TASPYTHON

ΕΚΜΑΘΗΣΗ PYTHON ΒΗΜΑ ΒΗΜΑ

Οδηγός Python Μέσω Παραδειγμάτων

Συγγραφέας:
Δημήτρης Λεβεντεας

Ομάδα:
TasPython

Ευχαριστίες

Ο παρόν οδηγός φιλοδοξεί να συμπληρώσει μια προσπάθεια που άρχισε το 2008 από ορισμένους φοιτητές του τμήματος Μηχανικών Η/Υ & Πληροφορικής, που σκοπό είχαν να μάθουν καλύτερα μια γλώσσα προγραμματισμού που συναντούσαν αρκετά συχνά, μέσα από μια συλλογική προσπάθεια η οποία θα οδηγούσε στην υπέρβαση τουλάχιστον των περισσοτέρων από τις καθημερινές δυσκολίες που θα συναντούσαν. Η προσπάθεια αυτή ευδοκίμησε, και συνεχίζεται ακόμα, με συναντήσεις περίπου κάθε δυο εβδομάδες αλλά και καινούργια μέλη με τα οποία μοιραζόμαστε τις γνώσεις μας, τις εμπειρίες μας και όχι μόνο.

Θα ήθελα να ευχαριστήσω τον φίλο μου, Κωνσταντίνο Αραβανή, με τον οποίο αυτή η φιλία μας έχει προσλάβει διάφορα σχήματα μέσα από τις ιδιότητες μας ως συνεργάτες, διαχειριστές, προγραμματιστές, ονειροπόλους και ένα σωρό άλλα για την έμπνευση και την δύναμη που μου προσέφερε. Η συνεργασία μας είναι η ζωντανή απόδειξη τουλάχιστον για μένα ότι, παρά τα στερεότυπα, το να πετύχεις κάτι σημαντικό στον τομέα των υπολογιστών δεν είναι μια μοναχική δουλειά απαραίτητα.

Ένα μεγάλο ευχαριστώ οφείλω και στα άτομα τα οποία συνάντησα στην πορεία της ομάδας TasPython με τους οποίους μοιραστήκαμε γνώσεις, εμπειρίες, κίνητρα και ένα σωρό άλλα πράγματα.

Επίσης, θέλω να ευχαριστήσω όλους όσους έχουν διαβάσει τον οδηγό και έκαναν υποδείξεις ώστε να διορθωθεί και να εμπλουτιστεί.

Δημήτρης Λεβεντέας

Αφιερώνεται στη δημιουργικότητα και την συνεργατική της έκφραση.

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

> In times of change, learners will inherit the earth while the learned will find themselves beautifully equipped to deal with a world that no longer exists.
>
> Eric Hoffer

Ο οδηγός αυτός βασίζεται σε παραδείγματα τα οποία φιλοδοξούν να μας δώσουν μια πρώτη εξοικίωση με την Python 3. Ο συγκεκριμένος οδη.γός θα βρίσκεται υπό συνεχή ανανέωση και θα εμπλουτίζεται με το πέρασμα του χρόνου. Φιλοδοξεί να καλύψει το κενό που υπάρχει από οδηγούς για την συγκεκριμένη έκδοση της γλώσσας, ιδιαίτερα στην ελληνική. Όπως λέει και ένα αρχαίο ρωμαϊκό ρητό:

Longum iter est per preaecepta, breve et efficax per exempla!

Που σημαίνει ότι είναι ένας μεγάλος δρόμος μέσω των κανόνων, αλλά σύντομος και αποδοτικός με παραδείγματα. Ελπίζουμε πως και σε αυτό τον οδηγό, το παραπάνω ρητό θα αποδειχθεί αληθές.

Παράλληλα, θα προσπαθεί να ενσωματώνει όλα τα καινούργια χαρακτηριστικά της γλώσσας που κάθε καινούργια έκδοση της φέρνει.

1.1 Περιεχόμενα κεφαλαίου

Στο κεφάλαιο αυτό θα δούμε κάποιες γενικές ιδέες που αφορούν τον προγραμματισμό, πως να εγκαταστήσουμε την Πψτηον, που μπορούμε να γράφουμε τον κώδικα μας και πως τον εκτελούμε. Στο τέλος του κεφαλαίου παρατίθενται συνοπτικά τα περιεχόμενα του υπόλοιπου οδηγού. Αν έχετε ήδη κάποια εξοικίωση με τις διαδικασίες που περιγράφονται, προτείνεται να συνεχίσετε με το επόμενο κεφάλαιο. Ένας τρόπος να το εξακριβώσετε αυτό είναι να προσπαθήσετε να ακολοθήσετε τις οδηγίες που παρατίθενται παρακάτω. Αν βρείτε κάποια δυσκολία σε οποιοδήποτε βήμα, προτείνεται να διαβάσετε τις αντίστοιχες ενότητες από αυτό το κεφάλαιο, αλλιώς μπορείτε να προχωρήσετε στο επόμενο.

1. Ανοίξτε ένα αρχείο κειμένου.
2. Γράψτε τον ακόλουθο κώδικα.

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

3. Αποθηκεύστε τον σε ένα αρχείο με όνομα *ηελλο_ωορλδ.πψ*.
4. Εκτελέστε τον, γράφοντας *πψτηον ηελλο_ωορλδ.πψ*.

Αν τα καταφέρατε επιτυχώς, τότε έχετε όλες τις απαραίτητες τεχνικές γνώσεις για να προχωρήσετε στα επόμενα κεφάλαια. Αν έχετε επιπλέον χρόνο, μπορείτε να δείτε μια επισκόπηση των χαρακτηριστικών που θα γνωρίσουμε στις ακόλουθες ενότητες, παρακάτω καθώς και ορισμένες γενικότερες συμβουλές.

1.2 Πώς να μάθω προγραμματισμό

Πολύ συχνά η Python αποτελεί την πρώτη γλώσσα με την οποία έρχεται κάποιος σε επαφή. Άλλες φορές πάλι γίνεται ιδιαίτερα γνωστή λόγω της έντονης παρουσίας της στις διανομές Linux και τα τελευταία χρόνια στην ανάπτυξη εφαρμογών για το διαδίκτυο web applications (για παράδειγμα το Django). Όπως και να έχει, υπάρχουν ορισμένα βασικά πράγματα που

μπορούμε να κάνουμε για να γράφουμε πιο καλά προγράμματα, που δεν έχουν τόσο σχέση με την ίδια την γλώσσα αλλά ως γενικότερη φιλοσοφία. Μην ξεχνάμε όμως το πιο σημαντικό που είναι η καλή διάθεση και συνεχής εξάσκηση.

1.2.1 Χρησιμοποιείστε τον υπολογιστή μας

Αν θέλουμε να γίνουμε προγραμματιστής, δεν αρκεί μόνο να διαβάζουμε προγραμματιστικά βιβλία. Πρέπει να εφαρμόζουμε και αυτά που διαβάζουμε γράφοντας τα δικά μας προγράμματα. Επομένως το να πληκτρολογούμε προγράμματα, να τα μεταγλωττίζουμε (compile)[1] και να τα εκτελούμε πέρα από το προσδοκώμενο, είναι και κάτι αναπόφευκτο, ακόμα και κατά τη διάρκεια της εκπαιδευτικής διαδικασίας. Άλλωστε ο προγραμματισμός βρίσκεται στη τομή της τέχνης με την επιστήμη, καθώς χρειάζονται οι θεωρητικές γνώσεις για το γιατί δουλεύει κάτι και πως αυτό μπορεί να γίνει πιο αποτελεσματικά, αλλά ως γλώσσα, έστω και σε ένα τεχνικό περιβάλλον ο τρόπος που ξεδιπλώνεται ο κώδικας του προγράμματος μας είναι σε άμεση σχέση με τον τρόπο αυτού που το γράφει.

Για να προγραμματίσουμε δεν χρειάζεται να έχετε τον πιο γρήγορο υπολογιστή που κυκλοφορεί. Ακόμα και ένας υπολογιστής αρκετά παλιός μπορεί να κάνει αυτή την δουλειά. Επομένως δεν υπάρχουν δικαιολογίες.

1.2.2 Διαβάστε τον κώδικα άλλων προγραμματιστών

Όσο έξυπνοι και να είμαστε, δεν γίνεται να είμαστε οι καλύτεροι με ό,τι καταπιανόμαστε, τουλάχιστον όχι αμέσως. Για να γράψουμε πολύ καλά προγράμματα, πρέπει να δούμε πως γράφουν άλλοι προγραμματιστές καλά προγράμματα και να μάθουμε από αυτούς. Ευτυχώς υπάρχουν πολλές εφαρμογές ανοικτού λογισμικού των οποίων μπορούμε να μελετήσουμε τον κώδικα, ακόμα και αν δεν υπάρχει κάποιος άλλος προγραμματιστής στο περιβάλλον μας. Ίσως μάλιστα αργότερα να μπορέσουμε να συνεισφέρουμε σε κάποια από αυτά, αποδεικνύοντας σε έναν δυνητικό εργοδότη μας πως εκτός από να διαβάζουμε για προγραμματισμό, ξέρουμε να γράφουμε και καλά προγράμ-

[1] Η διαδικασία μετατροπής τους σε εκτελέσιμο κώδικα

ματα. Φυσικά βέβαια, αν αυτή η συνεισφορά μας γίνει σε προγράμματα που χρησιμοποιούμε και εμείς οι ίδιοι (πράγμα που συνιστάται ιδιαίτερα), τώρα θα έχουμε τη χαρά όταν χρησιμοποιούμε μια συγκεκριμένη λειτουργία, να νιώθουμε περήφανοι ότι έχουμε συμβάλει και εμείς το μικρό λιθαράκι μας σε αυτή τη προσπάθεια.

Πέρα από αυτό, αν δουλέψουμε ως προγραμματιστής, είναι σίγουρο πως θα χρειαστεί να διαβάσουμε τον κώδικα κάποιου άλλου και θα κληθούμε να τον συντηρήσουμε ή να τον βελτιώσουμε. Καλύτερα λοιπόν να έχουμε εξοικειωθεί με αυτό όσο το δυνατόν νωρίτερα, πόσο μάλλον στα πρώτα μας βήματα όπου και θα πάρουμε πολλές ιδέες. Βλέποντας τον κώδικα άλλων, θα γνωρίσουμε τον τρόπο με τον οποίο προγραμματίζουν, τις συνήθειες τους και πως αποφεύγουν λάθη στα οποία αλλιώς ίσως και να υποπέπταμε.

1.2.3 Πειραματιστείτε

Δεν πρέπει να μένουμε μόνο σε αυτά που μας λένε οι καθηγητές μας, οι φίλοι μας ή ακόμα και αυτός ο οδηγός. 'Πειραματιστείτε!', πρέπει να είναι ένα από τα συνθήματα μας. Ένα μεγάλο ποσοστό ανακαλύψεων έγιναν μέσω του πειραματισμού, ή ακόμα και κατά λάθος. Ανεξάρτητα από το πόσο καλή ή χαζή μας φαίνεται η ιδέα μας, ποτέ δεν θα μάθουμε πραγματικά μέχρι να δοκιμάσουμε. Αν δούμε περίεργα αποτελέσματα, πρέπει να προσπαθήσουμε να τα δικαιολογήσουμε. Αν βρούμε κάτι πολύ καλό, να το πούμε σε κάποιον φίλο μας για να το κοιτάξετε μαζί. Ο προγραμματισμός δεν είναι απαραίτητα κάτι μοναχικό. Που ξέρεις, μπορεί να ανακαλύψουμε κάτι καινούργιο εκεί που δεν το περιμένουμε!

Για αυτό τον λόγο κιόλας δεν αξίζει να αντιγράφουμε απλώς τον κώδικα από αυτό τον οδηγό, αλλά να τον πληκτρολογούμε μόνοι σας, προσπαθώντας να καταλάβετε γιατί υπάρχει κάθε τι στη θέση που είναι. Μάλιστα, στη διαδικασία πληκτρολόγησης μπορεί κάτι να φανεί τόσο βαρετό, που μόνο του να γίνει το κίνητρο ώστε να βρεθεί ένας πιο σύντομος και αποδοτικός τρόπος για να υλοποιηθεί κάτι. Αν τυχόν γίνει κάτι τέτοιο, δεν πρέπει να διστάσει κανείς να επικοινωνήσει με τον συγγραφέα του οδηγού! Και αυτός προσπαθεί να μάθει συνεχώς, μέσα από τα καλύτερα παραδείγματα!

1.2.4 Δώστε προσοχή στη λεπτομέρεια

Ένα χαρακτηριστικό που ξεχωρίζει τους κακούς προγραμματιστές από τους καλούς είναι η προσοχή τους στη λεπτομέρεια. Χωρίς την απαραίτητη προσοχή σε πράγματα που ίσως να φαίνονται λεπτομέρειες, εμφανίζονται στη συνέχεια προβλήματα στο κώδικα και δυσκολία ανάπτυξης της εφαρμογής που επιθυμούμε. Επίσης πολύ σημαντικό είναι να μπορεί κάποιος να ξεχωρίζει οπτικά τι κάνουν διαφορετικά πράγματα έστω και αν διαφέρουν ελαφρώς. Πολύ συχνά αυτό αποτελεί και το κλειδί στο πως μπορεί να γίνει κάτι. Αρχικά πρέπει να δούμε πως διαφοροποιούνται οι αιτίες του από κάτι που θα προκαλούσε μια ανεπιθύμητη συμπεριφορά και στη συνέχεια εκμεταλλευόμενοι την ύπαρξη αυτών να οδηγήσουμε το πρόγραμμα μας να παράγει το επιθυμητό αποτέλεσμα.

1.2.5 Μάθετε μόνοι σας

Ίσως αυτή η ενότητα να μην είναι ό,τι καλύτερο για την εισαγωγή ενός βιβλίου, αλλά περιέχει μια μεγάλη αλήθεια. Μην τα περιμένουμε όλα έτοιμα. Το διαδίκτυο είναι πια αχανές. Μπορείτε να βρεθεί πάρα πολύ υλικό σε αυτό. Οφείλουμε να ψάξουμε τα βιβλία ή τους οδηγούς που περιέχουν αυτά που χρειαζόμαστε, να διαβάσουμε τα εγχειρίδια της Python και να ρωτήσουμε άλλους ανθρώπους που πιθανώς να γνωρίζουν αυτό που ψάχνουμε. Η τεχνολογία αλλάζει πολύ γρήγορα. Ας μη περιμένουμε, λοιπόν, πρώτα άλλοι να μάθουν τις καινούργιες εξελίξεις για να τις μάθουμε ύστερα εμείς. Ας πάρουμε την τύχη στα χέρια μας.

1.2.6 Δείξτε και σε άλλους

Πολλοί δάσκαλοι λένε ότι μαθαίνεις κάτι πραγματικά όταν το διδάσκεις. Αυτή είναι μια πρόταση που περιέχει μια μεγάλη δόση αλήθειας. Όταν προσπαθείς να μάθεις κάτι σε κάποιον, αυτός θα σου κάνει ερωτήσεις για πράγματα που δεν κατάλαβε. Αυτές οι ερωτήσεις θα βοηθήσουν και εμάς να μάθουμε και να ξεδιαλύνετε κάποια σημεία, όπως επίσης θα μπορούμε να κοιμηθούμε και με ένα μεγαλύτερο χαμόγελο το βράδυ έχοντας βοηθήσει κάποιον συνάνθρωπο μας.

1.2.7 Υπομονή

Είναι πολύ εύκολο να κατηγορήσουμε τον μεταφραστή (compiler) ότι ευθύνεται που δεν δουλεύει το πρόγραμμα μας. Όμως ας το σκεφθούμε ξανά. Δεν γίνεται να τα κάναμε όλα σωστά και να μην δουλεύει. Το πιο πιθανό είναι ότι αν είχε κάποιο σφάλμα ο μεταφραστής, τότε κάποιος θα το είχε υποδείξει ήδη και θα είχε διορθωθεί.

Επίσης, δεν ωφελεί να αναμένουμε ότι γράψουμε αμέσως προγράμματα που θα αλλάξουν την ροή της ιστορίας. Πρέπει να αρχίζουμε από μικρές εφαρμογές και όσο τελειοποιούμε τις γνώσεις μας, να προχωράμε σε πιο περίπλοκες. Αν δείξουμε την κατάλληλη υπομονή, θα δούμε ότι πολύ σύντομα θα μπορούμε να δημιουργήσουμε εφαρμογές που και εμάς τους ίδιους θα εκπλήσσουν με τις δυνατότητες τους. Και που ξέρεις, ίσως σύντομα να υλοποιηθεί μια εφαρμογή που θα αλλάξει τον κόσμο μας, όπως έχει δείξει πολλές φορές η σύντομη και πολύ πρόσφατη ιστορία των υπολογιστών και της τεχνολογίας γενικότερα.

1.2.8 Επαναχρησιμοποιήστε κώδικα

Δεν είναι ανάγκη να ανακαλύπτουμε κάθε φορά τον τροχό από την αρχή. Στα πρώτα μας βήματα είναι καλό να δημιουργούμε μόνοι μας απλές συναρτήσεις ώστε να καταλαβαίνετε πως λειτουργούν. Αργότερα, όμως, ας μη διστάσουμε να χρησιμοποιήσουμε βιβλιοθήκες. Η Python είναι μια αρκετά ώριμη γλώσσα και θα βρούμε πολλές βιβλιοθήκες για αυτή. Αν αντιμετωπίζουμε ένα πρόβλημα αρκετά κοινό, είναι πολύ πιθανό να το έχει λύσει κάποιος άλλος για εμάς. Η λύση του θα είναι δοκιμασμένη και χρησιμοποιώντας της θα αποφύγουμε πιθανά σφάλματα στον κώδικα μας (bugs), αλλά και θα επιταχύνουμε την ανάπτυξη της εφαρμογής μας.

1.3 Τι είναι η Python

Η Python είναι μια διερμηνευόμενη, υψηλού επιπέδου γλώσσα με δυναμική σημασιολογία (semantics). Η φιλοσοφία της ενθαρρύνει την αναγνωσιμότητα του κώδικα και έχει μια αρκετά μεγάλη κύρια βιβλιοθήκη (standard library). Ανάμεσα στα κύρια χαρακτηριστικά της είναι:

- Εύκολη
  - ◇ Εκμάθηση
  - ◇ Αναγνωσιμότητα (πολύ καθαρό, αναγνώσιμο συντακτικό)
  - ◇ Συντήρηση
- Γρήγορη Ανάπτυξη Εφαρμογών
- Διερμηνευόμενη
- Πολύ υψηλού επιπέδου δομές δεδομένων
- Επεκτάσιμη
- Ανοικτού Κώδικα
- Παίζει σχεδόν παντού
- Ώριμη
- Όχι πια segmentation faults
- Αυτόματη διαχείριση μνήμης

Από την αρχή της ανάπτυξης της ενθαρρύνεται η ανάπτυξη των εφαρμογών μέσω της Python να είναι όσο πιο απλή γίνεται. Αυτό γίνεται και όσον αφορά την εκμάθηση της ίδιας της γλώσσας, όπου προσπαθείται να υπάρχει μια ομαλή καμπύλη εκμάθησης και όσον αφορά την αναγνωσιμότητα του παραγόμενου κώδικα. Απότοκος των παραπάνω είναι η ευκολία στην συντήρηση του κώδικα και την επέκταση του. Όπως χαρακτηριστικά έχει γραφτεί, για να κάνουμε αποσφαλμάτωση σε ένα κομμάτι κώδικα χρειαζόμαστε την διπλάσια ευφυΐα από όταν τον γράψαμε. Συνεπώς, αν γράφεις όσο πιο 'έξυπνο'-δύσκολο κώδικα μπορείς, εκ των πραγμάτων δεν μπορείς να τον αποσφαλματώσεις.

Όλα τα παραπάνω συνηγορούν στην δυνατότητα της Python να επιτρέπει την ταχύτατη ανάπτυξη εφαρμογών ειδικά σε σχέση με άλλες γλώσσες χαμηλότερου επιπέδου (πχ C, C++) ενώ λέγεται ότι συνήθως τα προγράμματα

σε Python είναι $3-5$ φορές μικρότερα σε σχέση με τα αντίστοιχα σε Java. Όσο πιο υψηλού επιπέδου μια γλώσσα προγραμματισμού είναι, τόσο πιο κοντά στην σκέψη του ανθρώπου βρίσκεται. Αυτό σημαίνει ότι είναι πιο εύκολο να γραφτούν προγράμματα σε υψηλού επιπέδου γλώσσες (υψηλό επίπεδο αφαίρεσης) και συνήθως λειτουργούν σε περισσότερες πλατφόρμες. Αυτό όμως γίνεται θυσιάζοντας μέρος της ταχύτητας των προς εκτέλεση προγραμμάτων. Στις μέρες μας παρατηρείται μια σταδιακή στροφή από γλώσσες που επικέντρωνόντουσαν στην απόδοση των προγραμμάτων (efficiency), να επικεντρώνουν στην απόδοση του προγραμματιστή (productivity).

Υπάρχουν δυο είδη προγραμμάτων που ασχολούνται με την μετατροπή του προγράμματος από γλώσσα προγραμματισμού που είναι κοντά στον άνθρωπο, σε γλώσσα μηχανής. Αυτά είναι οι *διερμηνείς* (interpreters) και οι *μεταφραστές* (compilers). Οι διερμηνείς μετατρέπουν γραμμή προς γραμμή τον πηγαίο κώδικα του προγράμματος μας σε γλώσσα μηχανής και τον εκτελούν άμεσα ενώ οι μεταφραστές πρέπει να μετατρέψουν όλο το πρόγραμμα σε γλώσσα μηχανής και στην συνέχεια αυτό μπορεί να εκτελεστεί. Υπάρχει η ειδική περίπτωση που χρησιμοποιείται από την γλώσσα προγραμματισμού εικονική μηχανή (virtual machine) όπου εκτελείται ο κώδικας (όπως συμβαίνει και με την Python). Πριν μετατραπεί σε γλώσσα μηχανής, που καταλαβαίνει τελικά ο υπολογιστής, μετατρέπεται σε μια ενδιάμεση γλώσσα (bytecode). Η τελική μορφή του κώδικα που έχει μεταφραστεί και μπορεί πλέον να εκτελεστεί ονομάζεται αντικειμενικός κώδικας (object code).

Η κύρια βιβλιοθήκη περιλαμβάνει τα πάντα από ασύγχρονη επεξεργασία έως συμπιεσμένα αρχεία. Επειδή ο κώδικας της έχει γραφτεί από πολλούς έξυπνους ανθρώπους, είναι πολύ γρήγορος για τις περισσότερες εφαρμογές που θα χρειαστεί κάποιος. Οι ευκολίες που παρέχει επίσης είναι πολύ σημαντικές καθώς καλύπτει ένα ευρύ φάσμα πιθανών προβλημάτων που μπορεί να αντιμετωπίσει κανείς, αποφεύγοντας έτσι την ανάγκη για κάποιον να προσπαθεί να ανακαλύψει από την αρχή τον τροχό.

Η ίδια η γλώσσα είναι επεκτάσιμη καθώς ένα βασικό σύνολο της γλώσσας αποτελεί τον πυρήνας της, ενώ όλα τα υπόλοιπα είναι αρθώματα (modules) που επεκτείνουν την λειτουργικότητα της, γεγονός που σε συνδυασμό με το ότι είναι ανοικτού κώδικα την βοηθάει να μην μένει στάσιμη, αλλά να παρακολουθεί πάντα τις εξελίξεις (παράδειγμα η έκδοση 3 της Python που

είναι και το αντικείμενο πραγμάτευσης του παρόντος οδηγού).

Το ότι παίζει σχεδόν παντού, δεν αναφέρεται μόνο σε λειτουργικά συστήματα όπου παίζει σε όλες τις κύριες πλατφόρμες (πχ Windows, Linux/Unix, OS/2, Mac, Amiga). Αναφέρεται ακόμα και σε άλλες γλώσσες προγραμματισμού, όπως η Java, όπου μέσω της Jython μπορούμε να χρησιμοποιήσουμε βιβλιοθήκες της Java, του .NET για το οποίο υπάρχει η πρόσφατη υλοποίηση της IronPython από την Microsoft. Ακόμα, μπορούμε να γράψουμε κώδικα σε C/C++ και στην συνέχεια να φτιάξουμε αρθώματα (modules) μέσω τον οποίων ο τελικός χρήστης του κώδικας μας δεν θα καταλαβαίνει καμία διαφορά σε σχέση με τον υπόλοιπο κώδικα Python.

Επιπρόσθετα, η γλώσσα είναι πια ώριμη. Υπάρχει από τα τέλη της δεκαετίας του 1980 και σε αυτή την πορεία του χρόνου πολλοί την έχουν υιοθετήσει και έχουν δημιουργηθεί πολλές βιβλιοθήκες για αυτή. Επίσης, έχει ξεπεράσει παιδικές ασθένειες και συμπεριλαμβάνει πολλά χαρακτηριστικά όπως αυτά προβλήθηκαν μέσα από τις ανάγκες των χρηστών της.

Τέλος, αν προέρχεστε από κάποια άλλη γλώσσα προγραμματισμού, με την Python ξεχάστε τα segmentation faults. Σε αντίστοιχες περιπτώσεις, ο διερμηνευτής της Python μας ενημερώνει με μια εξαίρεση που πετάει (γίνεται throw) και πλέον γνωρίζουμε σε ποια γραμμή υπάρχει το πρόβλημα ώστε να το αντιμετωπίσουμε.

Η αυτόματη διαχείριση μνήμης σημαίνει πως δεν χρειάζεται να ανησυχούμε πλέον για το πότε θα ελευθερώσουμε την μνήμη που δεσμεύουμε όταν φτιάχνουμε αντικείμενα. Επίσης, η Python αντιλαμβάνεται πότε το ίδιο αντικείμενο αναφέρεται πάνω από μια φορές και έτσι δεν το αποθηκεύει στη μνήμη αν δεν χρειάζεται. Η τεχνική αυτή ονομάζεται μέτρηση αναφορών (reference counting).

1.3.1 Όλα είναι αντικείμενα

Στην Python, τα πάντα είναι αντικείμενα. Ακόμα και οι συναρτήσεις, οι γεννήτορες (generators), οι εξαιρέσεις και ό,τι άλλο μπορείτε να σκεφθείτε. Έτσι, προσφέρεται ένας διαισθητικός τρόπος συγγραφής των προγραμμάτων μας με συνέπεια στην αντικειμενοστρέφεια, παρόλο που υποστηρίζονται και άλλοι τρόποι προγραμματισμού (όπως συναρτησιακός, διαδικαστικός κ.α.).

Στα βασικά χαρακτηριστικά της γλώσσας θα συναντήσουμε των χειρισμό εξαιρέσεων, πακέτα, κλάσεις, συναρτήσεις, γεννήτορες (ειδική περίπτωση συναρτήσεων) με ένα τρόπο όπου κάθε ένα θα συμπληρώνει αρμονικά το άλλο για την διευκόλυνση του προγραμματιστή. Αυτό το δέσιμο καθοδηγείται πάντα από την συνέπεια στην αρχή της Python ότι θα έπρεπε να υπάρχει ένας και μόνο ένας προφανής τρόπος για να γίνει κάτι, οδηγώντας έτσι στην απλοποίηση των προβλημάτων που μπορεί να αντιμετωπίσει ένα προγραμματιστής.

Τέλος, ένα βασικό χαρακτηριστικό της γλώσσας, που ίσως να ξενίσει κάποιους, αλλά αναβαθμίζει την αναγνωσιμότητα του κώδικα, είναι ότι κάθε μπλοκ κώδικα καθορίζεται από την στοίχιση του. Κατά αυτό τον τρόπο, κάποιος είναι υποχρεωμένος να τηρήσει του κανόνες 'καλής συμπεριφοράς' όπως αυτοί επιβάλλονται από τις υπόλοιπες γλώσσες προγραμματισμού καθώς θα πρέπει να ενσωματώσει στο τρόπο που γράφει τον κώδικα του μια συνέπεια στο καθορισμό της στοίχισης του. Για ακόμα καλύτερα αποτελέσματα έχουν γραφτεί ειδικές προτάσεις (Python Enhancement Proposal (PEP)) τα οποία διευκρινίζουν τα συγκεκριμένα θέματα και καθορίζουν έναν επίσημο τρόπο συγγραφής του κώδικα (όποιος ενδιαφέρεται μπορεί να ανατρέξει στα PEP 8 και PEP 257)

1.3.2 Python 3

Η Python όπως αναφέρθηκε και προηγουμένως, υπάρχει από τα τέλη της δεκαετίας του 1980. Αυτό σημαίνει πως εκτός από την εμπειρία και την ωρίμανση στο κώδικα που είχαν αποφέρει όλα αυτά τα χρόνια, υπήρχαν και βάρη από το παρελθόν που την εμπόδιζαν να προχωρήσει μπροστά όπως επιτάσσουν και οι αρχές πάνω στις οποίες είναι δημιουργημένη (δοκιμάστε στο περιβάλλον του διερμηνευτή[2] το import this) και ταυτόχρονα να καλύπτει τις σύγχρονες ανάγκες. Παραδείγματα περίεργων συμπεριφορών είναι:

- 7 / 4 = 1 επηρεασμένη από την αριθμητική ακεραίων σε C.
- range και γεννήτορες (άσκοπη χρήση μνήμης)

[2]Για να εκτελέσουμε τον διερμηνευτή σε Linux πληκτρολογούμε python3 αφού πρώτα έχουμε εγκαταστήσει την γλώσσα, ενώ σε Windows ανοίγουμε το IDLE.

- Εκτύπωση print statement όπου αν χρειαζόταν κάτι παραπάνω από την τυπική συμπεριφορά γινόταν σχετικά περίπλοκο.
- Οργάνωση κύριας βιβλιοθήκης η οποία πλέον ξέφευγε από τον κύριο τρόπο ονοματολογίας και περιείχε συναρτήσεις που προτεινόταν να μην χρησιμοποιούνται πια.
- Η υποστήριξη Unicode ήθελε κάποια (μικρή έστω) προσπάθεια.
- ...και άλλα.

Για να ξεπεραστούν τα προβλήματα που αναφέρονται παραπάνω, έπρεπε να παρθεί μια τολμηρή απόφαση. Το μεγάλο βήμα λοιπόν για την Python έγινε με την έκδοση 3 (αλλιώς py3k ή 3000). Το μεγαλύτερο μέρος της γλώσσας είναι σχεδόν ίδιο, αλλά πλέον έμειναν μια και καλή στο παρελθόν ιδιάζουσες συμπεριφορές ενώ εισήχθηκαν καλύτεροι τρόποι επίλυσης ορισμένων προβλημάτων.

Οι αλλαγές που έγιναν περιλαμβάνουν:

- 7 / 4 = 1.75 (και ο ειδικός τελεστής // πλέον αφορά αριθμητική ακεραίων)
- Εκτεταμένη χρήση γεννητορων. Προτιμήθηκε η ευρεία χρήση γεννητορων στην βασική βιβλιοθήκη ώστε για παράδειγμα η range να παράγει του αριθμούς που χρειάζονται μόνο όταν χρειάζονται[3] και να μην γεμίζει η μνήμη άσκοπα.
- Η print έγινε συνάρτηση
- Η βιβλιοθήκες αναδιοργανώθηκαν
- Εύκολη υποστήριξη Unicode, χωρίς να απαιτείται σχεδόν καθόλου προσπάθεια.
- ...και διάφορες άλλες αλλαγές

Στο συγκεκριμένο οδηγό επικεντρωνόμαστε σε αυτή την τελευταία έκδοση της Python.

[3] Βλέπε ενότητα 11.2.

1.3.3 Μετάβαση

Υπάρχουν τρία βασικά εργαλεία για να συνεπικουρούν την μετάβαση των υφιστάμενων προγραμμάτων από την έκδοση 2 της γλώσσας στην τελευταία έκδοση:

- 2to3
- 3to2
- python -3

Τα δύο πρώτα εργαλεία μετατρέπουν τον κώδικα από την έκδοση 2 στην έκδοση 3 και αντίστροφα, αντίστοιχα. Το τρίτο μπορούμε να το χρησιμοποιήσουμε με τις τελευταίες εκδόσεις της Python (2.6+) και μας πληροφορεί για την χρήση χαρακτηριστικών που δεν υπάρχουν στην έκδοση 3 της Python.

Αν κάποιον τον ανησυχούν όλες αυτές οι αλλαγές που γίνονται στην γλώσσα, πρόσφατα και με το PEP 3003 (Python Enhancement Proposal) προτάθηκε το πάγωμα των αλλαγών στο συντακτικό της γλώσσας για 2 χρόνια ώστε να προλάβουν άλλες υλοποιήσεις της Python πέρα από την βασική (CPython) (όπως πχ Jython, Pypy, IronPython) να μπορέσουν να φθάσουν την βασική υλοποίηση σε χαρακτηριστικά.

1.3.4 Χρήσεις

Αν κάποιος δεν έχει πιστεί ακόμα για την Python, αξίζει να αναφέρουμε κάποιους από τους χρήστες της:

- Google (παράδειγμα το Google App Engine)
- NASA
- Yahoo!
- Μεγάλα πανεπιστήμια (MIT, Stanford κτλ).
- Σχεδόν όλες οι διανομές linux.
- ...και πολλοί άλλοι!

Αν αναρωτιέστε τους λόγους για τους οποίους μπορεί όλοι αυτοί να χρησιμοποιούν την Python, μπορείτε να διαλέξετε ανάμεσα σε:

1. Γρήγορη προτυποποίηση (prototyping)
2. Προγραμματισμός στον Παγκόσμιο Ιστό
3. Scripting
4. Εκπαίδευση
5. Επιστήμη
6. Εφαρμογές με γραφική διεπαφή
7. ...και πολλές άλλες!

Με λίγα λόγια, το κύριο πλεονέκτημα της είναι ότι κάποιος επικεντρώνεται σε αυτό που θέλει να γράψει και όχι στις ιδιαιτερότητες της γλώσσας. Έμπειροι προγραμματιστές είναι σε θέση να μάθουν πολύ γρήγορα την Python και να είναι άμεσα παραγωγικοί.

1.3.5 Zen of Python

Το Zen of Python συνιστά ορισμένες βασικές αρχές της Python και είναι γραμμένο από τον Tim Peters. Στην πρωτότυπη έκδοση του στα αγγλικά μπορείτε να το βρείτε γράφοντας σε περιβάλλον διερμηνευτή:

```
import this
```

Όμορφο είναι καλύτερο από άσχημο.
Άμεσο είναι καλύτερο από έμμεσο.
Απλό είναι καλύτερο από σύνθετο.
Σύνθετο είναι καλύτερο από περίπλοκο.
Επίπεδο είναι καλύτερο από εμφωλευμένο.
Αραιό είναι καλύτερο από πυκνό.
Η αναγνωσιμότητα μετράει.
Οι ειδικές περιπτώσεις δεν είναι αρκετά ειδικές ώστε να σπάνε τους κανόνες.

Ωστόσο η πρακτικότητα υπερτερεί της αγνότητας.

Τα λάθη δεν θα πρέπει ποτέ να αποσιωπούνται.

Εκτός αν αποσιωπούνται ρητά.

Όταν αντιμετωπίζεις την αμφιβολία, αρνήσου τον πειρασμό να μαντέψεις.

Θα πρέπει να υπάρχει ένας– και προτιμητέα μόνο ένας –προφανής τρόπος να το κάνεις.

Αν και αυτός ο τρόπος μπορείς να μην είναι προφανής εκτός αν είσαι Ολλανδός.

Τώρα είναι καλύτερα από ποτέ.

Αν και ποτέ είναι συχνά καλύτερα από ακριβώς τώρα.

Αν η υλοποίηση είναι δύσκολο να εξηγηθεί, τότε είναι κακή ιδέα.

Αν η υλοποίηση είναι εύκολο να εξηγηθεί, τότε ίσως είναι καλή ιδέα.

Τα ονόματα χώρου (namespaces) είναι μια φοβερά καλή ιδέα – ας κάνουμε περισσότερα από αυτά!

1.4 Χρήση της Python

1.4.1 Εγκατάσταση

Η Python μπορεί να εγκατασταθεί σε όλες τις κύριες πλατφόρμες υπολογιστών που χρησιμοποιούμε. Η χρήση της είναι δυνατή ακόμα και σε κινητά, αν και αυτό ξεφεύγει από τους σκοπούς αυτού του οδηγού (τουλάχιστον προς το παρόν). Γενικές οδηγίες μπορούμε να βρούμε στον επίσημο ιστότοπο της.

Linux

Σε Linux προτείνεται η χρήση του package manager της διανομής που χρησιμοποιούμε, αν δεν είναι ήδη εγκατεστημένη (το πιο πιθανό). Συνήθως, κάνοντας την εγκατάσταση κατά αυτό τον τρόπο, είναι πιο εύκολη η αναβάθμιση της, η χρήση επιπρόσθετων βιβλιοθηκών καθώς και η υποστήριξη μας από την κοινότητα της διανομής που χρησιμοποιούμε. Προσοχή μόνο, ενδιαφερόμαστε για την τελευταία έκδοση της Python 3. Καλό είναι να βεβαιωθούμε πως έχουμε αυτή εγκατεστημένη.

Ωινδοως

Ως συνήθως, θα πρέπει να κατεβάσουμε το αντίστοιχο εκτελέσιμο από την επίσημη ιστοσελίδα. Αφού το εγκαταστήσουμε, συμφωνόντας στους όρους, μας γίνεται διαθέσιμο και ένα περιβάλλον συγγραφής κώδικας (το IDLE).

