπορούμε να αναπαραστήσουμε πίνακες. Για παράδειγμα, ο πίνακας $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$

μπορεί να αναπαρασταθεί ως εξής:

```
>>> matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> print('\n', matrix[0], '\n', matrix[1], '\n', matrix[2])

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

Τι θα τυπώσει στο παραπάνω παράδειγμα η εντολή `print(matrix[2][1])`

Λίστες και συμβολοσειρές

Με τη μέθοδο `split` μπορούμε να δημιουργήσουμε λίστα από συμβολοσειρά. Οι χαρακτήρες whitespace θεωρούνται όρια λέξεων. Προαιρετικά μπορεί να χρησιμοποιηθεί όρισμα που θα ορίζει τους χαρακτήρες που οριοθετούν λέξεις. Για παράδειγμα:

```
>>> phrase = 'What a wonderful world'
>>> phrasewords = phrase.split()
>>> print(phrasewords)
['What', 'a', 'wonderful', 'world']
>>> phrasewords1 = phrase.split('wo')
>>> print(phrasewords1)
['What a ', 'nderful ', 'rld']
>>>
```

Η συνάρτηση `list` «σπάει» μια συμβολοσειρά σε χαρακτήρες δημιουργώντας μία λίστα:

```
>>> s = 'spam'
>>> chlist = list(s)
>>> print(chlist)
['s', 'p', 'a', 'm']
>>>
```

Η μέθοδος `join` είναι η αντίστροφη της `split`. Δέχεται μία λίστα από συμβολοσειρές και συνενώνει τα στοιχεία της. Πρόκειται για μέθοδο για συμβολοσειρές, την επικαλούμαστε σε έναν οριοθέτη (delimiter) και περνάμε τη λίστα ως όρισμα. Για παράδειγμα:

```
>>> phrasewords = ['What', 'a', 'wonderful', 'world']
>>> delimiter1 = ' '
>>> phrase1 = delimiter1.join(phrasewords)
>>> print(phrase1)
What a wonderful world
>>> delimiter2 = '-'
>>> phrase2 = delimiter2.join(phrasewords)
>>> print(phrase2)
What-a-wonderful-world
>>>
```

Μέθοδοι για λίστες

Η Python διαθέτει έτοιμες μεθόδους και για τις λίστες. Ας δούμε μερικές από αυτές (για περισσότερες πληροφορίες πληκτρολογήστε `help(list)`):

Μέθοδοι για λίστες	
`l.append(x)`	Προσθέτει το στοιχείο `x` στο τέλος της λίστας `l`.
`l.sort()`	Ταξινομεί τη λίστα `l` κατά αύξουσα σειρά.
`l.extend(lst)`	Προσθέτει κάθε στοιχείο της `lst` στο τέλος της `l`.
`l.pop(i)`	Αφαιρεί και επιστρέφει το στοιχείο που βρίσκεται στο δείκτη `i`.
`l.remove(x)`	Αφαιρεί την πρώτη εμφάνιση από αριστερά του στοιχείου `x`.

Για παράδειγμα:

```
>>> mylist = ['a', 'b', 'c']
>>> mylist.append('d')
>>> mylist.append('e')
>>> print(mylist)
['a', 'b', 'c', 'd', 'e']
>>>
>>> mylist1 = ['c', 'a', 'e', 'b', 'd']
>>> mylist1.sort()
>>> print(mylist1)
['a', 'b', 'c', 'd', 'e']
>>>
```

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

8.2 Πλειάδες

Μία **πλειάδα** (**tuple**) είναι μια διατεταγμένη ακολουθία τιμών, οι οποίες αντιστοιχίζονται σε δείκτες. Οι τιμές που είναι μέλη μιας πλειάδας ονομάζονται **στοιχεία** (**elements**) και μπορεί να είναι οποιουδήποτε τύπου (αριθμοί, συμβολοσειρές, λίστες, πλειάδες). Οι πλειάδες μοιάζουν με τις λίστες στη χρήση δεικτών, στον τρόπο με τον οποίο διατρέχονται και στη χρήση του τελεστή φέτας. Όμως οι πλειάδες, όπως και οι συμβολοσειρές, είναι **αμετάβλητες**. Οι πλειάδες αξιοποιούνται συνήθως στις περιπτώσεις όπου πρόκειται να χρησιμοποιηθεί μια ακολουθία τιμών (πλειάδα) που δεν πρόκειται να αλλάξει.

Συντακτικά, μια πλειάδα είναι μια λίστα τιμών που χωρίζονται με κόμμα. Παρόλο που δεν είναι αναγκαίο, είναι σύνηθες να περικλείουμε τις πλειάδες με παρενθέσεις. Για παράδειγμα:

```
>>> t1 = 'a', 'b', 'c', 'd'
>>> print(t1)
('a', 'b', 'c', 'd')
>>> t2 = (1, 3, 5, 7, 9)
>>> print(t2)
(1, 3, 5, 7, 9)
>>>
```

Για να δημιουργήσουμε μια άδεια πλειάδα, χρησιμοποιούμε άδειες παρενθέσεις. Για να δημιουργήσουμε πλειάδα με ένα μόνο στοιχείο, πρέπει να προσθέσουμε ένα κόμμα. Χωρίς κόμμα η Python νομίζει ότι πρόκειται για συμβολοσειρά μέσα σε παρενθέσεις. Για παράδειγμα:

```
>>> t3 = ()
>>> print(t3)
()
>>> t4 = ('a',)
>>> print(t4)
('a',)
>>> t5 = ('a')
>>> print(t5)
a
>>> type(t5)
<class 'str'>
>>>
```

Οι πράξεις πάνω σε πλειάδες είναι παρόμοιες με τις πράξεις πάνω σε λίστες. Ο τελεστής `[]` επιλέγει ένα στοιχείο από μια πλειάδα. Ο τελεστής «φέτα» επιλέγει διάστημα τιμών, όπως ακριβώς και στις λίστες. Ο τελεστής `in` ελέγχει εάν μια τιμή ανήκει σε μια πλειάδα. Η συνάρτηση `len` επιστρέφει το μήκος μιας πλειάδας (τον αριθμό των στοιχείων που περιέχει). Για παράδειγμα:

```
>>> t1 = ('a', 'b', 'c')
>>> t2 = ('d', 'e')
>>> t = t1 + t2
>>> print(t)
('a', 'b', 'c', 'd', 'e')
>>> print(t[1])
b
>>> print(t[2:4])
('c', 'd')
>>> 'a' in t
True
>>> 'f' in t
False
>>> print(len(t))
5
>>> print(len((1, 2, 'a', [3, 4], (9, 12))))
5
>>>
```

Για να διατρέξουμε μια πλειάδα (πέρασμα πλειάδας), κάνουμε ακριβώς ό,τι κάναμε και στις λίστες. Για παράδειγμα:

Κώδικας 8.5

```
fruits = ('apple', 'orange', 'banana', 'mellon')
i = 0
while i < len(fruits):
    print(fruits[i])
    i = i + 1
```

Κώδικας 8.6

```
fruits = ('apple', 'orange', 'banana', 'mellon')
for fruit in fruits:
    print(fruit)
```


```
>>> ================================ RESTART ===
>>>
apple
orange
banana
mellon
```

Οι πλειάδες, όπως είπαμε, είναι αμετάβλητες. Αν προσπαθήσουμε να αλλάξουμε ένα από τα στοιχεία μιας πλειάδας, θα εμφανιστεί μήνυμα λάθους. Μπορούμε όμως να αντικαταστήσουμε μια πλειάδα με μία άλλη. Για παράδειγμα:

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

Η ανταλλαγή των τιμών δύο μεταβλητών γίνεται συμβατικά με χρήση μιας προσωρινής μεταβλητής. Στην Python η ανταλλαγή μπορεί να γίνει πιο «κομψά» με εκχώρηση με πλειάδες:

```
>>> a = 5
>>> b = 10
>>> a, b = b, a
>>> print(a)
10
>>> print(b)
5
>>>
```

Οι συναρτήσεις μπορούν να επιστρέφουν πλειάδες ως τιμές. Για παράδειγμα:

Κώδικας 8.7

```
def swap(x, y):
    return y, x

a = 5
b = 10
a, b = swap(a, b)
print('a =', a, 'b =', b)
```


```
>>> ================================ RESTART ===
>>>
a = 10 b = 5
```

Η συνάρτηση `tuple` δέχεται ως όρισμα μια συμβολοσειρά ή μια λίστα και επιστρέφει μια πλειάδα. Με κενό όρισμα επιστρέφει μια άδεια πλειάδα. Για παράδειγμα:

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

Η μέθοδος `count` επιστρέφει τον αριθμό των εμφανίσεων μιας τιμής σε μια πλειάδα. Η μέθοδος `index` επιστρέφει τον πρώτο δείκτη στον οποίο αντιστοιχεί μια τιμή. Αν δεν υπάρχει τιμή, εμφανίζεται μήνυμα λάθους. Για παράδειγμα:

