

Προγραμματισμός Η/Υ

Περιεχόμενα

- 1. Εισαγωγή στον προγραμματισμό
- 2. Τιμές, τύποι και μεταβλητές. Συμβολοσειρές
- 3. Λίστες, Εκφράσεις, τελεστές
- 4. Πλειάδες και λεξικά
- 5. Έλεγχος ροής εκτέλεσης
- 7. Ανάγνωση & εγγραφή αρχείων, μετονομασία, αναζήτηση, αντιγραφή μετακίνηση αρχείων και καταλογών

Τμήμα: Μηχανικών Χωροταξίας, Πολεοδομίας και Περιφερειακής Ανάπτυξης

Τίτλος Επιστημονικού Πεδίου: Πληροφορική και Ανάλυση Δεδομένων

Κωδικός Μαθήματος: ΜΕ0200

Τίτλος Μαθήματος: Προγραμματισμός Η/Υ

Κατηγορία Μαθήματος: Επιλογής

Εξάμηνο: Εαρινό

Περίγραμμα του μαθήματος

Το μάθημα εισάγει τους φοιτητές στις βασικές έννοιες και αρχές του προγραμματισμού και είναι προσαρμοσμένο στις ανάγκες και στο υπόβαθρο των χωροτακτών. Στόχος του μαθήματος είναι να αποκτήσουν οι φοιτητές τις αναγκαίες γνώσεις για την επεξεργασία και ανάλυση δεδομένων, την αυτοματοποίηση διαδικασιών και την σύνταξη σεναρίων (scripts) για την αναπαραγωγισιμότητα της ερευνάς τους. Πέρα από τις βασικές αρχές προγραμματισμού η διδασκαλία επεκτείνεται σε εξειδικευμένες θεματικές ενότητες που αφορούν τα γεωχωρικά δεδομένα και την ανάλυσή τους μέσω προγραμματισμού. Η διδασκαλία θα στηριχθεί στην [Python](#), μια σύγχρονη, ευρέως διαδεδομένη και υψηλού επιπέδου γλώσσα προγραμματισμού. Επιπλέον, όπου κριθεί αναγκαίο, θα επιδειχτούν συμπληρωματικά διαδικασίες με την γλώσσα προγραμματισμού [R](#).

Το πρόγραμμα των διαλέξεων καθώς και η προτεινόμενη βιβλιογραφία παρατίθενται στην συνέχεια.

1. Εισαγωγή στον προγραμματισμό

Ενότητες του μαθήματος

Το μάθημα χωρίζεται στις ακόλουθες ενότητες:

1. Εισαγωγή στον προγραμματισμό

Κατά την διάρκεια της διάλεξης διευκρινίζεται ο σκοπός του μαθήματος και περιγράφονται συνοπτικά οι ενότητες που θα διδαχθούν οι φοιτητές κατά την διάρκεια του εξαμήνου. Διατυπώνονται συγκεκριμένοι ορισμοί που αφορούν τον προγραμματισμό Η/Υ και αναπτύσσονται έννοιες για την επιστήμη των υπολογιστών. Στην συνέχεια εγκαθίσταται στους υπολογιστές των φοιτητών η γλώσσα προγραμματισμού Python μαζί με το απαραίτητο λογισμικό για την συγγραφή και αποσφαλμάτωση του κώδικα. Ακολουθεί εξοικείωση με το περιβάλλον εργασίας.

2. Τιμές, τύποι και μεταβλητές

Περιγραφή της έννοιας των μεταβλητών, των σταθερών, τύποι δεδομένων, εκχώρηση τιμών στις μεταβλητές, κανόνες ονοματοδοσίας των μεταβλητών.

3. Εκφράσεις, τελεστές

Ορισμός εκφράσεων, τι είναι τελεστές, ποια είναι η προτεραιότητα των τελεστών, πως εισάγουμε σχόλια στον κώδικα και γιατί είναι σημαντική πρακτική.

4. Έλεγχος ροής εκτέλεσης

Η λογική Boolean, Εκτέλεση υπό συνθήκη, αλυσιδωτές και εμφωλευμένες συνθήκες, βρόχος και οι εντολές επανάληψης for και while.

5. Συναρτήσεις

Ορισμός και κλήση συνάρτησης, παράμετροι συναρτήσεων, εμβέλεια μεταβλητών, αναδρομή.

6. Συμβολοσειρές/Δομές Δεδομένων

Προσπέλαση συμβολοσειρών, χαρακτήρες διαφυγής, υποσύνολα συμβολοσειράς, συγκρίσεις και ιδιότητες, μέθοδοι συμβολοσειρών. Λίστες, Πλειάδες, Λεξικά.

7. Ανάγνωση & εγγραφή αρχείων, φάκελοι

Ανάγνωση και εγγραφή σε αρχείο, σειριοποίηση (serialization) αντικειμένου, διαχείριση φακέλων και αρχείων.

8. Πίνακες και διαγράμματα

Ανάγνωση αρχείων csv ή excel, pandas dataframes

9. Πίνακες και διαγράμματα

Πίνακες στην βιβλιοθήκη numpy, διαγράμματα με την βιβλιοθήκη seaborn.

10. Γεωπεξεργασία διανυσματικών δεδομένων

Ανάγνωση και εγγραφή διανυσματικών δεδομένων, μετα-δεδομένα, φιλτράρισμα, αλλαγή προβολικού συστήματος.

11. Ανάλυση διανυσματικών δεδομένων

Χωρικές σχέσεις, στατιστικά ομαδοποιήσεων, οπτικοποίηση διανυσματικών δεδομένων.

12. Γεωπεξεργασία ψηφιδωτών δεδομένων

Ανάγνωση και εγγραφή διανυσματικών δεδομένων, μετα-δεδομένα, ορισμός μάσκας/αποκοπή περιοχής, αλλαγή τιμών, επαναταξινόμηση, αλλαγή προβολικού συστήματος.

13. Ανάλυση ψηφιδωτών δεδομένων

Άλγεβρα ψηφιδωτών αρχείων, στατιστικά ζωνών, ιστόγραμμα συχνοτήτων.

Ορισμοί

Definition 1

«Αλγόριθμος» ονομάζουμε κάθε πεπερασμένη και αυστηρά καθορισμένη σειρά βημάτων (οδηγιών) για την επίλυση ενός προβλήματος.

[Αγγελιδάκης, 2015]

Ένας αλγόριθμος είναι μια αυστηρά καθορισμένη διαδικασία που λαμβάνει μια τιμή ή ένα σύνολο τιμών εισόδου και αποδίδει μια ή περισσότερες τιμές εξόδου. Είναι κατά συνέπεια μια ακολουθία υπολογιστικών βημάτων που μετατρέπει την είσοδο δεδομένων σε έξοδο αποτελεσμάτων [Cormen, 2009].

Για παράδειγμα η αύξουσα (ή φθίνουσα) ταξινόμηση μιας λίστας αριθμών είναι ένα χαρακτηριστικό παράδειγμα αλγορίθμου. Οπότε με τιμές εισόδου {31, 41, 59, 26, 41, 58}, ο αλγόριθμος ταξινόμησης επιστρέφει ως τιμές εξόδου {26, 31, 41, 41, 58, 59}.

Definition 2

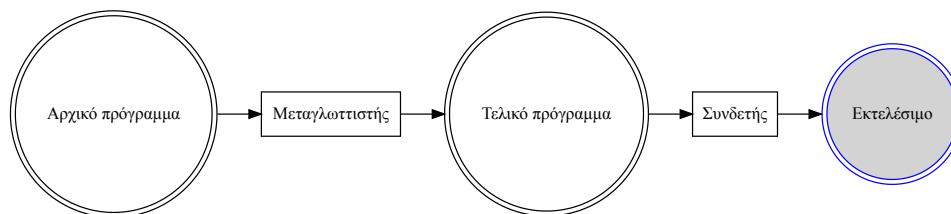
Ως «Πρόγραμμα» ορίζεται ένας αλγόριθμος γραμμένο σε γλώσσα κατανοητή για τον υπολογιστή και περιέχει εντολές (οδηγίες) που κατευθύνουν με κάθε λεπτομέρεια τον υπολογιστή, για να εκτελέσει μια συγκεκριμένη εργασία και να επιλύσει ένα πρόβλημα [Αγγελιδάκης, 2015]. Ένα Πρόγραμμα αναγνώσιμο από τον άνθρωπο ονομάζεται «πηγαίος κώδικας».

Definition 3

«Προγραμματισμός» ονομάζεται η διαδικασία συγγραφής προγραμμάτων και περιλαμβάνει τη διατύπωση των κατάλληλων εντολών προς τον υπολογιστή με τη χρήση τεχνητών γλωσσών, των γλωσσών προγραμματισμού [Αγγελιδάκης, 2015].

Ο προγραμματισμός είναι μια διαδικασία που απαιτεί μια σειρά εργαλείων και διαδικασιών τα οποία συνήθως ενσωματώνονται όλα μαζί σε ένα εντοπιζόμενο περιβάλλον που αποκαλείται «ολοκληρωμένο περιβάλλον ανάπτυξης εφαρμογών» (Integrated Development Environment, IDE).

Η διαδικασία μετατροπής του πηγαίου κώδικα σε εκτελέσιμο αρχείο περιγράφεται στο παρακάτω διάγραμμα:



Εικ. 1 Η ροή εκτέλεσης του κώδικα σε εκτελέσιμο.

Τα εργαλεία προγραμματισμού τα οποία κάνουν την μεταγλωττιστή του πηγαίου προγράμματος σε εκτελέσιμο πρόγραμμα είναι τα εξής:

- ο επεξεργαστής κειμένου με την βοήθεια οποίου συντάσσεται ο πηγαίος κώδικας του προγράμματος.
- ο μεταγλωττιστής ή διερμηνευτής οι οποίοι χρησιμοποιούνται για την μετατροπή του πηγαίου κώδικα σε γλώσσα μηχανής η οποία είναι απαραίτητη για την αναγνώριση και εκτέλεση των εντολών από τον Η/Υ. Τα παραγόμενα πρόγραμμα από την μεταγλώττιση ονομάζεται αντικείμενο πρόγραμμα (object). Ο διερμηνευτής διαβάζει διαδοχικά τις εντολές και για κάθε εντολή που διαβάζει, εκτελεί αμέσως μια ισοδύναμη ακολουθία εντολών μηχανής. Από την άλλη, ο μεταγλωττιστής δέχεται στην είσοδο ένα πρόγραμμα γραμμένο σε γλώσσα υψηλού επιπέδου (πηγαίος κώδικας) και παράγει ισοδύναμο πρόγραμμα σε γλώσσα μηχανής (αντικείμενο).
- ο συνδετής - φορτωτής (linker - loader) ο οποίος συνδέει το αντικείμενο πρόγραμμα με άλλα τμήματα του προγράμματος ή απαραίτητες βιβλιοθήκες που διατίθενται από την γλώσσα προγραμματισμού. Το τελικό πρόγραμμα που προκύπτει από την μεταγλώττιση και την σύνδεση των τμημάτων του προγράμματος είναι το εκτελέσιμο πρόγραμμα (executable) το οποίο μπορεί να διαβάσει και να εκτελέσει ο υπολογιστής.
- τα εργαλεία αποσφαλμάτωσης με την βοήθεια των οποίων δοκιμάζεται η εκτέλεση και η ορθότητα του πηγαίου κώδικα και εντοπίζονται λάθη σε αυτόν.

Τα λάθη στον κώδικα συνοψίζονται σε τρεις βασικές κατηγορίες:

1. σφάλμα μεταγλώττισης τα οποία προκύπτουν κατά την λανθασμένη συγγραφή του πηγαίου κώδικα. Ο μεταγλωττιστής δεν επιτρέπει την μετάφραση του πηγαίου κώδικα σε γλώσσα μηχανής αν προηγουμένως δεν έχει διορθωθεί το συντακτικό λάθος. Συντακτικά λάθη συμβαίνουν συνήθως όταν δεν ακολουθούνται οι κανόνες σύνταξης μια γλώσσας (π.χ. μια παρένθεση που δεν έχει κλείσει, ένα ξεχασμένο εισαγωγικό ή κόμμα κτλ.).

Το παρακάτω είναι ένα παράδειγμα συντακτικού σφάλματος και το μήνυμα που επιστρέφει ο μεταγλωττιστής της Python. Η αιτία του σφάλματος είναι η ξεχασμένη παρένθεση στην συνάρτηση (function) `print`

```
print("an example"
```

```
Input In [2]
  print("an example"
    ^
SyntaxError: '(' was never closed
```

1. *σφάλμα εκτέλεσης* (run-time errors) τα οποία συμβαίνουν κατά την εκτέλεση του προγράμματος παρότι δεν υπάρχουν σφάλματα σύνταξης. Χαρακτηριστικά παραδείγματα τέτοιων λαθών είναι η διαίρεση με το μηδέν, η πρόσβαση σε ένα στοιχείο μιας λίστας εκτός του εύρους της, η ανάγνωση ενός αρχείου το οποίο δεν υπάρχει, ή η πρόσβαση σε ένα ανύπαρκτο object. Τα σφάλματα εκτέλεσης έχουν επικρατήσει να αναφέρονται και ως «bugs» ^[1]. Παρακάτω δίνεται ένα σφάλμα που προκύπτει από την διαίρεση ενός ακέραιου με το μηδέν.

```
1/0
```

```
-----
ZeroDivisionError                                Traceback (most recent call last)
Input In [3], in <cell line: 1>()
----> 1 1/0

ZeroDivisionError: division by zero
```

1. *σφάλμα λογικής*, κατά το οποίο το πρόγραμμα εκτελείται κανονικά χωρίς σφάλματα αλλά δεν συμπεριφέρεται όπως έχει σχεδιαστεί να συμπεριφέρεται. Αυτά τα σφάλματα δεν σταματούν την εκτέλεση του προγράμματος αλλά το αποτέλεσμα της εκτέλεσης δεν είναι το αναμενόμενο.

```
x = 6
y = 4

z = x+y/2
print('Ο μέσος όρος των δύο αριθμών είναι:',z)
```

```
Ο μέσος όρος των δύο αριθμών είναι: 8.0
```

Το παραπάνω είναι σφάλμα λογικής γιατί έπρεπε να γραφτεί ως εξής (δώστε προσοχή στις παρενθέσεις που δίνουν προτεραιότητα στις πράξεις):

```
x = 6
y = 4

z = (x+y)/2
print('Ο μέσος όρος των δύο αριθμών είναι:',z)
```

```
Ο μέσος όρος των δύο αριθμών είναι: 5.0
```

Όλες οι παραπάνω μορφές σφαλμάτων εντοπίζονται μέσω της *αποσφαλμάτωσης*, της συστηματικής δηλαδή διαδικασίας εντοπισμού και επιδιόρθωσης σφαλμάτων. Η αποσφαλμάτωση συνοψίζεται στα εξής βήματα:

- Επανάληψη του προβλήματος
- Απομόνωση του σημείου που εμφανίζεται το σφάλμα
- Αναγνώριση της αιτίας που το προκαλεί
- Διόρθωση του σφάλματος
- Επιβεβαίωση της διόρθωσης





















Οι εντολές των προγραμμάτων γράφονται από τους προγραμματιστές σε τεχνητές γλώσσες που ονομάζονται «*γλώσσες προγραμματισμού*». Μια γλώσσα προγραμματισμού θα πρέπει να έχει αυστηρά ορισμένη σύνταξη και σημασιολογία. Η σύνταξη καθορίζει αν μια σειρά από σύμβολα αποτελούν «νόμιμες» εντολές ενός προγράμματος γραμμένου σε μια συγκεκριμένη γλώσσα προγραμματισμού και η σημασιολογία καθορίζει τη σημασία του προγράμματος, δηλαδή τις υπολογιστικές διαδικασίες που υλοποιεί. [\[Αγγελιδάκης, 2015\]](#).