Μας

Σε Mac, η Python έρχεται ήδη εγκατεστημένη. Ωστόσο, λόγω του ότι καινούργιες εκδόσεις του λειτουργικού βγαίνουν κάθε περίπου δύο χρόνια, προτείνεται να αναβαθμίσουμε την έκδοση που έχουμε. Αυτό γίνεται κατεβάζοντας την τελευταία έκδοση από τον ιστότοπο. Θα την βρούμε εγκατεστημένη, πηγαίνοντας στο Applications / Utilities / Terminal και γράφοντας Python. Προσοχή, και εδώ, θα χρειαστούμε την έκδοση 3 για τον υπόλοιπο οδηγό.

1.4.2 Πως εκτελούμε προγράμματα Python

Τα αρχεία πηγαίου κώδικα σε Python έχουν την κατάληξη .py. Στις επόμενες υποενότητες φαίνεται πως μπορούμε να εκτελούμε τις δηλώσεις που περιγράφονται σε αυτά. Τα περισσότερα αποσπάσματα κώδικα που θα δούμε τα γράφουμε σε ένα αρχείο .py και στην συνέχεια τα εκτελούμε από εκεί. Όπου όμως δείτε κώδικα που κάποιες γραμμές μοιάζουν όπως παρακάτω:

```
>>> a = 5
>>> b = 6
>>> a + b
11
```

Θα υπονοείται ότι εκτελείται στο περιβάλλον του διερμηνευτή (interpreter), ο οποίος καλείται γράφοντας σε περιβάλλον γραμμής εντολών (κονσόλα σε Linux, Έναρξη > Εκτέλεση > cmd σε Windows) python (ή python3 ανάλογα που βρίσκεται το αντίστοιχο εκτελέσιμο). Όποιες γραμμές αρχίζουν με τρία >>> περιέχουν δηλώσεις που πληκτρολογούμε εμείς, ενώ οι υπόλοιπες αφορούν το αποτέλεσμα που εμφανίζεται αντίστοιχα στην οθόνη μας.

Windows

Υπάρχουν δύο βασικοί τρόποι με τους οποίους μπορούμε να εκτελούμε εύκολα τα προγράμματα που ετοιμάζουμε σε python.

1. Ελέγχουμε ότι η Python βρίσκεται στο PATH. Για να το κάνουμε αυτό, πηγαίνουμε Πίνακας Ελέγχου -> Σύστημα -> Για προχωρημένους -> Μεταβλητές Περιβάλλοντος (Control Panel -> System -> Advanced -> Environment Variables) και θέτουμε την διαδρομή (path) όπου εγκαταστάθηκε η Python. Στην συνέχεια μπορούμε να τρέχουμε τα προγράμματα μας από γραμμή εντολών (command line) μέσω python onoma.py.

2. Ανοίγουμε το IDLE (το πρόγραμμα που εγκαθίσταται μαζί με την python στον υπολογιστή μας). Φορτώνουμε το αρχείο που έχουμε γράψει (File > Open). Μόλις φορτωθεί το πρόγραμμα, πηγαίνουμε Run Module > Run και βλέπουμε το πρόγραμμα μας να εκτελείται.

Προτείνεται για την επεξεργασία του πηγαίου κώδικα η αποφυγή χρήσης του Notepad ή του Wordpad. Αντίθετα, μια καλή επιλογή θα ήταν το Notepad++ το οποίο είναι ελεύθερο και ανοικτού κώδικα.

Linux

1. Ανοίγουμε κονσόλα και πηγαίνουμε στο path όπου βρίσκεται το αρχείο που θέλουμε να εκτελέσουμε. Αν πατήσουμε python3 onoma.py αυτό θα εκτελεστεί[4].

2. Με διπλό κλικ πάνω στο αρχείο που γράφουμε τον κώδικα, εφόσον αυτό είναι εκτελέσιμο και αρχίζει με

```
#!/usr/bin/env python3
```

Ίσως συχνά να δείτε ότι χρησιμοποιείται και το

[4] Εφόσον έχει εγκατασταθεί η python 3 και ονομάζεται το εκτελέσιμο της python3 το οποίο συνήθως βρίσκεται στο /usr/bin

```
#!/usr/bin/python3
```

ως εναλλακτική επιλογή που προστίθεται στην αρχή του αρχείου του πηγαίου κώδικα. Η διαφορά που έχει με το προτεινόμενο #/usr/bin/env python3 είναι ότι αυτό δεν προϋποθέτει την ύπαρξη του εκτελέσιμου της Python 3 σε συγκεκριμένο μονοπάτι (το /usr/bin/python3) αλλά βασίζεται στο περιβάλλον για να βρει που αυτό βρίσκεται. Έτσι, η εφαρμογή μας είναι πιο εύκολα να μεταφερθεί σε άλλο σύστημα. Πρακτικά, ωστόσο, σπάνια θα δείτε κάποια διαφορά ανάμεσα σε αυτά τα δυο.

Ρυθμίσεις επεξεργαστή κειμένου

Σύμφωνα και με το PEP[5] 8 που αποτελεί και το κύριο έγγραφο που προσδιορίζει ορισμένους κανόνες για την συγγραφή των προγραμμάτων μας σε Python, προτείνεται κάθε tab να αντικαθίσταται από τέσσερα κενά, τα οποία και αποτελούν και τον τρόπο στοιχειοθέτησης (indent) του κώδικα μας.[6]

Εάν εκτελώντας το κώδικα σας βρείτε το σφάλμα "Unknown option: -", το αρχείο του κώδικα σας μπορεί να έχει λάθος κωδικοποίηση τέλους γραμμής. Προτείνεται η χρήση Unix τέλους γραμμής. Παρακάτω ακολουθούν ορισμένες προτεινόμενες ρυθμίσεις σε επεξεργαστές κειμένου που είναι αρκετά δημοφιλείς.

1. *Notepad++*
    - Tabs: Settings > Preferences > Edit Components > Tab settings.
    - Settings > Preferences > MISC > auto-indent.
    - Τέλος γραμμής: Format > Convert, ρυθμίστε σε Unix.
2. *JEdit*
    - Τέλος γραμμής: Από τα 'U' 'W' 'M' στην γραμμή κατάστασης, επιλέξτε το 'U'

[5]PEP: Python Enhancement Proposal. Αφορούν προτάσεις για βελτίωση ορισμένων χαρακτηριστικών της γλώσσας, σχεδιαστικές αποφάσεις, καλές πρακτικές συγγραφής κώδικα κ.α.

[6]Αν κάποιος ενδιαφέρεται περισσότερο για τα PEPs μπορεί να τα βρει στην διεύθυνση http://python.org/dev/peps/.

3. *Mac TextWrangler*

   - Tabs: Από το κουμπί προτίμησης στο πάνω μέρος του παραθύρου επιλέξτε το Auto Expand Tabs. Μπορείτε να αλλάξετε τις προεπιλογές από το Defaults > Auto-Expand Tabs και Defaults > Auto-indent.
   - Τέλος γραμμής: Στο κάτω μέρος του κάθε παραθύρου, τοποθετήστε το σε Unix.

Προτείνουμε αρχικά να χρησιμοποιηθεί κάποιος απλός επεξεργαστής κειμένου. Αν έχετε Windows, μπορείτε αρχικά να μείνετε με τον IDLE (αν σας βολεύει) ή αν θέλετε ένα πλήρες περιβάλλον, προτείνεται το Eclipse με το pydev. Σε Linux θα πρότεινα τον kate ή gedit, ή για πιο προχωρημένο πάλι τον Eclipse. Ο επεξεργαστής κειμένου απλά μας παρέχει μια ευκολία στο να γράφουμε τον κώδικα μας (πχ χρώματα στις λέξεις κλειδιά). Πάντα όμως αυτό που μετράει στην τελική, είναι ο κώδικας που παράγεται και αν κάνει την δουλειά που θέλουμε.

1.5 Οργάνωση Οδηγού

Ο συγκεκριμένος οδηγός οργανώνεται ως εξής:

Στο Κεφάλαιο 1 βρίσκουμε μια μικρή εισαγωγή για το πως μπορούμε να μάθουμε προγραμματισμό αλλά και πιο συγκεκριμένα για την Python.

Στο Κεφάλαιο 2 βλέπουμε ορισμένες από τις βασικές έννοιες που κυριαρχούν στις περισσότερες γλώσσες προγραμματισμού. Συναντάμε τις έννοιες της μεταβλητής, της τιμής καθώς και εξετάζουμε ορισμένους τελεστές και πως μπορούν αυτοί να εφαρμοστούν.

Στο Κεφάλαιο 3 βρίσκουμε ορισμένες έννοιες για την ροή ελέγχου, οι οποίες είναι και απαραίτητες για τη δημιουργία πιο περίπλοκων προγραμμάτων.

Στο Κεφάλαιο 4 βλέπουμε ορισμένες βασικές αριθμητικές λειτουργίες καθώς και μαθαίνουμε για το πως αποθηκεύονται οι αριθμοί σε ένα πρόγραμμα.

Στο Κεφάλαιο 5 έχουμε μια σύντομη παρουσίαση των δυνατοτήτων που μας προσφέρουν οι συναρτήσεις όσον αφορά την επαναχρησιμοποίηση κώδικα αλλά και την λογική του ομαδοποίηση.

Στο Κεφάλαιο 6 γνωρίζουμε τον κινητήριο μοχλό των προγραμμάτων μας: ορισμένες από τις πιο βασικές δομές δεδομένων που απαντώνται σχεδόν σε κάθε πρόγραμμα.

Στο Κεφάλαιο 7 εισερχόμαστε στα ενδότερα της Python για να καταλάβουμε πως λειτουργούν οι αναφορές στα αντικείμενα ώστε να μπορέσουμε να 'δαμάσουμε' τη δύναμη που αυτές μας προσφέρουν.

Στο Κεφάλαιο 8 συναντούμε το βασικό συστατικό του αντικειμενοστραφούς μοντέλου ανάπτυξης προγραμμάτων: τις κλάσεις. Γνωρίζουμε τα βασικά τους χαρακτηριστικά και βλέπουμε παραδείγματα χρήσεως τους.

Στο Κεφάλαιο 9 μαθαίνουμε για τον χειρισμό των αρχείων και των φακέλων με τρόπο ανεξάρτητο του λειτουργικού συστήματος. Επίσης, μαθαίνουμε πως μπορούμε να αποθηκεύουμε τη κατάσταση του προγράμματος μας και να την επαναφέρουμε για μελλοντική χρήση.

Στο Κεφάλαιο 10 γνωρίζουμε πως μπορούμε να αντιμετωπίζουμε και να ανακάμπτουμε από καταστάσεις όπου μπορεί να προκληθεί κάποιο σφάλμα.

Στο Κεφάλαιο 11 αναφερόμαστε σε ένα πιο προχωρημένο χαρακτηριστικό της Python, τους γεννήτορες και τις δυνατότητες που μας δίνουν αυτοί.

Στο Κεφάλαιο 13 είναι οι περιγραφείς οι οποίοι μπορούν να συνδυαστούν άρτια με την ιδέα των κλάσεων.

Στο Κεφάλαιο 14 βλέπουμε ορισμένα παρά πολύ απλά γραφικά και τον τρόπο με τον οποίο μπορούμε να τα δημιουργήσουμε μέσω της βιβλιοθήκης γραφικών (tkinter) που έρχεται μαζί με την προεπιλεγμένη εγκατάσταση της Python.

Στο Κεφάλαιο 15 χρησιμοποιούμε τη βασική βιβλιοθήκη αποσφαλμάτωσης η οποία περιέχεται μέσα στη κύρια βιβλιοθήκη της γλώσσας.

Τέλος, το Κεφάλαιο 16 αφορά τρόπους με τους οποίους μπορούμε να μετρήσουμε τον χρόνο εκτέλεσης ενός προγράμματος (ή μέρους ενός προγράμματος).

Κεφάλαιο 2

Μεταβλητές και Βασικοί Τελεστές

> Εμπρός, ψυχή μου, μονολόγησα, μη σκέφτεσαι πια τη Γνώση: ζήτα βοήθεια από την Επιστήμη.
>
> Ουμπέρτο Έκο, 'Το Εκκρεμές του Φουκώ'

Στο κεφάλαιο αυτό παραθέτουμε ορισμένες βασικές έννοιες που απαντώνται σχεδόν σε κάθε γλώσσα προγραμματισμού. Ειδικότερα, θα δώσουμε τους βασικούς ορισμούς για το τι είναι οι μεταβλητές και τύποι καθώς και θα δούμε τους βασικούς τελεστές που χρησιμοποιούμε για την επεξεργασία τους.

Όταν μαθαίνουμε μια γλώσσα προγραμματισμού δεν γινόμαστε απλά οικείοι με το συντακτικό της και ίσως με τις ιδιοτροπίες που μπορεί αυτή να έχει. Μαθαίνουμε και έναν καινούργιο τρόπο σκέψης για να εκφραζόμαστε μέσω αυτής της γλώσσας, αλλιώς τότε είτε η γλώσσα έχει πολύ λίγα να μας προσφέρει, είτε εμείς δεν την έχουμε κατανοήσει σε βάθος. Επομένως, πρέπει να δίνουμε ιδιαίτερη προσοχή στο τρόπο με τον οποίο η συγκεκριμένη γλώσσα μας επιτρέπει να συνδυάζουμε τις απλούστερες ιδέες ώστε να σχηματίζουμε σιγά σιγά πιο σύνθετες μέχρι να λύσουμε το συγκεκριμένο πρόβλημα που επιθυμούμε. Οι κύριοι μηχανισμοί μιας γλώσσας είναι, σύμφωνα και με το Structure and Interpretation of Computer Programs είναι:

1. Πρωταρχικές εκφράσεις, οι οποίες αναπαριστούν τις απλούστερες οντότητες με τις οποίες ασχολείται η γλώσσα.

2. Τους τρόπους συνδυασμούς τους μέσω τον οποίο σύνθετα στοιχεία δημιουργούνται από απλούστερα.

3. Τους τρόπους αφαίρεσης μέσω σύνθετα αντικείμενα μπορούν να ονομαστούν και να χειριστούν ως μονάδες.

Στην παρούσα ενότητα θα ασχοληθούμε κυρίως με τις πρωταρχικές ενώ τους τρόπους συνδυασμού και αφαίρεσης θα τους δούμε σε επόμενες ενότητες.

2.1 Τύποι και τιμές

Πριν προχωρήσουμε παρακάτω, καλό είναι να ξεκαθαρίσουμε κάποιες έννοιες. Αφού τις ορίσουμε με ακρίβεια, θα δούμε πως η Python τις χειρίζεται.

Ορισμός 2.1.1. Τιμή είναι μια ακολουθία από bit η οποία ερμηνεύεται σύμφωνα με κάποιον τύπο δεδομένων. Είναι δυνατόν η ίδια ακολουθία από bits να έχει διαφορετική ερμηνεία ανάλογα με τον τύπο δεδομένων βάση του οποίου ερμηνεύεται.

Ορισμός 2.1.2. Τύπος δεδομένων είναι ένα σύνολο τιμών και οι λειτουργίες πάνω σε αυτές.

Ένα παράδειγμα όπου τυπώνουμε κάποιες τιμές ακολουθεί παρακάτω. Η συνάρτηση print τυπώνει σε κάθε περίπτωση την τιμή που ακολουθεί. Αν περιέχει μεταβλητή, τότε τυπώνει τα περιεχόμενα τις μεταβλητής που αποτελούν μια τιμή. Οι τιμές όπως είπαμε πιο πάνω, ερμηνεύονται με βάση κάποιον τύπο δεδομένων. Διαφορετικός τύπος δεδομένων, διαφοροποιεί, στην γενική περίπτωση, και την τιμή που αναπαρίσταται.

```
print(1)
print('asdf')
primeNumbers = 1, 2, 3, 5, 7
print(primeNumbers)
```

Στην Python δεν δηλώνουμε ρητά τι τύπος δεδομένων χρησιμοποιείται. Όπως θα λέγαμε σε έναν άνθρωπο έναν αριθμό για παράδειγμα, και θα

καταλάβαινε τι εννοούμε, έτσι γίνεται και εδώ. Αυτή η ιδιότητα ονομάζεται δυναμικός τύπος dynamic typing. Έτσι για παράδειγμα, αν κάνουμε πράξεις με αριθμητικούς τύπους η Python ορίζει αυτόματα την απαιτούμενη ακρίβεια ώστε να γίνουν σωστά οι πράξεις. Αυτό κάνει τον κώδικα μικρότερο και πιο ευέλικτο. Ένα παράδειγμα όπου μπορούμε να δούμε είναι:

```
a = 'asdf'
a = 2
a = pow(a, 100000)
print(a)
```

όπου βλέπουμε πως η μεταβλητή `a` δείχνει σε διαφορετικού τύπου αντικείμενα χωρίς να το δηλώνουμε και επίσης πως όταν κάνουμε αριθμητικές πράξεις ακεραίων χρησιμοποιείται η απαιτούμενη ακρίβεια για αυτούς.

Ορισμός 2.1.3. Μια γλώσσα χρησιμοποιεί *δυναμικούς τύπους* (dynamically typed), όταν η πλειοψηφία των ελέγχων των τύπων της γίνεται κατά τον χρόνο εκτέλεσης του προγράμματος αντί για την ώρα μεταγλώττισης. Με πιο απλά λόγια, ελέγχεται αν κάτι είναι ακέραιος ή αλφαριθμητικό για παράδειγμα όταν πλέον η μεταβλητή που περιέχει το αντίστοιχο αντικείμενο έχει πάρει συγκεκριμένη τιμή. Έτσι, οι μεταφραστές προγραμμάτων δυναμικών γλωσσών όπως της Python κάνουν λιγότερους ελέγχους κατά την μετάφραση του κώδικα οι οποίοι γίνονται κυρίως κατά την εκτέλεση του.

2.2 Μεταβλητές

Ορισμός 2.2.1. Μεταβλητές είναι ένας τρόπος για να αποθηκεύουμε δεδομένα. Η τρέχουσα τιμή είναι τα δεδομένα που είναι αποθηκευμένα στην μεταβλητή. Επειδή αυτή η τιμή μπορεί να αλλάξει, τις καλούμε μεταβλητές.

```
num = 17
pi = 3.14
paei = 'Allou'
paei = 'Molis gyrise'
```

Τις μεταβλητές μπορούμε να τις βλέπουμε σαν μεγάλα συρτάρια ενός γραφείου. Εκεί, μπορούμε να αποθηκεύσουμε διάφορα αντικείμενα, και κάθε μεταβλητή περιέχει αυτό το αντικείμενο. Όταν ανοίγουμε ένα συρτάρι, ουσιαστικά προσπελαύνουμε τα περιεχόμενα του. Έτσι, και με τις μεταβλητές, αποτελούν ένα τρόπο προσπέλασης των αντικειμένων τα οποία περιέχουν.

Για να χρησιμοποιήσουμε μια μεταβλητή, χρειαζόμαστε απλά να γνωρίζουμε το όνομα της μεταβλητής. Έτσι για παράδειγμα το ακόλουθο πρόγραμμα υψώνει την μεταβλητή pi στο τετράγωνο αφού την έχουμε αρχικοποιήσει με την τιμή 3.14 και στην συνέχεια τυπώνει το αποτέλεσμά της.

```
pi = 3.14
print(pi**2)
```

2.3 Εκφράσεις Boolean

Ορισμός 2.3.1. Οι εκφράσεις Boolean είναι εκφράσεις που η αποτίμηση τους είναι είτε αληθής είτε ψευδής.

- 5 == 5 (Αληθές)
- 5 == 6 (Ψευδές)
- True and False (Ψευδές)
- True or False (Αληθές)

Από μικρότερες εκφράσεις Boolean μπορούμε να δημιουργήσουμε μεγαλύτερες χρησιμοποιώντας τους λογικούς τελεστές. Συνήθως όμως επιθυμούμε από τις μεγαλύτερες εκφράσεις να δημιουργούμε τις μικρότερες ώστε να γίνεται πιο απλό αυτό που περιγράφουμε. Οι βασικοί τελεστές για εκφράσεις Boolean είναι:

- Άρνηση not
- Διάζευξη or
- Σύζευξη and

Παρακάτω ακολουθούν οι πίνακες αλήθειας που περιγράφουν πλήρως τις παραπάνω πράξεις.

Είσοδος	Έξοδος
Αληθής	Ψευδής
Ψευδής	Αληθής

Πίνακας 2.1: Πίνακας Αλήθειας Λογικής Πράξης not

Είσοδος a	Είσοδος b	Έξοδος
Ψευδής	Ψευδής	Ψευδής
Ψευδής	Αληθής	Ψευδής
Αληθής	Ψευδής	Ψευδής
Αληθής	Αληθής	Αληθής

Πίνακας 2.2: Πίνακας Αλήθειας Λογικής Πράξης and

Είσοδος a	Είσοδος b	Έξοδος
Ψευδής	Ψευδής	Ψευδής
Ψευδής	Αληθής	Αληθής
Αληθής	Ψευδής	Αληθής
Αληθής	Αληθής	Αληθής

Πίνακας 2.3: Πίνακας Αλήθειας Λογικής Πράξης or

Επομένως μπορούμε να συνδυάσουμε με αυτό τον τρόπο συνθήκες που μπορεί να θέλουμε να ικανοποιούνται ταυτόχρονα (και οι δύο αληθείς άρα χρησιμοποιούμε την and), να ικανοποιείται τουλάχιστον η μια (χρησιμοποιούμε την or) ή και καμία (χρησιμοποιούμε την not δυο φορές μπροστά και από τις δύο συνθήκες και την and)[1]. Ακολουθούν ορισμένα παραδείγματα.

[1]Σύμφωνα με τον κανόνα De Morgan μπορούμε να χρησιμοποιήσουμε και μια άρνηση στη σύζευξη των δύο όρων.

```
>>> a = 5
>>> b = 4
>>> a > b
True
>>> c = 6
>>> a > b and a > c
False
>>> a > b or a > c
True
>>> a > b and not a > c
True
>>> a > b or not a > c
True
>>> not a > b or not a > c
True
>>> not (a > b) or a > c
False
>>> not a > b or a > c
False
```

2.4 Τελεστές

Εκτός από τους τελεστές για τις εκφράσεις Boolean (δηλαδή τους λογικούς τελεστές) υπάρχουν και άλλοι που χρησιμοποιούμε σε διαφορετικές περιστάσεις για την διευκόλυνση μας.

Ορισμός 2.4.1. Τελεστές είναι ειδικά σύμβολα που αναπαριστούν υπολογισμούς όπως η πρόσθεση και ο πολλαπλασιασμός. Οι τιμές στις οποίες εφαρμόζονται οι τελεστές ονομάζονται τελούμενα.

Με άλλα λόγια, οι τελεστές επιτελούν μια λειτουργία όπως οι συναρτήσεις που θα δούμε στην ενότητα 5. Παραδείγματα αριθμητικών τελεστών με τους οποίους είμαστε ήδη εξοικειωμένοι αποτελούν:

- Πρόσθεση ($+$)
- Αφαίρεση ($-$)
- Πολλαπλασιασμός ($*$)
- Διαίρεση ($/$)
- Ύψωση σε δύναμη ($**$)
- Υπόλοιπο (modulo) ($\%$)

Παρακάτω ακολουθούν οι τελεστές συγκρίσης. Εύκολα μπορούμε να μαντέψουμε τη λειτουργία τους σε αριθμητικά δεδομένα. Η Python μας επιτρέπει να αντιστοιχίσουμε σε αυτούς ειδικές λειτουργίες και για αντικείμενα δικών μας κλάσεων.

- Ίσο με ($==$)
- Διάφορο από ($!=$)
- Μεγαλύτερο από ($>$)
- Μικρότερο από ($<$)
- Μεγαλύτερο ή ίσο με ($>=$)
- Μικρότερο ή ίσο με ($<=$)

Κεφάλαιο 3

Έλεγχος Ροής Εκτέλεσης

> It is not enough to do your best; you must know what to do, and then do your best.
>
> W. Edwards Deming

3.1 Εισαγωγή

Στην ενότητα αυτή θα μελετήσουμε την ροή των προγραμμάτων και της κατηγορίες που χωρίζεται. Πρώτα όμως πρέπει να δούμε τι είναι ένα πρόγραμμα.

Ορισμός 3.1.1. Πρόγραμμα είναι μια ακολουθία εντολών προς έναν υπολογιστή που προσδιορίζει μια λειτουργία (ή ένα σύνολο λειτουργιών). Το πρόγραμμα έχει μια εκτελέσιμη μορφή που μπορεί να χρησιμοποιήσει απευθείας ο υπολογιστής. Τα βασικά χαρακτηριστικά ενός προγράμματος είναι:

- *Είσοδος:* Τα δεδομένα από το περιβάλλον τα οποία χρειάζεται το πρόγραμμα για να παράγει την επιθυμητή έξοδο.
- *Τρόπος Εκτέλεσης:* Ο τρόπος με τον οποίο γίνεται η επεξεργασία των δεδομένων (ροή εκτέλεσης, αλγόριθμοι, μοντέλα υπολογισμού).
- *Έξοδος:* Τα τελικά αποτελέσματα.

Προσοχή! Στην Python μετράνε οι χαρακτήρες διαστήματος! Αν λοιπόν έχετε συναντήσει ποτέ μέχρι τώρα το ακόλουθο σφάλμα:

```
IndentationError: expected an indented block
```

τότε σημαίνει ότι ξεχάσατε να τοποθετήσετε τους χαρακτήρες διαστήματος.

Μια καλή πρακτική είναι να μην χρησιμοποιείται tabs εκτός και αν το πρόγραμμα όπου γράφετε τον κώδικα σας τους αντικαθιστά αυτόματα με κενά. Έτσι εξασφαλίζεται ότι ο κώδικας σας θα δείχνει ίδιος παντού και θα αποφύγετε την μίξη κενών και tabs που δεν επιτρέπεται. Όσον αφορά πόσα κενά πρέπει να χρησιμοποιείται, ο συνιστώμενος αριθμός σύμφωνα με το PEP 8 είναι τέσσερα.

Η στοίχιση του κώδικα χρησιμοποιείται για να καταδείξει σε ποιο μπλοκ ανήκουν οι δηλώσεις που γράφουμε. Πιο πρακτικά, μετά από άνω κάτω τελεία πρέπει να αυξάνεται την στοίχιση του κειμένου κατά τέσσερα κενά. Όταν τελειώσετε το μπλοκ που γράφατε, μειώνετε και την στοίχιση του κώδικα που ακολουθεί. Κατά αυτό τον τρόπο, η Python αντικαθιστά τις αγκύλες που χρησιμοποιούνται σε άλλες γλώσσες προγραμματισμού για την διευκρίνηση των μπλοκ κώδικα.

3.2 Ακολουθιακή Εκτέλεση

Ορισμός 3.2.1. Ο έλεγχος ροής αφορά την σειρά με την οποία ανεξάρτητες δηλώσεις, εντολές ή κλήσεις συναρτήσεων εκτελούνται ή αποτιμώνται.

Στην ακολουθιακή εκτέλεση, συνήθως ο δείκτης εντολών αυξάνεται αυτόματα μετά την προσκόμιση μιας εντολής προγράμματος, καθώς συνήθως οι εντολές σε ένα πρόγραμμα εκτελούνται ακολουθιακά από την μνήμη. Μπορούμε να σκεφθούμε την ακολουθιακή εκτέλεση σαν ένα αυτοκίνητο που κρατάει σταθερή ευθύγραμμη πορεία. Κάθε στιγμή γνωρίζουμε σε ποιο μέρος του δρόμου θα βρίσκεται. Το αυτοκίνητο απλά προχωράει λίγα μέτρα παρακάτω κάθε φορά. Έτσι και ένα πρόγραμμα που κάνει ακολουθιακή εκτέλεση εντολών sequential, εκτελεί κάθε φορά την επόμενη εντολή στη μνήμη. Ένα παράδειγμα ακολουθιακής εκτέλεσης φαίνεται παρακάτω:

```
print(123)
```

```
print()
print('Hello', end=' ')
print('World')
```

Η συνάρτηση print() στο τέλος του αλφαριθμητικού που τυπώνει, από προεπιλογή, τυπώνει τον χαρακτήρα μιας καινούργιας γραμμής. Το προαιρετικό όρισμα της end χρησιμοποιείται στην περίπτωση που θέλουμε να τυπώσουμε κάτι διαφορετικό από τον χαρακτήρα καινούργιας γραμμής.

Υπάρχουν όμως ειδικές εντολές όπως διακλάδωσης υπό συνθήκη, άλματα και υπορουτίνες που διακόπτουν την φυσιολογική ροή τοποθετώντας μια καινούργια τιμή στον μετρητή προγράμματος (program counter). Αυτές είναι που κάνουν ένα πρόγραμμα να μπορεί να 'προσαρμόζεται' στις συνθήκες του περιβάλλοντος του και να ακολουθεί διαφορετική πορεία ανάλογα με το τι του ζητείται.

3.3 Είδη Ελέγχου Ροής

Ορισμός 3.3.1. Τα είδη ελέγχου ροής είναι:

- Συνέχεια σε διαφορετική δήλωση (statement) (unconditional branch or jump).
- Εκτέλεση ενός συνόλου δηλώσεων μόνο αν ικανοποιείται κάποια συνθήκη.
- Εκτέλεση ενός συνόλου εντολών καμία ή περισσότερες φορές μέχρι κάποια συνθήκη να ικανοποιηθεί.
- Εκτέλεση ενός συνόλου απομακρυσμένων εντολών μετά τις οποίες ο έλεγχος της ροής συνήθως επιστρέφει (συναρτήσεις, μέθοδοι).
- Σταμάτημα ενός προγράμματος, αποτρέποντας οποιαδήποτε περαιτέρω εκτέλεση.

3.4 Δομή ελέγχου if

Αν επιθυμούμε την εκτέλεση μιας ακολουθίας εντολών μόνο εφόσον πληρείται μια συγκεκριμένη συνθήκη, τότε χρησιμοποιούμε την δομή if και στην συνέχεια την συνθήκη την οποία θέλουμε να ελέγξουμε. Αν αυτή η συνθήκη αποτιμάται ως αληθής, τότε το σύνολο των εντολών που περιέχονται στην εντολή if θα εκτελεστούν, αλλιώς η ροή του προγράμματος θα συνεχίσει από το τέλος της if.

```
answer = int(input('What is the product of 2 * 5: '))

if (answer == 10):
    print('Correct!')

print('Exiting program')
```

Αν δοκιμάσουμε να εκτελέσουμε τον παραπάνω κώδικα, τότε για οποιονδήποτε αριθμό επιτελεί την λειτουργία για την οποία διαφημίστηκε πριν λίγο. Αν όμως αντί για αριθμό δοκιμάσουμε να δώσουμε ως είσοδο ένα οποιοδήποτε γράμμα τότε παίρνουμε το ακόλουθο μήνυμα:

```
What is the product of 2 * 5: g
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'g'
```

Αυτό συμβαίνει γιατί προσπαθούμε να μετατρέψουμε το γράμμα 'g' σε ακέραιο μέσω της συνάρτησης int(), πράγμα που δεν γίνεται, και για αυτό εγείρεται μια εξαίρεση τύπου ValueError. Αν θέλαμε να είμαστε πιο σωστοί ως προγραμματιστές θα έπρεπε να χειριζόμαστε αυτή την εξαίρεση. Πως γίνεται αυτό θα το δούμε στην ενότητα Εξαιρέσεις.

Αν ανάλογα με την αποτίμηση μιας συνθήκης θέλουμε να εκτελέσουμε διαφορετικές ενέργειες, τότε μπορούμε να χρησιμοποιήσουμε την δομή if...else....

```
from math import sqrt
```

```
a = float(input("Coefficient of the x^2: "))
b = float(input("Coefficient of the x: "))
c = float(input("Constant: "))

if a == 0:
  print("This is not a 2nd degree equation")
  import sys
  exit()

D = b**2 - 4 * a * c

if D >= 0:
  x1 = (-b + sqrt(D)) / (2 * a)
  x2 = (-b - sqrt(D)) / (2 * a)
else:
  x1 = complex(-b / (2 * a), sqrt(-D) / 2 * a)
  x2 = complex(-b / (2 * a), -sqrt(-D) / 2 * a)

sf = "x1 = {0}, x2 = {1}"
print(sf.format(x1, x2))
```

Όπου στο παραπάνω πρόγραμμα υπολογίζονται οι λύσεις μιας δευτεροβάθμιας εξίσωσης. Ανάλογα με το αν η διακρίνουσα είναι θετική ή αρνητική χρησιμοποιούμε μιγαδικούς αριθμούς και στην συνέχεια τυπώνουμε το αποτέλεσμα. Υπενθυμίζουμε πως ο τύπος επίλυσης δευτεροβάθμιας εξίσωσης είναι: $x_{1,2} = \frac{-\beta + \sqrt{\Delta}}{2\alpha}$, όπου $\Delta = \beta^2 - 4\alpha\gamma$. Αν η διακρίνουσα Δ είναι θετική τότε έχουμε πραγματικές ρίζες (περίπτωση μέσα στο πρώτο μπλοκ κώδικα μετά το if ενώ αλλιώς έχουμε μιγαδικές[1] (περίπτωση μέσα στο μπλοκ else).

[1] Στην περίπτωση που η διακρίνουσα είναι 0 έχουμε μια διπλή πραγματική ρίζα, πράγμα που ισχύει και στο αποτέλεσμα του προγράμματος μας αφού δημιουργούνται δυο ίδιες ρίζες με μηδενικό φανταστικό μέρος.

3.4.1 Πολλαπλές περιπτώσεις

if...elif...else...

Στην Python, δεν υπάρχει η δήλωση switch όπου αναλόγως με μια παράμετρο εκτελείται μια λειτουργία από k διαφορετικές, όπου k είναι το πλήθος των διαφορετικών περιπτώσεων που έχουμε συμπεριλάβει στην switch. Η δήλωση switch είναι γνωστή από γλώσσες όπως οι C/C++ ή Java. Στην Python, ο προτεινόμενος τρόπος είναι η χρήση της δομής if...elif...else.... Ένα παράδειγμα αυτής της χρήσης είναι:

```
x = int(input("Please enter an integer: "))
if x < 0:
  print('Negative number, transforming into positive')
  x = -x # make it positive
# Be careful, because the following statements will be
# examined only if the first condition is false, they
# won't be executed
elif x == 0:
  print('Zero')
elif x == 1:
  print('One')
else:
  print('Great than 1')
```

Αυτός ο προτεινόμενος τρόπος, όταν τα elif που χρησιμοποιούνται είναι σχετικά λίγα, κρατώντας έτσι τον κώδικα ευανάγνωστο.

Χρήση Λεξικού

Ένας άλλος τρόπος να εξομοιώσουμε την λειτουργία μιας δήλωσης switch, είναι με την χρήση λεξικών. Αν μόλις γνωρίζετε τα βασικά χαρακτηριστικά της Python μπορείτε προς το παρόν να τον παραβλέψετε και μόλις εξοικειωθείτε περισσότερο με τις δομές δεδομένων που αυτή περιέχει να ξαναγυρίσετε σε αυτό το κομμάτι.

Τα λεξικά μας επιτρέπουν να αντιστοιχήσουμε σε ένα κλειδί τους (key) μια μεταβλητή. Χρησιμοποιώντας λοιπόν ως κλειδιά τις επιλογές που θέλου-

με να έχουμε, και αντιστοιχίζοντας τα με τις συναρτήσεις που θέλουμε να χρησιμοποιήσουμε, παίρνουμε λειτουργικότητα εφάμιλλη με την επιθυμητή.

```
# Sample functions
def square(x):
    return x**2

def double(x):
    return x*2

opdict = {"a":square, "b":square, "c":double}

# Example (random access O(1))
print(opdict["b"](5))

# Example 2 (checking everything)
for operation in "abc":
    print(operation, opdict[operation](3))
```

Η δομή που προκύπτει είναι εφάμιλλη αυτή της if...elif...else.... Προσοχή πρέπει να δοθεί στο γεγονός πως, όπως και παραπάνω, εκτελείται μόνο η πρώτη ενέργεια που αντιστοιχίζεται σε συνθήκη που βρίσκουμε αληθινή.

Κάποιος, θα μπορούσε να συνδυάσει την παραπάνω ιδέα με εξαιρέσεις, ώστε να μπορέσει να χειριστεί και περιπτώσεις όπου χρησιμοποιείται ως κλειδί μια μεταβλητή που δεν είχαμε προβλέψει.

```
# Sample functions
def square(x):
  return x**2

def double(x):
  return x*2

def decrease(x):
  return x-1
```

```
opdict = {"a":square, "b":double, "c":decrease}

try:
  print(opdict["d"](5))
except KeyError:
  print('Invalid use of index')


# Example 2 (checking everything)
for operation in "abc":
  print(operation, opdict[operation](3))
```

3.5 Βρόγχοι επανάληψης

Ένας βρόγχος είναι μια ακολουθία εντολών οι οποίες δηλώνονται μια φορά, αλλά μπορούν να εκτελεστούν πολλές διαδοχικές φορές. Ο κώδικας μέσα στον βρόγχο (το σώμα του βρόγχου) θα εκτελείται για έναν καθορισμό αριθμό επαναλήψεων, ή για όσο ισχύει μια συνθήκη. Κατά αυτό τον τρόπο, έχουμε και τον διαχωρισμό σε for και σε while βρόγχους επανάληψης.

3.5.1 Βρόγχοι for

Οι βρόγχοι for εκτελούνται για συγκεκριμένο πλήθος φορών. Για την δημιουργία τους χρησιμοποιείται η συνάρτηση range(). Έτσι, όσο η συνάρτηση range() επιστρέφει ένα καινούργιο αντικείμενο, οι εντολές στο σώμα του βρόγχου συνεχίζουν να εκτελούνται.

