J.Courtin *PC* LVH Caen*

INFORMATIQUE

Les listes par compréhension

Compréhension de listes

<u>L'objectif de la compréhension de liste est double :</u>

- C'est un outil qui permet de créer une liste en une seule ligne de commande à partir de l'idée abstraite de son contenu.

- C'est une syntaxe qui appelle un programme compilé en C, ce qui permet de créer des listes de grande taille en un temps toujours optimisé.

La compréhension de liste est à la fois très simple d'usage et très efficace.

Ex : Liste des carrés des nombres impairs mais non divisibles par 5 !

```
>>> [ (2*k+1)**2  for k in range(100)  if  ((2*k+1)**2)%5 != 0 ]

[1, 9, 49, 81, 121, 169, 289, 361, 441, 529, 729, 841, 961, 1089, 1369, 1521, 1681, 1849, 2209, 2401, 2601, 2809, 3249, 3481, 3721,
3969, 4489, 4761, 5041, 5329, 5929, 6241, 6561, 6889, 7569, 7921, 8281, 8649, 9409, 9801, 10201, 10609, 11449, 11881, 12321,
12769, 13689, 14161, 14641, 15129, 16129, 16641, 17161, 17689, 18769, 19321, 19881, 20449, 21609, 22201, 22801, 23409, 24649,
25281, 25921, 26569, 27889, 28561, 29241, 29929, 31329, 32041, 32761, 33489, 34969, 35721, 36481, 37249, 38809, 39601]
```

On peut de même créer des ensembles ou des dictionnaires

```
>>> {k**3 for k in range(10)}
set([0, 1, 8, 64, 512, 343, 216, 729, 27, 125])
```

```
>>> {k : 2*k+1 for k in range(10)}
{0: 1, 1: 3, 2: 5, 3: 7, 4: 9, 5: 11, 6: 13, 7: 15, 8: 17, 9: 19}
```

Application : inversion {clef : valeur} d'un dictionnaire

```
>>> monDico = {123 : "riri",   231 : "fifi",    421 : "loulou"}
>>> monDico
{123: 'riri',   421: 'loulou',    231: 'fifi'}
```

```
>>> { monDico[k] : k for k in monDico}
{'riri': 123, 'fifi': 231, 'loulou': 421}
```

Attention : bijection

Création d'une table de mutiplication :

Compréhension de listes imbriquées

```
>>> Table
[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
 [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
 [0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30],
 [0, 4, 8, 12, 16, 20, 24, 28, 32, 36, 40],
 [0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50],
 [0, 6, 12, 18, 24, 30, 36, 42, 48, 54, 60],
 [0, 7, 14, 21, 28, 35, 42, 49, 56, 63, 70],
 [0, 8, 16, 24, 32, 40, 48, 56, 64, 72, 80],
 [0, 9, 18, 27, 36, 45, 54, 63, 72, 81, 90],
 [0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]]
```

```
>>> Table = [ [i*j for j in range(11) ]  for i in range(11) ]
```

```
>>> Table[3][7] ; Table[5][7] ; Table[7][5]
21
35
35
```

En réalité les «boucle for» simples sont également précompilées d'où le faible écart.

Ainsi, en fonction de la structure du code, cet écart sera plus ou moins prononcé ....

```
import time
N = 10000

#Version reconstruction de liste
T1 = time.clock()

L1 = []
for i in range(N):
    for j in range(N):
        if (i!=j):
            L1 + = [i*j]
T2 = time.clock()
print(T2-T1)

#Version comprehension
T1=time.clock()
L2=[j*i for j in range(N) for i in range(N) if i!=j]
T2=time.clock()
print(T2-T1)
```

1 ligne contre 5 lignes et ~ 3 fois plus rapide

	Reconstruction	Compréhension
n=10	0,000109	0,000047
n=100	0,0093	0,003
n=1000	0,99	0,34
n=10000	98,209	32,963
	$O(N^2)$	$O(N^2)$

÷ 3

La Compréhension de liste est toujours plus rapide

Attention : dans certains cas, une syntaxe trop concise devient incompréhensible.....

