Προγραμματιστικά Εργαλεία και Τεχνολογίες για Επιστήμη Δεδομένων

Παράδοση 10/1/2019, Νίκος Παπασπύρου.

Διάβασμα από το standard input

1. Συμβολοσειρά από μία γραμμή

   ```
   name = input()
   print("Your name is:", name)
   ```

2. Λίγο πιο διαδραστικά

   ```
   print("What's your name?", end=" ")
   name = input()
   print("Hello", name)
   print("What's up?")
   ```

3. Ακέραιο από μία γραμμή

   ```
   n = int(input())
   print("Your number was:", n)
   ```

4. Και πάλι, πιο διαδραστικά

   ```
   print("What's your name?", end=" ")
   name = input()
   print("Hello", name)
   print("What's your age?", end=" ")
   age = int(input())
   print("Next year you'll be", age+1, "years old")
   ```

5. Δύο λέξεις από μία γραμμή

   ```
   first, last = input().split()
   print("Your first name is", first, "and your last name is", last)
   ```

6. Δύο αριθμοί από μία γραμμή

   ```
   first, last = input().split()
   n = int(first)
   m = int(last)
   ```

   Με χρήση list comprehension (βλ. και παρακάτω)

   ```
   [n, m] = [int(word) for word in input().split()]
   ```

   ισοδύναμα

   ```
   n, m = [int(word) for word in input().split()]
   ```

   ή καλύτερα — η `map` εφαρμόζει τη συνάρτηση `int` πάνω σε όλα τα στοιχεία της λίστας που δίνεται ως δεύτερη παράμετρος και επιστρέφει τη λίστα (ακριβέστερα, έναν iterator) που περιέχει τα αποτελέσματα των επιμέρους εφαρμογών.

   ```
   n, m = map(int, input().split())
   ```

Η πρώτη μας προγραμματιστική άσκηση

Γράψτε ένα πρόγραμμα που να διαβάζει από την πρώτη γραμμή της εισόδου δύο αριθμούς, χωρισμένους μεταξύ τους με ένα κενό διάστημα, και να εκτυπώνει το άθροισμά τους.

1. Πρώτη λύση

```
line = input()
first, second = line.split()
print(int(first) + int(second))
```

2. Με χρήση της `map`

```
first, second = map(int, input().split())
print(first + second)
```

3. Γενίκευση σε one-liner

   Η παρακάτω λύση δουλεύει για οσοδήποτε πολλούς αριθμούς.

```
print(sum(map(int, input().split())))
```

Πρόβλημα "exclude"

Δίνονται δύο ακολουθίες a$(1), ..., a(N)$ και $b(1), ..., b(M)$, αποτελούμενες από φυσικούς αριθμούς. Ζητείται να βρεθούν οι αριθμοί της πρώτης ακολουθίας που δεν ανήκουν στη δεύτερη.

Δεδομένα εισόδου

Η πρώτη γραμμή της εισόδου θα περιέχει δύο αριθμούς χωρισμένους μεταξύ τους με ένα κενό διάστημα: τις τιμές των και M. Η δεύτερη γραμμή της εισόδου θα περιέχει φυσικούς αριθμούς, που αντιστοιχούν στους όρους της πρώτης ακολουθίας, χωρισμένους ανά δύο με ένα κενό διάστημα. Ομοίως, η τρίτη γραμμή της εισόδου θα περιέχει τους M φυσικούς αριθμούς της δεύτερης ακολουθίας. Να θεωρήσετε ως δεδομένο ότι η είσοδος θα είναι έγκυρη και ότι οι αριθμοί δε θα υπερβαίνουν τα όρια που αναγράφονται παρακάτω.

Δεδομένα εξόδου

Η έξοδος πρέπει να αποτελείται από τόσες γραμμές όσοι όροι της πρώτης ακολουθίας δεν εμφανίζονται στη δεύτερη. Κάθε γραμμή θα περιέχει ακριβώς έναν όρο της πρώτης ακολουθίας. Η σειρά εμφάνισης των όρων θα είναι η ίδια με τη σειρά που αυτοί εμφανίζονται στην είσοδο.