Ως παράδειγμα βλέπουμε τον υπολογισμό των 20 πρώτων αριθμών Fibonacci. Υπενθυμίζουμε πως ο τύπος για τον υπολογισμό του n-οστού αριθμού Fibonacci είναι $F_n = F_{n-1} + F_{n-2}$ όπου $F_0 = 0$ και $F_1 = 1$.

```
# the 2 first fibonacci numbers are always known
a, b = 0, 1
# print the first fibonacci number (a)
print(a, end=' ')
```

```
# find the next 18 fibonacci numbers and print them
for i in range(18):
    print(b, end=' ')
    a, b = b, a + b

# print the last fibonnaci number (of totally 20)
print(b)
```

Όπου το i θα πάρει διαδοχικά τις τιμές από 0 έως και 17, με αποτέλεσμα να εκτελεστούν οι εντολές στο σώμα του βρόγχου 18 φορές. Η συνάρτηση[2] range() δουλεύει μόνο για ακεραίους. Θα μπορούσαμε να παράγουμε με την βοήθεια της range() του αριθμούς μέσα σε ένα συγκεκριμένο εύρος, ή με ένα συγκεκριμένο βήμα όπως φαίνεται στις ακόλουθες περιπτώσεις.

```
>>> for i in range(100, 120):
...     print(i, end=' ')
...
100 101 102 103 104 105 106 107 108 109 110
111 112 113 114 115 116 117 118 119
```

```
>>> for i in range(100, 120, 2):
...     print(i, end=' ')
...
100 102 104 106 108 110 112 114 116 118
```

Στην πρώτη περίπτωση τυπώνουμε όλους τους αριθμούς στο διάστημα $[100, 120)$ ενώ στην δεύτερη περίπτωση τυπώνουμε τους αριθμούς στο διάστημα $[100, 120)$ αλλά με βήμα ανά 2, επομένως τυπώνονται μόνο οι άρτιοι αριθμοί μέσα σε αυτό το διάστημα.

Προσοχή πρέπει να δωθεί ότι το σώμα του βρόγχου είναι υποχρεωτικά στοιχισμένα έτσι ώστε να ξεχωρίζουν ποιές εντολές βρίσκονται σε αυτό.

[2] γεννήτορας για την ακρίβεια, βλέπε αντίστοιχη ενότητα.

3.5.2 Βρόγχοι while

Αντίθετα με τους βρόγχους for, οι βρόγχοι while θα τρέχουν συνεχώς όσο πληρείται μια συγκεκριμένη συνθήκη.

```
def fib(a, b):
  sum = 0;
  while a < 4 * (10 ** 6):
      sum += a
      a, b = b, a + b
  return sum

a, b = 0, 1
print(fib(a,b))
```

Το παραπάνω παράδειγμα υπολογίζει το άθροισμα όλων των αριθμών της ακολουθίας Fibonnaci οι οποίοι είναι μικρότεροι από το $4 \cdot 10^6$.

3.6 Η δήλωση break

Η δήλωση break χρησιμοποιείται για να βγει το πρόγραμμα από έναν βρόγχο μόλις την συναντήσει. Συνήθως για να εκτελεστεί πρέπει να πληρείται μια προϋπόθεση, και για αυτό τον λόγο συνοδεύεται από μια δομή *if* που καθορίζει πότε εκτελείται.

Καλό θα ήταν να αποφεύγεται η χρήση της δήλωσης break καθώς κάνει πιο δυσανάγνωστο τον κώδικα. Συχνά μπορεί να αφαιρεθεί με την αλλαγή της συνθήκης στο βρόγχο. Κάποιες άλλες φορές, βοηθάει στο να κατανοήσουμε πως τερματίζεται ο βρόγχος λόγω κάποιας ειδικής περίπτωσης.

Στο ακόλουθο παράδειγμα βλέπουμε τον υπολογισμό όλων των υποομάδων ενός συνόλου Z_n με αριθμητική υπολοίπου (modulo). Όταν βρούμε το 0, σημαίνει πως έχουμε διατρέξει όλα τα στοιχεία στα οποία μπορούμε να φθάσουμε από το συγκεκριμένο.

```
def printSubgroups(n):
    """
    Prints all the subgroups of Zn
```

```
    """
    for i in range(0, n):
        print('<', end='')
        print(i, end='>: ')
        for j in range(1, n + 1):
            num = (i*j)%n
            print(num, end = '')
            if (num % n) == 0:
                break
            print(', ', end=' ')
        print()
```

3.7 Η δήλωση with

Στην έκδοση 3 της Python, η λέξη with αποτελεί λέξη κλειδί, και μπορεί να έχει διάφορες χρήσεις.

Η κύρια χρήση της είναι για την απόκτηση και απελευθέρωση πόρων. Μπορούμε με το with να ανοίξουμε ένα αρχείο, χρησιμοποιώντας το with για τον έλεγχο του επιτυχούς ανοίγματος αυτού του αρχείου. Στο παράδειγμα παρακάτω, το αρχείο κλείνει μόλις τελειώσει το block του with. Με την with εξασφαλίζουμε την κατάλληλη δέσμευση και αποδέσμευση των πόρων όταν φεύγουμε από το αντίστοιχο μέρος μπλοκ κώδικα ανεξάρτητα από το αν έχουν συμβεί ή όχι εξαιρέσεις.

```
filename = "CanadaSong.txt"

data = """\
First come the black flies,
Then the Horse flies,
Then the Deer flies,
Then the snow flies!
"""

with open(filename, "w") as fout:
```

```
    fout.write(data)
```

Έτσι, φεύγοντας από το with, το αρχείο με όνομα filename, έχει σίγουρα κλείσει. Αυτό συμβαίνει είτε εφόσον δεν έγινε κάποια εξαίρεση και άνοιξε κανονικά για εγγραφή όπως θα θέλαμε, είτε έγινε η εξαίρεση και το αρχείο δεν άνοιξε. Έτσι, χρησιμοποιώντας το with γίνεται η κατάλληλη δέσμευση και αποδέσμευση πόρων.

3.7.1 Πολλαπλό with

Η with μπορεί να χρησιμοποιηθεί και όπως παρουσιάζεται στο παρακάτω παράδειγμα, που είναι ισοδύναμο με την εμφώλευση των δυο with δηλώσεων. Κατά αυτό τον τρόπο μπορούμε να κάνουμε ακόμα πιο ευανάγνωστο τον κώδικα μας. Μια χρήση που βρίσκει αρκετά συχνά η συγκεκριμένη λειτουργικότητα είναι όταν θέλουμε να διαβάζουμε από ένα αρχείο και να γράφουμε σε ένα άλλο, αφού έχουμε κάνει κάποια επεξεργασία στο αρχείο από το οποίο διαβάζουμε.

```
# every line containing these words won't be written
# at the output
exclude_class = ['not', 'negation']

with open('infile', 'r') as f1, open('outfile', 'w') as f2:
    # for each line in the input file
    for line in f1.readlines():
        # by default write a line to the outfile unless
        # it doesn't contain one of the excluded words
        write_to_file = True
        # for each word in a line of the input file
        for word in line.split():
            # if this word doesn't belong to closed_class
            if word.rstrip() in exclude_class:
                write_to_file = False
                break
        if write_to_file:
```

```
f2.write(line)
```

3.7.2 Πώς δουλεύει

Αν η κλάση που περιγράφει ένα αντικείμενο υλοποιεί τις μεθόδους __enter__ και __exit__, τότε η δήλωση with εξασφαλίζει πως όταν ζητηθεί ο πόρος καλείται η συνάρτηση __enter__ και εφόσον δεσμευθεί αυτός ο πόρος, μετά το πέρας του μπλοκ κώδικα που εμπεριέχεται μέσα σε αυτό, καλείται η συνάρτηση __exit__.

Προσοχή πρέπει να δωθεί σε δύο περιπτώσεις. Μόνο στην περίπτωση που έχει δεσμευθεί σωστά ο πόρος που περιγράφεται στο with καλείται η __exit__, αφαιρώντας έτσι το βάρος από τον προγραμματιστή να πρέπει να ελέγχει για την δέσμευση του πόρου (αν δεν έχει γίνει, δεν έχει νόημα και η αποδέσμευση του). Αν πάλι δημιουργηθεί μια εξαίρεση λόγω κάποιας δήλωσης μέσα στο μπλοκ του with, τότε αν η εξαίρεση μπορεί να αντιμετωπιστεί από την συνάρτηση __exit__, τότε αυτή δεν φαίνεται εξωτερικά καθώς έχει αντιμετωπιστεί. Σε κάθε άλλη περίπτωση η εξαίρεση επιστρέφεται στο περιβάλλον που κάλεσε την συνάρτηση.

Κεφάλαιο 4

Αριθμοί και Αριθμητικές Λειτουργίες

> And there was huge numbers of UFOs around my parents home in Kingston.
>
> Betty Hill)

Η Python χαρακτηρίζεται ως διερμηνευόμενη γλώσσα επειδή εκτελείται από έναν διερμηνέα. Υπάρχουν δυο τρόποι να χρησιμοποιήσουμε τον διερμηνέα: *διαδραστικά* και σε προγράμματα σεναρίου (scripts). Μέχρι τώρα όλα τα παραδείγματα μας ήταν scripts. Τώρα θα δούμε ορισμένα διαδραστικά παραδείγματα.

Η Python μπορεί να χρησιμοποιηθεί για αριθμητικές πράξεις, ως υποκατάστατο σε μια αριθμομηχανή, μέσα από την διαδραστική διεπαφή γραμμής εντολών που χρησιμοποιεί. Ιδιαίτερα χαρακτηριστικά της, την καθιστούν ιδιαίτερα αποτελεσματική σε αυτό το έργο. Παράδειγμα αυτών είναι η δυνατότητα της να αποθηκεύει μεγάλους ακέραιους αριθμούς χωρίς να δημιουργείται υπερχείλιση (overflow). Άλλο παράδειγμα είναι η αποδοτική υλοποίηση αλγορίθμων στην βασική της βιβλιοθήκη.

Για να εισέλθουμε στον διερμηνευτή της python ώστε να μπορέσουμε να εκτελέσουμε τις λειτουργίες που θα περιγράψουμε παρακάτω, σε γραμμή εντολών πληκτρολογούμε:

```
python
```

Στην συνέχεια εμφανίζεται ο διερμηνευτής. Για να φύγουμε από το περιβάλλον διερμηνευτή, πατάμε ctrl+d σε Linux/BSD ή ctrl+z σε Windows.

4.1 Βασικές πράξεις

Οι βασικές πράξεις (πρόσθεση, αφαίρεση, πολλαπλασιασμός, διαίρεση) χρησιμοποιούν τους αντίστοιχους δυαδικούς τελεστές ($+, -, *, /$)

```
>>> 3+8
11
>>> 3-8
-5
>>> 3*8
24
>>> 3/8
0.375
```

4.1.1 Διαίρεση

Υπάρχουν δύο τελεστές διαίρεσης. Ο ένας αφορά την διαίρεση που επιστρέφει μόνο το πηλίκο χωρίς το δεκαδικό μέρος (διαίρεση ακεραίων) και ο άλλος την κλασική διαίρεση που μας επιστρέφει το αποτέλεσμα σε δεκαδική μορφή.

```
>>> 7/2
3.5
>>> 7//2
3
```

4.1.2 Ύψωση σε Δύναμη

Για την ύψωση σε κάποια δύναμη, χρησιμοποιείται ο δυαδικός τελεστής $**$ όπου προηγείται η βάση και ακολουθεί ο εκθέτης.

```
>>> 2**10
1024
>>> 3**8
6561
```

Αν κάποιος θέλει να υψώσει σε κάποια δύναμη και να κρατήσει το υπόλοιπο της διαίρεσης με το n ($mod\, n$ με μαθηματικό συμβολισμό), τότε η python κάνει χρήση ενός έξυπνου αλγορίθμου (modular exponentiation) και έτσι, η συγκεκριμένη πράξη εκτελείται πολύ γρήγορα. Προηγείται η βάση του αριθμού, ακολουθεί ο εκθέτης και στην συνέχεια ο αριθμός με τον οποίον κάνουμε διαίρεση και θέλουμε να κρατήσουμε μόνο το υπόλοιπο. Κατά αυτό τον τρόπο, αν κάποιος για παράδειγμα θέλει να βρει τα τρία τελευταία ψηφία του αριθμού 2^{1024}, τότε μπορεί να κάνει την ύψωση στη δύναμη 1024 και από τη διαίρεση με το 1000 να κρατήσει το υπόλοιπο. Όλα αυτά η Python τα υλοποιεί πολύ απλά.

```
>>> pow(2, 1024, 1000)
216
```

4.2 Ακέραιοι

Μπορούμε να τυπώνουμε ακεραίους σε διάφορα συστήματα, με την χρήση των τελεστών:

1. d: Για δεκαδικούς αριθμούς
2. b: Για δυαδικούς αριθμούς (ή εναλλακτικά bin())
3. o: Για οκταδικούς αριθμούς (ή εναλλακτικά oct())
4. x: Για δεκαεξαδικούς αριθμούς (ή εναλλακτικά hex())

Προσθέτοντας τον χαρακτήρα της δίεσης, εκτυπώνεται επιπλέον πληροφορία που δείχνει σε ποιο σύστημα ανήκει ο εκτυπωμένος αριθμός.

```
print('value = {0:2d}'.format(12))    # value = 12
print('value = {0:2b}'.format(12))    # value = 1100
print('value = {0:#2b}'.format(12))   # value = 0b1100
print('value = {0:2o}'.format(12))    # value = 14
print('value = {0:#2o}'.format(12))   # value = 0o14
print('value = {0:2x}'.format(12))    # value =  c
print('value = {0:#2x}'.format(12))   # value = 0xc

print('value = {0:5}'.format(bin(12)))   # value = 0b1100
print('value = {0:5}'.format(oct(12)))   # value = 0o14
print('value = {0:5}'.format(hex(12)))   # value = 0xc
```

4.3 Αριθμοί Κινητής Υποδιαστολής

Για την μορφοποίηση αριθμών κινητής υποδιαστολής, χρησιμοποιούμε τον προσδιοριστή (specifier) f. Μπορούμε να πούμε πόσα ψηφία επιθυμούμε να εκτυπωθούν από το δεκαδικό μέρος. Επίσης, με την χρήστη του τελεστή $+$, φαίνεται το πρόσημο και στην περίπτωση θετικών αριθμών.

Ακολουθούν και διάφορα παραδείγματα.

```
print('x = {0:5.3f}'.format(1234.567))     # x = 1234.567
print('x = {0:12.6f}'.format(1234.567))    # x = 1234.567000
print('x = {0:-12.9f}'.format(1234.567))   # x = 1234.567000
print('x = {0:+12.9f}'.format(1234.567))   # x = +1234.567000
print('x = {0:+12.9f}'.format(-1234.567))  # x = -1234.567000
print('x = {0:012.3f}'.format(1234.567))   # x = 00001234.567
print()
print('x = {0:1.5e}'.format(1234.567))     # x = 1.23457e+03
print('x = {0:1.5e}'.format(0.000001234))  # x = 1.23400e-06
print('x = {0:1.5g}'.format(0.000001234))  # x = 1.234e-06
print()
print('x = {0:2.3%}'.format(0.337))        # x = 33.700%
print()
```

```
print('pi = {0:.3f}'.format(math.pi))    # pi = 3.142
print('pi = {0:.8f}'.format(math.pi))    # pi = 3.14159265
```

4.4 Μιγαδικοί Αριθμοί

Η χρήση μιγαδικών αριθμών είναι πολύ εύκολη καθώς μπορούμε να κάνουμε τις πράξεις που γνωρίζουμε με τους συνηθισμένους τελεστές (πρόσθεση, αφαίρεση, πολλαπλασιασμό). Για την δήλωση τους χρησιμοποιούμε την συνάρτηση complex. Για όσους δεν γνωρίζουν τι είναι μιγαδικοί αριθμοί, μπορούν να φανταστούν ζεύγη αριθμών όπως για παράδειγμα κάποιες συντεταγμένες.

```
a = complex(1, 1)
b = complex(1, 2)
print(a+b)
print(a*b)
```

Κεφάλαιο 5

Συναρτήσεις

> Αν υποθέσουμε ότι αυτό είναι δυνατό, (να μεταδώσουμε τη σοφία παντού) τότε ειλικρινά ο τρόπος ζωής των θεών θα περάσει στους ανθρώπους. Τα πάντα θα είναι γεμάτα δικαιοσύνη και φιλαλληλία και δεν θα έχουμε ανάγκη από τείχη ή νόμους.
>
> Διογένης ο Οινοανδέας

ΈναΣ μεγάλος αριθμός από κατασκευές, είτε φυσικές είτε τεχνητές, βασίζεται στην ιδέα της χρησιμοποίησης μικρότερων δομικών μονάδων. Αυτές οι δομικές μονάδες μπορούν να συνδυαστούν μεταξύ τους με πολλούς διαφορετικούς τρόπους, παρέχοντας μια μεγάλη ποικιλία αντικειμένων με περίπλοκες ιδιότητες. Η ιδέα των μικρότερων δομικών μονάδων, απλοποιεί την ανάπτυξη αυτών των αντικειμένων παρέχοντας μικρότερες δομικές λειτουργίες. Επιπρόσθετα, καθίσταται ευκολότερος ο έλεγχος αυτών.

Το ίδιο σκεπτικό υλοποιείται και στην Python χρησιμοποιώντας τις συναρτήσεις. Οι συναρτήσεις μας επιτρέπουν την ομαδοποίηση κώδικα που επιτελεί μια συγκεκριμένη λειτουργία και, κατά συνέπεια, την ευκολότερη επαναχρησιμοποίησή του. Επιπλέον, επιτρέπουν τον σχεδιασμό του τελικού συστήματος λογισμικού σε ένα υψηλότερο επίπεδο αφαίρεσης, όπου οι λειτουργικότητα παρέχεται μέσω συναρτήσεων. Έτσι, στην συνέχεια απομένει

η υλοποίηση αυτών, μέχρι του σημείου όπου οι μικρότερες δομικές μονάδες που χρησιμοποιούνται, παρέχονται ήδη από το σύστημα.

5.1 Βασικοί ορισμοί

Ορισμός 5.1.1. *Συνάρτηση* είναι μια ονομαζόμενη ακολουθία δηλώσεων η οποία πραγματοποιεί έναν συγκεκριμένο υπολογισμό. Αποτελούν μια τεχνική αφαίρεσης μέσω της οποίας μπορεί να δοθεί όνομα σε μια σύνθετη λειτουργία και στην συνέχεια να παρέχεται η δυνατότητα να αναφερόμαστε σε αυτή τη σύνθετη λειτουργία ως μονάδα.

Αν μια συνάρτηση εφαρμόζεται σε αντικείμενο, ονομάζεται *μέθοδος*.

Ορισμός 5.1.2. *Όρισμα* είναι μια τιμή η οποία παίρνεται από το πρόγραμμα στην κλήση της συνάρτησης. Μπορεί να χαρακτηριστεί και ως είσοδος της συνάρτησης από το πρόγραμμα από το οποίο καλείται.

Μια συνάρτηση μπορεί να επιδέχεται πολλά ορίσματα.

Ορισμός 5.1.3. *Επιστρεφόμενη τιμή* είναι η τιμή η οποία επιστρέφεται από την συνάρτηση στο περιβάλλον που την κάλεσαι μέσω μιας δήλωσης return. Όταν συναντάται η δήλωση return, τερματίζεται η εκτέλεση της συνάρτησης και η ροή εκτέλεσης επιστρέφει στο περιβάλλον από όπου καλέστηκε.

Στο ακόλουθο παράδειγμα βλέπουμε την δημιουργία μιας συνάρτηση με δυο ορίσματα και την επιστροφή του τελικού της αποτελέσματος. Στο τέλος, εκτυπώνουμε την τιμή που επιστρέφεται καλώντας την συνάρτηση.

```
def add(a, b):
    c = a + b

    return c

print(add(2,3))
```

Μπορούμε να επιστρέψουμε και περισσότερες από μια τιμές μέσω κάποιας συνάρτησης.

```
from math import ceil

def fun2(c):
    a = c // 2
    b = c - a

    return a, b

d, e = fun2(7)
print(d, e)
```

Αξίζει να προσεχθούν τα ακόλουθα σημεία:

1. Η πρώτη γραμμή, είναι η δήλωση της συνάρτησης. Ονομάζεται και υπογραφή της μεθόδου και ξεκινάει με την λέξη κλειδί *def*.
2. Η υπογραφή της μεθόδου τελειώνει με άνω κάτω τελεία (:). Αυτή καταδεικνύει ότι θα ακολουθήσει το σώμα της συνάρτησης, που περιέχει και την υλοποίηση της λειτουργικότητας της.
3. Οι δηλώσεις μέσα στο σώμα της συνάρτησης έχουν μεγαλύτερη εσοχή από ότι η δήλωση (υπογραφή) της συνάρτησης (ή ορισμός). Πρέπει να βρίσκονται πιο 'μέσα' όπως και στο σώμα ενός βρόγχου ή μιας συνθήκης.

5.2 Αγνές Συναρτήσεις και Συναρτήσεις Τροποποίησης

Αγνές Συναρτήσεις (Pure Functions) ονομάζονται οι συναρτήσεις που δεν προκαλούν καμία αλλαγή στο αντικείμενο στο οποίο καλούνται. Αντιθέτως, συναρτήσεις τροποποίησης (modifier functions) καλούνται αυτές που μπορούν να προκαλέσουν αλλαγές στο συγκεκριμένο αντικείμενο. Αυτές οι συναρτήσεις λέμε ότι έχουν και παρενέργειες (side effects), καθώς πέρα από την διεργασία που επιτελούν, μεταβάλουν και την τιμή του αντικειμένου στο οποίο κλήθηκαν.

```
a = 'Python'
print(a.upper())
print(a)
```

Επειδή ένα αλφαριθμητικό είναι σταθερό, η συνάρτηση upper() δεν θα μπορούσε παρά να είναι μια αγνή συνάρτηση καθώς δεν γίνεται να μεταβληθεί ένα σταθερό αντικείμενο. Έτσι, αυτό που κάνει είναι να δημιουργεί ένα καινούργιο string που έχει όλα τα γράμματα κεφαλαία χωρίς να επηρεάζεται το a.

Αντιθέτως, πάνω σε ένα μεταβλητό αντικείμενο (mutable), μια συνάρτηση μπορεί να είναι είτε αγνή, είτε συνάρτηση τροποποίησης. Επομένως, ενώ όπως περιμέναμε, η count() δεν επηρεάζει την λίστα b, η συνάρτηση reverse() επιδρά απευθείας πάνω σε αυτή και την αντιστρέφει.

```
b = ['a', 'b', 'c', 'd',]
print(b.count('a'))
print(b)
print(b.reverse())
print(b)
```

5.3 Συμβολοσειρές Τεκμηρίωσης (Docstrings)

Για να διευκολύνουμε την ανάγνωση του κώδικα, χρησιμοποιούμε σχόλια. Μια συνηθισμένη πρακτική, είναι πριν προχωρήσουμε στην υλοποίηση μιας συνάρτησης να καταγράφουμε την επιθυμητή της λειτουργία με απλά λόγια. Στην `Python`, υπάρχει η σύμβαση αφού ορίσουμε μια συνάρτηση, να δίνουμε μια περιγραφή της σε φυσική γλώσσα μέσα από ένα αλφαριθμητικό το οποίο μπορεί να συνεχιστεί σε παραπάνω από μια γραμμές.

```
def fibonacci(n):
    """ Return the nth Fibonacci number counting from 0. """

    # we should have used the closed formula for fibonacci
    # TODO: throw an error for negative numbers
```

```
    if n <= 1:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)

print(fibonacci(10))
```

Με απλά λόγια, οι συμβολοσειρές τεκμηρίωσης είναι απλά σχόλια που όμως αποκτούν ιδιαίτερη σημασιολογία λόγω των συμβάσεων που ακολουθούνται.

Υπάρχει μια κύρια σύμβαση για το ύφος του κειμένου που γράφουσε στις συμβολοσειρές τεκμηρίωσης. Η πρώτη γραμμή είναι σε προστακτική και αναφέρεται στο τι θέλουμε να κάνει η συνάρτηση. Στη συνέχεια, και αφήνοντας μια γραμμή κενή, μπορούμε να εξηγήσουμε (προαιρετικά) με περισσότερα λόγια τη συνάρτηση μας. Λεπτομέρειες για το ύφος των συμβολοσειρών τεκμηρίωσης μπορούν να βρεθούν στο PEP 257,

Αν χρησιμοποιούμε συμβολοσειρές τεκμηρίωσης, τότε υπάρχουν αρκετά αυτόματα εργαλεία που θα παράγουν αυτόματα την τεκμηρίωση του κώδικα μας σε μια φιλική μορφή με βάση τις συμβολοσειρές τεκμηρίωσης. Σε αντίθεση με τις συμβολοσειρές τεκμηρίωσης, τα απλά σχόλια αγνοούνται.

Παραδείγματα όπου οι συμβολοσειρές τεκμηρίωσης χρησιμοποιούνται μπορούμε να δούμε ακόμα και μέσα από την Python.

```
>>> help(fibonnaci)
>>> print(fibonacci.__doc__)
```

Όπου παρατηρούμε πως ο προγραμματιστής μπορεί να μάθει σχετικά εύκολα τι κάνει μια συνάρτηση χρησιμοποιώντας τις παραπάνω εντολές. Η δεύτερη μάλιστα καταδεικνύει πως γράφοντας μια συμβολοσειρά τεκμηρίωσης, η τιμή της τοποθετείται σε μια μεταβλητή με όνομα `__doc__` η οποία και είναι μέρος του αντικειμένου που περιγράφει την συνάρτηση της Python. Σε αυτό το σημείο, αξίζει να θυμηθούμε πως ακόμα και οι συναρτήσεις στην Python είναι αντικείμενα.

5.4 Προεπιλεγμένα ορίσματα

Μερικές φορές περιμένουμε μια συνάρτηση να καλείται συνήθως με το ίδιο όρισμα εκτός από εξαιρετικές περιπτώσεις. Μπορούμε λοιπόν να ορίσουμε ένα προεπιλεγμένο όρισμα που θα περνιέται στην συνάρτηση εφόσον δεν δηλώσουμε ρητά κάτι διαφορετικό. Αν στην συνέχεια επιθυμούμε να περάσουμε κάποια διαφορετική τιμή ορίσματος, μπορούμε να το κάνουμε όπως φαίνεται στην τελευταία γραμμή του ακόλουθου παραδείγματος.

```
def hello(message='Hello World!'):
    print(message)

hello()
hello(message='Hello Brave New World!')
```

5.4.1 Λίστα ορισμάτων

Η συγκεκριμένη ενότητα πολύ πιθανόν να μη σας χρειαστεί αρχικά. Μπορείτε να την παραβλέψετε. Κυρίως παρατίθεται για περίπτωση που μπορεί να συναντήσετε κάτι αντίστοιχο σε κώδικα που διαβάζετε.

Γενικά, μπορούμε να χρησιμοποιήσουμε και ένα tuple ως λίστα ορισμάτων μιας συνάρτησης.

```
>>> a = (1, 2, 3, 4)
>>> def add(a, b, c, d):
...     return a + b + c + d
...
>>> add(*a)
10
```

Οι τιμές 'ξεδιπλώνονται' και περνάνε στις αντίστοιχες μεταβλητές ως ορίσματα. Στην συνέχεια μπορούμε να τις χειριστούμε σαν να είχαν εισαχθεί κανονικά, μια μια.

Αντίστοιχη λειτουργία μπορούμε να επιτύχουμε και μέσω ενός λεξικού, όπου εκεί μπορούμε και να καθορίσουμε ένα όνομα (κλειδί του λεξικού) σε

κάθε τιμή και ύστερα με βάση αυτό το κλειδί να γίνει το ‘ξεδίπλωμα’ των τιμών. Προσέξτε μόνο ότι σε αυτή την περίπτωση χρησιμοποιούνται δυο αστερίσκοι.

```
b = {'a':2, 'c': 3, 'd': 5, 'b':1}

def print_variables(a, b, c, d):
    print('a: {}'.format(a))
    print('b: {}'.format(b))
    print('c: {}'.format(c))
    print('d: {}'.format(d))

print_variables(**b)
```

Μερικές φορές τα προεπιλεγμένα ορίσματα ονομάζονται και varargs ή default arguments (πιο συνηθισμένο) στην Python.

5.5 Ανώνυμες συναρτήσεις

Υπάρχουν φορές που θέλουμε να ορίσουμε μια συνάρτηση ώστε να την περάσουμε ως όρισμα κάπου αλλού. Παράδειγμα αποτελούν οι αλγόριθμοι ταξινόμησης στους οποίους μπορούμε να περνάμε δικές μας συναρτήσεις σύγκρισης στοιχείων. Για αυτό τον λόγο, υπάρχουν οι ανώνυμες συναρτήσεις. Ας δούμε ένα παράδειγμα πως μετατρέπουμε μια συνάρτηση σε ανώνυμη.

Θα χρησιμοποιήσουμε τη λέξη κλειδί `lambda` από το ελληνικό γράμμα λ. Αυτός ο όρος χρησιμοποιείται ως αναφορά σε ένα κλάδο των μαθηματικών που ασχολείται με συναρτήσεις. Πρακτικά, η λέξη κλειδί `lambda` αντικαθιστά τη λέξη κλειδί `def`. Στη συνέχεια παραλείπουμε το όνομα της συνάρτησης[1] καθώς και τις παρενθέσεις γύρω από τα ορίσματα. Τέλος, γράφουμε τι επιστρέφει η συνάρτηση, χωρίς όμως να χρησιμοποιούμε τη λέξη κλειδί `return`. Ένα παράδειγμα αυτής της στρατηγικής μπορούμε να δούμε παρακάτω.

```
def double(x):
    return x * x
```

[1] Για αυτό το λόγο και λέγονται ανώνυμες συναρτήσεις.

```
def print_results(f, *args):
    print(f(*args))

print_results(double, 5) # prints 25
print_results(lambda x: x * x, 5) # prints 25
```

5.6 Διακοσμητές (Decorators)

Οι διακοσμητές είναι ειδικές συναρτήσεις τις οποίες μπορούμε να χρησιμοποιήσουμε αν θέλουμε να κάνουμε κάτι επιπρόσθετο πριν ή μετά την κλήση μιας συνάρτησης. Για παράδειγμα αν θέλουμε το αποτέλεσμα μιας συνάρτησης να το προσαρμόσουμε σε μια συγκεκριμένη διαμόρφωση, ή αν θέλουμε να αλλάξουμε την σειρά των περιεχομένων σε μια λίστα ώστε να δημιουργήσουμε έναν τυχαιοποιημένο αλγόριθμο (randomized algorithm).

Ο λόγος που αυτό είναι εφικτό είναι επειδή στην Python τα πάντα είναι αντικείμενα. Επομένως, και τις συναρτήσεις μπορούμε να τις χρησιμοποιήσουμε ως αντικείμενα. Μέχρι τώρα είχαμε συνηθίσει να χρησιμοποιούμε πάντα μια συνάρτηση με τις παρενθέσεις δίπλα της. Έτσι, όταν καλέσουμε την συνάρτηση, εκτελείτε το σώμα της. Αν δεν χρησιμοποιήσουμε αυτές τις παρενθέσεις, τότε αναφερόμαστε απευθείας στο αντικείμενο που προσδιορίζει αυτή. Έτσι λοιπόν, παίρνοντας αυτό το όρισμα ως αντικείμενο σε μια άλλη συνάρτηση είναι δυνατό τελικά να τροποποιήσουμε τα αποτελέσματα που εξάγονται ή τα ορίσματα που περνιούνται σε αυτή τη συνάρτηση.

Οι διακοσμητές είναι ένα ιδίωμα της Python που επιτρέπει την κλήση κάποιας συνάρτησης η οποία τροποποιεί ελαφρώς την συμπεριφορά της συνάρτηση που καλούμε. Για να γίνει αυτό πιο ξεκάθαρο, θα δούμε δυο παραδείγματα που έχουν την ίδια συμπεριφορά. Στο πρώτο παράδειγμα καλούμε απλά άλλη μια συνάρτηση ώστε να τροποποιηθεί η έξοδος, ενώ στο δεύτερο κάνουμε ακριβώς το ίδιο πράγμα με την χρήση διακοσμητών (decorators).

```
def makebold(fn):
    def wrapped():
        return "<b>" + fn() + "</b>"
```

```
    return wrapped

def message():
    return "TasPython Guide"

message = makebold(message)
print(message())
```

Μπορούμε να ορίσουμε μια καινούρια συνάρτηση g μέσα σε μια προϋπάρχουσα συνάρτηση f. Έτσι η g φαίνεται μόνο μέσα στην συνάρτηση f. Στο παράδειγμα μας, βλέπουμε ότι μέσα στην συνάρτηση makebold() ορίζεται η συνάρτηση wrapped(). Προσοχή πρέπει να δοθεί στο ότι δεν καλείται αυτόματα η συνάρτηση wrapped() με τον ορισμό της. Πρέπει να την καλέσουμε εμείς ρητά στο σημείο που την χρειαζόμαστε. Επίσης, δεν χρειάζεται να περάσουμε ως όρισμα το αντικείμενο fn στην wrapped() ώστε να μπορέσουμε να κάνουμε πράξεις με αυτό. Η wrapped() το βλέπει ήδη καθώς εμπεριέχεται στο σώμα της μακεβολδ οπότε στο περιβάλλον της είναι ο,τιδήποτε έχει το περιβάλλον της makebold().

Ίσως φανεί παράξενο το γεγονός ότι στο message εκχωρούμε την makebold(message). Όπως είπαμε και πριν, στην Python όλα είναι αντικείμενα. Επομένως, με την makebold(message) έχουμε δημιουργήσει μια καινούργια συνάρτηση την οποία την εκχωρούμε στην message. Έτσι πλέον η message δείχνει σε καινούργια περιεχόμενα πια και έχει τροποποιημένη συμπεριφορά, την οποία και βλέπουμε όταν εκτυπώνουμε ό,τι επιστρέφει.

Αυτό που κάνει η makebold(), είναι πως μέσα της δημιουργείται μια συνάρτηση wrapped(). Η wrapped() χρησιμοποιεί το όρισμα της makebold() και συνενώνει το string που επιστρέφει η fn με κάποια άλλα. Το τελικό αντικείμενο που δημιουργείται επιστρέφεται. Επειδή αυτό το αντικείμενο είναι συνάρτηση, εμείς μπορούμε να το αναθέσουμε σε μια μεταβλητή και στην συνέχεια να το χρησιμοποιήσουμε κανονικά σαν συνάρτηση.

Η ίδια λειτουργικότητα, με πιο κομψό κώδικα επιτυγχάνεται όπως φαίνεται και στο παρακάτω παράδειγμα με την χρήση διακοσμητών.

```
def makebold(fn):
    def wrapped():
```

```
        return "<b>" + fn() + "</b>"
    return wrapped

@makebold
def message():
    return "Python Guide"

print(message())
```

Το αποτέλεσμα είναι ακριβώς το ίδιο, μόνο που σε αυτή την περίπτωση, πριν ορίσουμε την συνάρτηση message() χρησιμοποιούμε τον διακοσμητή ο οποίος προσδιορίζει την συμπεριφορά που αυτός περιγράφει και θα περιλαμβάνει η συνάρτηση μας, και στην συνέχεια υλοποιούμε κανονικά την συνάρτηση. Από εκεί και πέρα, όποτε καλούμε την συνάρτηση message(), αυτή θα περιλαμβάνει και τη συμπεριφορά που ορίζεται από την makebold.

Κεφάλαιο 6

Δομές Δεδομένων

> There is no abstract art. You must always start with something. Afterward you can remove all traces of reality.
>
> Pablo Picasso

Οι δομές δεδομένων μας παρέχουν έναν αποτελεσματικό τρόπο διαχείρισης της διαθέσιμης πληροφορίας. Αποτελούν την ραχοκοκαλία σχεδόν κάθε προγράμματος μας και η κατάλληλη επιλογή τους μπορεί να οδηγήσει σε πολύ μεγάλα οφέλη στην ταχύτητα εκτέλεσης των υπολογισμών. Ο αποτελεσματικός τους σχεδιασμός και η υλοποίηση τους αποτελούν σημαντικό μέρος της επιστημονικής έρευνας στον χώρο των υπολογιστών. Συνήθως, όμως, ένας προγραμματιστής δεν χρειάζεται να τις κατασκευάσει από την αρχή και μπορεί να χρησιμοποιήσει κάποια από αυτές που μοιράζονται με τη βασική βιβλιοθήκη, η οποία είναι αρκετά πλούσια στη Python και η οποία μας διευκολύνει αρχικά στη χρήση τους. Επομένως, στη συνέχεια αυτού του κεφαλαίου θα δούμε ορισμένες από τις πιο βασικές δομές δεδομένων που απαντώνται σχεδόν σε κάθε πρόγραμμα.