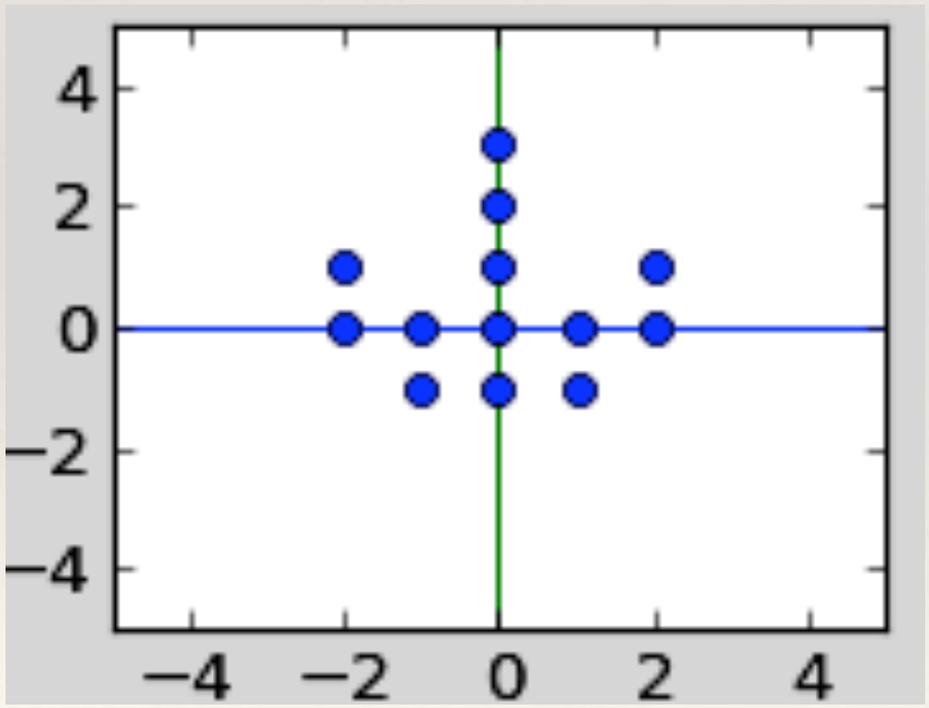
Application : Transformations géométriques simples

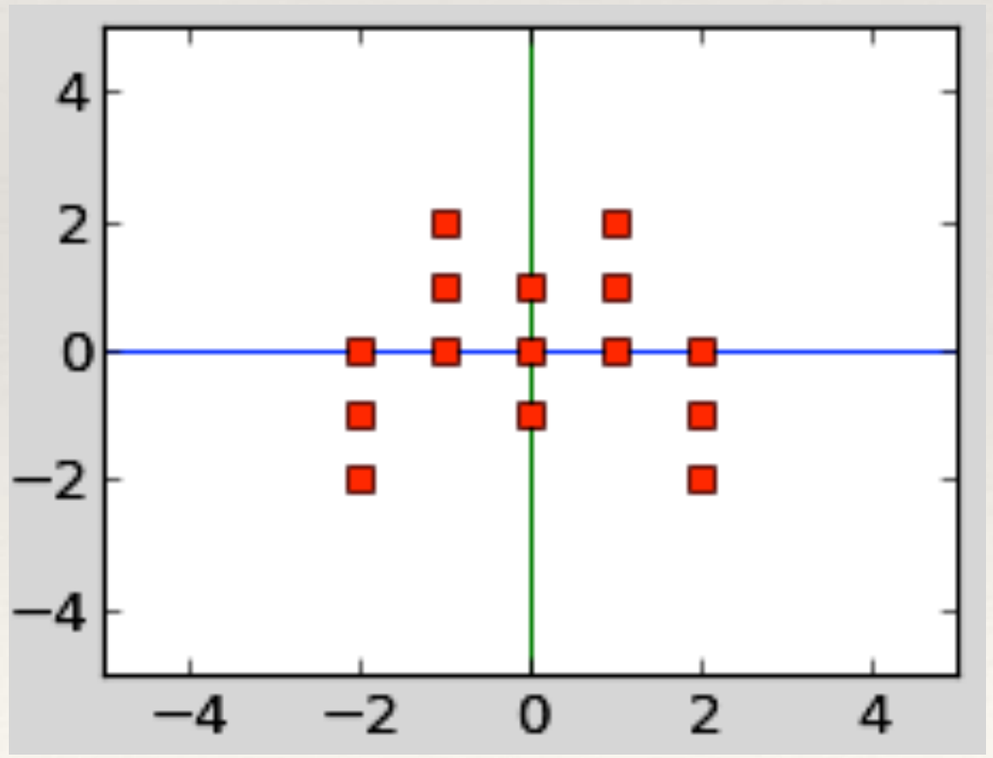
!!! INVADERS !!!

