Feltételes és ismétlődő elágazás

5.1 Feltételes végrehajtás

Az algoritmusok felépítéséhez elengedhetetlen

5.1.1 `if` statement

A szintaxisa (`if`):

if *Boole kifejezés* **:** *utasítások*

Ha a *Boole kifejezés* értéke `True` , akkor a *Utasítás(ok)* végrehajtásra kerülnek; egyébként nem.

```
>>> if 5 > 2: print("Hurray")
Hurray
>>> if 2 > 5: print("No I will not believe that !")
```

Általában nem egy hanem több utasítást hajtunk végre.

Pl:

```
>>> if 5 > 2:
...     print(7*6)
...     print("Hurray")
...     x = 6
42
Hurray
>>> print(x)
6
```

In [ ]:

```
if 5 > 2:
  print(7*6)
  print("Hurray")
  x = 6
print(x)
```

```
42
Hurray
6
```

Van amikor egy utasítást akkor kell végrehajtani ha `False` értéket kapunk.

```
>>> if x == 6: print("still it is six")
>>> if not (x == 6): print("it is no more six")
```

A fenti példa is lehetne a megoldás. De nem ezt használjuk!

`if-else` erre való:

if *Boole kifejezés* **:** *Utasítás*

else : *Utasítás*

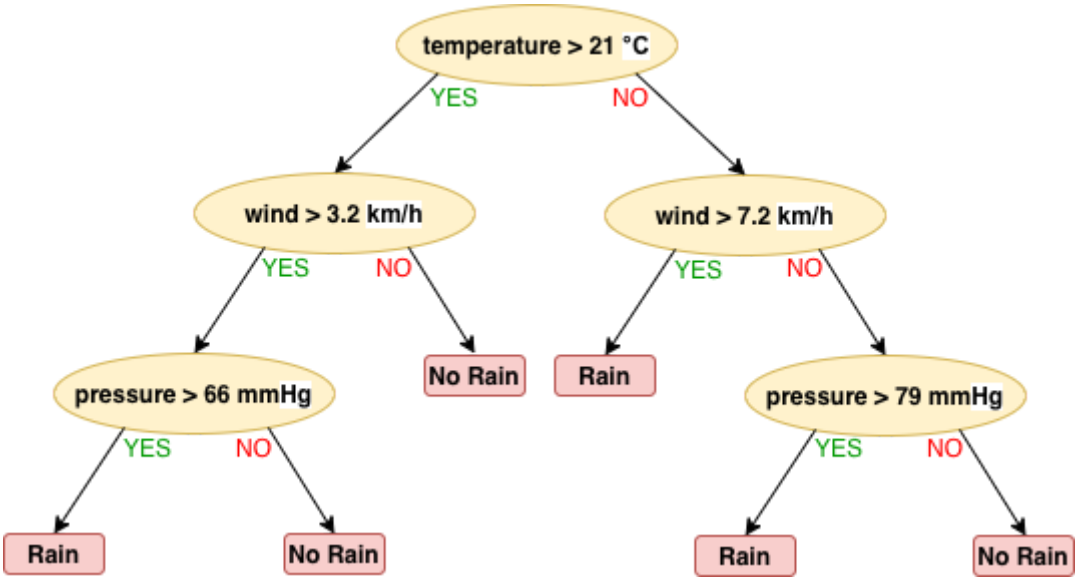
A példánk:

```
>>> if x == 6: print("still it is six")
... else: print("it is no more six")
```

5.1.2 Beágyazott if utasítások

Lehetséges az `if-else` utasítások egymásba ágyazása. A beágyazás mélysége tulajdonképpen korlátlan.

Így hozhatunk lltre bináris döntési fákat.



Döntési fa.

A fenti döntési fa így nézne ki Pythonban:

```
if temperature > 21:
     if wind > 3.2:
          if pressure > 66: rain = True
          else: rain = False
     else: rain = False
else:
     if wind > 7.2: rain = True
     else:
          if pressure > 79: rain = True
          else: rain = False
```

