Université Nice Sophia Antipolis
Faculté des Sciences

Programmation avec Python - 2019
Etienne Lozes
L1-Sciences

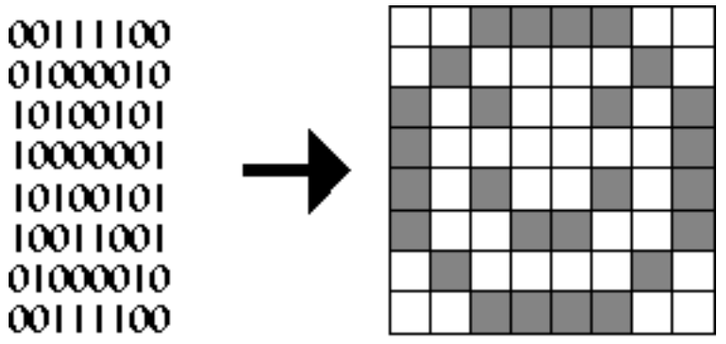
Cours n°8

http://deptinfo.unice.fr/~elozes

Dictionnaires,

Ensembles,

Matrices et Bitmaps



Où en sommes-nous des types de données ?

- Quels types de données avons-nous déjà rencontrés ?

- Quelques *types simples* : `int`, `float`, `bool`

```
>>> type(-45)
<class 'int'>
```

```
>>> type(23.7)
<class 'float'>
```

```
>>> type(False)
<class 'bool'>
```

- Quelques *types composés*, suites numérotées d'objets. Nous avons vu trois sortes de **séquences** : `str`, `list`, `tuple`

```
>>> type('foo')
<class 'str'>
```

```
>>> type([2,6,1])
<class 'list'>
```

```
>>> type((2,6,1))
<class 'tuple'>
```

- Nous allons découvrir deux nouveaux types composés, les **ensembles** et les **dictionnaires** : `set`, `dict`

```
>>> type({2,6,1})
<class 'set'>
```

```
>>> type({'prix':50, 'nom':'Eska'})
<class 'dict'>
```

