Κεφάλαιο

3 Λογικές συνθήκες και δομές ελέγχου

Περίγραμμα

Στόχοι

Το κεφάλαιο αυτό στοχεύει στην:

- Εξήγηση του τρόπου σχηματισμού λογκών εκφράσεων.
- Εξήγηση της έννοιας και της χρήσης των λογκών τελεστών.
- Εξήγηση του τρόπου σχηματισμού ομάδων εντολών στην Python.
- Εξήγηση των δομών επιλογής στην Python.
- Εξήγηση των δομών βρόχου και επαναληπτικής εκτέλεσης εντολών στην Python.
- Εξήγηση των συμπεριλήψεων σε λίστες, σύνολα, λεξικά.
- Εξήγηση των μηχανισμών, επαναληπτών και γεννητόρων της Python.

3.1 Λογικές εκφράσεις

Ένα στοιχείο στον προγραμματισμό το οποίο μπορεί να θεωρηθεί ως θεμελιώδες, είναι οι ***λογικές εκφράσεις*** (*Boolean expressions*). Μέσω της χρήσης τους, σχηματίζονται λογικές συνθήκες, με βάση τις οποίες ειδικές εντολές ελέγχουν τη διαδοχή εκτέλεσης των υπόλοιπων εντολών· αν θα εκτελεστούν κάποιες εντολές, ενώ κάποιες άλλες όχι, ή αν κάποιες άλλες θα εκτελεστούν επαναληπτικά, υλοποιώντας κατ' αυτόν τον τρόπο τη λογική δομή κάθε προγράμματος.

Οι λογικές εκφράσεις, υπολογίζονται πάντα σε μία από τις δύο λογικές τιμές, `True` - `False` (*αλήθεια* - *ψεύδος*, με τα `T` και `F` κεφαλαία, βλ. 2.3.1), οι οποίες, όπως ήδη έχει αναφερθεί, στην Python είναι τύπου `bool`.

3.1.1 Σχεσιακοί τελεστές

Για το σχηματισμό απλών λογικών εκφράσεων, χρησιμοποιούνται ειδικοί τελεστές γνωστοί ως ***σχεσιακοί τελεστές*** (*relational operators*). Στο παράδειγμα που ακολουθεί, χρησιμοποιείται ο σχεσιακός τελεστής ==, ο οποίος υπολογίζεται στην τιμή `True` αν οι τελεστέοι είναι ίσοι, διαφορετικά υπολογίζεται σε `False`:

```
>>> 10 == 10
True
>>> 10 == 200
False
```

Γενικότερα, οι σχεσιακοί τελεστές χρησιμοποιούνται όταν χρειάζεται να γίνει σύγκριση τιμών δύο ή περισσότερων παραστάσεων μεταξύ τους. Οι παραστάσεις αυτές μπορεί να είναι αριθμητικές παραστάσεις, μεταβλητές, σταθερές (αριθμοί, συμβολοσειρές, λίστες, σύνολα κ.λπ.), ή ακόμα και αντικείμενα. Μία σύγκριση υφίσταται, συντακτικά μέσα σε ένα πρόγραμμα, ως μία λογική έκφραση (*συνθήκη*).

Πίνακας 3.1: Σχεσιακοί τελεστές στην Python.

Τελεστής	Περιγραφή
==	Αν οι τελεστέοι είναι ίσοι, τότε η συνθήκη αληθεύει.
!=	Αν οι τελεστέοι *δεν* είναι ίσοι, τότε η συνθήκη αληθεύει.
>	Αν ο τελεστέος αριστερά είναι μεγαλύτερος από τον τελεστέο δεξιά, τότε η συνθήκη αληθεύει.
>=	Αν ο τελεστέος αριστερά είναι μεγαλύτερος ή ίσος από τον τελεστέο δεξιά, τότε η συνθήκη αληθεύει.
<	Αν ο τελεστέος αριστερά είναι μικρότερος από τον τελεστέο δεξιά, τότε η συνθήκη αληθεύει.
<=	Αν ο τελεστέος αριστερά είναι μικρότερος ή ίσος από τον τελεστέο δεξιά, τότε η συνθήκη αληθεύει

Οι δύο πρώτοι τελεστές του πίνακα 3.1, ονομάζονται και ***τελεστές ελέγχου ισότητας***, γιατί απλά ελέγχουν την ισότητα ή μη των δύο τελεστέων. Οι υπόλοιποι λέγονται και ***τελεστές σύγκρισης*** (μεγέθους), γιατί ελέγχουν το ποιος από τους δύο τελεστέους είναι μεγαλύτερος.

Μπορούμε να χρησιμοποιήσουμε τους τελεστές ελέγχου ισότητας μεταξύ οποιουδήποτε τύπου δεδομένων, τους υπόλοιπους, όμως, μόνο μεταξύ ιδίων τύπων ή μεταξύ συγκεκριμένων αριθμητικών τύπων όπως περιγράφεται παρακάτω.

3.1.2 Συγκρίσεις διαφόρων τύπων δεδομένων

Για να μπορέσει ο προγραμματιστής να χρησιμοποιήσει σωστά και με αποτελεσματικό τρόπο τους τελεστές σύγκρισης, πρέπει πρώτα να γίνει κατανοητή η λογική σύγκρισης των διαφόρων τύπων δεδομένων.

Αριθμοί

Όλοι οι σχεσιακοί τελεστές μπορούν να χρησιμοποιηθούν μεταξύ αριθμών, όπως ακριβώς και στα μαθηματικά, εκτός από τους μιγαδικούς (*complex*) όπου δεν νοείται η απ' ευθείας σύγκριση μεγέθους, οπότε είναι δυνατή μόνο η χρήση των τελεστών ελέγχου ισότητας. Για παράδειγμα,

```
>>> 50 == 50
True
>>> 42 == 14
False
>>> 4 != 5
True
>>> 4 != 4
False
>>> 52 < 60
True
>>> 42 > 120
False
>>> 20 < 20
False
>>> duzina = 12
>>> duzina <= 12
True
>>> hlikia = 39
>>> hlikia >= 10
True
```

Όπως είναι αναμενόμενο, τιμές που ελέγχονται με τον τελεστή `==` υπολογίζονται σε `True` όταν οι συγκρινόμενες αριθμητικές τιμές είναι ίσες, ενώ με χρήση του τελεστή `!=` (*διάφορο του*), υπολογίζονται σε `True` όταν διαφέρουν.

Στην Python, μπορούν να γίνουν και πολλαπλές - χωρισμένες με κόμμα (`,`) - ανεξάρτητες συγκρίσεις αριθμών σε μία γραμμή. Τα επιμέρους αποτελέσματα επιστρέφονται σε μορφή πλειάδας (βλ. 4.7):

```
>>> i = 3
>>> k = 8
>>> i > k, i == k, i != k
(False, False, True)
```

Προσοχή χρειάζεται όταν γίνεται σύγκριση δύο αριθμών, από τους οποίους ο ένας τουλάχιστον είναι κινητής υποδιαστολής (*float*), κυρίως όταν αυτοί είναι αποτέλεσμα κάποιας παράστασης. Λόγω της -εγγενούς από τη φύση των υπολογιστών- περιορισμένης σημαντικότητας δεκαδικών ψηφίων, μπορεί δύο αριθμοί να φαίνονται μαθηματικά ίσοι αλλά ο διερμηνευτής να μην τους θεωρεί ίσους ή το αντίθετο:

```
>>> 1 > 0.9999999999999999
True
>>> 1 > 0.99999999999999999
False
```

Στο πρώτο παράδειγμα, η παράσταση είναι προφανώς αληθής. Στο δεύτερο παράδειγμα, όμως, παρ' όλο που, σύμφωνα με τα μαθηματικά, η παράσταση θα έπρεπε να είναι αληθής, ο διερμηνευτής τη θεωρεί ψευδή. Αυτό συνέβη γιατί τα σημαντικά δεκαδικά ψηφία σε αριθμούς τύπου `float` είναι 16. Έτσι, ενώ στο πρώτο παράδειγμα υπάρχουν 16 ψηφία μετά την υποδιαστολή, στο 2ο παράδειγμα προστέθηκε και 17ο ψηφίο, κι έτσι ο διερμηνευτής έκανε προς τα άνω στρογγυλοποίηση (εδώ συγκεκριμένα προς τη μονάδα). Το αποτέλεσμα είναι ότι η συνθήκη είναι ψευδής, αφού το 1 δεν είναι μεγαλύτερο από το 1. Όμως `1 >= 1`, άρα το παρακάτω παράδειγμα καταλήγει σε τιμή `True`:

```
>>> 1 >= 0.99999999999999999
True
```

Τέλος, επειδή ο τύπος δεδομένων `bool` είναι πρακτικά ένα υποσύνολο του τύπου `int` με πεδίο ορισμού τους ακεραίους 0 και 1, (`False` → `0`, `True` → `1`), έχουν νόημα συγκρίσεις μεταξύ αριθμών και `True` ή `False`:

```
>>> 1 == True
True
>>> 1 > False
True
>>> 0 > True
False
>>> True <= 1
True
>>> True == True
True
>>> True != False
True
```

Λίστες

Δύο λίστες είναι ίσες όταν έχουν το ίδιο πλήθος στοιχείων, και αυτά είναι ένα προς ένα ίσα μεταξύ τους:

```
>>> a_list = [1, 2, 3]
>>> a_list == [1, 2, 3]
True
>>> [4, 5, 6] == [4.0, 5.0, 5.99999999999999999999]
True
>>> [4, 5, 6] == [4.0, 5.0, 5.99999]
False
>>> a_complex_list = [1, 'alpha', [1, 2]]
>>> a_complex_list == [1, 'alpha', [1, 2]]
True
>>> a_complex_list == [1, 'alpha', [2, 1]]
False
```

Η διαδικασία σύγκρισης δύο λιστών είναι η εξής: ο διερμηνευτής συγκρίνει ένα προς ένα τα στοιχεία των λιστών που βρίσκονται στις ίδιες θέσεις, αντιστοίχως, με τη σειρά, από το πρώτο προς το τελευταίο (δηλ. το πρώτο της πρώτης λίστας με το πρώτο της δεύτερης, μετά το δεύτερο της πρώτης με το δεύτερο της δεύτερης κ.ο.κ.). Όσο τα αντίστοιχα στοιχεία είναι ίσα μεταξύ τους, η σύγκριση συνεχίζεται στα επόμενα. Αν εξαντληθούν τα στοιχεία και στις δύο λίστες ταυτόχρονα, οι λίστες θεωρούνται ίσες. Αν εξαντληθούν μόνο στη μία, η σύγκριση τερματίζεται και αυτή η λίστα θεωρείται μικρότερη (οπότε θα αληθεύουν οι σχετικές συνθήκες). Αν σε κάποιο ζευγάρι αντίστοιχων στοιχείων δεν υπάρχει ισότητα, τότε εξετάζεται ο τύπος δεδομένων τους.

Αν είναι ίδιος ή συμβατός (π.χ. `int` και `float`), τότε η σύγκριση τερματίζεται και το αποτέλεσμα της σύγκρισης των λιστών ορίζεται ως το αποτέλεσμα της σύγκρισης των δύο στοιχείων.

Αλλιώς, στην περίπτωση των τελεστών ελέγχου ισότητας οι λίστες θεωρούνται μη ίσες, με το ανάλογο αποτέλεσμα στην σύγκριση. Στην περίπτωση των τελεστών σύγκρισης μεγέθους δεν επιτρέπεται σύγκριση ασύμβατων (π.χ. `int` και `complex`) ή διαφορετικών τύπων (π.χ. `int` και `str`), και θα εγερθεί εξαίρεση - μήνυμα σφάλματος τύπου *`TypeError`*.

Στο επόμενο παράδειγμα, ο διερμηνευτής συγκρίνει το 1 με το 1 που είναι ίσα, επομένως συνεχίζει στο επόμενο ζευγάρι στοιχείων και εξετάζει αν το 2 είναι μικρότερο από το 'a'. Μία τέτοιου είδους σύγκριση δεν επιτρέπεται, γι' αυτό εγείρεται εξαίρεση - μήνυμα σφάλματος τύπου *`TypeError`*.

```
>>> [1, 2] < [1, 'a']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: int() < str()
```

Στο παράδειγμα που ακολουθεί, παρ' όλο που η πρώτη λίστα έχει τρία στοιχεία και η δεύτερη εννιά, η πρώτη θεωρείται μεγαλύτερη γιατί το τρίτο στοιχείο της (3) είναι μεγαλύτερο από το αντίστοιχο (1) της δεύτερης λίστας.

```
>>> [1, 2, 3] > [1, 2, 1, 5, 6, 8, 2, 3, 5]
True
```

Πλειάδες

Για τις πλειάδες ισχύει ακριβώς ό,τι ισχύει για τις λίστες.

Συμβολοσειρές

Όσον αφορά τους λογικούς τελεστές, η Python χειρίζεται τις συμβολοσειρές ως ακολουθίες χαρακτήρων, επομένως και εδώ ισχύουν οι ίδιοι κανόνες. Αυτό όμως που πρέπει να τονιστεί είναι ότι η σύγκριση των στοιχείων των συμβολοσειρών γίνεται λεξικογραφικά, με βάση τη θέση κάθε χαρακτήρα (*κωδικοσημείο - code point*) της συμβολοσειράς στον κώδικα Unicode (βλ. κεφ.12).

```
>>> 'Alpha' > 'Beta'
False
```

Εδώ η συμβολοσειρά `'Alpha'` δεν είναι μεγαλύτερη από την `'Beta'` γιατί ο χαρακτήρας `'A'` είναι πριν τον `'B'` στον κώδικα Unicode.

```
>>> 'alpha' > 'Beta'
True
```

Στο τελευταίο παράδειγμα, όμως, ο χαρακτήρας `'a'` είναι μετά από τον `'B'` γι' αυτό το λόγο η συμβολοσειρά `'alpha'` εκλαμβάνεται ως μεγαλύτερη της `'beta'`. Με την ίδια λογική:

```
>>> 'hello' == 'hello'
True
>>> 'hello' == 'Hello'
False
>>> 'Σκύλος' != 'Γάτα'
True
>>> 12 == '12'
False
```

Ο έλεγχος ισότητας μιας αριθμητικής τιμής με μια συμβολοσειρά, υπολογίζεται σε `False` (12 διάφορο του `'12'`).