Η γλώσσα προγραμματισμού Python

Η Python είναι μια ευρέως διαδομένη, αντικειμενοστραφής, υψηλού επιπέδου γλώσσα προγραμματισμού γενικής χρήσης. Η Python είναι μια γλώσσα που εκτελεί τις εντολές στον διερμηνέα, που όπως αναφέρθηκε, διαβάζει τον πηγαίο κώδικα γραμμή προς γραμμή και το μετατρέπει σε γλώσσα μηχανής. Αυτός ο τρόπος λειτουργίας της Python την καθιστά πιο αργή σε σύγκριση με άλλες γλώσσες μεταγλωττιστού όπως η C. Η Python είναι διαδραστική υπό την έννοια ότι ο χρήστης εκτελεί εντολές μέσω της γραμμή εντολών της Python, εκτελείται άμεσα και λαμβάνει το αποτέλεσμα εξόδου.

Δημιουργήθηκε από τον Guido van Rossum και πρωτοκυκλοφόρησε στις 20 Φεβρουαρίου του 1991. Το όνομά της, αν και παρεπένπει, δεν έχει σχέση με το φίδι Πύθωνα αλλά προέρχεται από την γνωστή κωμική σειρά του BBC, Monty Python's Flying Circus. Αν και αρχικά αναπτύχθηκε σαν μεμονωμένη ατομική προσπάθεια στην συνέχεια υποστηρίχθηκε από μια παγκόσμια κοινότητα προγραμματιστών και χρηστών. Στις 6 Μαρτίου 2001 ιδρύθηκε το αμερικάνικο μη κερδοσκοπικό ίδρυμα *Python Software Foundation (PSF)*, το οποίο στόχο έχει την διάδοση και υποστήριξη της Python μέσω της διοργάνωσης συνεδρίων, την ανάπτυξη κοινοτήτων χρηστών, την υποστήριξη προσπαθειών μέσω υποτροφιών και την διασφάλιση οικονομικών πόρων για την ανάπτυξη της γλώσσας. Το ίδρυμα κατέχει τα πνευματικά δικαιώματα της γλώσσας και διασφαλίζει ότι αυτή θα διατίθεται με όρους ελεύθερου λογισμικού προς το ευρύτερο κοινό.

Οι βασικοί στόχοι που έθεσε ο δημιουργός κατά την ανάπτυξή της είναι να είναι εύκολη και κατανοητή με ισχυρές δυνατότητες εφάμιλλες των ανταγωνιστικών γλωσσών. Ταυτόχρονα έθεσε το όρο να είναι ανοιχτού κώδικα (open source) για να μπορεί εύκολα να αναπτύσσεται απο τους ενδιαφερόμενους προγραμματιστες και να έχει πρακτική αξία σε καθημερινές εργασίες ρουτίνας. Την τρέχουσα περίοδο (03/2022) κατατάσσεται ως η κορυφαία γλώσσα προγραμματισμού σύμφωνα με την κοινότητα προγραμματιστών [TIOBE](#) αλλά και τον δείκτη [Popularity of Programming Language Index \(PYPL\)](#).

Mar 2022	Mar 2021	Change	Programming Language	Ratings	Change
1	3	▲	 Python	14.26%	+3.95%
2	1	▼	 C	13.06%	-2.27%
3	2	▼	 Java	11.19%	+0.74%
4	4		 C++	8.66%	+2.14%
5	5		 C#	5.92%	+0.95%
6	6		 Visual Basic	5.77%	+0.91%
7	7		 JavaScript	2.09%	-0.03%
8	8		 PHP	1.92%	-0.15%
9	9		 Assembly language	1.90%	-0.07%
10	10		 SQL	1.85%	-0.02%
11	13	▲	 R	1.37%	+0.12%
12	14	▲	 Delphi/Object Pascal	1.12%	-0.07%
13	11	▼	 Go	0.98%	-0.33%
14	19	▲	 Swift	0.90%	-0.05%
15	18	▲	 MATLAB	0.80%	-0.23%
16	16		 Ruby	0.66%	-0.52%
17	12	▼	 Classic Visual Basic	0.60%	-0.66%
18	20	▲	 Objective-C	0.59%	-0.31%
19	17	▼	 Perl	0.57%	-0.58%
20	38	▲	 Lua	0.56%	+0.23%

Εικ. 2 Η κατάταξη σύμφωνα με την κοινότητα TIOBE (Μάρτιος 2022)

Η Python πλέον είναι μια ώριμη γλώσσα προγραμματισμού με εφαρμογές στην ανάπτυξη διαδικτυακών εφαρμογών και υπηρεσιών, την εκπαίδευση, την ανάλυση δεδομένων, την τηλεπισκόπηση και τα ΣΓΠ, την δημιουργία γραφικών, την διαχείριση συστημάτων, τα παιχνίδια, το εμπόριο και την επιχειρηματικότητα, τους μικροελεγκτές και το Internet of Things (IOT).

Η φιλοσοφία της Python ως προς την μεθοδολογία ανάπτυξης και προγραμματισμού συνοψίζεται σε 20 αρχές, οι οποίες εκτυπώνονται μέσω της γλώσσας με την παρακάτω εντολή:

```
import this
```

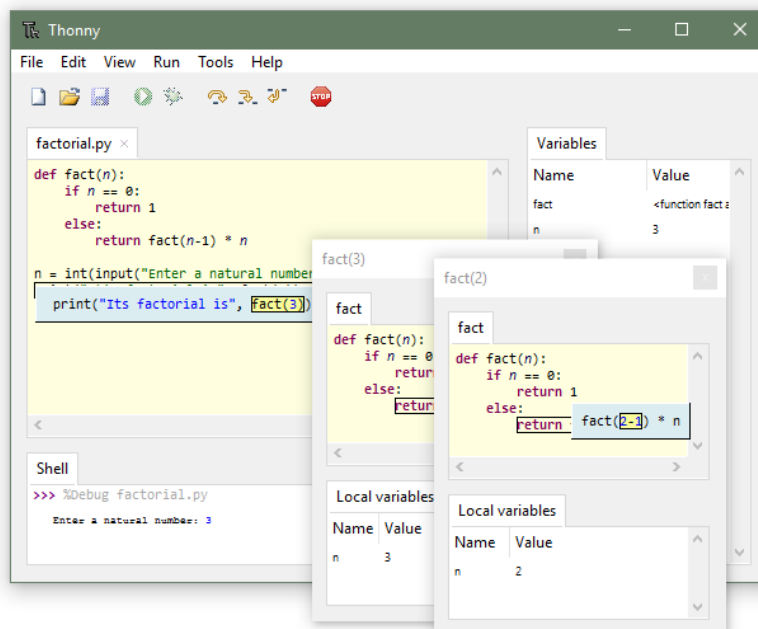
The Zen of Python, by Tim Peters

```
Beautiful is better than ugly.  
Explicit is better than implicit.  
Simple is better than complex.  
Complex is better than complicated.  
Flat is better than nested.  
Sparse is better than dense.  
Readability counts.  
Special cases aren't special enough to break the rules.  
Although practicality beats purity.  
Errors should never pass silently.  
Unless explicitly silenced.  
In the face of ambiguity, refuse the temptation to guess.  
There should be one-- and preferably only one --obvious way to do it.  
Although that way may not be obvious at first unless you're Dutch.  
Now is better than never.  
Although never is often better than *right* now.  
If the implementation is hard to explain, it's a bad idea.  
If the implementation is easy to explain, it may be a good idea.  
Namespaces are one honking great idea -- let's do more of those!
```

Το ολοκληρωμένο περιβάλλον ανάπτυξης thonny

Η συγγραφή κώδικα θα γίνει στο ολοκληρωμένο περιβάλλον ανάπτυξης (Integrated Development Environment, IDE) [thonny](#). Δεν απαιτείται η προεγκατάσταση της Python καθώς το thonny έρχεται με ενσωματωμένη την γλώσσα προγραμματισμού Python 3.7 και διατίθεται για Windows, Mac και Linux. Το thonny αποτελεί εκπαιδευτικό περιβάλλον για την συγγραφή και αποσφαλμάτωση κώδικα Python. Για τον λόγο αυτό διαθέτει πολύ συγκεκριμένες αλλά ζωτικές λειτουργίες και δεν είναι επιφορτισμένο με δυνατότητες που απαιτούνται από προχωρημένους προγραμματιστές. Το Γραφικό Περιβάλλον Χρήστη (GUI, Graphical User Interface) είναι λιτό ώστε να μην αποπροσανατολίζει τον αρχάριο χρήστη. Το thonny παρέχει βοηθητικές λειτουργίες για τον χρήστη όπως είναι η σήμανση συντακτικών λαθών, η αυτόματη συμπλήρωση κώδικα και η ευκολία στην επέκταση των λειτουργιών της Python με την εγκατάσταση συμπληρωματικών πακέτων.

Ενναλλακτικά, στους χρήστες παρέχεται online περιβάλλον ανάπτυξης που βασίζεται στο [Jupyter Lab](#). Η χρήση του δεν απαιτεί την εγκατάσταση λογισμικού παρά μόνον έναν απλό φυλλομετρητή (προτείνεται Chrome, Safari ή Firefox). Το online περιβάλλον Jupyter είναι προσβάσιμο από εδώ: <https://kokkytos.github.io/programming>



Εικ. 3 Το περιβάλλον εργασία thonny.

Εκτέλεση εντολών στο περιβάλλον thonny

Στην παρακάτω ενότητα παρουσιάζονται μερικά εισαγωγικά παραδείγματα από εντολές της Python. Δεν θα επικεντρωθούμε σε λεπτομέρειες ούτε θα αναλύσουμε τις εντολές που διατυπώνονται στα αρχεία. Παρουσιάζονται σαν μια μορφή συνοπτικής επίδειξης των δυνατοτήτων που διαθέτει η γλώσσα και θα περιγράψουμε σε επόμενα μαθήματα.

Δοκιμάστε να τρέξετε την παρακάτω εντολή στην γραμμή εντολών της Python στο thonny:

```
25+30
```

```
55
```

Τι παρατηρείτε; Η Python λειτούργησε σαν μια απλή αριθμομηχανή.

Στην συνέχεια δοκιμάστε να τρέξετε την παρακάτω εντολή γραμμή προς γραμμή:

```
number1 = 25
number2 = 30
number3 = number1+number2
number3
```

```
55
```

Το αποτέλεσμα είναι το ίδιο με το προηγούμενο.

Δώστε την παρακάτω εντολή. Αντικαταστήστε την συμβολοσειρά *AnyName* με το ονομά σας.

```
name="AnyName"

for i in range(10):
    print("Εκτύπωση", i, ":", name)
```

```
Εκτύπωση 0 : AnyName
Εκτύπωση 1 : AnyName
Εκτύπωση 2 : AnyName
Εκτύπωση 3 : AnyName
Εκτύπωση 4 : AnyName
Εκτύπωση 5 : AnyName
Εκτύπωση 6 : AnyName
Εκτύπωση 7 : AnyName
Εκτύπωση 8 : AnyName
Εκτύπωση 9 : AnyName
```

Όπως βλέπετε η Python επανέλαβε την εκτύπωση του ονόματός σας 10 φορές. Από που ξεκινά όμως η αρίθμηση της πρώτης εκτύπωσης;

Στο επόμενο παράδειγμα η Python θα σας ενημερώσει αν τρέχετε γρήγορα ή αργά ή αν είστε ακίνητος:

```
speed=70

if speed>50:
    if speed>=100:
        print("Τρέχεις πολύ γρήγορα")
    else:
        print("Τρέχεις γρήγορα")
else:
    if speed==0:
        print("Είσαι ακίνητος".upper())
    if speed>0:
        print("Τρέχεις αργά")
```

```
Τρέχεις γρήγορα
```

Έστω ότι κινείσθε σε αυτοκινητόδρομο με 120 km/h. Αν ορίσετε την ταχύτητα (speed) στον κώδικα, τι θα σας απαντήσει η Python; Αν κινείστε με μηδενική ταχύτητα (speed=0) τι μήνυμα θα λάβετε; Υπάρχουν περιπτώσεις που η Python, και δικαιολογημένα, αγνοεί να απαντήσει με μήνυμα στην ταχύτητα που ορίζεται. Μπορείτε να εντοπίσετε σε ποιές περιπτώσεις;

[1] Η πρώτη περίπτωση *bug* σε υπολογιστή καταγράφεται το 1947 από τον Grace Murray Hopper και πρόκειται για την κυριολεκτική έννοια του όρου. Στο ημερολόγιό του καταγράφει προβλήματα στην λειτουργία του υπολογιστή του Harvard, Mark II, από την ύπαρξη ενός εντόμου στο εσωτερικό του κύκλωμα.

2. Τιμές, τύποι και μεταβλητές. Συμβολοσειρές

Σταθερές (Constants)

Η Python δεν διαθέτει προκαθορισμένες *σταθερές* όπως άλλες γλώσσες προγραμματισμού. Όμως κατά σύμβαση και όχι κατά κανόνα έχει συμφωνηθεί οι *σταθερές* να ονοματίζονται με κεφαλαίους χαρακτήρες. Η αδυναμία της Python στην περίπτωση της δήλωσης *σταθερών* είναι ότι επιτρέπεται η αλλαγή των τιμών τους Παρακάτω παρατίθεται ένα παράδειγμα δήλωσης *σταθερών*.

```
RATIO_FEET_TO_METERS = 3.281
RATIO_LB_TO_KG = 2.205
PI = 3.14
```

Κυριολεκτικές σταθερές (literal constants)

Η κυριολεκτική *σταθερά* ή τιμή είναι ένας αριθμός, ή χαρακτήρας ή μιά συμβολοσειρά. Για παράδειγμα τα παρακάτω αποτελούν τιμές: 3.25 (στην python η υποδιαστολή ορίζεται με . και όχι .), «ένα τυχαίο κείμενο», 5.25e-1. Αυτές οι τιμές δεν μεταβάλλονται κατά τη διάρκεια εκτέλεσης του προγράμματος γι' αυτό και λέγονται σταθερές. Μπορούν να εκχωρηθούν σε μεταβλητές και να χρησιμοποιηθούν σαν τελεστές σε λογικές εκφράσεις ή σαν παραμέτροι σε συναρτήσεις.

Τύποι δεδομένων

Οι τιμές ανήκουν σε τρεις τύπους δεδομένων (data types) ή κλάσσεις (class):

- τους ακέραιους αριθμούς (integer) π.χ. το 15
- τους αριθμούς κινητής υποδιαστολής (floating point) π.χ. το 201.25)
- τις συμβολοσειρές (string) π.χ. το «Time is money»

Με την εντολή **type** ο διερμηνευτής μας απαντάει με τον τύπο της τιμής, όπως παρακάτω:

```
type("No news, good news.")
```

```
str
```

Η Python είναι *Dynamic typing* δηλαδή δεν ο τύπος των μεταβλητών δεν προκαθορίζεται κατά την συγγραφή αλλά κατά την εκτέλεση.

