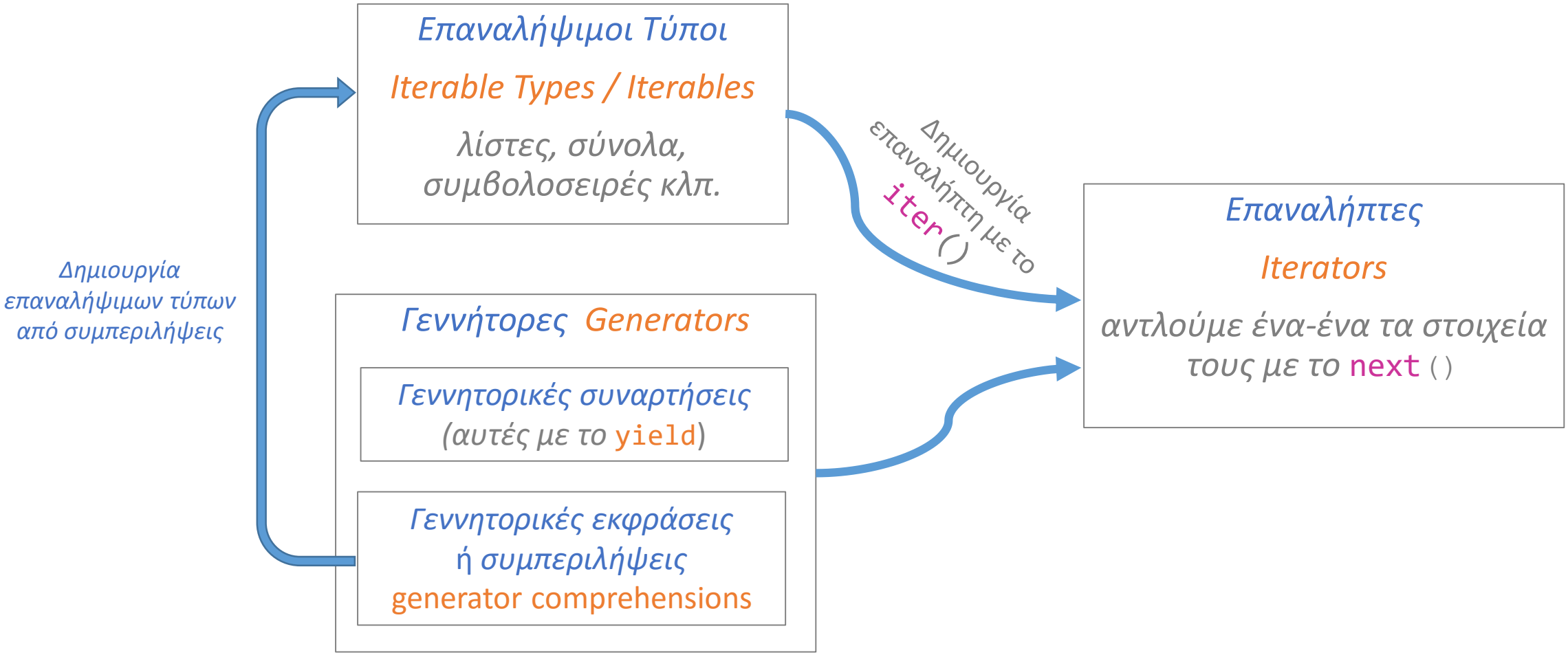
Ελληνικό Μεσογειακό Πανεπιστήμιο
Σχολή Μηχανικών / Τμήμα Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών

HMY01K06
Επιστημονικός Προγραμματισμός με Python

Διάλεξη Ένατη
Επαναλήψιμοι τύποι / Επαναλήπτες
Γεννήτορες / Συμπεριλήψεις

Φθινόπωρο 2025

Στο σημερινό μάθημα θα μιλήσουμε για...



Η έννοια του επαναλήπτη στο φυσικό μας κόσμο

Ο *επαναλήπτης* δρα πάνω στον *επαναλήψιμο τύπο* και καθορίζει *ποιο είναι το επόμενο στοιχείο του κάθε φορά*

Επαναλήψιμοι τύποι και Επαναλήπτες

Μια πρώτη ματιά του τι θα μάθουμε σήμερα

Ο μηχανισμός πίσω από τις δομές επανάληψης της Python βασίζεται στις έννοιες του *επαναλήψιμου τύπου* (iterable) και του *επαναλήπτη* (iterator)

Επαναλήψιμοι τύποι (iterables) είναι αντικείμενα της Python που μπορούν να επιστρέφουν τα στοιχεία τους *ένα-ένα*, *με μια συγκεκριμένη σειρά* -ή και χωρίς, όταν τους ζητηθεί *(παραδειγματα: λίστες, πλειάδες, συμβολοσειρές, σύνολα)*

Επαναλήπτες (iterators) είναι αντικείμενα που δρουν πάνω σε επαναλήψιμους τύπους και προσδιορίζουν *το πώς ακριβώς υλοποιείται μια επανάληψη*, δηλαδή πιο συγκεκριμένα *ποιο είναι το επόμενο στοιχείο* κάθε φορά

Για να δημιουργήσουμε εναν *επαναλήπτη* εφαρμόζουμε τη συνάρτηση `iter()` πάνω σε ένα επαναλήψιμο τύπο. Στη συνέχεια *χρησιμοποιούμε* διαδοχικά την συνάρτηση `next()` για να διατρέξουμε τα στοιχεία του

Όλες οι δομές επανάληψης της Python όπως το `for` βασίζονται στον μηχανισμό (`iter`/`next`) που είναι *"κρυμμένος"* πίσω από το πιο εύχρηστο συντακτικό τους

Επαναλήψιμοι τύποι (iterables)

- *Επαναλήψιμοι τύποι* (iterables) είναι αντικείμενα της Python που μπορούν να επιστρέφουν τα στοιχεία τους *ένα-ενα*, *με μια συγκεκριμένη σειρά* - ή και χωρίς, όταν τους ζητηθεί *(κάτι που συνήθως γίνεται μέσα από μια δομή επανάληψης)*

  π.χ. `for x in 'Dog': print(x)` θα εκτυπώσει
  ```
  D
  o
  g
  ```

- Οι βασικότεροι επαναλήψιμοι τύποι της Python είναι οι *συμβολοσειρές*, οι *λίστες*, οι *πλειάδες*, τα *λεξικά* και τα *αρχεία*. Οι τύποι αυτοί έχουν *σειριακή δομή* (τα στοιχεία τους είναι *διατεταγμένα* σε μια ακολουθία) οπότε και η *σειρά πρόσβασης* των στοιχείων τους μέσα σε ένα for loop είναι σειριακή *(πρώτο στοιχείο, δεύτερο…)*