```
>>> t = ('a', 'b', 'c', 'a')
>>> t.count('a')
2
>>> t.index('a')
0
>>>
```

8.3 Λεξικά

Το **λεξικό** (**dictionary**) είναι μια δομή δεδομένων για αποθήκευση ζευγαριών τιμών της μορφής *κλειδί:τιμή* (*key:value*). Πρόκειται για έναν σύνθετο τύπο. Κάθε κλειδί αντιστοιχίζεται σε μια τιμή και είναι μοναδικό σε ένα λεξικό. Μπορούμε να χρησιμοποιούμε μόνο αμετάβλητα αντικείμενα (όπως ακέραιους αριθμούς, συμβολοσειρές, πλειάδες) για κλειδιά ενός λεξικού, αλλά μπορούμε να έχουμε είτε αμετάβλητα ή μετατρέψιμα αντικείμενα για τις τιμές του. Τα λεξικά είναι μετατρέψιμα, και μπορούμε εύκολα να προσθέσουμε και να διαγράψουμε στοιχεία. Επίσης, ένα λεξικό αποτελεί μια μη διατεταγμένη συλλογή από ζεύγη κλειδιών-τιμών, τα οποία δεν ταξινομούνται με κανένα τρόπο (απροσδιόριστη σειρά). Δεν υπάρχει η έννοια της θέσης δείκτη και έτσι σε ένα λεξικό δεν μπορούμε να κάνουμε πέρασμα ή να χρησιμοποιήσουμε φέτες.

Ας φτιάξουμε για παράδειγμα ένα αγγλο-ισπανικό λεξικό. Μπορούμε να ξεκινήσουμε με ένα άδειο λεξικό που περικλείεται σε ζευγάρι αγκίστρων `{}` και στη συνέχεια να το εμπλουτίσουμε με στοιχεία:

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

Εναλλακτικά, μπορούμε με μία μόνο εντολή να δημιουργήσουμε ένα νέο λεξικό. Μπορούμε και να τυπώσουμε την τιμή που αντιστοιχεί σε ένα κλειδί. Για παράδειγμα:

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

Μπορούμε να δημιουργήσουμε ένα λεξικό και με χρήση της ενσωματωμένης συνάρτησης `dict`. Για παράδειγμα:

```
>>> d1 = dict()
>>> print(d1)
{}
>>> d2 = dict(red = 1, green = 2, blue = 3)
>>> print(d2)
{'green': 2, 'blue': 3, 'red': 1}
>>>
```

Ο τελεστής `del` αφαιρεί ένα ζευγάρι κλειδιού-τιμής από ένα λεξικό. Για παράδειγμα:

```
>>> inventory = {'apples':400, 'oranges':200, 'bananas':500}
>>> print(inventory)
{'oranges': 200, 'apples': 400, 'bananas': 500}
>>> del inventory['oranges']
>>> print(inventory)
{'apples': 400, 'bananas': 500}
>>>
```

Η συνάρτηση `len` χρησιμοποιείται και στα λεξικά και επιστρέφει το πλήθος των ζευγαριών κλειδιών-τιμών. Επίσης, ο τελεστής `in` ελέγχει το αν υπάρχει ένα κλειδί (και όχι μια τιμή) σε ένα λεξικό. Για παράδειγμα:

```
>>> inventory = {'apples':400, 'oranges':200, 'bananas':500}
>>> print(len(inventory))
3
>>> 'apples' in inventory
True
>>> 'mangos' in inventory
False
>>>
```

Μπορούμε να επικαλεστούμε πάνω σε ένα λεξικό τις μεθόδους `keys`, `values`, `items` για να επιστρέψουμε μια **εικόνα** (**view**) των κλειδιών, των τιμών και των ζευγαριών κλειδιών-τιμών αντίστοιχα. Για παράδειγμα:

```
>>> inventory = {'apples':400, 'oranges':200, 'bananas':500}
>>> keys_view = inventory.keys()
>>> print(keys_view)
dict_keys(['oranges', 'apples', 'bananas'])
>>> values_view = inventory.values()
>>> print(values_view)
dict_values([200, 400, 500])
>>> items_view = inventory.items()
>>> print(items_view)
dict_items([('oranges', 200), ('apples', 400), ('bananas', 500)])
>>>
```

Δημιουργήστε ένα δικό σας λεξικό και στη συνέχεια τυπώστε τα κλειδιά, τις τιμές και τα ζευγάρια κλειδιών τιμών.

Μπορούμε, αν θέλουμε, να χρησιμοποιήσουμε την `for`, για να τυπώσουμε τα κλειδιά, τις τιμές και τα ζευγάρια κλειδιών-τιμών ενός λεξικού. Για παράδειγμα:

Κώδικας 8.8

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

for key in keys_view:
    print(key)

for value in values_view:
    print(value)

for key,value in items_view:
    print(key,value)
```


```
>>> ================================ RESTART ===
>>>
three
two
one
tres
dos
uno
three tres
two dos
one uno
```

Με τη βοήθεια της συνάρτησης `list` μπορούμε να αποθηκεύσουμε τα κλειδιά, τις τιμές και τα ζευγάρια κλειδιών-τιμών ενός λεξικού σε αντίστοιχες λίστες (π.χ. `keys_list = list(eng2sp.keys())`). Για περισσότερες μεθόδους ζητήστε βοήθεια πληκτρολογώντας την εντολή `help(dict)`.

Κεφάλαιο 9

Αρχεία

Όσο εκτελείται ένα πρόγραμμα, τα δεδομένα που επεξεργάζεται είναι αποθηκευμένα προσωρινά στην κύρια μνήμη του υπολογιστή (μνήμη RAM). Όταν ολοκληρωθεί η εκτέλεση του προγράμματος ή κλείσει ο υπολογιστής, τα δεδομένα που βρίσκονται στη μνήμη RAM σβήνονται. Τα δεδομένα αποθηκεύονται μόνιμα και μπορούν να ξαναχρησιμοποιηθούν με κάθε νέο άνοιγμα του υπολογιστή μόνο αν τοποθετηθούν σε **αρχεία** σε κάποιο αποθηκευτικό μέσο (σκληρό δίσκο, flash memory, οπτικό δίσκο). Στην περίπτωση που έχουμε μεγάλο πλήθος αρχείων είναι χρήσιμο να τα οργανώνουμε σε **φακέλους** (**folders**). Κάθε αρχείο ταυτοποιείται με ένα μοναδικό όνομα μέσα σε έναν φάκελο ή έναν συνδυασμό ονόματος αρχείου και μονοπατιού φακέλων (pathname).

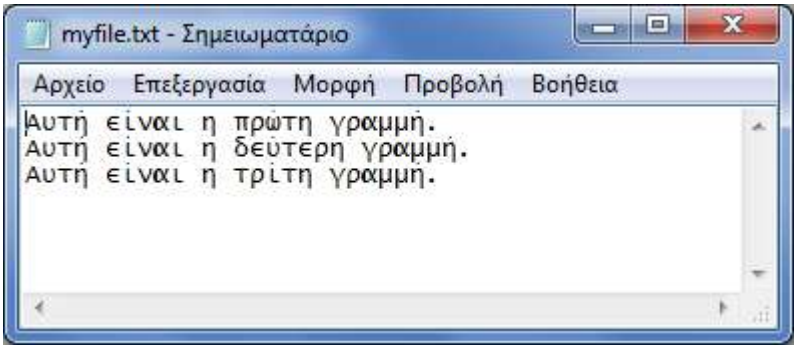
Τα αρχεία τα μεταχειριζόμαστε σαν τετράδια. Ένα τετράδιο πρώτα το ανοίγουμε και, όταν τελειώσουμε το γράψιμο, το κλείνουμε. Όταν είναι ανοικτό, μπορούμε να διαβάσουμε από αυτό ή να γράψουμε σε αυτό. Σε κάθε περίπτωση ξέρουμε πού ακριβώς είμαστε μέσα στο τετράδιο. Όλα αυτά εφαρμόζονται και στα αρχεία. Για να ανοίξουμε ένα αρχείο, προσδιορίζουμε ένα όνομα και δηλώνουμε εάν θέλουμε να το **διαβάσουμε** (**read**) ή να το **γράψουμε** (**write**). Όταν τελειώσουμε την εργασία μας με ένα αρχείο, θα πρέπει να το **κλείσουμε** (**close**). Στο κεφάλαιο αυτό θα διαχειριστούμε **αρχεία κειμένου** (**text files**), τα οποία δεν είναι τίποτε άλλο παρά μια σειρά από χαρακτήρες (συμβολοσειρές) που είναι αποθηκευμένοι σε ένα μόνιμο μέσο αποθήκευσης (σκληρό δίσκο, κ.λπ.). Τα αρχεία κειμένου μπορούν να διαβαστούν από έναν άνθρωπο, σε αντίθεση με τα **δυαδικά αρχεία** (**binary files**) για τα οποία απαιτούνται κατάλληλα προγράμματα για την ανάγνωση και την αξιοποίησή τους.