Κανόνες ονοματοδοσίας μεταβλητών

Τα ονόματα των μεταβλητών στην Python υπακούουν στους παρακάτω κανόνες:

- Το όνομα μίας μεταβλητής μπορεί να ξεκινά από ένα γράμμα ή από κάτω πάλυα.
- Το όνομα μίας μεταβλητής δεν μπορεί με αριθμό.
- Το όνομα μίας μεταβλητής μπορεί να περιέχει μόνο αλφαριθμητικούς χαρακτήρες.
- Στα ονόματα των μεταβλητών γίνεται διάκριση ανάμεσα σε πεζά και κεφαλαία (case sensitive).
- Οι δεσμευμένες λέξεις της Python (keywords) δεν μπορούν να χρησιμοποιηθούν σε ονόματα μεταβλητών.

Συμβολοσειρές (Strings)

Μια συμβολοσειρά είναι μια ακολουθία από χαρακτήρες όπως το "Το πεπρωμένον φυγείν αδύνατον.". Μπορεί να είναι σε κάθε γλώσσα που υποστηρίζεται από το πρότυπο Unicode. Οι συμβολοσειρές περικλείονται σε μονά, διπλά ή τριπλά εισαγωγικά. Με τριπλά εισαγωγικά μπορούν να ενσωματωθούν με ευκολία συμβολοσειρές σε πολλές γραμμές και πολλαπλά εισαγωγικά εντός αυτών. Ακολουθούν παραδείγματα συμβολοσειρά.

Χαρακτήρες διαφυγής,κενά, νέες γραμμές

Μπορούμε να σπάσουμε μια συμβολοσειρά κατά την συγγραφή σε νέα γραμμή με τον χαρακτήρα \ και κατά την εκτέλεση με τον χαρακτήρα \n π.χ.

```
message = 'There is no smoke \
without fire'

print(message)
```



```
There is no smoke without fire
```

```
message = 'There is no smoke \nwithout fire'  
print(message)
```

```
There is no smoke  
without fire
```

Ή να ορίσουμε κενά με το `\t`

```
message = 'There is no smoke \twithout fire'  
print(message)
```

```
There is no smoke      without fire
```

Ο χαρακτήρας `\` είναι χαρακτήρας διαφυγής που απενεργοποιεί την ειδική λειτουργία των παραπάνω ή την παράθεση εισαγωγικών μέσα σε εισαγωγικά.

```
print('There is no smoke \\n without fire')
```

```
There is no smoke \n without fire
```

```
print('Where there\'s a will, there\'s a way')
```

```
Where there's a will, there's a way
```

Ανεπεξέργαστες συμβολοσειρές (Raw Strings)

Παρόμοιο αποτέλεσμα με τα παραπάνω πετυχαίνουμε τις ανεπεξέργαστες συμβολοσειρές οι οποίες ορίζονται με ένα `r` σαν πρόθεμα

```
print(r"It was made by \n συνέχεια")
```

```
It was made by \n συνέχεια
```

Αφαίρεση κενών

Σε αρκετές περιπτώσεις οι συμβολοσειρές περιέχουν κενά είτε στην αρχή είτε στο τέλος. Για παράδειγμα οι παρακάτω συμβολοσειρές δεν είναι το ίδιες για την Python. Και επιβεβαιώνεται σε μέσω ελέγχου ισότητας.

```
departmentA='ΤΜΧΠΑ'  
departmentB = ' ΤΜΧΠΑ '  
print(departmentA == departmentB) #not equal
```

```
False
```

Για την αφαίρεση των κενών αριστερά, δεξιά ή ταυτόχρονα και στις δύο πλευρές της συμβολοσειρας χρησιμοποιούμε την μέθοδο `strip` και τις παραλλαγές της `rstrip` και `lstrip`

```
print(departmentB.rstrip())  
print(departmentB.lstrip())  
print(departmentB.strip())
```

```
ΤΜΧΠΑ  
ΤΜΧΠΑ  
ΤΜΧΠΑ
```

Συνένωση (Concatenation) συμβολοσειρών

Η απλή παράθεση συμβολοσειρών οδηγεί στην συνενωσή τους δηλ.

```
message = "Curiosity " "killed " 'the ' "'cat'"
print(message)
```

```
Curiosity killed the cat
```

Συνένωση συμβολοσειρών και μεταβλητών

Η συνένωση μεταβλητών και συμβολοσειρών γίνεται με τον τελεστή +.

```
city='Βόλος'
perifereia='Θεσσαλία'

print('Ο '+city+' είναι πόλη της Ελλάδα στην ' +perifereia)
```

```
Ο Βόλος είναι πόλη της Ελλάδα στην Θεσσαλία
```

Η μέθοδος format

Άλλη μια πιο πρακτική μέθοδος κατά την συνένωση μεταβλητών και συμβολοσειρών είναι η μέθοδος format.

```
print('Ο {0} έχει υψόμετρο {1} μέτρα'.format("Όλυμπος", 2918))
print('Ο {} έχει υψόμετρο {} μέτρα'.format("Όλυμπος", 2918))
print('Ο {name} έχει υψόμετρο {height} μέτρα'.format(name="Σμόλικας", height= 2637
))
```

```
Ο Όλυμπος έχει υψόμετρο 2918 μέτρα
Ο Όλυμπος έχει υψόμετρο 2918 μέτρα
Ο Σμόλικας έχει υψόμετρο 2637 μέτρα
```

Δεσμευμένες λέξεις (reserved words)

Ορισμένες λέξεις έχουν ιδιαίτερη σημασία για την ρύθμιση και δεν μπορούν να χρησιμοποιηθούν σαν ονόματα μεταβλητών. Τα παρακάτω κομμάτια κώδικα θα εκδηλώσουν σφάλμα μεταγλώττισης.

Πρόκειται για 33 λέξεις στην τρέχουσα έκδοση της Python. Μπορούμε να δούμε ποιές είναι αυτές οι δεσμευμένες λέξεις με την παρακάτω εντολή:

```
help("keywords")
```

```
Here is a list of the Python keywords. Enter any keyword to get more help.
```

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

Η εντολή help

Γενικά με την εντολή `help` καλούμε για βοήθεια και πληροφορίες την Python:

```
help(print)
```

Help on built-in function print in module builtins:

```
print(...)  
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)  
  
    Prints the values to a stream, or to sys.stdout by default.  
    Optional keyword arguments:  
    file: a file-like object (stream); defaults to the current sys.stdout.  
    sep:  string inserted between values, default a space.  
    end:  string appended after the last value, default a newline.  
    flush: whether to forcibly flush the stream.
```

help(abs)

Help on built-in function abs in module builtins:

```
abs(x, /)  
    Return the absolute value of the argument.
```

help(max)

Help on built-in function max in module builtins:

```
max(...)  
    max(iterable, *[, default=obj, key=func]) -> value  
    max(arg1, arg2, *args, *[, key=func]) -> value  
  
    With a single iterable argument, return its biggest item. The  
    default keyword-only argument specifies an object to return if  
    the provided iterable is empty.  
    With two or more arguments, return the largest argument.
```

Αλλαγή Πεζών Κεφαλαίων (Convert case)

Μπορούμε να κάνουμε αλλαγή ανάμεσα σε κεφαλαία και πεζά με τις παρακάτω μεθόδους συμβολοσειρών: `upper()`, `title()`, `lower()`. Αξίζει να σημειώσουμε ότι οι μέθοδοι αυτές δεν έχουν επίδραση στην μεταβλητή που τις καλούμε αλλά πρέπει να επαναεκχωρήσουμε το αποτέλεσμα της μεθόδου στην μεταβλητή με το ίδιο όνομα.

```
agios="άγιος νικόλαος"  
  
print(agios.upper())  
print(agios) # ο agios παραμένει "άγιος νικόλαος"  
  
print(agios.title())  
print('ΑΓΙΑ ΕΛΕΝΗ'.lower())  
  
agios = agios.upper()  
print(agios) # ο agios μετά την εκχώρηση στην ίδια μεταβλητή γίνεται ΑΓΙΟΣ  
ΝΙΚΟΛΑΟΣ
```

```
ΑΓΙΟΣ ΝΙΚΟΛΑΟΣ  
άγιος νικόλαος  
Άγιος Νικόλαος  
αγία ελένη  
ΑΓΙΟΣ ΝΙΚΟΛΑΟΣ
```

Οι συμβολοσειρές είναι μη μεταβαλλόμενη δομή δεδομένων

Οι συμβολοσειρές αποτελούνται από ακολουθίες χαρακτήρων με σταθερό μέγεθος και μη μεταβαλλόμενα περιεχόμενα. Αυτό σημαίνει ότι δεν είναι δυνατόν να προστίθενται ή να αφαιρούνται χαρακτήρες, ούτε να τροποποιούνται τα περιεχόμενα του αλφαριθμητικού. Πρόκειται για μια μη μεταβαλλόμενη (immutable) δομή της Python. Η αρίθμηση των χαρακτήρων σε ένα αλφαριθμητικό ξεκινάει από το 0.

Έτσι στην συμβολοσειρά `country = Ελλάδα` έχουμε:

`country[0]` → Ε (η αρίθμηση ξεκινά από το 0)

`country[1]` → λ

`country[2]` → λ

country[3] → ά

country[4] → δ

country[5] → α

Η παραπάνω συμβολοσειρά έχει μήκος 6 χαρακτήρες.

Μήκος συμβολοσειράς

Μέσω της συνάρτησης `len` η Python μας επιστρέφει το μήκος συμβολοσειράς δηλαδή το πλήθος των χαρακτήρων (μαζί με τα κενά) από τους οποίους αποτελείται.

```
message = 'Η τώρα ή ποτέ.'  
len(message)
```

14

Η μέθοδος find

Η μέθοδος `find` μας επιτρέπει να αναζητήσουμε μια συμβολοσειρά μέσα σε μια άλλη συμβολοσειρά. Η μέθοδος μας επιστρέφει την τοποθεσία από την ξεκινάει η αναζητούμενη συμβολοσειρά δηλαδή τον δείκτη (index) στην οποία εντοπίζεται ο πρώτος χαρακτήρας της αναζητούμενης συμβολοσειράς μέσα στα περιεχόμενα της αρχικής συμβολοσειράς. Στην παρακάτω συμβολοσειρά θα αναζητήσουμε την λέξη `ποτέ`.

```
stixos = 'Η Ελλάδα ποτέ δεν πεθαίνει'  
index = stixos.find('ποτέ')
```

Κανονικά αν πάμε στον χαρακτήρα με ευρετήριο (index) 9 πρέπει να εντοπίσουμε τον πρώτο χαρακτήρα της συμβολοσειράς που είναι το `π`. Πράγματι:

```
stixos[index]
```

'π'

Αν δεν εντοπιστεί η λέξη που αναζητούμε στην συμβολοσειρά η Python θα επιστρέψει: `-1`

```
stixos.find('πάντα')
```

-1

Η αναζήτηση είναι case sensitive δηλαδή γίνεται διάκριση ανάμεσα σε πεζά και κεφαλαία.

```
stixos.find('Ελλάδα') # επιστρέφει τον δείκτη 2 γιατί εντοπίστηκε η λέξη κλειδί
```

2

```
stixos.find('ελλάδα') # επιστρέφει -1 γιατί δεν εντοπίστηκε η λέξη κλειδί
```

-1

Μια άλλη σημαντική μέθοδος των συμβολοσειρών είναι η μέθοδος `replace` κατά την οποία μπορούμε να αντικαταστήσουμε τα περιεχόμενα μιας συμβολοσειράς. Στην πρώτη παράμετρο ορίζουμε την συμβολοσειρά που θέλουμε να αντικαταστήσουμε με την δεύτερη παράμετρο.

```
stixos.replace('ποτέ', 'πάντα')
```

'Η Ελλάδα πάντα δεν πεθαίνει'

3. Λίστες. Εκφράσεις, τελεστές

Στην τρέχουσα ενότητα γίνεται αναφορά σε μια από τις πιο δυνατές και χρήσιμες δομές της Python, τις λίστες. Οι λίστες μας επιτρέπουν να αποθηκεύουμε σύνολα πληροφορίας σε ένα μέρος είτε πρόκειται για ένα είτε για εκατομμύρια στοιχεία. Οι λίστες αποτελούνται από μία σειρά από στοιχεία, καθένα από τα οποία μπορεί να ανήκει σε διαφορετικό τύπο δεδομένων.

Definition 4

Μία **λίστα** (list) είναι μια διατεταγμένη συλλογή τιμών, οι οποίες αντιστοιχίζονται σε δείκτες. Οι τιμές που είναι μέλη μιας λίστας ονομάζονται **στοιχεία** (elements). Τα στοιχεία μιας λίστας δεν χρειάζεται να είναι ίδιου τύπου και ένα στοιχείο σε μία λίστα μπορεί να υπάρχει περισσότερες από μία φορές. Μία λίστα μέσα σε μία άλλη λίστα ονομάζεται **εμφωλευμένη λίστα** (nested list). Επιπρόσθετα, τόσο οι λίστες όσο και οι συμβολοσειρές, που συμπεριφέρονται ως διατεταγμένες συλλογές τιμών, ονομάζονται **ακολουθίες** (sequences). Τα στοιχεία μιας λίστας διαχωρίζονται με κόμμα και περικλείονται σε τετράγωνες αγκύλες ([και]). Μία λίστα που δεν περιέχει στοιχεία ονομάζεται άδεια λίστα και συμβολίζεται με [] [[Αγγελοδάκης, 2015](#)].

Παρακάτω δίνονται μερικά παραδείγματα από λίστες

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
colors = ["red", "green", "black", "blue"]
scores = [10, 8, 9, -2, 9]
myList = ["one", 2, 3.0]
languages=[['English'], ['Gujarati'], ['Hindi'], 'Romanian', 'Spanish'] # εμφωλευμένη
λίστα (nested list)
list_A = [] # άδεια λίστα
```

Οι λίστες είναι ταξινομημένες συλλογές δεδομένων. Η πρόσβαση στα στοιχεία της λίστας γίνεται μέσω του δείκτη ή της θέσης του κάθε στοιχείου. Το σημαντικό που πρέπει να συγκρατηθεί είναι ότι η αρίθμηση των στοιχείων σε μια λίστα ξεκινάει από το μηδέν. Το πρώτο στοιχείο έχει δείκτη 0, το δεύτερο 1 κ.ο.κ. Το τελευταίο στοιχείο στην λίστα έχει τον δείκτη -1, το δεύτερο στοιχείο από το τέλος τον δείκτη -2 κ.ο.κ. Δείτε στο παρακάτω παράδειγμα τι εκτυπώνεται με βάση την θέση που δηλώνουμε στην λίστα.

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles[0])
print(bicycles[1])
print(bicycles[-1])
print(bicycles[1:3])
print(bicycles[1:3])
```

```
trek
cannondale
specialized
['cannondale', 'redline']
['cannondale', 'redline']
```

Παράδειγμα με εμφωλευμένη λίστα (nested list):

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

print(two_by_two[0][1])
print(two_by_two[1][1])
```

```
2
4
```

Οι λίστες που περιέχουν συνεχόμενους ακέραιους αριθμούς μπορούν εύκολα να δημιουργηθούν ως εξής:

```
mylist = list(range(1,20))
print(mylist)

mylist1 = list(range(10))
print(mylist1)

mylist2 = list(range(1,20,4))
print(mylist2)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[1, 5, 9, 13, 17]
```

Η μέθοδος `index()` μιας λίστας επιστρέφει το ευρετήριο (index) μιας τιμής:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles.index('redline'))
```

```
2
```

Σε αντίθεση με τις συμβολοσειρές (strings) οι λίστες είναι μεταβαλλόμενες δομές δεδομένων. Αυτό σημαίνει ότι μπορούμε να τροποποιήσουμε τα στοιχεία της λίστα, να προσθέσουμε νέα ή να αφαιρέσουμε.

