Programmation impérative en Python

Cours 8. Ensembles, dictionnaires et matrices

Olivier Baldellon

Courriel : `prénom.nom@univ-cotedazur.fr`

Page professionnelle : `http://deptinfo.unice.fr/~obaldellon/`

Licence 1 — Faculté des sciences de Nice — Université Côte d'Azur

- Il y aura pour les volontaires un projet Tk noté.
- La note sera un bonus qui comptera pour le contrôle continu.
- Les règles et les objectifs :
  - Vous devez utiliser Python et Tk (et rien d'autre).
  - Votre code doit être lisible, propre et commenté.
  - Votre code doit être générique et paramétrable.
  - Votre code doit fonctionner sans problème sur les machines Linux du Petit Valrose.
  - Date limite : date de l'examen final ?

Test Moodle

- Il y aura un test sur Moodle la dernière semaine (semaine 9).
- Obligation de venir dans le bon groupe
- Le programme : tous les chapitres de 1 à 8 (donc sauf le 9)

Sommaire

- Nous avons eu l'occasion de voir plusieurs types de données.

Des types simples

```
>>> type(-29)
<class 'int'>
>>> type(42.23)
<class 'float'>
>>> type(True)
<class 'bool'>
```

Des types des séquences

```
>>> type((1,2,3))
<class 'tuple'>
>>> type([11,1.2])
<class 'list'>
>>> type("Salut à toi")
<class 'str'>
```

- Nous allons voir deux autres types

```
>>> type({1,2,3}) # Les ensembles
<class 'set'>
>>> type({'prix':1.2 , 'nom':'banane'}) # Les dictionnaires
<class 'dict'>
```

- Un ensemble en Python est une collection finie d'objets
  - Une collection sans répétition et sans ordre
- Un ensemble n'est pas une séquence !
  - On ne peut pas accéder aux éléments via des indices. ~~`E[i]`~~
- Ils sont notés avec des accolades comme en mathématiques.

SHELL
```
>>> { 1, 2, 3, 1, 2 } # Ni répétition
{1, 2, 3}
>>> { 1, 2, 3 } == { 3, 1, 2 } # ni ordre
True
>>> {False, 'bleu', 2, 'bleu'} == {2, False, 'bleu', 2, 2}
True
```

- L'ensemble vide est noté : `set()` (et non pas `{}` qui est un dictionnaire)

SHELL
```
>>> type({})
<class 'dict'>
>>> type(set())
<class 'set'>
>>> set()
set()
```

Accès aux éléments d'un ensemble

- La notation ~~`E[i]`~~ n'a pas de sens
  - Les éléments ne sont pas ordonnés : ils n'ont donc pas d'indice.

SHELL

```
>>> E = { 22, 31.2, 'Salut', True }
>>> E[2]
Traceback (most recent call last):
 File "<console>", line 1, in <module>
TypeError: 'set' object is not subscriptable
```

- On peut parcourir un ensemble avec une boucle for :
  - L'ordre n'est pas respecté (car il n'y a pas d'ordre !)

SHELL

```
>>> for x in E:
...     print(x)
True
Salut
22
31.2
```

- L'opérateur `in` permet de savoir si un élément appartient à un ensemble

SHELL
```
>>> A = {22, 'Salut'} ; B = { 22, 31.2, 'Salut', True }
>>> print( 31.2 in A , 31.2 in B)
False True
```

- En mathématiques, A est inclus dans B si tout élément de A appartient à B
  - $A \subseteq B \quad \equiv \quad \forall x \in A, x \in B$

- Traduisons cela en Python avec une boucle `for`

SCRIPT
```
def inclusion(A,B):
    for x in A:
        if not(x in B):
            return False
    return True
```

SHELL
```
>>> inclusion(A,B)
True
>>> inclusion(B,A)
False
>>> inclusion(A,A)
True
```

- Ou directement avec l'opérateur `<=` (si on veut l'inclusion stricte : `<`) :

SHELL
```
>>> print( A<=B , B<=A , A<=A, A<A , set()<A)
True False True False True
```

Nombre d’éléments d’un ensemble

- En mathématiques, le cardinal est le nombre d’éléments d’un ensemble.