You gotta save the earth !

Pour réaliser un jeu de War Game, on se donne des envahisseurs et un vaisseau spatial par un tuple contenant les 2 listes des pixels (x, y)

```
spaceship0 = ( [-2,-2,-1,-1,0,0,0,0,0,1,1,2,2], [0,1,-1,0,-1,0,1,2,3,-1,0,0,1] )
```






```
invader0 = (  [-2, -2, -2, -1, -1, -1, 0, 0, 0, 1, 1, 1, 2, 2, 2],[-2, -1, 0, 0, 1, 2, -1, 0, 1, 0, 1, 2, -2, -1, 0] )
```

Translation d'un objet :

```
#Deplace l'objet en x,y
def move_Object(object, x,y):
    return  ( [k+x for k in object[0]],  [k+y for k in object[1]])
```

On met a profit les compréhensions de liste

La fonction mange un objet de type tuple de listes :

([.. abscisses ..], [.. ordonnées ..])

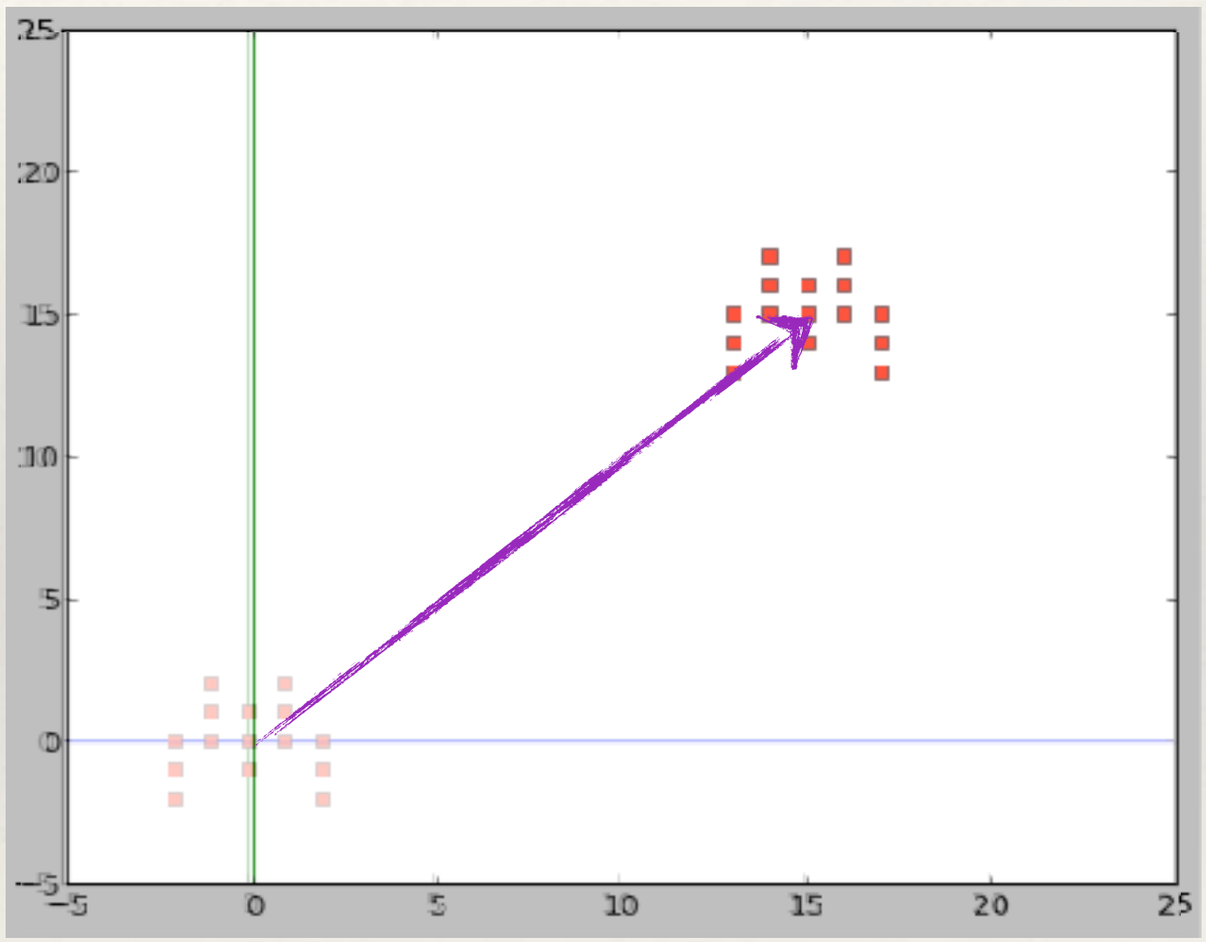
et retourne un objet de type exactement identique