9.1 Γράψιμο σε ένα αρχείο

Για να γράψουμε δεδομένα σε ένα αρχείο κειμένου, θα πρέπει πρώτα να το ανοίξουμε, μετά να γράψουμε και στο τέλος να το κλείσουμε. Για παράδειγμα:

```
>>> f = open('C:\myfolder\myfile.txt', 'w')
>>> line1 = 'Αυτή είναι η πρώτη γραμμή.\n'
>>> f.write(line1)
27
>>> line2 = 'Αυτή είναι η δεύτερη γραμμή.\n'
>>> f.write(line2)
29
>>> line3 = 'Αυτή είναι η τρίτη γραμμή.\n'
>>> f.write(line3)
27
>>> f.close()
>>>
```

Το αρχείο κειμένου που δημιουργήθηκε είναι το παρακάτω:

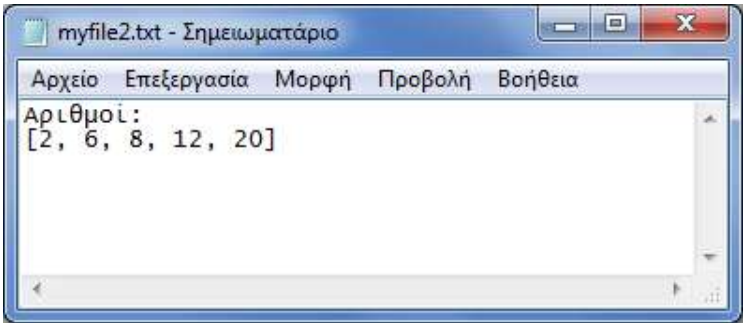


Η συνάρτηση `open` δέχεται δύο ορίσματα: το πρώτο είναι το όνομα του αρχείου (χωρίς αναφορά μονοπατιού εξ ορισμού ο φάκελος του αρχείου είναι ο φάκελος στον οποίο είναι εγκατεστημένη η Python) και το δεύτερο όρισμα είναι ο **τρόπος ανοίγματος (mode)**. Ο mode **'w'** (write, γράψιμο) σημαίνει ότι ανοίγουμε το αρχείο για γράψιμο. Αν το αρχείο που ανοίγουμε για γράψιμο υπάρχει ήδη, τότε τα δεδομένα που περιέχει σβήνονται όλα και αντικαθίστανται από τα νέα που γράφουμε. Αν το αρχείο δεν υπάρχει, τότε αυτό απλά δημιουργείται και είναι έτοιμο για να γράψουμε δεδομένα σε αυτό. Η εκτέλεση της εντολής `open`

δημιουργεί ένα αντικείμενο αρχείου πάνω στο οποίο μπορούμε να επικαλεστούμε μεθόδους. Με τη μέθοδο `write` μπορούμε να γράψουμε δεδομένα σε ένα αρχείο. Κατά την επίκληση της `write` εμφανίζεται το πλήθος των χαρακτήρων που γράφονται στο αρχείο. Επίσης, το όρισμα της `write` πρέπει να είναι συμβολοσειρά. Έτσι, αν θέλουμε να γράψουμε και άλλες τιμές (π.χ. αριθμούς), θα πρέπει πρώτα να τις μετατρέψουμε σε συμβολοσειρές με χρήση της συνάρτησης `str`. Για να αλλάξουμε γραμμή σε ένα αρχείο, χρησιμοποιούμε τον χαρακτήρα διαφυγής `\n`. Στο τέλος το κλείσιμο του αρχείου γίνεται με την επίκληση της μεθόδου `close` πάνω στο αντικείμενο του αρχείου.

Ας δούμε άλλο ένα παράδειγμα:

```
>>> f = open('C:\myfolder\myfile2.txt', 'w')
>>> f.write('Αριθμοί:\n')
9
>>> f.write(str([2, 6, 8, 12, 20]))
17
>>> f.close()
>>>
```

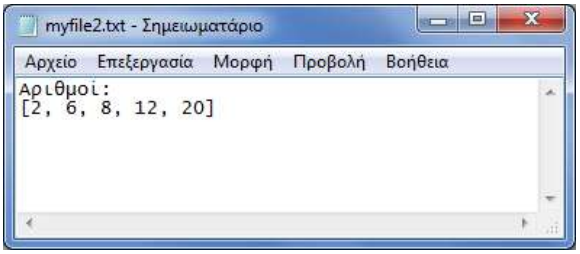


Είναι αυτονόητο ότι θα πρέπει να προϋπάρχει το μονοπάτι φακέλων που δίνεται ως όρισμα στη συνάρτηση `open` (π.χ. στα προηγούμενα παραδείγματα το `C:\myfolder\`), διαφορετικά θα εμφανιστεί μήνυμα λάθους. Επίσης, προτιμήσαμε να δώσουμε στα αρχεία μας την κατάληξη `.txt`, ώστε να ανοίγουν εύκολα με το `Σημειωματάριο` των `Windows`.

9.2 Ανάγνωση από ένα αρχείο

Για να ανοίξουμε ένα αποθηκευμένο αρχείο κειμένου για ανάγνωση, πρέπει να χρησιμοποιήσουμε τον mode **'r'** (read, ανάγνωση). Αν προσπαθήσουμε να ανοίξουμε για ανάγνωση ένα αρχείο που δεν υπάρχει, τότε θα εμφανιστεί μήνυμα λάθους. Το περιεχόμενο ενός αρχείου διαβάζεται ως συμβολοσειρά.

Ας δούμε ένα παράδειγμα (χρησιμοποιούμε το αρχείο του προηγούμενου παραδείγματος):

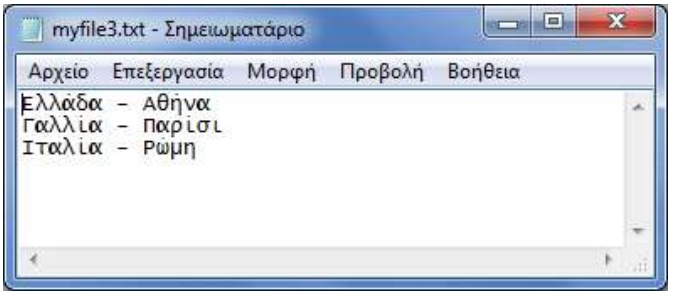



```
>>> f = open('C:\myfolder\myfile2.txt', 'r')
>>> text = f.read()
>>> type(text)
<class 'str'>
>>> print(text)
Αριθμοί:
[2, 6, 8, 12, 20]
>>> f.close()
>>>
```

Η μέθοδος `read` διαβάζει δεδομένα από ένα αρχείο. Χωρίς ορίσματα διαβάζει ως συμβολοσειρά όλο το περιεχόμενο του αρχείου. Μπορεί όμως να πάρει ως όρισμα έναν αριθμό που καθορίζει το πλήθος των χαρακτήρων που θα αναγνωστούν. Για παράδειγμα:

```
>>> f = open('C:\myfolder\myfile2.txt', 'r')
>>> text = f.read(7)
>>> print(text)
Αριθμοί
>>> ch = f.read(1)
>>> print(ch)
:
>>> f.close()
>>>
```

Στο παρακάτω παράδειγμα θα δημιουργήσουμε μια συνάρτηση για να τυπώνουμε γραμμή-γραμμή το περιεχόμενο ενός αρχείου με χρήση βρόγχου `for`:



Κώδικας 9.1

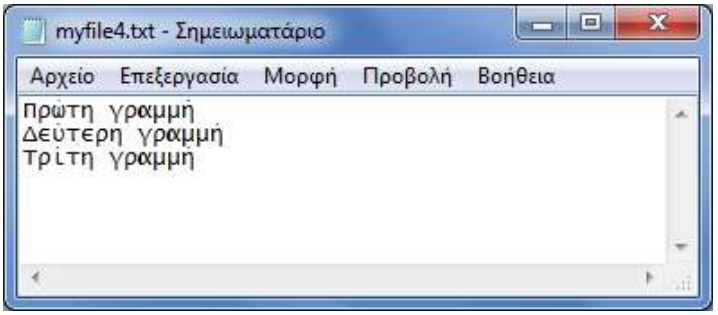
```
def printFile(fname):
    f = open(fname, 'r')
    for line in f:
        print(line, end = '')
    f.close()

printFile('C:\myfolder\myfile3.txt')
```

Run

```
>>> ================================ RESTART ===
>>>
Ελλάδα - Αθήνα
Γαλλία - Παρίσι
Ιταλία - Ρώμη
```

Η Python διαθέτει δύο ακόμα χρήσιμες μεθόδους για αρχεία. Η μέθοδος `readline` διαβάζει όλους τους χαρακτήρες μέχρι και τον επόμενο χαρακτήρα αλλαγής γραμμής. Η μέθοδος `readlines` επιστρέφει όλες τις εναπομένουσες τιμές ως λίστα από συμβολοσειρές. Για παράδειγμα:

```
>>> f = open('C:\myfolder\myfile4.txt', 'w')
>>> f.write('Πρώτη γραμμή\nΔεύτερη γραμμή\nΤρίτη γραμμή\n')
41
>>> f.close()
>>>
```