Φυσική συνέπεια των ανωτέρω είναι να λαμβάνεται υπόψη η διάκριση πεζών - κεφαλαίων κατά τις συγκρίσεις συμβολοσειρών (*είναι case sensitive*):

```
>>> car = 'Avensis'
>>> car == 'avensis'
False
```

Στις περιπτώσεις που η διάκριση πεζών - κεφαλαίων δεν είναι επιθυμητή, μπορεί να γίνει μετατροπή σε πεζά πριν τη σύγκριση:

```
>>> "Άλλος" == "άλλος"
False
>>> "Άλλος".lower() == "άλλος"
True
>>> car = 'Avensis'
>>> car.lower() == 'avensis'
True
```

```
>>> car
'Avensis'
```

Στο αμέσως προηγούμενο παράδειγμα, μέσω της μεθόδου `lower()`, επιστρέφεται το περιεχόμενο της συμβολοσειράς σε πεζά, οπότε ο έλεγχος υπολογίζεται σε `True`. Ας σημειωθεί ότι το περιεχόμενο της μεταβλητής δεν επηρεάζεται, απλά επιστρέφεται η συμβολοσειρά με όλους τους χαρακτήρες πεζούς.

Σύνολα

Δύο σύνολα είναι ίσα όταν κανένα δεν έχει περισσότερα, λιγότερα ή διαφορετικά στοιχεία από το άλλο. Αλλιώς θεωρούνται μη ίσα. Η σύγκριση μεγέθους (<, <=, >, >=) αντικατοπτρίζει την έννοια του υποσυνόλου στα μαθηματικά. Μεταξύ δύο συνόλων A και B ισχύει A < B όταν A γνήσιο υποσύνολο του B, A <= B όταν A υποσύνολο του B και αντιστρόφως. Επομένως είναι πιθανό σε μία σύγκριση δύο συνόλων κανένας τελεστής σύγκρισης μεγέθους να μη δώσει αποτέλεσμα `True`.

```
>>> {1, 2, 3} == {1, 2, 3}
True
>>> {'a', 'c', 'd'} != {'a', 'c', 'd'}
False
>>> {10, 20, 30} < {10, 20, 30, 50}
True
>>> {10, 20, 30} <= {10, 20, 30}
True
>>> {1, 2, 3} < {1, 2, 4}
False
>>> {1, 2, 3} <= {1, 2, 4}
False
>>> {1, 2, 3} > {1, 2, 4}
False
>>> {1, 2, 3} >= {1, 2, 4}
False
```

Λεξικά

Στα λεξικά επιτρέπονται μόνο οι τελεστές ελέγχου ισότητας. Δύο λεξικά είναι ίσα όταν έχουν το ίδιο πλήθος κλειδιών/τιμών, τα ίδια ακριβώς κλειδιά, και οι αντίστοιχες τιμές τους είναι και αυτές ίσες, ανεξάρτητα από τη σειρά που εμφανίζονται μέσα στο λεξικό.

```
>>> {'red' : 1, 'green' : 2, 'yellow' : 3} == {'red' : 1, 'green' :
2, 'yellow' : 4}
False
>>> {'red' : 1, 'green' : 2, 'yellow' : 3} == {'black' : 1, 'green'
: 2, 'yellow' : 3}
False
>>> {'red' : 1, 'green' : 2, 'yellow' : 3} == {'red' : 1, 'green' :
2, 'yellow' : 4}
True
```

Π3.1 Άλλες τιμές ως True ή False

Στην Python, εκτός από τη λογική ρητή τιμή `False`, οποιοδήποτε από τα ακόλουθα, λογίζεται επίσης ως `Fasle`:

Η τιμή None.

Η τιμή μηδέν (`0`) του ακέραιου τύπου δεδομένων καθώς και η `0.0` του τύπου κινητής υποδιαστολής.

Η κενή συμβολοσειρά: ‘ ’ ή ””.

Η κενή λίστα: `[]`.

Η κενή πλειάδα: `()`

Το κενό λεξικό: `{}`

Το κενό σύνολο: `set(())`

Το κενό παγωμένο σύνολο: `frozenset(())`

Οτιδήποτε άλλο, για παράδειγμα αριθμοί, όπως όλοι οι ακέραιοι -πλην του μηδενός- ή μη κενές λίστες, πλειάδες κ.λπ. λογίζονται ως `True`.

3.1.3 Έλεγχος ταυτότητας

Επειδή στην Python οι μεταβλητές είναι αναφορές σε αντικείμενα, μερικές φορές προκύπτει η ανάγκη να ελεγχθεί αν δύο μεταβλητές αναφέρονται στο ίδιο αντικείμενο. Προς το σκοπό αυτό, η Python περιλαμβάνει τον τελεστή `is` και την άρνησή του `is not`, μέσω των οποίων μπορεί να πραγματοποιηθούν έλεγχοι του τύπου αυτού:

```
>>> i = 3
>>> f = 3.0
>>> i == f
True
>>> i is f
False
>>> i not is f
True
>>> f = i
>>> i is f
True
```

Στην πράξη, οι εν λόγω τελεστές δεν χρησιμοποιούνται για τη σύγκριση τιμών, αλλά μόνο στις περιπτώσεις που είναι επιθυμητό να ελεγχθεί αν δύο αναφορές σε αντικείμενα δείχνουν στο ίδιο αντικείμενο ή αν ένα αντικείμενο είναι None. Για τη σύγκριση τιμών χρησιμοποιούνται οι σχεσιακοί τελεστές.

3.1.4 Έλεγχος μέλους

Για τους τύπους δεδομένων που εντάσσονται στις συλλογές (συμβολοσειρές, λίστες, πλειάδες, σύνολα, λεξικά), μπορεί κανείς να εκτελέσει έλεγχο μέλους (*membership*), με χρήση του τελεστή `in`, και το αντίστροφο, με τον τελεστή `not in`. Για παράδειγμα:

```
>>> a_lst = ["Σκύλος", "Γάτα", "Βάτραχος"]
>>> "Γάτα" in a_lst
True
>>> "Ποντικός" in a_lst
False
```

Ας σημειωθεί ότι, λόγω του τρόπου εσωτερικής υλοποίησης, οι τελεστές ελέγχου μέλους, μπορεί να αποδειχθούν λιγότερο αποτελεσματικοί (είναι σχετικά αργοί) στην περίπτωση που χρησιμοποιούνται σε λίστες ή πλειάδες μεγάλου μεγέθους. Αντίθετα, ελέγχουν ταχύτατα όταν χρησιμοποιούνται σε σύνολα και λεξικά. Επίσης, στην περίπτωση των συμβολοσειρών, μπορούν να χρησιμοποιηθούν και για έλεγχο υποσυμβολοσειρών οποιουδήποτε μεγέθους:

```
>>> str = "Σκουληκομυρμηγκότρυπα"
>>> "λ" in str
True
>>> "ά" in str
False
>>> "α" in str
True
>>> "κου" in str
True
```

3.1.5 Λογικοί τελεστές

Συνθετότερες λογικές παραστάσεις μπορούν να σχηματιστούν με χρήση των ***λογικών τελεστών*** (*logical* ή *Boolean operators*).

Λογικοί ονομάζονται οι τελεστές, οι οποίοι εφαρμόζονται σε μία, δύο ή περισσότερες λογικές παραστάσεις, οι οποίες υπολογίζονται σε τιμή `True` ή `False`. Η παράσταση που προκύπτει από την εφαρμογή των λογικών τελεστών σε λογικές παραστάσεις είναι πάντοτε λογική παράσταση.

Η λειτουργία των λογικών τελεστών μπορεί να απεικονιστεί σε αντίστοιχους πίνακες, γνωστούς ως ***πίνακες αληθείας*** (*truth tables*). Στους πίνακες αυτούς, απεικονίζονται όλα τα δυνατά αποτελέσματα από τη χρήση των αντίστοιχων λογικών τελεστών.

Οι λογικοί τελεστές είναι οι εξής:

Πίνακας 3.2: Πίνακας αληθείας λογικής σύζευξης (and)

A	B	A and B
False	False	False
False	True	False
True	False	False
True	True	True

and

Η παράσταση που προκύπτει υπολογίζεται στην τιμή `True`, αν όλες οι επί μέρους παραστάσεις τις οποίες συνδέει έχουν την τιμή `True`.

Για πράδειγμα,

```
>>> 1 != 2 and 2 != 3
True
>>> 1 != 2 and 2 == 3 and 3 != 4
False
>>> a = 10
>>> 2 < a and a < 20
True
>>> 2 < a and a < 5
False
```

Αν έστω και μία έχει την τιμή `False`, τότε υπολογίζεται στην τιμή `False`. Ο τελεστής `and` υλοποιεί τη ***λογική σύζευξη*** και, κατά συνέπεια, σε αυτόν αντιστοιχεί ο πίνακας αληθείας που απεικονίζεται στον πίνακα 3.2.

Η Python, επιτρέπει την απευθείας σύζευξη λογικών παραστάσεων όπως στο επόμενο παράδειγμα:

```
>>> x = 7
>>> 5 < x < 10
True
```

Το αμέσως προηγούμενο παράδειγμα, ισοδυναμεί με το:

```
>>> x = 7
>>> 5 < x and x < 10
True
```

Στην ίδια λογική, μπορούν να σχηματιστούν συνθετότερες παραστάσεις:

```
>>> 5 < x < 10 < 999
True
```

Πίνακας 3.3: Πίνακας αληθείας λογικής διάζευξης (or)

A	B	A or B
False	False	False
False	True	True
True	False	True
True	True	True

or

Η παράσταση που προκύπτει υπολογίζεται στην τιμή `False`, αν όλες οι επί μέρους παραστάσεις τις οποίες συνδέει έχουν την τιμή `False`. Αν έστω και μία έχει την τιμή `True`, τότε υπολογίζεται στην τιμή `True`.

Για παράδειγμα,

```
>>> True or False
True
>>> a = 10
>>> 2 < a or a < 20
True
>>> 2 < a or a < 5
True
```

Ο τελεστής `or` υλοποιεί τη ***λογική διάζευξη*** και κατά συνέπεια παρουσιάζει τον πίνακα αληθείας που απεικονίζεται στον πίνακα 3.3.

not

Αντιστρέφει την τιμή της λογικής παράστασης στην οποία εφαρμόζεται (***λογική άρνηση***).

Για παράδειγμα,

```
>>> not True
False
>>> not not True
True
>>> a = 10
>>> not a == 10
False
```

Ο πίνακας αληθείας του συγκεκριμένου λογικού τελεστή, έχει όπως απεικονίζεται στον πίνακα 3.4.

Πίνακας 3.4: Πίνακας αληθείας λογικής άρνησης (not)

A	not A
False	True
True	False

Σε μία λογική παράσταση με χρήση πολλών λογικών τελεστών, τη μεγαλύτερη προτεραιότητα μεταξύ τους έχει ο τελεστής `not` και μετά οι τελεστές `and` και `or`. Επίσης, η σειρά υπολογισμού των επί μέρους λογικών παραστάσεων είναι από αριστερά προς τα δεξιά. Για τις προτεραιότητες των λογικών τελεστών σε σχέση και με τους υπόλοιπους βλ. *Παράρτημα Α*.

Ένα σημείο που χρήζει προσοχής, επειδή μπορεί να οδηγήσει σε δύσκολα εντοπίσιμα προγραμματιστικά σφάλματα λογικής, είναι το ότι, σε μια σύνθετη λογική παράσταση (με χρήση λογικών τελεστών), δεν υπολογίζονται όλες οι επί μέρους λογικές παραστάσεις, αλλά από αριστερά προς τα δεξιά *μόνο όσες χρειάζονται* έτσι ώστε η γενική παράσταση να εξασφαλίσει μία σίγουρη τιμή:

```
>>> a = 0
>>> a > -1 or b / a > 10
True
```

Το τμήμα κώδικα στο παραπάνω παράδειγμα φαινομενικά θα έπρεπε να ενεργοποιήσει ένα σφάλμα - εξαίρεση του τύπου `ZeroDivisionError`, αφού η μεταβλητή `a` έχει τιμή `0`, άρα έχουμε διαίρεση με το `0` στην αριθμητική παράσταση `b/a`, στο δεύτερο μέρος της λογικής παράστασης. Όμως, η παράσταση `a > -1` είναι αληθής και αυτό εξασφαλίζει το αληθές ολόκληρης της παράστασης, αφού έχουμε χρήση του τελεστή `or`. Αν, αντί για τον τελεστή `or`, υπήρχε ο τελεστής `and`, τότε θα έπρεπε να υπολογιστεί και το δεύτερο μέρος της παράστασης, και εκεί θα εγείρετο εξαίρεση - μήνυμα σφάλματος `ZeroDivisionError`.

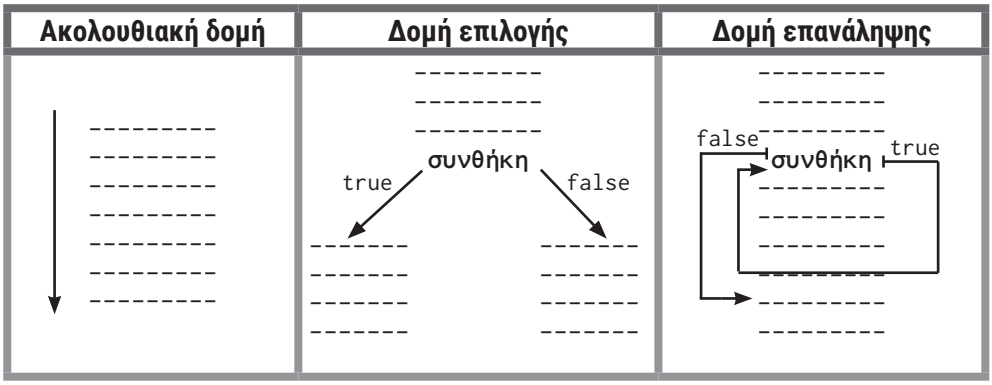
Αυτός ο τρόπος υπολογισμού λογικών παραστάσεων ονομάζεται ***υπολογισμός βραχυκύκλωσης*** ή ***οκνηρός*** (*short-circuit* ή *lazy evaluation*).