Les ensembles

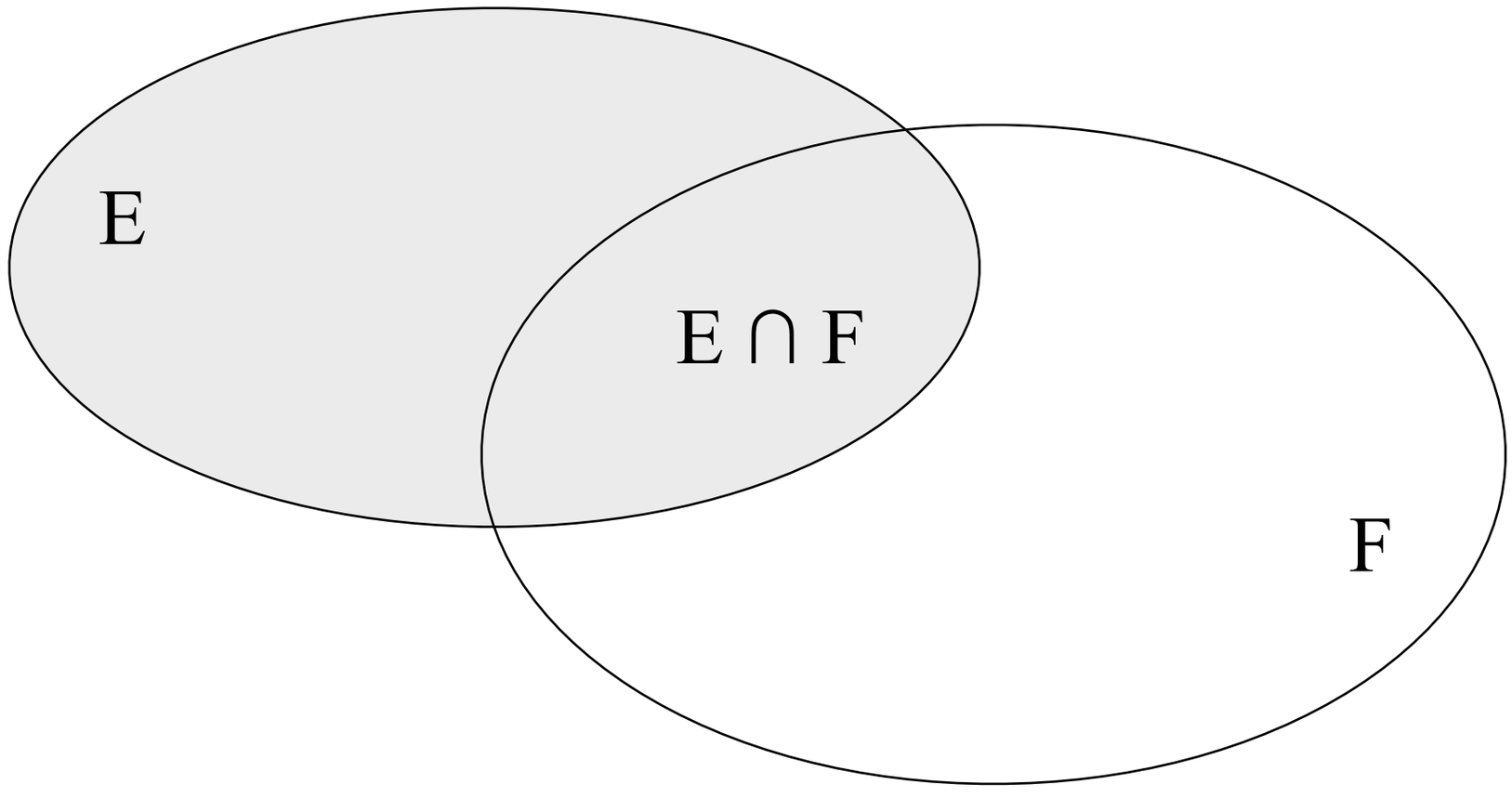


Diagramme de Venn

Origine :
Georg Cantor (1845-1918),
mathématicien allemand.

Qu'est-ce qu'un ensemble ?

• Un **ensemble** en Python est une collection d'objets ***sans répétition*** et **sans ordre** (donc **sans numérotation**). Ce n'est PAS une séquence !

• On les note comme en maths avec des accolades {...}. Les éléments sont de types quelconques. Exemples :

```
{5,3,-8,2}    {'o','e','y','u','a','i'}    {5,'foo',(3,-2),7.4}
```

• L'ensemble vide se note `set()` et non `{}` qui crée un dictionnaire !

• Un ensemble est défini à l'ordre près :

```
>>> {3,1,2} == {2,3,1}
True
>>> {3,2,3,2} == {2,3}
True
```

• Un ensemble paraît trié en interne mais c'est uniquement pour accélérer la recherche d'un élément :

```
>>> e = {3,'b',2,'a',5}
>>> e
{2,3,'a',5,'b'}
```

Accès aux éléments d'un ensemble

• Les éléments d'un ensemble ne sont **pas numérotés**. On ne peut pas utiliser une notation comme `e[i]` puisque parler de l'élément numéro i n'a pas de sens ! Le type **set** n'est **pas une séquence** !

• L'opérateur `in` permet de savoir si un objet **appartient** à un ensemble.

• L'opération E < F permet de tester si l'ensemble E est **strictement inclus** dans l'ensemble F.

```
>>> {2,4} < {1,2,3,4}
True
```

• Axiome du choix : il est possible d'obtenir un élément (lequel ?) d'un ensemble `E` et de le supprimer en même temps avec la méthode `pop()`.

```
>>> E = {5,2,3,1}
>>> x = E.pop()
>>> (x, E)
(1, {2,3,5})
```

*donc un ensemble est **mutable** !*

Le nombre d'éléments d'un ensemble

- On parle aussi du ***cardinal*** d'un ensemble. Il s'agit de la fonction `len`.

```
>>> E = {5,2,3,1}
>>> len(E)
4
```

- Si elle n'existait pas, on pourrait la programmer. Notons au passage qu'un ensemble est un objet itérable et mutable.

```
def cardinal(E) :
    res = 0
    while E != set() :
        x = E.pop()
        res = res + 1
    return res
```

Aïe : mutation !

```
def cardinal(E) :
    res = 0
    for x in E :
        res = res + 1
    return res
```

```
def cardinal(E) :
    return sum(1 for x in E)
```

Comment construire un ensemble ?

