Γλώσσες Προγραμματισμού Ι

Παραδόσεις 7/5 και 8/5/2020, Νίκος Παπασπύρου.

Οι σημειώσεις αυτές περιέχουν περισσότερη ύλη από αυτήν που καλύψαμε στις παραδόσεις. Δείτε και τα αντίστοιχα Jupyter notebooks, στη σελίδα του μαθήματος.

Εισαγωγή στις γλώσσες σεναρίων (scripting languages)

Βλ. π.χ. Κεφάλαιο 13 του βιβλίου *Πραγματολογία των Γλωσσών Προγραμματισμού* του Michael L. Scott.

- Κοινά χαρακτηριστικά
  - Χρήση μαζική (batch) αλλά και διαδραστική (interactive)
  - Οικονομία έκφρασης
  - Απουσία δηλώσεων, απλοί κανόνες εμβέλειας
  - Ευέλικτο, δυναμικό σύστημα τύπων
  - Εύκολη πρόσβαση σε άλλα προγράμματα
  - Πολύπλοκο ταίριασμα προτύπων (pattern matching) και επεξεργασία συμβολοσειρών
  - Τύποι δεδομένων υψηλού επιπέδου
- Περιοχές εφαρμογών
  - Γλώσσες εντολών φλοιού (shell command languages)
    - `{c,tc,k,ba,z}?sh` — δηλαδή `sh`, `csh`, `tcsh`, `ksh`, `bash`, `zsh`
  - Επεξεργασία κειμένου και παραγωγή εκθέσεων
    - `sed`, `awk`, Perl
  - Μαθηματικά και στατιστική
    - "3M" (Maple, Mathematica, Matlab), S, R
  - Γλώσσες "συγκόλλησης" (glue) και σενάρια γενικού σκοπού
    - Tcl, Python, Ruby
  - Γλώσσες επέκτασης (extension languages)
    - JavaScript, Visual Basic, AppleScript, Emacs Lisp
  - Σενάρια στον παγκόσμιο ιστό
    - CGI (Perl, …)
    - Server-side embedded (PHP, Visual Basic, …)
    - Client-side (JavaScript, …)
    - Applets (Java, …)
    - διάφορες άλλες τεχνολογίες (XSLT, XML, …)

Εισαγωγή στην Python 3

0. Documentation και Anaconda
1. Γεια σου κόσμε!

   ```
   print("Hello world!")
   ```

2. "Εκτελέσιμα" προγράμματα Python

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

   print("Hello world!")
   ```

   και φροντίστε να είναι εκτελέσιμο το αρχείο όπου θα αποθηκεύσετε το script (`chmod +x` σε Unix/Linux).

Αριθμοί, συμβολοσειρές και πράξεις

1. Ακέραιοι αριθμοί (`int`)

   ```
   42
   42 - 17
   -1
   6 * 7
   42 // 17     # πηλίκο ακέραιας διαίρεσης (= 2)
   42 % 17      # υπόλοιπο ακέραιας διαίρεσης (= 8)
   2 ** 78      # ύψωση σε δύναμη: ακέραιοι απεριόριστου μεγέθους!
   ```

2. Πραγματικοί αριθμοί (`float`)

```
42.0         # διαφορετικό από το 42, float αντί int
3.14
-5.3
6.0 * 7      # = 42.0, όχι 42
42 / 17      # πραγματική διαίρεση (= 2.4705882352941178)
42.0 / 17.0  # το ίδιο
12.2 // 3.1  # = 3.0 (πόσες φορές χωράει το 3.1 στο 12.3)
12.2 % 3.1   # = 2.9 (και τι περισσεύει)
```

Στη πραγματικότητα, το τελευταίο δεν έχει ως αποτέλεσμα `2.9` αλλά `2.899999999999999`. Αυτό οφείλεται σε περιορισμούς στην ακρίβεια των πράξεων και αριθμητικά σφάλματα: οι υλοποιήσεις της Python δεν φημίζονται για την ακρίβεια στους αριθμητικούς υπολογισμούς μεταξύ πραγματικών αριθμών.

3. Λογικές τιμές (`bool`)

```
True
False
not True         # λογική άρνηση   (= False)
True and False   # λογική σύζευξη  (= False)
False or True    # λογική διάζευξη (= True)
```

Στην πραγματικότητα είναι `True` = 1 και `False` = 0.

```
1 + False        # = 1
5 * True         # = 5
```

4. Συγκρίσεις

```
42 == 17         # ισότητα (= False)
42 == 42.0       # = True (!) αλλά μη χρησιμοποιείτε == για πραγματικούς
42 != 17         # διάφορο (= True)
17 < 42          # μικρότερο (= True)
19 <= 18         # μικρότερο ή ίσο (= False)
17 > 42          # μεγαλύτερο (= False)
19 >= 18         # μεγαλύτερο ή ίσο (= True)
```

5. Συμβολοσειρές

```
"In double quotes"
'or in single quotes'

"""Multiline strings
   You must try this!"""

'''One of the simplest programs in Python 3 is:
   print("Hello world!")'''
```

6. Σχόλια

   - Μίας γραμμής:

     ```
     # this is a comment
     n = 42  # and so is this
     ```

   - Πολλών γραμμών δεν υπάρχουν, αλλά οι συμβολοσειρές δουλεύουν και ως σχόλια:

     ```
     """This string serves the purpose of
     a multiline comment in the program that follows"""
     print("Hello world!")
     ```

7. Μετατροπές

```
int(3.14)        # αποκοπή δεκαδικού μέρους (= 3)
int(2.718)       # = 2
float(42)        # 42.0 (!)
round(3.14)      # στρογγυλοποίηση (= 3)
round(2.718)     # = 3
```