6.1 Βασικές Δομές

6.1.1 Βασικά Χαρακτηριστικά

- *Αλφαριθμητικά (Strings):* Περικλείονται σε αποστρόφους (quotes). Είναι σταθερά immutable που σημαίνει ότι δεν γίνεται να αλλάξουν αλλά μόνο να δημιουργηθούν καινούργια.

```
'This is a string'
```

- *Πλειάδες (Tuples):* Τιμές χωρισμένες από κόμμα, οι οποίες συνήθως περικλείονται από παρενθέσεις και μπορούν να περιέχουν οποιοδήποτε τύπο δεδομένων. Είναι σταθερά.

```
('These', 3, 5.711, True, 'Prime')
```

- *Λίστες (Lists):* Περικλείονται σε αγκύλες [ ]. Αντικαθιστούν τους πίνακες που γνωρίζουμε από άλλες γλώσσες προγραμματισμού. Γίνεται δυναμική δέσμευση της μνήμης και είναι τροποποιήσιμες (mutable).

```
['These', 3, 5.711, True, 'Prime']
```

- *Λεξικά (Dictionaries):* Περικλείονται σε αγκύλες { } με τα στοιχεία που περιέχουν διαχωρισμένα με κόμμα. Κάθε στοιχείο (ζευγάρι στοιχείων) του λεξικού αποτελείται από δυο μέρη:
  - ◇ *Κλειδί key:* Τα κλειδιά είναι σταθερές τιμές που δεν αλλάζουν.
  - ◇ *Τιμή value:* Οι τιμές που αντιστοιχούν σε ένα κλειδί, μπορεί να είναι οποιουδήποτε τύπου και μπορούν να αλλαχθούν.

```
{1: 'alpha', 2: 'beta', 3: 'gamma', 4: 'delta'}
```

- *Σύνολα (Sets):* Για να τα ορίσουμε, χρησιμοποιούμε την συνάρτηση set και στην συνέχεια εισάγουμε μια λίστα ως όρισμα. Κάθε τιμή στο σύνολο υπάρχει μόνο μια φορά και δεν είναι είναι ταξινομημένη κατά οποιονδήποτε τρόπο. Οι πράξεις που μπορούμε να εφαρμόσουμε είναι

οι πράξεις που γνωρίζουμε από συνολοθεωρία (ένωση (union), τομή (intersection), διαφορά (difference), υπερσύνολο (issuperset), υποσύνολο(issubset)) καθώς και προσθήκη (add), αφαίρεση (remove) στοιχείου.

Ο πίνακας 6.1 παρουσιάζει συνοπτικά τις βασικές λειτουργίες των δομών δεδομένων που περιγράψαμε παραπάνω.

Δομή Δεδομένων	Κύρια Χρήση
Αλφαριθμητικό (string)	Αλφαριθμητικά
Πλειάδα (tuple)	Αντικείμενα που συσχετίζονται το ένα με το άλλο. Επιστροφή πολλών αντικειμένων από συναρτήσεις. Αντικείμενα που δεν αλλάζουν.
Λίστα (list)	Δυναμική αποθήκευση πολλών αντικειμένων.
Λεξικό (dictionary)	Αναζήτηση αντικειμένων με κλειδιά keys. Τα κλειδιά πρέπει να είναι σταθερά.
Σύνολο	Για ομαδοποίηση διαφορετικών αντικειμένων

Πίνακας 6.1: Ανασκόπηση κύριων λειτουργιών βασικών δομών δεδομένων

6.2 Αλφαριθμητικά

6.2.1 Βασικά στοιχεία αλφαριθμητικών

Τα αλφαριθμητικά, όπως και οι πλειάδες είναι αμετάβλητα, που σημαίνει έπειτα από την δημιουργία τους δεν μπορούν να αλλάξουν τιμή. Ο τελεστής ανάθεσης, μπορεί να κάνει μια μεταβλητή να δείχνει σε ένα αλφαριθμητικό όπως και στους απλούς τύπους. Η προεπιλεγμένη κωδικοποίηση για ένα αλφαριθμητικό είναι UTF-8.

```
# the variable myString points to an immutable string
myString = 'This is a string.'
```

Τα αλφαριθμητικά αποτελούν ένα είδος ακολουθίας (sequence). Αυτό σημαίνει πως κάθε στοιχείο είναι σε μια αριθμημένη σειρά, με βάση και την οποία μπορεί να προσπελαστεί.

```
myString = 'This is a string.'
# the first character of myString
print(myString[0])
# the second character of myString
print(myString[1])
print(myString[2])
print(myString[3])
# prints the last character which is '.'
print(myString[len(myString) - 1])
```

Μπορούμε να αρχίσουμε την προσπέλαση των στοιχείων μετρώντας από το τέλος του αλφαριθμητικού προς την αρχή του αν χρησιμοποιήσουμε το σύμβολο − (μείον) πριν από τον δείκτη (index).

```
# the variable myString points to an immutable string
myString = 'This is a string.'
# the last character of myString
# prints '.'
print(myString[-1])
# prints 'g'.
print(myString[-2])
# prints 'n'.
print(myString[-3])
```

6.2.2 Αντιστροφή Αλφαριθμητικού

Αφού μπορούμε να προσπελάσουμε ένα αλφαριθμητικό ως μια ακολουθία, μπορούμε να χρησιμοποιήσουμε όποια γνωρίσματα μιας ακολουθίας επιθυμούμε για να επιτύχουμε επιθυμητές ενέργειες. Έτσι, για παράδειγμα η αντιστροφή ενός αλφαριθμητικού μπορεί να γίνει πολύ εύκολα.

```
def reverse(text):
    return text[::-1]
```

```
sentence = input('Enter your text: ')

invert = reverse(sentence)

print('The inverted text is:', invert)
```

Όπως βλέπουμε, καθορίζουμε από το τέλος του αλφαριθμητικού μέχρι την να πάρουμε ένα ένα τα γράμματα με την αντίστροφη σειρά. Θα μπορούσαμε αντίστοιχα να αντιστρέψουμε μόνο ένα κομμάτι του αλφαριθμητικού ως εξής:

```
def reversepartially(text):
    return text[3:1:-1]

sentence = input('Enter your text: ')

invert = reversepartially(sentence)

print('The inverted text is:', invert)
```

όπου τώρα πια θα ξεκινήσουμε από τον χαρακτήρα με δείκτη 3, και θα τους τυπώσουμε όλους μέχρι και τον χαρακτήρα με δείκτη 2, ένα πριν δηλαδή. Υπενθυμίζουμε πως στην Python, όπως και σε πολλές άλλες γλώσσες προγραμματισμού, η δεικτοδότηση αρχίζει από το 0.

6.2.3 Μορφοποίηση αλφαριθμητικού

Η μορφοποίηση ενός αλφαριθμητικού μπορεί να γίνει μέσω της συνάρτησης string.format(). Στην python, τα αλφαριθμητικά strings είναι ο προεπιλεγμένος τύπος και δεν χρειάζονται κάποια δήλωση.

Για να μπορέσουμε να δείξουμε ποια μεταβλητή θα πάει σε πια θέση, χρησιμοποιούμε το $\{x\}$, όπου x είναι ένας δείκτης που δείχνει ποια από τις ακόλουθες μεταβλητές θα τοποθετηθεί στην θέση του.

```
sf = 'the {} jumped over the {}!'
print(sf.format('mouse', 'moon'))
```

6.2.4 Συναρτήσεις Αλφαριθμητικών

Μέσω της συνάρτησης repr() μετατρέπουμε ένα αριθμό σε αλφαριθμητικό, και στην συνέχεια μπορούμε να χρησιμοποιήσουμε πάνω του συναρτήσεις που αντιστοιχούν σε αλφαριθμητικά ώστε να τροποποιήσουμε τον τρόπο εμφάνισης του στην οθόνη για παράδειγμα.

Η rjust() κάνει δεξιά στοίχιση σε ένα string, χρησιμοποιώντας τόσους χαρακτήρες, όσοι και προσδιορίζονται στο όρισμα της.

```
for x in range(1, 11):
    print(repr(x).rjust(2), repr(x*x).rjust(3),\
        repr(x*x*x).rjust(4))
```

Ένας, εναλλακτικός τρόπος εμφάνισης του παραπάνω αποτελέσματος, σύμφωνα με τα όσα έχουμε ήδη πει, θα ήταν με την χρήση της συνάρτησης format().

```
for x in range(1, 11):
    print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
```

Μια άλλη χρήσιμη συνάρτηση στα αλφαριθμητικά είναι η split().

```
>>> a = 'asdf'
>>> b = 'asdasdf-nbfds'
>>> a.split('-')
['asdf']
>>> b.split('-')
['asdasdf', 'nbfds']
```

Η συνάρτηση split() χωρίζει ένα αλφαριθμητικό με βάση το όρισμα που δέχεται και είναι και αυτό αλφαριθμητικό. Αν το αλφαριθμητικό που δέχεται βρεθεί σε μια λέξη, επιστρέφεται μια λίστα που περιέχει όλα τα κομμάτια στα οποία χωρίζει την λέξη, αλλιώς επιστρέφεται η λέξη ενιαία σε μια λίστα.

6.2.5 Στατιστικά Εγγράφου

Ένας τρόπος να δούμε συνδυασμένες αρκετές λειτουργίες αλφαριθμητικών, είναι μέσω της δημιουργίας ενός προγράμματος το οποίο μας επιστρέφει τα

στατιστικά ενός εγγράφου απλού κειμένου.

```
# count lines, sentences, and words of a text file

# set all the counters to zero
lines, blanklines, sentences, words = 0, 0, 0, 0

# try to open the file
try:
  filename = 'random_file'
  textf = open(filename, 'r')
except IOError:
  print('Cannot open file {1} for reading', filename)
  # import sys only if needed
  import sys
  # exit the program
  sys.exit(0)

# reads one line at a time
for line in textf:
  # increase line counter
  lines += 1

  # if it is an empty line
  if line.startswith('\n'):
    blanklines += 1
  else:
    # assume that each sentence ends with . or ! or ?
    # so simply count these characters
    sentences += line.count('.') + line.count('!') + \
       line.count('?')

    # create a list of words
    # use None to split at any whitespace regardless of length
```

```
    # so for instance double space counts as one space
    tempwords = line.split(None)

    # word total count
    words += len(tempwords)

# close text file
textf.close()


print("Lines       : ", lines)
print("Blank lines: ", blanklines)
print("Sentences   : ", sentences)
print("Words       : ", words)
```

6.3 Λίστα

Ορισμός 6.3.1. *Λίστα* είναι μια δομή δεδομένων η οποία περιέχει σε μια συγκεκριμένη σειρά μια συλλογή τιμών. Η ίδια τιμή μπορεί να υπάρχει περισσότερες από μια φορές.

Οι λίστες χρησιμοποιούνται αρκετά συχνά στον προγραμματισμό. Αν είστε εξοικειωμένοι με κάποια άλλη γλώσσα, αντικαθιστούν την έννοια του πίνακα που ίσως ήδη γνωρίζετε. Η κύρια διαφορά τους στην Python είναι ότι γίνεται αυτόματα η κατάλληλη δέσμευση της μνήμης που αυτοί απαιτούν. Επομένως δεν πρέπει να τις συγχέουμε με τις διασυνδεδεμένες λίστες που ίσως γνωρίζουμε. Η συμπεριφορά τους έχει όλα τα χαρακτηριστικά των πινάκων.

Για να ορίσουμε μια λίστα, αρκεί να περιλάβουμε μέσα σε αγκύλες τα αντικείμενα που θέλουμε να περιέχει.

```
list1 = ['pick up groceries', 123,\
    'do laundry', 3.14, 'return library books']
# an empty list
list2 = []
```

```
# a list may contain any type of object
list3 = [list1, list2, 1.234, 'This is a diverse list']
print(list3)
```

Όπως βλέπουμε παραπάνω, μια λίστα μπορεί να περιέχει οποιουδήποτε τύπου αντικείμενα. Έτσι, μπορούμε να προσθέσουμε δικά μας αντικείμενα, άλλες λίστες, ακόμα και αντικείμενα που αναπαριστούν συναρτήσεις.

6.3.1 Δημιουργία λίστας

Ο πιο απλός τρόπος να δημιουργήσουμε μια κενή λίστα φαίνεται είναι με την χρήση των τετράγωνων αγκυλών.

```
list1 = []
```

Τέλος, υπάρχουν και λεπτομέρειες που συχνά μας ξεφεύγουν όσον αφορά τις λίστες. Αν χρησιμοποιήσουμε την συνάρτηση list() σε μια λίστα, τότε μας επιστρέφεται ένα αντίγραφο αυτής της λίστας χωρίς να έχει αλλάξει κάτι. Αν όμως περιλάβουμε την παλιά λίστα μέσα σε αγκύλες, τότε δημιουργείται μια καινούργιο που περιέχει ένα αντικείμενο.

```
a = [1,2,3,4]

b = list(a)
c = [a]

print(b)
print(c)
```

Προσοχής χρήζει στην διαφορά που παρατηρείται αν χρησιμοποιούμε την συνάρτηση list() ή όχι. Αυτό γιατί η list() δημιουργεί ένα αντίγραφο της λίστας από προεπιλογή, ενώ αλλιώς η Python επιστρέφει απλώς μια αναφορά για το αντικείμενο που πλέον μπορούμε να το προσπελάσουμε με έναν ακόμα τρόπο.

```
a = [1,2,3,4]
```

```
b = list(a)
c = a

print(b)
print(c)

# b is a copy of a and any change
# on it doesn't affect a
b[0] = 5
print(a)
# c is a reference to a and all
# changes are reflected to a
c[0] = 10
print(a)
```

6.3.2 Πρόσβαση σε στοιχεία λίστας

Η πρόσβαση στα στοιχεία μιας λίστας είναι αρκετά εύκολη, αρκεί να θυμόμαστε πως η μέτρηση, όπως σχεδόν σε όλα στους υπολογιστές, αρχίζει από το μηδέν και όχι από το ένα. Επομένως το πρώτο στοιχείο μιας λίστας αριθμείται με το μηδέν και όλα τα μετέπειτα έχουν δείκτη αυξημένο κατά ένα. Ένα άλλο σημαντικό στοιχείο που πρέπει να θυμόμαστε όταν προσπελαύνουμε στοιχεία όπως φαίνεται στον τρίτο τρόπο, είναι ότι το τελευταίο στοιχείο όταν διαλέγουμε από ένα σύνολο δεν είναι το στοιχείο $list1[j]$ αλλά το $list1[j-1]$. Αυτό αν το σκεφθούμε λίγο είναι λογικό, καθώς αν μια λίστα έχει n στοιχεία, το τελευταίο της δεν είναι το $list1[n]$ αλλά το $list1[n-1]$ αφού η αρίθμηση αρχίζει από το μηδέν.

- Πλήθος στοιχείων

```
n = len(list1)
```

- Πρώτο στοιχείο

```
list1[0]
```

- Στοιχεία στο $[i, j)$:

```
list1[i:j]
```

Εκτός από το να διαλέξουμε από που έως που θα πάρουμε τα στοιχεία μιας λίστα, μπορούμε επίσης να καθορίσουμε και ανά πόσα στοιχεία θα τα λαμβάνουμε.

Αν έχουμε μια λίστα με όλους τους αριθμούς μέχρι το 10 και θέλουμε τους άρτιους από τον δεύτερο αριθμό έως τον όγδοο (χωρίς να τον συμπεριλαμβάνουμε) τότε μπορούμε να κάνουμε ακολούθως, όπου ο τελευταίος αριθμός στις αγκύλες είναι το βήμα:

```
a = [x for x in range(10)]
# or a = list(range(10))
print(a[2:8:2])
```

- Στοιχεία στο $[i, j)$ ανά k

```
list1[i:j:k]
```

- Το k μπορεί να πάρει αρνητικές τιμές

```
list1[::k]
```

Τέλος παρατηρούμε ότι το k, που αποτελεί το βήμα μπορεί να πάρει αρνητικές τιμές, επομένως μπορούμε έτσι εύκολα να κατασκευάσουμε μια συνάρτηση αντιστροφής ενός πίνακα (χρησιμοποιώντας τις ιδιότητες αυτές των ακολουθιών).

```
def reverse(lis):
    return lis[::-1]
```

6.3.3 Διάτρεξη στοιχείων λίστας

Μπορούμε πολύ εύκολα να διατρέξουμε τα στοιχεία μια λίστας ένα ένα χρησιμοποιώντας μια απλή δομή for.

```
a = [1, 1, 2, 3, 5, 8]

for i in a:
    print(i)
```

Αν θέλουμε, μπορούμε να χρησιμοποιήσουμε επαναλήπτες για την διάτρεξη των στοιχείων μιας λίστας, οι οποίοι όπως θα δούμε αποτελούν και μια εισαγωγή στην έννοια των γεννήτορων.

```
a = [1, 1, 2, 3, 5, 8]

b = iter(a)
print(next(b))
print(next(b))
print(next(b))
```

6.3.4 Διαγραφή στοιχείων

Για να διαγράψουμε κάποια στοιχεία από μια λίστα, προσθέτουμε μπροστά την λέξη κλειδί del και στην συνέχεια το κομμάτι της λίστας το οποίο θέλουμε να διαγράψουμε. Αυτό το κομμάτι το προσδιορίζουμε όπως ακριβώς θα το προσδιορίζαμε και αν θέλαμε να κάνουμε προσπέλαση στα στοιχεία αυτού του κομματιού.

```
a = [0, 1, 2, 3, 4, 5]

del a[2:4]
print(a)
```

6.3.5 Κατανοήσεις λίστας (Lists comprehensions)

Μπορούμε επίσης να δημιουργήσουμε μια καινούργια λίστα με την χρήση κατανοήσεων λίστας (list comprehensions). Στο παρακάτω παράδειγμα βλέπουμε μια ήδη υπάρχουσα λίστα της οποίας τα στοιχεία διατρέχουμε ένα ένα, και κάθε ένα το υψώνουμε στο τετράγωνο. Το υψωμένο στο τετράγωνο στοιχείο πλέον ανήκει στην καινούργια λίστα.

```
primes = [1, 2, 3, 5, 7]

square_primes = [i**2 for i in primes]
print(square_primes)
```

Μια list comprehension δημιουργείται μέσω ενός ειδικού συντακτικού με αγκύλες. Πάντα ως αποτέλεσμα του είναι ένα καινούργιο αντικείμενο λίστας. Το γενικό μοντέλο είναι:

```
result = [transform iteration filter]
```

Το κομμάτι του μετασχηματισμού (transform) γίνεται για κάθε φορά που προκύπτει από την διάτρεξη[1] (iteration) και εφόσον ισχύει η συνθήκη φιλτραρίσματος (filter), αν αυτή υπάρχει. Η σειρά με την οποία γίνεται η διάτρεξη είναι συγκεκριμένη και καθορίζει την σειρά εμφάνισης των αποτελεσμάτων στη λίστα.

Ένα παράδειγμα πλήρους χρήσης αυτού του μοντέλου αποτελεί το παρακάτω. Με την χρήση λιστών, πολλές φορές μπορούμε να αποφύγουμε την συγγραφή βρόγχων for, και να δημιουργήσουμε πιο αποδοτικό και ευανάγνωστο κώδικα. Αυτό μπορεί να γίνει αξιοποιώντας τις ειδικές λειτουργίες που μας προσφέρουν οι λίστες. Μπορούμε, για παράδειγμα, να αθροίσουμε όλους τους αριθμούς που περιέχονται σε μια λίστα, μέσω της χρήσης της συνάρτησης sum(). Παρακάτω, αθροίζουμε όλους τους αριθμούς μέχρι το 100 που είναι πολλαπλάσια του 3 και του 7.

```
nums = [i for i in range(1, 101) if i % 3 == 0 or i % 7 == 0]
sum(nums) #2208
```

[1] Διάτρεξη είναι η διαδικασία μέσω της οποία παίρνουμε ένα ένα τα στοιχεία μιας ακολουθίας.

6.3.6 Στοίβα

Με την χρήση μιας λίστας, είναι πολύ εύκολο να κατασκευάσουμε άλλες δομές. Ένα παράδειγμα είναι μια στοίβα.

Ορισμός 6.3.2. Η *στοίβα* έχει δυο βασικές λειτουργίες. Αυτές είναι οι:

- push (ονομάζεται append στην Python)
- pop

Ένα απλό παράδειγμα υλοποίησης μια στοίβας φαίνεται παρακάτω.

```
a = [1, 'TasPython', 'geia']
a.append('xara')
print(a)
b = a.pop()
print(b)
```

6.4 Πλειάδα

Ορισμός 6.4.1. *Πλειάδα* είναι ένα στιγμιότυπο της λίστας. Δεν αλλάζει μέγεθος ούτε και στοιχεία.

Χρήσιμες ιδιότητες της αποτελούν ότι:

- Τα στοιχεία της δεν αλλάζουν
- Χρήσιμη για να επιστρέφουμε πολλές τιμές σε συναρτήσεις

Η δημιουργία μιας πλειάδας είναι πολύ απλή. Αρκεί να χωρίσουμε με κόμμα τα αντικείμενα που θέλουμε και αυτά, αυτομάτως ανήκουν σε μια πλειάδα. Προαιρετικές, και κάνουν πιο ευανάγνωστο τον κωδικό, είναι οι παρενθέσεις γύρω από αυτά τα αντικείμενα.

```
a = (1, 'asdf', 3.14)
print(a)
```

Χρησιμοποιώντας μια πλειάδα, ουσιαστικά δημιουργούμε ένα αντικείμενο το οποίο είναι μια ομάδα άλλων αντικειμένων. Αν για μια λειτουργία μπορούμε να χρησιμοποιήσουμε είτε πλειάδες είτε λίστες, προτιμούμε τις πλειάδες καθώς είναι ο πιο γρήγορος τρόπος.

6.5 Λεξικό

Με το λεξικό (dictionary) μπορούμε να αντιστοιχήσουμε σε λέξεις κλειδιά κάποιες τιμές. Μπορούμε να δούμε το λεξικό ως μια γενίκευση των λιστών[2], όπου αντί να δεικτοδοτούμε ένα αντικείμενο με έναν ακέραιο, μπορούμε να το δεικτοδοτούμε με οποιοδήποτε αντικείμενο, φθάνει αυτό να είναι σταθερό (δηλαδή να μην αλλάζει τιμή) και μοναδικό.

6.5.1 Δημιουργία Λεξικού

Μπορούμε να ενθέσουμε κάποιες τιμές στο λεξικό με την δημιουργία του.

```
d = {
'milk' : 3.67,
'butter' : 1.95,
'bread' : 1.67,
'cheese' : 4.67
}
```

Ακόμα και όταν έχουμε ήδη δημιουργήσει ένα λεξικό μπορούμε εύκολα να ενθέτουμε τιμές σε αυτό. Στο παρακάτω παράδειγμα δημιουργούμε ένα άδειο λεξικό και στην συνέχεια προσθέτουμε μια τιμή.

```
d = {}
d['key'] = value
```

Μπορούμε επίσης να δημιουργήσουμε ένα λεξικό μέσω της μεθόδου dict().

[2]Δεν είναι ακριβώς το ίδιο γιατί χρησιμοποιούνται hash δομές για να επιτευχθεί αυτό το αποτέλεσμα.

```
d = dict([('milk',   3.67),
          ('butter', 1.95),
          ('bread',  1.67),
          ('cheese', 4.67)])
```

6.5.2 Λειτουργίες σε Λεξικό

- *Διαγραφή:* Χρησιμοποιούμε το del. Για παράδειγμα η del d['milk'] διαγράφει την αντίστοιχη εγγραφή από το παραπάνω λεξικό.
- *Μέγεθος:* Η ανάκτηση του μεγέθους ενός λεξικού (αριθμός ζευγαριών κλειδί/τιμή) ανακτάται μέσω της μεθόδου len(d).
- *Κλειδιά:* Για να βρούμε τα κλειδιά ενός λεξικού (χρήσιμο σε βρόγχους) χρησιμοποιούμε την μέθοδο d.keys().
- *Τιμές:* Αντίστοιχα, για τις τιμές μέσω της μεθόδου d.values().
- *Κλειδί/Τιμή:* Για να ανακτήσουμε όλα τα ζευγάρια κλειδί/τιμή από ένα συγκεκριμένο λεξικό d.items() η οποία και μας επιστρέφει πλειάδες από δυο στοιχεία, όπου το πρώτο είναι το κλειδί ακολουθούμενο από την αντίστοιχη τιμή.

Μια λειτουργιά που συναντιέται αρκετά συχνά στην πράξη, είναι να θέλουμε να προσθέσουμε μια τιμή σε μια συγκεκριμένη θέση του λεξικού. Πολλές φορές θέλουμε αν το κλειδί του λεξικού υπάρχει ήδη, να ανακτήσουμε την ήδη υπάρχουσα τιμή και να την ενημερώσουμε με βάση μια συνάρτηση, ενώ αν δεν υπάρχει το συγκεκριμένο κλειδί, να το δημιουργήσουμε και να τοποθετήσουμε την τιμή που θέλουμε. Παρακάτω βλέπουμε δύο τρόπους για την υλοποίηση αυτής της λειτουργίας. Και οι δύο τρόποι έχουν ακριβώς το ίδιο αποτέλεσμα.

```
def increaseValue(d, key, value):
    """ Increase value of d[key].
    Precondition: We can add value to 0.
    """
```

```
    if key not in d:
        d[key] = 0

    d[key] += value


def increaseValue2(d, key, value):
    """ Increase value of d[key].
    Precondition: We can add value to 0.
    """

    # get the value of d[key] if it exists,
    # or zero if it doesn't exist.
    # Then add it to value and set it
    # as the new value of d[key]
    d[key] = d.get(key, 0) + value

# testing
d = {}
increaseValue2(d, 'dimitris', 5)
print(d)
```

Η πρώτη συνάρτηση κοιτάζει αν υπάρχει το κλειδί μέσα στο λεξικό. Αν δεν υπάρχει, το δημιουργεί και του δίνει μια αρχική τιμή (μηδέν στην περίπτωση μας). Στην συνέχεια προσθέτει στην τιμή του λεξικού που ανακτά την τιμή που έχουμε περάσει στο όρισμα της συνάρτησης (value).

Η δεύτερη συνάρτηση επιτυγχάνει το ίδιο αποτέλεσμα με έναν πιο κομψό τρόπο. Χρησιμοποιούμε την μέθοδο get που εφαρμόζεται σε ένα λεξικό. Αυτή μας επιστρέφει την τιμή του d[key] ή την τιμή που προσδιορίζουμε με τον δεύτερο ορισμό εφόσον δεν υπάρχει αυτό το κλειδί στο λεξικό. Στην συνέχεια προσθέτουμε την τιμή της value στην ανακτηθείσα τιμή και το αποτέλεσμα το εκχωρούμε στο d[key] έχοντας κάνει την ενημέρωση που επιθυμούσαμε.

6.5.3 Διάτρεξη τιμών

Η διάτρεξη των τιμών σε ένα λεξικό γίνεται όπως θα κάνουμε και σε μια λίστα. Μόνο που εδώ πρέπει να προσέξουμε ότι δεν είναι εγγυημένη η σειρά με την οποία θα διατρεχθούν τα στοιχεία του λεξικού, και για αυτό τον λόγο δεν πρέπει να βασιζόμαστε σε αυτή.

```
d = {
'milk' : 3.67,
'butter' : 1.95,
'bread' : 1.67,
'cheese' : 4.67
}
for food in d:
    print('{} costs {}'.format(food, d[food]))
```

Οι items() επιστρέφεις πλειάδες κλειδί, τιμή αντίστοιχα για κάθε θέση του λεξικού.

```
thoughts = {'Omada': 'TasPython',\
    'Ti einai o Anthropos': 'Command not found'}
for k, v in thoughts.items():
    print(k, v)
```

6.5.4 Αναφορά και Τροποποίηση

Τα λεξικά είναι τροποποιήσιμα αντικείμενα (mutable). Για αυτό τον λόγο απαιτείται προσοχή όταν τροποποιούμε ένα λεξικό καθώς όλες οι αναφορές σε αυτό το λεξικό τροποποιούνται επίσης. Αν επιθυμούμε την αποφυγή μιας τέτοιας συμπεριφοράς, τότε πρέπει να χρησιμοποιούμε την μέθοδο copy().

```
d['a'] = 123
c = d
c['a'] = 1821
print(d['a'])
```

Σε αντιδιαστολή με:

```
d = {}
d['a'] = 123
c = d.copy()
c['a'] = 1821
print(d['a'])
```

6.5.5 Κατανοήσεις λεξικού (Dict comprehension)

Αντίστοιχα με τις κατανοήσεις λίστας, υπάρχουν και κατανοήσεις λεξικών. Η διαφορά, πέρα από την προφανή συντακτική διαφορά με τις {} έγκειται στο γεγονός ότι τώρα πρέπει να χρησιμοποιούμε δυο διατρέξιμα αντικείμενα (iterable), ομαδοποιημένα. Ακολουθούν δυο παραδείγματα που κάνουν ακριβώς το ίδιο πράγμα.

```
d = {k:v for k,v in enumerate('taspython') if v not in 'tas'}
print(d)      # {8: 'n', 3: 'p', 4: 'y', 6: 'h', 7: 'o'}
```

```
d = {k:v for k,v in zip(range(15), 'taspython') if v not in 'tas'}
print(d)      # {8: 'n', 3: 'p', 4: 'y', 6: 'h', 7: 'o'}
```

Αν προσέξετε στο αποτέλεσμα που παρουσιάζεται, δεν τυπώνονται σε σειρά τα γράμματα της λέξης Python. Αυτό συμβαίνει γιατί η δομή του λεξικού δεν εγγυάται κάποια σειρά. Επίσης, αξίζει να προσέξουμε πως η διάτρεξη σταματάει ότι τελειώσουν τα στοιχεία ενός από όλα τα αντικείμενα που διατρέχουμε.

6.5.6 Ταξινομημένο Λεξικό

Η ευρέως χρησιμοποιούμενη δομή ελέγχου του λεξικού, δεν εξασφαλίζει κάποια σειρά στα ζευγάρια κλειδί/τιμή που αποθηκεύονται. Έτσι, δυσχεραίνεται η χρήση λεξικών σαν μέσα αποθήκευσης σε ορισμένες περιπτώσεις. Ορισμένες δυναμικές γλώσσες μπορούν και εγγυώνται μια συγκεκριμένη σειρά προσπέλασης των στοιχείων στις αντίστοιχες δομές. Η σειρά καθορίζεται από τον χρόνο όπου εισήχθηκε κάθε κλειδί. Καινούργια κλειδιά τοποθετούνται στο τέλος του λεξικού. Ωστόσο, κλειδιά στα οποία ανανεώθηκε η τιμή

τους, δεν αλλάζουν θέση.

Η δομή του ταξινομημένου λεξικού είναι ουσιαστικά ένα λεξικό (με την έννοια ότι παρέχει την ίδια λειτουργικότητα), μόνο που έχει και την ιδιότητα που περιγράψαμε παραπάνω.

```
from collections import OrderedDict

d = OrderedDict()
d['python'] = 'TasPython'
d['guide'] = 'Greek'
print(d.items())
```

6.6 Σύνολο

Τα σύνολα μας διευκολύνουν στην ομαδοποίηση πολλών αντικειμένων και στην εφαρμογή στην συνέχεια πράξεων όπως η ένωση τους με αποδοτικό τρόπο, εξασφαλίζοντας πως κάθε στοιχείο, αν περιέχεται σε πάνω από ένα σύνολα, τελικά θα βρεθεί μόνο μια φορά στο τελικό αποτέλεσμα. Πρόκειται για μια δομή δεδομένων που αναπαριστάται ως ένα μη διατεταγμένο σύνολο μοναδικών στοιχείων.

6.6.1 Δημιουργία

Για την απευθείας δημιουργία ενός συνόλου χρησιμοποιούμε τις αγκύλες. Η Python καταλαβαίνει πως δεν είναι λεξικό, αλλά αναφερόμαστε σε σύνολο επειδή δεν υπάρχει η άνω κάτω τελεία η οποία υποδηλώνει την τιμή που αντιστοιχίζεται σε κάθε κλειδί.

```
s = {'a', 'a', 'b', 'c', 'a'}
print(s)
```

Μπορούμε επίσης να δημιουργήσουμε ένα λεξικό από οποιοδήποτε τύπο δεδομένων φθάνει να μπορούμε να διατρέξουμε τα στοιχεία αυτού του τύπου (iterable).

```
s = set('TasPython ...because simplcity matters!')
print(s)
```

6.6.2 Βασικές Πράξεις Συνόλων

Όπως γνωρίζουμε και από τα μαθηματικά, οι βασικές πράξεις συνόλων είναι:

- Πρόσθεση στοιχείου σε ένα σύνολο
- Αφαίρεση στοιχείου από ένα σύνολο
- Ένωση Συνόλων
- Τομή Συνόλων
- Διαφορά Συνόλων
- Συμμετρική Διαφορά Συνόλων

οι οποίες παρουσιάζονται στο παρακάτω παράδειγμα.

```
a = set('abracadabra')
b = set('alacazam')
print('A = ', a)
print('B = ', b)
a.add('z')                  # add element
b.remove('z')               # remove element
print('A = ', a)
print('B = ', b)
print('A - B = ', a - b)    # difference
print('A | B = ', a | b)    # union
print('A & B = ', a & b)    # intersection
print('A ^ B = ', a ^ b)    # symmetric difference
```

Τέλος ένα παράδειγμα όπου φαίνονται και κάποιες επιπλέον λειτουργίες ακολουθεί:

```
a = ['apple', 'bread', 'carrot', 'carrot']
set1 = set(a)
print(set1)
print('apple' in set1)
set2 = set1.copy()
set2.add('delicious')
print(set1 < set1)
set1.remove('bread')
# prints {'carrot', 'apple'}
print(set1 & set2)
# prints {'carrot', 'apple', 'delicious', 'bread'}
print(set1 | set2)
```

Κεφάλαιο 7

Αξία ή Αναφορά

> Η φύσις και η διδαχή παραπλήσιον εστι· και γάρ η διδαχή μεταρρυσμοί, μεταρρυσμούσα δε φυσιοποιεί.
>
> Δημόκριτος

Η συγκεκριμένη ενότητα είναι λίγο πιο προχωρημένη και αναφέρεται στο πως χειρίζεται η Python τα αντικείμενα. Είναι αρκετά πιθανό να αργήσει να σας χρειαστεί, όμως παρέχει χρήσιμες γνώσεις σχετικά με 'μυστήρια' προβλήματα που μπορεί να εμφανιστούν αλλά και το απαραίτητο οπλοστάσιο για έξυπνες και αποδοτικές υλοποιήσεις ιδιαίτερα σύνθετων δομών σε Python.

7.1 Απλοί τύποι (immutable objects)

Στην Python ακόμα και οι απλοί τύποι που βλέπουμε παρακάτω είναι αντικείμενα με τις δικές τους μεθόδους και κλάσεις. Παρόλα αυτά έχουν μια ιδιαιτερότητα. Από την στιγμή της δημιουργίας τους δεν μπορούμε να αλλάξουμε την τιμή τους, επομένως αν επιθυμούμε κάτι τέτοιο πρέπει να δημιουργήσουμε ένα καινούργιο αντικείμενο.

- bool
- int

- float
- string

Κάθε μεταβλητή που αναφέρεται σε διαφορετική τιμή ή ακόμα περισσότερο σε διαφορετικό απλό τύπο, αποθηκεύεται σε διαφορετική θέση μνήμης. Με την χρήση της συνάρτησης id() μπορούμε να βρούμε την θέση μνήμης[1] όπου έχει αποθηκευτεί ένα αντικείμενο. Ο τελεστής *is* συγκρίνει τις θέσεις δυο μεταβλητών και μας επιστρέφει *True* αν δείχνουν στις ίδιες θέσεις μνήμης.

```
>>> a = 5
>>> b = 5
>>> c = 5.0
>>> d = '5'
>>> id(a)
137061184
>>> id(d)
3073418688
>>> a is b
True
>>> c is d
False
```

Ωστόσο, ίδιοι πρωταρχικοί τύποι που δείχνουν ακριβώς στα ίδια περιεχόμενα, δείχνουν και στην ίδια θέση μνήμης όπως είδαμε παραπάνω αφού *a is b* επιστρέφει αληθές.

Άλλο ένα σημείο που πρέπει να προσέξουμε είναι ότι για τα παραπάνω αντικείμενα δημιουργούνται αυτόματα ψευδώνυμα όταν έχουν την ίδια τιμή αλλά στην συνέχεια μια αλλαγή σε ένα δεν επηρεάζει τα υπόλοιπα. Αυτό γίνεται καθώς η αλλαγή συνεπάγεται την δημιουργία ενός καινούργιου αντικειμένου και την μετακίνηση του δείκτη να δείχνει προς αυτό το αντικείμενο, χωρίς όμως παράλληλη μετακίνηση των υπόλοιπων αναφορών που προϋπήρχαν για το προηγούμενο αντικείμενο.

[1] Για να είμαστε πιο ακριβείς, μας επιστρέφει έναν μοναδικό αριθμό που χαρακτηρίζει ένα αντικείμενο, που όμως λόγω υλοποίησης, αυτός αποτελεί την διεύθυνση του.

```
>>> a = 4
>>> b = 4
>>> a is b
True
>>> a += 1
>>> a
5
>>> b
4
```

7.2 Τοπικές και καθολικές μεταβλητές

Ας δούμε, αρχικά, τι είναι οι καθολικές και οι τοπικές μεταβλητές.

Ορισμός 7.1. *Μια τοπική μεταβλητή είναι προσβάσιμη μόνο στο τοπικό εύρος.*

Ορισμός 7.2. *Μια καθολική μεταβλητή είναι προσβάσιμη σε κάθε εύρος (scope).*

Μια άλλη χρήσιμη έννοια είναι αυτή του εύρους. Συνήθως, έχουμε το εύρος μια συνάρτησης, μια κλάσης, ενός αρχείου κοκ που ορίζεται από κάποιο μπλοκ κώδικα που ανήκει στο ίδιο μπλοκ.