Περιορισμοί

- $1 \leq N, M \leq 1.000.000$
- $0 \leq a(i), b(j) \leq 1.000.000$

Παράδειγμα εισόδου 1

```
5 5
4 9 5 1 10
5 7 2 4 1
```

Παράδειγμα εξόδου 1

```
9
10
```

Παράδειγμα εισόδου 2

```
10 7
5 17 15 11 13 10 5 1 4 9
14 1 8 11 19 13 9
```

Παράδειγμα εξόδου 2

```
5
17
15
10
5
4
```

1. Πρώτη λύση, μη αποδοτική

```
N, M = map(int, input().split())
A = list(map(int, input().split()))
B = list(map(int, input().split()))
for a in A:
    if a not in B:
        print(a)
```

Η λύση αυτή είναι σωστή αλλά δεν είναι αποδοτική. Η χρήση του τελεστή `in` στην έκφραση `a not in B` οδηγεί στη διάσχιση της λίστας `B` μέσα στην οποία αναζητάται το στοιχείο `a`. Η διάσχιση της λίστας έχει κόστος $O(M)$ (μία σύγκριση για κάθε στοιχείο της λίστας B, στη χειρότερη περίπτωση) και γίνεται N φορές, άρα το συνολικό κόστος είναι στη χειρότερη περίπτωση $O(NM)$.

2. Καλύτερη λύση

```
N, M = map(int, input().split())
A = map(int, input().split())
B = set(map(int, input().split()))
for a in A:
    if a not in B:
        print(a)
```

Η μοναδική διαφορά είναι ότι το `B` αντί για λίστα είναι τώρα σύνολο. (Για την ακρίβεια, έχει αφαιρεθεί και το `list` από το `A` που τώρα είναι ένας iterator, αλλά αυτό δεν έχει ουσιαστικό αντίκτυπο στο κόστος.) Τώρα κάθε χρήση του τελεστή `in` στην έκφραση `a not in B` κοστίζει πολύ λιγότερο γιατί ο έλεγχος αν ένα στοιχείο ανήκει σε ένα σύνολο υλοποιείται πολύ αποδοτικά — τα σύνολα υλοποιούνται με hash tables και το κόστος του ελέγχου είναι πρακτικά $O(1)$. Επομένως, το συνολικό κόστος αυτής της λύσης είναι πρακτικά $O(N + M)$.

Περισσότερα για τα σύνολα της Python και τις πράξεις που ορίζονται για αυτά υπάρχουν παρακάτω, στις ίδιες σημειώσεις.

Πρόβλημα "κυλικείο"

Μια ομάδα μαθητών στη σχολική αυλή, στέκονται σε μια ευθεία γραμμή, το ένα πίσω από το άλλο, περιμένοντας τη σειρά τους στο κυλικείο του σχολείου. Το πρώτο παιδί προφανώς βλέπει την είσοδο του κυλικείου, όσα παιδιά όμως στέκονται πίσω του δεν είναι σίγουρο ότι και αυτά τη βλέπουν. Για να βλέπει ένα παιδί την είσοδο του κυλικείου πρέπει όλα τα παιδιά που στέκονται μπροστά του να είναι κοντύτερα από αυτό.

Να γράψετε ένα πρόγραμμα το οποίο, αφού διαβάσει ένα αρχείο με τη λίστα των υψών των παιδιών, θα εκτυπώνει πόσα παιδιά βλέπουν την είσοδο.

Δεδομένα εισόδου

Η είσοδος περιέχει μόνο δύο γραμμές. Στην πρώτη γραμμή υπάρχει ένας ακέραιος αριθμός N: το πλήθος των παιδιών που στέκονται στη γραμμή. Στη δεύτερη γραμμή υπάρχουν N ακέραιοι αριθμοί, χωρισμένοι ανά δύο με ένα κενό διάστημα. Οι αριθμοί αυτοί είναι τα ύψη των παιδιών, τα οποία δίνονται με τη σειρά που αυτά στέκονται στη γραμμή με κατεύθυνση από πίσω προς τα μπρος. Δηλαδή, ο πρώτος αριθμός της δεύτερης γραμμής είναι το ύψος του τελευταίου παιδιού (αυτού που βρίσκεται μακριά από την είσοδο του κυλικείου) ενώ ο τελευταίος αριθμός είναι το ύψος του πρώτου παιδιού (αυτού που βρίσκεται κοντά στην είσοδο).