- En **extension** :

```
>>> E = {5,2,3,1}
>>> E
{1, 2, 3, 5}
```

- En **compréhension** :

```
>>> E = {x*x for x in range(20) if x % 3 == 0}
>>> E
{0, 225, 36, 9, 144, 81, 324}
```

- Avec le **constructeur** `set(...)` de la classe `set` :

```
>>> E = set('aeiouy')
>>> E
{'o', 'i', 'e', 'a', 'y', 'u'}
```

```
>>> E = set([5,2,5,6,2])
>>> E
{2, 5, 6}
```

- En ajoutant un élément à un ensemble E avec la méthode `E.add(x)`

```
>>> E = {5,3,2,1}
>>> E.add(8)
```

```
>>> E
{8,1,2,3,5}
```

mutation !

Opérations sur les ensembles

- L'**ensemble vide** se note `set()` et non `{}` !

- La **réunion** E ∪ F = {x : x ∈ E ou x ∈ F} se note E | F en Python.

```
>>> {3,2,5,4} | {1,7,2,5}
{1, 2, 3, 4, 5, 7}
```

- L'**intersection** E ∩ F = {x : x ∈ E et x ∈ F} se note E & F en Python.

```
>>> {3,2,5,4} & {1,7,2,5}
{2, 5}
```

- La **différence** E - F = {x : x ∈ E et x ∉ F} se note E - F en Python.

```
>>> {3,2,5,4} - {1,7,2,5}
{3, 4}
```

- Ne pas confondre `E - {x}` qui construit un nouvel ensemble, avec la méthode `E.remove(x)` qui supprime x de l'ensemble `E` (mutation).

Que puis-je mettre dans un ensemble?

```
>>> {(1,2),(3,4)}          # un ensemble de tuples
{(1, 2), (3, 4)}
>>> E = {[1,2],[3,4]}      # un ensemble de listes ?
TypeError: unhashable type: 'list'
```

- Un ensemble construit par `set(...)` est **mutable**. Mais ses éléments ne peuvent pas être mutables. On peut faire des ensembles de tuples (par exemple des ensembles de points du plan) mais pas d'ensembles dont les éléments sont des listes ou des ensembles !

- En interne, chaque opération sur un ensemble Python fait appel à une **fonction de hachage.**

Qu'est-ce qu'une fonction de hachage?

• Une fonction de hachage h est une fonction mathématique qui permet de **calculer un résumé** (en anglais **digest**) d'une suite de bits de longueur quelconque. La longueur du digest, elle, est fixée à l'avance. Une fonction de hachage courante, MD5, calcule un digest de 128 bits, représenté le plus souvent par son écriture hexadécimale.

```
>>> import hashlib
>>> def h(x) : # MD5: {valeurs Python} -> {suite de 128 bits}
. . .     return hashlib.md5(repr(x).encode()).hexdigest()
>>> h(3)   # le digest de l'entier 3
'eccbc87e4b5ce2fe28308fd9f2a7baf3'
>>> len(h(3)) * 4   # la longueur du digest en bits
128
>>> h(fr_dic) # le digest du dictionnaire de la langue française
'cb17bf00f0ad5d36a4ec81f699438641'
```

A quoi sert une fonction de hachage?

Essentiellement à détecter rapidement que deux données volumineuses sont différentes.

Exemple : soit s1 et s2 deux très longues chaînes de caractères (les textes de deux livres). Je veux savoir si les deux livres sont identiques. A priori je dois comparer caractère par caractère pour être sûr que s1 et s2 sont bien identiques. S'ils sont différents uniquement au dernier caractère, je vais mettre du temps à me rendre compte que $s1 \neq s2$.