Για παράδειγμα μπορούμε να τροποποιήσουμε τα στοιχεία μιας λίστας ως εξής:

```
colors=['caramel','gold','silver','occur']
colors[3]='bronze'
print(colors)
```

```
['caramel', 'gold', 'silver', 'bronze']
```

```
colors=['caramel','gold','silver','occur']
colors[2:]=['bronze','silver']
print(colors)
```

```
['caramel', 'gold', 'bronze', 'silver']
```

```
colors=['caramel','gold','silver','occur']
colors[2:3]=['bronze','silver']
print(colors)
```

```
['caramel', 'gold', 'bronze', 'silver', 'occur']
```

Μπορούμε να προσθέσουμε ένα στοιχείο σε μια λίστα με την μέθοδο *append* π.χ.

```
gods = ['Δίας', 'Ερμής', 'Ποσειδώνας', 'Ήφαιστος']
print("{} θεοί".format(len(gods)))
print(gods)

gods.append("Απόλλωνας")
gods.append("Άρης")
print("{} θεοί".format(len(gods)))
print(gods)
```

```
4 θεοί
['Δίας', 'Ερμής', 'Ποσειδώνας', 'Ήφαιστος']
6 θεοί
['Δίας', 'Ερμής', 'Ποσειδώνας', 'Ήφαιστος', 'Απόλλωνας', 'Άρης']
```

Αν επιθυμούμε να προσθέσουμε πολλά στοιχεία τότε χρησιμοποιείται η μέθοδος *extend*

```
months=["Ιανουάριος", "Φεβρουάριος", "Μάρτιος"]
months.extend(["Απρίλιος", "Μαΐος"]) # προσοχή το όρισμα στην μέθοδο extend είναι
λίστα (ή κάποιο iterable object)
```

Μπορούμε να αφαιρέσουμε στοιχεία από μία λίστα με διάφορους τρόπους.

Με την πρόταση *del*:

Όταν γνωρίζουμε την θέση του στοιχείου που θέλουμε να αφαιρεθεί από μια λίστα χρησιμοποιούμε την πρόταση *del*. Για παράδειγμα

```
gods = ['Δίας', 'Ερμής', 'Ποσειδώνας', 'Ήφαιστος']
del gods[2]
print(gods)

# Το μήκος άλλαξε και μερικές από τις αντιστοιχίες στην λίστα. Πλέον ο θεός στην
θέση 2 είναι ο:
print(gods[2])
```

```
['Δίας', 'Ερμής', 'Ήφαιστος']  
Ήφαιστος
```

Με την μέθοδο *pop*:

Μία άλλη χρήσιμη μέθοδο για την αφαίρεση στοιχείων από μια λίστα είναι η μέθοδος *pop*. Μέσω της μεθόδου αυτής όχι απλά αφαιρείται το στοιχείο από την λίστα αλλά επιστρέφεται και ως τιμή διαθέσιμη να την εκμεταλλευτεί ο προγραμματιστής π.χ. σε μία νέα μεταβλητή. Δείτε το εξής παράδειγμα:

```
cars=["Alfa Romeo", "Renault", "BMW", "Renault", "Porsche"]  
speed_car=cars.pop()  
print(cars)  
print(speed_car)  
  
speed_car2=cars.pop(0)  
print(speed_car2)
```

```
['Alfa Romeo', 'Renault', 'BMW', 'Renault']  
Porsche  
Alfa Romeo
```

Όπως φαίνεται από το παράδειγμα αν χρησιμοποιηθεί η μέθοδος *pop()* χωρίς δείκτη τότε αφαιρείται το τελευταίο στοιχείο της λίστας.

Με την μέθοδο *remove*:

Επιπλέον μπορούμε να αφαιρέσουμε στοιχεία από μία λίστα με την χρήση μιας τιμής και όχι με βάση τον δείκτη. Ωστόσο η παραπάνω τεχνική θα αφαιρέσει το πρώτο στοιχείο που θα εντοπιστεί με την τιμή αυτή.

```
gods = ['Δίας', 'Ερμής', 'Ποσειδώνας', 'Ερμής', 'Ήφαιστος']  
  
gods.remove("Ερμής")  
print(gods)
```

```
['Δίας', 'Ποσειδώνας', 'Ερμής', 'Ήφαιστος']
```

Χρήσιμες μέθοδοι (methods) και συναρτήσεις (functions)

Μία από τις πλέον χρήσιμες μεθόδους της κλάσης *list* είναι η ταξινόμηση (*sort*).

```
cars= ["Porsche", "Alfa Romeo", "Renault", "BMW", "Audi" ]  
cars.sort()  
print(cars)
```

```
['Alfa Romeo', 'Audi', 'BMW', 'Porsche', 'Renault']
```

όπως φαίνεται η ταξινόμηση των στοιχείων της λίστας είναι μόνιμη. Ωστόσο αν θέλουμε προσωρινή ταξινόμηση τότε χρησιμοποιείται η function *sorted*.

```
cars= ["Porsche", "Alfa Romeo", "Renault", "BMW", "Audi" ]  
print(sorted(cars))  
print(cars)
```

```
['Alfa Romeo', 'Audi', 'BMW', 'Porsche', 'Renault']  
['Porsche', 'Alfa Romeo', 'Renault', 'BMW', 'Audi']
```

Όπως βλέπετε κατά την τελευταία εκτύπωση η λίστα διατηρεί την αρχική ταξινόμηση.

Επιπλέον με την μέθοδο *reverse* μπορούμε να αντιστρέψουμε την διάταξη των στοιχείων της λίστας. Και εδώ το αποτέλεσμα είναι μόνιμο.

```
cars= ["Porsche", "Alfa Romeo", "Renault", "BMW", "Audi" ]  
  
cars.reverse()  
print(cars)
```

```
['Audi', 'BMW', 'Renault', 'Alfa Romeo', 'Porsche']
```

Ακόμα μέσω της συνάρτησης `len` επιστρέφεται το πλήθος των στοιχείων της λίστας.

```
languages=['English', 'Gujarati', 'Hindi', 'Romanian', 'Spanish']  
print(len(languages))
```

5

Εδώ πρέπει να δοθεί προσοχή. Γιατί ενώ η αρίθμηση των δεικτών ξεκινά από το 0 το μήκος της λίστας ξεκινά από το 1 για λίστα με ένα στοιχείο. Οπότε στο παρακάτω παράδειγμα θα λάβουμε σφάλμα εκτέλεσης αν πάμε να πάρουμε το 5ο και τελευταίο στοιχείο της λίστας. Γιατί αυτό ορίζεται με τον δείκτη 4 και όχι 5.

```
cars= [ "Porsche", "Alfa Romeo", "Renault", "BMW", "Audi" ]  
print("Το μήκος της λίστας (πλήθος στοιχείων) είναι: ", len(cars))  
print(cars[5])
```

Το μήκος της λίστας (πλήθος στοιχείων) είναι: 5

```
-----  
IndexError                                Traceback (most recent call last)  
Input In [18], in <cell line: 3>()  
      1 cars= [ "Porsche", "Alfa Romeo", "Renault", "BMW", "Audi" ]  
      2 print("Το μήκος της λίστας (πλήθος στοιχείων) είναι: ", len(cars))  
----> 3 print(cars[5])  
  
IndexError: list index out of range
```

Σημείωση

Συχνά χρησιμοποιούμε έναν βρόγχο (loop) όπως το `for` για να προσπελάσουμε ένα προς ένα τα στοιχεία μιας λίστας. Δείτε το επόμενο παράδειγμα.

```
list = ['physics', 'chemistry', 1997, 2000]  
for item in list:  
    print(item)
```

physics
chemistry
1997
2000

Σημαντικό

Ιδιαίτερη προσοχή πρέπει να δίνεται στον τρόπο που αντιγράφουμε λίστες. Δείτε γιατί στο επόμενο παράδειγμα.

```
nisia = ["Μήλος", "Κρήτη", "Λέσβος"]  
greek_islands = nisia  
  
greek_islands.append("Κέρκυρα")  
  
print(nisia)
```

['Μήλος', 'Κρήτη', 'Λέσβος', 'Κέρκυρα']

Όπως φαίνεται οι μεταβλητές `nisia` και `greek_islands` αντιστοιχούν στο ίδιο υποκείμενο object και αν μεταβάλλοντας τα στοιχεία της μίας μεταβλητής η αλλαγή αντικατοπτρίζεται και στα στοιχεία της δεύτερης. Ο ενδεξιγμένος τρόπος για να αντιγράψουμε μια λίστα είναι να αντιγράψουμε όλα τα στοιχεία της ώστε να έχουμε δύο ανεξάρτητα objects (κλωνοποίηση).


```
nisia = ["Μήλος", "Κρήτη", "Λέσβος"]
greek_islands = nisia[:]

greek_islands.append("Κέρκυρα")

print(nisia)

print(greek_islands)
```

```
['Μήλος', 'Κρήτη', 'Λέσβος']
['Μήλος', 'Κρήτη', 'Λέσβος', 'Κέρκυρα']
```

Εκφράσεις και τελεστές

Μία **έκφραση (expression)** είναι ένας συνδυασμός από *τιμές, μεταβλητές, τελεστές* και *κλήσεις σε συναρτήσεις*. Οι **τελεστές (operators)** είναι λειτουργίες που κάνουν κάτι και μπορούν να αναπαρασταθούν με σύμβολα όπως το + ή με λέξεις κλειδιά όπως το and. Η αποτίμηση μιας έκφρασης παράγει μία τιμή και αυτός είναι και ο λόγος που μία έκφραση μπορεί να βρίσκεται στο δεξί μέρος μια εντολής εκχώρησης. Όταν μία μεταβλητή εμφανίζεται σε έκφραση, αντικαθίσταται από την τιμή της, προτού αποτιμηθεί η έκφραση [Αγγελιδάκης, 2015]. Δεν απαιτείται μία έκφραση να περιέχει ταυτόχρονα και τιμές και μεταβλητές και τελεστές. Μία τιμή, όπως και μία μεταβλητή, από μόνες τους είναι επίσης εκφράσεις.

Οι τελεστές χρησιμοποιούνται με αριθμητικές τιμές για την εκτέλεση μαθηματικών πράξεων. Ορισμένοι τελεστές έχουν εφαρμογή κα σε συμβολοσειρές. Πιο συγκεκριμένα διατίθενται οι παρακάτω τελεστές:

Αριθμητικοί τελεστές

Τελεστής	Όνομα	Παράδειγμα
+	Πρόσθεση αριθμών ή συνένωση συμβολοσειρών	x + y
-	Αφαίρεση	x - y
*	Πολλαπλασιασμός ή επανάληψη συμβολοσειράς	x * y
/	Διαίρεση	x / y
%	Υπόλοιπο διαίρεσης δύο αριθμών	x % y
**	Ύψωση αριθμού σε δύναμη	x ** y
//	Διαίρεση δύο αριθμών στρογγυλοποιημένη προς τα κάτω	x // y

Τελεστές εκχώρησης

Χρησιμοποιούνται για να αποδώσουν τιμές σε μεταβλητές.

Τελεστής	Παράδειγμα	Αντίστοιχο με
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
=	x = 3	x = x 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Τελεστές σύγκρισης

Χρησιμοποιούνται για την σύγκριση 2 τιμών

Τελεστής	Σύγκριση	Παράδειγμα
==	Ίσον	x == y
!=	Διαφορετικό	x != y
>	Μεγαλύτερο από	x > y
<	Μικρότερο από	x < y
>=	Μεγαλύτερο ή ίσον από	x >= y
<=	Μικρότερο ή ίσον από	x <= y

Λογικοί τελεστές

Τελεστής	Περιγραφή	Παράδειγμα
and	Επιστρέφει Αληθές (True) αν και οι δύο προτάσεις είναι αληθείς	x < 5 and x < 10
or	Επιστρέφει Αληθές (True) αν έστω μία από τις προτάσεις είναι αληθή	x < 5 or x < 4
not	Αντιστροφή αποτελέσματος, επιστρέφει Μη Αληθές όταν το αποτέλεσμα είναι αληθές	not(x < 5 and x < 10)

Εκτός από τους παραπάνω τελεστές υπάρχουν πιο εξειδικευμένοι τελεστές που δεν θα αναφερθούμε (Bitwise, Membership, Identity operators).

Οι τελεστές στη Python τηρούν την αντίστοιχη προτεραιότητα που χρησιμοποιείται και στα μαθηματικά.

Οι παρενθέσεις έχουν τη μεγαλύτερη προτεραιότητα. Πολλαπλασιασμός και διαίρεση έχουν υψηλότερα προτεραιότητα από την πρόσθεση και αφαίρεση.

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

4. Πλειάδες και λεξικά

Στην συνέχεια παρουσιάζονται δύο άλλες καθιερωμένες δομές στην Python, οι πλειάδες (tuples) και τα λεξικά (dictionaries).