Ορισμός 7.3. *Εύρος (scope) το πλαίσιο ενός προγράμματος στο οποίο το όνομα μια μεταβλητής είναι έγκυρο και μπορεί να χρησιμοποιηθεί. (scope).*

Στην Python, μπορούμε να προσπελάσουμε τα περιεχόμενα μιας καθολικής μεταβλητής από οπουδήποτε. Αντίθετα, μπορούμε να έχουμε πρόσβαση σε μια τοπική μεταβλητή μόνο μέσα στα πλαίσια του εύρους (στο παράδειγμα μας συνάρτησης) όπου έχει οριστεί. Αυτός είναι και ο λόγος που συνήθως καθολικές μεταβλητές βρίσκονται στο πάνω μέρος του αρχείο του κώδικα.

```
g = "I am a global variable"

def function1(arg):
```

```
        a = "I am a local variable"
        print(g)
        # g += ' modified' # produces an error
        print(arg)

#print(a) # produces an error

function1("A reference to me is passed.")
```

Πιο συγκεκριμένα, στο παραπάνω παράδειγμα, η μεταβλητή `g` είναι καθολική. Αυτό σημαίνει ότι μπορούμε να προσπελάσουμε τα περιεχόμενα της μέσα στην συνάρτηση `function1` αλλά και εκτός. Αντίθετα, η μεταβλητή `a` ορίζεται μόνο μέσα στα πλαίσια αυτής της συνάρτησης, και επομένως η προσπέλαση της έχει νόημα μόνο εντός αυτής. Οποιαδήποτε προσπέλαση της συγκεκριμένης μεταβλητής `a` εκτός της συνάρτησης `function1` θα οδηγήσει σε σφάλμα.

Μια άλλη διαφορά που έχουν οι τοπικές και οι καθολικές μεταβλητές είναι τα δικαιώματα τροποποίησης τους. Επειδή γενικά είναι επικύνδινο να τροποποιούμε καθολικές μεταβλητές, η Python δε μας επιτρέπει να τις τροποπιούμε εκτός και αν δηλώσουμε ρητά την επιθυμία μας για το αντίθετο. Αυτό μπορεί να γίνει με τη λέξη κλειδί `global`.

```
g = "I am a global variable"

def function1(arg):
        global g
        a = "I am a local variable"
        print(g)
        g += ' modified'
        print(g)
        print(arg)

function1("A reference to me is passed.")
```

Ιδιαίτερη προσοχή θέλει ότι σε περίπτωση που προσπαθήσουμε να θέσου-

με μια καθολική μεταβλητή χωρίς να χρησιμοποιήσουμε τη λέξη κλειδί `global`, τότε δημιουργούμε μια καινούργια τοπική μεταβλητή χωρίς να επηρρεάζουμε τα περιεχόμενα της καθολικής μεταβλητής.

```
g = "I am a global variable"

def function1(arg):
    g = "I thought I was global"
    print(g)

function1("A reference to me is passed.")

print(g)
```

Τέλος, έχουμε την περίπτωση όπου μέσα σε μια συνάρτηση ορίζουμε μια άλλη συνάρτηση. Σε αυτή την περίπτωση, πρέπει να χρησιμοποιήσουμε τη λέξη κλειδί `nonlocal` αντί για το `global` προκειμένου να μπορούμε να τροποποιήσουμε τα περιεχόμενα της μεταβλητής. Κατά τα άλλα, κινούμαστε σε παρόμοια πλαίσια.

```
def function1():
    a = "Scope1 variable"
    def function2():
        nonlocal a
        a += " modified"
        print(a)
        function2()

function1()
```

Η λέξη κλειδί `nonlocal` γενικά χρησιμοποιείται για να αναφερθούμε στο κοντινότερο εύρος στο οποίο περιέχεται ο κώδικας τον οποίο εκτελούμε.

7.3 Σύνθετα αντικείμενα

Σε αυτή την ενότητα θα αναφερθούμε στα υπόλοιπα αντικείμενα στην Python που περιέχουν άλλα αντικείμενα (container types). Μερικά από αυτά είναι:

- Πλειάδες (tuple)
- Αλφαριθμητικά (string)
- Λίστες (list)
- Σύνολα (set)
- Λεξικά (dictionary)

Στην ίδια κατηγορία μπορούμε να εντάξουμε αντικείμενα από κλάσεις που φτιάχνουμε εμείς.

Αυτά καταλαμβάνουν διαφορετική θέση μνήμης ακόμα και αν περιέχουν αντικείμενα με τις ίδιες τιμές. Μόνη εξαίρεση είναι αν μια μεταβλητή αποτελεί ψευδώνυμο (alias) μιας άλλης, όπως θα δούμε παρακάτω.

```
>>> a = 'Python by example'
>>> b = 'Python by example'
>>> a is b
False
>>> a = 1
>>> b = 2
>>> a is b
False
>>> b = 1
>>> a is b
True
```

Εδώ αξίζει να επισημάνουμε και την διαφορά του τελεστή *is* από τον τελεστή ==. Ο πρώτος ελέγχει αν δυο μεταβλητές αναφέρονται σε ακριβώς το ίδιο αντικείμενο[2] ενώ ο τελεστής == ελέγχει μόνο τα περιεχόμενα των δυο αντικειμένων.

[2] Δεν αρκεί τα περιεχόμενα τους να είναι ίδια, αλλά πρέπει και η θέση μνήμης να είναι η ίδια.

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

7.3.1 Ψευδώνυμα

Ψευδώνυμο (alias) είναι όταν αναφερόμαστε σε ένα αντικείμενο με ένα διαφορετικό όνομα, χρησιμοποιώντας μια διαφορετική μεταβλητή για να αναφερθούμε σε αυτό. Όταν γίνεται κάτι τέτοιο, ό,τι αλλαγές γίνουν, ανεξάρτητα από τον τρόπο αναφοράς στο συγκεκριμένο αντικείμενο, θα επηρεάσουν όλες τις μεταβλητές που αναφέρονται σε αυτό.

```
a = [1, 2, 3]
b = a
a.append(4)
print(b)
```

Παρατηρούμε δηλαδή ότι η προσθήκη του αριθμού 4 στην λίστα που έδειχνε το a, επηρεάζει και την λίστα b, αφού στην ουσία το b είναι ένα ψευδώνυμο του a και αναφέρονται ακριβώς στο ίδιο αντικείμενο.

7.3.2 None

Το *None* είναι ένας ειδικός τύπος ο οποίος αναπαριστά την ιδέα ότι δεν υπάρχει κάτι. Σε άλλες γλώσσες η ίδια έννοια συχνά απαντάται από το *Null*. Η τιμή Null στην Python δεν ισούται με καμία άλλη τιμή παρά μόνο με τον εαυτό της και ο μόνος τρόπος για να αναφερθούμε σε αυτή είναι μέσω της λέξης κλειδί None.

```
def helloWorld():
    print('Hello World')
```

```
print(helloWorld() == None)
```

Η τιμή None είναι η τιμή που μας επιστρέφουν συναρτήσεις που τελικά δεν επιστρέφουν κάτι. Αφού όπως είπαμε η τιμή None αντιπροσωπεύει την έννοια του δεν υπάρχει κάτι, η συγκεκριμένη συμπεριφορά είναι κάτι που θα έπρεπε να περιμένουμε.

Σημείωση: Η συνάρτηση θα μπορούσε να επιστρέφει την τιμή None. Τώρα όμως είμαστε στην περίπτωση όπου επειδή δεν επιστρέφεται κάτι, βρίσκουμε αυτή την τιμή. Αυτό οδηγεί σε μια σχεδιαστική αρχή όπου συνήθως, όταν οι συναρτήσεις επηρεάζουν τα ορίσματα τους, επιστρέφουν την τιμή None έτσι ώστε αφού ο χρήστης να βλέπει ρητά πως τα αποτελέσματα της συνάρτησης αποθηκεύονται μέσα στο αντικείμενο.

```
p = [1, 2, 3, 4]
print(p)
# illegal statement because the returned value is None
#p = p.pop()
p.pop()
print(p)
```

7.4 Χώρος Ονομάτων

Ο χώρος ονομάτων ουσιαστικά αντιστοιχεί σε ένα 'περιβάλλον' για τα αντικείμενα που περιέχει και μας επιτρέπει να αντιληφθούμε (και εμείς και ο υπολογιστής) που αναφέρεται μια μεταβλητή. Ονόματα μέσα σε έναν χώρο ονομάτων δεν μπορούν να έχουν πάνω από νόημα.

Στην Python, για να απλοποιήσουμε τα πράγματα, μπορούμε να σκεφθούμε ότι παρέχεται ένας χώρος ονομάτων για κάθε αρχείο. Έτσι, όταν κάνουμε:

```
from file import *
```

εισάγουμε όλες τις μεταβλητές από το αρχείο python.py στον χώρο ονομάτων.

Απλοποιώντας κάποια πράγματα, μπορούμε να σκεφθούμε ότι κάθε φορά που κάνουμε στοίχηση σε κώδικα προς τα μέσα, δημιουργείται ένας καινούρ-

γιος χώρος ονομάτων. Σε αυτό το χώρο ονομάτων ανήκουν όλες οι μεταβλητές που δηλώνονται για πρώτη φορά εκεί. Αν κάποια μεταβλητή δε βρεθεί εκεί, η αναζήτηση συνεχίζεται στο χώρο ονομάτων που υπάρχει ένα επίπεδο πριν (μικρότερη στοίχηση προς τα μέσα) μέχρι να φθάσουμε στο τέρμα αριστερά επίπεδο.

Ορισμός 7.4.1. Ένα *άρθρωμα* (module) είναι ένα αρχείο που περιέχει δηλώσεις και ορισμούς (συναρτήσεις, κλάσεις κτλ). Το όνομα του αρθρώματος είναι το όνομα του αρχείου χωρίς την κατάληξη .py. Οι δηλώσεις που βρίσκονται στο πιο εξωτερικό μπλοκ κώδικα (πρακτικά αυτές οι οποίες δεν προηγούνται από κενά) εκτελούνται από πάνω προς τα κάτω την πρώτη φορά που το άρθρωμα εισάγεται κάπου, αρχικοποιώντας μεταβλητές και συναρτήσεις (πχ ορίζοντας τες). Ένα άρθρωμα μπορεί να εκτελεστεί απευθείας (πχ python file.py) είτε να εισαχθεί import και να χρησιμοποιηθεί από κάποιο άλλο άρθρωμα. Όταν ένα αρχείο Python εκτελείται απευθείας, η ειδικού σκοπού μεταβλητή "__name__" ορίζεται απευθείας με την τιμή "__main__". Για το λόγο αυτό αρκετά συχνά ελέγχεται η τιμή της μεταβλητής "__name__" ώστε να διαπιστωθεί αν το συγκεκριμένο άρθρωμα εκτελείται απευθείας ή έχει εισαχθεί σε κάποιο άλλο για να το που παρέχει τη λειτουργικότητα που αυτό υποστηρίζει.

Αν έχουμε χρησιμοποιήσει τη δήλωση from module import * που είδαμε πριν, τότε δε πρέπει να ξεχνάμε ότι θα πρέπει να έχουμε συνεχώς στο νου μας και για τις μεταβλητές που είχαν δημιουργηθεί εκεί. Για αυτό το λόγο καλό είναι να αποφεύγεται αυτή τη δήλωση και να προτιμάμε είτε το from module import function που εισάγει μόνο μια συνάρτηση (ή και κλάση μπορούμε) είτε να κάνουμε import module, οπότε και στη συνέχεια για όλα τα αντικείμενα του module, θα αναφερόμαστε μέσω του module.variable, δηλαδή θα βάζουμε στην αρχή το όνομα του αρχείου (πιο επίσημα άρθρωμα όπως θα δούμε και παρακάτω) ακολουθούμενο από μια τελεία και το αντικείμενο/μεταβλητή που θέλουμε.

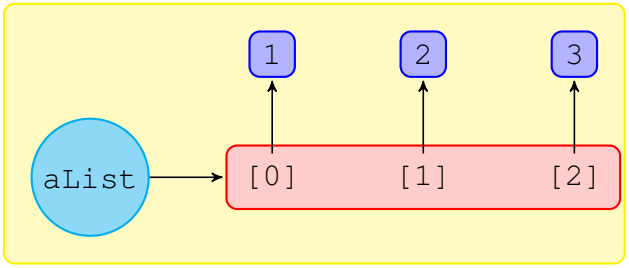
Παρακάτω βλέπουμε μια σχηματική αναπαράσταση των εννοιών που είδαμε μέχρι στιγμής. με τη σύμβαση:

- Πράσινα *αλφαριθμητικά*.

- Κόκκινες *λίστες*.
- Καφέ *λεξικά*.
- Μπλε *ακέραιοι*.

Τα σχήματα που ακολουθούν αντιστοιχούν δείχνουν πως θα έχει διαμορφωθεί το namespace μετά από κάθε μία από τις εντολές που ακολουθούν:

```
List1 = [1, 2, 3]
List2 = [1, 2, 3]
List2[2] = "new"
List1 = {"a" : [1], "b" : [2]}
```

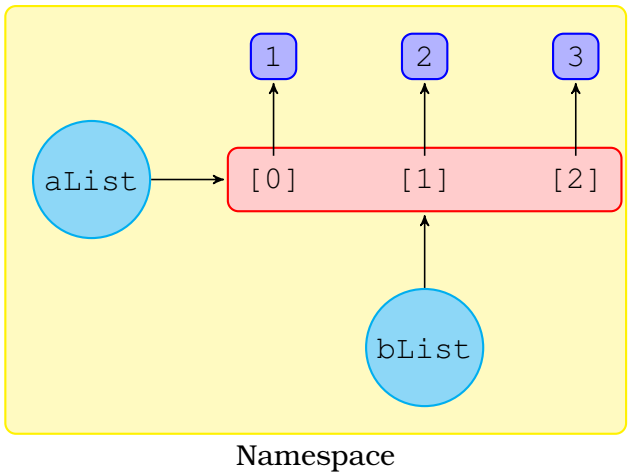


Σχήμα 7.1: Ο χώρος ονομάτων (namespace) μετά τη δημιουργία μίας λίστας.

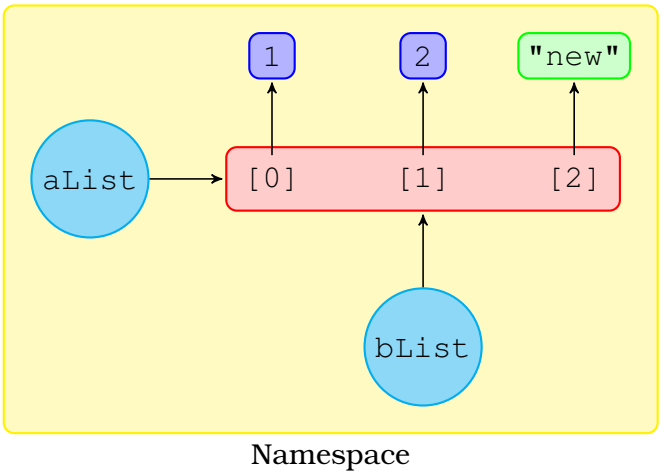
7.5 Εμβέλεια

Άλλη μια σημαντική παράμετρος ως προς τις μεταβλητές που πάντα πρέπει να έχουμε υπόψη μας είναι η εμβέλεια τους.

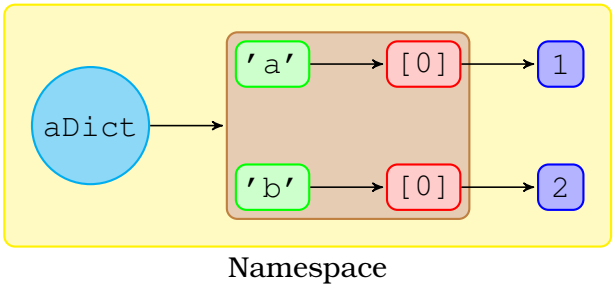
```
>>> def f(a):
...     a = 5
...
>>> a = 4
>>> f(a)
>>> a
4
```



Σχήμα 7.2: Ο χώρος ονομάτων μετά τη δημιουργία και μίας δεύτερης λίστας που δείχνει στην πρώτη.



Σχήμα 7.3: Η αλλαγή των στοιχείων αλλάζει και τις δύο λίστες.



Σχήμα 7.4: Ο χώρος ονομάτων μετά τη δημιουργία ενός λεξικού που περιέχει λίστες.

Η μεταβλητή a μέσα στη συνάρτηση f επειδή βρίσκεται στο αριστερό μέλος μιας ανάθεσης, 'κρύβει' την μεταβλητή που δεχόμαστε ως όρισμα. Έτσι, η ανάθεση $a = 5$ ισχύει μόνο τοπικά και δεν επηρεάζει το περιβάλλον εκτέλεσης. Αν σας μπερδεύει κάτι τέτοιο μια απλή λύση είναι να μην χρησιμοποιείται στο σώμα μιας συνάρτησης, στο αριστερό μέρος μεταβλητές με το ίδιο όνομα με τα ορίσματά σας.

Στο παρακάτω παράδειγμα, το αντικείμενο λίστας L καλείται κατά αναφορά. Έτσι, κατά αρχάς το x δείχνει στην ίδια διεύθυνση που έδειχνε και το L. Στην συνέχεια όμως, δημιουργείται ένα καινούργιο αντικείμενο και το x πλέον δείχνει σε αυτό. Για αυτόν το λόγο και αλλάζει η διεύθυνση όπου δείχνει πια. Τέλος, η συνάρτηση τερματίζει, και το L δείχνει πια στην αρχική του θέση, και έχει τις ίδιες τιμές με πριν κληθεί η συνάρτηση.

Παρατηρούμε, με άλλα λόγια, πως αρχικά περνιέται μια αναφορά του αντικειμένου μέσα στην συνάρτηση. Όμως μέσα στην συνάρτηση δημιουργείται ένα νέο αντικείμενο αφού οι λίστες δεν αναφέρονται με ψευδώνυμα, και το x τώρα πια δείχνει σε αυτό. Μόλις τερματίσει η συνάρτηση, η μεταβλητή L που η τιμή της δεν έχει αλλάξει από την κλήση της συνάρτησης *foo()* συνεχίζει να δείχνει στο ίδιο αντικείμενο το οποίο και δεν έχει τροποποιηθεί από την συνάρτηση.

```
def foo(x):
  print('point 2:', end=' ')
  print(id(x))
  x = [1, 2, 3]
  print('point 3:', end=' ')
  print(id(x))
  print(x)

L = [3, 2, 1]
print('point 1:', end=' ')
print(id(L))
foo(L)
print('point 4:', end=' ')
print(id(L))
```

```
print(L)
```

Τώρα γίνεται ακόμα πιο ξεκάθαρο γιατί το όνομα του ορίσματος στην συνάρτηση δεν παίζει κάποιο ρόλο και γιατί θα μπορούσαμε να το έχουμε αντικαταστήσει με οποιαδήποτε άλλη έγκυρη τιμή, όπως και με το L ακόμα. Ένας πρακτικός κανόνας για να αναγνωρίζουμε σε ποιο L σε αυτή την περίπτωση αναφερόμαστε, είναι στο να έχουμε υπόψη μας αυτό που βρίσκεται πιο κοντά.

Αξίζει να σημειώσουμε πως αν η *foo()* δεν έπαιρνε όρισμα, και την καλούσαμε χρησιμοποιώντας μέσα στο σώμα της την μεταβλητή L, τότε η python δεν θα έλεγχε αν τυχόν υπάρχει από το εξωτερικό περιβάλλον μια μεταβλητή με το όνομα L αλλά θα έβγαζε error κατευθείαν λέγοντας πως δεν βρέθηκε μεταβλητή με το συγκεκριμένο όνομα μέσα στην εμβέλεια της συνάρτησης. Αυτή η αλλαγή σε σχέση με τις προηγούμενες εκδόσεις της Python συνέβη για την αποφυγή δύσκολων προς εντοπισμό λαθών που θα μπορούσαν να συμβούν με αυτό τον τρόπο.

```
def foo():
  print('point 2:', end=' ')
  # Error. L is not declared in this scope
  print(id(L))
  L = [1, 2, 3]
  print('point 3:', end=' ')
  print(id(L))
  print(L)

L = [3, 2, 1]
print('point 1:', end=' ')
print(id(L))
foo()
print('point 4:', end=' ')
print(id(L))
print(L)
```

Συνοψίζοντας λοιπόν μέχρι στιγμής καταλήγουμε στο συμπέρασμα ότι: Αναθέσεις σε τοπικές μεταβλητές μέσα σε μια συνάρτηση δεν επηρεάζουν τις τιμές των μεταβλητών που είναι δηλωμένες έξω από αυτή την συνάρτηση.

Αντίθετα όμως, όπως βλέπουμε στο ακόλουθο παράδειγμα, αν μια μεταβλητή αναφέρεται σε αντικείμενο που έχει δημιουργηθεί έξω από την συνάρτηση που βρισκόμαστε τώρα, και εφόσον αυτή η αναφορά δεν αλλάζει, οποιαδήποτε αλλαγή κάνουμε σε αυτό το αντικείμενο, επηρεάζεται και εξωτερικά της συνάρτησης, αφού αναφερόμαστε στις ακριβώς ίδιες θέσεις μνήμης.

```
def foo(x):
  print('point 2:', end=' ')
  print(id(L))
  x.append(0)
  print('point 3:', end=' ')
  print(id(x))
  print(x)

L = [3, 2, 1]
print('point 1:', end=' ')
print(id(L))
foo(L)
print('point 4:', end=' ')
print(id(L))
print(L)
```

7.6 Αντιγραφή αντικειμένων

Όπως έχουμε ήδη δει, ανάλογα με τον τύπο του αντικειμένου ο τελεστής ανάθεσης μπορεί να δημιουργήσει μόνο μια καινούργια αναφορά σε μια μεταβλητή και να μην αντιγράψει τα περιεχόμενα του αντικείμενου. Αν θέλουμε κάτι τέτοιο, πρέπει να χρησιμοποιήσουμε τη συνάρτηση deepcopy.

Στο επόμενο παράδειγμα θα δούμε πως μπορούμε να εκμεταλλευτούμε την ευελιξία που μας παρέχει η Python με τις αναφορές αλλά και όταν χρειαστεί να δημιουργούμε καινούργια αντίγραφα των αντικειμένων.

```
from copy import deepcopy

def access_trie(d, sequence, position=None):
    """
    Access the dictionary which is referred by applying
    consequently each term of the sequence.  In a more python
    terms, if sequence is 'key', access: d['k']['e']['y'].
    Assume that the dictionary is at the `position` of a list,
    if `position` is an argument.

    >>> a = {'k': [0, {'a': [0, {'l': [0, {'o': [1, {}]}]}]}]}
    >>> access_trie(a, 'kal', 1)
    {'o': [1, {}]}
    >>> access_trie(a, 'kalo', 1)
    {}
    >>> a = {'p': {'y': {'t': {'h': {'o': {'n': {}}}}}}}
    >>> access_trie(a, 'pyt')
    {'h': {'o': {'n': {}}}}
    >>> access_trie(a, '')
    {'p': {'y': {'t': {'h': {'o': {'n': {}}}}}}}
    >>> access_trie(a, 'python')
    {}
    >>> b = access_trie(a, 'pyth')
    >>> b['O'] = '123'
    >>> a
    {'p': {'y': {'t': {'h': {'O': '123', 'o': {'n': {}}}}}}}

    """

    for c in sequence:
        d = d[c]
        if position is not None:
            d = d[position]
```

```
    return d


def populate_trie(trie, sequence, position=None):
    """
    Populate a trie.

    Assume that the counter is always at `position` 0 while the
    `position` of the dictionary is the last one.

    >>> trie = {}
    >>> populate_trie(trie, 'python')
    {'p': {'y': {'t': {'h': {'o': {'n': {}}}}}}}
    >>> trie = {}
    >>> populate_trie(trie, 'kalo', 1)
    {'k': [1, {'a': [1, {'l': [1, {'o': [1, {}]}]}]}]}
    >>> trie = {}
    >>> populate_trie(trie, 'heh', 2)
    {'h': [1, 0, {'e': [1, 0, {'h': [1, 0, {}]}]}]}
    >>> trie = {}
    >>> trie = populate_trie(trie, 'heh', 1)
    >>> populate_trie(trie, 'ha', 1)
    {'h': [2, {'a': [1, {}], 'e': [1, {'h': [1, {}]}]}]}


    """

    if (position is not None) and (position >= 1):
        embedded_obj = [0] * position
        embedded_obj.append({})
    else:
        embedded_obj = {}
```

```
    d2 = trie
    for i, character in enumerate(sequence):
        d2 = access_trie(trie, sequence[:i], position)
        if character not in d2:
            if position is None:
                d2[character] = deepcopy(embedded_obj)
            else:
                d2[character] = d2.get(character,\
                    deepcopy(embedded_obj))
                d2[character][0] += 1
        elif position is not None:
            d2[character][0] += 1
    return trie
```

Στην πρώτη συνάρτηση *access_trie()* βλέπουμε πως χρησιμοποιούμε τις αναφορές που δημιουργούμε με τον τελεστή της ανάθεσης ώστε να μπορέσουμε να προσπελάσουμε κάθε εμφωλευμένο λεξικό αφού έχουμε πρώτα προσπελάσει αυτό που το περιέχει. Για παράδειγμα, ας υποθέσουμε πως το όρισμα position είναι None, που σημαίνει πως ο κώδικας μέσα στο if δεν εκτελείται ποτέ, επομένως μπορούμε να επικεντρωθούμε στις υπόλοιπες δυο γραμμές του κώδικα.

Προσπελαύνουμε ακολουθιακά κάθε στοιχείο της ακολουθίας sequence και με βάση αυτό το στοιχείο δεικτοδοτούμε ένα λεξικό. Αρχικά, το λεξικό που δεικτοδοτούμε είναι αυτό το οποίο έχουμε πάρει ως όρισμα από το περιβάλλον της συνάρτησης. Το αποτέλεσμα που μας επιστρέφεται είναι μια αναφορά σε ένα άλλο λεξικό που περιέχεται στο αρχικό. Αυτή η αναφορά αποθηκεύεται στη τοπική μεταβλητή d η οποία και κρύβει (hides) πλέον το όρισμα που είχαμε δεχθεί αφού έχουν το ίδιο όνομα. Έτσι, οποιαδήποτε τροποποίηση του d από εδώ και πέρα θα έχει ισχύ μόνο στο τοπικό περιβάλλον της συνάρτησης και δεν θα επηρεάσει την τιμή των ορισμάτων της. Συνεχίζοντας την ίδια διαδικασία βρίσκουμε το επιθυμητό αποτέλεσμα το οποίο και μπορούμε να επιστρέψουμε στο περιβάλλον εκτέλεσης της συνάρτησης.

Όσον αφορά τη δεύτερη συνάρτηση *populate_trie* η λογική που ακολουθείται είναι παρόμοια, όμως επειδή θέλουμε να τροποποιήσουμε το αντικε-

ίμενο και να του προσθέτουμε κάποια άλλα προκαθορισμένα αντικείμενα μας δίνεται η ευκαιρία να μελετήσουμε την συνάρτηση deepcopy.

Η λογική σε αυτό τη συνάρτηση είναι παρόμοια με την προηγούμενη. Το ενδιαφέρον κομμάτι αρχίζει μόλις βρούμε το μέρος του λεξικού όπου θέλουμε να εμφωλεύσουμε το καινούργιο αντικείμενο. Αν δεν αντιγράφαμε το embedded_obj εκεί, θα είχαμε εισάγει μια αναφορά στο ίδιο αντικείμενο σε διαφορετικά σημεία. Με άλλα λόγια θα είχαμε πάει ένα επίπεδο πιο μέσα στη δομή και θα είχαμε εισάγει μέσα στο embedded_obj μια αναφορά στο embedded_obj. Αυτό ονομάζεται κυκλική δομή. Αντιγράφοντας λοιπόν το αντικείμενο, δημιουργείται ένα καινούργιο αντίγραφο (διαφορετικό από το προηγούμενο) το οποίο και εισάγουμε στη κατάλληλη θέση.

Ένα πιο κλασικό παράδειγμα κυκλικής δομής:

```
>>> L = list(range(5))
>>> L.append(L)
>>> L
[0, 1, 2, 3, 4, [...]]
```

Για μια πιο έξυπνη υλοποίηση της ίδιας συμπεριφοράς όταν δεν περνάμε κάποιο όρισμα για τη θέση, υπάρχει στο τέλος της ενότητας 8.8, ενώ για παρόμοια συμπεριφορά βλέπουμε ένα πιο πλήρες παράδειγμα με την χρήση κλάσεων στην ενότητα 8.3.

Κεφάλαιο 8

Κλάσεις και Αντικείμενα

> What can be said at all can be said clearly,
> and what we cannot talk about we must pass
> over in silence.
>
> Ludwig Wittgenstein

Στη συγκεκριμένη ενότητα, θα γνωρίσουμε ορισμένες από τις βασικές αρχές του αντικεμενοστραφούς προγραμματισμού καθώς και πως αυτές αντικατοπτρίζονται από το συντακτικό της Python.

8.1 Εισαγωγή

Ο Ludwig Wittgenstein χωρίς να το γνωρίζει, έχει διατυπώσει απόψεις που μπορούν να χρησιμοποιηθούν για να περιγράψουν τις κλάσεις και τα αντικείμενα :

- Αντικείμενα περιέχουν την δυνατότητα όλων των καταστάσεων. (μέθοδοι)
- Τα αντικείμενα είναι απλά. (σχεδιαστική αρχή)
- Κάθε δήλωση για σύνθετα μπορεί να αναλυθεί σε δήλωση για αυτά τα οποία τα αποτελούν και σε προτάσεις που περιγράφουν τα σύνθετα πλήρως. (κλάσεις)

- Είτε κάτι έχει ιδιότητες που δεν έχει τίποτα άλλο, οπότε μπορούμε άμεσα να το διαχωρίσουμε από όλα που αναφέρονται σε αυτό, είτε υπάρχουν άρκετά αντικείμενα που έχουν το ίδιο κοινό σύνολο ιδιοτήτων, οπότε και είναι αδύνατο να διακρίνουμε το ένα από το άλλο. (αντιγραφή μέσω αναφοράς)

- Είναι μορφή και περιεχόμενο. (μέθοδοι και χαρακτηριστικά)

- Τα στιγμιότυπα των αντικειμένων παράγουν ένα σύνολο καταστάσεων. (εκτέλεση του προγράμματος)

- Σε μια κατάσταση, τα αντικείμενα βρίσκονται σε μια ντετερμινιστική σχέση μεταξύ τους. (αλληλεπιδράσεις αντικειμένων)

Ο αντικειμενοστραφής προγραμματισμός αποτελεί μια προγραμματιστική τεχνική όπου βασιζόμαστε στα αντικείμενα και στις αλληλεπιδράσεις μεταξύ τους ώστε να σχεδιάσουμε τα προγράμματα που θέλουμε να αναπτύξουμε. Έτσι, στον αντικειμενοστραφή προγραμματισμό, μπορούμε να δούμε ένα πρόγραμμα επικεντρώνοντας στην οπτική μιας συλλογής αντικειμένων και των αλληλεπιδράσεων μεταξύ τους. Κάθε αντικείμενο μπορεί να αλληλεπιδράσει με τα υπόλοιπα στέλνοντας μηνύματα ή λαμβάνοντας, επεξεργαζόμενο δεδομένα και γενικά σαν μια ανεξάρτητη οντότητα που όμως βρίσκεται σε ένα περιβάλλον αλληλεπίδρασης και συνεργασίας με τις υπόλοιπες που απαρτίζουν το πρόγραμμα. Για αυτό τον λόγο και οι μέθοδοι (ή συναρτήσεις) που χρησιμοποιούνται στα αντικείμενα, είναι στενά συνδεδεμένες με τα ίδια τα αντικείμενα. Ουσιαστικά πρόκειται για ένα τρόπο προγραμματισμού που συνδυάζει δεδομένα και λειτουργικότητα και τα ενσωματώνει σε κάτι που ονομάζεται αντικείμενο.

Κλάσεις και αντικείμενα είναι οι δυο κύριες πτυχές του αντικειμενοστραφούς προγραμματισμού. Πιο συγκεκριμένα η κλάση δημιουργεί ένα νέο τύπο, όπου τα αντικείμενα είναι στιγμιότυπα της κλάση.

Ως παραδείγματα μπορούμε να αναφέρουμε:

- **Κλάση:** Προγραμματιστής **Στιγμιότυπο:** Guido van Rossum

- **Κλάση:** Γλώσσα Προγραμματισμού **Στιγμιότυπο:** Python

Τα βασικά χαρακτηριστικά που παρατηρούμε κατά τον σχεδιασμό και την υλοποίηση ενός τυπικού προγράμματος μέσω αντικειμενοστρέφειας είναι:

- *Κλάση (Class):* Καθορίζει τα βασικά χαρακτηριστικά (attributes) και συμπεριφορές (methods) των αντικειμένων που περιγράφονται από αυτή την κλάση. Ένα παράδειγμα θα μπορούσε να είναι η κλάση σκύλος. Γενικότερα μπορούμε να σκεφτόμαστε ότι αντιπροσωπεύουν τα αδρά χαρακτηριστικά που ενυπάρχουν σε όλα τα στιγμιότυπα μιας κλάσης

- *Αντικείμενο (Object):* Μια συγκεκριμένη μορφή μιας κλάσης, δηλαδή ένα από τα αντικείμενα που περιγράφει η κλάση που το χαρακτηρίζει. Για παράδειγμα ο 'Αλήτης'.

- *Στιγμιότυπο (Instance):* Η κατάσταση ενός αντικείμενο κατά την εκτέλεση του προγράμματος μέσα από την συγκεκριμενοποίηση του.

- *Μέθοδος (Method):* Η συμπεριφορά και οι δυνατότητες ενός αντικειμένου.

- *Κληρονομικότητα (Inheritance):* Όταν μια κλάση αποτελεί εξειδίκευση μιας άλλης, και έτσι έχει όλα τα χαρακτηριστικά που περιγράφουν την κλάση από την οποία κληρονομεί, καθώς και κάποια ιδιαίτερα που περιγράφουν ειδικά την συγκεκριμένη ομάδα αντικειμένων. Για παράδειγμα η κλάση σκύλος μπορεί να κληρονομεί από μια γενικότερη κλάση ζώο.

  Θα μπορούσαμε να αντιστοιχήσουμε τις κλάσεις στις γενικές ομάδες που περιγράφουν τα αντικείμενα. Οι κλάσεις περιέχουν πεδία (fields) και μεθόδους ((methods). Τα πεδία ανάλογα με την τιμή τους υποδεικνύουν την κατάσταση του αντικειμένου το οποίο είναι στιγμιότυπο της κλάσης. Οι μέθοδοι περιγράφουν τις διαφορετικές συμπεριφορές του αντικειμένου.

8.2 Βασικές Έννοιες

Η πιο απλή κλάση που θα μπορούσαμε να γράψουμε είναι:

```
class Snake():
    pass

python = Snake()
```

όπου με την δήλωση class Snake(): δημιουργούμε μια νέα κλάση Snake ενώ στην συνέχεια, με την δήλωση python = Snake() δημιουργούμε ένα αντικείμενο αυτής της κλάσης. Παρακάτω μπορούμε να δούμε πως μπορούμε να γράψουμε μια κλάση η οποία να έχει μια μέθοδο και η οποία να τυπώνει ένα συγκεκριμένο μήνυμα όποτε καλείται από αντικείμενο αυτής της κλάσης.

```
class Snake():
    def helloWorld(self):
        print('Hello World!')

python = Snake()
python.helloWorld()
```

Αξίζει να παρατηρήσουμε πως, όπως και οι συναρτήσεις, έτσι και οι μέθοδοι μιας κλάσης ορίζονται με την λέξη κλειδί def. Επίσης, κατά την κλήση της μεθόδου ενός αντικειμένου δεν θέτουμε εμείς κάποια τιμή στο όρισμα self αλλά αυτό γίνεται αυτόματα. Κάθε μέθοδος μιας κλάσης πρέπει να έχει ως πρώτο όρισμα το self, έτσι ώστε στην συνέχεια μέσω αυτού του ορίσματος να γίνεται γνωστό πιο αντικείμενο κάλεσε τη μέθοδο.