Ας σημειωθεί, τέλος, ότι μπορούν να χρησιμοποιηθούν παρενθέσεις ώστε να επηρεαστεί η σειρά εκτέλεσης ή/και να γίνει ο κώδικας πιο ευανάγνωστος. Για παράδειγμα η παράσταση,

```
(4 < 5) and (5 < 6)
```

θα υπολογιστεί διαδοχικά ως:

```
(4 < 5) and (5 < 6) →
True and (5 < 6)    →
True and True       →
True
```



Σχήμα 3.1 ***Δομές ελέγχου ροής****. Οι βασικές δομές ελέγχου της διαδοχής εκτέλεσης είναι η ακολουθιακή εκτέλεση εντολών, η επιλεκτική εκτέλεση εντολών με βάση συνθήκη και η επαναληπτική εκτέλεση εντολών με βάση συνθήκη.*

Ας σημειωθεί τέλος ότι, σε τύπους που ορίζονται από τον προγραμματιστή, μπορούν να οριστούν ειδικές μορφές συγκρίσεων, με χρήση υπερφόρτωσης τελεστών (βλ. κεφ. 8).

3.2 Δομές ελέγχου ροής

Ο έλεγχος ροής σε ένα πρόγραμμα καθορίζει τη διαδοχή εκτέλεσης των εντολών του προγράμματος. Μια εντολή ελέγχου ροής είναι μια εντολή, μέσω της οποίας μπορεί να επηρεαστεί η ροή εκτέλεσης εντολών του προγράμματος με βάση μια καθορισμένη συνθήκη.

Γενικά, οι γλώσσες προγραμματισμού παρέχουν τρεις βασικούς τύπους δομών ελέγχου ροής· την ***ακολουθιακή*** (*sequential*), τη δομή ***επιλογής*** (*selection*) και τη δομή ***επανάληψης*** ή ***βρόχου*** (*iterative, loop*) - σχ. 3.1.

Στην ακολουθιακή δομή, οι εντολές εκτελούνται η μία μετά την άλλη, με τη σειρά που εμφανίζονται στο πρόγραμμα. Τα παραδείγματα που εξετάστηκαν έως τώρα, - με μία εξαίρεση, βρόχου `for` στο 1ο κεφάλαιο - περιελάμβαναν μόνο τη δομή αυτή. Στην πράξη, ελάχιστα προγράμματα, συνήθως μόνο τα πολύ μικρά σε έκταση, αποτελούνται από μία μοναδική ακολουθία εντολών που εκτελούνται σειριακά και μόνο, από την αρχή ως το τέλος. Στη συντριπτική τους πλειοψηφία τα προγράμματα, εκτός από τμήματα εντολών που εκτελούνται ακολουθιακά, εκτελούν και εντολές υπό συνθήκη, δηλαδή εντολές που εκτελούνται επιλεκτικά, με βάση το αποτέλεσμα του υπολογισμού μιας λογικής συνθήκης, αλλά και επαναλαμβανόμενα για όσο ισχύει μια λογική συνθήκη. Αυτό επιτυγχάνεται με τη χρήση ειδικών εντολών μέσω των οποίων σχηματίζονται δομές ελέγχου, οι οποίες δίνουν τη δυνατότητα επιλεκτικής ή επαναλαμβανόμενης εκτέλεσης των εντολών ή ομάδων εντολών που εντάσσονται στις δομές αυτές.

Μια ακόμα μορφή ελέγχου ροής μπορεί να θεωρηθεί ότι συνιστούν τα υποπρογράμματα (αυτόνομα τμήματα πηγαίου κώδικα τα οποία καλούνται ως μία εντολή). Τα υποπρογράμματα, στην Python, υλοποιούνται με τη χρήση των συναρτήσεων και θα εξεταστούν εκτενώς στο κεφάλαιο 6. Εδώ θα ασχοληθούμε με τις δομές επιλογής και επανάληψης.

Οι δομές ελέγχου ροής περιλαμβάνουν δύο βασικά δομικά στοιχεία: τις ***συνθήκες*** (*λογικές εκφράσεις*), οι οποίες καθορίζουν τη ροή εκτέλεσης των εντολών, και τις ομάδες εντολών που θα εκτελεστούν ανάλογα με τον υπολογισμό, σε `True` ή `False`, των λογικών συνθηκών. Οι ομάδες εντολών στην Python δεν ορίζονται μέσω κάποιου ζεύγους χαρακτήρων π.χ. παρενθέσεων, αλλά ως εδάφια ίδιας εσοχής (βλ. πλαίσιο Π3.2).

3.2.1 Δομές επιλογής

Οι δομές αυτές δίνουν τη δυνατότητα να εκτελεστεί κάποιο επιλεγμένο τμήμα πηγαίου κώδικα, είτε με βάση τον υπολογισμό απλών ή και σύνθετων λογικών εκφράσεων, είτε στην περίπτωση που έχει συμβεί κάποιο σφάλμα χρόνου εκτέλεσης (*run-time error*).

Δομή if

Η δομή αυτή καθιστά δυνατή την εκτέλεση διαφορετικών εντολών, ή ομάδων εντολών, ανάλογα με το αποτέλεσμα του υπολογισμού λογικών εκφράσεων, παρουσιάζει δε, συντακτικά, την παρακάτω γενική μορφή:

```
if συνθήκη_if:
        ομάδα_εντολών_if
elif συνθήκη_elif_1:
        ομάδα_εντολών_elif_1
…
elif συνθήκη_elif_N:
        ομάδα_εντολών_elif_N
else:
        ομάδα_εντολών_else
```

Στη δομή αυτή χρησιμοποιούνται οι δεσμευμένες λέξεις `if`, `elif` οι οποίες εισάγουν τα τμήματα ελέγχου της δομής, και η `else` που ορίζει τι θα εκτελεστεί αν καμία από τις συνθήκες `if` και `elif` δεν υπολογιστεί ως αληθής.

Ως *συνθήκη_if*, *elif* κλπ. μπορεί να θεωρηθεί οποιαδήποτε λογική έκφραση, όπως εξετάστηκε στην 3.1, ενώ ομάδα εντολών νοείται ένα σύνολο όμοια εδαφιοποιημένων εντολών. Αναλυτικότερα, αν αληθεύει η *συνθήκη_if*, τότε εκτελείται η *ομάδα_εντολών_if* και στη συνέχεια ο έλεγχος του προγράμματος προωθείται στην πρώτη εντολή που έπεται της συνολικής δομής. Αν η *συνθήκη_if* είναι ψευδής, ελέγχεται αν αληθεύει η *συνθήκη_elif_1* (αν υπάρχει). Αν αυτή αληθεύει, τότε εκτελείται η *ομάδα_εντολών_elif_1*. Αν όχι, ελέγχεται η *συνθήκη_elif_2* (αν υπάρχει), κ.ο.κ. μέχρι κάποια συνθήκη να είναι αληθής, οπότε εκτελείται η αντίστοιχη ομάδα εντολών. Αν καμία συνθήκη δεν αληθεύει

τότε εκτελείται η *ομάδα_εντολών_else* (αν υπάρχει). Σε κάθε περίπτωση, μετά την ολοκλήρωση των πιο πάνω ελέγχων, το πρόγραμμα συνεχίζει την εκτέλεσή του από την πρώτη εντολή που έπεται της συνολικής δομής.

Π3.2 Σύνθετες εντολές στην Python

Ένα, μάλλον μοναδικό, χαρακτηριστικό της Python, σε σχέση με άλλες διαδεδομένες γλώσσες προγραμματισμού, είναι ότι οι εσοχές που υπάρχουν σε κάθε γραμμή κώδικα είναι συντακτικά σημαντικές.

Στις περισσότερες γλώσσες προγραμματισμού, οι εσοχές που, ενδεχομένως, υπάρχουν σε γραμμές προγράμματος, δεν επηρεάζουν τη λογική του, απλά χρησιμοποιούνται ως μέθοδος στοίχισης, ώστε να ενισχυθεί η αναγνωσιμότητα του προγράμματος· οι ομάδες εντολών στις δομές ελέγχου καθορίζονται μέσω κάποιου ζεύγους χαρακτήρων, όπως για παράδειγμα αγκύλες ({…} - C, C++) ή ακόμα και ζεύγος αναγνωριστικών (begin…end - Pascal) κ.λπ.

Στην Python, η εδαφιοποίηση μέσω χρήσης ίδιας ποσότητας εσοχών είναι η μέθοδος που χρησιμοποιείται για τον καθορισμό ***ομάδων εντολών*** (*suite*s) στις διάφορες δομές της γλώσσας.

Για παράδειγμα, στη δομή if, με αναφορά στο σχήμα 3.2, μπορεί να διακρίνει κανείς μια ***επικεφαλίδα*** (*header*) η οποία σχηματίζεται από τη δεσμευμένη λέξη `if` και μια συνθήκη ακολουθούμενη από μια άνω - κάτω τελεία (`:`), η οποία και την οριοθετεί, καθώς και μια ***ομάδα εντολών*** (*suite*) κάτω από αυτήν. Η άνω - κάτω τελεία, χρησιμοποιείται για το διαχωρισμό της επικεφαλίδας από την ομάδα εντολών που ακολουθεί και θα εκτελεστεί αν η συνθήκη του `if` υπολογιστεί στην τιμή `True`. Εδαφιοποιημένες με τον ίδιο τρόπο, δηλ. με ακριβώς την ίδια ποσότητα εσοχής ως προς την επικεφαλίδα, είναι και οι εντολές του σκέλους `else` της ίδιας δομής.

```
if a > b :                          Επικεφαλίδα
        εντολή_if_1                 Ομάδα εντολών
        εντολή_if_2                 (suite)
        ...
else:                               Επικεφαλίδα
        εντολή_else_1               Ομάδα εντολών
        εντολή_else_2               (suite)
        ...
```

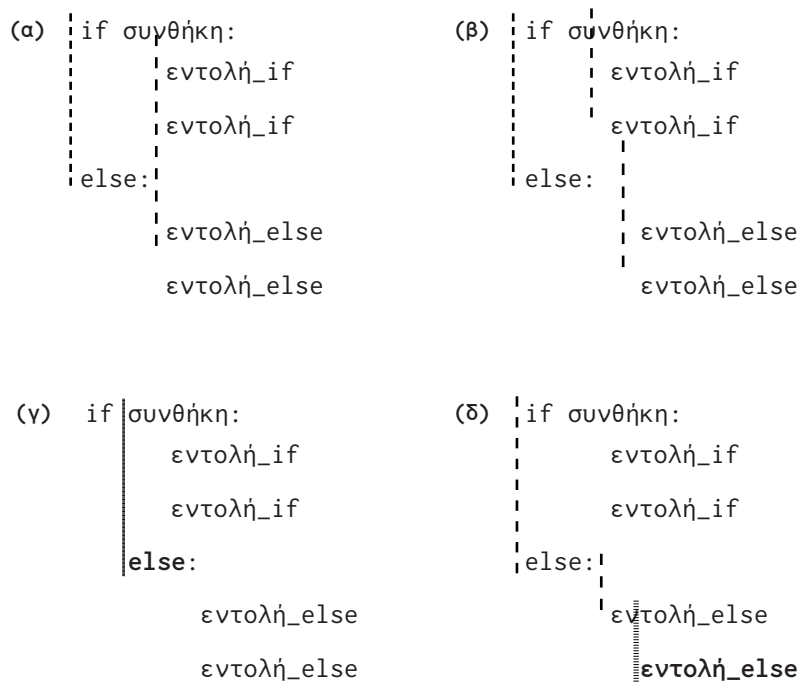
Σχήμα 3.2 ***Γενική δομή εντολών ελέγχου ροής****. Στη γλώσσα Python, οι ομάδες εντολών στις δομές ελέγχου ροής ορίζονται με εδαφιοποίηση με ίδια ποσότητα εσοχής ως προς τις επικεφαλίδες.*

Γενικότερα, η Python, αναγνωρίζει ποιες εντολές ανήκουν στην ομάδα εντολών που πρέπει να εκτελέσει, αν η συνθήκη που ορίζεται στην `if` υπολογιστεί ως `True`, από την εδαφιοποίηση τους μέσω ίδιας ακριβώς ποσότητας εσοχής (*indented block*) σε σχέση με την επικεφαλίδα. Μια επικεφαλίδα, μαζί με την αντίστοιχη ομάδα εντολών, αναφέρεται ως ***πρόταση*** της δομής (*clause*). ↓

Όπως θα εξεταστεί σε διάφορα παραδείγματα στη συνέχεια, συχνά, εντός μιας δομής ελέγχου, μπορεί να περιλαμβάνεται κάποια άλλη, οπότε πρέπει να δημιουργηθεί ένα νέο επίπεδο εδαφιοποίησης μετατοπισμένο με μια επιπλέον -ίδια- ποσότητα εσοχής ως προς το πρώτο κ.ο.κ. Μιλάμε τότε για ***φώλιασμα*** (*nesting*) μιας δομής σε μια άλλη.

Η γενικά προτεινόμενη ποσότητα εσοχής είναι τέσσερις χαρακτήρες κενού για κάθε επίπεδο εδαφιοποίησης (*http://legacy.python.org/dev/peps/pep-0008/*). Τα περισσότερα περιβάλλοντα ανάπτυξης που υποστηρίζουν τη γλώσσα δημιουργούν αυτόματα εσοχές στις δομές ελέγχου ροής της Python.