Πλειάδες (tuples)

Οι πλειάδες είναι μια δομή δεδομένων παρόμοια με τις λίστες. Όπως και οι λίστες αποτελούν μια συλλογή αντικειμένων (objects). Όμως διαφέρουν σε ένα σημαντικό χαρακτηριστικό από τις λίστες, είναι αμετάβλητες (immutable), δηλαδή δεν μπορούμε να μεταβάλλουμε το περιεχόμενό τους εφόσον τις δημιουργήσουμε. Κατά συνέπεια ενώ για την προσπέλαση των στοιχείων της χρησιμοποιούνται ίδιες τεχνικές και μέθοδοι (indexing, slicing κτλ) δεν υποστηρίζουν ωστόσο τις αντίστοιχες μεθόδους αφαίρεσης, τροποποίησης και προσθήκης στοιχείων. Υπό την έννοια αυτή οι πλειάδες είναι μια ασφαλή δομή δεδομένων μέσω της οποίας εξασφαλίζεται ότι τα δεδομένα δεν θα τροποποιηθούν από λάθος ή ακόμα και από επιλογή. Συνήθως χρησιμοποιούνται κατά την επιστροφή πολλαπλών τιμών από μια συνάρτηση αλλά και όταν θέλουμε να «πακετάρουμε» δεδομένα (tuple packing). Οι πλειάδες ορίζονται με παρόμοιο τρόπο σαν τις λίστες αλλά αντι για αγκύλες χρησιμοποιούνται παρανθέσεις (αν και δεν είναι απαραίτητες). Για παράδειγμα:

```
information = ('UTH', 'https://www.uth.gr/')
```

Το ίδιο μπορεί να γραφτεί και χωρίς παρενθέσεις

```
information = 'UTH', 'https://www.uth.gr/'
```

Όμως πρέπει να λαμβάνουμε υπόψιν ότι ο ορισμός μιας πλειάδας με ένα μόνο στοιχείο πρέπει να ορίζεται με το στοιχείο και να συνοδεύεται από ένα κόμμα. Σε διαφορετική περίπτωση θα επιστρέψει μια μεταβλητή με τύπο αυτόν που περάσαμε στο υποτιθέμενο πρώτο στοιχείο. Δηλαδή το παρακάτω θα δημιουργήσει μια μεταβλητή ακέрайου τύπου και όχι μια πλειάδα με ένα στοιχείο

```
year = (2022)
print(type(year))
```

```
<class 'int'>
```

Ο σωστός τρόπος συγγραφής αν θέλουμε να πάρουμε μια πλειάδα με ένα μόνο στοιχείο είναι:

```
year = (2022,)
print(type(year))
```

```
<class 'tuple'>
```

Ενώ αν θέλουμε να δημιουργήσουμε μία άδεια πλειάδα τότε χρησιμοποιούμε απλώς δύο παρενθέσεις:

```
empty_tuple = ()
```

Επιπλέον μέσω της συνάρτησης *tuple* η οποία δέχεται ως όρισμα μία συμβολοσειρά ή λίστα μπορούμε να δημιουργήσουμε νέες πλειάδες. Προσέξτε το αποτέλεσμα όταν ορίζουμε σαν παράμετρο συμβολοσειρά στην συνάρτηση *tuple*.

```
regions= tuple("Κρήτη")  
print(regions)
```

```
('Κ', 'ρ', 'ή', 'τ', 'η')
```

```
regions= tuple(["Κρήτη", "Ήπειρος", "Θράκη"])  
print(regions)
```

```
('Κρήτη', 'Ήπειρος', 'Θράκη')
```

Με κενό όρισμα επιστρέφεται μία άδεια λίστα:

```
regions= tuple()
```

Όπως προαναφέρθηκε οι πλειάδες είναι αμετάβλητες δομές δεδομένων και έτσι δεν μπορούμε να χρησιμοποιήσουμε μεθόδους όπως *append*, *del*, *sort* κ.α. Μπορούμε, όπως και στις λίστες, να αναφερθούμε στα στοιχεία της πλειάδας με τους τελεστές `[]` και με το ευρετήριο (index) ή να εξαγάγουμε τμήμα από τα στοιχεία (slice) με τον τελεστή `:`. Λαμβάνουμε το πλήθος των στοιχείων της πλειάδας (length) με την συνάρτηση *len*.

Οι πλειάδες χρησιμοποιούνται για να πακετάρουμε μια συλλογή δεδομένων σε ένα αντικείμενο (tuple packing) π.χ.

```
fruit1 = "Μήλος"  
fruit2 = "Πορτοκάλι"  
fruit1 = "Μανταρίνι"  
fruits = fruit1, fruit1, fruit2 # tuple packing  
print(fruits)
```

```
('Μανταρίνι', 'Μανταρίνι', 'Πορτοκάλι')
```

Βέβαια υπάρχει και η αντίστροφη διαδικασία, το «ξε-πακετάρισμα» δεδομένων (tuple unpacking). Όταν δηλαδή αποδίδουμε τα στοιχεία της πλειάδας σε ξεχωριστές μεταβλητές.

```
cities = "Αθήνα", "Βόλος", "Πάτρα"  
capital, city1, city2 = cities # tuple packing  
print(capital, city1, city2)
```

```
Αθήνα Βόλος Πάτρα
```

Ο αμετάβλητος χαρακτήρας των πλειάδων φαίνεται στα παρακάτω παραδείγματα στα οποία επιχειρείται η τροποποίηση των δεδομένων της.

```
cities = ("Αθήνα", "Βόλος", "Πάτρα")  
cities.pop(1)
```

```
-----  
AttributeError                                Traceback (most recent call last)  
Input In [11], in <cell line: 2>()  
      1 cities = ("Αθήνα", "Βόλος", "Πάτρα")  
----> 2 cities.pop(1)  
  
AttributeError: 'tuple' object has no attribute 'pop'
```

```
cities.append("Ρόδος")
```

```
-----
AttributeError                                Traceback (most recent call last)
Input In [12], in <cell line: 1>()
----> 1 cities.append("Ρόδος")

AttributeError: 'tuple' object has no attribute 'append'
```

Όπως μας ενημερώνει και το σχετικό σφάλμα εκτέλεσης δεν υπάρχουν οι σχετικές μέθοδοι για αντικείμενα της κλάσης *tuple*.

Αντίστοιχο σφάλμα λαμβάνουμε και με τον παρακάτω κώδικα όταν πάμε να τροποποιήσουμε ένα στοιχείο της πλειάδας:

```
mytuple = (1, 2, 3)
mytuple[0] = 999
```

```
-----
TypeError                                Traceback (most recent call last)
Input In [13], in <cell line: 2>()
      1 mytuple = (1, 2, 3)
----> 2 mytuple[0] = 999

TypeError: 'tuple' object does not support item assignment
```

💡 Tip

Μια χρήσιμη λειτουργία των πλειάδων είναι η χρήση τους κατά την αντιμετάθεση δύο μεταβλητών. Δείτε το παρακάτω παράδειγμα.

```
island='Λέσβος'
island2='Χίος'
island, island2 = island2, island

print(island)
print(island2)
```

```
Χίος
Λέσβος
```

⚠ Προειδοποίηση

Προσοχή! Αν και οι πλειάδες είναι αμετάβλητος τύπος δεδομένων ωστόσο στα στοιχεία τους μπορούν να περιλαμβάνουν μεταβλητούς τύπους δεδομένων. Αυτό σημαίνει ότι μπορούμε να τροποποιήσουμε τα στοιχεία αυτά. Παρακάτω δίνεται ένα παράδειγμα.

```
cities = ("Αθήνα", ["Βόλος", "Πάτρα"])
cities[1][0] = "Καβάλα"
print(cities)
```

```
('Αθήνα', ['Καβάλα', 'Πάτρα'])
```

Μια εξήγηση σε αυτήν την αντιφατική συμπεριφορά δίνεται στο παρακάτω νήμα: <https://stackoverflow.com/questions/9755990/why-can-tuples-contain-mutable-items>

Λεξικά (Dictionaries)

Τα λεξικά στην Python αποτελούν συλλογές αντικειμένων, όπως οι λίστες και οι πλειάδες. Ένα σημαντικό χαρακτηριστικό των λεξικών είναι ότι αυτά αποθηκεύουν δεδομένα κατά ζεύγη, με την μορφή κλειδί-τιμή (key-value pairs). Κάθε κλειδί σε ένα λεξικό συνοδεύεται από μία τιμή. Κάθε κλειδί αποτελεί ουσιαστικά ένα μοναδικό αναγνωριστικό για την συνοδευτική τιμή και γι'αυτό τον λόγο δεν μπορεί να υπάρξει δεύτερο ίδιο κλειδί. Ακόμα, τα κλειδιά πρέπει να ορίζονται από αμετάβλητους τύπους δεδομένων δηλαδή είτε από μία συμβολοσειρά είτε από έναν ακέραιο ή δεκαδικό. Δεν μπορεί όμως μια λίστα να είναι κλειδί. Μια πλειάδα μπορεί να είναι κλειδί αλλά με την προϋπόθεση ότι και αυτή δεν θα αποτελείται από μεταβλητούς τύπους δεδομένων. Τα λεξικά περικλείονται σε {}, τα ζεύγη ορίζονται υπό την μορφή *κλειδί:τιμή* και χωρίζονται μεταξύ τους με κόμμα (,) δηλαδή:

```
d = {key1 : value1, key2 : value2 }
```

Για παράδειγμα:

```
phones ={"Χρήστος": "69936565", "Κώστας": "246541353", "Βαγγέλης": "546546536"}
print(phones)
```

```
{'Χρήστος': '69936565', 'Κώστας': '246541353', 'Βαγγέλης': '546546536'}
```

Τα κλειδιά (keys) σε αυτήν την περίπτωση είναι τα ονόματα και οι αριθμοί τηλεφώνων (ως συμβολοσειρές ορισμένες) οι τιμές (values).

Εναλλακτικά μπορούμε να δημιουργήσουμε ένα άδειο λεξικό και να προσθέσουμε στην συνέχεια ζεύγη.

```
phones ={}
phones["Χρήστος"] ="69936565"
phones["Κώστας"] ="246541353"
phones["Βαγγέλης"] ="546546536"
```

Ένας άλλος τρόπος δημιουργίας λεξικών είναι με την συνάρτηση *dict*.

```
phones =dict(Χρήστος="69936565",Κώστας="246541353", Βαγγέλης="546546536")
```

Με αυτόν τον τρόπο ορίζουμε τα κλειδιά μέσω μεταβλητών και γι' αυτό τον λόγο εφαρμόζονται οι περιορισμοί που αφορούν την ονοματολογία των μεταβλητών.

Στα λεξικά τα ζεύγη αυτά δεν ταξινομούνται με κάποια συγκεκριμένη σειρά αλλά με έναν μηχανισμό της Python που λέγεται *hashing* και αποσκοπεί στην γρήγορη ανάκτηση τους. Η ταξινόμηση αυτή αλλάζει κάθε φορά κατά τυχαίο τρόπο όταν τροποποιούμε ένα λεξικό. Για τον λόγο αυτό δεν υπάρχει η έννοια της θέσης ή του δείκτη (index) όπως στις λίστες και στις πλειάδες. Η ανάκτηση μιας τιμής από ένα ζεύγος γίνεται με βάση το κλειδί π.χ.* `dl["a_key"]` *. Γίνεται δηλαδή με παρόμοιο τρόπο όπως στις λίστες και τις πλειάδες αλλά αντι για το ευρετήριο θέσης ορίζουμε το κλειδί.

```
print(phones["Χρήστος"])
print(phones["Βαγγέλης"])
```

```
69936565
546546536
```

Όπως προαναφέρθηκε δεν μπορούμε να έχουμε διπλό κλειδί σε ένα λεξικό. Έτσι αν επιχειρήσουμε να προσθέσουμε ένα ζεύγος με υφιστάμενο κλειδί στην πράξη αυτό που θα γίνει είναι να αντικαταστήσουμε την παλιά τιμή με μια νέα:

```
print(phones["Βαγγέλης"])
phones["Βαγγέλης"]="666666666"
print(phones["Βαγγέλης"])
```

```
546546536
666666666
```

Για την διαγραφή ενός ζεύγους από λεξικό χρησιμοποιείται η συνάρτηση *del* δηλ. `del(dl["key"])`. Δείτε το επόμενο παράδειγμα:

```
del(phones["Βαγγέλης"])
print(phones)
```

```
{'Χρήστος': '69936565', 'Κώστας': '246541353'}
```

Με την συνάρτηση *len* επιστρέφεται το μέγεθος του λεξικού δηλαδή το πλήθος των ζευγαριών κλειδιών/τιμών που περιέχει:

```
print(len(phones))
```

```
2
```

Μέσω του τελεστή `*in*` διαπιστώνεται αν υπάρχει ένα κλειδί σε ένα λεξικό:

```
print("Χρήστος" in phones)
```

```
True
```

Με την μέθοδο *keys* ενός λεξικού επιστρέφονται τα κλειδιά του. Αντίστοιχα με την μέθοδο *values* επιστρέφονται οι τιμές του ενώ με την μέθοδο *items* επιστρέφονται τα ζεύγη κλειδιών/τιμών. Τα επιστρεφόμενα objects είναι αντιστοίχα *dict_keys*, *dict_values*, *dict_items*. Ο παρακάτω κώδικας περιγράφει τις παραπάνω λειτουργίες:

```
d = {'a': 10, 'b': 20, 'c': 30}
print(d.keys())
print(d.values())
print(d.items())
```

```
dict_keys(['a', 'b', 'c'])
dict_values([10, 20, 30])
dict_items([('a', 10), ('b', 20), ('c', 30)])
```

Αν θέλουμε να ταξινομήσουμε το λεξικό με βάση τα κλειδιά τότε το κάνουμε με την συνάρτηση *sorted*:

```
thisdict = {
    "year": 1964,
    "brand": "Ford",
    "model": "Mustang"
}

print(sorted(thisdict))
```

```
['brand', 'model', 'year']
```

Πολύ συνηθισμένη περίπτωση είναι τα λεξικά να περιλαμβάνουν σαν τιμές άλλα λεξικά.

```
contacts = {"Χρήστος": {"Σπίτι": "457456456", "Εργασία": "48856"},
            "Γιάννης": {"Σπίτι": "8753778", "Εργασία": "45654656"},
            "Κώστας": {"Κινητό": "45475354"}}
```

Σε αυτή την περίπτωση ορίζοντας διαδοχικά τα κλειδιά παίρνουμε τις τιμές που επιθυμούμε:

```
print(contacts["Χρήστος"])
print(contacts["Χρήστος"]["Εργασία"])
```

```
{'Σπίτι': '457456456', 'Εργασία': '48856'}
48856
```

Μπορούμε να διατρέξουμε τα ζεύγη ενός λεξικού αλλά η σειρά που θα γίνεται η ανάκτηση μπορεί να μην είναι με την σειρά που ορίζονται και όχι πάντα η ίδια.

```
for phone in phones:
    print(phones[phone])
```

```
69936565
246541353
```

Όπως και με τις λίστες έτσι και με τα λεξικά πρέπει να είμαστε προσεκτικοί όταν αντιγράφουμε ένα λεξικό. Η αντιγραφή μπορεί να αναφέρεται στο ίδιο αντικείμενο και τροποποίηση των δεδομένων ενός λεξικού θα επιφέρει και την αντίστοιχη τροποποίηση στο άλλο.

```
names1 = {'name': "Κώστας"}
names2=names1
names2['name']="Γιάννης"
print(names1)
```

```
{'name': 'Γιάννης'}
```

Σε αυτή την περίπτωση χρησιμοποιείται η μέθοδος *copy* για την αντιγραφή των δεδομένων ενός λεξικού σε ένα άλλο.

```
names1 = {'name': "Κώστας"}
names2=names1.copy()
names2['name']="Γιάννης"
print(names1)
```

```
{'name': 'Κώστας'}
```

5. Έλεγχος ροής εκτέλεσης

Όλες οι γλώσσες προγραμματισμού appartίζονται από μια σειρά από προγραμματιστικές δομές. Οι τρεις βασικές δομές ελέγχου ροής προγράμματος είναι η δομή της ακολουθίας εντολών, η δομή της απόφασης και η δομή της επανάληψης. Κάποιες από αυτές αντιστοιχούν σε περισσότερες από μία εντολές [Manis, 2015].