mais avec des coordonnées translatées

Tracé à l'aide matplotlib :

```
#Trace d'un objet
def   plot_Object(object, Ob_style = 'bo'):
        plt.plot( object[0],  object[1], Ob_style)
```

plot_Object(move_Object(invader0, 15, 15), 'rs')



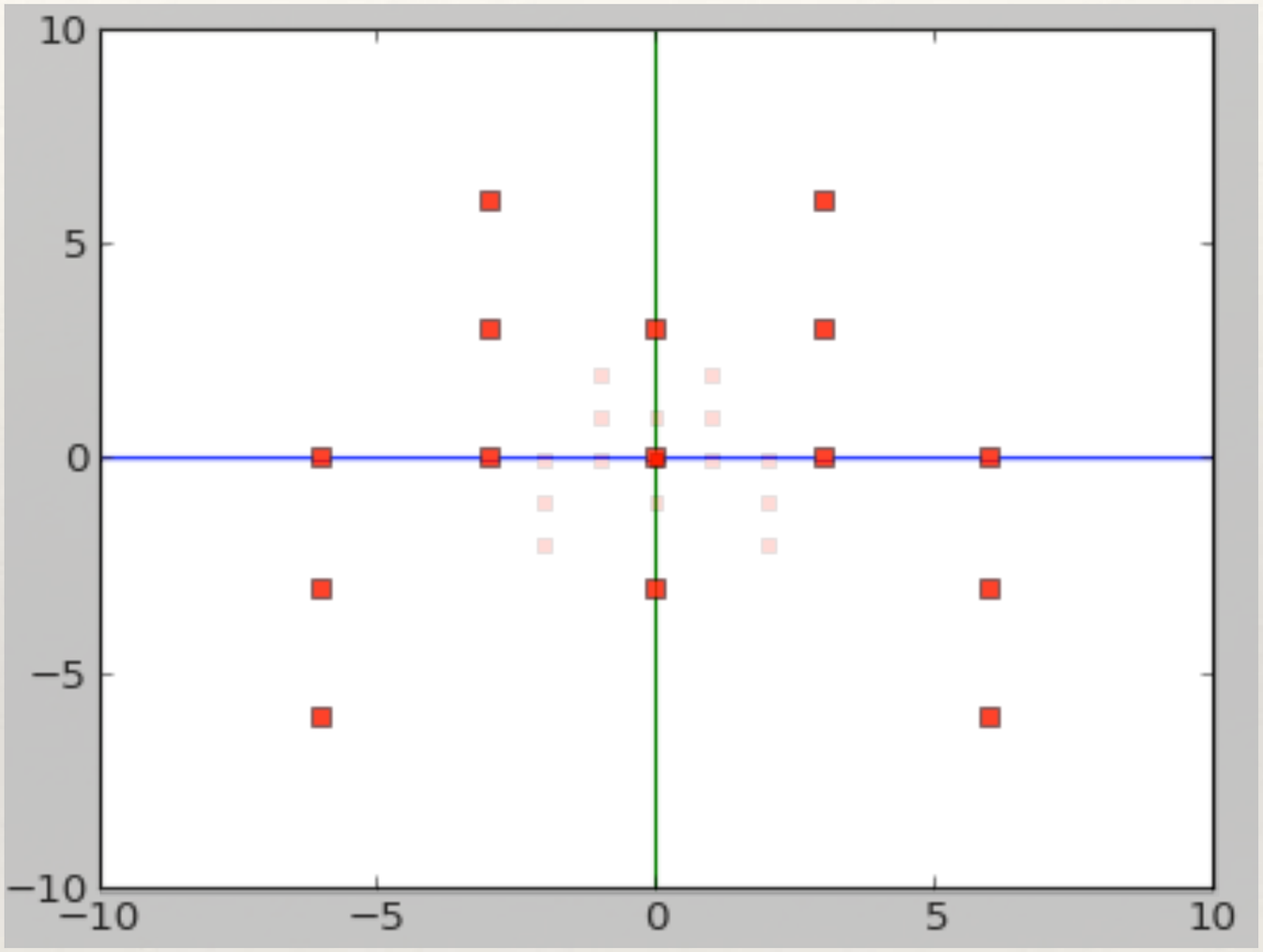
On utilise à nouveau : - la liste de abscisses
- la liste des ordonnées