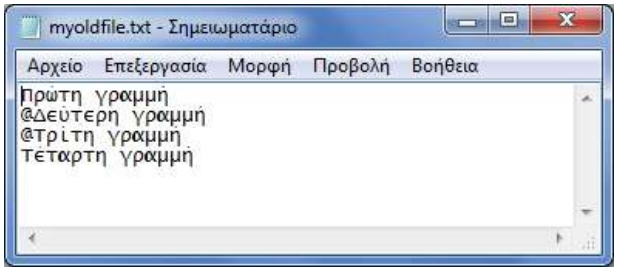


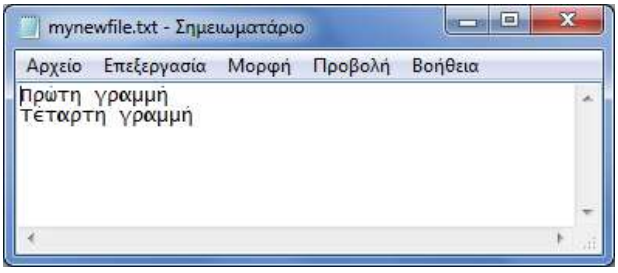

```
>>> f = open('C:\myfolder\myfile4.txt', 'r')
>>> line = f.readline()
>>> print(line)
Πρώτη γραμμή

>>> lines = f.readlines()
>>> print(lines)
['Δεύτερη γραμμή\n', 'Τρίτη γραμμή\n']
>>>
```

Ας δούμε άλλο ένα παράδειγμα προγράμματος που επεξεργάζεται γραμμές σε ένα αρχείο κειμένου. Πιο συγκεκριμένα, το πρόγραμμα αυτό δημιουργεί ένα αντίγραφο ενός αρχείου παραλείποντας γραμμές που αρχίζουν με τον χαρακτήρα @:

Κώδικας 9.2

```
def filterFile(oldFile, newFile):
    f1 = open(oldFile, 'r')
    f2 = open(newFile, 'w')

    while True:
        text = f1.readline()
        if text == '':
            break
        if text[0] == '@':
            continue
        f2.write(text)

    f1.close()
    f2.close()

filterFile('C:\myfolder\myoldfile.txt', 'C:\myfolder\mynewfile.txt')
```





Στην περίπτωση που θέλουμε να ανοίξουμε ένα αρχείο και να γράψουμε επιπλέον δεδομένα μετά το τέλος του, μπορούμε να χρησιμοποιήσουμε τον mode **'a'** (append, προσθήκη). Για παράδειγμα (χρησιμοποιούμε το αρχείο `mynewfile.txt` του προηγούμενου παραδείγματος):

```
>>> f = open('C:\myfolder\mynewfile.txt', 'a')
>>> f.write('Επιπλέον γραμμή')
15
>>> f.close()
>>> f = open('C:\myfolder\mynewfile.txt', 'r')
>>> text = f.read()
>>> print(text)
Πρώτη γραμμή
Τέταρτη γραμμή
Επιπλέον γραμμή
>>> f.close()
>>>
```

Όπως είδαμε και προηγουμένως, το όρισμα της μεθόδου `write` πρέπει να είναι συμβολοσειρά. Ο ευκολότερος τρόπος για να μετατρέψουμε μια τιμή που δεν είναι συμβολοσειρά σε συμβολοσειρά είναι με χρήση της συνάρτησης `str`. Εναλλακτικά, μπορούμε να χρησιμοποιήσουμε τον **τελεστή διάταξης** (**format**) %. Στους ακέραιους αριθμούς ο % είναι ο τελεστής modulus και επιστέφει το υπόλοιπο μιας διαίρεσης. Όταν όμως ο πρώτος τελεστέος είναι συμβολοσειρά, τότε ο τελεστής % λειτουργεί ως τελεστής διάταξης. Ο δεύτερος τελεστέος είναι πλειάδα εκφράσεων και το αποτέλεσμα είναι μια συμβολοσειρά. Για παράδειγμα:

```
>>> age = 12
>>> 'Η Κατερίνα είναι %d χρονών.' % age
'Η Κατερίνα είναι 12 χρονών.'
>>> '%5d' % 50
'   50'
>>> '%10.2f' % 50.1
'     50.10'
>>>
```

9.3 «Άλμευση» (Pickling)

Είδαμε ότι οτιδήποτε γράφουμε σε ένα αρχείο πρέπει πρώτα να μετατραπεί σε συμβολοσειρά. Αυτό έχει ως αποτέλεσμα να χάνεται η αρχική πληροφορία για τον τύπο των τιμών που γράφουμε στο αρχείο και μάλιστα δεν ξέρουμε πού τελειώνει η μια τιμή και πού αρχίζει η επόμενη. Για παράδειγμα:

```
>>> f = open('C:\myfolder\myfile5.txt', 'w')
>>> f.write(str(3.14))
4
>>> f.write(str([1,2,3]))
9
>>> f.close()
>>> f = open('C:\myfolder\myfile5.txt', 'r')
>>> f.readline()
'3.14[1, 2, 3]'
>>> f.close()
>>>
```

Η λύση είναι η **«άλμευση»** (**pickling**) η οποία «συντηρεί» τις δομές δεδομένων. Εισάγουμε το άρθρωμα `pickle` και μετά ανοίγουμε το αρχείο όπως κάνουμε συνήθως με mode **'wb'** (write binary). Για αποθήκευση μιας δομής δεδομένων χρησιμοποιούμε τη συνάρτηση `dump` και κλείνουμε το αρχείο με τον συνηθισμένο τρόπο. Για να διαβάσουμε μια δομή δεδομένων από ένα αρχείο, χρησιμοποιούμε τη συνάρτηση `load`. Το άνοιγμα του αρχείου πρέπει να γίνει με mode **'rb'** (read binary). Για παράδειγμα:

```
>>> import pickle
>>> f = open('C:\myfolder\myfile6.txt', 'wb')
>>> pickle.dump(3.14, f)
>>> pickle.dump([1,2,3], f)
>>> f.close()
>>>
```

```
>>> f = open('C:\myfolder\myfile6.txt', 'rb')
>>> x = pickle.load(f)
>>> type(x)
<class 'float'>
>>> print(x)
3.14
>>> l = pickle.load(f)
>>> type(l)
<class 'list'>
>>> print(l)
[1, 2, 3]
>>> f.close()
>>>
```

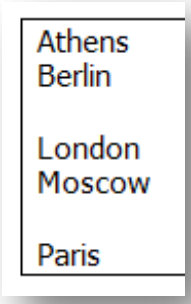
Γράψτε ένα πρόγραμμα το οποίο να αποτελείται από τα παρακάτω μέρη:

Α. Τον ορισμό συνάρτησης η οποία διαβάζει επαναληπτικά τα ονόματα κάποιων πόλεων και τα αποθηκεύει γραμμή γραμμή σε ένα νέο αρχείο. Η επανάληψη τερματίζει όταν δοθεί ως όνομα το `END`*. Το όνομα του αρχείου δίνεται ως όρισμα στη συνάρτηση. Για παράδειγμα, το αρχείο μπορεί να έχει την παρακάτω μορφή:*

```
Paris
Athens
London
Berlin
Moscow
```

Β Τον ορισμό συνάρτησης η οποία δέχεται ως όρισμα το όνομα ενός αρχείου το οποίο έχει τη μορφή του αρχείου που δημιουργεί η προηγούμενη συνάρτηση. Μετά το άνοιγμα του αρχείου η συνάρτηση θα πρέπει να διαβάζει γραμμή γραμμή τα δεδομένα που είναι αποθηκευμένα σε αυτό (τα ονόματα των πόλεων) και να τα αποθηκεύει σε μία λίστα την οποία και θα ταξινομεί κατά αύξουσα σειρά. Στη συνέχεια θα δημιουργεί ένα νέο αρχείο με όνομα `towns.txt` *και στο οποίο θα αποθηκεύει γραμμή γραμμή τα ονόματα των πόλεων που υπάρχουν στην*

ταξινομημένη λίστα και αφήνοντας μία κενή γραμμή ανά δύο πόλεις. Για το παράδειγμά μας το νέο αρχείο θα πρέπει να έχει την παρακάτω μορφή:



Κεφάλαιο 10

Εξαιρέσεις

Οι **εξαιρέσεις** (**exceptions**) μας βοηθούν να διαχειριστούμε απρόσμενα λάθη κατά τη διάρκεια της εκτέλεσης ενός προγράμματος. Ένα χαρακτηριστικό παράδειγμα είναι η προσπάθεια ανοίγματος για ανάγνωση ενός αρχείου που δεν υπάρχει.

Η Python, για να μας βοηθήσει στην περίπτωση ενός λάθους, **εγείρει μια εξαίρεση** (**raises an exception**). Μια εξαίρεση είναι ένα αντικείμενο λάθους που μπορούμε να το «πιάσουμε» (catch), να το εξετάσουμε και στη συνέχεια να το διαχειριστούμε με κατάλληλες εντολές. Αν δεν διαχειριστούμε μια εξαίρεση, το πρόγραμμά μας θα τερματίσει απρόσμενα και θα εμφανιστεί ένα αναλυτικό μήνυμα λάθους με τη μορφή του **ίχνους** (**traceback**). Το ίχνος εμφανίζει αναλυτικές πληροφορίες για να μπορέσουμε να προσδιορίσουμε ακριβώς τη γραμμή στην οποία συνέβη το λάθος κατά την εκτέλεση του προγράμματός μας.