SCRIPT
```python
def cardinal(E):
    c = 0
    for x in E:
        c=c+1
    return c
```

SHELL
```
>>> cardinal({2,2,1,1,3,3,2,1,2,3})
3
>>> {2,2,1,1,3,3,2,1,2,3} # ne contient bien que 3 éléments
{1, 2, 3}
>>> cardinal(set())
0
```

- Comme d’habitude, on ~~s’embête pour rien~~ : fonction `len`
    - C’est important pédagogiquement de savoir réécrire les fonctions de base.

SHELL
```
>>> len({2,2,1,1,3,3,2,1,2,3})
3
>>> len(set())
0
```

- On peut construire un ensemble en donnant directement ses valeurs

SHELL
```
>>> E = {1 , 2 , 3 , 4}
>>> E
{1, 2, 3, 4}
```

- De même que les listes, on peut les construire par compréhension.

SHELL
```
>>> L = [ x%10 for x in range(100) if x%6==0 ]
>>> L # L est une liste
[0, 6, 2, 8, 4, 0, 6, 2, 8, 4, 0, 6, 2, 8, 4, 0, 6]
>>> E = { x%10 for x in range(100) if x%6==0 }
>>> E # E est un ensemble
{0, 2, 4, 6, 8}
```

- On peut utiliser la fonction de conversion `set` (voir cours 5 page 18)

SHELL
```
>>> set('abc')
{'b', 'c', 'a'}
>>> set(['a','b','c'])
{'b', 'c', 'a'}
>>> set(('a','b','c'))
{'b', 'c', 'a'}
```

- Les ensembles sont mutables : Cela signifie que l'on peut les modifier.
- On peut ajouter un élément avec la méthode `add`

SHELL
```
>>> E            # E contient 3 éléments
{1, 2, 3}
>>> E.add(12)
>>> E            # E contient maintenant 4 éléments
{1, 2, 3, 12}
>>> E.add(1)
>>> E            # E contient toujours 4 éléments
{1, 2, 3, 12}
```

- On peut supprimer un élément avec la méthode `remove`

SHELL
```
>>> E = {'Salut',False,(1,2,3),'Ha ha ha'}
>>> E.remove('HA ha ha')
Traceback (most recent call last):
 File "<console>", line 1, in <module>
KeyError: 'HA ha ha'
>>> E
{False, 'Salut', (1, 2, 3), 'Ha ha ha'}
>>> E.remove('Ha ha ha')
```

- On veut écrire l'union de deux ensembles.

SCRIPT
```
def union(A,B):
    E = set() # ensemble vide
    for x in A:
        E.add(x)
    for x in B:
        E.add(x)
    return E
```

SHELL
```
>>> A={1,2,3,4}
>>> B={2,4,6,8}
>>> union(A,B)
{1, 2, 3, 4, 6, 8}
>>> union(A,A)
{1, 2, 3, 4}
```

- Comme toujours, la fonction existe déjà sous Python !
  - L'union $A \cup B = \{x : x \in A \text{ ou } x \in B\}$ se note `A | B`
  - L'intersection $A \cap B = \{x : x \in A \text{ et } x \in B\}$ se note `A & B`
  - La différence $A \setminus B = \{x : x \in A \text{ et } x \notin B\}$ se note `A - B`

SHELL
```
>>> A={1,2,3,4}
>>> B={2,4,6,8}
>>> A|B
{1, 2, 3, 4, 6, 8}
```

SHELL
```
>>> A&B
{2, 4}
>>> A-B
{1, 3}
```

- Exercice : écrire l'intersection et la différence sans utiliser d'opérateurs.

Que puis-je mettre dans un ensemble ?

- Bonne question : essayons !

SHELL
```
>>> { '123' , 'abc' }          # str   : immuable
{'abc', '123'}
>>> { (1,2,3) , ('a','b','c') } # tuple : immuable
{(1, 2, 3), ('a', 'b', 'c')}
>>> { [1,2,3] , ['a','b','c'] } # list  : mutable
Traceback (most recent call last):
 File "<console>", line 1, in <module>
TypeError: unhashable type: 'list'
>>> { {1,2,3} , {'a','b','c'} } # set : mutable
Traceback (most recent call last):
 File "<console>", line 1, in <module>
TypeError: unhashable type: 'set'
```

- Un ensemble est mutable, mais ses éléments doivent être immuable.
  - On peut faire des ensembles de tuple (ensembles de points du plan)
  - On ne peut pas faire des ensembles de listes ou d’ensemble.
- Que signifie *unhashable type* du message d’erreur ?
  - En interne, les ensembles utilisent des fonctions de hachage.

