Προγραμματιστικά Εργαλεία και Τεχνολογίες για Επιστήμη Δεδομένων

Παράδοση 29/9/2020, 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 N και M. Η δεύτερη γραμμή της εισόδου θα περιέχει N 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 $O(N + M)$.

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 $\mathrm{O}(N^2)$ 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. Μπορείτε να βρείτε μία αποδοτικότερη λύση που να υπολογίζει το ζητούμενο με κόστος $O(N)$;

```
N = int(input())
a = list(map(int, input().split()))
count = 0
maxSoFar = -1
for x in reversed(a):
    if x > maxSoFar:
        maxSoFar = x
        count += 1
print(count)
```

Η βιβλιοθήκη της Python

1. Είναι οργανωμένη σε modules. Τα χρησιμοποιούμε με εντολές `import`.
2. Documentation: https://docs.python.org/3/library/
3. 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)
```

4. Μπορεί κανείς να ορίζει τα δικά του modules.

Γεννήτριες τυχαίων δεδομένων

1. Θα χρησιμοποιήσουμε το module `random` της βιβλιοθήκης της Python για να κατασκευάσουμε τυχαία δεδομένα εισόδου, προκειμένου να ελέγξουμε τη λύση μας για το πρόβλημα "Κυλικείο".

   ```
   import random
   ```

2. Η συνάρτηση `random.randrange` επιστρέφει έναν (ψευδο)τυχαίο αριθμό στο αντίστοιχο `range`.

   ```
   N = random.randrange(1, 100)
   a = [random.randrange(1, 100) for i in range(N)]
   print(N)
   print(*a)
   ```

   Η τελευταία γραμμή τυπώνει όλα τα στοιχεία της λίστας `a` χωρισμένα μεταξύ τους με ένα κενό διάστημα. Είναι το ίδιο σαν να είχε κανείς δώσει ως ξεχωριστές παραμέτρους στην `print` όλα τα στοιχεία της λίστας `a`, ένα-ένα.

3. Παρομοίως μπορεί κανείς να γράψει αυτά τα τυχαία δεδομένα σε ένα αρχείο `data.txt`:

   ```
   f = open("data.txt", "wt")
   N = random.randrange(1, 100)
   a = [random.randrange(1, 100) for i in range(N)]
   print(N, file=f)
   print(*a, file=f)
   f.close()
   ```

4. Είναι όμως καλύτερα να επεξεργάζεται κανείς αρχεία με την εντολή `with`, η οποία φροντίζει να κλείσει σωστά το αρχείο αν προκληθεί κάποια εξαίρεση. Επίσης, με αυτήν δε χρειάζεται κανείς να καλέσει τη μέθοδο `close` ξεχωριστά.

   ```
   with open("data.txt", "wt") as f:
       N = random.randrange(1, 100)
       a = [random.randrange(1, 100) for i in range(N)]
       print(N, file=f)
       print(*a, file=f)
   ```

5. Χρησιμοποιώντας τα τυχαία δεδομένα που κατασκευάζουμε, μπορούμε να ελέγξουμε ότι οι δύο λύσεις μας για το πρόβλημα "Κυλικείο" (η μη αποδοτική `solve_slow` και η αποδοτική `solve_fast`) δίνουν το ίδιο αποτέλεσμα.

   ```
   def solve_slow(a):
       N = len(a)
       count = 0
       for i in range(N):
           i_am_the_tallest = True
           for j in range(i+1, N):
               if a[i] <= a[j]:
                   i_am_the_tallest = False
                   break
           if i_am_the_tallest:
               count += 1
       return count

   def solve_fast(a):
       N = len(a)
       count = 0
   ```

```
    maxSoFar = -1
    for x in reversed(a):
        if x > maxSoFar:
            maxSoFar = x
            count += 1
    return count