- Παρόλο που τα *σύνολα* δεν έχουν σειριακή δομή, αποτελούν και αυτά επαναλήψιμο τύπο. Εδώ όμως η *σειρά πρόσβασης* των στοιχείων τους μέσα σε ένα for loop ενδέχεται να είναι *τυχαία κάθε φορά*

  π.χ. `for x in {1,2,3}: print(x)` πιθανόν να εκτυπώσει `1,3,2` ή `3,1,2` ή …

Συναρτήσεις που δρουν πάνω σε επαναλήψιμους τύπους

Συναρτήσεις δημιουργίας / μετατροπής επαναλήψιμων τύπων *(ήδη γνωστές)*

`list()`, `tuple()`, `dict()`, `set()`

- δημιουργούν μια λίστα, πλειάδα… από το *περιεχόμενο* ενός επαναλήψιμου τύπου
- αποτελούν έναν *έμμεσο τρόπο μετατροπής* από έναν επαναλήψιμο τύπο σε έναν άλλο

  αν `s = {1,2,3}` τότε το `list(s)` επιστρέφει την λίστα `[2,1,3]`

  αν `L = [1,2,3]` τότε το `tuple(L)` επιστρέφει την πλειάδα `(1,2,3)`

  αν `t = "Hello"` τότε το `set(t)` επιστρέφει το σύνολο `{'o', 'e', 'H', 'l'}`

  η `dict([(1,'apple'), (2,'ball')])` επιστρέφει το λεξικό `{1: 'apple', 2: 'ball'}`

- ειδικά για τα *λεξικά*, κατά την μετατροπή τους σε άλλο επαναλήψιμο τύπο πρέπει να αποσαφηνίσουμε αν ο τύπος αυτός θέλουμε να σχηματιστεί από τα *κλειδιά* ή από τις *τιμές* τους

  αν `d = { 'A':100,'B':200, 'C':300}`

  τότε το `list(d)` επιστρέφει την λίστα `['A','B','C']` *λίστα κλειδιών του λεξικού*

  ενώ το `list(d.values())` επιστρέφει την λίστα `[100,200,300]` *λίστα τιμών του λεξικού*

Συναρτήσεις που δρουν πάνω σε επαναλήψιμους τύπους

- `sum()`: *αθροίζει* και επιστρέφει το περιεχόμενο ενός επαναλήψιμου τύπου
- `max()/min()`:επιστρέφει την *μεγαλύτερη/μικρότερη* τιμή ενός επαναλήψιμου τύπου
- `sorted()`:επιστρέφει *λίστα* με το *περιεχόμενο* ενός επαναλήψιμου τύπου *ταξινομημένο*

αν `t = (5, 2, 9, 4, -6, 7)` τότε

η `sum(t)` επιστρέφει `21`, η `min(t)` επιστρέφει `-6`, η `max(t)` επιστρέφει `9`

η `sorted(t)` επιστρέφει τη *λίστα* `[-6, 2, 4, 4, 5, 7, 9]`

Αν η συγκεκριμένη πράξη *δεν έχει νόημα* στα δεδομένα του επαναλήψιμου τύπου, τότε εγείρεται *εξαίρεση σφάλματος*

έτσι πχ. η `max([1,3,'hello'])` θα εγείρει την εξαίρεση σφάλματος

```
TypeError: '>' not supported between instances of 'str' and 'int'
```

Συναρτήσεις που δρουν πάνω σε επαναλήψιμους τύπους

Οι συναρτήσεις `any()` και `all()`

Πριν δούμε τι ακριβώς κάνουν οι δυο παραπάνω συναρτήσεις πρέπει να μάθουμε μια άλλη βασική συνάρτηση της Python η οποία ονομάζεται `bool()`

Η συνάρτηση `bool(x)` εφαρμόζεται σε οποιοδήποτε αντικείμενο x της Python και επιστρέφει την τιμή `False` όταν:

- το x είναι *κενός επαναλήψιμος τύπος* π.χ. `[ ]`, `( )`, `''`, `{ }`, `set()`, `range(0)`
- το x είναι το *μηδέν* σε οποιαδήποτε μορφή όπως: `0` `0.0` `0.0E+3` `0/17`
- το x είναι ένα από τα δυο κυριολεκτήματα *(literals)* `False` ή `None`

Σε κάθε άλλη περίπτωση η συνάρτηση `bool(x)` επιστρέφει την τιμή `True`

Γενικά μπορούμε να πούμε ότι η `bool(x)` επιστρέφει `True` όταν το αντικείμενο x είναι *"μη τετριμμένο"* και `False` στην αντίθετη περίπτωση

Συναρτήσεις που δρουν πάνω σε επαναλήψιμους τύπους

Η συνάρτηση `any()` διατρέχει τα στοιχεία ενός επαναλήψιμου τύπου μέχρι να βρει το πρώτο στοιχείο x για το οποίο η συνάρτηση `bool(x)` επιστρέφει `True`

Στην περίπτωση αυτή διακόπτεται άμεσα ο έλεγχος και η `any()` επιστρέφει `True`

Στην αντίθετη περίπτωση *(όταν δηλαδή το* `bool(x)` *είναι* `False` *για όλα τα στοιχεία του επαναλήψιμου τύπου)* η συνάρτηση `any()` επιστρέφει `False`

Παραδείγματα:

`any([1,2,0])` επιστρέφει `True`

`any({})` επιστρέφει `False`

`any([0,0,0])` επιστρέφει `False`

`any({'', '', (1,3)})` επιστρέφει `True`

`any([{}, False, None, 0/100])` επιστρέφει `False`

`any(7)` επιστρέφει… `TypeError: 'int' object is not iterable`

Η `any()` απαντά ουσιαστικά στην ερώτηση: "Περιέχει ο επαναλήψιμος τύπος *τουλάχιστον ένα μη τετριμμένο στοιχείο*;"

ενώ `any([0,0,[0]])` επιστρέφει `True` *(γιατί;)*

Συναρτήσεις που δρουν πάνω σε επαναλήψιμους τύπους

Η συνάρτηση `all()` διατρέχει τα στοιχεία ενός επαναλήψιμου τύπου μέχρι να βρει το πρώτο στοιχείο x για το οποίο η συνάρτηση `bool(x)` επιστρέφει `False`

Στην περίπτωση αυτή διακόπτεται άμεσα ο έλεγχος και η `all()` επιστρέφει `False`

Στην αντίθετη περίπτωση *(όταν δηλαδή το* `bool(x)` *είναι* `True` *για όλα τα στοιχεία του επαναλήψιμου τύπου)* η συνάρτηση `all()` επιστρέφει `True`

Ειδικά αν ο επαναλήψιμος τύπος είναι κενός, τότε η `all()` επιστρέφει `True`

Παραδείγματα:

```
all([1,2,0,4]) επιστρέφει False
all({}) επιστρέφει True (γιατί;)
all({'Hello', '0', (1,3)}) επιστρέφει True
all({'Hello', 0, (1,3)}) επιστρέφει False (γιατί;)
all(4) επιστρέφει TypeError: 'int' object is not iterable
```