Με απλά λόγια, μια κλάση είναι ένας τρόπος κατασκευής ενός αντικειμένου. Ένα αντικείμενο είναι ένας τρόπος να χειριζόμαστε πιο εύκολα χαρακτηριστικά και συμπεριφορές ομαδοποιημένα. Μπορούμε να σκεφθούμε τα αντικείμενα σαν κάτι που υπάρχει στον πραγματικό κόσμο (όπως θα δούμε όμως αυτό δεν είναι απαραίτητο) ενώ τις κλάσεις έναν τρόπο κατασκευής τους και προσδιορισμού των ιδιοτήτων τους. Για παράδειγμα έχουμε στο σπίτι μας τα σκυλάκια Αζορ, Ραφλ, Τόμπυ. Και τα τρία είναι σκυλάκια επομένως μοιράζονται ορισμένες κοινές ιδιότητες (έχουν τέσσερα πόδια, ουρά κτλ), αλλά και καθένα είναι μοναδικό και αυτόνομο. Έτσι παραδείγματος χάρη, όλα έχουν τρίχωμα, αλλά αυτό μπορεί να έχει διαφορετικά χρώματα. Ομοίως κάποια από αυτά να είναι μεγαλόσωμα, μικρόσωμα κοκ. Επομένως, μπορο-

ύμε να αντιστοιχίσουμε την έννοια σκύλος σε μια κλάση, ενώ οι Αζορ, Ραφλ, Τόμπυ είναι αντικείμενα αυτής της κλάσης.

```
class Dog():
    def __init__(self, name, color, height, mood, age):
        """
        This method is called when an new objected is
        initiallized.
        """

        # here we setup the attributes of our dog
        self.name = name
        self.color = color
        self.height = height
        self.mood = mood
        self.age = age
        self.hungry = False
        self.tired = False

    def print_attributes(self):
        """ Print all the attributes of the dog """

        print('Name is', self.name)
        print('Color is', self.color)
        print('Height is', self.height)
        print('Mood is', self.mood)
        print('Age is', self.age)
        print('Hungry is', self.name)
        print('Tired is', self.tired)

ralph = Dog('Ralph', 'blue', 1.80, 'good', 15)
ralph.print_attributes()
```

Όπως βλέπουμε παραπάνω, για να κατασκευάσουμε ένα αντικείμενο τύπου Dog() πρέπει να του περάσουμε τα κατάλληλα ορίσματα που θα προσδιο-

ρίσουν τις ιδιότητες του. Η ειδική μέθοδος που καλείται μόλις κατασκευάζουμε το αντικείμενο είναι η __init__(). Το όνομα της σε κάθε κλάση πρέπει να είναι το συγκεκριμένο. Η __init__() παίρνει πάντα ως πρώτο όρισμα την λέξη self που σημαίνει ότι η συμπεριφορά που προσδιορίζεται αφορά το αντικείμενο από το οποίο καλείται ή στην προκειμένη περίπτωση, το αντικείμενο που πάει να δημιουργηθεί. Τα επόμενα ορίσματα είναι για να προσδιορίσουμε τις ιδιότητες του αντικειμένου.

Κανόνας 8.2.1. Αν κάποιος είναι εξοικειωμένος με άλλες γλώσσες προγραμματισμού, ίσως να μπορεί να αντιστοιχήσει στο μυαλό του το self με το this που υπάρχει σε αυτές. Το self απαιτείται ως το πρώτο όρισμα σε κάθε μέθοδο μιας κλάσης. Επιπρόσθετα, το self δεν αποτελεί μια δεσμευμένη λέξη της γλώσσας, αλλά είναι μια σύμβαση που ακολουθείται να ονομάζεται έτσι. Ωστόσο, καλό είναι κάποιος να μην παρεκκλίνει από αυτή την σύμβαση, γιατί αποτελεί μια πολύ ισχυρή.

8.3 Παραδείγματα Χρήσης Κλάσεων

Ιδιαίτερα συνηθισμένη περίπτωση χρήσης των κλάσεων είναι σε περιπτώσεις που θέλουμε να δημιουργήσουμε δομές δεδομένων. Παρακάτω φαίνεται μια πολύ απλή υλοποίηση δυο συναρτήσεων από μια τροποποιημένη δομή ενός Trie όπου μπορούμε και καταμετρούμε πόσες φορές έχει εμφανιστεί μια ακολουθία χαρακτήρων σε μια συλλογή από λέξεις.

```
class Trie:
    """
    A Trie is like a dictionary in that it maps keys to values.
    However, because of the way keys are stored, it allows
    look up based on the longest prefix that matches.

    """

    def __init__(self):
        # Every node consists of a list with two position. In
        # the first one,there is the value while on the second
```

```
        # one a dictionary which leads to the rest of the nodes.
        self.root = [0, {}]


    def insert(self, key):
        """
        Add the given value for the given key.

        >>> a = Trie()
        >>> a.insert('kalo')
        >>> print(a)
        [0, {'k': [1, {'a': [1, {'l': [1, {'o': [1, {}]}]}]}]}]
        >>> a.insert('kalo')
        >>> print(a)
        [0, {'k': [2, {'a': [2, {'l': [2, {'o': [2, {}]}]}]}]}]
        >>> b = Trie()
        >>> b.insert('heh')
        >>> b.insert('ha')
        >>> print(b)
        [0, {'h': [2, {'a': [1, {}], 'e': [1, {'h': [1, {}]}]}]}]

        """

        # find the node to append the new value.
        curr_node = self.root
        for k in key:
            curr_node = curr_node[1].setdefault(k, [0, {}])
            curr_node[0] += 1


    def find(self, key):
        """
        Return the value for the given key or None if key not
        found.
```

```
        >>> a = Trie()
        >>> a.insert('ha')
        >>> a.insert('ha')
        >>> a.insert('he')
        >>> a.insert('ho')
        >>> print(a.find('h'))
        4
        >>> print(a.find('ha'))
        2
        >>> print(a.find('he'))
        1

        """

        curr_node = self.root
        for k in key:
            try:
                curr_node = curr_node[1][k]
            except KeyError:
                return 0
        return curr_node[0]

    def __str__(self):
        return str(self.root)

    def __getitem__(self, key):
        curr_node = self.root
        for k in key:
            try:
                curr_node = curr_node[1][k]
            except KeyError:
                yield None
        for k in curr_node[1]:
```

```
            yield k, curr_node[1][k][0]

if __name__ == '__main__':
    a = Trie()
    a.insert('kalo')
    a.insert('kala')
    a.insert('kal')
    a.insert('kata')
    print(a.find('kala'))
    for b in a['ka']:
        print(b)
    print(a)
```

8.4 Μεταβλητές Αντικειμένου (attributes)

Τα περιεχόμενα ενός αντικειμένου προσπελάζονται ως μεταβλητές αντικειμένου (ή χαρακτηριστικά αλλιώς) (attributes) χρησιμοποιώντας την τελεία. Για παράδειγμα αν ralph είναι ένα αντικείμενο στιγμιότυπο (instance) της κλάσης Dog, μπορούμε να προσπελάσουμε τη μεταβλητή αντικειμένου του name, γράφοντας ralph.name.

Παρατηρούμε ότι κάποιες μεταβλητές αντικειμένου αρχίζουν με την λέξη self ακολουθούμενες από μια τελεία. Αυτό σημαίνει πως οι συγκεκριμένες μεταβλητές αντικειμένου (θα) υπάρχουν μόνιμα στο αντικείμενο που κατασκευάζουμε και αυτός είναι ο τρόπος με τον οποίο τα προσπελαύνουμε. Το 'θα' το βάλαμε σε παρένθεση γιατί μπορούμε οποιαδήποτε στιγμή να δημιουργήσουμε μια καινούργια μεταβλητή αντικειμένου απλώς γράφοντας self.characteristic = .

Τέλος, βλέπουμε και την μέθοδο print_attributes(). Αυτή παίρνει ως όρισμα την μόνο την λέξη self που σημαίνει ότι αφορά το αντικείμενο από το οποίο καλείται. Προσπελαύνει τα χαρακτηριστικά του αντικειμένου και τα τυπώνει στην οθόνη.

8.5 Συναρτήσεις Μέλους

Θα μπορούσε κάποιος να σκεφθεί τις συναρτήσεις μέλους (member functions) στον αντικειμενοστραφή προγραμματισμό ως μια εξειδίκευση των συναρτήσεων που γνωρίζουμε με το πρώτο όρισμα να έχει ειδική σημασιολογία.

Συνεπώς, αυτά που έχουμε μάθει για τις συναρτήσεις, γενικά, ισχύουν και στις συναρτήσεις μέλους (Member Functions). Άλλες μπορεί να τροποποιούν το αντικείμενο στο οποίο χρησιμοποιούνται, ενώ άλλες απλά να μας επιστρέφουν κάποια πληροφορία.

Οι συναρτήσεις μέλους καλούνται μόνο πάνω σε αντικείμενα. Για αυτό τον λόγο και οι στους πρωταρχικούς τύπους δεν μπορούμε να καλέσουμε κάποια συνάρτηση μέλους καθώς αυτοί δεν αποτελούν αντικείμενα.

Το συντακτικό για να κλήση συναρτήσεων μέλους στην Python είναι:

```
variable.member_function( [any arguments required] )
```

Για να δούμε ποιες συναρτήσεις μέλους υποστηρίζει ένα αντικείμενο, μπορούμε να καλέσουμε την συνάρτηση *dir* που μας παρέχει η Python με όρισμα το αντικείμενο για το οποίο θέλουμε να μάθουμε τις συναρτήσεις μέλους του.

```
a = 'Python by example'
print('Member functions for string')
print(dir(a))

b = (1, 2, 3)
print('Member functions for tuple')
print(dir(b))

c = [1, 2, 3]
print('Member functions for list')
print(dir(c))
```

Οι συναρτήσεις που έχουν όνομα __functionname__ χρησιμοποιούνται εσωτερικά από την Python και δεν έχουν δημιουργηθεί για να καλούνται ρητά από προγράμματα που δημιουργεί ο χρήστης.

Αν κάποια στιγμή αμφιβάλλουμε για την χρήση μιας συνάρτησης, μπο-

ρούμε να συμβουλευτούμε την εσωτερική βοήθεια της Python.

```
c = [1, 2, 3]
print(help(c.sort))
```

8.6 Μεταβλητές Κλάσης και Στατικές Μέθοδοι

8.6.1 Μεταβλητές Κλάσης

Ορισμός 8.6.1. Για την δημιουργία μιας μεταβλητής κλάσης αρκεί η δήλωση της αμέσως μετά τον ορισμό της κλάσης (στο εσωτερικό της). Οι μεταβλητές κλάσης είναι διαθέσιμες και κοινές σε όλα τα αντικείμενα της κλάσης.

```
class Snake:
  noOfSnakes = 0

  def __init__(self, name = 'unknown'):
    self.name = name
    print('— Initializing {0} —'.format(self.name))
    Snake.noOfSnakes += 1

  def helloWorld(self):
    print('Hello World from {0}!'.format(self.name))

  @staticmethod
  def numberOfSnakes():
    print(Snake.noOfSnakes)

python = Snake('TasPython')
python.helloWorld()
Snake.numberOfSnakes()
cobra = Snake()
cobra.helloWorld()
Snake.numberOfSnakes()
```

Μέσα από το παραπάνω παράδειγμα μπορούμε να δούμε έναν απλό τρόπο όπου η τιμή μιας μεταβλητής αυξάνεται κάθε φορά που κατασκευάζεται ένα καινούργιο αντικείμενο. Για να το επιτύχουμε αυτό, κάθε φορά που καλείται ο δημιουργός της κλάσης (__init__, αυξάνουμε κατά ένα την τιμή της μεταβλητής κλάσης noOfSnakes.

8.6.2 Στατικές Μέθοδοι

Ορισμός 8.6.2. Μία μέθοδος που έχει δηλωθεί ως στατική, λειτουργεί στο επίπεδο της κλάσης και όχι στο επίπεδο του στιγμιότυπου της. Επομένως, μια στατική μέθοδος δεν μπορεί να αναφέρεται σε συγκεκριμένο στιγμιότυπο της κλάσης (δηλαδή δεν μπορεί να αναφέρεται στο self).

Για να δηλώσουμε μια στατική μέθοδο σε μια κλάση, χρησιμοποιούμε τον διακοσμητή (decorator) @staticmethod όπως είδαμε και στο παραπάνω παράδειγμα. Στην συνέχεια, για να χρησιμοποιήσουμε μια στατική μέθοδο, αρκεί να αναφερθούμε σε ποια κλάση αυτή ανήκει, ή αν χρησιμοποιήσουμε στιγμιότυπο κάποιας κλάσης, τότε αυτό αγνοείται εκτός από την κλάση στην οποία αυτό ανήκει.

8.7 Κληρονομικότητα

Η κληρονομικότητα μοιάζει με το να χρησιμοποιούμε ως μέλος μιας κλάσης A ένα αντικείμενο b_1 μιας άλλης κλάσης B. Έτσι, θα μπορούσαμε να καλέσουμε ορισμένες συναρτήσεις της B στα αντικείμενα της A, μέσω του αντικειμένου b_1 που υπάρχει σε αυτά[1]. Η κληρονομικότητα μας επιτρέπει να εφαρμόζουμε απευθείας αυτές τις συναρτήσεις στα αντικείμενα της κλάσης B[2]. Οι άνθρωποι διακρίνουν τις αφηρημένες έννοιες σε δυο διαστάσεις: *μέρος-από* (part-of) ή σχέση έχει (has-a) και *εξειδίκευση-του* ή είναι (is-a). Για παράδειγμα η Python *είναι* γλώσσα προγραμματισμού και έχει δυναμικούς τύπους. Η σχέση *έχει* συνήθως σημαίνει ότι πρέπει να χρησιμοποιήσουμε

[1] Για παράδειγμα θα μπορούσαμε να καλέσουμε τη συνάρτηση fun που εφαρμόζεται σε αντικείμενα της κλάσης B γράφοντας a1.b1.fun(), όπου a1 αντικείμενο της κλάσης A.

[2] Το προηγούμενο παράδειγμα θα γινόταν απλά a1.fun()

ένα αντικείμενο ως μέρος μιας κλάσης ενώ η σχέση *είναι* συνήθως επιτάσσει την χρήση κληρονομικότητας.

Ορισμός 8.7.1. Κληρονομικότητα είναι ένας τρόπος να δημιουργηθούν κοινές κλάσεις χρησιμοποιώντας άλλες που ήδη υπάρχουν. Η κληρονομικότητα εφαρμόζεται προκειμένου να γίνει επαναχρησιμοποίηση ήδη υπάρχοντα κώδικα με λίγες ή καθόλου αλλαγές. Η νέα κλάση που δημιουργείται (υποκλάση) κληρονομεί τα χαρακτηριστικά και την συμπεριφορά των ήδη υπαρχουσών κλάσεων (υπερκλάσεων).

```
class UniversityMember:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def who(self):
        print('Name:"{}" Age:"{}" '.format(self.name, self.age))

class Professor(UniversityMember):
    def __init__(self, name, age, salary):
        UniversityMember.__init__(self, name, age)
        self.salary = salary

    def who(self):
        UniversityMember.who(self)
        print('Salary: "{}" '.format(self.salary))

class Student(UniversityMember):
    def __init__(self, name, age, marks):
        UniversityMember.__init__(self, name, age)
        self.marks = marks

    def who(self):
        UniversityMember.who(self)
        print('Marks: "{}" '.format(self.marks))
```

```
p = Professor('Mr. Sipser', 40, 3000000)
s = Student('Kosmadakis', 25, 9.99)

for member in (p, s):
    print('--------------------')
    member.who()
```

Στο συγκεκριμένο παράδειγμα, η κλάση Professor κληρονομεί από την υπερκλάση της UniversityMember. Για να προσδιορίσουμε μια κλάση από ποια κλάση κληρονομεί, βάζουμε ως όρισμα της κλάσης Professor τις κλάσεις από τις οποίες θέλουμε να κληρονομεί (UniversityMember). Στην συνέχεια, μπορούμε να καλέσουμε συναρτήσεις από την υπερκλάση της Professor και να συμπληρώσουμε στα αποτελέσματα τους την επιπρόσθετη λειτουργία που απαιτείται.

Με αυτό τον τρόπο βλέπουμε πως οι υπερκλάσεις αποτελούν ένα επίπεδο αφαίρεσης ενώ αυτές που κληρονομούν από αυτές αφορούν πιο συγκεκριμένες έννοιες. Έτσι, μπορούμε να χρησιμοποιούμε υπερκλάσεις όταν θέλουμε να ομαδοποιούμε κοινή συμπεριφορά ανάμεσα σε κλάσεις και στην συνέχεια σε αυτές να υλοποιούμε μόνο ό,τι διαφέρει, επαναχρησιμοποιώντας έτσι όσο το δυνατόν περισσότερο κώδικα μπορούμε και επιμένοντας στην αρχή DRY – Don't Repeat Yourself που σημαίνει να μην γράφουμε τον ίδιο κώδικα ξανά και ξανά, πολλαπλασιάζοντας έτσι τα πιθανά λάθη που αυτός μπορεί να φέρει και κάνοντας πιο δύσκολη την συντήρησή τους (μια διόρθωση σε ένα σημείο πρέπει να γίνει ξανά και σε όλα τα σημεία που πιθανώς υπάρχει ο ίδιος κώδικας).

8.8 Ειδικές Μέθοδοι

Μέσω κλάσεων, μπορούμε να υλοποιήσουμε μεθόδους που να αντιστοιχούν σε ειδικές περιπτώσεις (όπως αριθμητικούς τελεστές, σύγκριση αντικειμένων, διαγραφή τους κ.α.). Αυτές οι μέθοδοι έχουν ειδικά ονόματα που αρχίζουν και τελειώνουν σε __. Παρακάτω θα δούμε ορισμένες βασικές τέτοιες μεθόδους. Αν είστε εξοικειωμένοι με άλλες γλώσσες προγραμματισμού, αυτός είναι ο

τρόπος της Python για υπερφόρτωση τελεστών operator overloading.

- *__init__(self, . . .):* Αποτελεί μια μέθοδο που ήδη έχουμε δει. Είναι ο δημιουργός μιας κλάσης και καλείται όταν δημιουργούμε ένα καινούργιο αντικείμενο. Αν μια υπερκλάση (base class) έχει την μέθοδο __init__(), τότε αυτή πρέπει να υλοποιείται ρητά και από όλες τις υποκλάσεις της. Επιπλέον, δεν μπορεί η μέθοδος __init__() να επιστρέφει κάποια τιμή

- *__del__(self):* Καλείται όταν ένα αντικείμενο καταστρέφεται. Ονομάζεται καταστροφέας (destructor). Αντίστοιχα με τον δημιουργό, και αυτή πρέπει να καλείται ρητά αν μια υπερκλάση την υλοποιεί.

- *__repr__(self):* Καλείται μέσω της repr() συνάρτησης έτσι ώστε να δημιουργήσει την 'επίσημη' αναπαράσταση σε αλφαριθμητικό του αντικειμένου. Αν είναι δυνατό, πρέπει να επιστρέφεται μια έκφραση σε Python που να δημιουργεί ένα αντικείμενο με την ίδια τιμή. Συνήθως χρησιμοποιείται για λόγους debugging.

- *__str__(self):* Καλείται μέσω της str() συνάρτησης ή όποτε προσπαθούμε να τυπώσουμε απευθείας το αντικείμενο μέσω της print(). Διαφέρει από την __repr__() στο ότι δεν χρειάζεται να είναι έκφραση. Πάντα πρέπει να είναι ένα αλφαριθμητικό.

- *__lt__(self, other)*, *__le__(self, other)*, *__eq__(self, other)*, *__ne__(self, other)*, *__ne__(self, other)*, *__gt__(self, other)*, *__ge__(self, other)*: Αποτελούν τελεστές συγκρίσεις και αντιστοιχούν στο *μικρότερο από* $(<)$, *μικρότερο ή ίσο* $(<=)$, *ίσο με* $(==)$, *όχι ίσο με* $(!=)$, *μεγαλύτερο από* $(>)$, *μεγαλύτερο ή ίσο από* $(>=)$. Οι παραπάνω συναρτήσεις καλούνται μέσω των τελεστών που βλέπουμε στις παρενθέσεις και είναι αρκετά χρήσιμες σε περιπτώσεις που για παράδειγμα μπορεί να θέλουμε να κάνουμε ταξινόμηση σε αντικείμενα που δημιουργήσαμε.

Επίσης, μπορούμε να υλοποιήσουμε μεθόδους που αντιστοιχούν σε μαθηματικές πράξεις, μερικές από τις οποίες βλέπουμε παρακάτω.

- *__add__(self, other):* Πρόσθεση $(+)$

- *__sub__(self, other):* Αφαίρεση $(-)$

- *__mul__(self, other):* Πολλαπλασιασμός $(*)$
- *__truediv__(self, other):* Διαίρεση $(/)$
- *__floordiv__(self, other):* Διαίρεση ακεραίων $(//)$
- *__mod__(self, other):* Υπόλοιπο $(\%)$

Πέρα από αυτές τις μεθόδους, υπάρχουν και άλλες που θα δούμε πιο αναλυτικά στην ενότητα των περιγραφέων (descriptors) ή άλλες που είναι πιο προχωρημένες. Προσοχή πρέπει να δοθεί στο ότι πρέπει να αποφεύγουμε να δημιουργούμε δικές μας συναρτήσεις που αρχίζουν και τελειώνουν με __, παρά μόνο να υλοποιούμε ήδη υπάρχουσες, όπως αυτές που φαίνονται παραπάνω.

Ένα παράδειγμα με το τι μπορούμε να κάνουμε χρησιμοποιώντας μερικές από αυτές φαίνεται παρακάτω :

```
class EmbeddedDict(dict):
    def __missing__(self, key):
        value = EmbeddedDict()
        self[key] = value
        return value

d = {}
d2 = EmbeddedDict(d)
d2['a']['b']['c'] = 1
print(d2)
```

Προσπαθούμε να προσπελαύσουμε ένα στοιχείο του λεξικού το οποίο αν δεν υπάρχει, το δημιουργούμε εκείνη τη στιγμή.

Κεφάλαιο 9

Αρχεία

> Το ανθρώπινο σώμα με την κατάλληλη φροντίδα μπορεί να κρατήσει μια ολόκληρη ζωή.
>
> Ανώνυμος

Μέσω των αρχείων μπορούμε να αποθηκεύσουμε ή να ανακτήσουμε πληροφορίες που μπορούν να εκτείνονται πέρα από τον χρόνο εκτέλεσης του προγράμματος μας. Στην πρώτη ενότητα θα δούμε πως αυτό είναι δυνατό, ενώ στη δεύτερη ενότητα του συγκεκριμένου κεφαλαίου θα μάθουμε ορισμένες βασικές λειτουργίες για τη διαχείριση του συστήματος αρχείων στον υπολογιστή μας.

9.1 Προσπέλαση

Προτού μπορέσουμε να επιτελέσουμε οποιαδήποτε λειτουργία πάνω σε ένα αρχείο, πρέπει να το ανοίξουμε, έτσι ώστε να ενημερώσουμε το λειτουργικό ότι πρόκειται να το χρησιμοποιήσουμε και να αναλάβει αυτό να το ανασύρει από τον σκληρό δίσκο. Για να ανοίξουμε ένα αρχείο, χρησιμοποιούμε την συνάρτηση open() με όρισμα το όνομα του αρχείου. Αυτή, στην συνέχεια μας επιστρέφει έναν περιγραφέα για το αρχείο, τον οποίο και χρησιμοποιούμε ώστε να το προσπελάσουμε. Έπειτα μπορούμε να εφαρμόσουμε, χρησιμοποιώντας αυτόν τον περιγραφέα, διάφορες λειτουργίες πάνω στο αρχείο. Οι

τρόποι που μπορούμε να ανοίξουμε ένα αρχείο μέσω της open() φαίνονται στον πίνακα 9.1.

Τρόπος ανοίγματος	Χρήση
"r"	Ανάγνωση
"w"	Εγγραφή (Διαγραφή τυχών προηγούμενων περιεχομένων)
"a"	Προσθήκη (Διατήρηση τυχών προηγούμενων περιεχομένων)
"b"	Αρχείο δυαδικής μορφής
"+"	Προσθήκη δεδομένων στο τέλος του αρχείου "r+" είναι άνοιγμα αρχείου και για ανάγνωση/εγγραφή

Πίνακας 9.1: Τρόποι αλληλεπίδρασης με αρχεία

Από τα "r", "w", "a" μόνο μια σημαία το πολύ μπορεί να επιλεχθεί. Αν δεν επιλεχθεί καμία, το αρχείο ανοίγει ως προεπιλογή μόνο για ανάγνωση (δηλαδή υπονοείται η σημαία "r".

9.2 Βασικές συναρτήσεις

9.2.1 Διάβασμα από αρχείο

Η κύρια συνάρτηση για διάβασμα από αρχείο είναι η read(). Χωρίς όρισμα θα διαβάσει μέχρι το τέλος του αρχείου, αλλιώς για τον αριθμό bytes όπως καθορίζεται από το όρισμα της.

```
f = open('input_file.txt') # open, read-only (default)
print(f.name) # recover name from file object
print(f.readlines())
print(f.readlines()) # already at end of file
f.seek(0) # go back to byte 0 of the file
print(f.read()) # read to EOF, returning bytes in a string
f.close()
```

Η διαφορά που υπάρχει ανάμεσα στην readlines και στην read, είναι ότι η πρώτη διαβάζει μια μια τις γραμμές και τις επιστρέφει σε μια λίστα, ενώ η δεύτερη διαβάζει όλο το αρχείο και επιστρέφει ό,τι διαβάσει μέσα σε μια μεταβλητή. Τέλος, η συνάρτηση close() αναλαμβάνει να κλείσει το αρχείο και να απελευθερώσει έτσι πόρους του συστήματος.

9.2.2 Εγγραφή σε αρχείο

Αντίστοιχα με το διάβασμα από αρχείο, μπορούμε να κάνουμε την εγγραφή, μόνο που τώρα πρέπει να το καθορίσουμε, βάζοντας το κατάλληλο όρισμα στην συνάρτηση open(). Προσοχή πρέπει να δοθεί μόνο στο γεγονός ότι με τον τρόπο που ανοίγουμε τώρα το αρχείο, οποιαδήποτε άλλα περιεχόμενα του διαγράφονται.

```
g = open('new_file','w') # open for writing
g.write('A new file begins') # takes just one argument
g.write('...today!\n')
g.close()
```

Αν θέλουμε να κρατήσουμε τα περιεχόμενα ενός αρχείου, και να προσθέσουμε κάποια άλλα, τότε πρέπει να ανοίξουμε το αρχείο με την ειδική σημαία για προσθήκη.

```
f = open('file.txt', 'a')
f.write('Append a new line at the end of the file\n')
f.close()
```

9.2.3 Διάτρεξη σε αρχεία

Μπορούμε να χρησιμοποιήσουμε βρόγχους for για να διατρέξουμε πάνω σε αρχεία (iterate), παίρνοντας για παράδειγμα μια μια τις γραμμές.

```
f = open('input_file.txt') # open, read-only (default)

for line in f:
    print(line)
```

Αν δοκιμάσουμε το παραπάνω παράδειγμα, θα διαπιστώσουμε πως εκτυπώνεται μια κενή γραμμή ανάμεσα σε κάθε γραμμή του αρχείου. Αυτό συμβαίνει, γιατί στο τέλος της γραμμής κάθε αρχείου, υπονοείται ο χαρακτήρας αλλαγής γραμμής $\backslash n$. Έτσι, όταν την εκτυπώνουμε, εκτυπώνεται και αυτός ο χαρακτήρας. Ο επιπλέον λοιπόν χαρακτήρας αλλαγής γραμμής προέρχεται από την print καθώς όταν εκτυπώνουμε το αλφαριθμητικό, στο τέλος τυπώνουμε και έναν επιπρόσθετο χαρακτήρα αλλαγής γραμμής.

9.2.4 Εγγραφή αντικειμένων σε αρχεία (σειριοποίηση)

Η σειριοποίηση (serialization) ενός αντικειμένου, είναι η διαδικασία μετατροπής του αντικειμένου σε μια σειρά από bits με σκοπό την εγγραφή του σε κάποιο μέσο αποθήκευση (συνήθως αρχεία) ή την μεταφορά του μέσω δικτύου. Ο σειριοποίηση πρέπει να γίνεται με τέτοιο τρόπο ώστε να καθίσταται δυνατή η ανάκτηση του αντικειμένου στην αρχική του μορφή και με τις ίδιες ιδιότητες που το χαρακτήριζαν. Η σειριοποίηση θα σας χρειαστεί εφόσον θέλετε ένα πρόγραμμα που έχετε φτιάξει να αποθηκεύσει ορισμένα δεδομένα από τη μνήμη στο σκληρό (σε συνήθως ακαταλαβίστικη μορφή από τον άνθρωπο) και στην συνέχεια θέλετε να τα ανακτήσετε.

Για τον σκοπό αυτό, η Python χρησιμοποιεί το άρθρωμα (module) pickle. Το pickle μπορεί να χρησιμοποιήσει διάφορους τρόπους για να αποθηκεύσει τα δεδομένα σε αρχείο.

- *Πρωτόκολλο έκδοσης 0:* η αρχική έκδοση του πρωτοκόλλου που τα δεδομένα αποθηκεύονται σε αναγνώσιμη μορφή από τον άνθρωπο
- *Πρωτόκολλο έκδοσης 1:* η παλιά έκδοση που αποθήκευε τα δεδομένα σε δυαδική μορφή και δεν χρησιμοποιείται πια
- *Πρωτόκολλο έκδοσης 2:* η έκδοση που χρησιμοποιόταν στην έκδοση 2 της Python
- *Πρωτόκολλο έκδοσης 3:* η τρέχουσα έκδοση

Συνήθως χρησιμοποιείται η πιο καινούργια έκδοση (προς το παρόν η 3), παραλείποντας το αντίστοιχο όρισμα. Όταν ανακτάται ένα αντικείμενο από κάποιο αρχείο, επιλέγεται αυτόματα η σωστή έκδοση του πρωτοκόλλου.

```
import pickle

write_data = [1, 2.0, 'asdf', [None, True, False]]

with open('data.pickle', 'wb') as f:
    pickle.dump(write_data, f)

with open('data.pickle', 'rb') as f:
    read_data = pickle.load(f)

print(read_data)
```

Προσοχή πρέπει να δοθεί στο γεγονός ότι δεν σειριοποιούνται όλα τα αντικείμενα της python αλλά ούτε και εγγυάται η προστασία από κακόβουλη τροποποίηση τους. Επίσης, επειδή η τελευταία έκδοση του πρωτοκόλλου που χρησιμοποιείται αυτή την στιγμή είναι σε δυαδική μορφή (binary format), κάθε αρχείο στο οποίο θα εγγραφεί μέσα κάποιο αντικείμενο πρέπει να ανοίγεται με τον αντίστοιχο τρόπο.

9.3 Φάκελοι

Πολλές φορές, κάνουμε διάφορα πράγματα για τα οποία χρειαζόμαστε γνώση για κάποιους φακέλους του συστήματος. Η python μπορεί να μας βοηθήσει πολύ εύκολα σε αυτό, αναλαμβάνοντας από μόνη της ο κώδικας της να τρέχει ανεξαρτήτως πλατφόρμας. Το μόνο που έχουμε να κάνουμε εμείς είναι να εισάγουμε την βιβλιοθήκη os που περιέχει τις συγκεκριμένες συναρτήσεις (που προφανώς υλοποιούνται διαφορετικά ανάλογα το σύστημα) και στην συνέχεια με απλές συναρτήσεις να πάρουμε τις απαραίτητες πληροφορίες.

9.3.1 Ανάκτηση Πληροφοριών

Οι κύριες συναρτήσεις που χρειάζονται για αυτόν τον σκοπό είναι:

- *getcwd():* Επιστρέφει το όνομα του τρέχοντος φακέλου.

- *listdir():* Παραθέτει τα αρχεία του τρέχοντος φακέλου.
- *chdir(path):* Αλλαγή φακέλου σε αυτόν που καταδεικνύεται από το path.

Το άρθρωμα (module) που χρησιμοποιούμε για την διαχείριση αρχείων και φακέλων είναι κυρίως το os. Οι συναρτήσεις που μας παρέχει είναι στην συντριπτική τους πλειοψηφία cross-platform, με εξαίρεση ορισμένες εξιδικευμένες περιπτώσεις.

Στο ακόλουθο παράδειγμα βλέπουμε βασικές λειτουργίες για τον καθορισμός μιας διαδρομής στο σύστημα αρχείων. Οι συναρτήσεις os.path.split() και os.path.join() πρέπει να χρησιμοποιούνται αν θέλουμε να έχουμε cross-platform εφαρμογή, γιατί υπάρχουν διαφορές στο πως αναπαριστώνται οι διαδρομές στις διάφορες πλατφόρμες.

```
import os

abs_path = os.path.abspath(os.path.curdir)
# os.path.join is used to append a relative path at the
# end of another path
path2 = os.path.join(abs_path, "new_dir")
print(path2)
# os.path.split "splits" the path in two pieces.
# The path until the last '/' and whatever follows
path0, path1 = os.path.split(path2)
print(path0)
print(path1)
```

Βασικές πληροφορίες για τα αρχεία αλλά και τους φακέλους που περιέχονται στο σύστημα αρχείων μπορούμε να πάρουμε μέσω της αρκετά απλής (και cross-platform) διεπαφής που μας προσφέρει η Python.

```
import os

print(os.path.curdir) # get current directory (Pythonic humor)
abs_path = os.path.abspath(os.path.curdir) # get its full path
print(abs_path)
```

```
full_path = os.getcwd()
print(full_path)

# get the tail end of the path
print(os.path.basename(full_path))

# list the contents of the current directory
os.listdir(full_path)

# get information about file.txt
# the numbers below are: inode protection mode; inode number;
# device inode resides on; number of links to the inode;
# user id of the owner; group id of the owner; size in bytes;
# time of last access; time of last modification;
# "ctime" as reported by OS
print(os.stat('file.txt'))


size = os.path.getsize('file.txt')
print('The file size is: ', size)
isdirectory = os.path.isdir('file.txt')
print('Is file.txt a directory: ', isdirectory)
# time of last modification (seconds since Epoch)
last_mod = os.path.getmtime('file.txt')
print('The last time this file modified is', last_mod)
```

Τέλος, να αναφέρουμε πως μπορούν να εκτελεστούν εντολές της πλατφόρμας στην οποία βρισκόμαστε μέσω της συνάρτησης os.system().

```
import os

os.system("ls")
```

Προσοχή μόνο πρέπει να δοθεί στο γεγονός ότι με τον παραπάνω τρόπο, ο κώδικας που παράγεται εξαρτάται από την πλατφόρμα στην οποία βρισκόμα-

στε.

9.3.2 Δημιουργία Φακέλων

Για την δημιουργία φακέλων υπάρχουν δυο βασικές συναρτήσεις.

- *mkdir(path):* Δημιουργία φακέλου με το όνομα path.
- *makedirs(path):* Δημιουργία του φακέλου που ορίζεται από το path καθώς και όλων των άλλων φακέλων που μπορεί να χρειάζεται να δημιουργηθούν, μέχρι να φθάσουμε σε αυτόν.

```
import os

# remove new_dir if already exists and it is empty
if os.path.isdir('new_dir'):
    os.rmdir('new_dir')
os.makedirs('new_dir')
print(os.listdir('.'))
# change to subdirectory new_dir
os.chdir('new_dir')
```

Κεφάλαιο 10

Εξαιρέσεις

> Success is not final, failure is not fatal: it is the courage to continue that counts.
>
> Winston Churchill

Οι εξαιρέσεις μας επιτρέπουν να διαχωρίσουμε τις περιπτώσεις όπου καλούμαστε να διαχειριστούμε μια 'εξαιρετική' περίπτωση ενός συμβάντος (πχ λείπει κάποιο αρχείο) από την λογική του προγράμματος μας. Στο συγκεκριμένο κεφάλαιο θα δούμε πως μπορούμε να τις διαχειριστούμε αποτελεσματικά ώστε να επιτύχουμε αυτό το σκοπό, καθώς και πως μπορούμε να αποφύγουμε ορισμένα λάθη που συχνά απαντώνται κατά τη χρήση τους.

10.1 Εισαγωγή

Ορισμός 10.1.1. Ο χειρισμός εξαιρέσεων είναι μια κατασκευή η οποία μας επιτρέπει να χειριστούμε ειδικές συνθήκες που αλλάζουν την φυσιολογική ροή του προγράμματος.

Όπως βλέπουμε και από τον παραπάνω ορισμό, τις εξαιρέσεις τις χρησιμοποιούμε σε ειδικές περιπτώσεις που η ροή του προγράμματος αλλάζει από την 'φυσιολογική' λόγω του ότι έχει συμβεί ένα εξαιρετικό συμβάν, που συνήθως δεν γίνεται. Ένα παράδειγμα θα μπορούσε να είναι η περίπτωση όπου

δεν μπορέσαμε να ανοίξουμε ένα αρχείο και να διαβάσουμε τα περιεχόμενα του. Γενικά οι εξαιρέσεις δημιουργούνται όταν κάτι πάει στραβά, ή όταν κάτι παρεκκλίνει σπάνια από μια συγκεκριμένη ροή (πχ υπάρχει μια πολύ πολύ μικρή πιθανότητα να συμβεί ένα γεγονός).

Ο γενικότερος μηχανισμός που λειτουργούν οι εξαιρέσεις είναι:

1. Αν κάτι πάει στραβά η Python εγείρει μια εξαίρεση.
2. Αναγνωρίζουμε την κλάση της εξαίρεσης (πχ NameError, TypeError, IndexError).
3. Εκτελούμε ειδικό κώδικα για αυτή την περίπτωση.
4. Αν δεν χειριστούμε μια εξαίρεση που εγείρεται, τότε το πρόγραμμα μας σταματάει την εκτέλεση του.
5. Εξαιρέσεις που δεν έχουν χειριστεί, εμφανίζονται στο traceback, όπου και μας ενημερώνει τι ακριβώς συνέβη.

Αξίζει να σημειώσουμε πως όταν κάποια ενδεχόμενα είναι σχεδόν ισοπίθανο να συμβούν, ή δεν αποτελεί ιδιάζων γεγονός η πραγματοποίηση κάποιου από αυτά, τότε είναι καλύτερα να χρησιμοποιούμε την δομή if... else ..., που είδαμε σε προηγούμενη ενότητα εφόσον αυτό είναι δυνατό. Αυτό, γιατί αν συμβεί εξαίρεση, τότε ο χειρισμός της είναι πολύ πιο αργός από το να πηγαίναμε στο κομμάτι του else. Από την άλλη, αν πρόκειται να χειριστούμε σφάλματα τότε και ο κώδικας μας γίνεται πιο 'καθαρός' αν χρησιμοποιούμε εξαιρέσεις, και πιο γρήγορος όταν μένει στην προκαθορισμένη του ροή και δεν υπάρχει ανάγκη εκτέλεσης των δηλώσεων στα μέρη του except.