Sommaire

- Qu'est-ce qu'une empreinte digitale ?
  - C'est un marqueur biologique en théorie unique.
  - Facile à stocker, « facile » à comparer, facile à obtenir

- Existe-il un équivalent numérique ?
  - Une fonction de hashage permet de construire une empreinte numérique.
- Exemple : la fonction `md5`
  - Elle calcule une empreinte (*digest*) de 128 bits.
  - On représente l'empreinte le plus souvent par son écriture hexadécimale

SHELL

```
>>> from hashlib import md5
>>> def h(x):
...     return md5(repr(x).encode()).hexdigest()
>>> h(3)
'eccbc87e4b5ce2fe28308fd9f2a7baf3'
>>> h([1,2,3])
'49a5a960c5714c2e29dd1a7e7b950741'
>>> h(min)
'0f61485fd84d673c233e28e1d2a5acfe'
```

- À quoi ces fonctions servent-elles ?
  - Elles permettent de créer un identifiant pour un objet quelconque.
  - En particulier cela sert à comparer des données volumineuses.
  - Sauriez-vous trouver la différence entre les deux chaînes ?
  - Grâce à la fonction de hachage, la non-égalité est immédiate.

SHELL
```
>>> A='1234567891011121314151617181920212223242526 27'
>>> B='123456789101112131415161718192021222324252627'
>>> h(A)
'c4201256748240403 0e362f0c3813c15'
>>> h(B)
'e5a41e2edb893242fe25aad75dfcca66'
```

- Soit `s1` et `s2` deux chaînes et `h(s1)` et `h(s2)` leurs empreintes.
- On a la garantie suivante :
  - Si `h(s1)`≠`h(s2)` alors forcément `s1`≠`s2` (car `h` est une fonction)
- Les propriétés suivantes sont extrêmement probables :
  - Si `h(s1)`=`h(s2)` on peut en pratique considérer que `s1`=`s2` (on a 1 chance sur 340 282 366 920 938 463 463 374 607 431 768 211 455 de se tromper)
  - Si `s1` et `s2` sont très proches, `h(s1)` et `h(s2)` sont très différents.

- Supposons que l'on souhaite implémenter les ensembles par des listes
  - On réimplémente les ensembles pour des questions pédagogiques

SCRIPT
```
def ajouter(L,x):
    for e in L:
        if x == e: return #terminaison car x est déjà dans L
    L.append(x) # sinon, on ajoute x
    return
```

- Pour ajouter un élément, je dois comparer avec tous les éléments.
  - S'il y a `n` éléments de taille `T`, il faudra faire `n*T` comparaisons.

- Pour gagner en efficacité, on stocke chaque élément avec son empreinte.

SCRIPT
```
def ajouter(L,x):
    hx=h(x) # je calcule l'empreinte de x
    for e in L:
        (hy,y) = e # y est stocké avec son empreinte
        if hx == hy: return # le programme termine
    L.append((hx,x))
    return
```

- Dorénavant je dois faire seulement `N` comparaisons d'empreintes.

- Dans Python
    - Les ensembles sont implémentés de manière bien plus élaborées.
    - Mais ils utilisent des tables de hachage.
    - L'ajout d'élément ne dépend pas de la taille des éléments de E.
- En informatique en général : on utilise les empreintes
    - pour vérifier qu'un téléchargement correspond au bon fichier
        - On compare les empreintes (*MD5 check sum*)
        - Si l'empreinte est bonne, on a bien une version correcte du bon fichier
    - pour stocker des mots de passe sans les révéler.
        - On stocke les empreintes
        - On compare avec l'empreinte du mot de passe fourni par l'utilisateur.
        - À aucun moment on ne stocke les mots de passe
        - Une empreinte ne permet pas de retrouver le mot de passe
    - pour certifier la liste chaînée d'une *blockchain* (bitcoin).

Sommaire

- Un dictionnaire est une collection de couples `clé:valeur`.
    - `clé` est forcément non mutable ;
    - `valeur` peut être modifiée.