Az `else` ág néha egy másik `if` utasítást tartalmaz. Ehelyett használhatjuk az `elif` utasítást:

if *Boolean expression*$_1$ **:** *Statement*$_1$

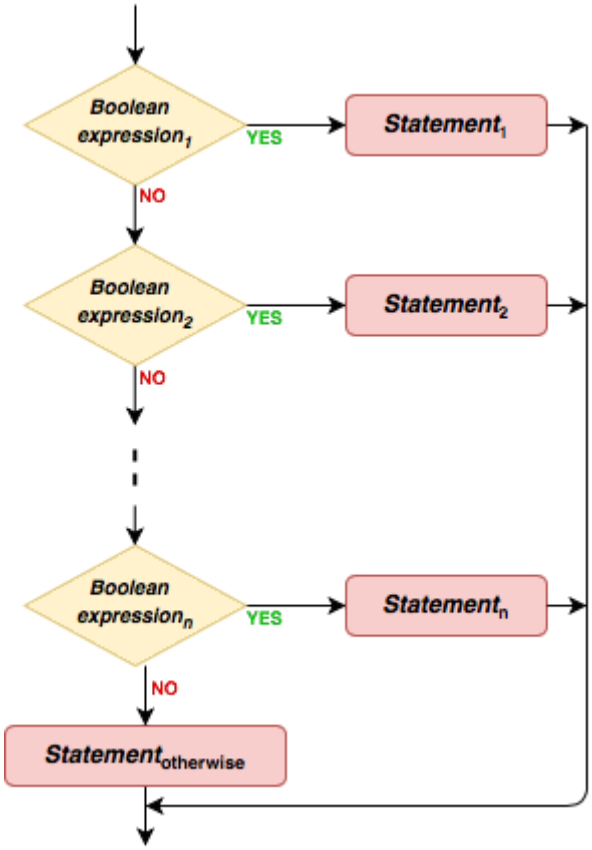
elif $Boolean\ expression_2$ **:** $Statement_2$

⋮

elif $Boolean\ expression_n$ **:** $Statement_n$

else : $Statement_{otherwise}$

A működése itt látható: `if-elif-else` .



- `if-elif-else` .*

Gyakorlat

Rendeljünk az x-hez egy értéke. Az `x` értékétől függően `s` a következő kifejezések halmaza:

$$s = \begin{cases} (x+1)^2, & x < 1 \\ x - 0.5, & 1 \leq x < 10 \\ \sqrt{x + 0.5}, & 10 \leq x < 100 \\ 0, & mindenegyébesetben \end{cases}$$

Pont erre jó az `if-elif-else` használata:

In [1]:
```
#@TODO Assign a value to x
x = 10

if x < 1: s = (x+1)**2
elif x < 10: s = x-0.5
elif x < 100: s = (x+0.5)**0.5
else: s = 0

print("s is: ", s)
```

```
s is:  3.24037034920393
```

5.1.3 Feltételes kifejezés

Az `if` utasítás nem ad vissza értéket. A feltételes kifejezés ad vissza értéket.

Ebben is az `if` és `else` a kulcsszó. De ekkor az `if` nem az utasítás elején van:

$$\boxed{expression_{YES}} \ \texttt{if} \ \boxed{Boolean\ expression} \ \texttt{else} \ \boxed{expression_{NO}}$$

Az egész egy kifejezés és értéket ad vissza. Végrehajtása:

- Először $Boolean\ expression$.
- Ha ez `True` akkor
  $expression_{YES}$ ez értékelődik ki és ez lesz az érték ($expression_{NO}$ ez érintetlen marad).
- Ha ez `False` akkor
  $expression_{NO}$ ez értékelődik ki és ez lesz az érték ($expression_{YES}$.

Nem kötelező de ajánlott ilyenkor a zárójel használata

Pl:

```
>>> x = -34.1905
>>> y = (x if x > 0 else -x)**0.5
>>> print(y)
5.84726431761
```

Az `y` a következő értéket fogja kapni $\sqrt{|x|}$.

5.2 Ismétlődő kifejezések

Fontos programozási módszer *iteration* vagy *loop*.

Python erre két módszert használ: `while` és `for` .

5.2.1 `while` utasítás###

A `while` szintexisa hasonlít az `if` utasításra:

$$\texttt{while} \ \boxed{Boolean\ expression} \ \texttt{:} \ \boxed{Statement}$$

Beágyazható akár magába akár másba:

```
while <condition-1>:
    statement-1
    statement-2
    ...
    while <condition-2>:
       statement-inner-1
       statement-inner-2
       ...
       statement-inner-M
    ... # statements after the second while
    statement-N
```

5.2.2 Példák

Pl. 1: Listában lévő számok átlaga

Így néz ki a feladat matematikai formája:

$$\text{avg}(L) = \frac{1}{N} \sum_{i=1}^{N} L_i,$$

ahol L_i az i-dik szám a listában aminek N eleme van.