Supposons maintenant que je connaisse h(s1) et h(s2). Je peux comparer très rapidement h(s1) et h(s2) [je compare 128 bits].
Si par chance $h(s1) \neq h(s2)$, j'en déduis que $s1 \neq s2$.
Vous voyez pourquoi? **h est une fonction!**

Et les ensembles dans tout ça?

Supposons que je veuille calculer l'ensemble E des livres de ma bibliothèque. Chaque fois que je rajoute un livre dans E, il faut que je le compare à tous ceux qui sont déjà dans E pour ne pas risquer de le répéter… Je fais donc |E| comparaisons. Si je fais chaque comparaison caractère par caractère, cela va me prendre un temps O(|E|x L), où L est la longueur moyenne d'un livre…

Mais je peux me souvenir du digest de chaque livre déjà dans E. Quand un nouveau livre s1 doit être ajouté, je calcule h(s1).
[ce me prend un temps un temps O(L) aussi long que pour UNE comparaison]
Puis je compare h(s1) à tous les h(s) pour s dans E.

- S'il n'y a pas de $s \in E$ tel que h(s1) = h(s), je sais que s1 est un nouveau livre.
- Si en revanche on trouve s2 tel que h(s1)=h(s2), il me reste à comparer s1 et s2 caractère par caractère… mais avec une forte chance, j'aurais s1=s2, et je le ferai une seule fois.

Au final, j'aurais pris un temps O(L) qui ne dépend pas de la taille de E.

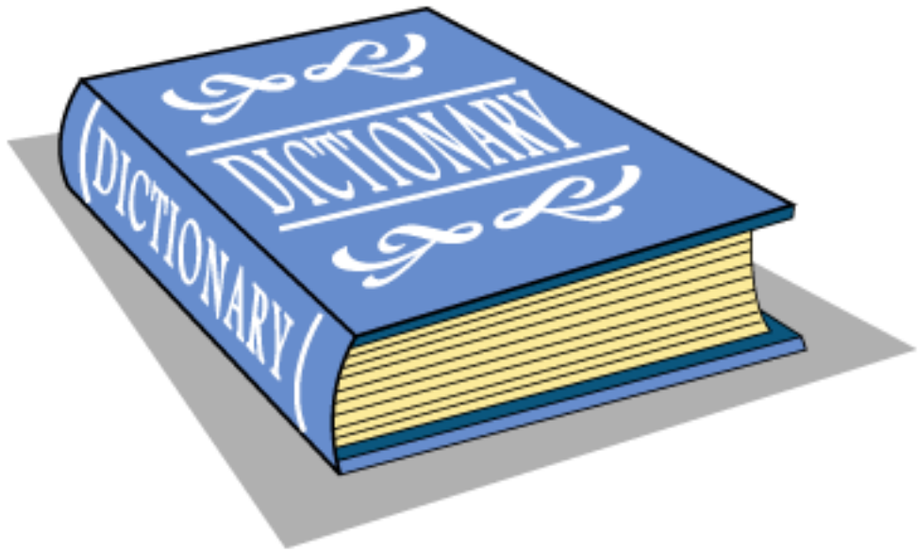
Et dans Python?

Python gère les ensembles de manière plus élaborées, à l'aide de **tables de hachage** que vous étudierez en L2, mais le principe général reste le même: quand j'ajoute un élément e à un ensemble E en Python, je prends un temps moyen proportionnel au temps mis pour calculer h(e), donc constant en la taille de E.

Et en dehors de Python?

Les fonctions de hachage sont omniprésentes en informatique, en particulier en cryptographie. Elles servent par exemple à vérifier qu'un fichier volumineux qu'on a téléchargé sur internet est arrivé sans être corrompu (**MD5 check sum**), elles permettent de **stocker des mots de passe** sans les révéler (on stocke uniquement le digest du mot de passe), elles sont utilisées pour certifier les listes chaînées de la **blockchain**, etc.