10.1 Διαχείριση εξαιρέσεων

Ας προσπαθήσουμε να ανοίξουμε για ανάγνωση ένα αρχείο που δεν υπάρχει, ώστε να δούμε το ίχνος (traceback) που εμφανίζεται:

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

Η εξαίρεση στο παραπάνω παράδειγμα είναι η `FileNotFoundError`. Ας δούμε πώς μπορούμε να διαχειριστούμε μια τέτοια εξαίρεση:

Κώδικας 10.1

```
fname = input('Δώστε όνομα αρχείου:')

try:
    f = open(fname, 'r')
    print(f.read())
    f.close()
except FileNotFoundError:
    print('Δεν υπάρχει αρχείο με όνομα', fname)
```

Run

```
>>> ================================ RESTART ===
>>>
Δώστε όνομα αρχείου:myfile.txt
Δεν υπάρχει αρχείο με όνομα myfile.txt
```

Για να «πιάσουμε» και να διαχειριστούμε εξαιρέσεις χρησιμοποιούμε ένα μπλοκ `try/except`. Στο `try` μέρος του μπλοκ βάζουμε όποιες εντολές θέλουμε, έχοντας υπόψη μας ότι η εκτέλεση κάποιας ή κάποιων από αυτές μπορεί να εγείρει μια εξαίρεση. Εάν εγερθεί εξαίρεση, τότε η ροή του προγράμματος μεταφέρεται άμεσα στο `except` μέρος του μπλοκ και εκτελούνται οι εντολές που υπάρχουν σε αυτό. Στον κώδικα 10.1 αν δώσουμε όνομα αρχείου που δεν υπάρχει, τότε αντί να εμφανιστεί ένα ενοχλητικό κόκκινο μήνυμα λάθους θα εμφανιστεί ένα φιλικό και κατατοπιστικό μήνυμα για το τι πήγε στραβά.

Μια άλλη συνηθισμένη εξαίρεση είναι η `ValueError` που έχει σχέση με τον τύπο μιας τιμής που διαχειρίζεται ένα πρόγραμμα. Ας υποθέσουμε ότι θέλουμε να δημιουργήσουμε ένα πρόγραμμα στο οποίο ο χρήστης που θα το εκτελέσει θα πρέπει να πληκτρολογήσει έναν ακέραιο αριθμό. Στον παρακάτω κώδικα βλέπουμε μια εκδοχή αντιμετώπισης του προβλήματος:

Κώδικας 10.2

```
def read_age():
    while True:
        try:
            n = int(input('Δώστε την ηλικία σας:'))
            return n
        except ValueError:
            print('Πρέπει να δώσετε ακέραιο αριθμό.')


age = read_age()
print(age)
```


```
>>> ================================ RESTART ===
>>>
Δώστε την ηλικία σας:deka
Πρέπει να δώσετε ακέραιο αριθμό.
Δώστε την ηλικία σας:10.0
Πρέπει να δώσετε ακέραιο αριθμό.
Δώστε την ηλικία σας:10
10
```

Ας δούμε και άλλους τύπους εξαιρέσεων (`ZeroDivisionError`, `IndexError`, `TypeError`) στο παρακάτω παράδειγμα:

```
>>> print(5/0)
Traceback (most recent call last):
  File "<pyshell#37>", line 1, in <module>
    print(5/0)
ZeroDivisionError: division by zero
>>> mylist = [1, 2, 3]
>>> print(mylist[4])
Traceback (most recent call last):
  File "<pyshell#39>", line 1, in <module>
    print(mylist[4])
IndexError: list index out of range
>>> s = 'hello'
>>> s[0] = 'H'
Traceback (most recent call last):
  File "<pyshell#41>", line 1, in <module>
    s[0] = 'H'
TypeError: 'str' object does not support item assignment
>>>
```

Γράψτε παραδείγματα προγραμμάτων τα οποία θα «πιάνουν» και θα διαχειρίζονται εξαιρέσεις `ZeroDivisionError` *και* `IndexError`*.*

Κεφάλαιο 11

Αντικειμενοστρεφής προγραμματισμός

Ο **αντικειμενοστρεφής προγραμματισμός** (**object-oriented programming**) αποτελεί μια διαδεδομένη προσέγγιση για τη δημιουργία προγραμμάτων. Αντικειμενοστρεφής είναι ο χαρακτηρισμός που σημαίνει «στραμμένος» (προσανατολισμένος) στα **αντικείμενα** (**objects**). Τα αντικείμενα συνδυάζουν δεδομένα και λειτουργίες, και είναι ιδιαίτερα χρήσιμα, όταν τα προγράμματα είναι μεγάλα σε μέγεθος και πολύπλοκα σε δομή και λειτουργικότητα. Μία **κλάση** (**class**) είναι ένα πρότυπο (καλούπι) για τη δημιουργία αντικειμένων. Στην ουσία μια κλάση δημιουργεί έναν νέο τύπο, στον οποίο τα αντικείμενα ονομάζονται και **υποστάσεις** (**instances**) της κλάσης. Η κλάση καθορίζει τι δεδομένα και ποιες λειτουργίες πάνω σε αυτά θα περιέχονται στα αντικείμενα. Τα δεδομένα αποθηκεύονται σε απλές μεταβλητές που ανήκουν στα αντικείμενα και ονομάζονται **ιδιότητες** (**properties**). Οι λειτουργίες ορίζονται με τη μορφή συναρτήσεων που ονομάζονται **μέθοδοι** (**methods**).

Η Python είναι αντικειμενοστρεφής γλώσσα προγραμματισμού. Οι αριθμοί, οι συμβολοσειρές, οι λίστες, τα λεξικά, τα αρχεία και οι συναρτήσεις είναι αντικείμενα. Για παράδειγμα, οι μεταβλητές που αποθηκεύουν ακέραιους αριθμούς είναι υποστάσεις (αντικείμενα) της κλάσης `int`:

```
>>> i = 5
>>> type(i)
<class 'int'>
>>>
```

Με τον ορισμό μιας κλάσης στην Python μπορούμε και εμείς να δημιουργήσουμε έναν νέο σύνθετο τύπο.

11.1 Ορισμός κλάσης

Ας προσπαθήσουμε να ορίσουμε μια νέα κλάση, η οποία θα αντιπροσωπεύει ένα πρόσωπο (person):

Κώδικας 11.1

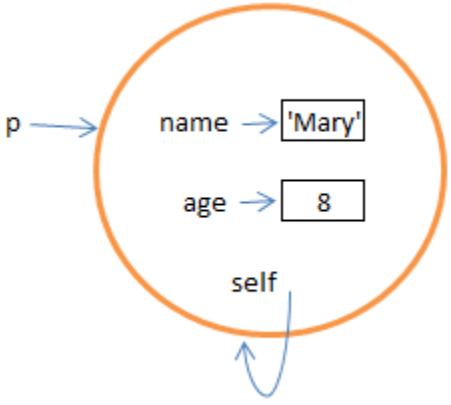
```
class Person:
    def __init__(self):
        self.name = ''
        self.age = 0
```

Στον κώδικα 11.1 ορίσαμε μια κλάση με όνομα `Person`. Τα δεδομένα για αυτή την κλάση είναι το όνομα και η ηλικία (ιδιότητες). Μοναδική συνάρτηση, προς το παρόν, ορίσαμε την ειδική συνάρτηση (μέθοδο) αρχικοποίησης τιμών `__init__` στην οποία θα αναφερθούμε λεπτομερώς παρακάτω. Ορισμοί κλάσης είναι δυνατό να βρίσκονται παντού σε ένα πρόγραμμα, αλλά συνήθως καλό είναι να τοποθετούνται κοντά στην αρχή (μετά τις εντολές `import`). Για να δημιουργήσουμε και να τυπώσουμε ένα αντικείμενο τύπου `Person`, μπορούμε να γράψουμε τις παρακάτω εντολές μετά την εκτέλεση του κώδικα 11.1:


```
>>> ================================ RESTART ===
>>>
>>> p = Person()
>>> p
<__main__.Person object at 0x02E274D0>
>>> p.name
''
>>> p.age
0
>>> p.name = 'Mary'
>>> p.name
'Mary'
>>> p.age = 8
>>> p.age
8
>>>
```

Βλέπουμε ότι, για να δημιουργήσουμε ένα αντικείμενο `Person`, απλά καλούμε `Person()`. Αυτό έχει ως αποτέλεσμα να εκτελεστεί η μέθοδος αρχικοποίησης `__init__` και να επιστραφεί ένα νέο αντικείμενο τύπου `Person`. Η `__init__` καλείται μόνο μία φορά, όταν το αντικείμενο δημιουργείται, και χρησιμοποιείται, για να δοθούν αρχικές τιμές στις ιδιότητες. Οι μεταβλητές `name` και `age` που αντιπροσωπεύουν τις ιδιότητες του αντικειμένου βρίσκονται μέσα σε αυτό και κάθε νέο αντικείμενο τύπου `Person` που δημιουργείται έχει το δικό του αντίγραφο των μεταβλητών αυτών. Για να προσπελαστεί η μεταβλητή `name` ή η μεταβλητή `age`, χρησιμοποιείται ο συμβολισμός dot notation (`p.name` ή `p.age`). Γενικά, τα αντικείμενα μπορούν να χρησιμοποιηθούν όπως οποιονδήποτε άλλο τύπο δεδομένων, π.χ. να αποθηκευτούν σε λίστες, πλειάδες και λεξικά ή να περαστούν ως ορίσματα σε συναρτήσεις.