A megoldása algoritmusa:

```
Bemenet: N elemű L lista
Output: avg, maga az eredmény

Step 1: sum változó legyen 0
Step 2: kell egy index változó 0 kezdő értékkel
Step 3: While i kisebb mint N,  Steps 4-5
Step 4:     sum = sum + L[i]
Step 5:     i = i + 1
Step 6: avg = sum/N
```

Ugyan ez Pythonban:

In [2]:
```python
# L: A lista
#
L = [10, -4, 4873, -18]
N = len(L)

sum = 0             # Step 1
i = 0               # Step 2

while i < N:        # Step 3
  sum = sum + L[i] # Step 4
  i = i + 1         # Step 5

avg = sum / N       # Step 6
print(avg)
```

```
1215.25
```

Példa 2: Szórás

A szórás így írható fel:

$$\text{std}(L) = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(L_i - \text{avg}(L))^2}.$$

In [3]:
```python
L = [10, 20, 30, 40]
N = len(L)

# szímtsuk ki előbb az átlagot (COPY-PASTE )
sum = 0
i = 0

while i < N:
  sum = sum + L[i]
  i = i + 1

avg = sum / N

# És ezután a szórást
sum = 0
i = 0

while i < N:
  sum = sum + (L[i] - avg)**2
  i = i + 1

std = (sum / N)**0.5

print("A lista átlaga és szórása: ", avg, std)
```

```
A lista átlaga és szórása:  25.0 11.180339887498949
```

Pl 3: Faktoriális A faktoriális jelölése $n!$,:

$$n! = \begin{cases} n \times (n-1) \times \ldots 2 \times 1, & \text{if } n > 0 \\ 1, & \text{if } n = 0. \end{cases}$$

Nézzük az algoritmust:

```
Bemenet: n
Kimenet: n_factorial

Step 1: Ha n 0 vagy negatív akkor n_factorial = 1. Vége
Step 2: Az n_factorial kezdeti értéke 1
Step 3: Kell egy index, i, értéke 1
Step 4: While i <= n, az Steps 5 és 6:
Step 5:     n_factorial = n_factorial * i
Step 6:     i = i + 1
Step 7: Az eredmény
```

ugyan ez Pythonban:

In [4]:
```
# Adjuk meg n értékét:
n = 5

if n <= 0: n_factorial = 1  # Step 1
elif n > 0:                 # Steps 2-6
  n_factorial = 1           # Step 2
  i = 1                     # Step 3
  while i <= n:             # Step 4
    n_factorial *= i        # Step 5
    i += 1                  # Step 6

print("n! értéke ", n_factorial)
```

```
n! értéke  120
```

pl 4: Szinusz számítása Taylor Sorral

A matematikában Taylor-sornak nevezünk hatványfüggvényeknek egy speciális alakú függvénysorát. A Taylor-sorok határértékben gyakran előállítanak bonyolultabb függvényeket (például trigonometrikus vagy hiperbolikus függvényeket), melyek közelítő értékei így pusztán hatványozással kiszámíthatók. A függvények Taylor-sor alakjában történő felírását a függvények hatványsorba fejtésének nevezzük.

Számítsuk ki $\sin(x)$, értékét egy adott x értékre. (a számítógép egyébként pont így számolja a szinuszt csak beépített cél-processzor segítségével)

A $\sin(x)$ Taylor sora 0 körül így néz ki:

$$\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} \cdots,$$

ami így is felírható:

$$\sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)!} x^{2k+1}.$$

Most valósítsuk meg ezt. A külső iteráció az összeg elemein fut végig amíg az érték kellően kicsi lesz. ($|term| < \epsilon$, ahol ϵ a kis érték (pontosság), pl. 10^{-12}).

```
epsilon = 1.0E-12
x = 3.14159265359/4.0   # i.e. pi/4 (45 fok radian)
result = 0.0
k = 0
term = 2*epsilon    #
while abs(term) > epsilon:
    # Calculate the denominator - i.e. (2k+1)!
    factorial = i = 1
    while i <= 2*k+1:
      factorial *= i
      i += 1

    # Now calculate the term
    term = (((-1)**k) / factorial) * (x**(2*k+1))

    result += term
    k += 1
```

Ez egy jól olvasható kód, de nagyon nem hatékony.

- `2*k+1` többször is kiszámolásra kerül. De kell egyáltalán a `2*k+1` ?
- Próbáljunk egyszerűbb kódit írni.

