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

HMY01K06
Επιστημονικός Προγραμματισμός με Python

6+1 ερωτήσεις επανάληψης για τη Διάλεξη 09

Επαναλήψιμοι τύποι / Επαναλήπτες

Γεννήτορες / Συμπεριλήψεις

Φθινόπωρο 2025

Ερώτηση 1/7

Τι ακριβώς κάνουν οι συναρτήσεις `any()` και `all()` ;

Είναι συναρτήσεις που δρούν πάνω σε επαναλήψιμους τύπους

- Η `any()` απαντά ουσιαστικά στην ερώτηση:
  "Περιέχει ο επαναλήψιμος τύπος *τουλάχιστον ένα μη τετριμμένο στοιχείο*;"
  πχ. `any([1,2,0])` επιστρέφει `True` ενώ `any([0,0,0])` επιστρέφει `False`

- Η `all()` απαντά ουσιαστικά στην ερώτηση:
  "Είναι όλα τα στοιχεία του επαναλήψιμου τύπου *μη τετριμμένα*;"
  πχ. `all([1,2,0])` επιστρέφει `False` ενώ `all({})` επιστρέφει `True` *(γιατί;)*

Ερώτηση 2/7

Τι ακριβώς κάνει η συνάρτηση `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

```
names =
["Alice", "Bob", "Carol", "David"]

     exam_scores =[90, 82, 79, 87]
```

```
[('Alice', 90), ('Bob', 82),
('Carol', 79), ('David', 87)]
```

```
zip(names, exam_scores)
```

Ερώτηση 3/7

Τι είναι οι επαναλήπτες (iterators) στην Python;

Αντικείμενα της Python τα οποία προκύπτουν ως αποτέλεσμα της εφαρμογής της συνάρτησης `iter()` πάνω σε ένα *επαναλήψιμο τύπο*

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

`i`: Επαναλήπτης

`iter('Hello')`: Επαναλήψιμος τύπος

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 του τη συνάρτηση `next()`

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

```
>>> i = iter([1,2,3])
>>> next(i)
1
>>> next(i)
2
>>> next(i)
3
```

```
>>> next(i)
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    next(i)
StopIteration
```

Ερώτηση 4/7

Τι είναι οι γεννήτορες επαναλήψιμων τύπων (generators) στην Python;

Αντικείμενα της Python τα οποία παράγουν επαναλήπτες *με έμμεσο τρόπο*, δηλαδή αντί να παράγουν *όλα τα στοιχεία τους από την αρχή* περιέχουν *οδηγίες για το πώς να τα παράγουν ένα-ενα* ***όποτε*** *χρειαστούν*

Διακρίνονται σε

- Γεννητορικές συναρτήσεις (generator function)
  - Κάποιες βρίσκονται *ενσωματωμένες* στην Python `range()`, `enumerate()`, `zip()`
  - Οι υπόλοιπες στο *δομοστοιχείο* `itertools` `count()`, `cycle()`, `permutations()`…
  - Μπορούμε να *κατασκευάσουμε δικές μας* αντικαθιστώντας το `return` με `yield`

- Γεννητορικές εκφράσεις (generator comprehensions)

  Απλές γεννητορικές συναρτήσεις που γράφονται *σε μια μόνο γραμμή* ακολουθώντας ένα *ειδικό τρόπο* γραφής πχ `g = (i*i for i in range(1,5) if i%2==0)`

Ερώτηση 5/7

Σε τι διαφέρουν οι γεννητορικές συναρτήσεις από τις απλές;

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

Ερώτηση 6/7

Τι ακριβώς είναι μια *συμπερίληψη λίστας* (list comprehension);

Ενας *άμεσος τρόπος* δημιουργίας μιας λίστας από μια *από γεννητορική έκφραση* αντικαθιστώντας τις παρενθέσεις `()` στον ορισμό της γεννητορικής έκφρασης με αγκύλες `[]`

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

Γράφουμε: `[3*i for i in range(4)]`

Παράγει την λίστα: [0, 3, 6, 9]

Αντίστοιχα μπορούμε να δημιουργούμε συμπεριλήψεις

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

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

αλλά *όχι* πλειάδων *(γιατί;)*

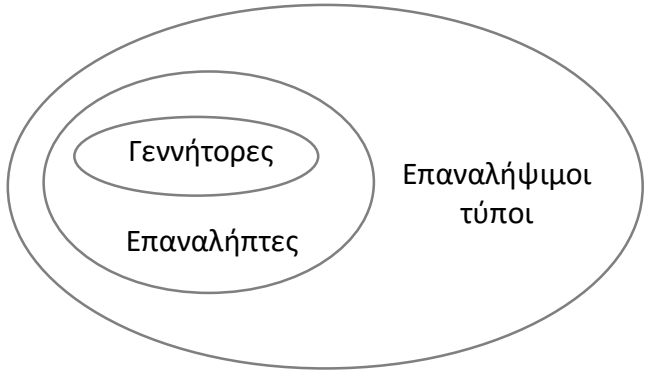
Bonus Ερώτηση 7/7

Πως συσχετίζονται οι επαναλήψιμοι τύποι (iterables) οι επαναλήπτες (iterators), οι γεννήτορες (generators) και της Python;

- *Επαναλήψιμοι τύποι* (iterables) : αντικείμενα που μπορούν να μας επιστρέφουν τα μέλη τους *ένα κάθε φορά*, προσφέροντας έτσι -μεταξύ άλλων- την δυνατότητα να δημιουργούμε επαναλήψεις (πχ for loops)

- *Επαναλήπτες* (iterators) : αντικείμενα τα οποία δρουν πάνω σε επαναλήψιμους τύπους και προσδιορίζουν *το πώς ακριβώς υλοποιείται μια επανάληψη*, δηλαδή πιο συγκεκριμένα *ποιο είναι το επόμενο στοιχείο* κάθε φορά. Υλοποιούνται με τον μηχανισμό `iter()` / `next()`

- *Γεννήτορες* (generators) *επαναλήψιμων τύπων* : αντικείμενα (συναρτήσεις) τα οποία παράγουν επαναλήπτες *με έμμεσο τρόπο*, δηλαδή αντί να παράγουν *όλα τα στοιχεία τους από την αρχή* περιέχουν *οδηγίες για το πώς να τα παράγουν ένα-ενα όποτε χρειαστούν*



Οι *Γεννήτορες* παράγουν
- *Επαναλήπτες*
- *Επαναλήψιμους τύπους*

Οι *Επαναλήπτες* παράγουν
- *Επαναλήψιμους τύπους*

Προσοχή! δεν παράγονται όλοι οι επαναλήψιμοι τύποι με αυτό τον μηχανισμό. (πχ. `[1, 45.2, 'Hello', False]`)