for test in range(10000):
    N = random.randrange(1, 1000)
    a = [random.randrange(1, 10000) for i in range(N)]
    if solve_slow(a) != solve_fast(a):
        print("The two solutions disagree here:", a)
    print("done")
```

Λύσεις brute force, το πρόβλημα "partition"

1. Σε κάποια προβλήματα δυστυχώς δεν υπάρχουν αποδοτικές λύσεις και είμαστε υποχρεωμένοι να ελέγξουμε όλους τους δυνατούς τρόπους επίλυσής τους έναν-έναν. Παράδειγμα τέτοιου προβλήματος είναι το πρόβλημα "partition":

   > Έστω S ένα πολυσύνολο (multiset) ακέραιων αριθμών. (*Πολυσύνολο* ονομάζεται ένα σύνολο που, εν αντιθέσει με τα κανονικά σύνολα, μπορεί να περιέχει πολλές φορές το ίδιο στοιχείο.) Ζητείται να διαμοιραστεί το S σε δύο πολυσύνολα S_1 και S_2 (δηλαδή $S_1 \cup S_2 = S$) τέτοια ώστε το άθροισμα των στοιχείων του S_1 να είναι ίσο με το άθροισμα των στοιχείων του S_2.

   Σημείωση: Το πρόβλημα αυτό είναι NP-complete και υπάρχει ψευδοπολυωνυμικός αλγόριθμος για την επίλυσή του, με χρήση δυναμικού προγραμματισμού.

2. Θα χρησιμοποιήσουμε τη γεννήτρια συνδυασμών της βιβλιοθήκης της Python για να λύσουμε το πρόβλημα "partition", ελέγχοντας όλους τους δυνατούς τρόπους με τους οποίους μπορούν να κατασκευαστούν τα S_1 και S_2. Χωρίς βλάβη της γενικότητας, θα υποθέσουμε ότι το S_1 δεν έχει περισσότερα στοιχεία από το S_2. Αυτό σημαίνει ότι αν ο πληθάριθμος του S_1 είναι ίσος με k και ο πληθάριθμος του S είναι ίσος με n, τότε $2k \leq n$.

```
def partition(L):
    s = sum(L)
    n = len(L)
    for k in range(n//2+1):
        for b in itertools.combinations(L, k):
            if 2 * sum(b) == s: return b

print(partition([1, 2, 3, 4, 2]))           # (2, 4)
print(partition([1, 4, 3, 8, 19, 12, 27]))  # None
```

Αντικείμενα

1. Απλές κλάσεις και αντικείμενα.

```
class person:
    def __init__(self, name, phone):
        self.name = name
        self.phone = phone

    def call(self, message):
        print("Calling", self.phone)
        print("Driiiiinnnn")
        print("Hi", self.name)
        print(message)