Η `all()` απαντά ουσιαστικά στην ερώτηση: "Είναι *όλα* τα στοιχεία του επαναλήψιμου τύπου *μη τετριμμένα*;"

Συναρτήσεις που δρουν πάνω σε επαναλήψιμους τύπους

`enumerate()`: δρα πάνω σε ένα επαναλήψιμο τύπο και επιστρέφει ένα *νέο επαναλήψιμο αντικείμενο*, κάθε στοιχείο του οποίου είναι μια *πλειάδα δυο στοιχείων* `(i,x)` όπου το `i` είναι ο αύξων αριθμός του στοιχείου `x` ξεκινώντας από το `0`

- για να μπορέσουμε να *αξιοποιήσουμε* τα στοιχεία του νέου αυτού επαναλήψιμου αντικειμένου πρέπει *είτε* α) να τα *απαριθμήσουμε μέσα σε ένα* `for`, *είτε* β) να *μετατρέψουμε* το αντικείμενο αυτό σε *λίστα* ή *πλειάδα*

```
for x in enumerate('test'):
    print(x)
```

θα εμφανίσει

```
(0, 't')
(1, 'e')
(2, 's')
(3, 't')
```

```
>>> e = enumerate([100, 500, 300])
>>> e
<enumerate object at 0x0308F4E0>
>>> list(e)
[(0, 100), (1, 500), (2, 300)]
>>> dict(e)
{}
```

και όχι `{0: 100, 1: 500, 2: 300}` όπως θα περιμέναμε.
Θα δούμε αργότερα το γιατί *(εξάντληση επαναλήπτη)*

Συναρτήσεις που δρουν πάνω σε επαναλήψιμους τύπους

`zip()`: δρα πάνω σε δυο *(ή και περισσότερους)* επαναλήψιμους τύπους και επιστρέφει ένα *νέο επαναλήψιμο τύπο*, που αποτελείται από πλειάδες ως εξής:

Η *πρώτη πλειάδα* περιέχει τα *πρώτα στοιχεία* των δυο *(ή και περισσότερων)* αρχικών τύπων, η *δεύτερη πλειάδα* τα *δεύτερα*, κοκ. μέχρι να *εξαντληθούν* τα στοιχεία του *μικρότερου σε μήκος* επαναλήψιμου τύπου *(εφόσον αυτοί έχουν διαφορετικά μήκη)*

- ιδιαίτερα χρήσιμη συνάρτηση για το *ζευγάρωμα* (pairing) αντίστοιχων στοιχείων ανάμεσα σε επαναλήψιμους τύπους

```
>>> names = ["Alice", "Bob", "Carol", "David"]
>>> exam_1_scores = [90, 82, 79, 87]
>>> exam_2_scores = [95, 84, 72, 91, 47]   το στοιχείο 47 περισσεύει

>>> zip(names, exam_1_scores, exam_2_scores)
<zip at 0x20de1082608>

>>> list(zip(names, exam_1_scores, exam_2_scores))
[('Alice', 90, 95), ('Bob', 82, 84), ('Carol', 79, 72), ('David', 87, 91)]
```

Ένα 'τρυκ' της Python πάνω στους επαναλήψιμους τύπους

Επαναλήψιμοι τύποι μπορούν να αναλύονται (*unpack*) ώστε
να εκχωρούνται απ' ευθείας σε απλούστερα αντικείμενα

Δείτε τα επόμενα παραδείγματα

```
>>> my_list = [7, 'Q', (1,2)]
>>> x, y, z = my_list
>>> print(x, y, z)
7 Q (1,2)
```

```
>>> x, y = input(">").split()
```

Ο χρήστης έχει τη δυνατότητα να εισάγει ταυτόχρονα περισσότερα από ένα δεδομένα *(στην συγκεκριμένη περίπτωση δυο* που χωρίζονται μεταξύ τους με κενό)

```
>>> grades = [("Alice", 93), ("Bob", 95), ("Carol", 84)]
```

```
>>> for x in grades:
    print(x)

('Alice', 93)
('Bob', 95)
('Carol', 84)
```

```
>>> for name, grade in grades:
    print(name, grade)

Alice 93
Bob 95
Carol 84
```

Πρέπει να υπάρχει ακριβής αντιστοιχία ανάμεσα στο μήκος του επαναλήψιμου τύπου
και το πλήθος των στοιχείων στα οποία αναλύεται

Επαναλήπτες (iterators)

Αντικείμενα της Python τα οποία προκύπτουν ως αποτέλεσμα της εφαρμογής της συνάρτησης `iter()` πάνω σε ένα *επαναλήψιμο τύπο* *(ή όπως θα δούμε αργότερα, σε έναν γεννήτορα)*

Επαναλήψιμος τύπος

```
i = iter('Hello')
```

Επαναλήπτης

δημιουργεί τον *επαναλήπτη* **i** πάνω στον *επαναλήψιμο τύπο* `'Hello'`

Χρησιμοποιούνται για να μας δίνουν *ένα-ένα* τα στοιχεία ενός επαναλήψιμου τύπου κάθε φορά που εφαρμόζουμε πάνω τους *(δηλ. στον επαναλήπτη)* τη συνάρτηση `next()`

Στο παράδειγμά μας, η *πρώτη* κλήση της συνάρτησης `next(i)` θα επιστρέψει `'H'`, η *δεύτερη* κλήση της θα επιστρέψει `'e'`, η *τρίτη* κλήση θα επιστρέψει `'l'` κοκ.

Όταν *εξαντληθούν / καταναλωθούν* όλα τα στοιχεία του επαναλήψιμου τύπου, τότε *κάθε επόμενη κλήση* του `next(i)` εγείρει την εξαίρεση `StopIteration`

Όλες οι επαναλήψεις της Python υλοποιούνται εσωτερικά με τον μηχανισμό `iter()/next()`.
Το `for` είναι ενας πιο ***φιλικός και κομψός*** εναλλακτικός τρόπος δημιουργίας επαναλήψεων, γιαυτό και χρησιμοποιείται σχεδόν πάντα στη θέση των `iter()/next()`

Παραδείγματα δημιουργίας και χρήσης επαναληπτών

Υλοποιούνται όπως είπαμε με χρήση των ενσωματωμένων συναρτήσεων `iter()` και `next()`

Δημιουργία ενός επαναλήπτη ***i*** *από τα στοιχεία μιας λίστας*

```
>>> i = iter([2,0,2,3])
>>> print(next(i))
2
>>> print(next(i))
0
>>> print(next(i))
2
>>> print(next(i))
3
>>> print(next(i))
Traceback (most recent call last):
  File "<pyshell#34>", line 1, in <module> next(i)
StopIteration
```