Η λογική boolean

Για την υλοποίηση των δομών απόφασης χρησιμοποιείται η λογική **boolean** κατά την οποία ελέγχονται μία ή περισσότερες συνθήκες και ανάλογα το αποτέλεσμα επιλέγεται ποια ακολουθία εντολών θα εκτελεστεί. Το όνομα **boolean** προέρχεται από τον Βρετανό Μαθηματικό George Boole, ο οποίος εισήγαγε την ομώνυμη άλγεβρα που αφορά σε λογικούς κανόνες συνδυασμού των δύο τιμών True (Αληθής) και False (Ψευδής). Οι τιμές αυτές ονομάζονται λογικές ή Boolean και ο τύπος τους στην Python είναι ο bool [Αγγελιδάκης, 2015].

```
type(False)
```

```
bool
```

Μία μεταβλητή μπορεί να δείχνει και σε μία τιμή τύπου bool. Για παράδειγμα:

```
raining=True  
type(raining)
```

```
bool
```

Οι λογικές μεταβλητές είναι μεταβλητές οι οποίες διέπονται από την δυαδική λογική στην επιστήμη της πληροφορικής (1 και 0) και παίρνουν δύο τιμές *True* ή *False*. Ουσιαστικά πρόκειται για συνώνυμα του 1 (True) και 0 (False).

```
True+True  
42 * True + False
```

```
42
```

Ο συνδυασμός των Boolean τιμών γίνεται με τη χρήση τριών βασικών λογικών τελεστών: **not**, **and** και **or**. Οι τελεστές αυτοί συμβολίζουν λογικές πράξεις (όχι, και, ή) και βοηθούν στη λήψη αποφάσεων σε ένα πρόγραμμα. Η σημαντική τους είναι πολύ παρόμοια με την καθημερινή τους σημασία [Αγγελιδάκης, 2015]. Ας υποθέσουμε ότι *A* και *B* είναι δύο μεταβλητές που δείχνουν σε Boolean τιμές ή δύο εκφράσεις (ονομάζονται λογικές ή Boolean) που αποτιμώνται σε Boolean τιμές. Ο επόμενος πίνακας, ο οποίος ονομάζεται πίνακας αληθείας, περιέχει τις τιμές που επιστρέφουν οι τρεις λογικές πράξεις για όλους τους συνδυασμούς τιμών των *A* και *B*.

A	B	A AND B	A OR B	NOT A
False	False	False	False	True
False	True	False	True	True
True	False	False	True	False
True	True	True	True	False

Εικ. 4 Πίνακας αληθείας για τους βασικούς λογικούς τελεστές.

Η λογική έκφραση **not A** είναι αληθής, όταν η *A* ψευδής, και ψευδής, όταν η *A* είναι αληθής. Η λογική έκφραση **A and B** είναι αληθής μόνο όταν και η *A* και η *B* είναι αληθείς, σε κάθε άλλη περίπτωση είναι ψευδής. Η λογική έκφραση **A or B** είναι αληθής όταν η *A* είναι αληθής ή η *B* είναι αληθής, ή όταν και οι δύο είναι αληθείς.

Όταν χρησιμοποιούμε τον όρο **and** σημαίνει ότι και οι δύο προτάσεις πρέπει να είναι αληθείς για να αποδόσουν αληθές αποτέλεσμα. **true**.

Γενικά μπορούμε να συνοψίσουμε την χρήση του τελεστή **and** και την σύγκριση ανάμεσα σε τιμές boolean με βάση των παρακάτω πίνακα [[Heisler, 2021](#)]:

Συνδυασμός με τον τελεστή and	Αποτέλεσμα
True and True	True
True and False	False
False and True	False
False and False	False

Με την χρήση του όρου **or** αρκεί μία από τις δύο προτάσεις να είναι True για να αποδόσει αποτέλεσμα αληθές. Αυτό φαίνεται πιο καλά στο παραπάνω πίνακα:

Συνδυασμός με τον τελεστή or	Αποτέλεσμα
True or True	True
True or False	True
False or True	True
False or False	False

Τέλος με την χρήση του όρου **not** αντιστρέφεται η τιμή True ή False μιας πρότασης.

Επίδραση του τελεστή not	Αποτέλεσμα
not True	False
not False	True

Φυσικά μπορούμε να συνδυάσουμε να συνδυάσουμε τους όρους **and or** και **not** και να διαχωρίσουμε τις συγκρίσεις με παρενθέσεις ώστε να δημιουργήσουμε πιο σύνθετες εκφράσεις σύγκρισης:


False or (False and True)
False
True and (False or True)
True
(not False) or True
True
False or (not False)
True

Σε μία λογική έκφραση μπορούμε να έχουμε και τελεστές σύγκρισης. Η Python έχει 6 τελεστές σύγκρισης:

Μικρότερο από (<) Μικρότερο/ίσο από (<=) Μεγαλύτερο από (>) Μεγαλύτερο/ίσο από (>=) Ίσο με(==) Διαφορετικό από (!=) Αυτοί οι τελεστές συγκρίνουν δύο τιμές και επιστρέφουν μια τιμή τύπου boolean δηλαδή είτε *True* είτε *False*.

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

<, >, <=, >=, !=, ==, not, and, or.



Προειδοποίηση
Προσοχή! Δεν πρέπει να συγχέεται ο τελεστής εκχώρησης τιμών σε μεταβλητή = με τον τελεστή ισότητας ==.

Μερικά παραδείγματα τελεστών σύγκρισης


```
print(1 != 2)
print(1 != 1)
```

True
False

```
a = 1
b = 2
print(a == b)
a = b
a==b
print(a!=b)
```

False
False

```
1 < 2 and 3 < 4
```

True

```
2 < 1 and 4 < 3
```

False

```
1 < 2 and 4 < 3
```

False

```
2 < 1 and 3 < 4
```

False

Τους τελεστές σύγκρισης μπορούμε να τους χρησιμοποιήσουμε και με συμβολοσειρές:

```
"dog" == "cat"
```

False

```
"dog" == "dog"
```

True

```
"dog" != "cat"
```

True

Για να είναι δύο συμβολοσειρές ίσες πρέπει να έχουν ακριβώς την ίδια τιμή. Διαφοροποιήσεις ανάμεσα σε κεφαλαία ή μικρά ή η ύπαρξη κενών ανάμεσα σε δύο μεταβλητές συμβολοσειρών θα αποδώσει False σε μια έκφραση σύγκρισης ισότητας δηλαδή ότι οι δύο συμβολοσειρές δεν είναι ίσες.

Μπορούμε να συνδυάσουμε τους τελεστές σύγκρισης με λογικές τιμές boolean και τους λογικούς τελεστές **or**, **and** και **not**.

Για παράδειγμα:

```
True and not (1 != 1)
```

True

```
("A" != "A") or not (2 >= 3)
```

```
True
```

α. Η δομή της ακολουθίας εντολών

Οι εντολές σε ένα πρόγραμμα εκτελούνται σειριακά ή μία μετά την άλλη ξεκινώντας από την πρώτη. Τελειώνει η εκτέλεση μίας εντολής και ακολουθεί η εκτέλεση της επόμενης. Κάθε εντολή θα εκτελεστεί ακριβώς μία φορά, χωρίς δηλαδή να παραλειφθεί καμία από αυτές ή να επιστρέψουμε για να ξαναεκτελέσουμε κάποια. Παρακάτω δίνεται ένα χαρακτηριστικό παράδειγμα δομής ακολουθίας εντολών.

```
A=1  
B=5  
C=A+B  
print(C)
```

```
6
```

Σε αυτό το παράδειγμα οι εντολές εκτελούνται στο σύνολό τους, διαδοχικά ή μία μετά την άλλη.

β. Η δομή της απόφασης

Στη δομή της απόφασης έχουμε μία περισσότερο πολύπλοκη δομή, στην οποία ο έλεγχος του προγράμματος καλείται να επιλέξει ανάμεσα σε δύο ή και περισσότερες διαφορετικές διαδρομές ανάλογα με το αν ισχύει ή όχι κάποια ή κάποιες συνθήκες. Για την υλοποίηση της απόφασης, κάθε γλώσσα μπορεί να έχει μία ή περισσότερες δομές. Η πιο συνηθισμένη είναι η δομή **if**, ενώ υπάρχει συνήθως και μία δομή για πολλαπλή απόφαση. Η Python έχει την **if-elif-else** για να υλοποιήσει την απόφαση, αλλά δεν έχει δομή για την πολλαπλή απόφαση και χρησιμοποιεί την **if-elif-else** για τον σκοπό αυτόν [\[Manis, 2015\]](#).

Η εντολή **if** χρησιμοποιείται για έλεγχο της ροής εκτέλεσης ενός προγράμματος. Ελέγχεται μία συνθήκη και ανάλογα με το αποτέλεσμα (Αληθής ή Ψευδής) εκτελείται ή δεν εκτελείται μία ή κάποια άλλη ομάδα (μπλοκ) εντολών [\[Αγγελιδάκης, 2015\]](#). Η εντολή **if** συντάσσεται ως εξής:

```
if συνθήκη:  
    μπλοκ εντολών 1 (true_block)  
  
else:  
    μπλοκ εντολών (false_block)
```

Κατά την εκτέλεση του παρακάτω κώδικα η Python θα δοκιμάσει την έκφραση (expression). Αν επιστρέψει True τότε θα εκτελέσει τις προτάσεις κώδικα που περιλαμβάνει η ενότητα `true_block`. Αν επιστρέψει όμως False θα εκτελέσει τις εντολές στην ενότητα `false_block`. Στην πιο απλή της μορφή η δομή της απόφασης μπορεί να παραλείπει το `else` τμήμα και το αντίστοιχο μπλοκ εντολών δηλ:

```
if συνθήκη:  
    μπλοκ εντολών (true_block)
```

Παρατηρήστε ότι η εντολή **if** μετά τον έλεγχο της συνθήκης κλείνει με `:` και στην συνέχεια ακολουθούν με εσοχή (indentation) οι εντολές/προτάσεις που θα εκτελεστούν εφόσον αληθεύει το αποτέλεσμα του ελέγχου. Από `:` ακολουθείται και η εντολή **else**. Το μπλοκ της εντολής ονομάζεται σώμα (body). Πρέπει να περιέχει υποχρεωτικά μια εντολή. Αν προσωρινά δεν θέλουμε να περιέχει κάποια εντολή μπορούμε να εισάγουμε την εντολή **pass**. Ιδιαίτερη σημασία πρέπει να δίνεται κατά την σύνταξη κατά την χρήση δηλαδή του σημείου `:` και την εσοχή των εντολών. Παρακάτω δίνεται ένα απλό παράδειγμα δομής απόφασης:

```
weight = 100  
if weight > 90:  
    print("Είστε υπέρβαρος.Παρακαλώ αποφύγετε τον ανελκυστήρα.")  
print("Ευχαριστούμε για την συνεργασία.")
```

```
Είστε υπέρβαρος.Παρακαλώ αποφύγετε τον ανελκυστήρα.  
Ευχαριστούμε για την συνεργασία.
```

Παράδειγμα κώδικα **if-else**:

```
temperature = 20
if temperature > 30:
    print('Φορέστε κοντομάνικα.')
else:
    print('Φορέστε μακρυμάνικα.')
print('Ευχαριστώ.')
```

Φορέστε μακρυμάνικα.
Ευχαριστώ.

Για να προσθέσουμε περισσότερες επιλογές κατά τον έλεγχο συνθηκών χρησιμοποιούμε την σύνταξη if-elif-else:

```
score=90
if score >= 90:
    letter = 'A'
elif score >= 80:
    letter = 'B'
elif score >= 70:
    letter = 'C'
elif score >= 60:
    letter = 'D'
else:
    letter = 'F'
```

Οι συνθήκες if είναι δυνατόν να είναι αλυσιδωτές (εμφωλευμένες) δηλαδή if πρόταση μέσα σε άλλη if πρόταση:

```
num = 15
if num >= 0:
    if num == 0:
        print("0 αριθμός δεν είναι ούτε αρνητικός ούτε θετικός")
    else:
        print("0 αριθμός είναι θετικός")
else:
    print("0 αριθμός είναι θετικός")
```

0 αριθμός είναι θετικός

γ. η δομή της επανάληψης

Η δομή της επανάληψης είναι μια μορφή κώδικα κατά την οποία περιλαμβάνει εντολές οι οποίες εκτελούνται επαναληπτικά για όσο τηρείται μια συνθήκη. Με αυτόν τον τρόπο αποφεύγεται η επανάληψη της συγγραφής του ίδιου κώδικα πολλές φορές. Στο παρακάτω τμήμα κώδικα είναι χαρακτηριστικό πως επαναλαμβάνεται ο ίδιος κώδικας πολλές φορές.

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

Hello World
Hello World
Hello World
Hello World
Hello World

Αυτό όμως θα μπορούσε να αποφευχθεί με την χρήση μια δομής επανάληψης. Μια δομή επανάληψη ονομάζεται και βρόγχος (loop). Οι δομές επανάληψης διαχωρίζονται σε 3 κατηγορίες:

- ο βρόγχος **while**, ο οποίος επαναλαμβάνει μια πρόταση ή μια σειρά προτάσεων όσο ισχύει μια συνθήκη. Κάθε φορά που επαναλαμβάνεται ένας κύκλος εκτέλεσης δοκιμάζεται αν ισχύει η συνθήκη αυτή.
- ο βρόγχος **for**, κατά τον οποίο επαναλαμβάνεται μια σειρά προτάσεων για κάθε ένα στοιχείο μιας ακολουθίας δεδομένων (πχ λίστας, πλειάδας, λεξικού κτλ.).
- εμφωλευμένοι βρόγχοι, που ουσιαστικά πρόκειται για συνδυασμό βρόγχων while ή for. Έτσι μπορούμε να δομήσουμε ένα βρόγχο while μέσα σε έναν for, έναν for μέσα σε έναν while, έναν for μέσα σε έναν for και έναν while μέσα σε έναν while.

Παρακάτω δίνονται παραδείγματα για την κάθε κατηγορία.

Βρόγχος while

```
count = 0
while (count < 10):
    count = count + 1
    print("Hello World")
```

```
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
```

Βρόγχος for

```
fruits = ["Μήλο", "Κεράσι", "Αχλάδι"]
for x in fruits:
    print(x)
```

```
Μήλο
Κεράσι
Αχλάδι
```

εμφωλευμένοι βρόγχοι (for μέσα σε for)

```
for i in range(1, 11):
    # nested loop
    # to iterate from 1 to 10
    for j in range(1, 11):
        # print multiplication
        print(i * j, end=' ')
```

```
1 2 3 4 5
```

```
6 7 8 9 10 2 4 6 8 10 12 14 16 18 20 3 6 9 12 15 18 21 24 27 30 4 8 12 16 20 24 28
32 36 40 5 10 15 20 25 30 35 40 45 50 6 12 18 24 30 36 42 48 54 60 7 14 21 28 35 42
49 56 63 70 8 16 24 32 40 48 56 64 72 80 9 18 27 36 45 54 63 72 81 90 10 20 30 40
50 60 70 80 90 100
```

Οι βρόγχοι for πρέπει να εκτελούν κάποια πρόταση. Σε διαφορετική περίπτωση αν θέλουμε να τους χρησιμοποιήσουμε χωρίς να κάνουν τίποτα (πχ για να συγγράψουμε αργότερα το περιεχόμενό τους) μπορούμε να χρησιμοποιήσουμε την πρόταση **pass**.

```
for x in [0, 1, 2]:
    pass
```

Μια χρήσιμη συνάρτηση που χρησιμοποιείται σε συνδυασμό με τους βρόγχους for είναι η συνάρτηση range(). Η συνάρτηση range δημιουργεί μια ακολουθία αριθμών και προσφέρει σχετικές παραμέτρους για την έναρξη και λήξη και το βήμα της ακολουθίας δηλαδή range(start, stop, step_size).

```
range(0, 10)
```

```
range(0, 10)
```

Σε συνδυασμό με τον βρόγχο for μπορούμε να επαναλάβουμε μια σειρά προτάσεων κώδικα για μια ακολουθία αριθμών.

```
genre = ['Φυσικά', 'Μαθηματικά', 'Χημεία']

# iterate over the list using index
for i in range(len(genre)):
    print("Διαβάζω ", genre[i])
```

```
Διαβάζω Φυσικά
Διαβάζω Μαθηματικά
Διαβάζω Χημεία
```

εμφωλευμένοι βρόγχοι (while μέσα σε for)

```
names = ['Νίκος', 'Κώστας', 'Ελένη']
# outer loop
for name in names:
    # inner while loop
    count = 0
    while count < 5:
        print(name, end=' ')
        # increment counter
        count = count + 1
    print()
```

```
Νίκος Νίκος Νίκος Νίκος Νίκος
Κώστας Κώστας Κώστας Κώστας Κώστας
Ελένη Ελένη Ελένη Ελένη Ελένη
```

εμφωλευμένοι βρόγχοι (while μέσα σε while)

```
i = 1
while i <= 4 :
    j = 0
    while j <= 3 :
        print(i*j, end=" ")
        j += 1
    print()
    i += 1
```

```
0 1 2 3
0 2 4 6
0 3 6 9
0 4 8 12
```

Η πρόταση else μέσα σε βρόγχους

Όταν η συνθήκη βάσει της οποίας ελέγχεται η εκτέλεση ενός βρόγχου while αποτυγχάνει (δεν είναι αληθής) τότε μπορούμε να εκτελέσουμε μια άλλη σειρά προτάσεων που ορίζεται από την ενότητα else. Για παράδειγμα:

```
count=0
while(count<5):
    print(count)
    count +=1
else:
    print("count value reached %d" %(count))
```

```
0
1
2
3
4
count value reached 5
```

Αντίστοιχα σε έναν βρόγχο for μπορούμε να εκτελέσουμε μια ομάδα προτάσεων που ορίζεται από το else όταν έχει εξαντληθεί η προσπέλαση όλων των στοιχείων μιας ακολουθίας. Δείτε το παρακάτω παράδειγμα:

```
for x in range(6):
    print(x)
else:
    print("Finally finished!")
```

```
0
1
2
3
4
5
Finally finished!
```

Οι προτάσεις break και continue

Οι προτάσεις break και continue χρησιμοποιούνται για το έλεγχο της ροής των επαναλήψεων και λειτουργούν με τον ίδιο τρόπο τόσο στους βρόγχους while όσο και στους βρόγχους for.