```
str(3.14)        # = "3.14"
int("42")        # = 42
```

Υπάρχουν επίσης πολλές built-in συναρτήσεις, όπως οι `abs`, `len`, `max`, `min`, κ.λπ., καθώς και συναρτήσεις στο module `math`, όπως οι `floor`, `ceil`, `gcd`, κ.λπ.

Ροή ελέγχου

1. Οι εντολές κανονικά εκτελούνται με τη σειρά

   ```
   print("First do this.")
   print("Then do that.")
   ```

2. Η εντολή `if`

   ```
   a = 3
   if a < 10:
       print("It is a small number")
   ```

3. Η εντολή `print` σε περισσότερο βάθος

   ```
   a = 3
   if a < 10:
       print("The number", a, "is a small number")
   ```

   Για να μην εκτυπώνονται κενά μεταξύ τους:

   ```
   print(4, 2, sep="")               # prints "42"
   ```

   ή για να εκτυπώνεται οτιδήποτε άλλο:

   ```
   print(4, 2, sep=" and then ")     # prints "4 and then 2"
   ```

   Για να μην αλλάζει γραμμή στο τέλος:

   ```
   print(4, end="")
   print(2)                          # these two print "42"
   ```

4. Η εντολή `for`

   ```
   for a in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]:
       if a < 10:
           print("The number", a, "is a small number")
   ```

5. Η εντολή `if` με `else`

   ```
   for a in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]:
       if a < 10:
           print("The number", a, "is a small number")
       else:
           print("But", a, "is a big number")
   ```

6. Μαθαίνω να μετράω: `for` και `range`

   - Από 0 έως και 9

     ```
     for i in range(10):
         print(i)
     ```

   - Από 17 έως και 41

     ```
     for i in range(17, 42):
         print(i)
     ```

   - Από 17 έως και 41 αλλά μόνο τους περιττούς:

     ```
     for i in range(17, 42):
         if i % 2 == 1:
             print(i)
     ```

   - Από 17 έως και 41 με βήμα 2 (πάλι μονο τους περιττούς)

```
for i in range(17, 42, 2):
    print(i)
```

- Από 42 έως και 18 αντίστροφα με βήμα 2

```
for i in range(42, 17, -2):
    print(i)
```

7. Η στοίχιση (indentation) είναι σημαντική!

```
for i in range(17, 42):
    if i % 2 == 1:
        print(i)
        print("just printed an odd number...")

for i in range(17, 42):
    if i % 2 == 1:
        print(i)
    print("still counting numbers...")

for i in range(17, 42):
    if i % 2 == 1:
        print(i)
print("done counting...")
```

Μεταβλητές

1. Untyped (με δυναμικούς τύπους) και χωρίς δηλώσεις μεταβλητών:

```
a = 42
print(a+1)     # prints "43"

a = "hello"    # η ίδια μεταβλητή παίρνει τώρα τιμή άλλου τύπου!
print(len(a))  # prints "5" (the length of "hello" in characters)
```

2. Πολλαπλή ανάθεση όπως στη C/C++ και στη Java:

```
a = b = c = 0
print(a, b, c)
```

3. Πολλαπλή ανάθεση με περισσότερες μεταβλητές στο αριστερό και στο δεξιό μέλος:

```
a, b, c = 1, 2, 3
print(a, b, c)
```

4. Αντιμετάθεση (swap) χωρίς βοηθητική μεταβλητή

```
a, b = 17, 42
a, b = b, a       # this indeed works!
print(a, b)
```

Συναρτήσεις

0. Μια απλή συνάρτηση που επιστρέφει τον επόμενο αριθμό

```
def f(x):
    return x+1
```

1. Παράδειγμα: αριθμοί Fibonacci (0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …)
2. Αριθμοί Fibonacci με αναδρομή: κακή ιδέα! – \(O(n^2)\)

```
def fib(n):
    if n <= 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

```
print(fib(7))    # prints "13"
print(fib(37))   # prints "24157817" (but after ~20 sec.)
print(fib(100))  # will not print anything for a VERY long time!
```

3. Υπολογισμός του n-οστού όρου της ακολουθίας Fibonacci

```
def fib(n):
    if n == 0: return 0
    a, b = 0, 1
    for i in range(n-1):
        c = a+b
        a, b = b, c
    return b

print(fib(7))                   # prints "13"
print(fib(1000))                # prints a huge number with 209 digits
print(len(str(fib(100000))))    # prints "20899", the number of digits
                                # of the 100000th Fibonacci number
```

4. Υπολογισμός του μικρότερου αριθμού Fibonacci που είναι μεγαλύτερος του n:

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

print(fib_gt(42))   # prints "55"
```

Συναρτήσεις και μετατροπή τύπων παραμέτρων

1. Οι παράμετροι έχουν δυναμικούς τύπους. Αν μια συνάρτηση θέλει να ελέγξει τους τύπους των παραμέτρων της, μπορεί να το κάνει π.χ. με τη συνάρτηση `type`. Οι συναρτήσεις `int` και `str` μετατρέπουν σε ακέραιο και string αντίστοιχα.

```
def fib(n):
    convert = False
    if type(n) == str:
        n = int(n)
        convert = True
    a, b = 0, 1
    if n == 0: return 0
    for i in range(n-1):
        c = a+b
        a, b = b, c
    if convert:
        return str(b)
    return b

print(fib(42))    # the result is the number 267914296
print(fib("42"))  # the result is the string "267914296"
```

2. Το τελευταίο `if` στο παραπάνω πρόγραμμα μπορεί να γραφεί:

```
return str(b) if convert else b
```

που είναι αντίστοιχο του τελεστή `a ? b : c` στη C/C++. Στην Python θα το γράφαμε `b if a else c`.

3. Έλεγχος τύπων: η συνάρτηση `type` επιστρέφει τον τύπο μίας τιμής. Π.χ.

```
type(42) == int
type("hello") == str
type(True) == bool
```

4. Καλύτερος έλεγχος τύπων που ακολουθεί την ιεραρχία των κλάσεων της Python: με τη συνάρτηση `isinstance`. Η διαφορά φαίνεται με τιμές του τύπου `bool`, ο οποίος όπως έχουμε πει είναι υποτύπος του `int`.

```
type(True) == bool      # επιστρέφει True
type(True) == int       # επιστρέφει False
isinstance(True, int)   # επιστρέφει True
isinstance(True, bool)  # επιστρέφει True, επίσης!
```

   Η συνάρτηση `isinstance` μπορεί να χρησιμοποιηθεί και για να ελέγξει αν μία τιμή ανήκει σε κάποιον από πολλούς εναλλακτικούς τύπους:

```
isinstance(x, (int, float, str))
```

5. Χρησιμοποιώντας την `isinstance`, η παραπάνω συνάρτηση `fib` γράφεται ως εξής:

```
def fib(n):
    convert = False
    if isinstance(n, str):
        n = int(n)
        convert = True
    a, b = 0, 1
    if n == 0: return 0
    for i in range(n-1):
        c = a+b
        a, b = b, c
    return str(b) if convert else b
```

Συναρτήσεις και default τιμές παραμέτρων

1. Γενικευμένοι αριθμοί Fibonacci. Οι δύο πρώτοι όροι της ακολουθίας είναι οι παράμετροι `start_a` και `start_b`.