Nézzük meg két egymást követő elem hányadosát $(k-n)$ és $(k-n+1)$:

In [5]:
```
epsilon = 1.0E-12
x = 3.14159265359/4.0   # i.e. pi/4 (45 degrees in radians)
x_square = x*x
term = x
result = term
d = 1    #  2*n+1
while abs(term) > epsilon:
    term *= -x_square/((d+1)*(d+2))
    result += term
    d += 2

print("sin(x) [Ours] is: ", result)

# Nézzük hozzá a beépített függvényt:
from math import *
print("sin(x) [CPU ] is: ", sin(x))
```

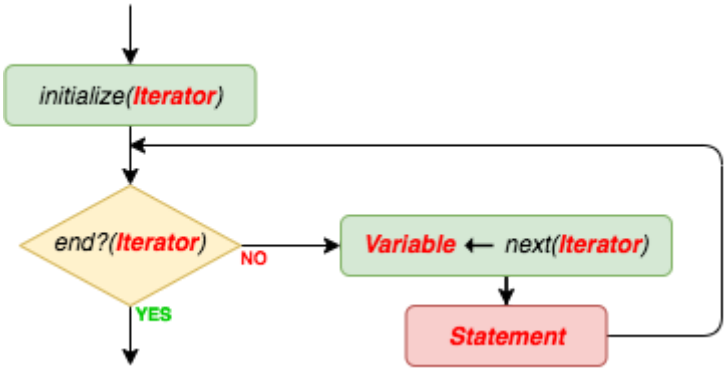
```
sin(x) [Ours] is:  0.707106781186584
sin(x) [CPU ] is:  0.7071067811865841
```

5.2.3 `for` utasítás###

A `for` szintaxisa:

for *Variable* **in** *Iterator* **:** *Statement*

Az *iterator* egy objektum ami megszámlálható darabszámú értéket ad.



A `for` utasítás végrehajtása.

Milyen iterátorokat használhatunk?

1. Minden tároló típusú adatot (string, list, dictionari, set).
2. A beépített `range` függvényt:

`range(` *start*, *stop*, *step* `)`

Parameter	Opt./Req.	Alapérték	Magyarázat
start	Optional	0	Egész szám, kezdő érték.
stop	Required		Egész, a végső érték (a végső érték nem lesz része)
step	Optional	1	Egész, lépésköz

5.2.4 Példák for utasításhoz

Példa 1: Szavak

Osszuk ketté a szavakat aszerint, hogy mással vagy magánhangzóval kezdődnek:

In [6]:
```
mixed = ["lorem","ipsum","dolor","sit","amet,","consectetur","adi
maganh = []
massalh = []
for word in mixed:
    if word[0] in ['a','e','i','o','u']: maganh += [word]
    else: massalh += [word]

print("Mássalhangzóval kezdődik:", massalh)
print("Magánhangzóval kezdődik:", maganh)
```

```
Mássalhangzóval kezdődik: ['lorem', 'dolor', 'sit', 'consectetur',
'sed', 'do', 'tempor', 'labore', 'dolore', 'magna']
Magánhangzóval kezdődik: ['ipsum', 'amet,', 'adipiscing', 'elit',
'eiusmod', 'incididunt', 'ut', 'et', 'aliqua']
```

Example 2: Futáshossz kódolás (Run-length Encoding)

A run-length encoding (RLE, futáshossz-kódolás) egy nagyon egyszerű tömörítési eljárás, melyben az adatban található, hosszasan ismétlődő karaktereket egyetlen értékként és számként tárolják, az eredeti teljes karaktersorozat helyett. Ez leginkább sok ilyen hosszú karaktersorozattal rendelkező adatra hasznos: például egyszerű grafikus képek, mint ikonok, vonalrajzok és animációk.

A lényeg így tárolható: `[character, repetition count]`.

pl az alábbi karaktersorozat `"aaaaaaxxxxmyyyaaaasssssssssttttuivvvv"` ilyen lesz:
`[['a',6],['x',4],'m',['y',3],['a',4],['s',9],['t',3],'u','i',['v',4]]`.