Χρήση του επαναλήπτη `i`

Δημιουργία ενός επαναλήπτη ***c*** *από τα στοιχεία μιας συμβολοσειράς*

```
>>> c = iter("Test")
>>> print(next(c))
'T'
>>> print(next(c))
'e'
>>> print(next(c))
's'
>>> print(next(c))
't'
>>> print(next(c))
Traceback (most recent call last):
  File "<pyshell#34>", line 1, in <module> next(c)
StopIteration
```

Χρήση του επαναλήπτη `c`

Πως υλοποιείται το *for* στην Python

Με αυτά που μάθαμε, μπορούμε τώρα πια να δούμε πως υλοποιείται ο "κρυμμένος" μηχανισμός επαναλήψεων πίσω από την δομή `for` της Python.

```
for <στοιχείο> in <επαναλήψιμος τύπος>:
    print(<στοιχείο>)
    μπλοκ του κώδικα for
```

Γράφοντας αυτό…

```
<επαναλήπτης> = iter(<επαναλήψιμος τύπος>)
while True:
    try:
        <στοιχείο> = next(<επαναλήπτης>)
    except StopIteration:
        break    # Ο επαναλήπτης εξαντλήθηκε, βγες από την ανακύκλωση
    else:
        print(<στοιχείο>)
        μπλοκ του κώδικα for
```

Στην πραγματικότητα υλοποιείται αυτό

Ένας επαναλήπτης ανά πάσα στιγμή *"γνωρίζει"* την τρέχουσα κατάστασή του, δηλαδή *σε ποιό σημείο της επανάληψης βρίσκεται* και άρα *τι θα επιστρέψει την επόμενη φορά* που θα κληθεί μέσα από τη συνάρτηση next()

Πως υλοποιείται το *for* στην Python

Παράδειγμα: Τα επόμενα δυο κομμάτια κώδικα είναι απολύτως ισοδύναμα

```
L = [1,2,3,4]
for x in L:
    print(x)
```

Υλοποίηση με for
(πολύ πιο απλή και φιλική)

```
L = [1,2,3,4] # επαναλήψιμος τύπος
i= iter(L) # δημιουργία επαναλήπτη από επαναλήψιμο τύπο
while True:   # άπειρη ανακύκλωση. βγαίνει μόνο με break
    try:
        x= next(i) #επόμενο στοιχείο επαναλήψιμου τύπου
    except StopIteration:
        break
    else:
        print(x)
```

Υλοποίηση με iterator
(έτσι μεταφράζεται το for εσωτερικά στην Python)

Και τα δυο κομμάτια κώδικα θα εμφανίσουν ένα-προς-ένα τα στοιχεία της λίστας `[1,2,3,4]`

Γεννήτορες (generators)

Αντικείμενα της Python που *παράγουν επαναλήπτες με έμμεσο τρόπο* καθώς περιέχουν *οδηγίες* για τη παραγωγή τους

Υπάρχουν δυο ειδών γεννήτορες

- *Γεννητορικές συναρτήσεις:* Ειδικές συναρτήσεις της Python οι οποίες εκτελούνται *τμηματικά* παράγουν τα στοιχεία του επαναλήπτη που υλοποιούν *ένα κάθε φορά*

  Παράδειγμα γεννητορικής συνάρτησης είναι η `range()` που ήδη γνωρίζουμε

- *Γεννητορικές εκφράσεις:* Απλούστερος συντακτικά τρόπος για τον ορισμό *απλών επαναληπτών* σε μία μόνο γραμμή κώδικα

Γεννήτορες (generators)

Παράδειγμα γεννήτορα που ήδη γνωρίζουμε και χρησιμοποιούμε:

Η γεννητορική συνάρτηση `range()` η οποία όταν καλείται παράγει έναν επαναλήπτη που στη συνέχεια τον χρησιμοποιούμε συνήθως μέσα σε ένα for

Επαναλήπτης από *επαναλήψιμο τύπο*

```
>>> i = iter([0,1,2])
>>> next(i)
0
>>> next(i)
1
>>> next(i)
2
>>> next(i)
Traceback (most recent call last):
next(i)  StopIteration
```

Ισοδύναμος επαναλήπτης από *γεννητορική συνάρτηση*

```
>>> i = iter(range(3))
>>> next(i)
0
>>> next(i)
1
>>> next(i)
2
>>> next(i)
Traceback (most recent call last):
next(i) StopIteration
```

Ουσιαστική διαφορά ενός επαναλήπτη που προέρχεται από γεννήτορα από έναν άλλο που προέρχεται από επαναλήψιμο τύπο

- Ενας επαναλήπτης `i` που δημιουργείται από ένα *επαναλήψιμο τύπο* *(πχ μια λίστα)* *διατηρεί όλα τα στοιχεία του επαναλήψιμου τύπου αποθηκευμένα στη μνήμη*, και κάθε φορά που καλείται με το `next(i)` επιστρέφει το *επόμενο στοιχείο* της επανάληψης από *την αντίστοιχη θέση αποθήκευσής του στη μνήμη*