SHELL
```
>>> mon_panier = {'pommes':243 , 'poires':123 }
>>> mon_panier
{'pommes': 243, 'poires': 123}
```

- On accède aux valeurs en utilisant les clés comme indices.

SHELL
```
>>> mon_panier['poires']
123
>>> mon_panier['pommes']
243
```

- Un dictionnaire vide se note `{}` ou mieux `dict()`

- Toutes les clés doivent être distinctes

SHELL
```
>>> {'pommes':243 , 'poires':123, 'pommes':23 }
{'pommes': 23, 'poires': 123}
```

- L'accès à une valeur est extrêmement rapide.
  - C'est le principal intérêt des dictionnaires.
  - Les dictionnaires utilisent des tables de hachage
- La recherche est unidirectionnelle
  - on va de la clé vers les valeurs
- La clé doit être non mutable.
  - Typiquement des chaînes et des nombres.
  - On peut utiliser des tuples ne contenant que des éléments non mutables.

SHELL

```
>>> dico = { 'un':234 , 2:[3,4,5] , 3.5:'Bonjour' }
>>> dico[3.5]
'Bonjour'
>>> dico['un']
234
>>> dico['Bonjour'] # 'Bonjour' est une valeur !
Traceback (most recent call last):
 File "<console>", line 1, in <module>
KeyError: 'Bonjour'
```

- Si on utilise une clé qui n'existe pas, une exception est levée.
  - Pour savoir si une clé existe on peut utiliser mot-clé `in`

SHELL
```
>>> dico[8] # la clé 8 n'existe pas
Traceback (most recent call last):
 File "<console>", line 1, in <module>
KeyError: 8
>>> 8 in dico # 8 est-il une clé
False
```

- On veut affecter à x la valeur associée à la clé 8
  - Si une telle clé n'existe pas, on pose `x='inconnu'`

SCRIPT
```
try:
    x = dico[8]
except KeyError:
    x = 'inconnu'
```

SCRIPT
```
if 8 in dict:
    x = dico[8]
else:
    x = 'inconnu'
```

- Ou plus simplement en une ligne avec la méthode get

SHELL
```
>>> dico.get(8,'inconnu') # 8 n'est pas une clé.
'inconnu'
>>> dico.get(2,'inconnu') # 2 est la clé associée à [3,4,5]
[3, 4, 5]
```

- Un dictionnaire est mutable : il est modifiable.
- On peut modifier la valeur associée à une clé en utilisant l'affectation.

SHELL
```
>>> dico = { 11: 'unu' , 22:'Du' , 33:'tri' }
>>> dico[22]='du'
>>> dico
{11: 'unu', 22: 'du', 33: 'tri'}
```

- On peut ajouter un nouveau couple `clé:valeur` en utilisant l'affectation

SHELL
```
>>> dico
{11: 'unu', 22: 'du', 33: 'tri'}
>>> dico[44]='kvar'
>>> dico
{11: 'unu', 22: 'du', 33: 'tri', 44: 'kvar'}
```

- On peut supprimer un couple (`clé:valeur`) avec la commande pop

SHELL
```
>>> dico.pop(33)
'tri'
>>> dico
{11: 'unu', 22: 'du', 44: 'kvar'}
```

- On peut parcourir un dictionnaire en parcourant les clés ou les valeurs.

SCRIPT
```
def clés(dico):
    for k in dico.keys():
        v = dico[k]
        print(f"{k} ({v})")
```

SCRIPT
```
def valeurs(dico):
    for v in dico.values():
    # pas d'accès aux clés
        print(v,end=' ')
    print('')
```

SHELL
```
>>> naissance["Athos"]=1615
>>> naissance["Porthos"]=1617
>>> naissance["Aramis"]=1620
>>> naissance["d'Artagnan"]=1615
>>> clés(naissance)
Athos (1615)
Porthos (1617)
Aramis (1620)
d'Artagnan (1615)
>>> valeurs(naissance)#une valeur n'est pas forcément unique
1615 1617 1620 1615
```

- À quoi correspondent `dico.keys()` et `dico.values()` ?

SHELL
```
>>> naissance.keys()
dict_keys(['Athos', 'Porthos', 'Aramis', 'd'Artagnan'])
>>> naissance.values()
dict_values([1615, 1617, 1620, 1615])
```

- Ce sont des vues (*view*).
  - Ce ne sont pas des listes !
  - Mais ce sont des objets itérables.
  - On peut les convertir en tuples, ensembles ou listes.