```
def fib(n, start_a=0, start_b=1):
    a, b = start_a, start_b
    if n == 0: return 0
    for i in range(n-1):
        c = a+b
        a, b = b, c
    return b
```

2. Τώρα μπορεί να κληθεί με οποιονδήποτε από τους παρακάτω τρόπους:

```
print(fib(100))           # start_a = 0, start_b = 1
print(fib(100, 17, 42))
print(fib(100, 17))       # start_b = 1
```

3. Επίσης, μπορεί κανείς να ονομάζει τις παραμέτρους κατά την κλήση και να τις δίνει με διαφορετική σειρά:

```
print(fib(100, start_b=42, start_a=17))
print(fib(100, start_b=42))              # start_a = 0
print(fib(start_a=17, n=100))            # start_b = 1
```

Εμβέλεια μεταβλητών

1. Τοπικές και καθολικές (global) μεταβλητές:

```
a = 17
print(a)

def fib(n):
    a, b = 0, 1
    if n == 0: return 0
    for i in range(n-1):
        c = a+b
        a, b = b, c
    return b
```

```
print(fib(7))
print(a)
print(b)
```

Το εξωτερικό a (global) είναι διαφορετικό από το εσωτερικό a (local).

2. Κανόνες εμβέλειας

```
a = 42              # this a is global

def reasonable():
    print(a)

def what():
    a = 17
    print(a)

reasonable()        # prints 42
print(a)            # prints 42
what()              # prints 17
print(a)            # prints 42, what???
```

Η διαφορά βρίσκεται στο ότι η συνάρτηση `reasonable` δεν αναθέτει στη μεταβλητή `a`, ενώ η `what` αναθέτει. Η ανάθεση κάνει την Python να πιστεύει ότι η μεταβλητή πρέπει να είναι local στη `what`. Αυτό μπορεί να διορθωθεί με χρήση του `global`:

```
a = 42              # this a is global

def what():
    global a
    a = 17
    print(a)

what()              # prints 17
print(a)            # prints 17
```

3. Περισσότερα επίπεδα εμβέλειας:

```
def foo(n):
    print(n)
    def bar():
        n = 17
        print(n)
    bar()
    print(n)

foo(42)
```

τυπώνει:

```
42
17
42
```

4. Το παραπάνω με χρήση του `global`:

```
def foo(n):
    print(n)
    def bar():
        global n
        n = 17
        print(n)
    bar()
    print(n)

foo(42)
```

```
print(n)
```

τυπώνει:

```
42
17
42
17
```

5. Το ίδιο με χρήση του `nonlocal`:

```
def foo(n):
    print(n)
    def bar():
        nonlocal n
        n = 39
        print(n)
    bar()
    print(n)

n = 17
foo(42)
print(n)
```

τυπώνει:

```
42
39
39
17
```

Προσέξτε ότι η μεταβλητή στην οποία ανατέθηκε το 39 ήταν η τοπική της `foo`, όχι η global, η οποία παρέμεινε ανέπαφη.

Συμβολοσειρές

1. Αποτελούνται από χαρακτήρες. Η αρίθμηση ξεκινάει από το μηδέν!

```
s = "hello world"
print(s[4])           # prints 'o'
```

2. Τα περιεχόμενά τους δεν αλλάζουν (strings are immutable)

```
s[4] = 'x'            # ERROR!
```

αλλά αυτό είναι OK (φτιάχνει νέο string):

```
s = s[:4] + "x" + s[5:]
print(s)
```

3. "Φέτες" (slices)

```
print(s[:4])      # "hell"        --- from start until 4
print(s[4:])      # "o world"     --- from 4 until end
print(s[4:8])     # "o wo"        --- from 4 until 8
print(s[::2])     # "hlowrd"      --- every second char (step = 2)
print(s[::-1])    # "dlrow olleh" --- in reverse (step = -1)

def palindrome(s):
    return s == s[::-1]   # that was quick...
```

Πλειάδες (tuples)

Αντιπαράβαλε με τα παραπάνω (strings):

1. Αποτελούνται από οτιδήποτε

```
s = (1, 2, "what", 3)
print(s[1])               # prints 2
```

```
s = tuple("hello world")  # ('h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd')
print(s[4])               # prints 'o'
```

2. Τα περιεχόμενά τους δεν αλλάζουν (tuples are immutable)

   ```
   s[4] = 'x'            # ERROR!
   ```

   αλλά αυτό είναι OK (φτιάχνει νέο tuple):

   ```
   s = s[:4] + ("x",) + s[5:]
   print(s)
   ```

   Προσέξτε το κόμμα στο `("x",)` που ορίζει ένα tuple με μήκος 1.

3. "Φέτες" (slices)

   ```
   print(s[:4])      # ('h', 'e', 'l', 'l')
   print(s[4:])      # ('o', ' ', 'w', 'o', 'r', 'l', 'd')
   print(s[4:8])     # ('o', ' ', 'w', 'o')
   print(s[::2])     # ('h', 'l', 'o', 'w', 'r', 'd')
   print(s[::-1])    # ('d', 'l', 'r', 'o', 'w', ' ', 'o', 'l', 'l', 'e', 'h')
   ```

Λίστες (lists)

Αντιπαράβαλε με τα παραπάνω (strings και tuples):

1. Αποτελούνται από χαρακτήρες

   ```
   s = [1, 2, "what", 3]
   print(s[1])               # prints 2