Les dictionnaires



Qu'est-ce qu'un **dictionnaire** ?

• Un **dictionnaire** est une collection non numérotée de couples `var:val` où `var` est un objet *non mutable* (la **clé**) et où `val` est n'importe quelle valeur. Toutes les clés sont **distinctes** !

```
>>> stock = {'poires':51, 'pommes':243}
>>> stock
{'pommes': 243, 'poires': 51}                    # aucun ordre !
```

```
>>> len(stock)
2
```

```
>>> 'pommes' in stock
True
```

```
>>> stock['pommes']
243
```

Le dictionnaire est une représentation en Python d'une fonction (au sens mathématique) qui à une clé associe une valeur.

$$\begin{array}{rccc} f: & \{\texttt{poires}, \texttt{pommes}\} & \to & \mathbb{N} \\ & \texttt{poires} & \mapsto & 51 \\ & \texttt{pommes} & \mapsto & 243 \end{array}$$

• Le **dictionnaire vide** se note `{}` ou mieux `dict()`.

Accès aux éléments d'un dictionnaire

• On peut TRES VITE **accéder à la valeur associée à une clé**. C'est le principal intérêt des dictionnaires (techniquement : des ***tables de hachage***). La recherche est pratiquement instantanée !

```
>>> stock['pommes']
243
```

```
>>> stock[243]
KeyError : 243
```

*243 n'est pas une **clé** !*

• La recherche est donc **unidirectionnelle:** on va de la clé vers la valeur

• **La clé doit être non mutable**. Donc essentiellement des chaînes et des nombres mais pas de listes. On peut utiliser des tuples comme clés à condition qu'ils ne contiennent aucun objet mutable.

```
>>> dico = {1:'one', 2:'two', 3:'three'}
>>> dico[2]
'two'
```

Clé inexistante : exception ?

- Si l'on demande la valeur associée à une clé inexistante, une exception `KeyError` est levée. On peut l'attraper au vol.

```
>>> x = dico[8]
KeyError : 8
```

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

- Il est aussi possible de demander si la clé existe, avec l'opérateur `in` :

```
>>> 2 in dico
True
```

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

On peut aussi tout faire en une ligne avec la méthode get

```
>>> dico.get(2, 'inconnu')
'two'
```

```
>>> dico.get(8, 'inconnu')
‘inconnu'
```

Comment construire un dictionnaire ?

- En extension : on fournit tous les couples, dans un ordre quelconque.

```
dico = {1:'one', 2:'two', 3:'three'}
```

- Un dictionnaire est **mutable**. On peut :

  - modifier la valeur associée à une clé :

```
>>> dico[2] = 'deux'
>>> dico
{1: 'one', 2: 'deux', 3: 'three'}
```

  - ajouter un nouveau couple clé-valeur :

```
>>> dico[4] = 'four'
>>> dico
{1: 'one', 2: 'deux', 3: 'three', 4: 'four'}
```

  - supprimer un couple dont on connaît la clé :

```
>>> dico.pop(3)
'three'
>>> dico
{1: 'one', 2: 'deux', 4: 'four'}
```

Voir les clés et les valeurs

• On peut vouloir obtenir la liste de toutes les clés ou bien la liste de toutes les valeurs. Les méthodes `keys()` et `values()` retournent des **vues** (*views*) sur les clés et les valeurs.

```
>>> dico.keys()
dict_keys([1, 2, 4])
>>> dico.values()
dict_values(['one', 'deux', 'four'])
```

des objets bizarres...