10.2 Είδη Εξαιρέσεων.

Υπάρχουν διάφορα είδη εξαιρέσεων που μπορούμε να συναντήσουμε, καθώς και να φτιάξουμε τα δικά μας. Ένα απλό παράδειγμα εξαίρεσης αποτελεί η διαίρεση με το μηδέν.

```
>>> 8/0
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
ZeroDivisionError: int division or modulo by zero
```

Κάνοντας διαίρεση με το μηδέν στο διαδραστικό κέλυφος (shell) της Python, μπορούμε να δούμε τι ακριβώς συμβαίνει μόλις προκληθεί μια εξαίρεση και δεν την χειριστούμε. Στο παράδειγμα μας, φαίνεται σε ποιο αρχείο συνέβη η εξαίρεση (<stdin>, αφού γράφαμε στο περιβάλλον του διερμηνευτή), σε ποια γραμμή (γραμμή 1), η κλάση της εξαίρεσης (ZeroDivisionError, που σημαίνει διαίρεση με το μηδέν) καθώς και ένα μήνυμα σφάλματος (int division or modulo by zero).

Αφού έχουμε αναγνωρίσει λοιπόν την εξαίρεση που συνέβη, μπορούμε πλέον να την χειριστούμε.

```
>>> try:
...     8/0
... except ZeroDivisionError:
...     print('Could not make the division')
...
Could not make the division
```

Για να την χειριστούμε, τοποθετούμε τις γραμμές που μπορούν να προκαλέσουν την εξαίρεση που θα χειριστούμε μέσα σε ένα μπλοκ try και εφόσον συμβεί μια εξαίρεση, τότε σταματάει η εκτέλεση των δηλώσεων αυτού του μπλοκ και περνάμε κατευθείαν στο αντίστοιχο μπλοκ except. Αν έχει συμβεί μια εξαίρεση που χειρίζεται το συγκεκριμένο μπλοκ (στην περίπτωση μας ZeroDivisionError), τότε εκτελούμε τις δηλώσεις που αυτό περιέχει και θεωρούμε πως έχουμε χειριστεί την εξαίρεση. Έτσι το πρόγραμμα μας δεν σταματάει απότομα πια την εκτέλεση του.

Άλλους τύπους εξαιρέσεων που θα μπορούσαμε να συναντήσουμε, φαίνονται στα παρακάτω παραδείγματα:

```
>>> a * 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

όπου προσπαθούμε να χρησιμοποιήσουμε τον τελεστή ∗ σε μια μεταβλητή a χωρίς όμως πρώτα να την έχουμε ορίσει. Κατά αυτό τον τρόπο προκαλείται μια εξαίρεση κλάσης NameError. Αν αυτή η εξαίρεση έγινε από αβλεψία του προγραμματιστεί, τότε πρέπει να ορίσουμε κατάλληλα την μεταβλητή a, αλλιώς μπορούμε να την χειριστούμε μέσω ενός try...except... μπλοκ, αφού τώρα πια έχουμε αναγνωρίσει την κλάση της εξαίρεσης (NameError).

Αντίστοιχο σφάλμα μπορούμε να παρατηρήσουμε και στην περίπτωση που προσπαθούμε να εφαρμόσουμε μια συνάρτηση (ή τελεστή) σε τύπους που δεν υποστηρίζουν αυτή τη λειτουργία.

```
>>> b = None
>>> b * 7
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for *:
'NoneType' and 'int'
```

Τέλος, ένα ακόμα παράδειγμα για μια εξαίρεση που τουλάχιστον αρχικά, πολλοί προγραμματιστές συναντούν, αφορά τα σφάλματα εισόδου εξόδου.

```
>>> with open('3BirdsAreSitting.mp3', 'r') as f:
...     f.read()
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory:
'3BirdsAreSitting.mp3'
```

Εδώ αξίζει να σημειώσουμε ότι ενώ λόγω της δομής with, αν είχε ανοίξει το αρχείο 3BirdsAreSitting.mp3, τότε θα βγαίνοντας από τη δήλωση with, θα είχε κλείσει κατάλληλα, αν δεν κάνουμε κατάλληλο χειρισμό του σφάλματος, τότε και πάλι θα προκληθεί μια εξαίρεση την οποία δεν θα έχουμε χειριστεί.

Τέλος, αξίζει να αναφέρουμε πως όπως χειριζόμαστε μια κλάση εξαιρέσεων σε ένα μπλοκ except, θα μπορούσαμε να χειριζόμαστε και πολλές κλάσεις εξαιρέσεων μαζί, χρησιμοποιώντας μια πλειάδα που να περιέχει τις κλάσεις όλων των εξαιρέσεων που θέλουμε να χειριστούμε.

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

Θα μπορούσαμε να χειριστούμε ακόμα και όλες τις εξαιρέσεις μαζί χρησιμοποιώντας ένα σκέτο except:, όπως θα δούμε παρακάτω, αν και δεν συνιστάται ως μέθοδος παρά μόνο σε πολύ συγκεκριμένες περιπτώσεις.

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as err:
    print("I/O error: {}".format(err))
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error:", sys.exc_info()[0])
    raise
```

Σε αυτό το παράδειγμα, βλέπουμε πως μπορούμε να χειριστούμε διαφορετικά, διαφορετικά είδη σφαλμάτων που μπορούν να προκύψουν, χρησιμοποιώντας περισσότερα από ένα except μπλοκ. Αρχικά, δοκιμάζουμε να ανοίξουμε ένα αρχείο με το όνομα myfile.txt, να διαβάσουμε την πρώτη του γραμμή και να την μετατρέψουμε σε ακέραιο αφού πρώτα τις αφαιρέσουμε τα κενά που μπορεί να περιέχει στην αρχή ή στο τέλος της.

Σε αυτές τις γραμμές κώδικά, μπορούν να συμβούν διάφορες εξαιρέσεις. Κατά αρχάς, θα μπορούσε να μην είναι δυνατό το άνοιγμα του αρχείου για διάφορους λόγους, οπότε πιάνουμε την εξαίρεση τύπου IOError και ονομάζουμε το στιγμυότυπο της err. Στην συνέχεια τυπώνουμε ένα μήνυμα που να περιέχει λεπτομέρειες σχετικά με την εξαίρεση που συνέβει. Εδώ να σημειώσουμε πως αφού η χειριστήκαμε την εξαίρεση μέσα στο except μπλοκ και δεν συνέβει κάποια άλλη, η ροή του προγράμματος μας θα συνεχίσει κανονικά, χωρίς να εκτελεστεί κάποιο άλλο except ή άλλη γραμμή από το

μπλοκ του try, αλλά με ότι υπάρχει πέρα του αυτού του μπλοκ. Ομοίως αν είχε συμβεί εξαίρεση ValueError κατά την μετατροπή του αλφαριθμητικού σε ακέραιο.

Τέλος, αν είχε συμβεί οποιαδήποτε άλλη εξαίρεση, τότε θα τυπώναμε ένα μήνυμα με ορισμένες πληροφορίες για την εξαίρεση που συνέβη και με το raise, θα εγείρουμε ξανά τη εξαίρεση που συνέβη ώστε να την χειριστεί το πρόγραμμα μας σε ένα πιο πάνω επίπεδο.[1] Έτσι, χειριζόμαστε αρχικά την εξαίρεση, αλλά επιτρέπουμε (ή μάλλον υποχρεώνουμε αν δεν θέλουμε να κρασάρει το πρόγραμμα) να χειριστεί ξανά η εξαίρεση σε ένα άλλο επίπεδο. Πιο αναλυτικά για το πως λειτουργεί η raise θα δούμε παρακάτω.

10.3 Είσοδος από τον Χρήστη

Γενικότερα, όποτε λαμβάνουμε είσοδο από τον χρήστη (ειδικά να μας ενδιαφέρουν web εφαρμογές) πρέπει να προσέχουμε πάρα πολύ και να ελέγχουμε την είσοδο που δεχόμαστε, είτε αυτό συνέβη κατά λάθος, είτε προσπάθησε να δοκιμάσει την ασφάλεια της εφαρμογής μας. Ο πιο απλός τρόπος είναι να ζητάμε ξανά την είσοδο από τον χρήστη εφόσον αυτή που μας παρείχε δεν ταιριάζει σε ορισμένα κριτήρια που έχουμε θέσει.

```
while True:
    try:
        x = int(input("Please enter a number: "))
        break
    except ValueError:
        print("Not a number. Try again...")
print(x)
```

Στο παραπάνω παράδειγμα, ζητάμε από τον χρήστη να εισάγει ένα ακέραιο. Όσο ο χρήστης εισάγει ως είσοδο, ένα αλφαριθμητικό που δεν μπορεί να μετατραπεί σε ακέραιος, τότε του ζητάμε ξανά την είσοδο. Πιο αναλυτικά, η συνάρτηση input εμφανίζει ένα μήνυμα και δέχεται ως είσοδο από το πε-

[1] Για παράδειγμα, αν αυτός ο κώδικας περιεχόταν μέσα σε μια συνάρτηση, θα μπορούσαμε να καλούμε αυτή την συνάρτηση μέσα σε ένα try... except... μπλοκ και να χειριζόμασταν και εκεί την εξαίρεση.

ριβάλλον ένα αλφαριθμητικό. Στην συνέχεια, η συνάρτηση int προσπαθεί να μετατρέψει αυτό το αλφαριθμητικό σε έναν ακέραιο. Αν η μετατροπή γίνει επιτυχώς, αυτός ο ακέραιος εκχωρείται στην μεταβλητή x και βγαίνουμε από τον βρόγχο επανάληψης while, αλλιώς η δήλωση break δεν εκτελείται και πηγαίνουμε στο επόμενο μπλοκ except ValueError, όπου και τυπώνουμε ένα μήνυμα σφάλματος. Επειδή η δήλωση break δεν εκτελείται, θα ξαναζητηθεί είσοδος από τον χρήστη. Όταν τελικά αυτός δώσει την κατάλληλη είσοδο, τότε θα τυπωθεί ο αριθμός που εισήγαγε με την συνάρτηση print.

10.4 Μηχανισμός

Πιο συγκεκριμένα, μέχρι τώρα έχουμε δει τα εξής για το πως λειτουργεί ο μηχανισμός των εξαιρέσεων στην Python.

1. Οι δηλώσεις του try μπλοκ εκτελούνται.
2. Αν καμία εξαίρεση δεν συμβεί, *δεν* εκτελείται το μπλοκ του except.
3. Μόλις συμβεί μια εξαίρεση στο try μπλοκ, *σταματάει* η εκτέλεση του και πάμε στο except.
4. Ψάχνουμε για except μπλοκ του συγκεκριμένου τύπου εξαίρεσης.
5. Αν βρεθεί, εκτελούνται οι δηλώσεις του.
6. Αν δεν βρεθεί, η εξαίρεση περνάει στο εξωτερικό try μπλοκ.
7. Αν η εξαίρεση δεν χειριστεί καθόλου, σταματάει η εκτέλεση του προγράμματος.

10.4.1 try:. . . else: . . .

Ωστόσο, υπάρχουν επιπρόσθετες ευκολίες πέρα από αυτές που περιγράφονται στον παραπάνω γενικό μηχανισμό και δεν έχουμε εξερευνήσει ακόμα. Μια καλή πρακτική είναι να έχουμε όσο το δυνατόν λιγότερο κώδικα μέσα σε ένα try μπλοκ. Και όπως πάντα, έτσι και τώρα η Python μας διευκολύνει παρέχοντας την δομή else, η οποία δεν χρησιμοποιείται μόνο στο if, όπως ίσως να νόμιζε κάποιος.

```
try:
    f = open('myfile.txt')
except IOError as err:
    print("I/O error: {}".format(err))
else:
    s = f.readline()
    print('The first line of f is:', s)
    f.close()
```

Ο κώδικας μέσα σε ένα else μπλοκ εκτελείται μόνο στην περίπτωση που δεν έχει συμβεί καμία εξαίρεση. Έτσι, στο συγκεκριμένο παράδειγμα, θα εκτελεστεί μόνο εφόσον το αρχείο άνοιξε κανονικά χωρίς να συμβεί κάποιο σφάλμα.

Επομένως θα έπρεπε να τον συμπληρώσουμε ως εξής:

1. Εκτελείται ο κώδικας μέσα στο μπλοκ try.
2. Αν συμβεί εξαίρεση μεταφερόμαστε στο μπλοκ except.
3. Αλλιώς (αν δεν συμβεί εξαίρεση) πηγαίνουμε στο μπλοκ else.

Προσοχή πρέπει να δοθεί στο γεγονός ότι το else πρέπει να είναι μετά από όλες τις προτάσεις except που πιθανώς υπάρχουν.

10.4.2 finally

Τέλος, άλλη μια χρήσιμη δομή αποτελεί το finally, που μας επιτρέπει να εκτελέσουμε κώδικα, ανεξαρτήτως αν έχει συμβεί εξαίρεση ή όχι.

```
# how many times we have requested for input
times = 0

while True:
    try:
        x = int(input("Please enter a number: "))
        break
```

```
    except ValueError:
        print("Not a number. Try again...")
    finally:
        times += 1
print('Requested input', times)
```

Στο παραπάνω παράδειγμα, μετράμε πόσες φορές ζητήθηκε από τον χρήστη να εισάγει έναν αλφαριθμητικό, μέχρι να εισάγει κάποιο το οποίο να μπορεί να μετατραπεί σε αριθμό. Έτσι, λόγω της πρότασης finally, είτε συμβεί εξαίρεση, είτε όχι μπορούμε να καταμετρήσουμε κάθε φορά που ο χρήστης εισήγαγε ένα αλφαριθμητικό. Η συγκεκριμένη δομή είναι ιδιαίτερα χρήσιμη όποτε θέλουμε να απελευθερώσουμε πόρους ανεξαρτήτως αν μπορέσαμε να τους χρησιμοποιήσουμε ή όχι λόγω κάποιας εξαίρεσης.

10.5 Δημιουργία Εξαιρέσεων

Μπορούμε να εγείρουμε ή και να δημιουργήσουμε τις δικές μας εξαιρέσεις ανάλογα με τις ανάγκες της εφαρμογής μας.

10.5.1 Ορίσματα Εξαιρέσεων

Όλες οι εξαιρέσεις[2] κληρονομούν από μια κλάση που ονομάζεται Exception. Τα ορίσματα των εξαιρέσεων μας βοηθούν με μια περιγραφή του τι ακριβώς συνέβη όταν έγινε μια εξαίρεση. Τα στιγμιότυπα αυτής της κλάσης έχουν μια μεταβλητής που ονομάζεται args και περιέχει τα ορίσματα που μπορεί να λαμβάνει μια εξαίρεση. Στο ακόλουθο παράδειγμα θα δούμε πως μπορούμε να τυπώσουμε αυτά τα ορίσματα.

```
try:
    raise Exception('Can you handle it?')
except Exception as inst:
    print(inst.args)
    print(inst)
```

[2]για την ακρίβεια αυτές που φτιάχνουμε εμείς ή χρησιμοποιούμε συνήθως

Καταρχάς, εγείρουμε μια εξαίρεση τύπου Exception. Σε αυτή την εξαίρεση περνάμε το όρισμα 'Can you handle it?'. Αφού χειριστούμε το στιγμιότυπο της εξαίρεσης που δημιουργείται χρησιμοποιώντας τη λέξη κλειδί as ώστε να ονομάσουμε το στιγμιότυπο αυτό ως inst, τυπώνουμε τα ορίσματα της με δυο τρόπους. Αρχικά μέσα της μεταβλητής args και στην συνέχεια απευθείας, τυπώνοντας την εξαίρεση. Ο δεύτερος τρόπος καθίσταται δυνατός καθώς υλοποιείται η μέθοδος __str__() την οποία θα δούμε και παρακάτω.

10.5.2 Εγείροντας Εξαιρέσεις (raise)

Θεωρητικά, όλες οι εξαιρέσεις αποτελούν αντικείμενα κλάσεων ή κλάσεις που κληρονομούν από την BaseException. Πρακτικά όμως, όπως ήδη έχουμε αναφέρει, όλες οι εξαιρέσεις που χρησιμοποιούμε κληρονομούν από μια υποκλάση της που καλείται Exception. Μπορούμε λοιπόν να εγείρουμε στιγμιότυπα εξαιρέσεων μέσω της λέξης κλειδί raise.

```
>>> raise NameError('TesseraPoulakiaKa8ontai')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: TesseraPoulakiaKa8ontai
```

Ή ακόμα και κλάσης εξαιρέσεων χωρίς να πρέπει αναγκαστικά να προσδιορίσουμε το στιγμιότυπο.

```
>>> raise NameError
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError
```

10.5.3 Δημιουργία Εξαιρέσεων από τον χρήστη

Μπορούμε να δημιουργήσουμε δικές μας εξαιρέσεις όπως θα δημιουργούσαμε οποιαδήποτε άλλη κλάση, με την διαφορά όμως ότι πρέπει να κληρονομούμε από την κλάση Exception.

```
>>> class MyError(Exception):
```

```
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return repr(self.value)
```

Πέρα από την συνάρτηση αρχικοποίησης __init__, χρήσιμη είναι και η συνάρτηση __str__ η οποία μας επιτρέπει να τυπώνουμε απευθείας μια εξαίρεση και να βλέπουμε τα αντίστοιχα μηνύματα λάθους.

Μόλις δημιουργήσουμε μια κλάση εξαιρέσεων, μπορούμε να την χειριστούμε κανονικά ή να εγείρουμε μια εξαίρεση από αυτή, όπως θα κάναμε και για οποιαδήποτε άλλη εξαίρεση.

```
>>> try:
...     raise MyError(4)
... except MyError as e:
...     print('My exception occurred:', e)
...
My exception occurred: 4
```

```
>>> raise MyError('oops!')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.MyError: 'oops!'
```

10.6 Σύγκριση με if . . . else

Οι εξαιρέσεις ουσιαστικά αλλάζουν την ροή του προγράμματος υπό ορισμένες προϋποθέσεις. Θα μπορούσε λοιπόν κάποιος να αναρωτηθεί αν θα μπορούσε να τις χρησιμοποιήσει έτσι ώστε να υποκαταστήσει δομές ελέγχου όπως το if . . . else. Γενικά ισχύει ο ακόλουθος κανόνας.

Κανόνας 10.6.1. Οι εξαιρέσεις χρησιμοποιούνται στις γλώσσες προγραμματισμού ώστε να κάνουν πιο ευανάγνωστο τον έλεγχο λαθών που προκύπτουν όπως για παράδειγμα η έλλειψη κάποιου αρχείου, η διακοπή της σύνδεσης με το διαδίκτυο ή και το τέλος αρχείου. Επομένως, οι εξαιρέσεις είναι για

πράγματα εξαιρετικά (δηλαδή συμβαίνουν σπάνια). Αντίθετα, αν κάτι συμβαίνει αρκετά συχνά, μάλλον πρέπει να το περιλάβουμε ως περίπτωση μέσα στον γενικότερο κανόνα και όχι ως εξαίρεση.

Ο παραπάνω κανόνας επιβεβαιώνεται και από την εξέταση του χρόνου εκτέλεσης συγκριτικά κώδικα που χρησιμοποιεί εξαιρέσεις και κώδικα χωρίς εξαιρέσεις.

```
from timeit import Timer
from random import randint

VALUES = 10000
inverseprob = 2
d = dict([(x,y) for x, y in zip(range(VALUES),\
    range(VALUES, 2 * VALUES))] )


class MyError(Exception):
    def __init__(self, value):
        self.value = value
    def __str__(self):
       return repr(self.value)

def try_except(d):
    try:
        d['not']
    except KeyError:
        pass


def if_else(d):
    if 'not' in d:
        d['not']
    else:
        pass
```

```
def try_exceptYES(d):
    try:
        d[1]
    except KeyError:
        pass


def if_elseYES(d):
    if 1 in d:
        d[1]
    else:
        pass


def try_exceptMAYBE(d, inverseprob):
    s = randint(0, inverseprob)
    try:
        if s == 0: raise MyError(2)
        d[1]
    except MyError:
        pass


def if_elseMAYBE(d, inverseprob):
    s = randint(0, inverseprob)
    if s == 0: return
    else: d[1]



def fail():
    print("Unsuccessful look up")
```

```
    t = Timer("try_except(d)",\
        "from __main__ import try_except, d")
    sf = 'Execution time with exceptions: {} seconds'
    print(sf.format(t.timeit()/ 10**6))

    t = Timer("if_else(d)",\
        "from __main__ import if_else, d")
    sf = 'Execution time with if/else: {} seconds'
    print(sf.format(t.timeit()/ 10**6))



def success():
    print("Successful look up")
    t = Timer("try_exceptYES(d)",\
        "from __main__ import try_exceptYES, d")
    sf = 'Execution time with exceptions: {} seconds'
    print(sf.format(t.timeit()/ 10**6))

    t = Timer("if_elseYES(d)",\
        "from __main__ import if_elseYES, d")
    sf = 'Execution time with if/else: {} seconds'
    print(sf.format(t.timeit()/ 10**6))

def maybe():
    print("Successful under a probability")
    t = Timer("try_exceptMAYBE(d, inverseprob)",\
        "from __main__ import try_exceptMAYBE, d, inverseprob")
    sf = 'Execution time with exceptions: {} seconds'
    print(sf.format(t.timeit()/ 10**6))

    t = Timer("if_elseMAYBE(d, inverseprob)",\
        "from __main__ import if_elseMAYBE, d, inverseprob")
    sf = 'Execution time with if/else: {} seconds'
```

```
    print(sf.format(t.timeit()/  10**6))

def main():
    success()
    fail()
    maybe()


if __name__ == '__main__':
    main()
```

Κεφάλαιο 11

Γεννήτορες

> A woodpecker can peck twenty times on a thousand trees and get nowhere, but stay busy. Or he can peck twenty-thousand times on one tree and get dinner.
>
> Seth Godin

Οι γεννήτορες μας δίνουν τη δυνατότητα να κάνουμε τη δουλειά που χρειάζεται, τη στιγμή που χρειάζεται. Μπορούμε να τους σκεφθούμε ως συναρτήσεις, όπου όμως παράγουν τμηματικά την έξοδο τους, ανάλογα με το τι τους ζητάτε. Λόγω του οφέλους που αυτό μπορεί να έχει (δεν 'κολλάει' πια ανεξήγητα ένα πρόγραμμα όταν του ζητάμε κάτι φαινομενικά απλό, χρησιμοποιείται όση μνήμη χρειάζεται κοκ), η βιβλιοθήκη της Python 3 κάνει εκτεταμένη χρήση της.

11.1 Επαναλήπτες (Iterators)

Για να καταλάβουμε πως λειτουργούν οι γεννήτορες, πρέπει πρώτα να καταλάβουμε τους επαναλήπτες (iterators). Όταν προσπελαύνουμε τα στοιχεία μιας λίστας ένα ένα, χρησιμοποιούμε επαναλήπτες.

```
mylist = [1, 2, 3]

for i in mylist :
```

```
    print(i)
```

Επαναλήπτης (iterator) είναι ο όρος που χρησιμοποιούμε για να καταδείξουμε ότι ένα αντικείμενο έχει μια next() μέθοδο.

11.1.1 Πώς δουλεύουν οι for βρόγχοι

Όποτε γράφουμε:

```
for i in mylist :
    ...loop body...
```

η Python κάνει τα ακόλουθα βήματα:

1. Παίρνει ένα επαναλήπτη για την next(). Καλεί την iter(mylist) η οποία επιστρέφει ένα αντικείμενο με την μέθοδο next().

2. Χρησιμοποιεί τον επαναλήπτη για να διατρέξει όλα τα αντικείμενα στην λίστα. Έτσι το i παίρνει επαναληπτικά όλες τις τιμές που βρίσκονται στην λίστα mylist. Για να γίνει αυτό, καλείται συνεχώς η συνάρτηση next() στον επαναλήπτη i που επιστρέφεται από το πρώτο βήμα. Η επιστρεφόμενη τιμή ανατίθεται στο i και το σώμα του βρόγχου εκτελείται. Εάν η εξαίρεση StopIteration προκύψει από την next() σημαίνει πως δεν υπάρχουν άλλες τιμές να ανακτηθούν από την mylist και έτσι ο βρόγχος τερματίζει.

11.2 Δημιουργία γεννητόρων

Οι γεννήτορες (generators) δημιουργούνται όταν χρησιμοποιούμε την λέξη κλειδί yield αντί της return. Μπορούμε να τις αντιμετωπίσουμε σαν τις κλασικές συναρτήσεις με μόνο μια διαφορά. Ένας γεννήτορας, επιστρέφει μια τιμή με την yield. Όταν ξανακλειθεί, συνεχίζει από την κατάσταση που ήταν μόλις κλήθηκε η yield, μέχρι να φθάσει ξανά σε κάποιο άλλο yield. Έτσι, μπορεί να χρησιμοποιηθεί για να παράγονται δυναμικά τιμές, καταλαμβάνοντας έτσι μικρότερο χώρο στην μνήμη, αφού δεν χρειάζεται να παραχθούν όλες και να επιστραφούν.

Ορισμός 11.2.1. *Γεννήτορας* μια ειδική συνάρτηση η οποία χρησιμοποιείται για την παραγωγή iterators. Κάθε φορά που καλούμε έναν γεννήτορα, συνεχίζει την εκτέλεση του από το σημείο που είχε μείνει στην προηγούμενη εκτέλεση του. Έτσι, η έξοδος του παράγεται τμηματικά. Συνεπώς είναι ιδιαίτερα χρήσιμος όταν η παραγόμενη έξοδος είναι πολύ μεγάλη και μπορούμε να την τμηματοποιήσουμε.

Στο ακόλουθο παράδειγμα βλέπουμε μια συνάρτηση η οποία δουλεύει σαν την range() με την διαφορά ότι αν το δεύτερο όρισμα (end) είναι μικρότερο από το πρώτο (start), τότε θα κάνει την πράξη modulo μέχρι να φθάσει σε αυτό.

```
def modulo_yield(start, end, modulo):
    """ Create a list of all the number from start_{modulo}
        until end_{modulo} is reached.
    """
    if (start > end):
        a = start % modulo
        b = (end % modulo) + modulo
    else:
        a = start
        b = end

    for num in range(a, b):
        yield num % modulo
```

Ο έλεγχος αν start > end γίνεται μόνο μια φορά, όταν καλείται για πρώτη φορά ο παραπάνω γεννήτορας. Στην συνέχεια, η yield βρίσκεται μέσα σε έναν βρόγχο for. Αυτό πρακτικά σημαίνει, ότι κάθε φορά που καλείται ο γεννήτορας, θα βρίσκεται πάντα μέσα σε αυτόν τον βρόχο, παράγοντας έτσι την επόμενη τιμή.

Ένα άλλο παράδειγμα είναι η δημιουργία της ακολουθίας των Fibonacci αριθμών.

```
def fibonacci(num):
    a = 0
```

```
    yield a
    b = 1
    yield b

    for i in range(2, num):
        b, a = a + b, b
        yield b

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

Τέλος, μπορούμε να δημιουργήσουμε γεννήτορες μέσω εκφράσεων γεννητόρων σε αντιστοιχία με τις lists comprehensions.

```
primes = [1, 2, 3, 5, 7]

square_primes = (i**2 for i in primes)
print(next(square_primes))
print(next(square_primes))
print(next(square_primes))
```

Ένα γεννήτορας, επομένως, μπορεί να θεωρηθεί ως ένα αντικείμενο στο οποίο μπορούμε να καλέσουμε την συνάρτηση next(). Κάθε κλήση της συνάρτησης next() προκαλεί την επιστροφή του επόμενου αντικειμένου μέχρι να δημιουργηθεί μια εξαίρεση τύπου StopIteraton η οποία και σημαίνει το τέλος των αντικειμένων των οποίων μπορούν να παραχθούν από τον συγκεκριμένο γεννήτορα.

Παρότι το αντικείμενο του γεννήτορα δημιουργείται μόνο μια φορά, ο κώδικας του δεν τρέχει μόνο μια φορά. Κάθε φορά τρέχει ένα μέρος του κώδικα του (μέχρι να συναντηθεί το yield) και κρατιέται η κατάσταση του στη μνήμη. Την επόμενη φορά συνεχίζει η εκτέλεση από το ίδιο σημείο, από την κατάσταση που είχε σταματήσει μέχρι το επόμενο yield.

11.3 Γράφοντας κώδικα φιλικό προς τους γεννήτορες

Ορισμένα πλεονεκτήματα των γεννήτορων είναι:

- Μικρότερη κατανάλωση μνήμης όταν χρειάζεται να παραχθούν πολλά και μεγάλα αντικείμενα. Αντί να πρέπει να δημιουργηθεί μια συνάρτηση που θα επιστρέφει μια τεράστια λίστα γεμάτη αντικείμενα, μπορούμε να γράψουμε έναν γεννήτορα ο οποίο θα παράγει τα απαραίτητα αντικείμενα όταν αυτά χρειαστούν (on the fly). Έτσι, μπορούμε να παράγουμε σειρές αντικειμένων που αλλιώς δεν θα χωρούσαν στη μνήμη.

- Είναι ένας πολύ αποτελεσματικός τρόπος να γίνει parsing σε μεγάλα αρχεία χωρίς να χρειάζεται να τα φορτώσουμε ολόκληρα στη μνήμη. Για παράδειγμα μπορούμε να τα διαβάζουμε γραμμή γραμμή και να τα επεξεργαζόμαστε.

- Αποτελούν έναν φυσικό τρόπο να περιγραφούν άπειρες σειρές δεδομένων. Παραδείγματος χάρη θα μπορούσαμε να δημιουργήσουμε έναν γεννήτορα που να μπορεί να παράγει σιγά σιγά κάθε αριθμό Fibonacci.

Πολλές φορές, συναρτήσεις που γράφουμε περιμένουν ως είσοδο κάποια λίστα ενώ θα μπορούσαν να δεχθούν και κάποιον γεννήτορα στην θέση της βελτιώνοντας έτσι την χρήση της κύριας μνήμης του προγράμματος. Για αυτό τον λόγο, αν δεν απαιτείται τυχαία προσπέλαση στα στοιχεία μιας λίστας αλλά αρκεί μια διάτρεξη π.χ. μέσω ενός βρόγχου for, τότε καλό είναι ο κώδικας μας να μπορεί να δουλέψει και με τους δύο τύπους εισόδου. Για αυτό το λόγο υπάρχουν ορισμένοι κανόνες που θα μπορούσαμε να ακολουθούμε:

1. Να μην χρησιμοποιείται η συνάρτηση len(). Πολλές φορές την χρησιμοποιούμε μόνο και μόνο για να προσδιορίσουμε πότε τελείωσε η διάτρεξη της λίστας (αντίστοιχα θα ήταν η παραγωγή των αντικειμένων από τον γεννήτορα) ενώ αυτό μπορεί να γίνει αυτόματα από το for βρόγχο.

2. Να μην ζητούνται αυθαίρετα στοιχεία (πχ το έβδομο στοιχείο) όταν τελικά θα τα προσπελαύσουμε όλα και δεν μας ενδιαφέρει η σειρά τους.

Ένα κλασικό παράδειγμα που συνοψίζει τους παραπάνω κανόνες, είναι η αποφυγή του:

```
for i in range(0, len(l)):
    e = l[i]
    ...
```

και η αντικατάσταση του με:

```
for i, e in enumerate(l):
    ...
```

που έχει ακριβώς το ίδιο αποτέλεσμα και είναι και πιο κομψό.

11.4 Προσπέλαση συγκεκριμένου στοιχείου γεννήτορα

Η παρακάτω τεχνική είναι αρκετά προχωρημένη, ωστόσο μπορεί να μας φανεί χρήσιμη σε ορισμένες περιπτώσεις και είναι ένα καλό παράδειγμα για την περαιτέρω κατανόηση της λειτουργίας των γεννήτορων συναρτήσεων. Πριν προχωρήσουμε στη τεχνική, καλό είναι να θυμηθούμε πως ένας γεννήτορας δεν συγκρατεί τα προηγούμενα στοιχεία που έχει παράγει, ούτε ξέρει πόσα είναι αυτά (εκτός βέβαια και αν εμείς έχουμε φροντίσει ειδικά για αυτό, που όμως έτσι αυξάνεται η κατανάλωση μνήμης). Επομένως δεν γνωρίζει ποιο στοιχείο κατά σειρά επιστρέφεται τώρα (αν είναι το 1ο, ή το 17ο).

Το πρόβλημα που θέλουμε να λύσουμε είναι η επιστροφή ενός συγκεκριμένου στοιχείου μετά από αυτό που μόλις μας επέστρεψε ο γεννήτορας. Στο ακόλουθο παράδειγμα θα δούμε πως μπορούμε να βρούμε τον 10ο αριθμό Fibonacci, και στην συνέχεια να εκτυπώσουμε τον 20ο, ή και ένας εύρος τιμών Fibonacci. Για αυτό τον σκοπό θα δημιουργήσουμε μια κλάση όπως φαίνεται ακολούθως. Η συνάρτηση fibonacci είναι αυτή που είδαμε λίγο παραπάνω.

```
import itertools
class Indexable(object):
    def __init__(self, it):
```

```
        self.it = it
    def __iter__(self):
        for elt in self.it:
            yield elt
    def __getitem__(self,index):
        try:
            return next(itertools.islice(self.it, index,\
                index + 1))
        # support for [start:end:step] notation
        except TypeError:
            return list(itertools.islice(self.it,\
                index.start, index.stop, index.step))

it = Indexable(fibonacci(45))
# prints the 10-th element
print(it[10])
# 55

# prints the element after 10 elements from the current
print(it[10])
# 10946

# prints the elements starting after where we had stopped
print(it[2:12:2])
# [46368, 121393, 317811, 832040, 2178309]
```

Κεφάλαιο 12

Κανονικές εκφράσεις

> Είναι ένας πραγματικός άνθρωπος των γραμμάτων, δουλεύει στο ταχυδρομείο.
>
> Ανώνυμος

Οι κανονικές εκφράσεις αποτελούν ένα από τα κύρια εργαλεία που έχουμε στη διάθεση μας όταν θέλουμε να βρούμε κομμάτια κειμένου, τα οποία μπορούν να εκφραστούν είτε ως αλφαριθμητικά που έχουμε δει, είτε ως γενικότερα ως κλάσεις αλφαριθμητικών (πχ όλα τα αλφαριθμητικά που αναπαριστούν δεκαδικούς αριθμούς). Αυτές τις κλάσεις αλφαριθμητικών τις ονομάζουν πρότυπα (pattern).

Υπάρχουν δυο κύριες χρήσεις των κανονικών εκφράσεων στην Python. Η *αναζήτηση* και το *ταίριασμα*. Στην αναζήτηση, μας ενδιαφέρει να βρούμε την πρώτη φορά που συναντάμε ένα πρότυπο. Στο ταίριασμα, βρίσκουμε όλες θέσεις όπου μπορεί να βρίσκεται το πρότυπο που αναζητούμε.

12.1 Αναζήτηση

Χρησιμοποιώντας τις τετράγωνες αγκύλες μπορούμε να ορίσουμε μια κλάση από χαρακτήρες[1] που μπορεί να ταιριάξει με κάποιον χαρακτήρα. Στο α-

[1] Εδώ ως κλάση χαρακτήρων ουσιαστικά εννοούμε περισσότερους από ένα χαρακτήρες που μπορούν να ταιριάξουν σε κάποια θέση του προτύπου που αναζητάμε.

κόλουθο παράδειγμα ορίζουμε την κλάση με τους χαρακτήρες που αντιστοιχούν σε αριθμούς.

Αντίθετα, όλοι οι χαρακτήρες που βρίσκονται εκτός των αγκυλών μπορούν να ταιριάξουν μόνο αν βρεθεί ακριβώς ο ίδιος χαρακτήρας στο αλφαριθμητικό στο οποίο ψάχνουμε. Έτσι, στο επόμενο παράδειγμα ψάχνουμε αριθμούς που ξεκινάνε με 4.

```
import re


def num_start4(text):
    """ Check whether any number which starts with 4. """
    pattern = re.compile(r'\b4[0-9]')

    return bool(pattern.search(text))

a = num_start4("asdf 1234")   # False
a = num_start4("asdf 4123")   # True
```

Μπορούμε να παρατηρήσουμε πως χρησιμοποιήθηκε ο ειδικός χαρακτήρας \β. Ο ειδικός αυτός χαρακτήρας ταιριάζει όρια των λέξεων. Κατά αυτό τον τρόπο, κάνουμε σίγουρο ότι θα βρούμε αριθμούς που αρχίζουν από 4 όπως το 4321 και όχι αριθμούς που απλά περιέχουν το 4 όπως το 3421.

Αντίστοιχα λειτουργεί και η συνάρτηση φινδαλλ η οποία όμως επιστρέφει όλες τις περιπτώσεις όπου βρέθηκε στο αλφαριθμητικό στο οποίο ψάχνουμε το πρότυπο. Στο επόμενο παράδειγμα, επιστρέφονται όλοι οι αριθμοί που βρίσκονται μέσα στο αλφαριθμητικό.