Η πρόταση `break` τερματίζει άμεσα την εκτέλεση του βρόγχου με βάση τον έλεγχο μιας συνθήκης και στην συνέχεια ακολουθεί η εκτέλεση της πρότασης που ακολουθεί τον βρόγχο.

Παράδειγμα με βρόγχο `for`:

```
for num in range(1, 11):
    if num == 5:
        break
    else:
        print(num)

print("Τέλος")
```

```
1
2
3
4
Τέλος
```

Παράδειγμα με βρόγχο `while`:

```
n = 5
while n > 0:
    n -= 1
    if n == 2:
        break
    print(n)
print('Loop ended.')
```

```
4
3
Loop ended.
```

Αντίστοιχη είναι η λειτουργία της πρότασης `continue`. Ωστόσο σε αυτήν την περίπτωση ο βρόγχος δεν τερματίζεται εντελώς αλλά σταματάει την τρέχουσα σειρά εντολών στο τρέχον σημείο του βρόγχου και συνεχίζει στο επόμενο στοιχείο της ακολουθίας και στον έλεγχο της αντίστοιχης συνθήκης.

7. Ανάγνωση & εγγραφή αρχείων, μετονομασία, αναζήτηση, αντιγραφή μετακίνηση αρχείων και καταλογών

Πριν ξεκινήσουμε την επίδειξη των εντολών για το σημερινό μάθημα, θα πρέπει να εισάγουμε μερικές απαραίτητες βιβλιοθήκες και να ορίσουμε τον τρέχων κατάλογο στη Python:

```
import os
import pickle
from pathlib import Path
from datetime import datetime, timezone
import fnmatch
import tempfile
from tempfile import TemporaryFile, TemporaryDirectory
import shutil
```

```
print(os.getcwd()) # εκτύπωση τρέχοντος καταλόγου στην Python

path="./some_directory/"
os.chdir(path) # ορισμός τρέχοντος καταλόγου

print(os.getcwd()) # επιβεβαίωση του τρέχοντος καταλόγου στην Python
```

```
/home/leonidas/Documents/uth/Programming/JupyterNotebooks/notes/notebooks
/home/leonidas/Documents/uth/Programming/JupyterNotebooks/notes/notebooks/some_directory
```

Ανάγνωση & εγγραφή αρχείων

Μπορούμε να ανοίξουμε ένα αρχείο με την μέθοδο `open`, να δούμε μερικές ιδιότητες του και να το κλείσουμε

```

myfile=open("foo.txt", "r")
print ("Name of the file: ", myfile.name)
print ("Closed or not : ", myfile.closed)
print ("Opening mode : ", myfile.mode)
myfile.close()
print(myfile.closed)

```

```

Name of the file:  foo.txt
Closed or not :   False
Opening mode :    r
True

```

Με την μέθοδο write() και με παράμετρους το όνομα του αρχείου και την επιλογή προσπέλασης «w» (w= εγγραφή, r= ανάγνωση, a=προσθήκη) σε ένα ανοικτό αρχείο μπορούμε να γράψουμε σε αυτό περιεχόμενο. Πάντα πρέπει να κλείνουμε το ανοικτό αρχείο με την μέθοδο close().

```

# Open a file
fo = open("foo.txt", "w") # Σημαντική παράμετρος το "w"
fo.write( "Ένα ταξίδι χιλίων χιλιομέτρων αρχίζει με ένα βήμα.\nΛάο Τσε, 6ος αιώνας π.Χ., Κινέζος φιλόσοφος")
# Close opened file
fo.close()

```

Επιβεβαιώνουμε ότι όντως έγινε η εγγραφή διαβάζοντας το αρχείο με την μέθοδο read() η οποία διαβάζει όλο το περιεχόμενο με την μία από το αρχείο.

```

# Open a file
fo = open("foo.txt", "r") # Σημαντική παράμετρος το "w"

text = fo.read()
print (text)

# Close opened file
fo.close()

```

```

Ένα ταξίδι χιλίων χιλιομέτρων αρχίζει με ένα βήμα.
Λάο Τσε, 6ος αιώνας π.Χ., Κινέζος φιλόσοφος

```

Επειδή υπάρχει ο κίνδυνος να καλέσουμε την μέθοδο close() σε ένα ανοικτό αρχείο, μπορούμε εναλλακτικά να ανοίξουμε ένα αρχείο με το with. Σε αυτήν την περίπτωση το αρχείο κλείνει αυτόματα όταν ολοκληρωθεί το μπλοκ εντολών εντός του with.

```

with open('foo.txt', 'r') as reader:
    # Note: readlines doesn't trim the line endings
    print(reader.read())

```

```

Ένα ταξίδι χιλίων χιλιομέτρων αρχίζει με ένα βήμα.
Λάο Τσε, 6ος αιώνας π.Χ., Κινέζος φιλόσοφος

```

Η μέθοδος read() δέχεται όρισμα την θέση από την οποία ξεκινά η ανάγνωση του περιεχομένου κειμένου.

```

with open('foo.txt', 'r') as fo:
    text = fo.read(10)
    print (text)

    text = fo.read(10)
    print (text)

```

```

Ένα ταξίδι
χιλίων χι

```

Με την χρήση της μεθόδου readlines() προσαρτούμε κάθε νέα γραμμή στο αρχείο σε μια λίστα. Όμως ο χαρακτήρας που ορίζει την νέα σειρά (\n) δεν αγνοείται από την ανάγνωση.

```

with open('dog_breeds.txt', 'r') as reader:
    # Note: readlines doesn't trim the line endings
    dog_breeds = reader.readlines()
    dog_breeds = [line.rstrip() for line in dog_breeds] # μπορούμε να αφαιρέσουμε
τα new line characters με αυτόν τον τρόπο
    print(dog_breeds)

```

```
['Pug', 'Jack Russell Terrier', 'English Springer Spaniel', 'German Shepherd',  
'Staffordshire Bull Terrier', 'Cavalier King Charles Spaniel', 'Golden Retriever',  
'West Highland White Terrier', 'Boxer', 'Border Terrier', 'Beagle', 'Beagle',  
'Beagle', 'Beagle']
```

Με την χρήση του with μπορούμε να εγγράψουμε δεδομένα κιάλας. Σημαντική παράμετρος το “w” κατά το open() και η μέθοδος write().

```
with open('dog_breeds_reversed.txt', 'w') as writer:  
    # Alternatively you could use  
    # writer.writelines(reversed(dog_breeds))  
  
    # Write the dog breeds to the file in reversed order  
    for breed in reversed(dog_breeds):  
        writer.write(breed)
```

Με το όρισμα “a” κατά το άνοιγμα ενός αρχείου μπορούμε να εγγράψουμε σε ένα αρχείο χωρίς να διαγραφεί το προηγούμενο περιεχόμενο.

```
with open('dog_breeds.txt', 'a') as a_writer:  
    a_writer.write('Beagle\n')
```

```
with open('dog_breeds.txt', 'r') as reader:  
    print(reader.read())
```

```
Pug  
Jack Russell Terrier  
English Springer Spaniel  
German Shepherd  
Staffordshire Bull Terrier  
Cavalier King Charles Spaniel  
Golden Retriever  
West Highland White Terrier  
Boxer  
Border Terrier  
Beagle  
Beagle  
Beagle  
Beagle  
Beagle
```

Μπορούμε να χρησιμοποιήσουμε παράλληλα δύο αρχείο με τις αντίστοιχες παραμέτρους ανάγνωσης, ή εγγραφής ή προσθήκης μέσω του with ως εξής:

```
d_path = 'dog_breeds.txt'  
d_r_path = 'dog_breeds_reversed.txt'  
with open(d_path, 'r') as reader, open(d_r_path, 'w') as writer:  
    dog_breeds = reader.readlines()  
    writer.writelines(reversed(dog_breeds))
```

```
with open('dog_breeds.txt', 'r') as reader:  
    # Read and print the entire file line by line  
    line = reader.readline()  
    while line != '': # The EOF char is an empty string  
        print(line, end='')  
        line = reader.readline()
```

```
Pug  
Jack Russell Terrier  
English Springer Spaniel  
German Shepherd  
Staffordshire Bull Terrier  
Cavalier King Charles Spaniel  
Golden Retriever  
West Highland White Terrier  
Boxer  
Border Terrier  
Beagle  
Beagle  
Beagle  
Beagle  
Beagle
```

Μπορούμε να διαβάσουμε γραμμή - γραμμή το περιεχόμενο ενός αρχείου μέσω ενός loop στην λίστα γραμμών που μας προσφέρει η μέθοδος readlines()


```
with open('dog_breeds.txt', 'r') as reader:
    for line in reader.readlines():
        print(line, end='')
```

```
Pug
Jack Russell Terrier
English Springer Spaniel
German Shepherd
Staffordshire Bull Terrier
Cavalier King Charles Spaniel
Golden Retriever
West Highland White Terrier
Boxer
Border Terrier
Beagle
Beagle
Beagle
Beagle
Beagle
```

Και για περισσότερο ευανάγνωστο κώδικα η Python μας δίνει την δυνατότητα να κάνουμε loop μέσω του reader object

```
with open('dog_breeds.txt', 'r') as reader:
    # Read and print the entire file line by line
    for line in reader:
        print(line, end='') # use end='' to avoid new line after each print
                             statement
```

```
Pug
Jack Russell Terrier
English Springer Spaniel
German Shepherd
Staffordshire Bull Terrier
Cavalier King Charles Spaniel
Golden Retriever
West Highland White Terrier
Boxer
Border Terrier
Beagle
Beagle
Beagle
Beagle
Beagle
```

Σειριοποίηση (pickle)

Μέσω της σειριοποίησης μπορούμε να αποθηκεύσουμε σε ένα binary αρχείο τα αντικείμενα της python με τις ιδιότητές τους. Όχι μόνον σαν απλές συμβολοσειρές. Στο παρακάτω κομμάτι κώδικα θα αποθηκεύσουμε μία λίστα και μια μεταβλητή σε ένα αρχείο με την βοήθεια της βιβλιοθήκης pickle.

```
mylist=['one', 2, 'tree']
pi=3.14
with open('pickle.txt', 'ab') as pickle_writer:
    pickle.dump(mylist, pickle_writer)
    pickle.dump(pi, pickle_writer)
```

Αφού έχουμε αποθηκεύσει τα σχετικά object σε ένα αρχείο, μπορούμε σε μεταγενέστερα στάδια του κώδικα να τα ανακαλέσουμε αυτούσια μέσω της ανάγνωσης αυτού του αρχείου.

```
with open('pickle.txt', 'rb') as pickle_read:
    pickle.load(pickle_read)
    print(pi)
    print(mylist)
```

```
3.14
['one', 2, 'tree']
```

Ανάκτηση περιεχομένων φακέλου

Μέσω της μεθόδου os.scandir() μπορούμε να λάβουμε μια λίστα με τα αρχεία και τους φακέλους σε ένα κατάλογο.

```
entries = os.scandir('.')
for entry in entries:
    print(entry.name)
```

```
sub_dir2
sub_dir
dog_breeds_reversed.txt
sub_dir3
data_02.txt
foo.txt
data_02_backup.txt
data_03_backup.txt
data_01_backup.txt
dog_breeds.txt
2018
example_directory
pickle.txt
file1.py
data_03.txt
data_01.txt
tests.py
admin.py
admin2.py
```

```
# εναλλακτικά
os.listdir('.')
```

```
['sub_dir2',
'sub_dir',
'dog_breeds_reversed.txt',
'sub_dir3',
'data_02.txt',
'foo.txt',
'data_02_backup.txt',
'data_03_backup.txt',
'data_01_backup.txt',
'dog_breeds.txt',
'2018',
'example_directory',
'pickle.txt',
'file1.py',
'data_03.txt',
'data_01.txt',
'tests.py',
'admin.py',
'admin2.py']
```

```
# Εναλλακτικά
with os.scandir('.') as entries:
    for entry in entries:
        print(entry.name)
```

```
sub_dir2
sub_dir
dog_breeds_reversed.txt
sub_dir3
data_02.txt
foo.txt
data_02_backup.txt
data_03_backup.txt
data_01_backup.txt
dog_breeds.txt
2018
example_directory
pickle.txt
file1.py
data_03.txt
data_01.txt
tests.py
admin.py
admin2.py
```

Εναλλακτικά το ίδιο μπορούμε να κάνουμε με την μέθοδο `iterdir()` σε ένα Path object από την βιβλιοθήκη `pathlib`

```
entries = Path('.')
for entry in entries.iterdir():
    print(entry.name)
```

```
sub_dir2
sub_dir
dog_breeds_reversed.txt
sub_dir3
data_02.txt
foo.txt
data_02_backup.txt
data_03_backup.txt
data_01_backup.txt
dog_breeds.txt
2018
example_directory
pickle.txt
file1.py
data_03.txt
data_01.txt
tests.py
admin.py
admin2.py
```

Η μέθοδος `is_file()` μας επιτρέπει να τεστάρουμε αν ένα αντικείμενο τύπου `Path` είναι αρχείο.

```
basepath = Path('./')
files_in_basepath = basepath.iterdir()
for item in files_in_basepath:
    if item.is_file():
        print(item.name)
```

```
dog_breeds_reversed.txt
data_02.txt
foo.txt
data_02_backup.txt
data_03_backup.txt
data_01_backup.txt
dog_breeds.txt
pickle.txt
file1.py
data_03.txt
data_01.txt
tests.py
admin.py
admin2.py
```