• Ces *vues* sont des **objets itérables** que l'on peut parcourir avec une boucle `for`, ou transformer en listes/ensembles :

```
cpt = 0
for k in dico.keys() :
    if k % 2 == 0 :
        cpt = cpt + 1
```

```
>>> list(dico.keys())
[1, 2, 4]
>>> set(dico.values())
{'four', 'one', 'deux'}
```

```
cpt = sum(1 for k in dico.keys() if k % 2 == 0)
```

Itération dans un dictionnaire

- Un dictionnaire étant un *objet itérable*, on peut… itérer dessus !

```
>>> dico
{1:'one', 2:'two', 3:'three'}
```

- **En réalité, on itère sur les clés** :

```
cpt = 0
for k in dico :
    if k % 2 != 0 :
        cpt = cpt + 1
```

⇔

```
cpt = 0
for k in dico.keys() :
    if k % 2 != 0 :
        cpt = cpt + 1
```

⇓

```
cpt = sum(1 for k in dico if k % 2 != 0)
```

```
>>> cpt
2
```

- Mais on peut aussi **itérer sur les valeurs** :

```
for v in dico.values() :
  if len(v) == 3 : print(v,end=' ')
```

⇒ one two

Application : vérifier si un dictionnaire représente une fonction injective

Je peux chercher si je trouve deux clés qui ont la même valeur

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

Je peux aussi vérifier si la fonction est bijective, car on travaille sur des ensembles finis. f:E->F est bijective ssi f(E)= F et |E| = |F|

```
def est_injectif(D) :
   return len(set(D.keys())) == len(set(D.values()))
```

Application : une mémo-fonction

Programmer **une fonction qui se souvient des calculs déjà effectués !**

• Exemple de la factorielle. Je calcule `100!` qui demande `99` multiplications. Je calcule ensuite `101!` qui demande `100` multiplications, **au lieu d'une seule** puisque `101! == 100! * 101`. Je veux que ma fonction `fac(n)` se souvienne qu'elle a déjà calculé `100!`. Comment ?

• Il me suffit de gérer un ***dictionnaire*** associé à la fonction qui va contenir tous les couples `n:v` tels que `fac(n) == v` ait déjà été calculé !

> Pour calculer `n!` :
> - si `n` est une clé du dictionnaire associé, fini.
> - sinon :
>
> calculer `v = (n-1)! * n` ; stocker `n:v` ; retourner `v`

• On dit que le dictionnaire associé est une ***mémoire cache***.

```
mem = {0:1}              # initialisation du dictionnaire associé
                         # je sais calculer fac(0) = 1
def memo_fac(n) :
    global mem
    if n not in mem.keys() :   # n! est-il déjà calculé ?
        # non? je le calcule, et je le mémorise pour plus tard
        mem[n] = memo_fac(n-1) * n
    return mem[n]
```

```
>>> memo_fac(100)
933262154439441526816992388562 6.....00000
>>> len(mem)   # la mémoire a grossi !
101
>>> memo_fac(30)
265252859812191058636308480000000
```

***Immédiat**, car déjà calculé !*

- remarque technique : le global mem est ici optionnel: contrairement à ce qu'on pourrait penser, la fonction memo_fac ne fait que lire la variable mem: elle modifie le contenu du dictionnaire mais pas la variable elle-même. La variable mem est donc automatiquement reconnue comme globale. Mais cela ne fait aucun mal de mettre le global!