Αρχεία Εξόδου

Η έξοδος πρέπει να περιέχει μόνο μία γραμμή που περιέχει μόνο έναν ακέραιο αριθμό K (όπου $1 \leq K \leq N$): το πλήθος των παιδιών που βλέπουν την είσοδο του κυλικείου.

Περιορισμοί

- $1 \leq N \leq 1.000.000$

Παράδειγμα εισόδου 1

```
7
5 6 4 6 3 4 1
```

Παράδειγμα εξόδου 1

```
3
```

Παράδειγμα εισόδου 2

```
4
23 17 7 42
```

Παράδειγμα εξόδου 2

```
1
```

1. Πρώτη λύση, μη αποδοτική

```
N = int(input())
A = list(map(int, input().split()))
count = 0
for i in range(N):
    good = True
    for j in range(i+1, N)
        if A[i] <= A[j]:
            good = False
            break
    if good:
        count += 1
print(count)
```

Η λύση αυτή είναι και πάλι σωστή αλλά όχι αποδοτική. Κάθε αριθμός συγκρίνεται με όλους τους επόμενούς του και, αν δε βρεθεί κανένας τουλάχιστον τόσο μεγάλος, προσμετράται στην απάντηση. Το συνολικό κόστος είναι (N^2) γιατί ο πρώτος αριθμός θα συγκριθεί με τους $N-1$ επόμενούς του, ο δεύτερος με τους $N-2$ επόμενούς του, κ.ο.κ. Επιπλέον, η λύση αυτή είναι γραμμένη σαν να την είχαμε γράψει σε C ή σε Java, όχι σε Python.

2. Δεύτερη λύση, μη αποδοτική αλλά "πιο Python"

```
N = int(input())
A = list(map(int, input().split()))
count = 0
for i in range(N):
    if all(A[i] > A[j] for j in range(i+1, N)):
        count += 1
print(count)
```

Η λύση αυτή κάνει το ίδιο με την προηγούμενη. Τώρα όμως, ο δεύτερος βρόχος είναι "κρυμμένος" μέσα στην ενσωματωμένη συνάρτηση `all`, η οποία ελέγχει αν ένας αριθμός είναι μεγαλύτερος από **όλους** τους επόμενούς του.

3. Τρίτη λύση, μη αποδοτική και ακόμα "πιο Python"

```
N = int(input())
A = list(map(int, input().split()))
count = sum(1 for i in range(N)
              if all(A[i] > A[j] for j in range(i+1, N)))
print(count)
```

Επίσης ισοδύναμη με την προηγούμενη αλλά υπολογίζει το συνολικό πλήθος όσων είναι μεγαλύτεροι από όλους τους επόμενούς τους με χρήση της ενσωματωμένης συνάρτησης `sum`.

4. Άσκηση για το σπίτι: Μπορείτε να βρείτε μία αποδοτικότερη λύση που να υπολογίζει το ζητούμενο με κόστος $O(N)$;

Σύνολα

1. Αναπαριστούν μαθηματικά σύνολα με στοιχεία (σχεδόν) οποιουδήποτε τύπου (πρέπει να πληρούν τις ίδιες προϋποθέσεις με τα κλειδιά ενός λεξικού).

```
s = set()
s.add(1)
s.add(2)
s.add(3)
print(2 in s)   # prints True
print(4 in s)   # prints False
s.remove(2)
print(2 in s)   # prints False now
```

2. Πράξεις συνόλων:

```
s1 = set([1, 2, 3])
s2 = {3, 4, 5}        # equivalent to set([3, 4, 5])

print(s1 | s2)        # union: {1, 2, 3, 4, 5}
print(s1 & s2)        # intersection: {3}
print(s1 - s2)        # difference: {1, 2}
```