Χαρακτηριστικά παραδείγματα σωστής και λάθος εδαφιοποίησης φαίνονται στο σχήμα 3.3 που ακολουθεί. Η λανθασμένη χρήση εσοχών αποτελεί μορφή συντακτικού λάθους, οπότε ο διερμηνευτής θα εμφανίσει μήνυμα λάθους και θα διακόψει τη λειτουργία του προγράμματος:



Σχήμα 3.3 ***Εδαφιοποίηση στην Python****. Σωστή (α, β) και λάθος (γ,δ) εδαφιοποίηση στην Python.*

Στο σχήμα 3.2α, αμφότερα τα τμήματα εντολών, και στο σκέλος του if και στο σκέλος του else, έχουν την ίδια ακριβώς εσοχή. Στην περίπτωση 3.2β έχουμε διαφορετική εσοχή για την ομάδα εντολών του if από αυτήν του else, αλλά ίδια για κάθε ↓

ομάδα. Αυτό είναι συντακτικά σωστό, αλλά δεν αποτελεί σωστή πρακτική, επειδή η συνολική δομή δεν εμφανίζει τις ίδιες εσοχές και ο κώδικας γίνεται δυσανάγνωστος. Το παράδειγμα του 3.2γ αποτελεί λάθος εδαφιοποίηση, επειδή οι επικεφαλίδες if και else δεν εμφανίζουν την ίδια εσοχή. Τέλος, το παράδειγμα του 3.2δ, είναι επίσης συντακτικά απαράδεκτο, επειδή, παρ' όλο που οι επικεφαλίδες if και else βρίσκονται στο ίδιο επίπεδο εσοχής, οι εντολές της ομάδας του else, δεν είναι σωστά εδοφιοποιημένες μιας και εμφανίζουν διαφορετικό επίπεδο εσοχής μεταξύ τους.

Μια εντολή η οποία περιέχει μία ή περισσότερες προτάσεις (*clauses*), όπως η δομή επιλογής `if`, αναφέρεται ως ***σύνθετη εντολή*** (*compound statement*). Εκτός της δομής `if`, οι εντολές `while`, `for`, `try`, αποτελούν επίσης σύνθετες εντολές που υλοποιούν δομές ελέγχου και υπακούουν στους ίδιους συντακτικούς κανόνες. Οι ορισμοί συναρτήσεων και κλάσεων (βλ. κεφ. 6 και 7 αντίστοιχα), συνιστούν, επίσης από συντακτική άποψη, σύνθετες εντολές. ■

Ως ένα πρώτο παράδειγμα προγράμματος με χρήση της δομής `if`, ας θεωρηθεί μια πιθανή κωδικοποίηση, σε Python, του αλγορίθμου εύρεσης του μέγιστου τριών δοθέντων αριθμών. Ο αλγόριθμος για το συγκεκριμένο πρόβλημα εξετάστηκε στην αρχή του κεφαλαίου 1.

Πρόγραμμα kef3_1.py: Υπολογίζει το μέγιστο τριών αριθμών.

```
01 x = int(input("Εισαγάγετε τον πρώτο αριθμό: "))
02 y = int(input("Εισαγάγετε τον δεύτερο αριθμό: "))
03 z = int(input("Εισαγάγετε τον τρίτο αριθμό: "))
04 if x > y:
05     max = x
06 else:
07     max = y
08 if z > max:
09     max = z
10 print("Ο μεγαλύτερος των αριθμών ", x, y, z, "είναι ο ", max)
```

Στο παραπάνω πρόγραμμα, στις γραμμές `01` - `03`, αποδίδονται από το χρήστη τιμές σε τρεις μεταβλητές. Στις γραμμές `04` - `07`, μέσω μιας δομής επιλογής `if` - `else`, συγκρίνονται οι δύο πρώτες μεταβλητές και το περιεχόμενο της μεγαλύτερης εκχωρείται στη μεταβλητή `max`. Στις γραμμές `08` - `09`, μια απλή δομή `if` συγκρίνει την τρίτη μεταβλητή, `z` με την `max` και, αν αυτή βρεθεί μεγαλύτερη, η τιμή της εκχωρείται στη `max`, η οποία και τυπώνεται στη γραμμή `10`.

Παρατηρήσεις σχετικά με τη δομή If

Πρέπει να έχει κανείς υπόψη του ότι κάθε ομάδα εντολών πρέπει να περιλαμβάνει τουλάχιστον μία εντολή, για να έχει προγραμματιστικό νόημα η αλήθεια της αντίστοιχης συνθήκης. Αν ο προγραμματιστής, για οποιοδήποτε λόγο, θέλει, στην περίπτωση αλήθειας μιας συνθήκης, το πρόγραμμά του να μην εκτελέσει

καμία εντολή, πρέπει στη θέση της αντίστοιχης ομάδας, να τοποθετήσει την εντολή pass και μόνο αυτή. Έτσι το τμήμα προγράμματος που ακολουθεί είναι λανθασμένο γιατί δεν περιέχει ομάδα εντολών μεταξύ if και else:

```
...
if a >= b:
else:
    print(“b is greater than a”)
...
```

εγείρει εξαίρεση - μήνυμα σφάλματος.

Η σωστή προσέγγιση, είναι:

```
...
if a >= b:
    pass
else:
    print(“b is greater than a”)
...
```

Ο έλεγχος της συνθήκης if (και οι αντίστοιχες εντολές) είναι υποχρεωτικές συντακτικά. Οι υπόλοιποι έλεγχοι όχι. Επιτρέπεται να υπάρχουν περισσότερες από μία συνθήκες elif με τις αντίστοιχες ομάδες εντολών, και μόνο ένα else, το οποίο πρέπει να είναι στο τέλος της δομής.

Όπως αναφέρθηκε ήδη στο πλαίσιο Π3.2, η άνω - κάτω τελεία (:), στο τέλος των τμημάτων που ξεκινούν με if, elif και else (επικεφαλίδες της δομής), σηματοδοτεί το τέλος της συνθήκης και είναι υποχρεωτική· η παράλειψή της συνιστά συντακτικό λάθος. Οι κενές γραμμές οπουδήποτε μέσα σε μία δομή, αγνοούνται. Μπορεί κανείς να εκμεταλλευτεί το γεγονός αυτό για να επαυξήσει την ευκολία παρακολούθησης της λογικής ενός προγράμματος.

Εάν η δομή if είναι απλά της μορφής

```
if συνθήκη:
       εντολή
```

(δηλ. η ομάδα εντολών αποτελείται από μία μόνο εντολή), τότε μπορεί να γραφτεί και σε μία γραμμή ως εξής:

```
if συνθήκη: εντολή
```

Για παράδειγμα:

```
>>> day = ‘Παρασκευή’
>>> if day == ‘Παρασκευή’:
...     print(‘Καλό Σαββατοκύριακο !!!’)
...
Καλό Σαββατοκύριακο !!!
>>> if day == ‘Παρασκευή’: print(‘Καλό Σαββατοκύριακο !!!’)
Καλό Σαββατοκύριακο !!!
```

Παρόμοια, αν η δομή είναι της μορφής

```
if συνθήκη:
       εντολή1
else:
       εντολή2
```

τότε μπορεί να γραφτεί και αυτή σε μία γραμμή, σύμφωνα με την σύνταξη:

```
εντολή1 if συνθήκη else εντολή2
```

Η συγκεκριμένη μορφή είναι πολύ χρήσιμη γιατί, αν οι εντολές 1 και 2 επιστρέφουν κάθε μία τη δική της τιμή, όλη η δομή είναι μία παράσταση, η οποία έχει τιμή που μπορεί να ανατεθεί σε μεταβλητή ή να χρησιμοποιηθεί σε κάποια λογική έκφραση.

Για παράδειγμα, ένα τμήμα κώδικα όπως το ακόλουθο:

```
...
if fylo == "Α":
     print("Γεια σας, κύριε !!!")
else:
     print("Γεια σας, κυρία !!!")
...
```

μπορεί να γραφτεί εναλλακτικά ως εξής:

```
>>> print("Γεια σας, κύριε !!!") if fylo == "Α" else print("Γεια σας,
κυρία !!!")
```

Φώλιασμα

Μια ομάδα εντολών σε μια δομή ελέγχου όπως η `if`, μπορεί να περιέχει οποιοδήποτε τύπο εντολής, κάποιας άλλης δομής ελέγχου συμπεριλαμβανομένης. Όπως ήδη αναφέρθηκε στο Π3.2, μια δομή ελέγχου εντός μιας άλλης δομής λέγεται ***φωλιασμένη*** (*nested*). Στο πρόγραμμα που ακολουθεί, χρησιμοποιείται μια δομή `if` (γραμμές 03 - 06) εντός μιας άλλης (γραμμές 02 - 08), για να ελεγχθεί η εγκυρότητα ενός αριθμού ο οποίος εισάγεται από το χρήστη.

Φώλιασμα μπορεί να εφαρμοστεί και σε άνω του ενός επίπεδα (φώλιασμα εντός φωλιάσματος, εντός φωλιάσματος κ.ο.κ). Η δυνατότητα αυτή πρέπει να χρησιμοποιείται με προσοχή, επειδή μπορεί να καταλήξει σε δυσανάγνωστο κώδικα.

Πρόγραμμα kef3_2.py: Δομή `if` φωλιασμένη σε άλλη.

```
01 inp_num = int(input("Εισάγετε έναν μονοψήφιο θετικό αριθμό ή \
       μηδέν: "))
02 if inp_num >= 0:
03    if inp_num < 10:
04        print(" -- Αποδεκτός αριθμός -- ")
05    else:
06        print(" -- Σφάλμα. Μη μονοψήφιος αριθμός -- ")
07 else:
08    print(" -- Σφάλμα. Αρνητικός αριθμός -- ")
```

Δομή try

Η δομή `try` χρησιμοποιείται, στην Python, για τη διαχείριση ***σφαλμάτων χρόνου εκτέλεσης*** του προγράμματος (*runtime errors*). Αυτά τα σφάλματα ονομάζονται ***εξαιρέσεις*** (*exceptions*) - λέμε ότι η Python ***εγείρει - ενεργοποιεί εξαίρεση*** (*raises exception*) - έχουν δε, ως αποτέλεσμα, τη διακοπή της εκτέλεσής του, εκτός εάν ο προγραμματιστής έχει μεριμνήσει για την αντιμετώπισή τους. Η μέριμνά του πρέπει να είναι, αφ' ενός μεν να γνωρίζει τα σημεία του προγράμματος που είναι πιθανό να συμβούν εξαιρέσεις, αφ' ετέρου δε να τοποθετήσει στον κώδικά του τους κατάλληλους ***χειριστές εξαιρέσεων*** (*exception handlers*). Οι χειριστές αυτοί δεν είναι τίποτε άλλο από μία ή περισσότερες δομές `try`. Η γενική μορφή των δομών αυτών είναι η εξής:

```
try:
        ομάδα_εντολών_try
except όνομα_εξαίρεσης_1:
        ομάδα_εντολών_1
except όνομα_εξαίρεσης_2:
        ομάδα_εντολών_2
        …
except όνομα_εξαίρεσης_N:
        ομάδα_εντολών_N
else:
        ομάδα_εντολών_else
finally:
        ομάδα_εντολών_finally
```

Εδώ χρησιμοποιούνται οι δεσμευμένες λέξεις της Python *try*, *except*, *else*, *finally*.

Η `try` εισάγει την ομάδα εντολών που θα εκτελεστεί και που, στο στενό της πλαίσιο εκτέλεσης, θα γίνεται ο χειρισμός των εξαιρέσεων, και είναι υποχρεωτική. Η `except` ορίζει τους χειριστές εξαιρέσεων για συγκεκριμένες εξαιρέσεις, έναν προς έναν, μέσω των *ομάδων_εντολών_1, 2, …, N*. Δεν είναι υποχρεωτικό να υπάρχουν εντολές `except`, για την ώρα, όμως, δεν θα τις παραλείπουμε. Είναι δυνατό ο χειριστής εξαιρέσεων να μην αφορά σε κάποια συγκεκριμένη εξαίρεση, αλλά να παρέχεται για οποιαδήποτε, εν είδει γενικού χειριστή (πάντοτε όμως στο συγκεκριμένο πλαίσιο εντολών). Αυτό γίνεται αν παραλειφθεί το όνομα της εξαίρεσης. Η `else` εισάγει μία ομάδα εντολών η οποία θα εκτελεστεί αν δεν ενεργοποιηθεί καμία εξαίρεση, και είναι και αυτή μη υποχρεωτική. Τέλος, η `finally`, που είναι και αυτή προαιρετική, εισάγει την ομάδα εντολών που θα εκτελεστούν σε κάθε περίπτωση, είτε ενεργοποιηθεί εξαίρεση είτε όχι.

Και εδώ οι άνω και κάτω τελείες (`:`) οριοθετούν τις επικεφαλίδες και κατά συνέπεια είναι υποχρεωτικές. Επίσης ισχύουν και εδώ οι κανόνες περί εσοχών: οι εντολές μέσα σε μία ομάδα εντολών πρέπει να είναι σε εσοχή σε σχέση με την εντολή εισαγωγής στην ομάδα, και οι εντολές εισαγωγής πρέπει να είναι στην ίδια στήλη μεταξύ τους.

Οι εξαιρέσεις και η δομή try θα εξεταστούν εκτενέστερα στο κεφάλαιο 9, όπου θα εξεταστούν και εναλλακτικοί τρόποι χρήσης και σύνταξης. Εδώ θα εξεταστεί η βασική χρήση της δομής.