Η μέθοδος `is_dir()` μας επιτρέπει να τεστάρουμε αν ένα αντικείμενο τύπου `Path` είναι φάκελος (directory).

```
# List all subdirectory using pathlib
basepath = Path('.')
for entry in basepath.iterdir():
    if entry.is_dir():
        print(entry.name)
```

```
sub_dir2
sub_dir
sub_dir3
2018
example_directory
```

Με την μέθοδο `stat()` μπορούμε να δούμε χρήσιμες λεπτομέρειες για ένα αρχείο και να ανακτήσουμε δεδομένα όπως το μέγεθος του, το όνομά του και η τελευταία ημερομηνία/ώρα τροποποίησης (δίνεται σε seconds απο την 1/1/1970)

```
current_dir = Path('./') #ορίστε έναν φάκελο. Στην συγκεκριμένη περίπτωση ορίζεται
ο τρέχον κατάλογος.
for path in current_dir.iterdir():
    info = path.stat()

    size=info.st_size
    modification_time=datetime.fromtimestamp(info.st_mtime, tz=timezone.utc) #
    st_time= the number of seconds passed since 1st January 1970 (epoch)
    name=path.name

    print("\nΌνομα αρχείου:", name)
    print("\tΜέγεθος: ",size) # το μέγεθος του αρχείου σε bytes
    print("\tΗμερομηνία τελευταίας τροποποίησης", modification_time)
```

```
Όνομα αρχείου: sub_dir2
Μέγεθος: 4096
Ημερομηνία τελευταίας τροποποίησης 2022-05-13 15:51:20+00:00

Όνομα αρχείου: sub_dir
Μέγεθος: 4096
Ημερομηνία τελευταίας τροποποίησης 2022-05-13 15:51:20+00:00

Όνομα αρχείου: dog_breeds_reversed.txt
Μέγεθος: 224
Ημερομηνία τελευταίας τροποποίησης 2022-05-13 18:29:57.968405+00:00

Όνομα αρχείου: sub_dir3
Μέγεθος: 4096
Ημερομηνία τελευταίας τροποποίησης 2022-05-13 15:51:20+00:00

Όνομα αρχείου: data_02.txt
Μέγεθος: 0
Ημερομηνία τελευταίας τροποποίησης 2022-05-13 07:01:42+00:00

Όνομα αρχείου: foo.txt
Μέγεθος: 168
Ημερομηνία τελευταίας τροποποίησης 2022-05-13 18:29:57.884406+00:00

Όνομα αρχείου: data_02_backup.txt
Μέγεθος: 0
Ημερομηνία τελευταίας τροποποίησης 2022-05-13 07:01:42+00:00

Όνομα αρχείου: data_03_backup.txt
Μέγεθος: 0
Ημερομηνία τελευταίας τροποποίησης 2022-05-13 07:01:42+00:00

Όνομα αρχείου: data_01_backup.txt
Μέγεθος: 0
Ημερομηνία τελευταίας τροποποίησης 2022-05-13 07:01:42+00:00

Όνομα αρχείου: dog_breeds.txt
Μέγεθος: 224
Ημερομηνία τελευταίας τροποποίησης 2022-05-13 18:29:57.948406+00:00

Όνομα αρχείου: 2018
Μέγεθος: 4096
Ημερομηνία τελευταίας τροποποίησης 2022-05-13 18:28:15.820793+00:00

Όνομα αρχείου: example_directory
Μέγεθος: 4096
Ημερομηνία τελευταίας τροποποίησης 2022-05-13 18:28:15.800793+00:00

Όνομα αρχείου: pickle.txt
Μέγεθος: 364
Ημερομηνία τελευταίας τροποποίησης 2022-05-13 18:29:58.020405+00:00

Όνομα αρχείου: file1.py
Μέγεθος: 0
Ημερομηνία τελευταίας τροποποίησης 2022-05-13 07:01:42+00:00

Όνομα αρχείου: data_03.txt
Μέγεθος: 0
Ημερομηνία τελευταίας τροποποίησης 2022-05-13 07:01:42+00:00

Όνομα αρχείου: data_01.txt
Μέγεθος: 0
Ημερομηνία τελευταίας τροποποίησης 2022-05-13 07:01:42+00:00

Όνομα αρχείου: tests.py
Μέγεθος: 0
Ημερομηνία τελευταίας τροποποίησης 2022-05-13 07:01:42+00:00

Όνομα αρχείου: admin.py
Μέγεθος: 0
Ημερομηνία τελευταίας τροποποίησης 2022-05-13 07:01:42+00:00

Όνομα αρχείου: admin2.py
Μέγεθος: 0
Ημερομηνία τελευταίας τροποποίησης 2022-05-13 18:29:22.292540+00:00
```

Μέσω του Path μπορούμε να φτιάξουμε και διαδρομές προς ένα κατάλογο του δίσκου μας

```
# build paths

in_file_1 = Path.cwd() / "in" / "input.xlsx"
out_file_1 = Path.cwd() / "out" / "output.xlsx"

print(in_file_1)
print(out_file_1)

# or

in_file_2 = Path.cwd().joinpath("in").joinpath("input.xlsx")
out_file_2 = Path.cwd().joinpath("out").joinpath("output.xlsx")
```

```
/home/leonidas/Documents/uth/Programming/JupyterNotebooks/notes/notebooks/some_directory/in/input.xlsx
/home/leonidas/Documents/uth/Programming/JupyterNotebooks/notes/notebooks/some_directory/out/output.xlsx
```

Δημιουργία καταλόγων

Επιβεβαιώνουμε για μια ακόμη φορά τον τρέχοντα κατάλογο

```
path = Path.cwd() # pathlib object, εναλλακτικό του os.getcwd()

print(str(path)) # print σαν συμβολοσειρά
```

```
/home/leonidas/Documents/uth/Programming/JupyterNotebooks/notes/notebooks/some_directory
```

Μπορούμε να δημιουργήσουμε καταλόγους με την μέθοδο mkdir()

```
# create directory
try:
    p = Path('example_directory/') # ορισμός absolute ή relative Path
    p.mkdir() # δημιουργία καταλόγου
except FileExistsError:
    print(f"Ο κατάλογος {str(p)} υπάρχει ήδη")
```

```
Ο κατάλογος example_directory υπάρχει ήδη
```

```
p.mkdir(exist_ok=True) # δημιουργία καταλόγου, αγνοεί την δημιουργία φακέλου αν αυτός υπάρχει ήδη
```

```
p = Path('2018/10/05') # δημιουργία καταλόγου 05 και όλων των γονικών (parent) καταλόγων
p.mkdir(parents=True, exist_ok=True)
```

Είτε να αναζητήσουμε αρχεία και καταλόγους που περιλαμβάνου συγκεκριμένους χαρακτήρες στο ονομά τους

```
content = os.listdir('.')
print(content)

print("Εύρεση αρχείων με καταληξη txt\n")

for file_name in content:
    if fnmatch.fnmatch(file_name, '*.txt'):
        print(file_name)
```

```
['sub_dir2', 'sub_dir', 'dog_breeds_reversed.txt', 'sub_dir3', 'data_02.txt',
'foo.txt', 'data_02_backup.txt', 'data_03_backup.txt', 'data_01_backup.txt',
'dog_breeds.txt', '2018', 'example_directory', 'pickle.txt', 'file1.py',
'data_03.txt', 'data_01.txt', 'tests.py', 'admin.py', 'admin2.py']
Εύρεση αρχείων με καταληξη txt

dog_breeds_reversed.txt
data_02.txt
foo.txt
data_02_backup.txt
data_03_backup.txt
data_01_backup.txt
dog_breeds.txt
pickle.txt
data_03.txt
data_01.txt
```

Εύρεση αρχείων που περιλαμβάνουν το όνομά τους έχει την παρακάτω μορφή:

```
data_*_backup.txt
```

το αστεράκι (*) αντιπροσωπεύει οποιοδήποτε αριθμό χαρακτήρων μέσα στο όνομα.

```
for filename in os.listdir('.'):
    if fnmatch.fnmatch(filename, 'data_*_backup.txt'):
        print(filename)
```

```
data_02_backup.txt
data_03_backup.txt
data_01_backup.txt
```

Εναλλακτικά με την χρήση της μεθόδου glob()

```
p = Path('.')
for name in p.glob('data_*_backup.txt'):
    print(name)
```

```
data_02_backup.txt
data_03_backup.txt
data_01_backup.txt
```

```
p = Path('.')
for name in p.glob('[0-9]*backup.txt'):
    print(name)
```

```
data_02_backup.txt
data_03_backup.txt
data_01_backup.txt
```

Οι παραπάνω αναζητήσεις αφορούσαν το περιεχόμενο μόνο στον τρέχοντα κατάλογο και όχι ταυτόχρονα και στους υποκείμενους καταλόγους (child) που υπάρχουν μέσα σε αυτόν. Για να αναζητήσουμε διαδοχικά και σε αυτούς τους καταλόγους χρησιμοποιούμε το παρακάτω πρόθεμα πριν από το κριτήριο αναζήτησης **/

```
# Append "**/" before the search term in pattern to recursively search this
directory
p = Path('.')
for name in p.glob('**/*.py'):
    print(name)
```

```
file1.py
tests.py
admin.py
admin2.py
sub_dir2/file2.py
sub_dir2/file1.py
sub_dir/file2.py
sub_dir/file1.py
sub_dir3/file2.py
sub_dir3/file1.py
```

```
# αναζήτηση για ότι περιέχει f και 1 στο filename
```

```
p = Path('.')  
for name in p.glob('**/f*1*'):  
    print(name)
```

```
file1.py  
sub_dir2/file1.py  
sub_dir/file1.py  
sub_dir3/file1.py
```

Για να ανατρέξουμε διαδοχικά σε όλους του φακέλους ενός καταλόγου χρησιμοποιούμε την μέθοδο os.walk

```
# walk  
# Walking a directory tree and printing the names of the directories and files  
for dirpath, dirnames, files in os.walk('.', topdown=False):  
    print(f'Found directory: {dirpath}')  
    for file_name in files:  
        print(file_name)
```

```
Found directory: ./sub_dir2  
file2.py  
file1.py  
Found directory: ./sub_dir  
file2.py  
file1.py  
Found directory: ./sub_dir3  
file2.py  
file1.py  
Found directory: ./2018/10/05  
Found directory: ./2018/10  
Found directory: ./2018  
Found directory: ./example_directory  
Found directory: .  
dog_breeds_reversed.txt  
data_02.txt  
foo.txt  
data_02_backup.txt  
data_03_backup.txt  
data_01_backup.txt  
dog_breeds.txt  
pickle.txt  
file1.py  
data_03.txt  
data_01.txt  
tests.py  
admin.py  
admin2.py
```

Με την python μπορούμε να δημιουργήσουμε προσωρινά αρχεία και καταλόγους οι οποίου παύουν να υπάρχουν μετά την εκτέλεση του κώδικα. Αυτό γίνεται με την βοήθεια του αρθρώματος (module) tempfile και της συνάρτησης TemporaryFile() και TemporaryDirectory()

```
from tempfile import TemporaryFile  
  
# temporary files  
with TemporaryFile('w+t') as fp:  
    fp.write('Hello universe!')  
    fp.seek(0)  
    print(fp.read())  
# File is now closed and removed
```

```
Hello universe!
```

```
#fp.seek(0)  
#print(fp.read())
```

```
with TemporaryDirectory() as tmpdir:  
    print('Created temporary directory ', tmpdir)  
    os.path.exists(tmpdir)
```

```
# Directory contents have been removed
```

```
Created temporary directory /tmp/tmpovfbsdpx
```

```
tmpdir
```

```
 '/tmp/tmpovfbsdpx'
```

```
os.path.exists(tmpdir)
```

```
False
```

Διαγραφή αρχείων και φακέλων

Με την μέθοδο `unlink()` μπορούμε να διαγράψουμε έναν **άδειο** κατάλογο ή ένα αρχείο.

```
data_file = Path('./data_04.txt')

if data_file.is_file():
    print ("Το αρχείο υπάρχει και θα διαγραφεί")
    data_file.unlink()
else:
    print ("Το αρχείο δεν υπάρχει")
```

```
Το αρχείο δεν υπάρχει
```

```
# διαγραφή άδειου φακέλου
my_dir = Path('./tmp')

if my_dir.is_dir():
    print ("Το directory υπάρχει και θα διαγραφεί")
    my_dir.rmdir()
else:
    print ("Το directory δεν υπάρχει")
```

```
Το directory δεν υπάρχει
```

Αν θέλουμε να διαγράψουμε έναν κατάλογο με περιεχόμενα τότε χρησιμοποιούμε η συνάρτηση `rmtree()` από την βιβλιοθήκη `shutil`.

```
# διαγραφή φακέλου με περιεχόμενα

dest = Path('./tmp2')
shutil.rmtree(dest, ignore_errors=True)
```

Αντιγραφή αρχείων και φακέλων

Ταυτόχρονα μπορούμε να αντιγράψουμε αρχεία με την συνάρτηση `copy()` πάλι από το άρθρωμα `shutil`.

```
# Αντιγραφή αρχείου

src = 'admin.py'
dst = 'admin2.py'
shutil.copy(src, dst)
```

```
'admin2.py'
```

ή ολόκληρους καταλόγους μέσω της συνάρτησης `copytree()` πάλι από το ίδιο άρθρωμα.

```
shutil.copytree('sub_dir', 'sub_dir3', dirs_exist_ok=True)
```

```
'sub_dir3'
```

Μετακίνηση

ή ακόμα και να μετακινήσουμε αρχεία και καταλόγους με την συναρτηση `move()`


```
try:
    shutil.move('data_04.txt', 'sub_dir/data_04.txt')
except FileNotFoundError:
    print("File does not exist")
```

File does not exist

```
# μετακίνηση αρχείου και μάλιστα με μετονομασία κατά την μετακίνηση data_04.txt -> data_05.txt

try:
    shutil.move('sub_dir/data_04.txt', 'data_05.txt' )
except FileNotFoundError:
    print("File does not exist")
```

File does not exist

```
# μετακίνηση φακέλου

try:
    shutil.move('tmp2', 'tmp/tmp2')
except FileNotFoundError:
    print("Directory does not exist")
```

Directory does not exist

```
# επιστροφή στην θέση του

try:
    shutil.move('tmp/tmp2', 'tmp2')
except FileNotFoundError:
    print("Directory does not exist")
```

Directory does not exist

Μετονομασία

Με την χρήση της μεθόδου `rename()` σε έναν `Path` αντικείμενο μπορούμε να το μετονομάσουμε.

```
# αρχείου
data_file = Path('data_01.txt')
data_file.rename('data.txt')
```

PosixPath('data.txt')

```
# ξανά όπως ήταν
data_file = Path('data.txt')
data_file.rename('data_01.txt')
```

PosixPath('data_01.txt')

Βιβλιογραφία

- Αγγελιδάκης, Ν., 2015. Εισαγωγή στον προγραμματισμό με την Python. Αγγελιδάκης, Ηράκλειο.
- Working With Files in Python, <https://realpython.com/read-write-files-python/>, Πρόσβαση: 13/05/2022
- Reading and Writing Files in Python, <https://realpython.com/read-write-files-python>, Πρόσβαση: 13/05/2022

Αξιολόγηση

Η αξιολόγηση γίνεται:

- 50% από ατομικές ασκήσεις στην διάρκεια του εξαμήνου.
- 50% από τελικές εξετάσεις.