```
import re

a = "3141/1000 reminds me of pi"
b = re.findall("[0-9]", a)
```

Κεφάλαιο 13

Περιγραφείς

> I can't understand why people are frightened of new ideas. I'm frightened of the old ones.
>
> Josh Cage

Οι περιγραφείς, αν χρησιμοποιηθούν σωστά, μας διευκολύνουν ώστε τα αντικείμενα που κατασκευάζουμε και επιτελούν μια λειτουργία, να είναι ακόμα πιο απλά στη χρήση τους ενθαρρύνοντας τον προγραμματιστή να ασχοληθεί με το πρόβλημα που πρέπει να λύσει και όχι με τις πιθανές ιδιοτροπίες της διεπαφής τους.

13.1 Εισαγωγή

Οι κλάσεις κληρονομούν (ή παράγονται) από την βασική κλάση object. Σε αυτές τις κλάσεις εισήχθησαν πολλά νέα χαρακτηριστικά όπως το πρωτόκολλο των περιγραφέων (descriptors). Οι περιγραφείς δίνουν την δυνατότητα στους προγραμματιστές να δημιουργούν να διαχειρίζονται τα χαρακτηριστικά (attributes) των αντικειμένων των κλάσεων που δημιουργούν. Όταν λέμε διαχειρίζονται, εννοούμε πριν την προσπέλαση τους να εφαρμόζονται ορισμένες μέθοδοι που να προσδιορίζουν την συμπεριφορά τους. Ένα παράδειγμα χρήσης μιας τέτοιας δυνατότητας είναι όταν θέλουμε να μην επιτρέπουμε την διαγραφή ενός χαρακτηριστικού ή αν θέλουμε να περιορίσουμε το εύρος τιμών στο οποίο μπορεί να κινηθεί ένα άλλο χαρακτηριστικό. Ακόμα, ίσως

κάποιος να ήθελε όποτε ενημερώνεται ένα χαρακτηριστικό x να αλλάζει αντίστοιχα και ένα άλλο χαρακτηριστικό y.

Για όσους είναι εξοικειωμένοι με άλλες γλώσσες προγραμματισμού, αυτός ο τύπος προσπέλασης συχνά αναφέρεται με τα ονόματα "getters" και "setters" που σημαίνει οι συναρτήσεις που παίρνουν κάτι και οι συναρτήσεις που θέτουν κάτι. Στις περισσότερες όμως γλώσσες, οι παραπάνω συναρτήσεις υπονοούν την χρήση ιδιωτικών μεταβλητών και δημοσίων συναρτήσεων. Στην Python όμως δεν υπάρχει αυτή η διάκριση, μέσα στα πλαίσια απλοποίησης της γλώσσας αλλά και επειδή θεωρεί ότι ο προγραμματιστής είναι υπεύθυνος και τηρεί ορισμένους απλούς κανόνες. Έτσι, οι περιγραφείς είναι ένας προτεινόμενος τρόπος για να επιτευχθεί μια παρόμοια συμπεριφορά.

13.2 Ορισμοί

13.2.1 Μέθοδοι

Ας ρίξουμε μια πιο προσεκτική ματιά στο πρωτόκολλο των περιγραφέων και ας δούμε πως μπορούμε να δημιουργήσουμε έναν. Ένας περιγραφέας υλοποιείται από τρεις μεθόδους:

- __get__(self, instance, owner)
- __set__(self, instance, value)
- __delete__(self, instance)

Αυτές οι τρεις μέθοδοι αντιπροσωπεύουν τις τρεις βασικές πράξεις που πραγματοποιούνται σε χαρακτηριστικά (attributes) αντίστοιχα:

- *Ανάκτηση:* Όποτε αναζητούμε τιμή από το αντικείμενο καλείται η μέθοδος __get__(self, instance, owner). Επιστρέφει την (υπολογισμένη) τιμή που ζητήθηκε ή εγείρει μια εξαίρεση τύπου AttributeError.
- *Ανάθεση:* Όποτε αναθέτουμε τιμή στο αντικείμενο καλείται η μέθοδος __set__(self, instance, value).
- *Διαγραφή:* Όποτε επιθυμούμε να διαγράψουμε το αντικείμενο καλείται η μέθοδος __delete__(self, instance).

Παράμετροι

Αν εξαιρέσουμε την παράμετρο self που πρέπει να είναι η πρώτη παράμετρος (όρισμα) σε κάθε μέθοδο που ανήκει σε μια κλάση, έχουμε τρία διαφορετικά ορίσματα.

- *owner:* Είναι η κλάση στην οποία ανήκει το αντικείμενο από το οποίο καλέστηκε η μέθοδος.
- *instance:* Είναι το αντικείμενο (ή στιγμιότυπο) από το οποίο καλέστηκε η μέθοδος. Αν είναι None σημαίνει ότι καλέστηκε από την κλάση αντί από ένα στιγμιότυπο της.
- *value:* Η τιμή η οποία θα χρησιμοποιηθεί για να τεθεί ένα χαρακτηριστικό.

Κεφάλαιο 14

Απλό GUI με tkinter

> Κάποιοι μου λένε πως επιτέλους βρίσκω την αναγνώριση που μου αξίζει. Σε αυτούς απαντώ: Μαμά σταμάτα. Με κάνεις ρεζίλι.
>
> Ανώνυμος

Με τον όρο GUI αναφερόμαστε στην γραφική διεπαφή (Graphical User Interface) που έχει η εφαρμογή μας με τον χρήστη. Για τον σκοπό αυτό υπάρχουν αρκετές βιβλιοθήκες που λειτουργούν σε διαφορετικές πλατφόρμες χωρίς αλλαγή στον κώδικα τους. Μερικές από αυτές είναι:

1. wxPython
2. pyQT
3. PyGTK
4. tkinter

Τις τρεις πρώτες ίσως να τις γνωρίζετε και από άλλες γλώσσες, καθώς έχουν γίνει οι απαραίτητες ενέργειες γύρω από τον κώδικα των αντίστοιχων βιβλιοθηκών ώστε να μπορούν να χρησιμοποιηθούν από το περιβάλλον της γλώσσας Python. Αντίθετα, η tkinter (= Tk interface) έρχεται μαζί με την Python, είναι σχετικά απλή ενώ και δεν θα χρειαστείτε καμία επιπλέον βιβλιοθήκη για να

την χρησιμοποιήσετε. Χαρακτηρίζεται για την απλότητα της και στηρίζεται πάνω σε Tcl/Tk. Το tkinter προσφέρει μια αντικειμενοστραφή διεπαφή προς την εργαλειοθήκη Tcl/Tk.

14.1 Βασικές Έννοιες

Το πιο απλό πρόγραμμα που μπορούμε να δημιουργήσουμε είναι:

```
from tkinter import *

root = Tk()

w = Label(root, text="Hello world!")
w.pack()

root.mainloop()
```

Παρατηρούμε πως αφού εισάγουμε από την βιβλιοθήκη tkinter ό,τι χρειαζόμαστε, μπορούμε να δημιουργήσουμε ένα απλό παράθυρο καλώντας τον δημιουργό της κλάσης. Στην συνέχεια προσθέτουμε μια ετικέτα (label) την οποία εμφανίζουμε με την συνάρτηση pack(). Τέλος, το πρόγραμμα μας εισέρχεται σε έναν βρόγχο (mainloop) όπου περιμένει από τον χρήστη ένα καινούργιο συμβάν.

14.2 Δημιουργία αριθμομηχανής

Παρατίθεται ο κώδικας μια απλής αριθμομηχανής, όπου υλοποιούνται μόνο οι τέσσερις βασικές πράξεις. Έτσι, η λογική του προγράμματος είναι σχετικά απλή, με στόχο να δοθεί έμφαση στην δημιουργία του γραφικού περιβάλλοντος. Μια βασική αρχή που πρέπει να έχουμε υπόψη μας, είναι ο διαχωρισμός της λογικής του προγράμματος από την διεπαφή με τον χρήστη. Αυτό βοηθάει στην συντηρησιμότητα και επεκτασιμότητα του κώδικα μας, καθώς επίσης δρα αποτρεπτικά για τον κακό σχεδιασμό από άποψη αρχιτεκτονικής λογισμικού.

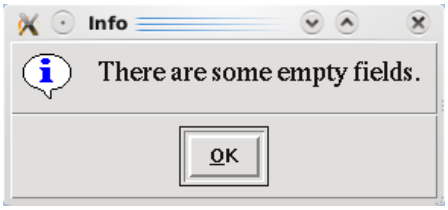
```
from tkinter import *
from tkinter.messagebox import *


class Calculator(Frame):
    """ Describes the main behaviour of a calculator
    """



    def d_result(self, c):
        """ Displays the result of an operation
        """

        self.label3.config(text = 'The result is ' + str(c))
        self.label3.pack()


    def d_add(self):
        """ Handles the addition in the GUI
        """
        a, b = self.getNum()
        c = self.addition(a, b)

        self.d_result(c)

    def d_sub(self):
        """ Handles the substraction in the GUI
        """
        a, b = self.getNum()
        c = self.substraction(a, b)

        self.d_result(c)
```

```
    def d_mul(self):
        """ Handles the multiplication in the GUI
        """
        a, b = self.getNum()
        c = self.multiplication(a, b)

        self.d_result(c)

    def d_div(self):
        """ Handles the division in the GUI
        """
        a, b = self.getNum()
        c = self.division(a, b)

        self.d_result(c)



    def addition(self, a, b):
        """ Add numbers a, b and return their result
        """
        c = a + b
        return c

    def substraction(self, a, b):
        """ Substract a - b and return the result
        """
        c = a - b
        return c

    def multiplication(self, a, b):
        """ Multiply a * b and return its result
        """
        c = a * b
```

```
        return c

    def division(self, a, b):
        """ Divide a / b and return its result
        """
        try:
            # if b != 0
            c = a / b
        except ZeroDivisionError:
            # if b == 0
            showinfo('Warning', 'Cannot divide by 0.')
            c = 0
        return c

    def getNum(self):
        """ Gets the two numbers on which the operation
        will be applied.
        In case of error, it returns both of them as zero
        and warns the user.
        """
        try:
            a = float(self.enter1.get())
            b = float(self.enter2.get())

        # if the fields were empty
        except ValueError:
            a, b = 0, 0
            showinfo('Info', 'There are some empty fields')

        return a, b


    def insertNum(self, label = "Insert a number:"):
        """ Draw the necessary elements for the insertion of
```

```
            two numbers
        """

        # The first number
        self.label1 = Label(self, text = label)
        self.label1.pack()
        # create the field for that number
        self.enter1 = Entry(self)
        self.enter1.pack()

        # the second number
        self.label2 = Label(self, text = label)
        self.label2.pack()
        # create the field for the new number
        self.enter2 = Entry(self)
        self.enter2.pack()


    def drawGUI(self):
        """
        It draws the GUI within the container from which is
        called. In this case, within the frame where the
        calculator belongs.
        """

        # draws the elements which are used to insert a number
        self.insertNum()

        # set the focus to enter1
        self.enter1.focus()

        # create the buttons
        self.button1 = \
        Button(self, text="add", command = self.d_add)
```

```
        self.button2 = \
        Button(self, text="sub", command = self.d_sub)
        self.button3 = \
        Button(self, text="mul", command = self.d_mul)
        self.button4 = \
        Button(self, text="div", command = self.d_div)

        # display them
        self.button1.pack(side=LEFT)
        self.button2.pack(side=LEFT)
        self.button3.pack(side=LEFT)
        self.button4.pack(side=LEFT)

        # create the label where we display the result
        self.label3 = Label(self)

        return root;


    def __init__(self, master = None):
        """
        The constructor is called with a parent widget.
        It creates the GUI for the calculator.
        """
        Frame.__init__(self, master)
        self.master.title("Calculator")
        self.pack()
        self.drawGUI()


root = Tk()
calc = Calculator(master = root)
calc.mainloop()
```



Σχήμα 14.1: showinfo('Info, 'There are some empty fields.')

Τα Entry χρησιμοποιούνται για την δημιουργία των πεδίων όπου εισάγονται οι αριθμοί ενώ τα Button για την δημιουργία των κουμπιών.

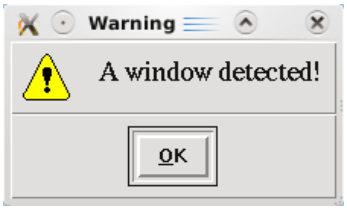
14.3 Αναδυόμενα παράθυρα διαλόγου

Τα αναδυόμενα παράθυρα διαλόγου pop-up μπορούν να χρησιμοποιηθούν για να ενημερώσουν τον χρήστη για ένα συμβάν, ή για να του ζητήσουν μια απλή επιλογή. Μπορούμε να τα δούμε σαν αντικαταστάτες των print που χρησιμοποιούν τα προγράμματα σε κονσόλα ώστε να ενημερώσουν σε συντομία για ένα γεγονός που συνέβη (και συνήθως διακόπτει την ροή του προγράμματος).

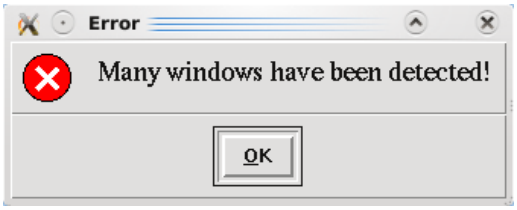
Έχουμε στην διάθεση μας επτά διαφορετικούς τύπους συνηθισμένων παραθύρων που μπορούμε εύκολα να χρησιμοποιήσουμε, αρκεί να εισάγουμε το tkinter.messagebox. Το αποτέλεσμα τους, εμφανίζεται στην αντίστοιχη εικόνα της κάθε συνάρτησης που αντιστοιχεί στα αναδυόμενα παράθυρα.

- showinfo()
- askokcancel()
- showwarning()
- showerror()
- askquestion()
- askyesno()
- askretrycancel()

Σχήμα 14.2: askokcancel('Go on?', 'Would you like to quit?')

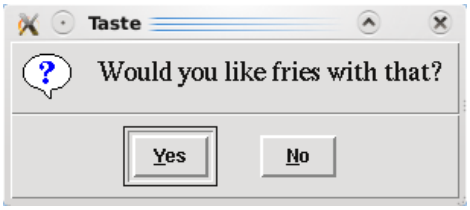


Σχήμα 14.3: showwarning('Warning', 'A window detected!')



Σχήμα 14.4: showerror ('Error', 'Many windows have been detected!')

Σχήμα 14.5: askquestion('Install', 'Would you like to install a door?')



Σχήμα 14.6: askyesno('Taste', 'Would you like fries with that?')

Σχήμα 14.7: askretrycancel('No media found', 'Would you like to try again?')

Τα ορίσματα που μπορούμε να περάσουμε με σειρά στις παραπάνω συναρτήσεις είναι:

- *Τίτλος:* Ο τίτλος του παραθύρου
- *Μήνυμα:* Το κείμενο που θα εμφανιστεί ως μήνυμα
- *Επιλογές:* Προαιρετικά. Χρησιμοποιείται για την επιλογή εικονιδίου, ποιο κουμπί να είναι προεπιλογή και άλλα.

Παρακάτω, ακολουθεί ένα μινιμαλιστικό παράδειγμα που μπορεί να χρησιμοποιεί κάποιος για να δοκιμάσει κάποιον από τους τύπους που αναφέραμε παραπάνω.

```
from tkinter import *
from tkinter.messagebox import *

showinfo('Info', 'There are some empty fields.')
showwarning('Warning', 'A window detected!')
showerror('Error', 'Many windows have been detected!')
askquestion('Install', 'Would you like to install a door?')
```

Στα παράθυρα όπου ο χρήστης έχει την δυνατότητα να διαλέξει κάποια επιλογή (πχ OK, ή άκυρο), ανάλογα με το τι διαλέγει επιστρέφεται από την συνάρτηση η τιμή true ή false αντίστοιχα.

Κεφάλαιο 15

Αποσφαλμάτωση

> All those who believe in psychokinesis raise my hand.
>
> Ανώνυμος

Δεν είναι λίγες οι φορές που καλούμαστε να αντιμετωπίσουμε μια ιδιάζουσα περίπτωση στο πρόγραμμα μας, η οποία όμως ξεφεύγει από τον αυστηρό στόχο όσον αφορά το πρόβλημα που έχουμε να λύσουμε και έχει να κάνει κυρίως με την διαχειριστική επίλυση περιπτώσεων όπως η απουσία ενός αρχείου. Σε αυτό το κεφάλαιο μελετάμε τις Εξαιρέσεις, που μας παρέχουν έναν τρόπο διαχωρισμού της λογικής του προγράμματος μας από τον κώδικα που αφορά αυτές τις περιπτώσεις.

15.1 Είδη σφαλμάτων

15.1.1 Συντακτικά σφάλματα

Το συντακτικό αναφέρεται στη δομή ενός προγράμματος και στους κανόνες αυτής. Ένα πρόγραμμα που δεν τηρεί το συντακτικό της γλώσσας δεν μπορεί να εκτελεστεί. Αν προσπαθήσουμε να εκτελέσουμε ένα πρόγραμμα που παραβιάζει αυτούς τους κανόνες, ο διερμηνές ή ο μεταφραστής θα παραπονεθούν για *συντακτικά λάθη*. Αυτό το είδος προβλημάτων είναι και τα πιο απλά

σχετικά και με τις τρεις κατηγορίες, γιατί ο διερμηνευτής ή ο μεταγλωττιστής μπορεί να μας ενημερώσει σε ποια γραμμή ακριβώς εντοπίζεται το σφάλμα.

15.1.2 Σφάλματα χρόνου εκτέλεσης

Τα σφάλματα χρόνου εκτέλεσης (run time errors) εμφανίζονται μόνο στην εκτέλεση του προγράμματος μας γιατί έχει συμβεί κάτι εξαιρετικό (τελείωσε η μνήμη, δεν έγινε σωστός χειρισμός μιας εξαίρεσης κοκ).

15.1.3 Λογικά σφάλματα

Αυτά είναι τα σφάλματα που ενώ εκτελείται το πρόγραμμα μας, παράγονται διαφορετικά από τα επιθυμητά αποτελέσματα. Αποτελούν την δυσκολότερη κατηγορία σφαλμάτων. Αν ο υπολογιστής μας μπορούσε να τα διορθώσει, θα μπορούσε να προγραμματίσει και μόνος τον εαυτό του!

15.2 Python Debugger

Ένα από τα πιο χρήσιμα έως προγράμματα για την συγγραφή ενός προγράμματος μεσαίου ή μεγαλύτερου μεγέθους είναι ένας debugger. Η Python έχει έναν debugger ο οποίος είναι διαθέσιμος με την μορφή ενός module, του pdb[1]. Συνήθως, οι μόνες στιγμές που χρησιμοποιούμε κάποιον debugger είναι όταν κάτι πηγαίνει στραβά και οι συνηθισμένες μας τακτικές (print, log messages) αδυνατούν να βρουν τη ρίζα του προβλήματος. Ευτυχώς για εμάς, η Python περιλαμβάνει στην βασική της βιβλιοθήκη έναν διαδραστικό αποσφαλματωτή (debugger) που μπορεί να βγάλει το 'φίδι' από την τρύπα στις δύσκολες στιγμές. Για να χρησιμοποιήσουμε τον debugger, πρέπει να εισάγουμε το κατάλληλο άρθρωμα.

```
import pdb
```

Ο διαδραστικός debugger εκτελεί τον κώδικα με έναν ελεγχόμενο τρόπο. Επιτρέπει ανάμεσα σε άλλα να:

- Εξετάζουμε ένα κομμάτι κώδικα ανά γραμμή κάθε φορά

[1] Python DeBugger

- Καλούμε και να επιστρέφουμε από συναρτήσεις
- Ορίζουμε σημεία διακοπής ροής του προγράμματος (breakpoints)
- Χρησιμοποιούμε απευθείας την δύναμη του κελύφους της Python.

15.2.1 Βηματική Εκτέλεση

Για την βηματική εκτέλεση του προγράμματος, χρησιμοποιούμε την συνάρτηση pdb.set_trace().

```
import pdb

a = "taspython"
b = ".eu\n"
pdb.set_trace()
c = "Because simplicity matters"
final = a + b + c
print(final)
```

Από το σημείο όπου χρησιμοποιείται η pdb.set_trace(), εκτελούμε την επόμενη δήλωση μέσω της εντολής "n" στο διαδραστικό κέλυφος που παρουσιάζεται μετά την εκτέλεση του προγράμματος. Μετά την πρώτη φορά που χρησιμοποιούμε το "n", μπορούμε να εκτελούμε κάθε φορά την επόμενη δήλωση πατώντας το πλήκτρο enter καθώς έτσι επαναλαμβάνεται η τελευταία εντολή αποσφαλμάτωσης. Για επιστρέψουμε από το περιβάλλον του debugger χρησιμοποιούμε το "q" που τερματίζει τον debugger ή το "c" που επιτρέπει την κανονική συνέχιση της εκτέλεσης του προγράμματος.

15.2.2 Συναρτήσεις

Όποτε προχωράμε προς την επόμενη προς εκτέλεση εντολή με το "n", τότε ο debugger συμπεριφέρεται σε κάθε δήλωση με τον ίδιο τρόπο, ανεξάρτητα αν καλεί κάποια συνάρτηση ή όχι. Αν νομίζουμε πως το πρόβλημα βρίσκεται εντός κάποιας συνάρτησης τότε μπορούμε να προχωράμε με το "s". Η συμπεριφορά του είναι ακριβώς η ίδια με το "n", με την εξαίρεση ότι μόλις

συναντά μια συνάρτηση που έχουμε γράψει εμείς, τότε συνεχίζεται η βηματική εκτέλεση για κάθε γραμμή του κώδικα αυτής της συνάρτησης.

Αν θέλουμε να επιστρέψουμε από μια συνάρτηση που κοιτάμε βηματικά μέσω της "s", τότε μπορούμε να χρησιμοποιήσουμε την "r". Με αυτή, συνεχίζεται η εκτέλεση του κώδικα χωρίς διακοπή μέχρι την δήλωση return της συγκεκριμένης συνάρτησης (ή μέχρι το τέλος της αν αυτή η δήλωση απουσιάζει).

15.2.3 Χαμένοι στην Διερμηνεία

Χρήσιμες είναι οι εντολές "l" και "p". Με την πρώτη μπορούμε να δούμε τις επόμενες γραμμές του κώδικα από το σημείο στο οποίο βρισκόμαστε, ενώ με την "p" ακολουθούμενη από το όνομα μιας μεταβλητής, μπορούμε να τυπώσουμε τα περιεχόμενα της.

Επιπρόσθετα σε αυτά, παρέχεται η δυνατότητα να γράψουμε απευθείας κώδικα σε Python μέσα από το περιβάλλον του αποσφαλματωτή. Έτσι, μπορούμε γρήγορα να θέσουμε τιμές σε μεταβλητές ώστε να δούμε αν όντως αυτές ευθύνονται για κάποιο σφάλμα ώστε να το διορθώσουμε στην συνέχεια, ή μπορούμε να δοκιμάσουμε απευθείας την λύση που νομίζουμε ότι θα λύσει το πρόβλημα χωρίς να χρειάζεται να ανατρέχουμε στο αρχείο του πηγαίου κώδικα για λίγες γραμμές μόνο και μόνο για να δοκιμάσουμε την λύση μας.

Αν τύχει και έχετε ονομάσει κάποια μεταβλητή σας με ένα όνομα που αντιστοιχεί σε κάποια εντολή, τότε μάλλον θα αντιμετωπίσετε προβλήματα με την συγγραφή κώδικα απευθείας σε Python. Για να αποφύγετε τέτοια κατατόπια, μπορείτε να αρχίζετε τις εντολές σας με ένα θαυμαστικός (!) υποδηλώνοντας έτσι στον debugger ότι η προς εκτέλεση εντολή είναι Python και όχι άμεση εντολή προς τον ίδιο.

15.2.4 Βασικές Λειτουργίες

Ο πίνακας 15.1 παρουσιάζει συνοπτικά τις βασικές λειτουργίες που μπορούμε να επιτελέσουμε στον debugger.

Εντολή	Λειτουργικότητα
import pdb	Εισαγωγή αρθρώματος debugger.
pdb.set_trace()	Εκκίνηση αποσφαλματωτή
<Enter>	Επανάληψη τελευταίας εντολής
q	Έξοδος (quit)
p	Εκτύπωση τιμής μεταβλητής (print)
c	Συνέχιση κανονικής εκτέλεσης (continue)
l	Εκτύπωση προς εκτέλεση κώδικα (listing)
s	Βηματική εκτέλεση υπορουτίνας (step into)
r	Επιστροφή από υπορουτίνα (return)

Πίνακας 15.1: Ανασκόπηση κύριων λειτουργιών βασικών δομών δεδομένων

Κεφάλαιο 16

Μέτρηση Χρόνου Εκτέλεσης

> The only reason for time is so that everything doesn't happen at once.
>
> Albert Einstein

Αρκετά συχνά θέλουμε να μετρήσουμε την χρόνο που παίρνει ένα συγκεκριμένο κομμάτι του προγράμματος μας έτσι ώστε να πάρουμε τις κατάλληλες υλοποιητικές αποφάσεις. Η χρονομέτρηση όμως της εκτέλεσης ενός προγράμματος είναι αρκετά πιο περίπλοκη διαδικασία από ό,τι ίσως να φαίνεται με μια πρώτη ματιά.

Το να γράψουμε μόνοι μας μια συνάρτηση που μετράει το χρόνο εκτέλεσης εμπεριέχει αρκετές δυσκολίες καθώς είναι μια εξαιρετικά περίπλοκη διαδικασία. Για παράδειγμα, στα σύγχρονα υπολογιστικά συστήματα, τρέχουν πολλές εφαρμογές παράλληλα, αξιοποιώντας μικρά κομμάτια επεξεργαστικού χρόνου. Πόσος λοιπόν είναι ο πραγματικός χρόνος εκτέλεσης του προγράμματος μας; Επίσης, ανάλογα και με τον τρόπο μέτρησης του χρόνου μπορεί να υπάρχουν διαφορές. Υπάρχει ενδεχόμενο να καλείται ο garbage collector την στιγμή που μετράμε τον χρόνο εκτέλεσης του προγράμματος, και αυτό να επηρεάζει το συνολικό χρόνο. Για αυτούς και για άλλους λόγους, καλό είναι να έχουμε ένα ενιαίο περιβάλλον με το οποίο παίρνουμε τις μετρήσεις τον προγραμμάτων μας.

Η Python, σύμφωνα και με την φιλοσοφία της ότι έρχεται με 'τις μπαταρίες να περιέχονται' ήδη, μας προσφέρει ανάμεσα στο υπόλοιπο πλούσιο σύνολο

βιβλιοθήκων, και το άρθρωμα timeit. Χρησιμοποιώντας το άρθρωμα timeit, δημιουργείται ένα απομονωμένο περιβάλλον, μέσα στο οποίο μπορούμε να πάρουμε εύκολα τις μετρήσεις μας. Επίσης, το συγκεκριμένο άρθρωμα αποκρύπτει λεπτομέρειες που αφορούν την πλατφόρμα στην οποία εκτελείται ο κώδικας μας, παρέχοντας μας ένα ενιαίο περιβάλλον εργασίας ανεξαρτήτως λειτουργικού συστήματος. Μπορούμε να το χρησιμοποιήσουμε τόσο μέσα στα προγράμματα μας, όσο και από την διεπαφή γραμμής εντολών.

16.1 Μέτρηση Χρόνου Εκτέλεσης Δηλώσεων

Η πιο απλή είναι η περίπτωση όπου θέλουμε να μετρήσουμε μόνο κάποιες δηλώσεις και όχι ολόκληρη συνάρτηση. Εδώ δεν χρειάζεται να αρχικοποιήσουμε το περιβάλλον[1] και μπορούμε απευθείας να χρονομετρήσουμε μετρήσουμε την απόδοση του προγράμματος μας.

```
def measure_time():
    from timeit import Timer

    s = """\
    x = 5
    if x > 5:
        p = 4
    else:
        p = 0
    """
    t = Timer(s)
    print(t.timeit())
```

Αξίζει να σημειωθεί ότι η συνάρτηση timeit() τρέχει τον κώδικα στον οποίο θα πάρει μετρήσεις $1.000.000$ φορές, επομένως, για να έχουμε τα σωστά αποτελέσματα πρέπει τον χρόνο που μας βγάζει ως έξοδο σε δευτερόλεπτα να τον διαιρούμε με τον αριθμό των φορών που εκτελείται. Αν δεν χρειάζεται να εκτελέσουμε τόσες πολλές φορές τον κώδικα μας, μπορούμε να καθορίσουμε

[1] Εκτός και αν αυτές οι δηλώσεις χρησιμοποιούν αντικείμενα που έχουμε δημιουργήσει εμείς.

επακριβώς τον αριθμό των φορών εκτέλεσης θέτοντας το όρισμα number στην timeit().

```
def measure_time():
    from timeit import Timer

    s = """\
    x = 5
    if x > 5:
        p = 4
    else:
        p = 0
    """
    t = Timer(s)
    print(t.timeit(number=100000))
```

16.2 Μέτρηση Χρόνου Εκτέλεσης Συνάρτησης

Για να μετρήσουμε τον χρόνο εκτέλεσης, καταρχάς πρέπει να εισάγουμε το άρθρωμα timeit. Ο πιο απλός τρόπος για να μετρήσουμε τον χρόνο εκτέλεσης μια συνάρτησης είναι να χρησιμοποιήσουμε ένα αντικείμενο Timer. Σε αυτό το αντικείμενο μας ενδιαφέρουν κυρίως δυο ορίσματα. Το πρώτο είναι οι δηλώσεις (statements)[2] που θέλουμε να εκτελέσουμε ενώ το δεύτερο αναφέρεται στην αρχικοποίηση του περιβάλλοντος στο οποίο θα γίνει η μέτρηση.

```
def measure_time():
    from timeit import Timer
    t = Timer("fast_function()",\
      "from __main__ import fast_function")
    sf = 'Execution time (for each call): {} seconds'
    print(sf.format(t.timeit()/ 10**6))

def fast_function():
```

[2] Αν είναι παραπάνω από μια τις χωρίζουμε με ερωτηματικό (;)

```
    """ A fast function or at least we hope so """

    # the 2 first fibonacci numbers are always known
    a, b = 0, 1

    # find the 20 first fibonacci numbers and print them
    for i in range(1, 20):
        a, b = b, a + b

def main():
    measure_time()

if __name__ == '__main__':
    main()
```

Για να μπορέσουμε να εκτελέσουμε την συνάρτηση fast_function(), πρέπει να την εισάγουμε από το περιβάλλον στο οποίο ανήκει. Για αυτό και το δεύτερο όρισμα για την αρχικοποίηση του περιβάλλοντος είναι απαραίτητο.

Προκειμένου να έχουμε αξιόπιστα αποτελέσματα, από προεπιλογή η συνάρτηση "fast_function" που βρίσκεται μέσα στον δημιουργό Timer θα εκτελεστεί 10^6 φορές, ώστε να πάρουμε πολλές μετρήσεις και να μην έχουμε λανθασμένα αποτελέσματα. Αν θέλουμε να καθορίσουμε κάτι διαφορετικό, θα πρέπει ρητά να το προσδιορίσουμε μετά, κατά τη κλήση t.timeit() όπου θα πρέπει ως όρισμα να εισάγουμε τον αριθμό των φορών που θέλουμε να επαναληφθεί η εκτέλεση της "fast_function". Για παράδειγμα, παρακάτω βλέπουμε πως θα μπορούσαμε να εκτελέσουμε τον κώδικα μόνο 1000 φορές.

```
t = Timer("fast_function()",\
  "from __main__ import fast_function")
t.timeit(1000)
```

Για αυτό τον λόγο λοιπόν, και στο αρχικό παράδειγμα διαιρούμε τον χρόνο που μας επιστρέφεται με το 10^6, αφού τόσες είναι οι φορές που εκτελείται ο κώδικας.

16.3 Ενεργοποίηση garbage collector

Όπως αναφέραμε και στην αρχή, για ακριβέστερες μετρήσεις, κατά την εκτέλεση της μεθόδου timeit() απενεργοποιείται ο garbage collector. Αν νομίζουμε πως αυτός είναι κρίσιμος για την ακριβέστερη μέτρηση του χρόνου της εφαρμογής μας μπορούμε να τον ενεργοποιήσουμε ως ακολούθως, χρησιμοποιώντας στο όρισμα που αφορά την αρχικοποίηση του περιβάλλοντος εκτέλεσης το gc.enable().

```
def measure_time():
    from timeit import Timer

    s = """\
    x = 5
    if x > 5:
        p = 4
    else:
        p = 0
    """
    t = Timer(s, 'gc.enable()')
    print(t.timeit(number=100000))
```

16.4 Εκτέλεση Συνάρτησης με Ορίσματα

Ένα πιο δύσκολο παράδειγμα αφορά την χρονομέτρηση συνάρτησης που δέχεται ορίσματα. Για να καταλάβουμε πως γίνεται αυτό, ας σκεφθούμε πρώτα πως χρονομετράτε η συνάρτηση. Δημιουργείται ένα αντικείμενο Timer() το οποίο δημιουργεί ένα απομονωμένο περιβάλλον. Για να το κάνει αυτό καλεί τις συναρτήσεις που υπάρχουν στο όρισμα για την αρχικοποίηση αυτού του περιβάλλοντος[3]. Επομένως για να εκτελέσουμε μια συνάρτηση που δέχεται ορίσματα, πρέπει στο περιβάλλον που δημιουργείται να υπάρχουν αυτά τα ορίσματα. Για αυτό τον λόγο καλούμε μια άλλη συνάρτηση πρώτα η οποία επιστρέφει τις τιμές των ορισμάτων που θα χρησιμοποιήσουμε

[3]Αν δεν υπάρχει καμία, υπονοείται η pass

στο περιβάλλον, και ύστερα τις περνάμε στην συνάρτηση της οποίας θέλουμε να μετρήσουμε την απόδοση.

```
def measure_time():
    from timeit import Timer

    t = Timer('fast_function(a, b)',\
    'from __main__ import fast_function,\
    setup_function; a,b = setup_function()',)
    print(t.timeit())

def fast_function(a, b):
    """ A fast function or at least we hope so """

    if a > b:
        l = []
    else:
        d = {}

def setup_function():
    """ A function which must be called first """

    a = 5
    #b = 4
    b = int(input("Insert b: "))

    return a, b

def main():
    measure_time()

if __name__ == '__main__':
    main()
```

Όπως βλέπουμε και στο παράδειγμα, μπορούμε κάλλιστα να έχουμε προκαθορισμένες τιμές των ορισμάτων, ή να ζητάμε από το χρήστη τις κατάλληλες τιμές.

16.5 Διεπαφή γραμμής εντολών

Διαφωνείτε με κάποιον φίλο σας τι είναι πιο γρήγορο όσον αφορά ένα πολύ μικρό κομμάτι κώδικα; Υπάρχει λύση! Αν θέλουμε να δοκιμάσουμε πόσο χρόνο απαιτείται για την εκτέλεση ενός συνόλου δηλώσεων, ένας πολύ γρήγορος τρόπος, αν αυτό είναι κάτι απλό, είναι να το κάνουμε μέσα από τη διεπαφή της γραμμής εντολών. Υπάρχει αντιστοιχία με αυτά που είδαμε παραπάνω, αλλά το κύριο που χρειάζεται να συγκρατήσει κανείς (ώστε να δούμε ποιος θα κερδίσει στη διαφωνία των φίλων) είναι η μορφή που θα έχει η δήλωση μας μέσω γραμμής εντολών και η οποία ακολουθεί το μοτίβο:

```
python -mtimeit -s"SETUP_COMMANDS" "COMMAND_TO_BE_TIMED"
```

Ένα απλό παράδειγμα όπου μετράμε πόσο χρόνο παίρνει η μετατροπή κάποιων αριθμών σε οκταδικό είναι το ακόλουθο, όπου μάλιστα έχουμε ενεργοποιήσει και τον σωρευτή απορριμάτων. (Garbage Collector)

```
python -mtimeit -s'for i in range(10): oct(i)' 'gc.enable()'
```

16.6 Εξαγωγή μετρικών (Profiling)

Όταν προσπαθούμε να βελτιώσουμε την ταχύτητα μιας εφαρμογής, σημαντικό ρόλο παίζει να βρούμε τα σημεία που την καθυστερούν ιδιαίτερα, καθώς μια βελτίωση σε αυτά θα επέφερε μεγαλύτερο κέρδος. Μάλιστα, στα οικονομικά υπάρχει ο όρος 'Νόμος των φθίνουσων αποδόσεων' (Law of diminishing returns). Ο νόμος αυτός περιγράφει πως η βελτίωση ενός συγκεκριμένου παράγοντα θα πάψει να αποφέρει κάποιο κέρδος από ένα σημείο και μετά, τουλάχιστον σε σχέση με τον κόπο που καταβάλλεται. Ο λόγος είναι ότι μπορεί να υπάρχουν κάποια άλλα σημεία που είναι πιο σημαντικά και θα πρέπει να επικεντρώσουμε τις προσπάθειες μας σε αυτά.

Στον προγραμματισμό, η διαδικασία προσδιορισμού αυτών των σημαντικών σημείων ονομάζεται profiling και η Python μας βοηθάει μέσω του cProfile. Μπορούμε να το τρέξουμε από την γραμμή εντολών δίνοντας του ως είσοδο το πρόγραμμα που θέλουμε να εξετάσουμε. Αυτό θα τυπώσει ως αποτέλεσμα πόσο χρόνο παίρνει η κάθε συνάρτηση που καλείται. Ο τρόπος που μπορούμε να χρησιμοποιήσουμε αυτή τη λειτουργία φαίνεται παρακάτω:

```
python -m cProfile filename.py
```

Ως αποτέλεσμα, βλέπουμε το συνολικό αριθμό κλήσεων μιας συνάρτησης, συνολικό χρόνο εκτέλεσης, χρόνο εκτέλεσης ανά κλήση και άλλα.