Η ροή της εκτέλεσης του προγράμματος είναι η εξής: Εκτελούνται κανονικά οι εντολές στην *ομάδα_εντολών_try*. Αν συμβεί κάποιο σφάλμα χρόνου εκτέλεσης (π.χ. διαίρεση με το μηδέν), τότε κανονικά το πρόγραμμα θα διακοπεί. Αν, όμως, σε κάποιο τμήμα χειριστών εξαιρέσεων, έχει προβλεφθεί ο κατάλληλος για τη συγκεκριμένη εξαίρεση χειριστής, τότε η ροή του προγράμματος μεταφέρεται εκεί. Εκτελούνται όλες οι εντολές της συγκεκριμένης ομάδας, και τέλος η ροή μεταφέρεται στην *ομάδα_εντολών_finally* (αν υπάρχει). Ας δούμε ένα παράδειγμα. Έστω το παρακάτω πρόγραμμα:

Πρόγραμμα kef3_3.py: Διαίρεση χωρίς πρόβλεψη μηδενικού διαιρέτη

```
01  D = int(input('Παρακαλώ δώστε τον διαιρετέο: '))
02  d = int(input('Παρακαλώ δώστε τον διαιρέτη: '))
03  P = D // d
04  Y = D % d
05  print('Πηλίκο και υπόλοιπο διαίρεσης: ')
06  print(P, Y)
```

Αν εδώ ο χρήστης δώσει στον διαιρέτη την τιμή 0, τότε θα δημιουργηθεί σφάλμα χρόνου εκτέλεσης '*integer division or modulo by zero*' και το πρόγραμμα θα διακοπεί:

```
01  Παρακαλώ δώστε τον διαιρετέο: 10
02  Παρακαλώ δώστε τον διαιρέτη: 0
03  Traceback (most recent call last):
04    File "kef3_3.py", line 3, in <module>
05      P = D // d
06  ZeroDivisionError: integer division or modulo by zero
```

Η γραμμή που μας ενδιαφέρει για την ώρα είναι η τελευταία (06), όπου φαίνεται το όνομα της εξαίρεσης (*ZeroDivisionError*) και η περιγραφή του σφάλματος ('*integer division or modulo by zero*'), και ή 04, όπου φαίνεται το σημείο στον πηγαίο κώδικα όπου συνέβη αυτό ('*File "kef3_3.py", line 3…*'). Η ουσία είναι ότι, επειδή δεν είχε προβλεφθεί χειριστής για την εξαίρεση *ZeroDivisionError* το πρόγραμμα διεκόπη. Για να αντιμετωπιστεί αυτό με ελεγχόμενο τρόπο, πρέπει να τοποθετηθούν, κάτω από μία εντολή try, τουλάχιστον οι εντολές που είναι πιθανό να ενεργοποιήσουν την εξαίρεση, και να γραφτεί ένας κατάλληλος χειριστής για την περίπτωση της ενεργοποίησης:

Πρόγραμμα kef3_4.py: Διαίρεση με πρόβλεψη μηδενικού διαιρέτη

```
01  D = int(input('Παρακαλώ δώστε τον διαιρετέο: '))
02  d = int(input('Παρακαλώ δώστε τον διαιρέτη: '))
03  try:
04      P = D // d
05      Y = D % d
06  except ZeroDivisionError:
07      print('ΠΡΟΣΟΧΗ: Ο διαιρέτης δεν μπορεί να είναι 0')
08  else:
09      print('Το πηλίκο και το υπόλοιπο είναι αντίστοιχα: ')
10      print(P, Y)
```

Αν ο χρήστης δώσει σωστά δεδομένα, η ροή του προγράμματος θα είναι η εξής: αφού διαβαστούν οι δύο όροι της διαίρεσης, το πρόγραμμα εισέρχεται στον χειριστή εξαιρέσεων μέσω της εντολής `try` (γραμμή `03`). Εκεί θα εκτελεστούν κανονικά όλες οι εντολές της *ομάδας_εντολών_try* (`04`, `05`) και, επειδή δεν ενεργοποιήθηκε καμία εξαίρεση (γιατί ο διαιρέτης δεν είναι μηδέν), δεν θα εκτελεστεί η *ομάδα_εντολών_except* (`07`) , αλλά η *ομάδα_εντολών_else* (`09`, `10`).

Αν, όμως, ο χρήστης δώσει στον διαιρέτη την τιμή `0`, τότε στη γραμμή `04` θα ενεργοποιηθεί η εξαίρεση *ZeroDivisionError*. Αμέσως διακόπτεται η εκτέλεση της *ομάδας_εντολών_try*. *Δεν* θα εκτελεστεί η εντολή στη γραμμή `05` (και καμία άλλη παρακάτω αν υπήρχε). Επειδή έχει προβλεφθεί, για τη συγκεκριμένη εξαίρεση, χειριστής, που έχει εισαχθεί με την εντολή `except` (γραμμή `06` - το όνομα της εξαίρεσης πρέπει να είναι ακριβώς το ίδιο με αυτό που μας δείχνει ο διερμηνευτής της Python, σε περίπτωση σφάλματος χρόνου εκτέλεσης λόγω κάποιας εξαίρεσης), το πρόγραμμα θα συνεχιστεί με την *ομάδα_εντολών_except* στη γραμμή `07`. Μετά θα τερματίσει, καθώς δεν θα εκτελεστεί η *ομάδα_εντολών_else*, αφού είχαμε ενεργοποίηση εξαίρεσης.

```
01  Παρακαλώ δώστε τον διαιρετέο: 10
02  Παρακαλώ δώστε τον διαιρέτη: 0
03  ΠΡΟΣΟΧΗ: Ο διαιρέτης δεν μπορεί να είναι 0
```

Ένας εναλλακτικός τρόπος αποφυγής της διαίρεσης με το `0` θα ήταν με μία δομή `if`, όπου θα ελεγχόταν η τιμή του `d` πριν τη διαίρεση:

```
01  D = int(input('Παρακαλώ δώστε τον διαιρετέο: '))
02  d = int(input('Παρακαλώ δώστε τον διαιρέτη: '))
03  if d == 0:
04      print('ΠΡΟΣΟΧΗ: Ο διαιρέτης δεν μπορεί να είναι 0')
05  else:
06      print('Το πηλίκο και το υπόλοιπο είναι αντίστοιχα: ')
07      print(P, Y)
```

Η έξοδος του προγράμματος θα ήταν ακριβώς η ίδια. Άρα, γιατί να χρησιμοποιηθούν οι χειριστές εξαιρέσεων, όταν με ένα απλό `if` θα μπορούσε να επιτευχθεί ο ίδιος σκοπός; Υπάρχουν περιπτώσεις όπου είναι αδύνατο να αποφευχθούν

οι εξαιρέσεις, ακόμα και με πλήρεις ελέγχους πριν την εκτέλεση συγκεκριμένων διαδικασιών, έτσι η χρήση χειριστών είναι αναγκαστική. Επίσης, κάθε εξαίρεση είναι ένα αντικείμενο (object - υπενθυμίζεται ότι η Python είναι μία αντικειμενοστρεφής γλώσσα προγραμματισμού), κάτι που δίνει πολλές δυνατότητες όσον αφορά στον χειρισμό της. Περισσότερα, όμως, θα δειχθούν στο κεφάλαιο 9.

3.2.3 Δομή επανάληψης while

Οι ***δομές επανάληψης*** (***βρόχοι*** - *loops*) δίνουν τη δυνατότητα της εκτέλεσης μιας ομάδας εντολών πολλές φορές, βάσει μίας συνθήκης η οποία καθορίζει τον αριθμό των επαναλήψεων. Η συνθήκη μπορεί να μη μεταβάλλεται, οπότε και ο αριθμός των εκτελέσεων είναι, πρακτικά, προδιαγεγραμμένος, μπορεί όμως και να μεταβάλλεται μέσα στο βρόχο.

Οι δύο δομές επανάληψης που υπάρχουν στην Python είναι η δομή `while` και η δομή `for`. Επειδή, στην Python, η δομή `for` σχετίζεται με ορισμένες ειδικές έννοιες και δυνατότητες, θα εξεταστεί ξεχωριστά στην επόμενη ενότητα (3.3).

Δομή while

Η μορφή της είναι:

```
while συνθήκη:
        ομάδα_εντολών_while
else:
        ομάδα_εντολών_else
```

Οι δεσμευμένες λέξεις που χρησιμοποιούνται είναι η `while` και η `else`.

Η `while` εισάγει τη συνθήκη βάσει της οποίας θα εκτελείται η *ομάδα_εντολών_while*. Όσο η συνθήκη αληθεύει, τόσο θα εκτελείται η εν λόγω ομάδα εντολών. Όταν η συνθήκη γίνει ψευδής ο βρόχος τερματίζεται.

Η `else` εισάγει την ομάδα εντολών που θα εκτελεστεί αμέσως μετά την τελευταία εκτέλεση της *ομάδας_εντολών_while* και είναι προαιρετική.

Η συνθήκη του βρόχου υπολογίζεται πάντα πριν την εκτέλεση των ομάδων εντολών `while`. Κατά συνέπεια, υπάρχει η πιθανότητα ο βρόχος να μην εκτελεστεί ποτέ, αν η συνθήκη υπολογιστεί από την αρχή ψευδής (`False`).

Επίσης, αν η συνθήκη είναι πάντα αληθής, και μέσα στην *ομάδα_εντολών_while* δεν υπάρχει κάποια εντολή `break` (βλ. *3.3.3* παρακάτω), ο βρόχος θα εκτελείται για πάντα -***ατέρμων βρόχος***- οπότε θα πρέπει να διακοπεί από το χρήστη με τρόπο που ορίζει το λειτουργικό σύστημα.

Ισχύουν οι γνωστοί κανόνες περί εδαφιοποίησης μέσω εσοχών σε σχέση με την επικεφαλίδα.

Πρόγραμμα kef3_5.py: Απλός βρόχος while.

```
01  count = 0
02  while count < 5:
03      print(count)
04      count += 1
05  else:
06      print("Τέλος του βρόχου")
```

Ο παραπάνω βρόχος θα εκτελείται όσο η μεταβλητή count έχει τιμή μικρότερη του 5. Όταν η μεταβλητή πάρει την τιμή 5 ο βρόχος θα τερματίσει, και θα κληθεί η συνάρτηση print() εντός του else. Η έξοδος του προγράμματος θα είναι:

```
0
1
2
3
4
Τέλος του βρόχου
```

Παράδειγμα βρόχου που δεν θα εκτελεστεί ποτέ γιατί η συνθήκη είναι εξαρχής ψευδής είναι το ακόλουθο:

```
>>> count = 10
>>> while count < 10:
...     print(count)
...     count += 1
...
```

Στο επόμενο παράδειγμα, παρουσιάζεται μια περίπτωση ατέρμονα βρόχου, μια και η συνθήκη ελέγχου των επαναλήψεων του βρόχου, δεν θα γίνει ποτέ ψευδής, αφού η μεταβλητή count που ελέγχεται αν είναι μεγαλύτερη του 0, πάντα θα αυξάνεται μέσα στο βρόχο.

```
>>> count = 10
>>> while count > 0:
...     print(count)
...     count += 1
...
10
11
...
```

Για να διακοπεί ένας ατέρμων βρόχος, πρέπει ο χρήστης να διακόψει όλο το πρόγραμμα, με τη χρήση των κατάλληλων μηχανισμών που απαιτεί το λειτουργικό σύστημα γι' αυτή την περίπτωση (Ctrl + C, για τα περιβάλλοντα Linux και Windows).

Σε ένα δεύτερο παράδειγμα ατέρμονα βρόχου, ως συνθήκη έχει τεθεί η δεσμευμένη λέξη True η οποία είναι ταυτόσημη με την έννοια '*αληθής έκφραση*'.

```
>>> while True:
...     name = input("Ποιο είναι το όνομά σου; ")
...     print("Γεια σου " + name)
...
Ποιο είναι το όνομά σου; Στράτος
Γεια σου Στράτος
Ποιο είναι το όνομά σου; Γιώργος
Γεια σου Γιώργος
Ποιο είναι το όνομά σου;
...
```

Ας σημειωθεί, τέλος, ότι, εντός μιας δομής `while`, μπορούν να χρησιμοποιηθούν οι εντολές `break` και `continue`. Με την εντολή `break` μπορεί να γίνει διακοπή της εκτέλεσης του βρόχου πριν το φυσιολογικό του τερματισμό, τυπικά μέσω μιας εντολής ελέγχου συνθήκης (`if`). Με την εντολή `continue`, δίνεται η δυνατότητα παράκαμψης κάποιων από τις εντολές του βρόχου πάλι με βάση μια εντολή ελέγχου συνθήκης. Παραδείγματα χρήσης των εντολών αυτών παρατίθενται στην 3.3.3 παρακάτω.

3.3 Επαναλήψιμοι τύποι δεδομένων και η δομή for

Λέμε ότι μία μεταβλητή είναι ***επαναλήψιμη*** (ή αλλιώς ο τύπος δεδομένων της είναι ***επαναλήψιμος*** - *iterable*) όταν μπορούμε να διατρέξουμε όλα τα στοιχεία που έχει εσωτερικά η μεταβλητή αυτή, ένα προς ένα, με τη σειρά. Μάλιστα, αυτό μπορεί να γίνει ακόμα και αν δεν είναι γνωστό το μέγεθος της· ακόμα και κατά τη δημιουργία της. Αυτός και μόνο ο ορισμός αφαιρεί την ιδιότητα της επαναληψιμότητας από τους απλούς τύπους δεδομένων όπως `bool`, `int`, `float`, `complex`. Αντίθετα, αυτή η ιδιότητα υπάρχει στους περισσότερους τύπους συλλογών δεδομένων, ακολουθιακούς και μη: λίστες, πλειάδες, σύνολα, λεξικά, εύρη τιμών, αρχεία κ.α. Έτσι, η λίστα [1, 2, 3, 1, 4, 2] είναι επαναλήψιμη, γιατί μπορούμε να διατρέξουμε τα στοιχεία της ένα προς ένα: 1, 2, 3, 1, 4, 2. Μια συμβολοσειρά είναι, επίσης, επαναλήψιμη, επειδή μπορεί να επιστραφεί από αυτήν ένας χαρακτήρας κάθε φορά. Επίσης, το σύνολο {10, 20, 30}, το λεξικό {'red' : 1, 'green' : 2, 'yellow' : 3} κ.α. (βλ. και πλαίσιο Π3.3).

3.3.1 Δομή for

Η γενική μορφή της δομής `for` είναι

```
for V in E:
        ομάδα_εντολών_for
else:
        ομάδα_εντολών_else
```

όπου *V* είναι μία ομάδα από μία ή περισσότερες μεταβλητές των οποίων οι τιμές θα αλλάξουν όσο εκτελείται η δομή βάσει της παράστασης *E* η οποία

είναι επαναλήψιμη, και ουσιαστικά είναι το πεδίο ορισμού τους. Στη δομή for χρησιμοποιούνται οι δεσμευμένες λέξεις **for** και **in** που εισάγουν τις *V* και *E* αντίστοιχα, και η **else** η οποία είναι προαιρετική και εισάγει την ομάδα εντολών που θα εκτελεστούν όταν τερματιστεί ο βρόχος.