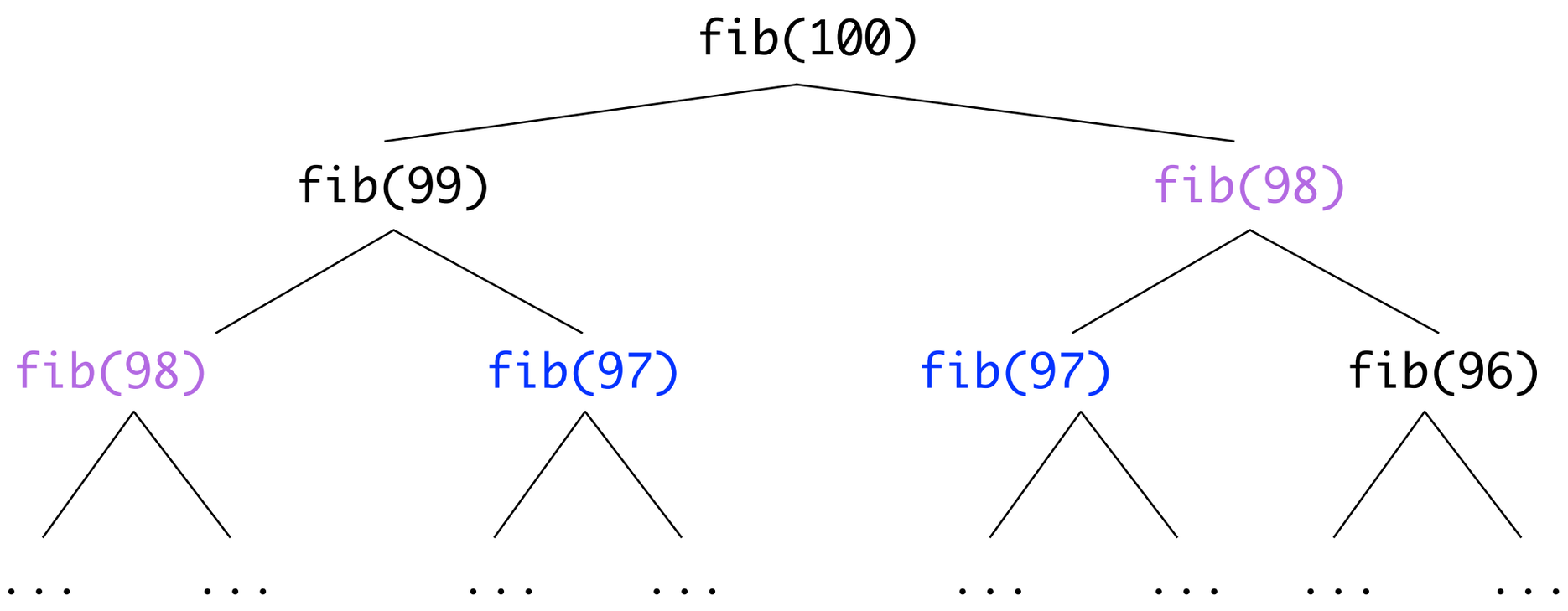
Application aux récurrences redondantes

Dans le cours 2, on a rencontré des récurrences doubles qui faisaient des calculs redondants, par exemple dans la suite de Fibonacci

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

```
>>> fib(4)
5
>>> fib(100)
 Keyboard interrupt
```

Ctrl-C



Le temps de calcul nécessaire est proportionnel à la taille de l'arbre.

Faisons le calcul

taille(A_{100}) = 2 * nb_feuilles(A_{100}) + 1

nb_feuilles(A_{100}) = nb_feuilles(A_{99}) + nb_feuilles(A_{98}) = fibo(100) !

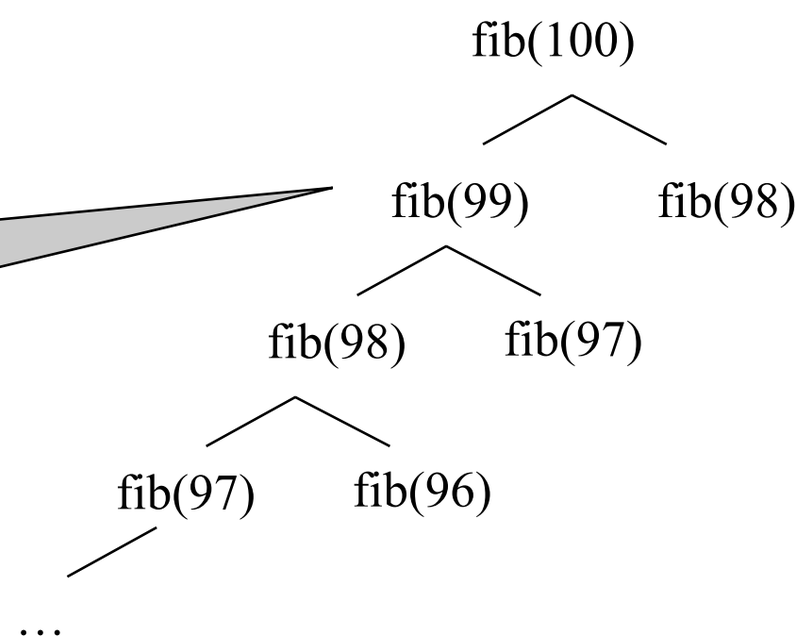
On en déduit un arbre d'environ 10^{68} noeuds. Si le calcul prend 10^{-12} sec par noeud, le temps nécessaire est ~ 10^{21} sec ~ 10^{5} milliards d'années