   s = list("hello world")   # ['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
   print(s[4])               # prints 'o'
   ```

2. Τα περιεχόμενά τους αλλάζουν (lists are mutable)

   ```
   s[4] = 'x'            # OK!
   print(s)
   ```

   και αυτό είναι OK αλλά φτιάχνει νέα λίστα:

   ```
   s = s[:4] + ["x"] + s[5:]
   print(s)
   ```

3. "Φέτες" (slices)

   ```
   print(s[:4])      # ['h', 'e', 'l', 'l']
   print(s[4:])      # ['o', ' ', 'w', 'o', 'r', 'l', 'd']
   print(s[4:8])     # ['o', ' ', 'w', 'o']
   print(s[::2])     # ['h', 'l', 'o', 'w', 'r', 'd']
   print(s[::-1])    # ['d', 'l', 'r', 'o', 'w', ' ', 'o', 'l', 'l', 'e', 'h']
   ```

4. Μετατροπή λίστας σε string (συνένωση πολλών strings)

   ```
   print("".join(s))   # "hello world"
   print("-".join(s))  # "h-e-l-l-o- -w-o-r-l-d"
   ```

Γεννήτριες (generators)

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. Η έκφραση `i*i for i in range(10)` παραπάνω είναι μία γεννήτρια (generator), όπως και το ίδιο το `range(10)`. Γεννάει μία ακολουθία από τιμές, τις οποίες μπορούμε να τοποθετήσουμε σε μία λίστα ή άλλη δομή δεδομένων, να τις διατρέξουμε με ένα `for` loop, κ.λπ. Η γεννήτρια αυτή αντιστοιχεί στο μαθηματικό σύνολο $\{i^2 \mid i \in \{0..9\}\}$.

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

```
def gen_fib(n):
    a, b = 0, 1
    yield 0
    for i in range(n-1):
        c = a+b
        a, b = b, c
        yield b

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

4. Μπορεί επίσης να κατασκευάσει "άπειρες" γεννήτριες, οι οποίες παράγουν διαρκώς τιμές:

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

Ο παρακάτω κώδικας τυπώνει τους πρώτους 100 αριθμούς Fibonacci (γιατί σταματά με το `break`), και στη συνέχεια τυπώνει τον 101ο με χρήση της συνάρτησης `next`, που φέρνει το επόμενο στοιχείο μίας γεννήτριας.

```
i = 0
g = gen_all_fib()
for x in g:
    print(x)
    i += 1
    if i >= 100: break
print(next(g))
```

Λεξικά (dictionaries)

1. Γνωστά και ως συσχετιστικοί πίνακες (associative arrays). Είναι ουσιαστικά arrays που τα στοιχεία τους προσπελάζονται μέσω κλειδιών (keys) οποιουδήποτε τύπου — όχι μόνο με ακέραιους όπως οι συμβολοσειρές, οι πλειάδες και οι λίστες.

```
d = {}
d["yes"] = "ναι"
d["no"] = "όχι"

print(d["yes"])
```

Το παρακάτω προκαλεί εξαίρεση τύπου `KeyError`:

```
print(d["maybe"])
```

2. Οι τύποι των κλειδιών μπορούν να είναι (σχεδόν) οτιδήποτε:

```
d = {
  "maybe": "ίσως",
  "why not?": "γιατί όχι;"
}
d[1, 2, 3] = "ναι"
d[4, 5, 6, 7] = "όχι"
print(d[4, 5, 6, 7])
```

Πρέπει όμως είτε να είναι immutable (δηλαδή όχι λίστες ή άλλα λεξικά) είτε να ορίζουν τις μεθόδους `__hash__` και `__eq__`.

3. Μπορούμε να διατρέξουμε όλα τα κλειδιά ενός λεξικού:

```
for key in d:
    print(key)
```

και να τυπώσουμε και τις τιμές:

```
for key in d:
    print(key, d[key])
```

ή καλύτερα:

```
for key, value in d.items():
    print(key, value)
```

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. 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)}
```

4. 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
```

Εξαιρέσεις

1. Η χρήση του `d["haha"]` σε ένα λεξικό `d` που δεν περιέχει κλειδί `"haha"` προκαλεί μία εξαίρεση τύπου `KeyError`. Η εξαίρεση αυτή μπορεί να προκληθεί και ρητά, με την εντολή `raise`:

```
raise KeyError
```

2. Η Python υποστηρίζει ένα μηχανισμό χειρισμού εξαιρέσεων, με τη δομή `try ... except ...`

```
try:
    print(d["oui"])
    print(d["haha"])
    print(d["non"])
except:
    print("something wrong happened")
```

Το παραπάνω τυπώνει:

```
yes
something wrong happened
```

γιατί όταν προκαλείται η εξαίρεση `KeyError` στη δεύτερη γραμμή του `try`, η ροή ελέγχου μεταφέρεται στο σκέλος `except` και το πρόγραμμα συνεχίζει από εκεί.

3. Χειρισμός συγκεκριμένων τύπων εξαιρέσεων:

```
try:
    print(d["oui"])
    print(d["haha"])
    print(d["non"])
except KeyError:
    print("something wrong happened")
```

Όμως, αυτό το σκέλος `except` δε θα πιάσει κάποια εξαίρεση άλλου τύπου, όπως για παράδειγμα μία διαίρεση με το μηδέν:

```
try:
    print(d["oui"])
    print(1//0)
    print(d["haha"])
    print(d["non"])
except KeyError:
    print("something wrong happened")
```

Για περισσότερους τύπους εξαίρεσης:

```
try:
    print(d["oui"])
    print(1//0)
    print(d["haha"])
    print(d["non"])
except (KeyError, ZeroDivisionError):
    print("something wrong happened")
```

ή και για διαφορετικό τρόπο χειρισμού κάθε μίας:

```
try:
    print(d["oui"])
    print(1//0)
    print(d["haha"])
    print(d["non"])
except KeyError:
    print("something wrong happened")
except ZeroDivisionError:
    print("something wrong with your math")
```