Στην απλούστερη μορφή της δομής, το *V* είναι μία απλή μεταβλητή, και το *E* μία παράσταση που έχει ως τύπο δεδομένων έναν επαναλήψιμο τύπο π.χ. μια λίστα. Για παράδειγμα:

```
>>> for i in [1, 2, 3]:
...     i *= 2
...     print(i)
...
2
4
6
```

Εδώ η μεταβλητή `i` παίρνει τιμές από τα μέλη της λίστας, δηλ. διαδοχικά τις `1`, `2`, `3`. Για κάθε μία από αυτές, τυπώνει στο τερματικό το διπλάσιό της.

Στη $2^{η}$ γραμμή φαίνεται ότι μεταβάλλεται η τιμή της μεταβλητής `i`, αυτό όμως ισχύει μόνο για την τρέχουσα επανάληψη του βρόχου. Στην επόμενη επανάληψη η `i` θα πάρει την επόμενη τιμή από τη λίστα κανονικά. Αυτό σημαίνει ότι οι τιμές *V* είναι καθορισμένες πριν την είσοδο στη δομή, βάσει της *E*, και αυτό δεν μπορεί να αλλάξει, εκτός αν η *E* είναι μία μεταβλητή που μεταβάλλεται εντός του βρόχου. Αυτό, όμως, δεν είναι καλή προγραμματιστική πρακτική, μπορεί να οδηγήσει σε παρενέργειες και σφάλματα τα οποία είναι δύσκολο να ανιχνευθούν. Ως εκ τούτου, καλό είναι να αποφεύγεται.

Στο παράδειγμα που ακολουθεί, η μεταβλητή `i` παίρνει τιμές τους χαρακτήρες που απαρτίζουν τη λέξη *'Python'*. Επειδή η Python μεταχειρίζεται τις συμβολοσειρές ως λίστες χαρακτήρων, η έκφραση *`'i in "Python"'`* είναι ισοδύναμη με την *`'i in ['P', 'y', 't', 'h', 'o', 'n']`*, οπότε επιστρέφονται και τυπώνονται στο τερματικό, σε ατομική βάση, οι χαρακτήρες της συγκεκριμένης συμβολοσειράς:

```
>>> for i in "Python":
...     print(i)
...
P
y
t
h
o
n
```

Εκτός από λίστα, η παράσταση *E* μπορεί να επιστρέφει πλειάδα, λεξικό, σύνολο, εύρος τιμών ή ακόμα και αρχείο δεδομένων. Το παράδειγμα που ακολουθεί, τυπώνει στοιχεία πλειάδας:

```
>>> for i in (1, “Κώστας Παπαδόπουλος“, 1.80, 77, “Αθήνα“, 1975, \
...      [“Μαρία“, “Νίκος“]):
...    print(i)
...
1
Κώστας Παπαδόπουλος
1.8
77
Αθήνα
1975
[‘Μαρία’, ‘Νίκος’]
```

Όσον αφορά στα λεξικά, η Python τα αποθηκεύει στη μνήμη με τρόπο που θεωρεί βέλτιστο για το σύστημα. Για το παράδειγμα που ακολουθεί, αυτό σημαίνει ότι δεν είναι σίγουρο ότι θα τυπώσει τα τρία χρώματα με τη συγκεκριμένη σειρά. Αυτό εξαρτάται από την υλοποίηση, την έκδοση του διερμηνευτή κ.α.

```
>>> for i in {“red“ : 1, “green“ : 2, “yellow“ : 3}:
...    print(i)
...
green
yellow
red
```

Ας παρατηρηθεί εδώ ότι τυπώθηκαν μόνο τα κλειδιά του λεξικού και όχι οι τιμές. Για να τυπωθούν και τα δύο πρέπει το συγκεκριμένο απόσπασμα προγράμματος να γραφτεί ως εξής:

```
>>> dictionary = {“red“ : 1, “green“ : 2, “yellow“ : 3}
>>> for i in dictionary:
...    print(i, dictionary[i])
...
green 2
yellow 3
red 1
```

Ή, διαφορετικά,

```
>>> dictionary = {‘red’ : 1, ‘green’ : 2, ‘yellow’ : 3}
>>> for key, value in dictionary.items():
...    print(key, value)
...
green 2
yellow 3
red 1
```

Στην αμέσως προηγούμενη παραλλαγή φαίνεται η δυνατότητα της `for` να έχει περισσότερες από μία μεταβλητές που παίρνουν τιμή από την παράσταση *E*. Η μέθοδος `items()` είναι μία μέθοδος των λεξικών που επιστρέφει τα ζευγάρια (κλειδί, τιμή) ως λίστα από πλειάδες 2 μεταβλητών (βλ. κεφ. 5). Έτσι οι μεταβλητές `key` και `value` παίρνουν τις τιμές των μεταβλητών της κάθε πλειάδας. Επομένως το τμήμα

```
{‘red’ : 1, ‘green’ : 2, ‘yellow’ : 3}.items()
```

ισοδυναμεί με:

```
[(‘red’, 1), (‘green’, 2), (‘yellow’, 3)]
```

Το V μπορεί να είναι και πολυπλοκότερης μορφής, αρκεί να ταιριάζει στη δομή του E. Για παράδειγμα,

```
>>> for (i, j) in [(1, 2), (3, 4), (5, 6)]:
...    print(i)
...    print(j)
...    print(i, j)
...
1
2
1 2
3
4
3 4
5
6
5 6
```

Π3.3 Επαναλήπτες, επαναλήψιμοι τύποι και γεννήτορες

Στην Python, ο μηχανισμός που βρίσκεται πίσω από δομές επανάληψης, όπως αυτές που εξετάζονται στην 3.3, βασίζεται στις έννοιες του *επαναλήπτη* και του *επαναλήψιμου τύπου*.

Ένας ***επαναλήπτης*** (*iterator*) είναι ένα αντικείμενο (μεταβλητή), το οποίο διαχειρίζεται μια επανάληψη, μέσω μιας σειράς τιμών. Αν μια μεταβλητή `i` αποτελεί ένα αντικείμενο επαναλήπτη, τότε κάθε κλήση στην ενσωματωμένη συνάρτηση `next(i)` παράγει ένα επόμενο στοιχείο, αντλώντας διαδοχικά στοιχεία από έναν επαναλήψιμο τύπο. Όταν εξαντληθούν τα διαθέσιμα στοιχεία, εγείρεται μια εξαίρεση *`StopIteration`* μέσω της οποίας σηματοδοτείται το τέλος της επανάληψης.

Ένας ***επαναλήψιμος τύπος*** (*iterable*) είναι ένα αντικείμενο `obj`, κατάλληλου τύπου, ώστε να μπορεί να δημιουργήσει έναν επαναλήπτη μέσω της σύνταξης `iter(obj)`. Όπως έχει αναφερθεί, τύποι όπως οι συμβολοσειρές, οι λίστες, οι πλειάδες, τα σύνολα, τα λεξικά είναι επαναλήψιμοι. Σε τύπους όπως οι λίστες και οι πλειάδες, τα στοιχεία τυπικά επιστρέφονται σε ακολουθιακή διάταξη αρχίζοντας από το πρώτο στοιχείο (με δείκτη θέσης `0`). Άλλοι επαναλήψιμοι τύποι, όπως τα σύνολα και τα λεξικά, επιστρέφουν στοιχεία σε τυχαία σειρά.

Σύμφωνα με τα παραπάνω, στο επόμενο παράδειγμα χρησιμοποιείται η συνάρτηση `iter()` (στην πραγματικότητα η μέθοδος `__iter__()`) η οποία επιστρέφει έναν επαναλήπτη με βάση μια συμβολοσειρά. Στη συνέχεια, καλείται διαδοχικά η συνάρτηση `next()` (μέθοδος `__next__()`) με όρισμα τον επαναλήπτη, για την άντληση των χαρακτήρων της συμβολοσειράς.

```
>>> i = iter(“αβγδ“)
>>> next(i)
‘α’
```

⬇

```
>>> next(i)
'β'
>>> next(i)
'γ'
>>> next(i)
'δ'
>>> next(i)
Traceback (most recent call last):
      File «<pyshell#5>», line 1, in <module>
next(i)
StopIteration
```

Ο βρόχος `for` της Python απλά αυτοματοποιεί διαδικασίες του συγκεκριμένου τύπου, δημιουργώντας αυτόματα τον επαναλήπτη για δεδομένο επαναλήψιμο τύπο, και, στη συνέχεια, καλώντας επαναληπτικά το επόμενο στοιχείο μέχρι τη εμφάνιση μιας εξαίρεσης `StopIteration` την οποία επίσης χειρίζεται αυτόματα.

Γενικότερα, είναι δυνατό να δημιουργηθούν πολλαπλοί επαναλήπτες, βασισμένοι πάνω στο ίδιο επαναλήψιμο αντικείμενο, με κάθε επαναλήπτη να τηρεί τη δική του κατάσταση προόδου.

Η Python υποστηρίζει, επίσης, οντότητες οι οποίες παράγουν έμμεσα επαναλήπτες, δηλ. χωρίς να σχηματίζουν αντίστοιχη δομή δεδομένων, π.χ. μια λίστα, για να αποθηκεύσουν όλες τις τιμές από την αρχή. Οι οντότητες αυτές είναι γνωστές ως ***γεννήτορες*** (*generators*). Ένα παράδειγμα γεννήτορα που παρέχει η γλώσσα είναι η `range()`, η οποία εξετάζεται αμέσως μετά το παρόν πλαίσιο. Στην τυπική περίπτωση, αρχικά καλείται ο γεννήτορας ο οποίος επιστρέφει έναν επαναλήπτη:

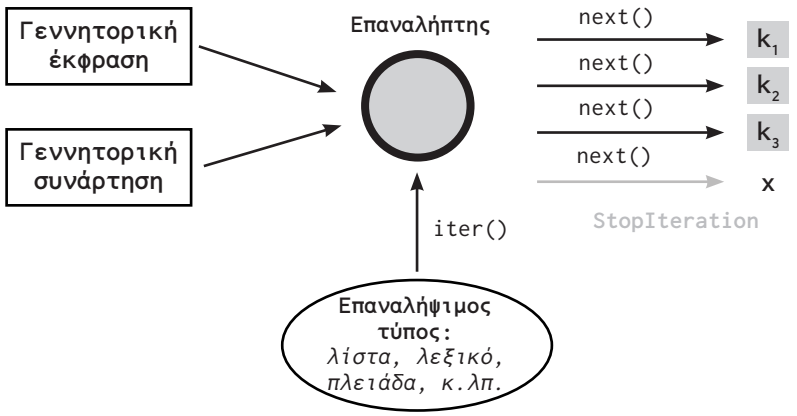
```
>>> g = range(3)
>>> g
range(0, 3)
```

Πρόκειται για πραγματικό επαναλήπτη, εφοδιασμένο με τις `iter()` και `next()`. Κατά συνέπεια, μπορεί κανείς να τον χειριστεί άμεσα με τη `next()`, όπως πιο πάνω:

```
>>> i = iter(g)
>>> next(i)
0
>>> next(i)
1
…
```

ή να χρησιμοποιήσει τον παραγόμενο επαναλήπτη σε μια δομή βρόχου `for`. Η διαφορά έγκειται στο γεγονός ότι, με το γεννήτορα, η ακολουθία προκύπτει *καθ' υπαγόρευση μία τιμή κάθε φορά*, σύμφωνα με τις ανάγκες. Η προσέγγιση αυτή είναι γνωστή ως ***οκνηρή αποτίμηση*** (*lazy evaluation*), και είναι πιο αποδοτική σε σχέση με τη άμεση χρήση επαναληπτικών τύπων, επειδή κάθε καινούργια τιμή δημιουργείται όποτε αυτή χρειάζεται. Για παράδειγμα, η κλήση `range(1000000)` δεν επιστρέφει μια λίστα ενός εκατομμυρίου αριθμών αλλά μια τιμή κάθε φορά, μόνο σύμφωνα με τις ανάγκες, και χωρίς να δεσμεύσει μνήμη για αποθήκευση ενός εκατομμυρίου τιμών.

Μια οπτική απεικόνιση των παραπάνω, επιχειρείται στο σχήμα 3.4 της επόμενης σελίδας. ⬇



Σχήμα 3.4 ***Γεννήτορες, επαναλήψιμοι τύποι και επαναλήπτες****. Στην Python, μπορεί κανείς να χρησιμοποιήσει ειδικευμένους τύπους για την παραγωγή και διάτρεξη τιμών.*

Εκτός από την `range()`, η Python, περιλαμβάνει και μια σειρά άλλων επαναληπτών, όπως οι `count()`, `cycle()`, `islice()` κ.λπ., ομαδοποιημένων στο δομοστοιχείο `itertools`.