3. Τα σύνολα (set) είναι mutable. Υπάρχει και η immutable εκδοχή τους (frozenset) που μπορούν να χρησιμοποιηθούν ως κλειδιά σε λεξικά ή ως στοιχεία σε άλλα σύνολα.

```
s1 = frozenset([1, 2, 3])   # two immutable sets
s2 = frozenset([4, 5, 6])
s = {s1, s2}                # and a (mutable) set containing them
```

Comprehensions

Χρησιμοποιούνται για την κατασκευή δομών δεδομένων (π.χ. λιστών, συνόλων, λεξικών, κ.λπ.) παραθέτοντας τον τρόπο απαρίθμησης των στοιχείων τους.

1. List comprehensions

```
b = [i*i for i in range(10)]
print(b)    # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

b = [i*i for i in range(10) if i*i%2==1]
print(b)    # [1, 9, 25, 49, 81]
```

2. Set comprehensions:

```
s = set(i*i for i in range(10))     # {0, 1, 64, 4, 36, 9, 16, 49, 81, 25}
```

ή ισοδύναμα:

```
s = {i*i for i in range(10)}
```

Προσέξτε ότι τα σύνολα είναι unordered (δεν υπάρχει διάταξη στοιχείων).

3. Dict comprehensions:

```
d = {i*i: i for i in range(10)}
```

print(d[49]) # 7 print(d[81]) # 9 print(d[42]) # exception, element not defined

Iterables και generators

1. Η έκφραση `range(10)` είναι ένα **iterable**. Μπορεί να χρησιμοποιηθεί για να απαριθμήσει μία ακολουθία στοιχείων (συγκεκριμένα, τους αριθμούς $0, 1, \ldots 9$).

   Ένα iterable μπορεί να χρησιμοποιηθεί π.χ. για την ανάθεση σε ίσου πλήθους μεταβλητές:

```
a, b, c = range(3)   # a = 0, b = 1, c = 2
```

   ή όπως έχουμε δει σε συνδυασμό με την εντολή `for`:

```
for i in range(3):
    print(i)
```

2. Η έκφραση `i*i for i in range(10)`, που χρησιμοποιήθηκε στα comprehensions παραπάνω, είναι μία γεννήτρια (**generator**). Γεννάει μία ακολουθία από τιμές, τις οποίες μπορούμε να διατρέξουμε — κάθε γεννήτρια είναι ένα iterable. Η γεννήτρια αυτή αντιστοιχεί στο μαθηματικό σύνολο $\left\{i^2 \mid i \in \{0..9\}\right\}$.

3. Μπορεί κανείς να φτιάξει μία δική του γεννήτρια, ορίζοντας μία συνάρτηση που αντί για `return` κάνει `yield` τις τιμές που γεννάει:

```
def fib(n):
    a, b = 0, 1
    yield a
    while b <= n:
        yield b
        c = a+b
        a, b = b, c

for i in fib(42):
    print(i)        # prints 0, 1, 1, 2, 3, 5, 8, 13, 21, 34
```