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

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 " + self.phone
```

Εξερεύνηση χώρου καταστάσεων

Ο γρίφος του **λύκου, της κατσίκας και του λάχανου** (γνωστός επίσης και με διαφορετικούς συνδυασμούς ζώων και ζαρζαβατικών) διατυπώνεται ως εξής:

> A farmer wants to cross a river and take with him a wolf, a goat, and a cabbage.
>
> There is a boat that can fit himself plus either the wolf, the goat, or the cabbage.
>
> If the wolf and the goat are alone on one shore, the wolf will eat the goat. If the goat and the cabbage are alone on the shore, the goat will eat the cabbage.

How can the farmer bring the wolf, the goat, and the cabbage across the river?

1. Θέλουμε να γράψουμε ένα πρόγραμμα που θα βρίσκει τη λύση του γρίφου, αντιμετωπίζοντας τον σαν ένα πρόβλημα εξερεύνησης ενός χώρου καταστάσεων. Οι καταστάσεις του προβλήματος περιγράφουν πού βρίσκονται οι τέσσερις πρωταγωνιστές του γρίφου. Στην αρχική κάτασταση, όλοι βρίσκονται στην αριστερή όχθη, και στην τελική (επιδιωκόμενη) κατάσταση, όλοι βρίσκονται στη δεξιά.

2. Θα ορίσουμε μία κλάση `state` για να υλοποιήσουμε τις καταστάσεις.

   ```
   class state:
       def __init__(self, left, right):
           self.left = frozenset(left)
           self.right = frozenset(right)

   init = state(['man', 'cabbage', 'goat', 'wolf'], [])
   ```

   Ο κατασκευαστής της δέχεται δύο παραμέτρους: αυτά που βρίσκονται στην αριστερή όχθη και αυτά που βρίσκονται στη δεξιά. Τα πεδία `left` και `right` των αντικειμένων της κλάσης `state` θα είναι immutable σύνολα (`frozenset`) στα οποία ο κατασκευαστής αναθέτει τις παραμέτρους του.

3. Στη συνέχεια θα ορίσουμε μία μέθοδο `accessible`. Αν `s` είναι μία δοθείσα κατάσταση, τότε `s.accessible()` θα είναι μία γεννήτρια που θα παράγει όλες τις καταστάσεις που είναι "προσβάσιμες" από την `s`. Η κατάσταση `t` είναι προσβάσιμη από την `s` αν η πρώτη μπορεί να προκύψει από τη δεύτερη με μία μόνο μετακίνηση του ανθρώπου (και πιθανώς ενός ακόμη ζώου ή αντικειμένου).

   ```
   def accessible(self):
       if 'man' in self.left:
           for x in self.left:
               moving = frozenset(['man', x])
               yield state(self.left - moving, self.right | moving)
       else:
           for x in self.right:
               moving = frozenset(['man', x])
               yield state(self.left | moving, self.right - moving)
   ```

   Προσέξτε τις πράξεις συνόλων που αναλαμβάνουν τη μετακίνηση από τη μία όχθη στην άλλη των πρωταγωνιστών του γρίφου.

4. Προτού γράψουμε ένα απλό πρόγραμμα που θα επιδεικνύει τη λειτουργία της `accessible`, καλό είναι να υλοποιήσουμε μία μέθοδο που θα αναλάβει την εκτύπωση των καταστάσεων σε μια μορφή εύληπτη για τον άνθρωπο. Χωρίς αυτήν, αν π.χ. στο παραπάνω πρόγραμμα προσθέσουμε:

   ```
   print(init)
   ```

   θα δούμε κάτι σαν το παρακάτω:

   ```
   <__main__.state object at 0x110057be0>
   ```

   που μας δείχνει τον τύπο του αντικειμένου και τη διεύθυνσή του στη μνήμη. Προσθέτουμε λοιπόν την ειδική μέθοδο `__str__`, που αναλαμβάνει τη μετατροπή ενός αντικειμένου σε συμβολοσειρά. Η μέθοδος αυτή καλείται αυτόματα από την `print`.

   ```
   def __str__(self):
       return "left: {}, right: {}".format(
           " & ".join(self.left), " & ".join(self.right)
       )
   ```

   Τώρα η εκτύπωση της αρχικής κατάστασης `init` θα έχει ως αποτέλεσμα:

   ```
   left: cabbage & wolf & man & goat, right:
   ```

5. Ας δούμε τώρα ποιες καταστάσεις `s` είναι προσβάσιμες από την αρχική και ποιες καταστάσεις `t` είναι προσβάσιμες από αυτές τις `s`, αντίστοιχα:

   ```
   for s in init.accessible():
       print(s)
       for t in s.accessible():
           print("  ", t)
   ```

   Το αποτέλεσμα του παραπάνω θα είναι το εξής:

```
left: wolf & goat, right: cabbage & man
   left: cabbage & wolf & man & goat, right:
   left: wolf & man & goat, right: cabbage
left: cabbage & goat, right: wolf & man
   left: cabbage & wolf & man & goat, right:
   left: cabbage & man & goat, right: wolf
left: cabbage & goat & wolf, right: man
   left: cabbage & goat & man & wolf, right:
left: cabbage & wolf, right: goat & man
   left: cabbage & goat & man & wolf, right:
   left: cabbage & man & wolf, right: goat
```

6. Έχουμε μοντελοποιήσει τις καταστάσεις του προβλήματος και τώρα είμαστε έτοιμοι να εξερευνήσουμε τον χώρο των καταστάσεων για να βρούμε τη λύση του γρίφου. Ξεκινώντας από την αρχική κατάσταση, βρήκαμε προηγουμένως ποιες καταστάσεις είναι προσβάσιμες μετά από δύο μετακινήσεις του ανθρώπου. Το αποτέλεσμα αυτό μας επισημαίνει τα εξής:

   - Δεν έχουμε μοντελοποιήσει κανένα χαρακτηρισμό των καταστάσεων. Δύο ενδιαφέρουσες περιπτώσεις καταστάσεων είναι οι εξής:
     - Η τελική κατάσταση, στην οποία όλοι οι πρωταγωνιστές του γρίφου βρίσκονται στη δεξιά όχθη.
     - Μη ασφαλείς καταστάσεις, στις οποίες είτε ο λύκος και η κατσίκα είτε η κατσίκα και το λάχανο βρίσκονται στη μία όχθη του ποταμού, ενώ ο άνθρωπος βρίσκεται στην άλλη.
   - Δεν έχουμε κάποιο συστηματικό τρόπο εξερεύνησης των καταστάσεων. Δε θα πρέπει να σταματάμε μετά από δύο κινήσεις αλλά να εξερευνούμε οσοδήποτε μεγάλο πλήθος κινήσεων, μέχρι να φτάσουμε στην τελική κατάσταση.
   - Κάποιες κινήσεις είναι άσκοπες. Π.χ. στις πρώτες δύο γραμμές της εκτύπωσης, ο άνθρωπος μεταφέρει το λάχανο στη δεξιά όχθη και στη συνέχεια επιστρέφει με αυτό στην αριστερή όχθη. Ο συνδυασμός των δύο αυτών κινήσεων είναι άσκοπος γιατί η κατάσταση που προκύπτει ως αποτέλεσμα είναι η αρχική. Κατά τη διάρκεια της εξερεύνησης, πρέπει να θυμόμαστε ποιες καταστάσεις έχουμε ήδη επισκεφθεί.

7. Πρώτα θα ορίσουμε τη μέθοδο `success` που θα επιστρέφει `True` αν μία κατάσταση είναι τελική.

   ```
   def success(self):
       return not self.left
   ```

   Η μέθοδος ελέγχει αν δεν υπάρχει κανείς στην αριστερή όχθη. Από αυτό συμπεραίνουμε ότι (εφόσον οι κινήσεις που γίνονται είναι νόμιμες) όλοι οι πρωταγωνιστές του γρίφου βρίσκονται στη δεξιά όχθη.

8. Στη συνέχεια θα ορίσουμε τη μέθοδο `safe` που θα επιστρέφει `True` αν μία κατάσταση είναι ασφαλής.

   ```
   def safe(self):
       def tragic(bank):
           BAD = [('cabbage', 'goat'), ('wolf', 'goat')]
           return 'man' not in bank and \
                   any(frozenset(pair) <= bank for pair in BAD)
       return not tragic(self.left) and not tragic(self.right)
   ```

   Εδώ ελέγχουμε αν συμβαίνει κάτι τραγικό (συνάρτηση `tragic`) είτε στην αριστερή είτε στη δεξιά όχθη του ποταμού. Το τραγικό θα είναι ο άνθρωπος να μη βρίσκεται σε αυτή την όχθη αλλά να βρίσκεται εκεί ένα από τα "κακά" ζευγάρια του γρίφου, δηλαδή είτε η κατσίκα και το λάχανο είτε ο λύκος και η κατσίκα. Προσέξτε τον τελεστή `<=` που συμβολίζει τη σχέση υποσυνόλου ($\subseteq$), όταν εφαρμόζεται σε σύνολα.

9. Τώρα μπορούμε να συνδυάσουμε τις δύο αυτές μεθόδους με την εξερεύνηση δύο κινήσεων που γράψαμε, για να αποφύγουμε τις μη ασφαλείς καταστάσεις και για να σταματήσουμε αν η εξερεύνηση μάς οδηγήσει στην τελική κατάσταση (που δυστυχώς δε θα συμβεί τόσο εύκολα).

   ```
   for s in init.accessible():
       if not s.safe(): continue
       print(s)
       for t in s.accessible():
           if not t.safe(): continue
           print("  ", t)
           if t.success(): print("We're finished!")
   ```

   Το αποτέλεσμα του παραπάνω θα είναι τώρα το εξής:

```
left: cabbage & wolf, right: goat & man
   left: goat & cabbage & man & wolf, right:
   left: cabbage & man & wolf, right: goat
```

Αυτό σημαίνει ότι η πρώτη κίνηση είναι υποχρεωτική: ο άνθρωπος πρέπει να μεταφέρει την κατσίκα στη δεξιά όχθη. Στη συνέχεια μπορεί είτε να επιστρέψει με την κατσίκα (αργότερα θα αποκλείσουμε αυτή την κίνηση ως άσκοπη) ή να επιστρέψει μόνος του.

10. Προτού προχωρήσουμε στην πιο συστηματική εξερεύνηση του χώρου των καταστάσεων, θα χρειαστεί να ασχοληθούμε με τη σχέση ισότητας των καταστάσεων. Πότε δύο καταστάσεις είναι ίσες; Οι καταστάσεις είναι αντικείμενα στην Python και την απάντηση σε αυτή την ερώτηση πρέπει να τη δούμε γενικότερα.

    Αν `x` και `y` είναι δύο αντικείμενα, ο έλεγχος ισότητας `x != y` καλεί αυτόματα την ειδική μέθοδο `__eq__` του αντικειμένου `x`, περνώνας ως παράμετρο το αντικείμενο `y`. (Το ίδιο συμβαίνει και με τους άλλους τελεστές σύγκρισης, που έχουν αντίστοιχες μεθόδους.) Για τα προκαθορισμένα αντικείμενα της Python (π.χ. `int`, `str`, λίστες, σύνολα, κ.λπ.), η μέθοδος αυτή είναι ορισμένη να συμπεριφέρεται όπως κανείς περιμένει (π.χ. δύο σύνολα είναι ίσα όταν έχουν ακριβώς τα ίδια στοιχεία). Για αντικείμενα που ορίζουμε εμείς, όπως αυτά της κλάσης `state` παραπάνω, η default μέθοδος `__eq__` κάνει κάτι πολύ απλό: συγκρίνει το `id` των αντικειμένων (δηλαδή τη διεύθυνσή τους) στη μνήμη. Επομένως, αν έχουμε:

    ```
    init1 = state(['man', 'cabbage', 'goat', 'wolf'], [])
    init2 = state(['man', 'cabbage', 'goat', 'wolf'], [])

    print("init1 has id", id(init1))
    print("init2 has id", id(init2))
    print(init1 == init2)
    ```

    το αποτέλεσμα της σύγκρισης θα είναι `False` γιατί πρόκειται για δύο διαφορετικά αντικείμενα (δηλαδή δύο αντικείμενα που κατασκευάστηκαν με δύο διαφορετικές κλήσεις του κατασκευαστή):

    ```
    init1 has id 4426181152
    init2 has id 4426181264
    False
    ```

    Αυτό φυσικά δεν μας εξυπηρετεί, γιατί στο μυαλό μας οι δύο καταστάσεις `init1` και `init2` είναι ίσες, εφόσον οι πρωταγωνιστές του γρίφου βρίσκονται στις ίδιες θέσεις και στις δύο.

11. Για να το διορθώσουμε αυτό, ορίζουμε την ειδική μέθοδο `__eq__` της κλάσης `state` ως εξής:

    ```
    def __eq__(self, other):
        return self.left == other.left
    ```

    Δύο καταστάσεις είναι ίσες αν και μόνο αν τα σύνολα όσων βρίσκονται στην αριστερή όχθη είναι ίσα. Το αποτέλεσμα της παραπάνω σύγκρισης τώρα θα είναι `True`, παρότι τα `id` των δύο αντικειμένων θα είναι διαφορετικά.

12. Προσέξτε ότι στην Python, εν αντιθέσει με τις γλώσσες που διαθέτουν στατικό σύστημα τύπων, οποιαδήποτε δύο αντικείμενα μπορούν να συγκριθούν μεταξύ τους, π.χ. το αποτέλεσμα της έκφρασης `[3] == "hello"` είναι `False`. Η μέθοδος `__eq__` που ορίσαμε παραπάνω θα οδηγήσει σε εξαίρεση, αν το δεύτερο αντικείμενο (`other`) δεν είναι κατάσταση (για την ακρίβεια, αν δε διαθέτει πεδίο `left`). Για να επιτρέπουμε τη σύγκριση και με αντικείμενα άλλων κλάσεων, θα ήταν καλύτερα να ορίσουμε την `__eq__` ως εξής:

    ```
    def __eq__(self, other):
        return isinstance(other, state) and self.left == other.left
    ```

    Η συνάρτηση `isinstance` ελέγχει αν η πρώτη παράμετρος είναι ένα αντικείμενο προερχόμενο από την κλάση που δίνεται στη δεύτερη παράμετρο.

13. Για τη συστηματική εξερεύνηση του χώρου των καταστάσεων θα εφαρμόσουμε την τεχνική της αναζήτησης κατά πλάτος (breadth-first search, BFS). Η βασική ιδέα είναι ότι χρησιμοποιούμε μία **ουρά**, στην οποία αρχικά τοποθετούμε την αρχική κατάσταση. Στη συνέχεια, όσο η ουρά δεν είναι άδεια, αφαιρούμε το πρώτο στοιχείο της ουράς, βρίσκουμε τις καταστάσεις που είναι προσβάσιμες από αυτήν και τις προσθέτουμε στο τέλος της ουράς. Με τον τρόπο αυτό εξασφαλίζουμε ότι θα επισκεφθούμε όλες τις καταστάσεις με τις ελάχιστες δυνατές κινήσεις, ξεκινώντας από την αρχική.

    Ορίζουμε μία συνάρτηση `solve` που εφαρμόζει τον αλγόριθμο του BFS όπως παραπάνω. Για την υλοποίηση της ουράς, χρησιμοποιεί την κλάση `deque` που ορίζεται στο module `collections` της βιβλιοθήκης της Python. Η συνάρτηση `solve` θα επιστρέψει την τελική κατάσταση, αν τη βρει. Αν αντίθετα η ουρά αδειάσει χωρίς να φτάσουμε στην τελική κατάσταση, τότε η συνάρτηση θα επιστρέψει χωρίς κάποια τιμή (δηλαδή θα επιστρέψει την τιμή `None`).