Η μέθοδος `__init__` έχει ως πρώτη παράμετρο τη `self`. Η `self` είναι μια μεταβλητή που αναφέρεται στον εαυτό του αντικειμένου (στο ίδιο το αντικείμενο). Με άλλα λόγια η λέξη `self` σημαίνει ότι η συμπεριφορά που προσδιορίζεται αφορά το αντικείμενο από το οποίο καλείται. Η `self` απαιτείται να υπάρχει ως η πρώτη παράμετρος και σε κάθε άλλη μέθοδο που ορίζεται σε μια κλάση. Παρόλο που μπορούμε να δώσουμε οποιοδήποτε όνομα σε αυτή την παράμετρο, συνιστάται ως σύμβαση στην Python το όνομα `self`. Το βασικότερο πλεονέκτημα για τη χρήση ενός πρότυπου ονόματος μεταβλητής στα προγράμματά μας είναι ότι ο κάθε αναγνώστης τους θα το αναγνωρίσει άμεσα. Σε άλλες αντικειμενοστρεφείς γλώσσες προγραμματισμού, όπως την Java και την C++, χρησιμοποιείται υποχρεωτικά το όνομα `this`. Ας δούμε σχηματικά το αντικείμενο που δημιουργήσαμε στο προηγούμενο παράδειγμα:



11.2 Εμφάνιση αντικειμένων

Ένας απλός τρόπος για να τυπώνουμε με ευκρινή τρόπο τα περιεχόμενα (δεδομένα) ενός αντικειμένου είναι να ορίσουμε μια κατάλληλη μέθοδο στην κλάση του. Για παράδειγμα:

Κώδικας 11.2

```
class Person:
    def __init__(self):
        self.name = ''
        self.age = 0

    def display(self):
        print('Πρόσωπο(%s, %d)' % (self.name, self.age))
```


```
>>> ================================ RESTART ===
>>>
>>> p = Person()
>>> p.name = 'Mary'
>>> p.age = 8
>>> p.display()
Πρόσωπο(Mary, 8)
```

Εναλλακτικά, η Python μάς προσφέρει και τις ειδικές μεθόδους `__str__` και `__repr__` για την εμφάνιση με τη μορφή συμβολοσειράς της αναπαράστασης ενός αντικειμένου. Ταυτόχρονα, μπορούμε να βελτιώσουμε και τη μέθοδο αρχικοποίησης `__init__` περνώντας ως ορίσματα τις ιδιότητες του αντικειμένου που δημιουργείται κάθε φορά. Για παράδειγμα:

Κώδικας 11.3

```
class Person:
    def __init__(self, name = '', age = 0):
        self.name = name
        self.age = age

    def __str__(self):
        return 'Πρόσωπο(%s, %d)' % (self.name, self.age)

    def __repr__(self):
        return str(self)
```


```
>>> ================================ RESTART ===
>>>
>>> p = Person('Mary', 8)
>>> str(p)
'Πρόσωπο(Mary, 8)'
>>> p
Πρόσωπο(Mary, 8)
```

11.3 Ισότητα αντικειμένων

Όταν μιλάμε για ισότητα στα αντικείμενα, υπάρχει αμφισημία. Όταν λέμε ότι δύο αντικείμενα είναι τα ίδια, εννοούμε ότι περιέχουν τα ίδια δεδομένα ή ότι είναι στην πραγματικότητα το ίδιο αντικείμενο; Ας δούμε ένα παράδειγμα:

Εκτέλεση κώδικα 11.3


```
>>> ================================ RESTART ===
>>>
>>> p1 = Person('Mary', 8)
>>> p2 = Person('Mary', 8)
>>> p1 == p2
False
>>> p3 = p1
>>> p3 == p1
True
```

11.4 Κληρονομικότητα

Η **κληρονομικότητα** (**inheritance**) είναι η διεργασία μέσω της οποίας μια κλάση μπορεί να αποκτήσει (κληρονομήσει) τις ιδιότητες και τις μεθόδους μιας άλλης κλάσης. Η κληρονομικότητα βοηθάει στην ουσία στην επαναχρησιμοποίηση κώδικα με ελάχιστες ή και καθόλου αλλαγές. Η αρχική κλάση ονομάζεται **γονική κλάση** (**parent class**) και η νέα κλάση που κληρονομεί ιδιότητες και μεθόδους της γονικής κλάσης ονομάζεται **κλάση-παιδί** (**child class**) ή **υποκλάση** (**subclass**). Σε μια υποκλάση μπορούμε να ορίσουμε, αν χρειάζεται, και νέες επιπρόσθετες ιδιότητες και μεθόδους. Στον ορισμό μιας υποκλάσης το όνομα της γονικής κλάσης εμφανίζεται μέσα σε παρενθέσεις:

```
class ParentClass:
    pass

class ChildClass(ParentClass):
    pass
```

Ας δούμε ένα παράδειγμα που αφορά σε μια στοιχειώδη καταγραφή δεδομένων που αφορούν σε καθηγητές και φοιτητές ενός Πανεπιστημίου. Οι καθηγητές και οι φοιτητές έχουν κάποια κοινά χαρακτηριστικά (π.χ. όνομα, ηλικία), αλλά και κάποια ιδιαίτερα χαρακτηριστικά (π.χ. μισθός για καθηγητές, ΑΜ για 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 ανεξάρτητες κλάσεις. Η προσέγγιση αυτή είναι δύσχρηστη. Η καλύτερη λύση είναι να δημιουργήσουμε μια κοινή γονική κλάση (`UniversityMember`), και οι κλάσεις Καθηγητής (`Professor`) και Φοιτητής (`Student`) να κληρονομήσουν από αυτήν:

Κώδικας 11.4

```
class UniversityMember:
    def __init__(self, name, age):
        self.name = name
        self.age = age
        print('Αρχικοποίηση μέλους του Πανεπιστημίου:', self.name)
    def display(self):
        print('Όνομα:"{0}" Ηλικία:"{1}"'.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 display(self):
        UniversityMember.display(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 display(self):
        UniversityMember.display(self)
        print('AM:', self.am)
```


```
>>> ================================ RESTART ================================
>>>
>>> p = Professor('Αναστασίου Αναστάσιος', 50, 1500)
Αρχικοποίηση μέλους του Πανεπιστημίου: Αναστασίου Αναστάσιος
Αρχικοποίηση καθηγητή: Αναστασίου Αναστάσιος
>>> p.display()
Όνομα:"Αναστασίου Αναστάσιος" Ηλικία:"50" Μισθός: 1500
>>> s = Student('Ελευθερίου Ελευθέριος', 20, 476)
Αρχικοποίηση μέλους του Πανεπιστημίου: Ελευθερίου Ελευθέριος
Αρχικοποίηση φοιτητή: Ελευθερίου Ελευθέριος
>>> s.display()
Όνομα:"Ελευθερίου Ελευθέριος" Ηλικία:"20" AM: 476
```

*Η **στοίβα** είναι μια αφηρημένη δομή δεδομένων που μπορεί να περιέχει πολλαπλά στοιχεία και στην οποία μπορούν να εφαρμοστούν μόνο οι παρακάτω λειτουργίες:*

Αρχικοποίηση μιας νέας κενής στοίβας.

Είσοδος ενός στοιχείου στη στοίβα.

Αφαίρεση και επιστροφή του τελευταίου στοιχείου που έχει εισαχθεί στη στοίβα.

Έλεγχος για το αν η στοίβα είναι άδεια.

Το τελευταίο στοιχείο που εισάγεται στη στοίβα είναι αυτό που αφαιρείται πρώτο

Να γράψετε ένα πρόγραμμα το οποίο να αποτελείται από τα παρακάτω μέρη:

*Τον ορισμό μιας κλάσης με όνομα **Stack** που να υλοποιεί μία **στοίβα** με τη χρήση **λίστας**. Πρέπει να περιλαμβάνει μέθοδο αρχικοποίησης νέας κενής στοίβας, μέθοδο εισόδου στοιχείου στη στοίβα (**push**), μέθοδο αφαίρεσης και επιστροφής στοιχείου από τη στοίβα (**pop**) και μέθοδο ελέγχου για το αν η στοίβα είναι άδεια (**isEmpty***

*Τον ορισμό μιας υποκλάσης (κλάσης παιδί) της **Stack** με όνομα **StackPlus** η οποία να περιέχει τον ορισμό μιας μεθόδου που επιστρέφει την αναπαράσταση ενός αντικειμένου τύπου στοίβας ως συμβολοσειρά ώστε να μπορεί να τυπώνεται με χρήση της εντολής* `print`

*Κυρίως πρόγραμμα το οποίο να δημιουργεί μία νέα στοίβα ως ένα αντικείμενο της κλάσης **StackPlus**, να εισάγει σε αυτήν 3 στοιχεία (π.χ. συμβολοσειρές), να αφαιρεί ένα στοιχείο από αυτήν, αφού πρώτα ελέγξει για το αν είναι άδεια, και στο τέλος να τυπώνει το αντικείμενο (τη στοίβα).*

Κεφάλαιο 12

Γραφική διεπαφή χρήστη

Η γραφική διεπαφή χρήστη (**graphical user interface**, **GUI**) προσφέρει ένα σύνολο γραφικών στοιχείων τοποθετημένων σε παράθυρα (π.χ. κουμπιών, μενού, μπαρών κύλισης, εικονιδίων, κ.λπ.) και με σκοπό την ευκολότερη και διαισθητικότερη αλληλεπίδραση του χρήστη με ένα πρόγραμμα.