Τέλος, οι γεννήτορες μπορούν να διακριθούν σε ***γεννητορικές εκφράσεις*** (*generator expressions*) και ***γεννητορικές συναρτήσεις*** (*generator fumctions*). Μπορεί κανείς να δημιουργήσει τις δικές του γεννητορικές συναρτήσεις (*βλ. 6.7*), ενώ μπορεί, επίσης, να δημιουργήσει και γεννητορικές εκφράσεις (*βλ. 3.4.4*). ■

3.3.2 Η συνάρτηση range()

Σε πολλές γλώσσες προγραμματισμού, η δομή βρόχου `for` χρησιμοποιείται παραδοσιακά στην περίπτωση ανάγκης επανάληψης ενός τμήματος κώδικα πολλές φορές, συνήθως από το 1 ως κάποιον αριθμό ή αντίθετα, με αύξηση (ή μείωση) του δείκτη του βρόχου κατά 1 ή κάποιο άλλο αριθμό. Π.χ. από 1 ως 10 με βήμα 1: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. Από 10 ως 20 με βήμα 2: 10, 12, 14, 16, 18, 20. Από 20 ως 0 με βήμα -4: 20, 16, 12, 8, 4, 0 κ.λπ. Και στην Python ισχύει το ίδιο, τα τρία προηγούμενα παραδείγματα θα μπορούσαν να κωδικοποιηθούν ως εξής:

```
for i in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:
    …
for i in [10, 12, 14, 16, 18, 20]:
    …
for i in [20, 16, 12, 8, 4, 0]:
    …
```

Κάτι τέτοιο είναι σαφώς εφικτό, αλλά άκομψο, στατικό και μη πρακτικό αν η λίστα αυτή έπρεπε να έχει π.χ. 1000 στοιχεία! Τη λύση σε αυτή την περίπτωση έρχεται να δώσει η συνάρτηση `range()`. Όπως αναφέρθηκε στο πλαίσιο Π3.3, η συνάρτηση αυτή παράγει έναν επαναλήπτη, ο οποίος μπορεί να χρησιμοποιηθεί στη θέση της παράστασης *Ε*, σε μία δομή `for`, χωρίς να χρειάζεται η δημιουργία, εκ των προτέρων, μιας άλλης επαναλήψιμης δομής. Έχει τρεις μορφές χρήσης:

range*(αρχή, τέλος, βήμα)*
range*(αρχή, τέλος)*
range*(τέλος)*

Η πρώτη είναι η πλήρης μορφή, όπου επιστρέφεται ένα εύρος τιμών (ακολουθία ακεραίων), με τιμές από '*αρχή*' (συμπεριλαμβανομένης) ως '*τέλος*' (μη συμπεριλαμβανομένου), με βήμα '*βήμα*'.

Στη δεύτερη μορφή, το '*βήμα*' παραλείπεται και θεωρείται ίσο με 1.

Στην τρίτη μορφή, παραλείπεται, επιπλέον, και η '*αρχή*', η οποία θεωρείται ίση με `0`. Κατά συνέπεια, δημιουργεί μια ακολουθία από ακεραίους, από το `0` έως τον *τέλος*-1.

Έτσι οι 3 παραπάνω εντολές for μπορούν να γραφτούν και ως εξής:

```
for i in range(1, 11, 1):
    …
for i in range(10, 21, 2):
    …
for i in range(20, -1, -4):
    …
```

Οι τρεις αυτές μορφές εκτέλεσης της for μπορούν, εναλλακτικά, να υλοποιηθούν και με τη χρήση της δομής `while`, ως εξής:

```
i = αρχή
while i < τέλος:
    …
    i += βήμα
```

Θα πρέπει να έχει, όμως, κανείς υπόψη του ότι η συνάρτηση `range()` δεν επιστρέφει λίστα, ή παρόμοια δομή. Αν είναι επιθυμητό να δημιουργηθεί λίστα μέσω της `range()`, μπορεί να χρησιμοποιηθεί η `list()` για την απαιτούμενη μετατροπή σε λίστα, η `set()` για μετατροπή σε σύνολο κ.λπ.

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> set(range(10))
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
>>>
```

Μια συνηθισμένη χρήση της `range()`, είναι η:

for j in range(len(data)):

Με την συγκεκριμένη επικεφαλίδα for μπορεί κανείς να διατρέξει όλους τους δείκτες μιας ακολουθιακής δομής, επειδή η len() μπορεί να εφαρμοστεί σε όλες τις δομές της συγκεκριμένης κατηγορίας τύπων (βλ. επόμενο κεφάλαιο). Για παράδειγμα:

```
>>> data = 'Μήλο'
>>> for j in range(len(data)):
...    print(j, data[j])
...
0 M
1 ή
2 λ
3 ο
```

Πρέπει να σημειωθεί, τέλος, ότι η range() αποτελεί έναν αποδοτικό τρόπο επανάληψης επί αριθμών, μέσω του επαναλήπτη που παράγει. Ένας βρόχος for, όμως, μπορεί να λειτουργήσει με επαναλήπτες τιμών και άλλων τύπων, όχι μόνον με αριθμητικές τιμές.

3.3.3 Οι εντολές break και continue

Αντίθετα από τη δομή while, η οποία μπορεί να τερματιστεί σε οποιαδήποτε επανάληψη εκτέλεσης του βρόχου της, αφού αυτό εξαρτάται από τη συνθήκη τερματισμού, η δομή for κανονικά τερματίζει όταν εξαντληθούν οι τιμές που μπορεί να πάρει η παράσταση *E*. Επειδή η Python δεν διαθέτει εντολή τύπου *goto*, θα πρέπει να προσφέρεται στον προγραμματιστή η δυνατότητα εξόδου από τη δομή, πριν την εξάντληση των τιμών της παράστασης *E*, αν αυτό κριθεί αναγκαίο. Ο τρόπος αυτός είναι η εντολή break, η οποία σταματάει το βρόχο, οδηγεί τη ροή του προγράμματος στην εντολή που έπεται της δομής for, (παρακάμπτεται και η ομάδα εντολών else (αν υπάρχει)). Για παράδειγμα:

Πρόγραμμα kef3_6.py: Πρώιμη έξοδος από βρόχο for.

```
01 for i in range(10):
02    if i == 4:
03        break
04    else:
05        print(i)
06 else:
07    print('Τέλος του βρόχου')
```

Ο βρόχος for στο παραπάνω πρόγραμμα θα τερματιστεί όταν το i γίνει 4 και όχι 10 όπως ορίζεται στην επικεφαλίδα:

```
0
1
2
3
```

Μπορεί να χρησιμοποιηθεί και στη δομή `while`, η ύπαρξη, όμως, της συνθήκης εξόδου δίνει τη δυνατότητα να επιτευχθεί ό ίδιος στόχος με την εντολή `break` προγραμματιστικά.

Αν χρειαστεί να παρακαμφθούν κάποιες εντολές από την ομάδα εντολών for ή την ομάδα εντολών `while` σε μια επανάληψη, τυπικά με βάση κάποια συνθήκη, και να συνεχιστεί ο βρόχος για την επόμενη τιμή της παράστασης *E*, μπορεί να χρησιμοποιηθεί η εντολή `continue`. Στο παράδειγμα που ακολουθεί, στη γραμμή `02` γίνεται έλεγχος αν η `i` έχει την τιμή `5`, περίπτωση κατά την οποία παρακάμπτεται το τύπωμά της (γραμμή `05`).

Πρόγραμμα kef3_7.py: Παράκαμψη εντολών σε βρόχο for.

```
01 for i in range(8):
02    if i == 5:
03        continue
04    else:
05        print(i)
06 else:
07    print('Τέλος του βρόχου')
```

```
0
1
2
3
4
6
7
Τέλος του βρόχου
```

Με τον ίδιο τρόπο η `continue` μπορεί να χρησιμοποιηθεί και στη δομή `while`.

3.4 Συμπεριλήψεις

Η Python περιλαμβάνει μια ιδιαίτερα ισχυρή προγραμματιστική δυνατότητα, τις *συμπεριλήψεις* (*comprehensions - σε λίστα / λεξικό / σύνολο*), ως ένα τρόπο δημιουργίας συλλογών δεδομένων με βάση μία ή περισσότερες επαναλήψιμες παραστάσεις. Στην ουσία, πρόκειται για συντακτικές δομές, οι οποίες καθιστούν δυνατό το συνδυασμό βρόχων και συνθηκών ελέγχου, μέσω μιας ιδιαίτερα συμπαγούς σύνταξης. Η χρήση τους μειώνει τον αριθμό γραμμών ενός προγράμματος, κάτι που διευκολύνει πολύ τη συντήρησή και την αναγνωσιμότητά του. Ο όρος, αλλά και η δομή της σύνταξης, παραπέμπει στη σημειολογία ορισμού συνόλων στα μαθηματικά

3.4.1 Συμπεριλήψεις σε λίστες

Για παράδειγμα, έστω ότι πρέπει να κατασκευαστεί μία λίστα με τα τετράγωνα των πρώτων 10 θετικών φυσικών αριθμών (1 έως 10). Ένας τρόπος για να γίνει αυτό είναι το παρακάτω τμήμα κώδικα:

```
>>> squares = []
>>> for i in range(1, 11):
...     squares += [i * i]
...
>>> print(squares)
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Εναλλακτικά της παραπάνω, ήδη γνωστής προσέγγισης, μπορεί να χρησιμοποιηθεί η σύνταξη ***συμπερίληψης σε λίστα*** (*list comprehension*), η οποία στην απλούστερή της μορφή έχει ως ακολούθως:

```
[ έκφραση for στοιχείο in επαναλήψιμος_τύπος ]
```

όπου *έκφραση* είναι ένας υπολογισμός ή πράξη ή ακόμα και συνάρτηση ορισμένη από το χρήστη, η οποία δρα επί της μεταβλητής *στοιχείο*. Η τελευταία παίρνει τιμές από τον *επαναλήψιμο_τύπο*. Κάθε μέλος του *επαναλήψιμου_τύπου* τίθεται, διαδοχικά, ως τιμή στο *στοιχείο* και, με βάση αυτό, υπολογίζεται μια τιμή με χρήση της έκφρασης. Τέλος, συλλέγονται και επιστρέφονται οι παραγόμενες τιμές σε μορφή λίστας.

Κατά συνέπεια, ισοδύναμα αλλά συμπαγέστερα, μπορεί να χρησιμοποιηθεί η ακόλουθη σύνταξη:

```
>>> squares = [i * i for i in range(1, 11)]
>>> print(squares)
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Φυσικά ο *επαναλήψιμος_τύπος* δεν είναι απαραίτητο να επιστρέφεται από την `range()`:

```
>>> [ x * 5 for x in [1, 2, 3, 4, 5]]
[5, 10, 15, 20, 25]
```

Επίσης, οι συμπεριλήψεις δεν περιορίζονται σε υπολογισμούς με αριθμούς:

```
>>> [ c * 3 for c in 'Όνομα']
['ΟΟΟ', 'νννν', 'ooo', 'μμμ', 'ααα']
```

Μια συμπερίληψη σε λίστα μπορεί να συμπεριλαμβάνει και συνθήκη όπως στον ακόλουθο τύπο:

```
[ έκφραση for στοιχείο in επαναλήψιμος_τύπος if συνθήκη ]
```

Αυτή η συντακτική δομή εμπεριέχει επανάληψη, φίλτρο υπό συνθήκη και επεξεργασία. Κατά συνέπεια, η περίπτωση αυτή είναι λογικά ισοδύναμη με την ακόλουθη παραδοσιακή δομή ελέγχου (η μέθοδος `append()` χρησιμοποιείται για την προσάρτηση του ορίσματος στη λίστα `result`):

```
result = []
for στοιχείο in επαναλήψιμος_τύπος:
      if συνθήκη:
            result.append(έκφραση)
```

Ως παράδειγμα, ας θεωρηθεί το ακόλουθο, κατά το οποίο δημιουργείται μια λίστα με συμπερίληψη μόνο των περιττών αριθμών από το 1 έως το 10. Ως έλεγχος συνθήκης συμπερίληψης χρησιμοποιείται το υπόλοιπο της διαίρεσης με το 2 (`k % 2`):

```
>>> odds_lc = [ k for k in range(1, 11) if k % 2 == 1 ]
>>> odds_lc
[1, 3, 5, 7, 9]
```

Στο επόμενο παράδειγμα. στην παραγόμενη λίστα συμπεριλαμβάνονται μόνο τα γράμματα 'Π','Κ','Ρ' της συμβολοσειράς 'ΑΜΠΡΑΚΑΤΑΜΠΡΑ':

```
>>> [c for c in 'ΑΜΠΡΑΚΑΤΑΜΠΡΑ' if c in ['Π','Κ','Ρ']]
['Π', 'Ρ', 'Κ', 'Π', 'Ρ']
```

Όπως μπορεί να χρησιμοποιήσει κανείς φωλιασμένους βρόχους `for`, το ίδιο μπορεί να κάνει και σε μια δομή συμπερίληψης λίστας. Στο παράδειγμα που ακολουθεί, κατά την κλασική προσέγγιση, υπάρχει μια δομή `for` φωλιασμένη σε μια δεύτερη, ώστε για κάθε `i` από το εύρος `1-3` (εξωτερικό `for`) να μεταβάλλεται το `j` για το ίδιο εύρος τιμών και να εισάγονται τα παραγόμενα ζεύγη `i,j` υπό μορφήν πλειάδας σε μια λίστα:

```
>>> pairs = []
>>> for i in range(1, 3):
...     for j in range(1, 3):
...             pairs += [ (i, j) ]
...
>>> pairs
[(1, 1), (1, 2), (2, 1), (2, 2)]
```

Εναλλακτικά, μπορεί να χρησιμοποιηθεί η ισοδύναμη συμπερίληψη σε λίστα για την παραγωγή των ίδιων ζευγών:

```
>>> pairs = [(i, j) for i in range(1, 3) for j in range(1, 3)]
>>> print(pairs)
[(1, 1), (1, 2), (2, 1), (2, 2)]
```

Αν και κάθε συμπερίληψη λίστας μπορεί να γραφεί ως βρόχος `for`, δεν ισχύει και το αντίστροφο. Σε κάθε περίπτωση, μια συμπερίληψη λίστας εκτελείται ταχύτερα από τον ισοδύναμο βρόχο (βλ. 16.2.4).

Τέλος, μπορεί να τεθεί συνθήκη (`if`) και στη μορφή συμπερίληψης σε λίστα με διπλό for. Το παράδειγμα που ακολουθεί αποτελεί τροποποίηση του προηγούμενου, ώστε, μέσω ελέγχου συνθήκης ισότητας, να επιστρέφονται μόνο τα ζεύγη με ίδια μέλη:

```
>>> pairs = [(i,j) for i in range(1,3) for j in range(1,3) if i==j]
>>> print(pairs)
[(1, 1), (2, 2)]
```

Αν και οι συμπεριλήψεις σε λίστα συνιστούν δομή διαδικασιακού τύπου, μπορούν να υποκαταστήσουν αποδοτικότερα δομές του συναρτησιακού υποδείγματος προγραμματισμού (βλ. κεφ. 6).

3.4.2 Συμπεριλήψεις σε λεξικά

Η Python υποστηρίζει, επίσης, συμπεριλήψεις σε λεξικά (*dictionary comprehensions)*.

Η απλούστερη σύνταξη έχει ως ακολούθως:

{*έκφραση_κλειδιού* : *έκφραση_τιμής* **for** *έκφραση* **in** *επαναλήψιμος_τύπος*}

Στο παρακάτω παράδειγμα, κατασκευάζεται ένα λεξικό που αντιστοιχίζει καθέναν από τους αριθμούς από 1 έως 8, στο τετράγωνό του συν 1:

```
>>> squares = {i: i * i + 1 for i in range(1, 9)}
>>> print(squares)
{1: 2, 2: 5, 3: 10, 4: 17, 5: 26, 6: 37, 7: 50, 8: 65}
```

Ως ένα πιο σύνθετο παράδειγμα, ας θεωρηθεί αυτό της δημιουργίας ενός λεξικού από δύο λίστες, με χρήση συμπερίληψης σε λεξικό· η πρώτη λίστα περιέχει τα κλειδιά και η δεύτερη τις αντίστοιχες τιμές:

```
>>> keys = [“Μαρία“, “Γιάννης“, “Σπύρος“]
>>> values = [23, 45, 18]
>>> { keys[i] : values[i] for i in range(len(keys)) }
{‘Σπύρος’: 18, ‘Μαρία’: 23, ‘Γιάννης’: 45}
```

Παρόμοια με τις συμπεριλήψεις σε λίστες, μπορούν να χρησιμοποιηθούν έλεγχοι συνθήκης με `if` καθώς και πολλαπλά `for`.

3.4.3 Συμπεριλήψεις σε σύνολα

Εναλλακτικά της δημιουργίας συνόλων με εκχώρηση αντίστοιχου κυριολεκτήματος σε μια μεταβλητή, ή τη χρήση της συνάρτησης `set()` -βλ. μεθεπόμενο κεφάλαιο- μπορεί κανείς να δημιουργήσει σύνολα, με αντίστοιχες συμπεριλήψεις (*set comprehensions*). Προς το σκοπό αυτό, μπορεί να χρησιμοποιηθεί η σύνταξη:

{ *έκφραση* **for** *στοιχείο* **in** *επαναλήψιμος_τύπος* }

Σύμφωνα με τη σύνταξη αυτή, το παράδειγμα παραγωγής λίστας με τα τετράγωνα των πρώτων 10 θετικών φυσικών αριθμών (1 έως 10), στην έκδοσή του για παραγωγή συνόλου (συμπερίληψη σε σύνολο) έχει ως ακολούθως:

```
>>> squares = {x * x for x in range(1, 11)}
>>> print(squares)
{1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
```

Αξίζει να παρατηρηθεί η ομοιότητα με την αντίστοιχη μαθηματική σημειολογία για τον ορισμό συνόλων, $\{ x^2 : x \in S\}$ με το `for` να παίζει το ρόλο του : (όπου), του `in` το ρόλο του $\in$ (ανήκει), και του επαναλήψιμου τύπου το ρόλο του S.

Και εδώ, μπορεί να χρησιμοποιηθεί συνθήκη `if`.

3.5 Γεννητορικές εκφράσεις

Ενδεχομένως να έχει υποθέσει ο αναγνώστης με βάση τα προηγούμενα ότι, αντικαθιστώντας τις αγκύλες με παρενθέσεις, θα μπορέσει κανείς να αναπτύξει συμπεριλήψεις σε πλειάδες. Στην πραγματικότητα, ο συγκεκριμένος συντακτικά όμοιος τρόπος, δεν μπορεί να χρησιμοποιηθεί άμεσα για δημιουργία πλειάδων. Πράγματι, κατ᾽ αναλογία με τα προηγούμενα παραδείγματα η επόμενη έκφραση:

```
>>> (i * i for i in [1, 2, 3])
<generator object <genexpr> at 0x023BA3C0>
```

αναφέρεται ως ***γεννητορική έκφραση*** (*generator expression*)· δημιουργεί δε μία δομή η οποία ονομάζεται *γεννήτορας* (*generator - βλ. πλαίσιο Π3.3 πριν*). Από αυτήν μπορεί να προκύψει ένας επαναλήπτης:

```
>>> g_iter = (i * i for i in [1, 2, 3])
>>> next(g_iter)
1
>>> next(g_iter)
4
>>> next(g_iter)
9
>>> next(g_iter)
Traceback (most recent call last):
...
StopIteration
```

Μπορεί, επίσης, να χρησιμοποιηθεί για τη δημιουγία επαναλήψιμων τύπων, με τη βοήθεια της αντίστοιχης συνάρτησης μετατροπής τύπου:

```
>>> g_iter = (i * i for i in [1, 2, 3])
>>> g_tuple = tuple(g_iter)
>>> g_tuple
(1, 4, 9)
```

Μια γεννητορική έκφραση μπορεί να χρησιμοποιθεί μία μόνο φορά. Έτσι, σε συνέχεια του προηγούμενου παραδείγματος:

```
>>> g_list = list(g_iter)
>>> g_list
[]
```

αλλά,

```
>>> g_iter = (i * i for i in [1, 2, 3])
>>> g_list = list(g_iter)
>>> g_list
[1, 4, 9]
```

Η σύνταξη γεννητορικής έκφρασης είναι ιδιαίτερα καλή επιλογή, όταν τα αποτελέσματα δεν χρειάζεται να αποθηκευτούν στη μνήμη.

Περίληψη

Οι λογικές συνθήκες και ο τρόπος σχηματισμού τους αποτελούν θεμελιώδες στοιχείο στην εκμάθηση μιας γλώσσας προγραμματισμού. Αποτελούν τη βάση για το σχηματισμό δομών ελέγχου της ροής των εντολών ενός προγράμματος. Μέσω των εντολών ελέγχου ροής υλοποιείται η βασική λογική δομή κάθε προγράμματος σε έναν υπολογιστή. Η Python, πέρα από τις εντολές που υλοποιούν τις βασικές δομές ελέγχου ροής, περιλαμβάνει μια σειρά από επιπλέον σχετικές συντακτικές δομές μέσω των οποίων μπορεί να εξοικονομηθεί πλήθος εντολών καταλήγοντας σε συμπαγέστερα και πιο ευανάγνωστα προγράμματα.

Λέξεις κλειδιά

- ***Λογική έκφραση*** *(logical ή boolean expression).* Έκφραση που υπολογίζεται πάντα σε μια από τις τιμές `True` ή `False`.
- ***Σχεσιακός τελεστής*** (*relational operator*). Τελεστές που χρησιμοποιούνται για έλεγχο ισότητας και συγκρίσεις (==, !=, >, <, >= και <=).
- ***Λογικοί τελεστές*** *(logical operator).* Τελεστές μέσω των οποίων συνδυάζονται λογικές εκφράσεις (and, or, not).
- ***Λογική συνθήκη*** (*condition*). Λογική έκφραση σε δομή επιλογής ή βρόχου με βάση την οποία επιλέγεται ομάδα εντολών για εκτέλεση ή πραγματοποιείται ο έλεγχος τερματισμού επαναλήψεων σε δομές βρόχου.
- ***Πρόταση*** (*clause*). Εντολή που αποτελείται από μια επικεφαλίδα η οποία αρχίζει με μια δεσμευμένη λέξη της Python και τερματίζεται με άνω κάτω τελεία (:) και ένα σώμα εντολών ομοιόμορφα εδαφιοποιημένο ως προς την επικεφαλίδα.

- Σ***ύνθετη εντολή*** **(***compound statement*). Εντολή που αποτελείται από μία ή περισσότερες προτάσεις, όπως οι δομές ελέγχου ροής. Σύνθετες εντολές, συντακτικά, θεωρούνται και οι ορισμοί συναρτήσεων και κλάσεων.
- ***Φώλιασμα*** (*nesting*). Δομή ελέγχου εντός άλλης δομής ελέγχου.
- *Επαναλήπτης (iterator).* Αντικείμενο (μεταβλητή), το οποίο χρησιμοποιείται για τη διαχείριση επαναλήψεων, μέσω μιας σειράς τιμών που λαμβάνει.
- ***Επαναλήψιμοι τύποι*** (*iterables*). Τύποι δεδομένων, στους οποίους μπορεί να διατρέξει κανείς όλα τα στοιχεία τους και να αποτελέσουν τη βάση για δημιουργία επαναληπτών.
- ***Συμπεριλήψεις*** (*comprehensions*). Συμπαγής τρόπος δημιουργίας συγκεκριμένων δομών δεδομένων, με βάση μία ή περισσότερες επαναλήψιμες παραστάσεις

Ασκήσεις

1. Σε τι τιμή νομίζετε ότι θα υπολογιστούν οι ακόλουθες εκφράσεις:

```
(α) not 10   (β) not ‘‘   (γ) -55 == True     (δ) ‘’ == False
```

Δικαιολογήστε την απάντησή σας.

2. Οι ακόλουθες δύο δομές `if` είναι συντακτικά ορθές;

```
(α)    if -55:              (β) if ‘‘:
           print(‘ok’)              print(‘ok’)
       else:                    else:
           print(‘not ok’)          print(‘not ok’)
```

Αν ναι, ποιο μήνυμα νομίζετε ότι θα τυπωθεί σε κάθε περίπτωση;

3. Γράψτε ένα πρόγραμμα, το οποίο να τυπώνει τα γράμματα της αλφαβήτου και, δίπλα σε κάθε γράμμα, το χαρακτηρισμό **“**Φωνήεν**“** ή **“**Σύμφωνο**“**, ανάλογα.

4. Γράψτε ένα πρόγραμμα το οποίο να τυπώνει τα φωνήεντα που περιέχονται σε μια λέξη.

5. Γράψτε ένα πρόγραμμα το οποίο θα παίρνει ως είσοδο ένα εύρος ετών (έτος αρχής - έτος τέλους) και θα τυπώνει τα δίσεκτα έτη που περιλαμβάνονται στο δοθέν εύρος. Υλοποιήστε το με δύο τρόπους. (*Ένα έτος είναι δίσεκτο αν διαιρείται διά 4. Αν όμως διαιρείται επίσης ακριβώς και δια 100, τότε δεν είναι δίσεκτο, εκτός και αν διαιρείται, επίσης, δια 400, οπότε είναι*).

6. Γράψτε μια συμπερίληψη σε λεξικό που λαμβάνει, ως είσοδο, μια λέξη και θα επιστρέφει ένα λεξικό με ζεύγη της μορφής χαρακτήρας - πλήθος στη λέξη. (*Υπόδειξη: η μέθοδος* `s.count(χαρακτήρας)` *επιστρέφει το πλήθος των υπάρξεων του χαρακτήρα στην συμβολοσειρά* `s`).

7. Γράψτε ένα πρόγραμμα, το οποίο θα τυπώνει τους χαρακτήρες μιας συμβολοσειράς, με αντίστροφη διάταξη από την αρχική. Π.χ. για τη συμβολοσειρά 'Για' θα τυπώνει:

```
α
ι
Γ
```

8. Δημιουργήστε ένα λεξικό από δύο λίστες, όπως στο παράδειγμα της σελίδας 116, χρησιμοποιώντας ένα ισοδύναμο βρόχο for.

9. Δημιουργήστε την ακόλουθη έξοδο στην Python:

```
....1
...2
..3
.4
5
```

10. Γράψτε ένα πρόγραμμα που να ζητάει έναν αριθμό από το 1 έως το 7, και να τυπώνει την αντίστοιχη μέρα της εβδομάδας. Το πρόγραμμα πρέπει να ελέγχει αν ο αριθμός είναι στο συγκεκριμένο εύρος, και να τυπώνει αντίστοιχο μήνυμα αν δεν είναι.

11. Τροποποιήστε το προηγούμενο πρόγραμμα ώστε να επαναλαμβάνεται, μέχρι ο χρήστης να δώσει το χαρακτήρα 'q', οπότε θα τερματίζεται.

12. Γράψτε ένα πρόγραμμα που θα τυπώνει τον πίνακα της προπαίδειας.

13, Τι παράμετροι πρέπει να δοθούν στην range() για προκύψουν τα ακόλουθα:

```
0,1,2,3,4,5,6
1,3,5,7,9
10,8,6,4,2
-3,-2,-1,0,1,2,3
```

14. Γράψτε ένα πρόγραμμα το οποίο θα τυπώνει όλες τις δυνατές μεταθέσεις τριών χαρακτήρων, έστω των ΑΒΓ.

15. Με βάση τη λίστα

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

δημιουργήθηκε, μέσω γεννητορικής έκφρασης η

```
lazy_doubles = (2 * x for x on Tree_numbers)
```

Στη συνέχεια, εκτελέστηκαν κάποιες εντολές και κάποια στιγμή δόθηκε η:

```
list(lazy_doubles)
[4,6]
```

Ποια ήταν, σίγουρα, μια από τις εντολές που παρεμβλήθηκαν ;

15. Γράψτε ένα πρόγραμμα, το οποίο θα τυπώνει τους ακέραιους, μεταξύ των 100 και 500 συμπεριλαμβανομένων, που διαιρούνται ακριβώς με το 5 και το 7.

16. Γράψτε ένα πρόγραμμα, το οποίο, για τους ακέραιους από το 1 έως το 40, όποτε κάποιος από αυτούς είναι πολλαπλάσιο του 3, θα τυπώνει 'Γκλιν' αντί για τον αριθμό, ενώ όποτε είναι πολλαπλάσιο του 5 θα τυπώνει 'Γκλον' αντί για τον αριθμό. Αν ο εξεταζόμενος αριθμός είναι πολλαπλάσιο και των δύο, θα τυπώνει 'Γκλιν - Γκλον'. Π.χ.

```
1
2
Γκλιν
4
Γκλον
...
14
Γκλιν - Γκλον
...
```

17. Γράψτε ένα πρόγραμμα το οποίο θα τυπώνει τους αριθμούς από το 100 έως το 500, των οποίων όλα τα ψηφία είναι άρτιοι.