```
from collections import deque

def solve():
    init = state(['man', 'cabbage', 'goat', 'wolf'], [])
    Q = deque([init])
    while Q:
        s = Q.popleft()
        for t in s.accessible():
            if t.success():
                return t
            if t.safe():
                Q.append(t)

print(solve())
```

Το πρόγραμμά μας τώρα είναι σε θέση να βρει την τελική κατάσταση. Εκτυπώνει:

```
left: , right: goat & man & wolf & cabbage
```

14. Στη συστηματική αναζήτηση του χώρου των καταστάσεων έχουμε παραβλέψει μία σημαντική λεπτομέρεια. Το παραπάνω πρόγραμμα επισκέπτεται την ίδια κατάσταση πολλές φορές. Για παράδειγμα, με δύο κινήσεις μπορεί να προκύψει και πάλι η αρχική κατάσταση αν ο άνθρωπος φύγει με την κατσίκα στην πρώτη κίνηση και επιστρέψει μαζί της στη δεύτερη. Αυτό έχει ως αποτέλεσμα να εξερευνούμε πολλές φορές τις ίδιες καταστάσεις — η εύρεση της λύσης καθυστερεί πολύ. Ο μόνος λόγος που το πρόγραμμά μας λειτουργεί σωστά είναι ότι για αυτόν το γρίφο ο χώρος καταστάσεων είναι πολύ μικρός (όλες οι δυνατές καταστάσεις είναι 16 και οι ασφαλείς είναι μόνο 10).

    Για να περιορίσουμε την αναζήτηση έτσι ώστε να επισκεπτόμαστε κάθε κατάσταση ακριβώς μία φορά, πρέπει να προσθέτουμε όλες τις καταστάσεις που έχουν προστεθεί στην ουρά (ανεξαρτήτως αν έχουν αργότερα αφαιρεθεί) σε ένα σύνολο `seen` και να μην ξαναπροσθέτουμε στην ουρά κάποια κατάσταση που ανήκει ήδη στο σύνολο `seen`. Η προσθήκη του συνόλου των καταστάσεων που έχουμε επισκεφθεί είναι βασικό συστατικό του αλγορίθμου BFS.

```
def solve():
    init = state(['man', 'cabbage', 'goat', 'wolf'], [])
    Q = deque([init])
    seen = set([init])
    while Q:
        s = Q.popleft()
        for t in s.accessible():
            if t.success():
                return t
            if t.safe() and t not in seen:
                Q.append(t)
                seen.add(t)
```

15. Δυστυχώς το πρόγραμμά μας τώρα δεν τρέχει. Αποτυγχάνει με το εξής μήνυμα:

```
Traceback (most recent call last):
  File "./2.py", line 58, in <module>
    print(solve())
  File "./2.py", line 48, in solve
    seen = set([init])
TypeError: unhashable type: 'state'
```

    Το μήνυμα αυτό λέει ότι το αντικείμενο `init` που ανήκει στην κλάση `state` δεν μπορεί να τοποθετηθεί στο σύνολο `seen` γιατί ο τύπος (η κλάση) `state` δεν είναι "hashable". Τα σύνολα στην Python υλοποιούνται με πίνακες κατακερματισμού (hash tables), δομές δεδομένων που εξασφαλίζουν σχεδόν σταθερό κόστος αναζήτησης, εισαγωγής και διαγραφής. Για να μπορεί κάποιο στοιχείο να εισαχθεί σε έναν πίνακα κατακερματισμού θα πρέπει να ορίζεται για αυτό η τιμή κάποιας συνάρτησης κατακερματισμού (hash function), που θα επιστρέφει για αυτό το στοιχείο έναν ακέραιο αριθμό. Στην Python, αυτό μπορεί να γίνει ορίζοντας για την κλάση `state` την ειδική μέθοδο `__hash__`, π.χ. ως εξής:

```
def __hash__(self):
    return 42         # this is a very bad idea!
```

    Με τον τρόπο αυτό, η τιμή της συνάρτησης κατακερματισμού για κάθε κατάσταση είναι ίδια και ίση με 42. Αυτό

δουλεύει μεν, αλλά είναι πολύ κακή ιδέα.[1] Η συνάρτηση κατακερματισμού είναι καλό να επιστρέφει διαφορετικές τιμές για διαφορετικές καταστάσεις. Ο ευκολότερος τρόπος να την ορίσουμε σωστά στην Python είναι να χρησιμοποιήσουμε την προκαθορισμένη συνάρτηση `hash` που ορίζεται για όλους τους βασικούς τύπους της Python και να την καλέσουμε για το πεδίο `left` της κατάστασης (ένα `frozenset`). Δεν είναι τυχαίο ότι επιλέξαμε ακριβώς αυτό το πεδίο που συμμετέχει στον έλεγχο ισότητας καταστάσεων.

```
def __hash__(self):
    return hash(self.left)
```

Τώρα, το πρόγραμμα θα έχει το ίδιο αποτέλεσμα με το προηγούμενο, δηλαδή οδηγείται στην τελική κατάσταση και την εκτυπώνει. Όμως, επειδή αποφεύγει να επισκέπτεται ξανά καταστάσεις που έχει ήδη δει, τοποθετεί στην ουρά συνολικά μόνο 9 καταστάσεις, αντί 113 καταστάσεων του προηγούμενου προγράμματος! Σε κάποιο πρόβλημα με μεγαλύτερο χώρο καταστάσεων, η διαφορά θα ήταν τεράστια και το πρώτο πρόγραμμα πρακτικά δε θα τερμάτιζε ποτέ.

16. Το τελευταίο μας θέμα, τώρα που υλοποιήσαμε σωστά το BFS, είναι ότι το πρόγραμμά μας εκτυπώνει την τελική κατάσταση, δε μας λέει όμως με ποια σειρά κινήσεων οδηγηθήκαμε εκεί, δηλαδή δε μας δίνει τη λύση του προβλήματος.

Για πάρουμε τη λύση του προβλήματος πρέπει να γνωρίζουμε για κάθε κατάσταση την οποία επισκεπτόμαστε ποια ήταν η προηγούμενή της, δηλαδή ποια κίνηση μας οδήγησε σε αυτήν. Υπάρχουν (τουλάχιστον) δύο τρόποι να αποθηκεύσουμε αυτή την πληροφορία:

- Να την προσθέσουμε μέσα στις καταστάσεις: κάθε κατάσταση θα περιέχει ως επιπλέον πεδία την προηγούμενη κατάσταση (ή `None`, στην περίπτωση της αρχικής) και πιθανώς την κίνηση που μας οδήγησε εκεί (αν και θα μπορούσαμε εύκολα να την υπολογίζουμε).
- Να μετατρέψουμε το σύνολο `seen` σε ένα λεξικό, το οποίο για κάθε κατάσταση που έχουμε επισκεφθεί θα μας δίνει την προηγούμενή της κατάσταση.

Στην παράδοση αποφασίσαμε να υλοποιήσουμε τη δεύτερη προσέγγιση. (Στη σελίδα του μαθήματος μπορείτε να δείτε το πλήρες πρόγραμμα και για την πρώτη προσέγγιση.) Η συνάρτηση `solve` και η κλήση της γίνονται τώρα:

```
def solve():
    init = state(['man', 'cabbage', 'goat', 'wolf'], [])
    Q = deque([init])
    seen = {init: None}
    def solution(s):
        t = seen[s]
        if t is None: return [s]
        return solution(t) + [s]
    while Q:
        s = Q.popleft()
        for t in s.accessible():
            if t.success():
                seen[t] = s
                return solution(t)
            if t.safe() and t not in seen:
                Q.append(t)
                seen[t] = s


for s in solve():
    print(s)
```

Η εντολή `seen = {init: None}` ορίζει την αρχική τιμή του λεξικού. Περιέχει μόνο το κλειδί `init` (δηλαδή την αρχική κατάσταση), η οποία δεν έχει προηγούμενη (η τιμή που της αντιστοιχεί είναι `None`). Όταν επισκεφτόμαστε μία νέα κατάσταση, προσθέτουμε στο λεξικό την προηγούμενή της με την εντολή `seen[t] = s`. Προσέξτε τον έλεγχο `t not in seen`, που επιστρέφει `True` αν η κατασταση `t` δεν υπάρχει ως κλειδί στο λεξικό `seen`.

Η συνάρτηση `solution(s)` αναλαμβάνει να επιστρέψει τη λίστα των καταστάσεων που οδηγούν στην κατάσταση `s` ξεκινώντας από την αρχική. Χρησιμοποιεί τις τιμές που έχουν αποθηκευθεί στο λεξικό `seen` και είναι αναδρομική. Τη γράφουμε έτσι γιατί γνωρίζουμε ότι η λίστα των κινήσεων είναι σχετικά μικρή — αν ήταν μεγαλύτερη θα έπρεπε να τη γράψουμε αποδοτικότερα.

[1] Αν διαβάσετε πώς λειτουργεί ένας πίνακας κατακερματισμού, θα καταλάβετε ότι με αυτή τη συνάρτηση όλες οι καταστάσεις θα προστίθενται στο ίδιο κελί του πίνακα και άρα αντί για σύνολο θα έχουμε στην πραγματικότητα μία λίστα — το κόστος αναζήτησης, εισαγωγής και διαγραφής θα είναι γραμμικό, αντί σχεδόν σταθερό.

Το πρόγραμμά μας τώρα εκτυπώνει τη σειρά των καταστάσεων από την αρχική μέχρι και την τελική:

```
left: goat & man & cabbage & wolf, right:
left: cabbage & wolf, right: goat & man
left: man & cabbage & wolf, right: goat
left: wolf, right: goat & man & cabbage
left: goat & man & wolf, right: cabbage
left: goat, right: man & cabbage & wolf
left: goat & man, right: cabbage & wolf
left: , right: goat & man & cabbage & wolf
```