Περιέχει όλα τα χαρτάκια εκτυπωμένα μέσα του

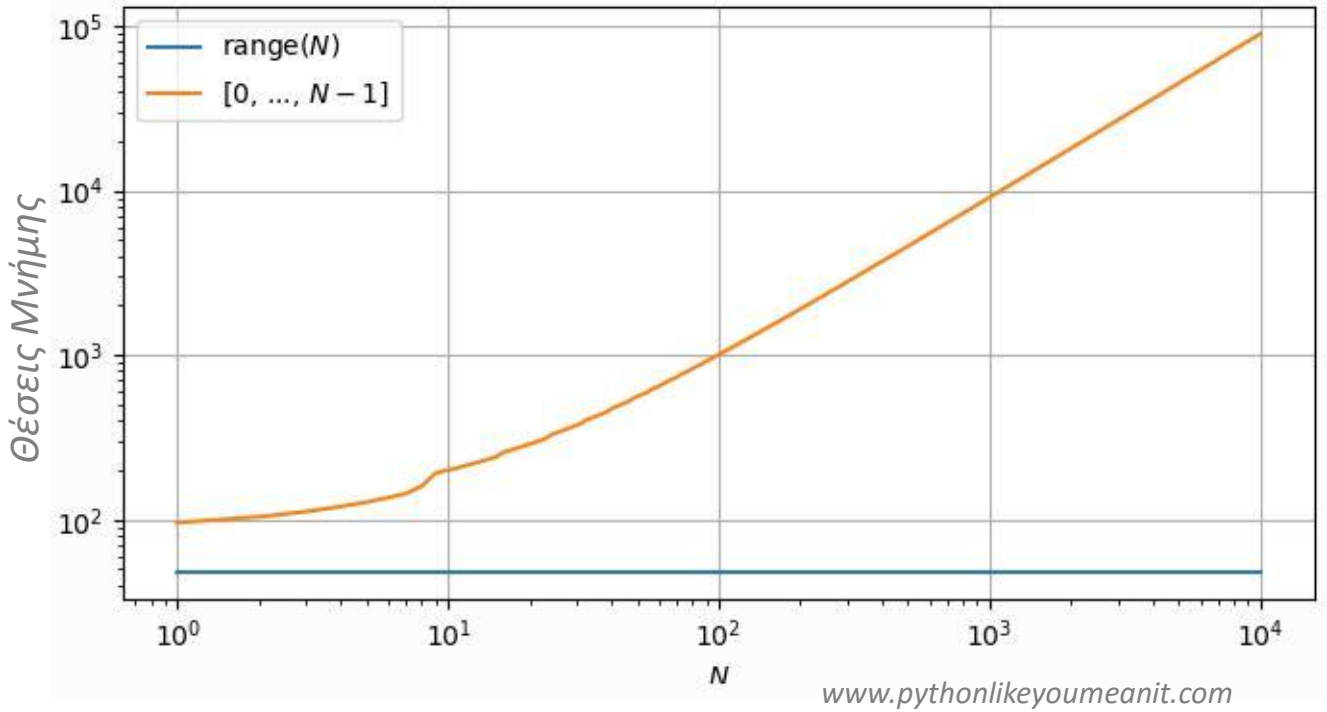
- Ενας επαναλήπτης `j` που δημιουργείται από *γεννήτορα ενσωματώνει την πληροφορία (αλγόριθμο) παραγωγής των στοιχείων του*, και κάθε φορά που καλείται με το `next(j)` δημιουργεί ***επιτόπου*** το επόμενο στοιχείο της επανάληψης

Εκτυπώνει ένα χαρτάκι κάθε φορά που χρειάζεται

Η προσέγγιση αυτή ονομάζεται *οκνηρή αποτίμηση (lazy evaluation)* και είναι σαφώς πιο αποδοτική επειδή κάθε καινούρια τιμή δημιουργείται *μόλις -και εφόσον- χρειαστεί*.

Συγκριτική χρήση μνήμης για γεννήτορες και επαναλήψιμους τύπους

Πχ ο επαναλήπτης `j = range(1000)` δεν δεσμεύει χίλιες θέσεις στη μνήμη, αλλά *επιστρέφει μια μοναδική τιμή* κάθε φορά που καλείται με το `next(j)` σύμφωνα με τις ανάγκες του προγράμματος.



Η μνήμη που απαιτείται για τον καθορισμό μιας επανάληψης από `0` έως `N-1` με το `range(N)` είναι *σταθερή (ανεξάρτητη από το N)* ενώ με τη χρήση λίστας *αυξάνεται γραμμικά* με το N

(Επιπλέον, μια επανάληψη ενδέχεται να διακοπεί πολύ νωρίτερα, από την αρχική πρόβλεψη)

Κατασκευάζοντας γεννητορικές συναρτήσεις

Εκτός από τις γεννητορικές συναρτήσεις της Python όπως την `range()` που ήδη ξέρουμε *(και άλλες που θα δούμε αργότερα)* μπορούμε *να κατασκευάσουμε κι εμείς τις δικές μας*

Συντακτικά, διαφέρουν από μια απλή συνάρτηση της Python στο ότι επιστρέφουν με την εντολή `yield` αντί για `return`. (δεν απαγορεύεται όμως να περιέχουν και `return` σε άλλα σημεία του κώδικά τους)

Βασικές διαφορές ανάμεσα σε `return` και `yield`

`return x`	`yield x`
Επιστρέφεται η τιμή του x, και η συνάρτηση ολοκληρώνει *(ends)* την εκτέλεσή της	Επιστρέφεται η τιμή του x, και η συνάρτηση αναστέλλει *(suspends)* την εκτέλεσή της
Όλοι οι *πόροι* που χρησιμοποιούσε η συνάρτηση *(τοπικές μεταβλητές κλπ.) απελευθερώνονται* και *επιστρέφουν* στο σύστημα	Οι *πόροι* της συνάρτησης *παραμένουν ενεργοί*, οι *μεταβλητές διατηρούν τις τιμές τους* μέχρι τη επόμενη κλήση
Σε επόμενη κλήση, η συνάρτηση δεσμεύει εκ νέου τους απαιτούμενους πόρους και *ξεκινάει να εκτελείται από την αρχή*	Σε επόμενη κλήση, η συνάρτηση *συνεχίζει να εκτελείται* από το σημείο που διέκοψε

Τρόπος λειτουργίας γεννητορικών συναρτήσεων

- Όταν καλείται μια γεννητορική συνάρτηση *υλοποιεί και επιστρέφει έναν επαναλήπτη*

  πχ. αν η `f()` *είναι γεννητορική συνάρτηση, τότε με την εντολή* `g = f()` *υλοποιείται ενας επαναλήπτης* `g`
- Την *πρώτη φορά* που ο επαναλήπτης αυτός καλείται μέσα από ένα `next()`, η γεννητορική συνάρτηση *ξεκινάει να εκτελείται* μέχρι να συναντήσει ένα `yield` οπότε και *σταματά προσωρινά* η εκτέλεσή της
- Η τιμή που επιστρέφει το `yield` περνάει στο πρόγραμμα ως αποτέλεσμα της `next()`

  πχ. με `a = next(g)` *το* `a` *θα πάρει την τιμή που θα επιστρέψει η* `f()` *μέσω του* `yield`
- Κάθε *επόμενη φορά* που ο επαναλήπτης καλείται μέσα από ένα `next()` η γεννητορική συνάρτηση *συνεχίζει να εκτελείται από το σημείο που είχε σταματήσει* μέχρι να συναντήσει εκ νέου ένα `yield` οπότε *σταματά και πάλι προσωρινά* η εκτέλεσή της
- Αν κατά την κλήση μιας γεννητορικής συνάρτησης αυτή τερματίσει χωρίς να εκτελεστεί η εντολή `yield`, *(είτε με* `return` *είτε φτάνοντας στο τέλος του μπλοκ του κώδικά της)* τότε εγείρεται η εξαίρεση `StopIteration`

Τρόπος λειτουργίας γεννητορικών συναρτήσεων

Παράδειγμα:

```
>>> def factors_gen(n): # Γεννητορική συνάρτηση που υπολογίζει τους διαιρέτες του n
        for j in range(1,n+1):      # το j διατρέχει την περιοχή από 1 έως και n
            if n%j == 0:            # αν το j διαιρείται ακριβώς με το n....
                yield j             # τότε σταμάτα προσωρινά και επέστρεψε το j
    # τέλος του μπλοκ της γεννητορικής συνάρτησης
```

```
>>> g = factors_gen(15)
>>> g
<generator object factors_gen at 0x012402F0>
>>> next(g)
1
>>> next(g)
3
>>> next(g)
5
>>> next(g)
15
>>> next(g)
Traceback (most recent call last): next(g) StopIteration
```