In [2]:
```
text = "aaaaaaxxxxmyyyaaaasssssssssttttuivvvv"
code_list = []
last_character = text[0]
count = 1

# menjunk végig az összes karaketeren, kivéve az elsőt
for curr_character in text[1:]:
    # Ha curr_character egyenlő last_character, akkor találtunk e
    if last_character == curr_character:
        count += 1
    else:
        # Befekeztük az azonos karaketerek keresését
        # uj last_character
        code_list += [last_character if count==1
                      else [last_character, count]]
        count = 1
        last_character = curr_character

# a last_character kezelése:
code_list += [last_character if count==1 else [last_character,cou

print(code_list)
```

```
[['a', 6], ['x', 4], 'm', ['y', 3], ['a', 4], ['s', 9], ['t', 3],
'u', 'i', ['v', 4]]
```

Feladat

írjuk át úgy a kódot hogy az eltávolítsa az ismétlődő karaketereket. vagyis, `"aaaabbbcdd"` ez lesz: `"abcd"`.

pl 3: Permutáció

A permutáció segítségével keverhetünk össze adatsorokat.

In [8]:
```
word_list = ["he","came","home","late","yesterday"]
permutation = [4,3,2,0,1]
length = len(permutation)
new_list = [None]*length

for i in range(length):
    new_list[i] = word_list[permutation[i]]

print(new_list)
```

```
['yesterday', 'late', 'home', 'he', 'came']
```

pl 4: Lista vágása

Ketté osztunk egy listát az alapján hogy a benne lévő számok az első számnál nagyobb vagy kisebb.

In [9]:
```
list_to_split = [42, 59, 53, 84, 43, 8, 75, 34, 40, 89, 29, 15, 5
list_smaller = []
list_not_smaller = []

for x in list_to_split[1:]:
    if x < list_to_split[0]: list_smaller += [x]
    else: list_not_smaller += [x]

print("Az első elem értéke.        :",list_to_split[0])
print("Kisebb:    :", list_smaller)
print("Nagyobb :", list_not_smaller)
```

```
Az első elem értéke.        : 42
Kisebb:    : [8, 34, 40, 29, 15, 6, 32, 4, 24]
Nagyobb : [59, 53, 84, 43, 75, 89, 51, 90, 58, 77]
```

Example 5: Vektorok szorzása

A vektorok szorzása: $\mathbf{u}$ and $\mathbf{v}$:

$$\mathbf{u} \cdot \mathbf{v} = \sum_{i=1}^{n} u_i v_i,$$

In [11]:
```
# A vektorokat listaként definiáljuk
u = [1, 2, 4, 10]
v = [10, 4, 2, 1]
n = len(u)
dot_prod = 0

if n != len(v): print("Nem egyezik a vektorok mérete")
else:
  for i in range(n):
    dot_prod += u[i] * v[i]
  print("u . v eredménye: ", dot_prod)
```

```
u . v eredménye:  36
```

Example 6: Vektorok által bezért szög

A vektorok által bezárt szög $\mathbf{u}$ and $\mathbf{v}$ szorosan kapcsolódik a szorzatukhoz:

$$\mathbf{u} \cdot \mathbf{v} = ||\mathbf{u}|| \, ||\mathbf{v}|| \, \cos(\theta),$$

ahol θ a két vektor által bezárt szög, és $||\cdot||$ jelöli a vektor normáját:

$$||\mathbf{u}|| = \sqrt{\mathbf{u} \cdot \mathbf{u}} = \sqrt{\sum_{i=1}^{n} u_i^2}.$$

A szög ezekből így számítható ki:

$$\theta = \arccos\left(\frac{\mathbf{u} \cdot \mathbf{v}}{||\mathbf{u}|| \, ||\mathbf{v}||}\right).$$

In [ ]:

```python
from math import sqrt, acos

# Define two vectors that have 90deg (pi/2) between them
u = [1, 0, 0]
v = [0, 1, 0]
n = len(u)

if n != len(v):
  print("Sizes don't match!")

else:
  # Calculate dot product & norms
  dot_prod = u_norm_sum = v_norm_sum = 0
  for i in range(n):
    dot_prod   += u[i] * v[i]
    u_norm_sum += u[i] ** 2
    v_norm_sum += v[i] ** 2

  u_norm = sqrt(u_norm_sum)
  v_norm = sqrt(v_norm_sum)

  theta = acos(dot_prod / u_norm * v_norm)

  print("angle between u and v is: ", theta)
```

```
angle between u and v is:  1.5707963267948966
```

példa 7: Matrix szorzás

Két mátrix A és B akkor szorozható össze ha az A oszlop száma megegyezik B sorainka a számával.

$$(A \cdot B)_{ij} = \sum_{k=0}^{m-1} A_{ik} \cdot B_{kj}.$$

A mátrixokat általában listák listájaként definiáljuk. Pl:

$$\begin{pmatrix} -2 & 3 & 5 & -1 \\ 0 & 3 & 10 & -7 \\ 11 & 0 & 0 & -8 \end{pmatrix}$$

:

```
A = [[-2,3,5,-1], [0,3,10,-7], [11,0,0,-8]]
```

Az indexelés is hasonló: A_{ij} Pythonban `A[i][j]` lesz.