Η Python μάς προσφέρει πάρα πολλές επιλογές για την ανάπτυξη προγραμμάτων που βασίζονται σε γραφική διεπαφή χρήστη. Στο κεφάλαιο αυτό θα δούμε τη βασική και συνάμα απλή επιλογή που βασίζεται στο άρθρωμα `tkinter` (tk interface), το οποίο εγκαθίσταται μαζί με την εγκατάσταση της Python.

12.1 Η πρώτη μας γραφική διεπαφή

Για να δημιουργήσουμε μια γραφική διεπαφή εισάγουμε αρχικά το άρθρωμα `tkinter`, το οποίο περιέχει όλες τις απαραίτητες κλάσεις και μεθόδους που θα χρειαστούμε. Στη συνέχεια, κάνουμε αρχικοποίηση με τη δημιουργία ενός γραφικού στοιχείου ρίζας (αντικείμενο παραθύρου) της κλάσης `Tk`. Πρόκειται για ένα κανονικό παράθυρο με τη γραμμή τίτλου και την οποιαδήποτε άλλη διακόσμηση που προβλέπεται από τον διαχειριστή παραθύρων που τρέχουμε.

Για κάθε πρόγραμμα δημιουργούμε ένα μόνο γραφικό στοιχείο ρίζα και πάντα πριν τα άλλα γραφικά στοιχεία. Μετά δημιουργούμε ένα γραφικό στοιχείο `Label` (αντικείμενο ετικέτας) ως παιδί του παραθύρου-ρίζας. Ένα στοιχείο `Label` μπορεί να δείξει κείμενο ή κάποια εικόνα. Κατόπιν, επικαλούμαστε τη μέθοδο `pack` πάνω στο γραφικό στοιχείο `Label`, ώστε να πάρει κατάλληλο

μέγεθος, για να χωράει το κείμενο και να κάνει ορατό τον εαυτό του. Το γραφικό στοιχείο γίνεται ορατό, όταν το πρόγραμμά μας μπει στον κύριο βρόγχο συμβάντων (event loop) του `tkinter`. Το πρόγραμμα θα μείνει σε αυτό τον βρόγχο συμβάντων μέχρι να κλείσουμε το παράθυρο. Ο βρόγχος συμβάντων δεν διαχειρίζεται μόνο συμβάντα από τον χρήστη (π.χ. κλικ ποντικιού, πάτημα πληκτρολογίου) ή του συστήματος διαχείρισης παραθύρων (π.χ. συμβάντα επανασχεδίασης), χειρίζεται επίσης και πράξεις που βάζει στην ουρά συμβάντων το ίδιο το `tkinter` (π.χ. διαχείριση της γεωμετρίας και διάφορες αλλαγές στα παράθυρα). Τα γραφικά στοιχεία ονομάζονται widgets.

Ας δούμε ένα πρώτο απλό παράδειγμα:

Κώδικας 12.1

```
from tkinter import *

root = Tk()
root.title('Hello window')
w = Label(root, text = 'Hello, World!')
w.pack()
root.mainloop()
```

⬇ Run:

Η κλάση `Tk` διαθέτει τις μεθόδους `title` για εμφάνιση τίτλου σε ένα παράθυρο (π.χ. `root.title('Hello window')`) και `destroy` για «καταστροφή» (κλείσιμο) παραθύρου (π.χ. `root.destroy()`).

12.2 Δημιουργία πλαισίου και κουμπιών

Κώδικας 12.2 (ενδεικτικό απόσπασμα παραδείγματος)

```
from tkinter import *

class Application():
    def __init__(self, master):
        frame = Frame(master)
        frame.pack()
        self.qbutton = Button(frame, text='Quit', fg='red', command=root.destroy)
        self.qbutton.pack(side=LEFT)
        self.mbutton = Button(frame, text='Message', command=self.printMessage)
        self.mbutton.pack(side=LEFT)
    def printMessage(self):
        print('Hello, World!')

root = Tk()
app = Application(root)
root.mainloop()
```

⬇ Run:

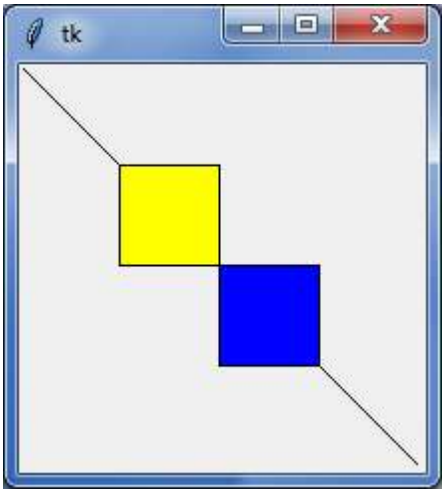
Στον κώδικα 12.2 το πρόγραμμα είναι γραμμένο ως κλάση. Το γραφικό στοιχείο `Frame` (πλαίσιο) μπορεί να περιέχει άλλα γραφικά στοιχεία και να έχει ακόμα και τρισδιάστατο περίγραμμα. Το γραφικό στοιχείο `Button` (κουμπί) χρησιμοποιείται για τη δημιουργία κουμπιών. Στο παράδειγμα, κάθε φορά που πατάμε το κουμπί `Message` εμφανίζεται το μήνυμα `Hello, World!` στο Python Shell. Αν πατήσουμε το κουμπί `Quit`, κλείνει το παράθυρο. Το πλαίσιο και τα κουμπιά «πακετάρονται» με επίκληση της μεθόδου `pack`.

12.3 Δημιουργία καμβά για ζωγραφική

Κώδικας 12.3 (ενδεικτικό απόσπασμα παραδείγματος)

```
from tkinter import *

root = Tk()

canvas = Canvas(root, width=200, height=200)
canvas.pack()
canvas.create_line(0, 0, 200, 200)
canvas.create_rectangle(50, 50, 100, 100, fill = 'yellow')
canvas.create_rectangle(100, 100, 150, 150, fill = 'blue')

root.mainloop()
```

⬇ Run:

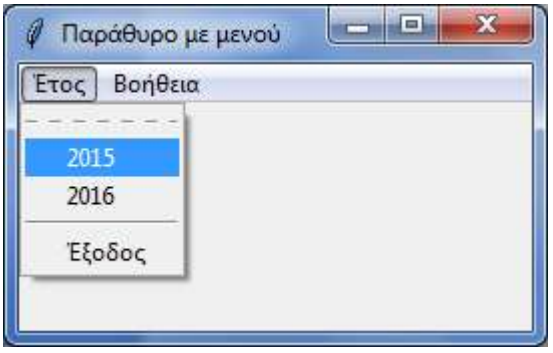


12.4 Δημιουργία μενού επιλογών

Κώδικας 12.4 (ενδεικτικό απόσπασμα παραδείγματος)

```
from tkinter import *

def testing():
    print('Just testing!')

root = Tk()
root.title('Παράθυρο με μενού')
menu = Menu(root)
root.config(menu = menu)
yearmenu = Menu(menu)
menu.add_cascade(label = 'Έτος', menu = yearmenu)
yearmenu.add_command(label = '2015', command = testing)
yearmenu.add_command(label = '2016', command = testing)
yearmenu.add_separator()
yearmenu.add_command(label = 'Έξοδος', command = root.destroy)
helpmenu = Menu(menu)
menu.add_cascade(label = 'Βοήθεια', menu = helpmenu)
helpmenu.add_command(label = 'About...', command = testing)

root.mainloop()
```

⬇ Run:

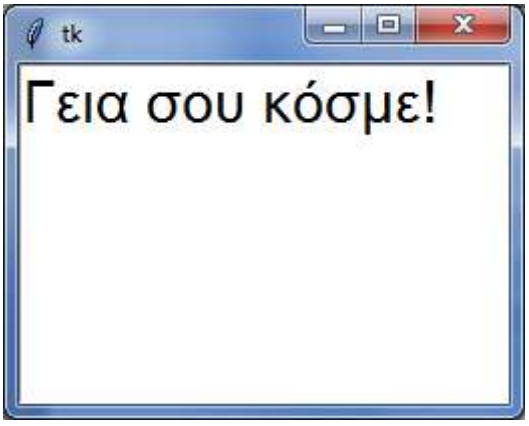


12.5 Δημιουργία περιοχής κειμένου

Κώδικας 12.5 (ενδεικτικό απόσπασμα παραδείγματος)

```
from tkinter import *

root = Tk()

textfr = Frame(root)
textfr.pack(side = TOP)
text = Text(textfr, height = 5, width = 15, font = 'Arial 22')
text.pack(side = LEFT)
text.insert(END, 'Γεια σου κόσμε!')

root.mainloop()
```