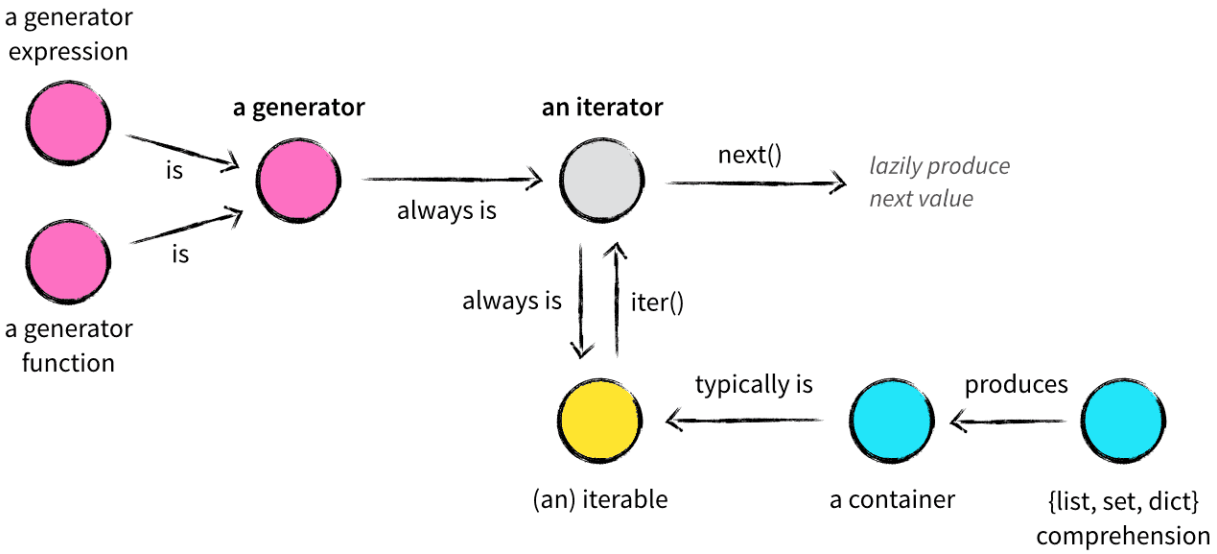
   Κάθε συνάρτηση που περιέχει την εντολή `yield` αντιστοιχεί σε ένα αντικείμενο generator. Οι τιμές που γεννάει είναι ακριβώς αυτές που θα εκτυπώνονταν, αν κάθε εντολή `yield` είχε αντικατασταθεί από μία εντολή `print`.

4. Οι τιμές μίας γεννήτριας δεν αποθηκεύονται κάπου. Καταναλώνονται όπως γεννιούνται. Μπορούμε επομένως να ορίσουμε άπειρες γεννήτριες:

```
def fib():
    a, b = 0, 1
    yield a
    while True:
        yield b
        c = a+b
        a, b = b, c

for i in fib():
    print(i)
    if i > 10**6: break   # prints 0, 1, 1, 2, ... 832040, 1346269
```

5. Για την ακριβή σχέση μεταξύ iterables, iterators, generators και containers στην Python, μπορείτε να διαβάσετε το σχετικό άρθρο, από το οποίο προέρχεται το παρακάτω σχήμα.



Η βιβλιοθήκη της Python

1. Είναι οργανωμένη σε modules. Τα χρησιμοποιούμε με εντολές `import`.
2. Documentation: https://docs.python.org/3/library/
3. Το module `random` έχει πολλές συναρτήσεις που μπορούν να χρησιμοποιηθούν για τη γέννηση (ψευδο)τυχαίων αριθμών. Το παρακάτω πρόγραμμα κατασκευάζει με τυχαίο τρόπο δεδομένα εισόδου για το πρόβλημα "κυλικείο" που περιγράφηκε παραπάνω.

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

import random
import sys

N = int(sys.argv[1])
A = [random.randrange(100, 180) for i in range(N)]

print(N)
for x in A:
    print(x, end=" ")
print()
```

Αν το πρόγραμμα αυτό αποθηκευθεί στο εκτελέσιμο αρχείο `gen.py`, τότε η εκτέλεσή του με την εντολή:

```
$ ./gen.py 1000 > input.txt
```

θα κατασκευάσει ένα αρχείο εισόδου με $N = 1000$, στο οποίο τα ύψη των παιδιών θα είναι φυσικοί αριθμοί στο διάστημα $[100, 180)$, και θα το αποθηκεύσει με όνομα `input.txt`.

4. Π.χ. το module `itertools` έχει πολλές χρήσιμες συναρτήσεις για να κατασκευάζουμε γεννήτριες:

```
import itertools

# all (6) permutations of elements 1, 2, 3
for l in itertools.permutations([1, 2, 3]):
    print(l)

# all (6) pairs of Daltons
for l in itertools.combinations(["joe", "jack", "william", "averel"], 2):
    print(l)
```

5. Μπορεί κανείς να ορίζει τα δικά του modules.