(quel que soit l'objet)

Homothétie d'un objet :

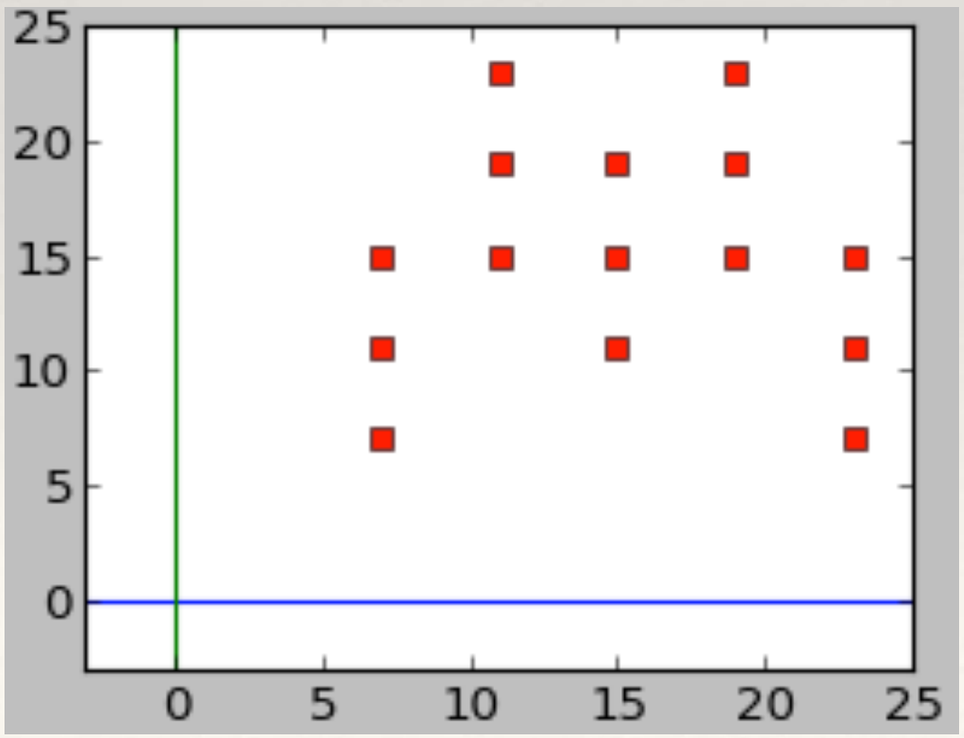
```
def super_size(object, alpha=1.):
    return ( [alpha*k for k in object[0]], [alpha*k for k in object[1]] )
```

plot_Object(super_size(invader0, 3), 'rs')



A nouveau, la structure de l'objet ne doit pas être modifiée pour pouvoir le tracer.