SHELL
```
>>> list(naissance.keys())
['Athos', 'Porthos', 'Aramis', 'd'Artagnan']
>>> set(naissance.values())
{1617, 1620, 1615}
```

- On peut itérer directement sur un dictionnaire.
  - C'est comme si on itérait sur les clés.
  - `for k in dico.keys():` ⇔ `for k in dico:`

- En mathématique une application est injective si :
    - $f(x) = f(y)$ implique $x = y$, dit autrement, si $x \neq y$ alors $f(x) \neq f(y)$
- On cherche à savoir si un dictionnaire est injectif.
    - C'est-à-dire si chaque valeur est unique

SCRIPT

```python
def est_injectif(dico):
    for k1 in dico:
        for k2 in dico:
            if k1!=k2 and dico[k1]==dico[k2]:
                return False # collision !
    return True
```

- On peut aussi comparer la taille des ensembles de clés et de valeurs.

SCRIPT

```python
def est_injectif(dico):
    nb_clés =  len(set(dico.keys()))
    nb_values = len(set(dico.values()))
    return  nb_clés==nb_values
```

SHELL

```
>>> est_injectif(naissance)
False
>>> est_injectif({ 1:'eins' , 2:'zwei' , 3:'drei' })
True
```

- Objectif : mémoriser les calculs déjà faits pour pouvoir les réutiliser.
- Exemple :
  - on calcule `100!` c'est « long », il faut 100 multiplications.
  - on calcule ensuite `103!` : il faut 103 multiplications.
  - Si on avait mémorisé `100!`, il aurait suffit de 3 multiplications.
  - car `103! = 100! * 101 * 102 * 103`
- Il suffit de stocker les résultats dans un dictionnaire.
  - `n` sera la clé
  - le résultat de `fact(n)` sera la valeur.
- Principe de l'algorithme
  - Si `n` est une clé du dictionnaire, on renvoie la valeur associée.
  - Sinon, on calcule `v=n*fact(n-1)` et on ajoute `n:v` dans le dictionnaire.
- Cette méthode s'appelle la mémoïsation.

SCRIPT

```
mémoire_cache = { 0:1 } # fact(0)=1

def fact(n):
    global mémoire_cache # global est facultatif
    # on ne modifie pas l'ensemble mais son contenu
    if n in mémoire_cache:
        return mémoire_cache[n]
    else:
        v = n*fact(n-1)
        mémoire_cache[n]=v
        return v
```

SHELL

```
>>> from time import time
>>> len(mémoire_cache)
1
>>> t = time() ; x=fact(800) ; t800 = time()-t
>>> len(mémoire_cache)
801
>>> t = time() ; x=fact(810) ; t810 = time()-t
>>> len(mémoire_cache)
811
>>> print(t810/t800)
0.009580838323353293
>>> print(f'{t810/t800:.1%}')
1.0%
```

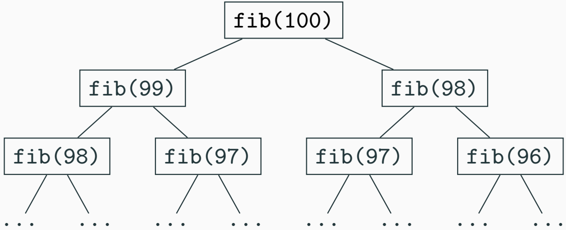
- Le calcul de `fact(810)` est 100 fois plus efficace que celui de `fact(800)`

- On a rencontré des récurrences doubles. Exemple : la suite de Fibonacci.
  - Très peu efficaces
  - Les mêmes calculs sont faits de nombreuses fois.