⬇ Run:

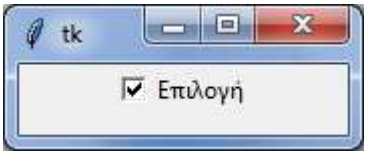


12.6 Δημιουργία check button

Κώδικας 12.6 (ενδεικτικό απόσπασμα παραδείγματος)

```
from tkinter import *

root = Tk()

cbfr = Frame(root)
cbfr.pack(side = TOP)
var = IntVar()
c = Checkbutton(cbfr, text = 'Επιλογή', state = NORMAL, variable = var)
c.pack(side = LEFT)

root.mainloop()
```

➡ Run:

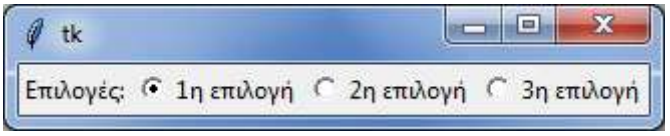


12.7 Δημιουργία radio button

Κώδικας 12.7 (ενδεικτικό απόσπασμα παραδείγματος)

```
from tkinter import *

def radiofunction1():
    print('1η επιλογή')
def radiofunction2():
    print('2η επιλογή')
def radiofunction3():
    print('3η επιλογή')

root = Tk()

rbfr = Frame(root)
rbfr.pack(side=TOP, fill=X)
Label(rbfr,text='Επιλογές:').pack(side=LEFT)
basis = IntVar()
r1 = Radiobutton(rbfr,text='1η επιλογή',value=1,variable=basis,command=radiofunction1)
r1.pack(side=LEFT)
r2 = Radiobutton(rbfr,text='2η επιλογή',value=2,variable=basis,command=radiofunction2)
r2.pack(side=LEFT)
r3 = Radiobutton(rbfr,text='3η επιλογή',value =3,variable=basis,command=radiofunction3)
r3.pack(side=LEFT)

root.mainloop()
```

➡ Run:

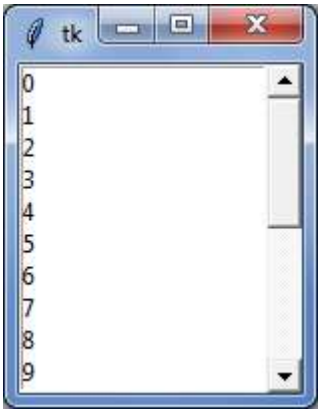


12.8 Δημιουργία scrollbar

Κώδικας 12.8 (ενδεικτικό απόσπασμα παραδείγματος)

```
from tkinter import *

root = Tk()

scrollbar = Scrollbar(root)
scrollbar.pack(side = RIGHT, fill = Y)

listbox = Listbox(root)
listbox.pack()

for i in range(20):
    listbox.insert(END, i)

listbox.config(yscrollcommand = scrollbar.set)
scrollbar.config(command = listbox.yview)

root.mainloop()
```

⬇ Run:

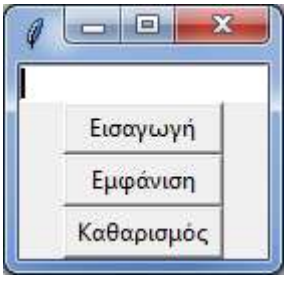


12.9 Δημιουργία πεδίου εισόδου

Κώδικας 12.9 (ενδεικτικό απόσπασμα παραδείγματος)

```
from tkinter import *

root = Tk()
e = Entry(root)
e.pack()
e.focus_set()

def getFunction():
    t = e.get()
    print(t)
def insertFunction():
    e.delete(0,10)
    print(e.insert(0, 'Hello'))
def clearFunction():
    e.delete(0,10)

b1 = Button(root, text = 'Εισαγωγή', width = 10, command = getFunction)
b1.pack()
b2 = Button(root, text = 'Εμφάνιση', width = 10, command = insertFunction)
b2.pack()
b3 = Button(root, text = 'Καθαρισμός', width = 10, command = clearFunction)
b3.pack()

root.mainloop()
```

➡ Run:



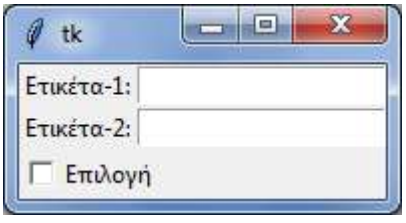
12.10 Grid geometry

Για να δημιουργήσουμε πιο πολύπλοκες εμφανίσεις γραφικών στοιχείων, χρησιμοποιούμε Grid geometry αντί του «πακεταρίσματος» με επίκληση της μεθόδου `pack`. Ο διαχειριστής του Grid geometry τοποθετεί τα γραφικά στοιχεία σε έναν δισδιάστατο πίνακα με γραμμές (rows) και στήλες (columns). Το κάθε κελί του πίνακα κρατάει ένα μόνο γραφικό στοιχείο.

Κώδικας 12.10 (ενδεικτικό απόσπασμα παραδείγματος)

```
from tkinter import *

root = Tk()

Label(root, text='Ετικέτα-1:').grid(row=0, sticky=W)
Label(root, text='Ετικέτα-2:').grid(row=1, sticky=W)

e1 = Entry(root)
e2 = Entry(root)

e1.grid(row=0, column=1)
e2.grid(row=1, column=1)

cb = Checkbutton(root, text='Επιλογή')
cb.grid(row=2, columnspan=2, sticky=W)

root.mainloop()
```

⬇ Run:



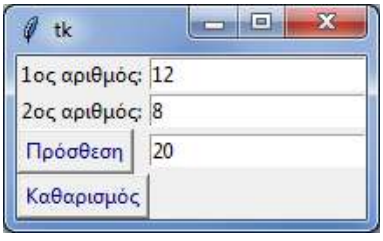
12.11 Παράδειγμα δημιουργίας αριθμομηχανής

Στο παρακάτω παράδειγμα θα αναπτύξουμε μια απλή αριθμομηχανή που μπορεί να προσθέτει δύο αριθμούς:

Κώδικας 12.11

```
from tkinter import *

def add_function():
    e3.insert(0, str(int(e1.get())+int(e2.get())))

def clear_entries():
    e1.delete(0,10)
    e2.delete(0,10)
    e3.delete(0,10)

root = Tk()

l1 = Label(root, text='1ος αριθμός:')
l1.grid(row=0, sticky=W)
e1 = Entry(root)
e1.grid(row=0, column=1)
l2 = Label(root, text='2ος αριθμός:')
l2.grid(row=1, sticky=W)
e2 = Entry(root)
e2.grid(row=1, column=1)
b1 = Button(root, text='Πρόσθεση', fg='navy', command=add_function)
b1.grid(row=2, sticky=W)
e3 = Entry(root)
e3.grid(row=2, column=1)
b2 = Button(root, text='Καθαρισμός', fg='navy', command=clear_entries)
b2.grid(row=3, sticky=W)

root.mainloop()
```

⬇ Run:



Επεκτείνατε τον κώδικα 12.11 ώστε η αριθμομηχανή να εκτελεί όλες τις βασικές αριθμητικές πράξεις (πρόσθεση, αφαίρεση, πολλαπλασιασμό, διαίρεση). Προσέξτε το θέμα της διαίρεσης με το μηδέν.

Γράψτε ένα πρόγραμμα το οποίο να δημιουργεί μία γραφική διεπαφή χρήστη, η οποία θα περιλαμβάνει όλα τα απαραίτητα γραφικά στοιχεία (ετικέτες, πεδία εισόδου, κουμπιά), ώστε ο χρήστης να εισάγει μια θερμοκρασία σε κλίμακα Κελσίου (C) και με το πάτημα ενός κουμπιού αυτή να μετατρέπεται σε κλίμακα Φαρενάιτ (F).

Σημείωση

Βιβλιογραφία:

Downey, A., Elkner, J., and Meyers, C. (2002)

.

Swaroop, C. H. (2008). .

Wentworth, P., Elkner, J., Downey, A., and Meyers, C. (2012

.

Python Software Foundation (2015). Ημερομηνία τελευταίας προσπέλασης: Αύγουστος 2015, από https://www.python.org

Πανσεληνάς, Γ., Αγγελιδάκης, Ν., Μιχαηλίδη, Α., Μπλάτσιος, Χ., Παπαδάκης, Σ., Παυλίδης, Γ., Τζαγκαράκης, Ε., Τζωρμπατζάκης, Α. (2014). *Εφαρμογές Πληροφορικής Α΄ Γενικού Λυκείου.* Υπουργείο Παιδείας και Θρησκευμάτων. Ινστιτούτο Εκπαιδευτικής Πολιτικής. ΙΤΥΕ-Διόφαντος. ISBN: 978-960-06-4894-2.

ΝΙΚΟΛΑΟΣ Α. ΑΓΓΕΛΙΔΑΚΗΣ

Εισαγωγή στον προγραμματισμό με την Python

ISBN: 978-960-93-7364-7