Παράδειγμα: Διαφορά ανάμεσα σε απλή και γεννητορική συνάρτηση

Απλή συνάρτηση: Υπολογίζει και επιστρέφει μια λίστα με *όλους* τους διαιρέτες του n

```
def factors_list(n):
    L = [] #αρχικοποίηση κενής λίστας
    for i in range(1,n+1):
        if n%i == 0:
            L.append(i)
    return L #λίστα με όλους τους διαιρέτες
```

λίστα (αποτέλεσμα κλήσης συνάρτησης)

```
for k in factors_list(15):
    print(k)
```

```
1
3
5
15
```

το k διατρέχει μια λίστα που βρίσκονται αποθηκευμένα όλα τα αποτελέσματα και τα εκτυπώνει

Γεννητορική συνάρτηση: Υπολογίζει και επιστρέφει *έναν-έναν* τους διαιρέτες του n

```
def factors_gen(n):
    for i in range(1,n+1):
        if n%i == 0:
            yield i #επόμενος παράγοντας
```

γεννήτρια (γεννητορική συνάρτηση)

```
for k in factors_gen(15):
    print(k)
```

```
1
3
5
15
```

Τα αποτελέσματα παράγονται ένα-ένα καθώς εκτυπώνονται. Αν η εκτύπωση διακοπεί νωρίτερα, τότε σταματά και η παραγωγή των αποτελεσμάτων

Μπορούμε πάντα να χρησιμοποιούμε γεννητορικές συναρτήσεις στη θέση επαναλήψιμων τύπων;

***Εν γένει* όχι**

Θα πρέπει να υπάρχει κάποια *μαθηματική (ή άλλου είδους) έκφραση* η οποία να μπορεί να περιγράψει όλα τα στοιχεία του επαναλήψιμου τύπου

πχ. η λίστα `[3, 5, 7, 11, 13, 17, 19]` μπορεί να περιγραφεί σαν *λίστα όλων των πρώτων ακεραίων από το 3 μέχρι και το 19* και επομένως να παραχθεί από μια γεννητορική συνάρτηση που υλοποιεί τη σχετική μαθηματική έκφραση

ενώ πχ. για τη λίστα `[12, 45, -6, 172, 99]` είναι δύσκολο να βρούμε κάποια μαθηματική έκφραση που να περιγράφει το πώς παράγονται στοιχεία της *(πιθανότατα να μην υπάρχει καν)*, και επομένως δεν είναι δυνατόν να χρησιμοποιήσουμε μια γεννητορική συνάρτηση για την παραγωγή της

Κατασκευάζοντας γεννητορικές εκφράσεις

Η ακόλουθη σύνταξη είναι εξαιρετικά χρήσιμη και εμφανίζεται πολύ συχνά στην Python:

```
g = ( <έκφραση> for <μεταβλητή> in <επαναλήψιμος τύπος> [if <συνθήκη>] )
```

Οι εκφράσεις αυτές ονομάζονται *γεννητορικές εκφράσεις* ή *συμπεριλήψεις* (generator comprehensions) και είναι ένας κομψός τρόπος της Python για τον ορισμό *απλών επαναληπτών σε μία μόνο γραμμή κώδικα*, αντί να γράφουμε γεννητορικές συναρτήσεις

```
>>> g_iter = (i**2 for i in range(1,4))
>>> next(g_iter)
1
>>> next(g_iter)
4
>>> next(g_iter)
9
>>> next(g_iter)
Traceback (most recent call last)
StopIteration
```

Ο επαναλήπτης `g_iter` εδώ ορίζεται μέσω *γεννητορικής έκφρασης*. Αν επιλέγαμε εναλλακτικά να τον ορίσουμε μέσω *γεννητορικής συνάρτησης* τότε θα γράφαμε πχ.

```
def squares(n):
    for i in range(1,n+1):
        yield i**2
g_iter = squares(3)
```

Ο πρώτος τρόπος είναι πιο απλός και συνοπτικός

Δημιουργία επαναλήψιμων τύπων από γεννητορικές εκφράσεις

Μια *γεννητορική έκφραση* μπορεί επίσης να χρησιμοποιηθεί για *δημιουργία επαναλήψιμων τύπων* με τη βοήθεια της αντίστοιχης συνάρτησης μετατροπής τύπου (`tuple()`,`list()`,`set()` κ.λπ.)

Πχ έστω ότι θέλουμε να δημιουργήσουμε μια λίστα με τα τετράγωνα των αριθμών από 1-10

```
>>> g_iter = (i*i for i in range(1,11)) #δημιουργία επαναλήπτη από γεννητορική έκφραση
>>> g_list = list(g_iter) #δημιουργία λίστας από τον παραπάνω επαναλήπτη
>>> g_list
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

>>> g_iter = (i*i for i in range(1,5) if i%2==0 ) απλό if
>>> g_list = list(g_iter)
>>> g_list
[4,16]

>>> g_iter = (i for i in range(20) if i%2==0 if i%3==0 ) διπλό if
>>> g_list = list(g_iter)
>>> g_list
[0,6,12,18]
```

Δημιουργεί μια λίστα *διατρέχοντας* και *εξαντλώντας* τον επαναλήπτη `g_iter` που δημιουργήθηκε από την γεννητορική έκφραση

Εξάντληση επαναλήπτη

Κατά την δημιουργία ενός *επαναλήψιμου τύπου* από έναν *επαναλήπτη ο οποίος ορίζεται από γεννητορική έκφραση*, ο επαναλήπτης αυτός *εξαντλείται* καθώς παράγει ένα-ένα τα στοιχεία του επαναλήψιμου τύπου μέχρι το `StopIteration`. Αυτό έχει ως αποτέλεσμα *να μπορεί να χρησιμοποιηθεί μόνο μία φορά*

```
>>> g_iter = (i**3 for i in range(1,5))
>>> g_tuple = tuple(g_iter)
>>> g_tuple
(1, 8, 27, 64)
```

Αν αμέσως μετά προσπαθήσουμε να δημιουργήσουμε πχ ***και μια λίστα*** από *τον ίδιο επαναλήπτη*, θα διαπιστώσουμε ότι *η λίστα που παίρνουμε είναι κενή*

```
>>> g_list = list(g_iter)
>>> g_list
[]   <- κενή λίστα αντί για την [1, 8, 27, 64] που ενδεχομένως περιμέναμε
```