SCRIPT

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

On souhaite retenir les résultats intermédiaires (mémoïsation)



Implémentation de Fibonacci

- Soit T_n le nombre d'appels récursif lors du calcul de `fib(n)` on a
  - $1 + T_n = 2 \cdot \texttt{fib(n)}$ car $1 + T_n$ vérifie la même formule de récurrence que `fib(n)` (mais en partant de 2 au lieu de 1)

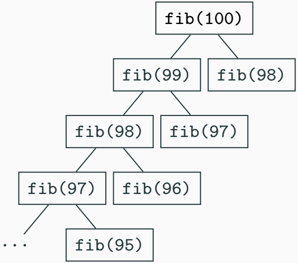
$$\begin{cases} T_n = 1 + T_{n-1} + T_{n-2} \\ T_0 = T_1 = 1 \end{cases} \quad \text{et donc} \quad \begin{cases} (1+T_n) = (1+T_{n-1}) + (1+T_{n-2}) \\ (1+T_0) = (1+T_1) = 2 \end{cases}$$

- Pour calculer `fib(100)` il faut donc $2 \cdot \texttt{fib}(100) - 1$ appels récursifs
  - Supposons que le calcul ne prenne que 10^{-12} s par appel ;
  - le temps nécessaire sera de $\approx 10^9$s ≈ 36 années.

SCRIPT
```python
mem = dict()
def fib(n):
    if n==0 or n==1:
        mem[n] = 1
    elif n not in mem:
        mem[n] = fib(n-1)+fib(n-2)
    return mem[n]
```

SHELL
```
>>> fib(100) #Quasi instantané
573147844013817084101
```

Le nouvel arbre d'appels est un « peigne » qui compte seulement 201 appels récursifs.



Sommaire

- Nous avons représenté une matrice 2×2 par une liste de listes (cours 7).
- Généralisons cette idée aux matrices $m\times n$ (m lignes et n colonnes).

$$A = \begin{pmatrix} 0 & 1 \\ 1 & 2 \\ 2 & 3 \end{pmatrix} = \begin{pmatrix} A_{0,0} & A_{0,1} \\ A_{1,0} & A_{1,1} \\ A_{2,0} & A_{2,1} \end{pmatrix}$$

SHELL
```
>>> A = [[0, 1], [1, 2], [2, 3]]
>>> len(A)
3
>>> (A[0], len(A[0]))
([0, 1], 2)
>>> A[2][1]
3
```

- Les lignes et colonnes sont numérotées à partir de 0.
  - `len(A)` donne le nombre de lignes (hauteur)
  - `len(A[0])` donne le nombre de colonnes (largeur)
  - `A[li]` donne la ligne d'indice li
  - `A[li][co]` donne le coefficient à la ligne `li` et à la colonne `co` : $A_{li,co}$
  - et pour la colonne d'indice `co` ?

SCRIPT
```
def dimensions(A): # Fonction utile pour la suite
    return (len(A),len(A[0])) # nombre de lignes et de colonnes
```

- On ne peut accéder directement qu'aux lignes.
- Comment accéder aux colonnes ?

SCRIPT
```
def colonne(A, co):
    res = []
    for li in range(len(A)):
        res.append(A[li][co])
    return res
```

- En plus pythonesque, en utilisant les compréhensions de listes.

SCRIPT
```
def colonne(A, co):
    return [A[li][co] for li in range(len(A))]
```

$$A = \begin{pmatrix} 0 & 1 \\ 1 & 2 \\ 2 & 3 \end{pmatrix}$$

SHELL
```
>>> A = [[0, 1], [1, 2], [2, 3]]
>>> len(A)
3
>>> colonne(A,0)
[0, 1, 2]
>>> colonne(A,1)
[1, 2, 3]
```

- Comment déterminer si un objet Python est une matrice ?
  - c'est une liste de listes
  - Les lignes et les colonnes doivent être non vides
  - toutes les colonnes ont la même taille

SCRIPT

```python
def est_matrice(A):
    # A doit être une liste non-vide
    if type(A) != list or A==[]:
        return False
    # A[0] doit être une liste non-vide
    if type(A[0]) != list or A[0]==[]:
        return False
    #Toutes les lignes doivent être des listes de même taille
    for ligne in A:
        if type(ligne) != list or len(ligne) != len(A[0]):
            return False
    return True
```

SHELL

```
>>> est_matrice([])
False
>>> est_matrice([[],[],[]])
False
>>> est_matrice(12)
False
```

SHELL

```
>>> est_matrice([1,2,3])
False
>>> est_matrice([[1,2],[3,4],[5,6]])
True
>>> est_matrice([[1,2],[3],[5,6]])
False
```

- Une matrice nulle est une matrice ne contenant que des 0

SCRIPT
```
def matrice_nulle(n,m):
    A=[]
    for ligne in range(n):
        L=[]
        for colonne in range(m):
            L.append(0)
        A.append(L)
    return A
```

Matrice 2×3 nulle :

$$\begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}$$

- Une matrice identité est une matrice carrée contenant des 1 sur la diagonale et des 0 ailleurs.