p = person("Nikos", "123")
p.call("It's already 6 o'clock, go home!")
```

2. Τα αντικείμενα περιέχουν **πεδία** και **μεθόδους**. Τα πεδία δε χρειάζεται να δηλώνονται, μπορούμε απλώς να αναθέτουμε τιμές σε αυτά. Όλες οι μέθοδοι, που ορίζονται με `def` μέσα στις κλάσεις, δέχονται ως πρώτη παράμετρο το αντικείμενο για το οποίο καλούνται. Κατά σύμβαση, την παράμετρο που αντιστοιχεί σε αυτό το αντικείμενο την ονομάζουμε `self`.

3. Η ειδική μέθοδος `__init__` είναι ο **κατασκευαστής**. Καλείται αυτόματα όταν κατασκευάζεται ένα νέο αντικείμενο, π.χ. στη γραμμή `p = person("Nikos", "123")` παραπάνω. Ο εν λόγω κατασκευαστής αναθέτει τις δυο παραμέτρους του (πλην του self) στα αντίστοιχα πεδία του τρέχοντος αντικειμένου.

4. Υπάρχουν και άλλες ειδικές μέθοδοι, όπως π.χ. η `__str__` που αναλαμβάνει τη μετατροπή ενός αντικειμένου σε συμβολοσειρά, κυρίως για να διευκολύνεται η εκτύπωση. Για την κλάση `person` θα μπορούσε να οριστεί ως εξής:

   ```
   def __str__(self):
       return self.name + " with phone " + str(self.phone)
   ```

   Στη συνέχεια, π.χ. η εκτύπωση ενός αντικειμένου χρησιμοποιεί αυτή τη μέθοδο:

   ```
   p = person("Nikos", "123")
   print(p)   # prints "Nikos with phone 123"
   ```

   Χωρίς να οριστεί αυτή η μέθοδος, η εκτύπωση του αντικειμένου θα είχε ως αποτέλεσμα να εμφανιστεί κάτι σαν το παρακάτω:

   ```
   <__main__.person object at 0x1108ffb20>
   ```

   το οποίο ονομάζει την κλάση από την οποία προέρχεται το αντικείμενο και την "ταυτότητά του" (ουσιαστικά τη θέση μνήμης στην οποία βρίσκεται). Αυτή η ταυτότητα ενός αντικειμένου, με τη μορφή ακέραιου αριθμού, είναι το αποτέλεσμα της συνάρτησης `id`:

   ```
   print(id(p))  # prints 4572838688 (or 0x1108ffb20 in hexadecimal)
   ```

5. Τα αντικείμενα είναι mutable, δηλαδή τα περιεχόμενά τους μπορούν να μεταβάλλονται.

   ```
   p = person("Nikos", "123")
   print(p.phone)               # prints "123"
   p.phone = "456"
   print(p.phone)               # prints "456"
   ```

   Επίσης, δεν υπάρχει δυνατότητα περιορισμού της πρόσβασης στα πεδία ή τις μεθόδους των αντικειμένων: είναι όλα δημόσια και ορατά.

6. Η ισότητα των αντικειμένων είναι άλλη μία ιδιότητα που μπορεί να οριστεί για τη νέα κλάση που ορίζουμε. Ας θυμίσουμε πρώτα ότι οι μεταβλητές περιέχουν αναφορές σε αντικείμενα και όχι τα ίδια τα αντικείμενα:

   ```
   p = person("Nikos", "123")
   q = p
   print(p == q)          # prints "True"
   print(id(p) == id(q))  # prints "True"
   print(p is q)          # prints "True"
   ```

   Τα `p` και `q` αναφέρονται στο ίδιο αντικείμενο — η ανάθεση `q = p` δε δημιουργεί αντίγραφο του αντικειμένου.

   Η default υλοποίηση της ισότητας αντικειμένων στην Python απλά συγκρίνει τις ταυτότητες των αντικειμένων. Αυτού του είδους η ισότητα ονομάζεται **φυσική ισότητα** και μπορεί να ελεγχθεί είτε συγκρίνοντας τις ταυτότητες, είτε με τον ειδικό τελεστή `is`, όπως φαίνεται παραπάνω.

   Για την κλάση `person` θα μπορούσαμε να ορίσουμε την ισότητα με διαφορετικό τρόπο, π.χ. να συγκρίνονται τα *περιεχόμενα* των αντικειμένων. Αυτό τώρα δε συμβαίνει, δηλαδή:

   ```
   p = person("Nikos", "123")
   q = person("Nikos", "123")  # an object with the same contents
   print(p == q)               # prints "False"
   print(id(p) == id(q))       # prints "False"
   print(p is q)               # prints "False"
   ```

   Αν όμως ορίσουμε στην κλάση `person` τη μέθοδο:

   ```
   def __eq__(self, other):
       return self.name == other.name and self.phone == other.phone
   ```

   τότε θα χρησιμοποιείται αυτή, αντί της φυσική ισότητας, όταν γίνεται έλεγχος ισότητας:

   ```
   p = person("Nikos", "123")
   q = person("Nikos", "123")  # an object with the same contents
   print(p == q)               # prints "True"
   print(id(p) == id(q))       # prints "False"
   ```

```
print(p is q)               # prints "False"

r = person("Maria", "456")
print(p == r)               # prints "False"
```

Συνήθως όταν γράφουμε υλοποιήσεις της ισότητας πρέπει να φροντίζουμε να δουλεύουν σωστά ακόμα κι όταν συγκρίνουμε αντικείμενα που δεν ανήκουν στην κλάση μας. Για παράδειγμα, με την παραπάνω υλοποίηση, ο έλεγχος:

```
print(p == "hello")
```

θα οδηγούσε σε σφάλμα εκτέλεσης (εξαίρεση), καθώς η μέθοδος `__eq__` θα προσπαθούσε να συγκρίνει τα πεδία `name` και `phone` της συμβολοσειράς `"hello"`. Θα ήταν σωστότερο να έχει υλοποιηθεί ως εξής:

```
def __eq__(self, other):
    return isinstance(other, person) and \
           self.name == other.name and self.phone == other.phone
```

Τώρα η παραπάνω σύγκριση θα έχει ως αποτέλεσμα `False`.