Αυτό συμβαίνει επειδή ο επαναλήπτης μας *εξαντλήθηκε* δημιουργώντας την `g_tuple`. Άν όμως τον *ορίσουμε εκ νέου* με την ίδια γεννητορική έκφραση, τότε η λίστα μας θα δημιουργηθεί κανονικά

```
>>> g_iter = (i**3 for i in range(1,4)) # ορίζουμε εκ νέου  τον επαναλήπτη μας
>>> g_list = list(g_iter)
>>> g_list
[1, 8, 27, 64]
```

Συμπεριλήψεις επαναλήψιμων τύπων

Η Python μας παρέχει ένα *ακόμα πιο άμεσο τρόπο* δημιουργίας *επαναλήψιμων τύπων από γεννητορικές εκφράσεις* αντικαθιστώντας τις παρενθέσεις `()` στον ορισμό της γεννητορικής έκφρασης με αγκύλες `[]` για τις λίστες και άγκιστρα `{}` για τα σύνολα / λεξικά

Οι δομές αυτές ονομάζονται αντίστοιχα ***συμπεριλήψεις (comprehensions) λιστών, συνόλων ή λεξικών***

Αντί για: `my_list = list(3*i for i in range(4))`

Γράφουμε : `my_list = [3*i for i in range(4)]` ← *list comprehension*

*δημιουργεί την **λίστα** `[0,3,6,9]`*

Όμοια: `my_set = {x*x for x in range(1,11)}` ← *set comprehension*

*δημιουργεί το **σύνολο** `{64, 1, 4, 36, 100, 9, 16, 49, 81, 25}`*

`my_dict = {i : i**2 + 1 for i in range(1,7)}` ← *dictionary comprehension*

*δημιουργεί το **λεξικό** `{1:2, 2:5, 3:10, 4:17, 5:26, 6:37}`*

Προσοχή όμως: `my_tuple = (i+3 for i in range(2))`

ορίζει μια γεννητορική έκφραση, δεν δημιουργεί μια πλειάδα

Ενώ: `my_tuple = tuple(i+3 for i in range(2))`

*δημιουργεί την **πλειάδα** `(3,4)`*

Δεν μπορούμε να χρησιμοποιήσουμε σκέτες τις παρενθέσεις επειδή *αυτός ο συμβολισμός ήδη χρησιμοποιείται για ορισμό των γεννητορικών εκφράσεων*. Επομένως για δημιουργία πλειάδων χρησιμοποιούμε αναγκαστικά το `tuple()`

Παραδείγματα συμπεριλήψεων λιστών (1/2)

```
>>> odds = [x for x in range(1,11) if x%2 != 0]
>>> print(odds)
[1, 3, 5, 7, 9]

>>> [c for c in 'υπερτυχερος' if c in ('α', 'ε', 'η', 'ι', 'ο', 'υ', 'ω')]
['υ', 'ε', 'υ', 'ε', 'ο']

>>> [c*3 for c in "Test"]
['TTT', 'eee', 'sss', 'ttt']

>>> [i for i in range(8) if i%2==0 if i%3==0]   nested if
[0,6]
```

Παραδείγματα συμπεριλήψεων λιστών (2/2)

```
>>> [(i,j) for i in [0,1] for j in ('a', 'b', 'c')]
[(0, 'a'), (0, 'b'), (0, 'c'), (1, 'a'), (1, 'b'), (1, 'c')]

>>> [(i,j) for i in [0,1] for j in ['a', 'b', 'c'] if j != 'b']
[(0, 'a'), (0, 'c'), (1, 'a'), (1, 'c')]

>>> [(i,j) for i in range(1,4) for j in range(1,4)]
[(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3), (3, 1), (3, 2), (3, 3)]

>>> [(i,j) for i in range(1,4) for j in range(1,4) if i==j]
[(1, 1), (2, 2), (3, 3)]

>>> [(i, i**2, i**3) for i in range(5)]
[(0, 0, 0), (1, 1, 1), (2, 4, 8), (3, 9, 27), (4, 16, 64)]
```

Παραδείγματα συμπεριλήψεων συνόλων

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

Αξίζει να παρατηρηθεί η ομοιότητα με την αντίστοιχη μαθηματική σημειολογία για τον ορισμό του συνόλου $x^2 : x \in S$

```
>>> {(m, n) for n in range(2) for m in range(3, 6)}
{(3, 0), (3, 1), (5, 0), (5, 1), (4, 1), (4, 0)}
```

προσέξτε την σειρά ορισμού των m και n

```
>>> {(m, n) for n in range(2) for m in range(3, 6) if m+n==4}
{(3, 1), (4, 0)}
```

Παραδείγματα συμπεριλήψεων λεξικών

```
>>> {i : i**2 + 1 for i in range(1,7)}
{1: 2, 2: 5, 3: 10, 4: 17, 5: 26, 6: 37}

>>> keys = ["Μαρία", "Κώστας", "Στέλιος"]
>>> values = [23, 45, 18]
>>> { keys[i] : values[i] for i in range(len(keys))}
{'Μαρία': 23, 'Κώστας': 45, 'Στέλιος': 18}
```

Το δομοστοιχείο *itertools* της Python

- Περιλαμβάνει μια σειρά *γρήγορων και αποδοτικών εργαλείων* για την *δημιουργία επαναληπτών*
- Τρεις από τις συναρτήσεις που περιλαμβάνονται σε αυτό, οι `range()`, `enumerate()` και `zip()`, είναι τόσο χρήσιμες που υπάρχουν και ως *ενσωματωμένες συναρτήσεις* στον πυρήνα της Python

Αντιπροσωπευτικά στοιχεία *από το δομοστοιχείο itertools*

`count(start, [step])`	Δημιουργεί έναν επαναλήπτη ο οποίος επιστρέφει ***συνεχώς*** ακεραίους ξεκινώντας από το start και με βήμα step. Η διαδικασία επαναλαμβάνεται επ' άπειρον (ουδέποτε εγείρεται `StopIteration`) `count(5, 2)` → `5 7 9 11 13 ...`
`cycle(p)`	Δημιουργεί έναν επαναλήπτη ο οποίος επιστρέφει ***συνεχώς*** τα στοιχεία του επαναληπτικού τύπου p, κάνοντας συνεχώς κύκλους Η διαδικασία επαναλαμβάνεται επ' άπειρον (ουδέποτε εγείρεται `StopIteration`) `cycle('ABCD')` → `A B C D A B C D ...`
`repeat(e, n)`	Δημιουργεί έναν επαναλήπτη ο οποίος επαναλαμβάνει **`n`** φορές το στοιχείο **`e`** και αμέσως μετά εγείρει `StopIteration` `repeat(10, 3)` → `10 10 10`