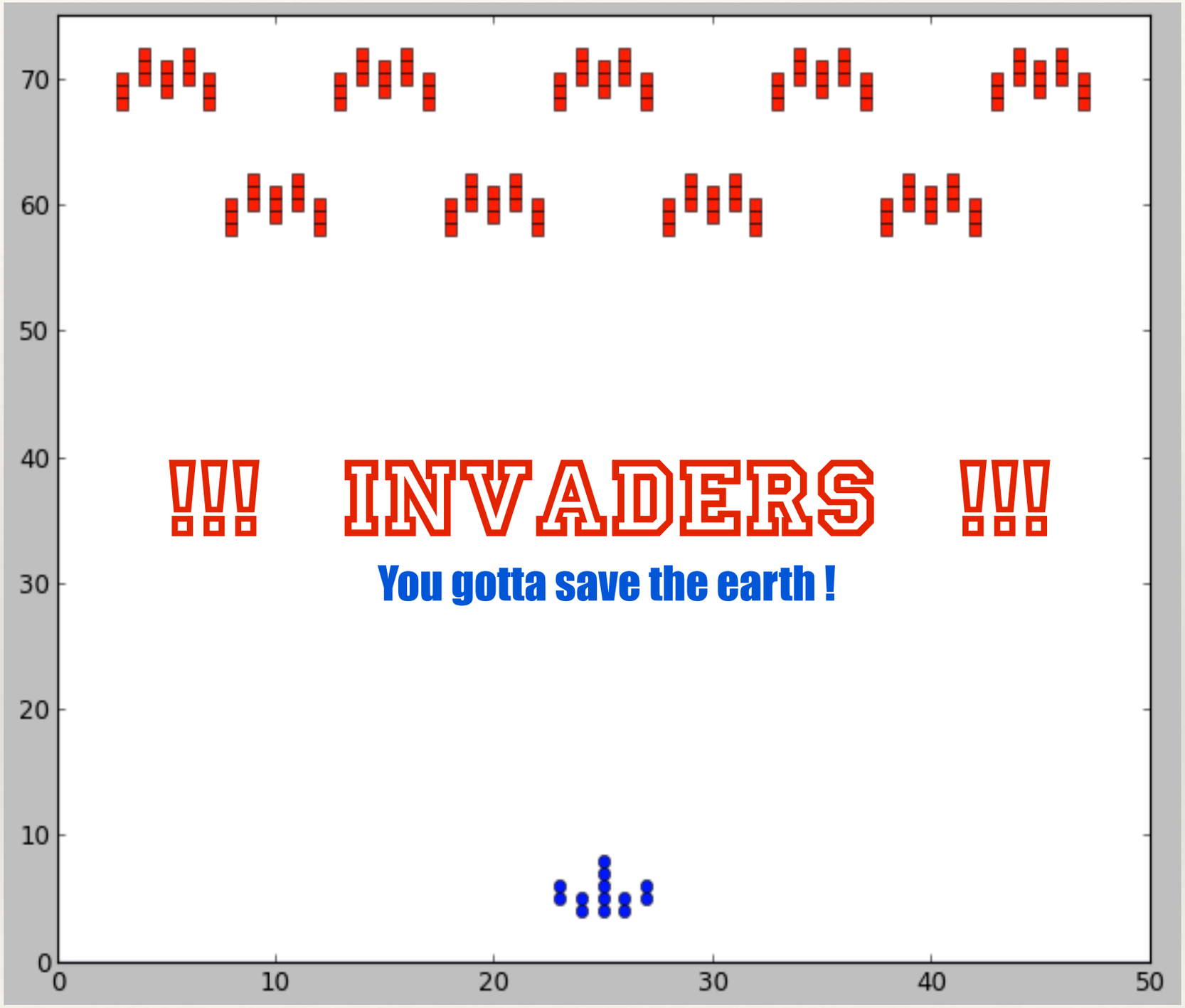
Seules les valeurs des coordonnées sont modifiées.



On peut ensuite combiner ces deux opérations :

plot_Object(move_Object(super_size(invader0, 4) , 15,15), 'rs')

Question : - Peut-on interchanger l'ordre des opérations **move_Object** et **super_size** ?
- Pour quelle raison ?



Exercice : Réaliser l'affichage d'écran ci-dessus à l'aide des fonctions introduites dans le cours. (optionnel et sans le titre ...)

Application : Tracé efficace d'une courbe !

Intervalle de largeur 17 : -5 —> 12 : un point tous les 0.1 => 170 points

Interpolation linéaire sur [-5, 12 [

```
x = [-5. + (12.+5.)*k/170   for k in range(170)]  #création des abscisses
y = [k**3-8*k**2-15*k+1 for k in x]               #calcul des ordonnées associées
                                                  #par compréhension de liste

from matplotlib import pyplot as plt
plt.plot(x,y)                                     #Tracé !
plt.show()
```

Compréhension de liste

$$y = x^3 - 8x^2 - 15x + 1$$