In [12]:

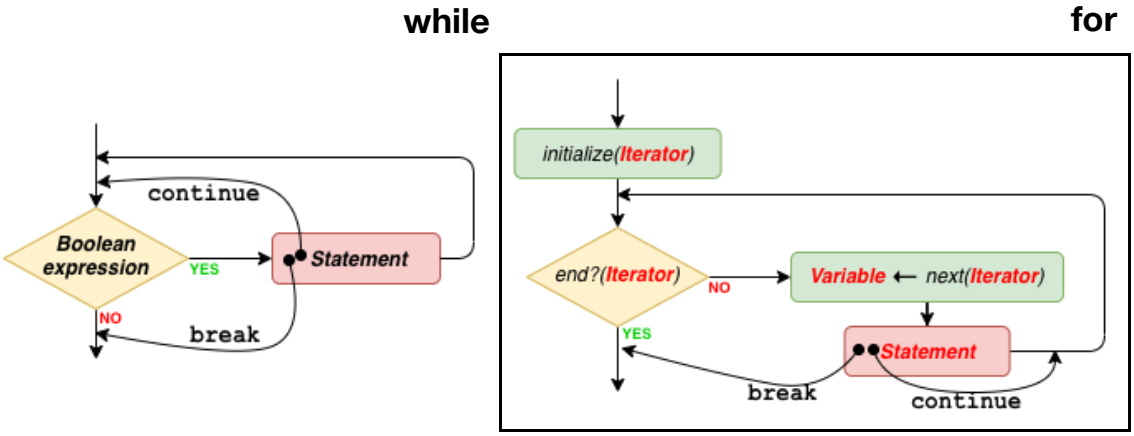
```python
A = [[-2,3,5,-1], [0,3,10,-7], [11,0,0,-8]]
B = [[2,1], [-1,1], [0,4], [8,0]]

# C = A*B

# Csináljuk meg az eredmény mátrixot csupa 0-val
C = []
for i in range(len(A)):
  C += [[0] * len(B[0])]  #

for i in range(len(C)):
    for j in range(len(C[0])):
        for k in range(len(B)):
            C[i][j] += A[i][k]*B[k][j]

print(C)
```

```
[[-15, 21], [-59, 43], [-42, 11]]
```

5.2.5 `continue` és `break` utasítások

A `while` és `for` utasításokba lehet beavatkozni.

A `break` utasítás azonnal megszakítja a ciklust.



A `continue` működése.

5.2.6 Set és list

Matematikában gyakran halmazokkal írjuk le a változók értékét

Pl:

$$\{x^3 \mid x \in \{0, 1, \ldots, 7\}\}$$

ami ezt adja:

$$\{0, 1, 8, 27, 64, 125, 216, 343\}$$

vagy így is írható:

$$\left\{x^3 \mid x \in \{0, 1, \ldots, 7\} \land (x \bmod 2) = 1\right\}$$

ami

$$\{1, 27, 125, 343\}$$

Ugyan ez kódban:

```
>>> [x**3 for x in range(8)]
[0, 1, 8, 27, 64, 125, 216, 343]
>>> {x**3 for x in range(8)}
{0, 1, 64, 8, 343, 216, 27, 125}
```

Ha `x` páratlan:

```
>>> [x**3 for x in range(8) if x%2 == 1]
[1, 27, 125, 343]
>>> {x**3 for x in range(8) if x%2 == 1}
{1, 27, 125, 343}
```

In [13]:

```
print( [[0 for i in range(3)] for j in range(4)] )
print( [[1 if i==j else 0 for i in range(3)] for j in range(3)] )
print( [[(i,j) for i in range(3)] for j in range(4)] )
print( [[(i+j)%2 for i in range(3)] for j in range(4)] )
print( [[i+3*j for i in range(3)] for j in range(4)] )
```

```
[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]
[[(0, 0), (1, 0), (2, 0)], [(0, 1), (1, 1), (2, 1)], [(0, 2), (1,
2), (2, 2)], [(0, 3), (1, 3), (2, 3)]]
[[0, 1, 0], [1, 0, 1], [0, 1, 0], [1, 0, 1]]
[[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 10, 11]]
```