Το *δομοστοιχείο itertools*

Αντιπροσωπευτικά στοιχεία *από το δομοστοιχείο itertools (συνέχεια)*

`islice(seq, [start,] end, [,step])`	Δημιουργεί έναν επαναλήπτη με στοιχεία ένα τμήμα (slice) του επαναληπτικού τύπου `seq` ξεκινώντας από το στοιχείο `start` και μέχρι το στοιχείο `end-1` ανά βήμα `step`, και αμέσως μετά εγείρει `StopIteration` . Αν το `end` είναι `None` τότε λαμβάνονται τα στοιχεία μέχρι το τέλος του `seq` `islice('ABCDEFG', 2, None)` → `C D E F G`
`permutations(p,k)`	Δημιουργεί έναν επαναλήπτη ο οποίος επιστρέφει όλους τους δυνατές μεταθέσεις των στοιχείων του επαναληπτικού τύπου p ανά k, και αμέσως μετά εγείρει `StopIteration` `permutations('ABCD', 2)` → `AB AC AD BA BC BD CA CB CD DA DB DC`
`combinations(p,k)`	Δημιουργεί έναν επαναλήπτη ο οποίος επιστρέφει όλους τους δυνατές συνδυασμούς των στοιχείων του επαναληπτικού τύπου p ανά k, και αμέσως μετά εγείρει `StopIteration` `combinations('ABCD', 2)` → `AB AC AD BC BD CD`

Το *δομοστοιχείο itertools – Παραδείγματα χρήσης 1/2*

```
>>> L = [100,200,300]
>>> q = enumerate(L)
>>> q
<enumerate object at 0x00213760>
>>> r = list(q)
>>> r
[(0, 100), (1, 200), (2, 300)]
```

Θυμόμαστε: Με το enumerate απαριθμούμε τα στοιχεία ενός επαναλήψιμου τύπου, ζευγαρώνοντάς τα σε πλειάδες με τους ακέραιους 0, 1, 2, …

```
>>> L1 = [ "Alice", "Bob", "Chris" ]
>>> L2 = [32, 18, 27]
>>> z = zip(L1, L2)
>>> print(z)
<zip object at 0x00210EE0>
>>> print(list(z))
[('Alice', 32), ('Bob', 18), ('Chris', 27)]
```

Θυμόμαστε: Με το zip ταιριάζουμε σε πλειάδες τα στοιχεία δυο ή περισσότερων επαναλήψιμων τύπων

Θυμόμαστε: οι 3 συναρτήσεις `range()`, `enumerate()` και `zip()` του `itertools` είναι τόσο χρήσιμες ώστε υπάρχουν και ως *ενσωματωμένες συναρτήσεις* στον πυρήνα της Python.

Έτσι, *δεν χρειάζεται* να κάνουμε `import itertools` πριν αρχίσουμε να τις χρησιμοποιούμε

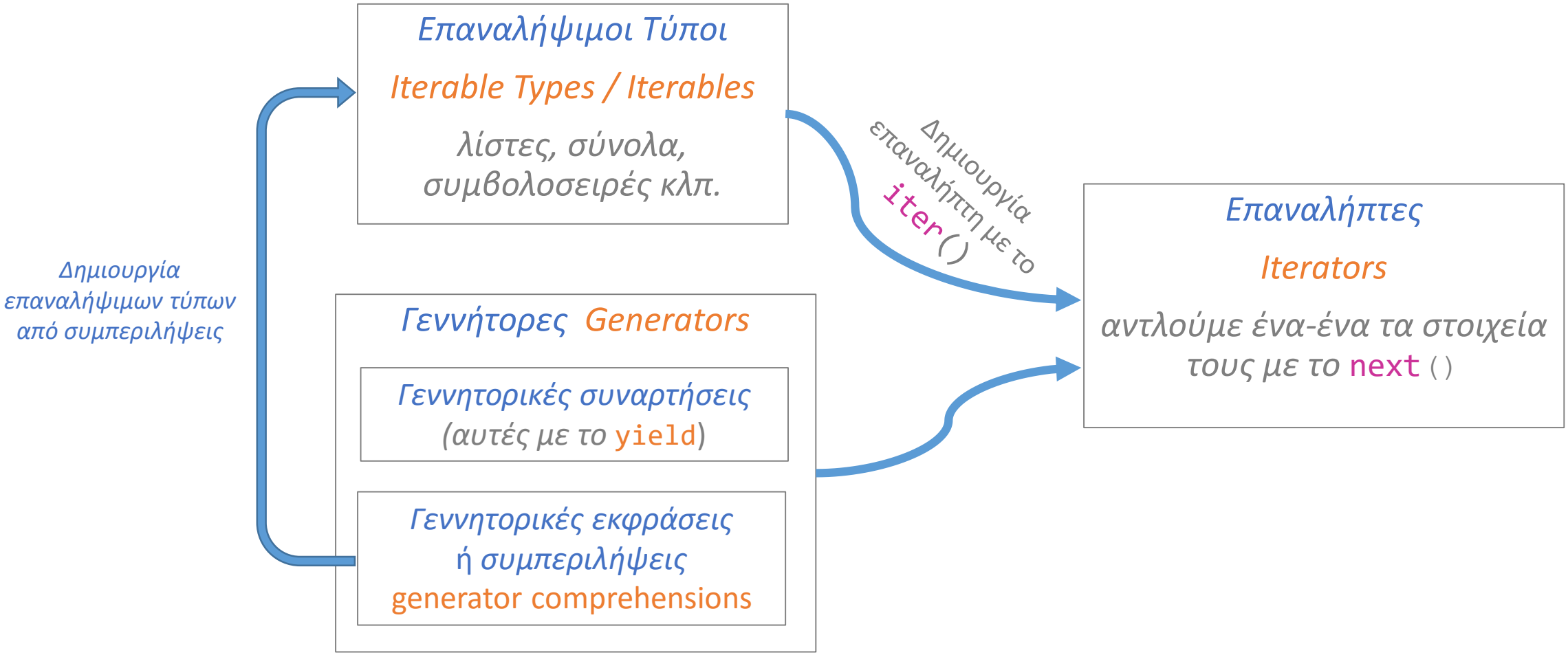
Το *δομοστοιχείο itertools – Παραδείγματα χρήσης 2/2*

```
>>> import itertools
>>> q = itertools.count(5,2)
>>> q
count(5, 2)
>>> next(q)
5
>>> next(q)
7
```

```
>>> for x in itertools.repeat('Hi',3):
        print(x, end = " ")
Hi Hi Hi
```

```
>>> L = [z for z in itertools.islice('ABCDEGH', 0, None, 2)]
>>> L
['A', 'C', 'E', 'H']
```

Συνοψίζοντας το σημερινό μάθημα…



Τέλος Διάλεξης

Ερωτήσεις;

Τμήματα αυτής της διάλεξης περιέχουν πληροφορίες από πηγές που είναι ελεύθερες στο διαδίκτυο όπως η Βικιπαίδεια και ανοιχτές σημειώσεις παρεμφερών διαλέξεων