SCRIPT
```
def matrice_identité(n):
    A = matrice_nulle(n,n)
    for i in range(n):
        A[i][i]=1
    return A
```

Matrice identité 4×4 :

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Exercice : écrire ces fonctions en une ligne avec des compréhensions de liste

- L'opposé d'une matrice est formé des opposés des élèments initiaux.

$$\text{L'opposé de } \begin{pmatrix} 1 & -2 \\ -3 & 4 \end{pmatrix} \text{ est } \begin{pmatrix} -1 & 2 \\ 3 & -4 \end{pmatrix}$$

- Comment rédiger un tel programme ?
  - On crée une matrice nulle de la bonne taille
  - On y affecte ensuite les bonnes valeurs

SCRIPT

```
def opposé(A):
    (n,m) = dimensions(A)
    B = matrice_nulle(n,m)
    for i in range(n):
        for j in range(m):
            B[i][j] = - A[i][j]
    return B
```

- On est obligé de partir d'une nouvelle matrice.
  - En effet si j'écris B=A, toute modification de B affectera A.
  - Voir le cours 5 sur la gestion de la mémoire concernant les listes.

- La trace d'une matrice est la somme des éléments diagonaux.

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{pmatrix}$$

La trace de A vaut $1+6+2+7=16$

- Le concept n'a de sens que dans une matrice carrée

SCRIPT

```
def trace(A):
    (n,m)=dimensions(A)
    if n != m :
        raise ValueError('Trace : matrice non carrée')
    tr = 0
    for i in range(n):
        tr = tr + A[i][i]
    return tr
```

- On peut ajouter deux matrices coefficient par coefficient
  - les matrices doivent être de même dimensions.

$$\begin{pmatrix} 30 & 1 & 20 \\ 3 & 11 & 50 \end{pmatrix} + \begin{pmatrix} 4 & 60 & 3 \\ 20 & -1 & 7 \end{pmatrix} = \begin{pmatrix} 34 & 61 & 23 \\ 23 & 10 & 57 \end{pmatrix}$$

SCRIPT

```
def somme(A,B):
    (n,m) = dimensions(A)
    if (n,m) != dimensions(B):
        raise ValueError('Dimensions incompatibles')
    C = matrice_nulle(n,m)
    for i in range(n):
        for j in range(m):
            C[i][j] = A[i][j] + B[i][j]
    return C
```

- On peut multiplier deux matrices entre elles [Wikipédia]
  - la longueur de la première doit être égale à la hauteur de la deuxième
  - On pose N ce nombre en commun.
  - La matrice produit $M = AB$ est définie par $M_{i,j} = \sum_{k=1}^{N} A_{i,k} \times B_{k,j}$

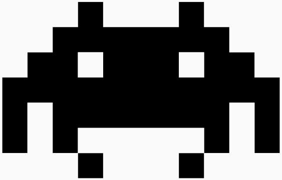
SCRIPT

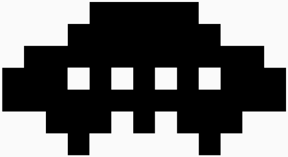
```
def coefficient_produit(A,B,i,j):
    (n,m)=dimensions(A)
    c = 0
    for k in range(n):# de 0 à n-1 (et non comme en math de 1 à n)
        c = c + A[i][k]*B[k][j]
    return c

def produit(A,B):
    (la,ca) = dimensions(A) ; (lb,cb) = dimensions(B)
    if lb != ca:
        raise ValueError('Dimensions incompatibles')
    C = matrice_nulle(la,cb)
    for i in la:
        for j in cb:
            C[i][j] = coefficient_produit(A,B,i,j)
    return C
```

- On utilise une sous-fonction pour éviter d'avoir trop de `for` imbriqués.

- Une image est un tableau de pixels.
- Un pixel (en noir et blanc) ne peut avoir que deux valeurs :
    - 0 : pixel blanc
    - 1 : pixel noir
- Une image pourra donc être représentée par une matrice de 0 et de 1





- Voir TP !

?

Cours 8 — Ensembles, dictionnaires et matrices