Essayons en mémoïsant…

```
mem = {0:1, 1:1}

def fib(n) :
  if n not in mem.keys() :
     mem[n] = fib(n-1) + fib(n-2)
  return mem[n]
```

```
>>> fib(4)
6
>>> fib(100)
573147844013817084101
>>>
```

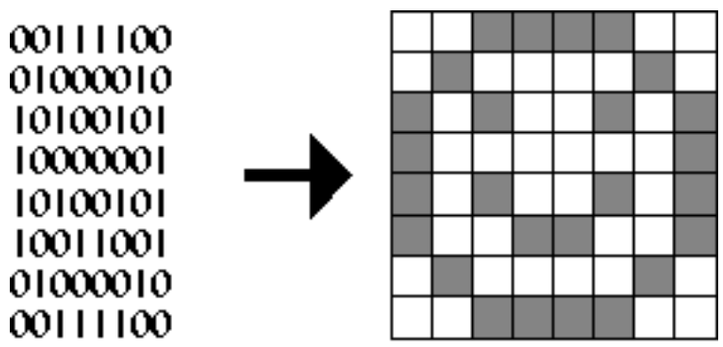
Le nouvel arbre d'appel est un « peigne » qui compte seulement 198 appels récursifs!



Matrices

et

Bitmaps



Les matrices n x m

Dans le cours précédent, on a vu que l'on pouvait représenter une matrice 2x2 par une liste de listes. Généralisons cette idée aux matrices n x m.

$$\begin{pmatrix} M_{0,0} & M_{0,1} & M_{0,2} & M_{0,3} \\ M_{1,0} & M_{1,1} & M_{1,2} & M_{1,3} \\ M_{2,0} & M_{2,1} & M_{2,2} & M_{2,3} \end{pmatrix}$$

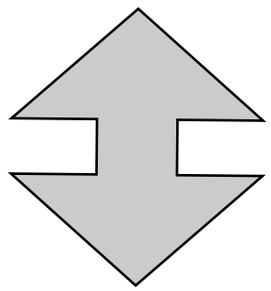
$$A = \begin{pmatrix} 0 & 1 \\ 1 & 2 \\ 2 & 3 \end{pmatrix}$$

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

- `len(A)` me donne le nombre de lignes (hauteur)
- `len(A[0])` correspond au nombre de colonnes (largeur)
- `A[li][co]` : le nombre à la ligne d'indice `li` et colonne d'indice `co`
- `A[li]` : la ligne d'indice `li`
- **la colonne d'indice co?**

Extraire une colonne

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



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

- `len(A)` me donne le nombre de lignes (hauteur)
- `len(A[0])` correspond au nombre de colonnes (largeur)
- `A[li][co]` : le nombre à la ligne d'indice `li` et colonne d'indice `co`
- `A[li]` : la ligne d'indice `li`

Reconnaitre une matrice

```
def est_matrice(M) :
    if type(M) != list : return False
    if len(M) == 0 : return True
    if type(M[0]) != list : return False
    largeur = len(M[0])
    if largeur == 0 : return False
    for li in range(len(M)) :
        if type(M[li]) != list or len(M[li]) != largeur :
            return False
    return True
```

```
>>> est_matrice([[0, 1], [1, 2], [2, 3]])
True
>>> est_matrice([])
True
>>> est_matrice([[]])
False
>>> est_matrice([[1], [2, 3]])
False
```

Quelques matrices particulières

```
def matrice_nulle(hauteur, largeur) :
    return [[0 for co in range(largeur)] for li in
range(hauteur)]
```

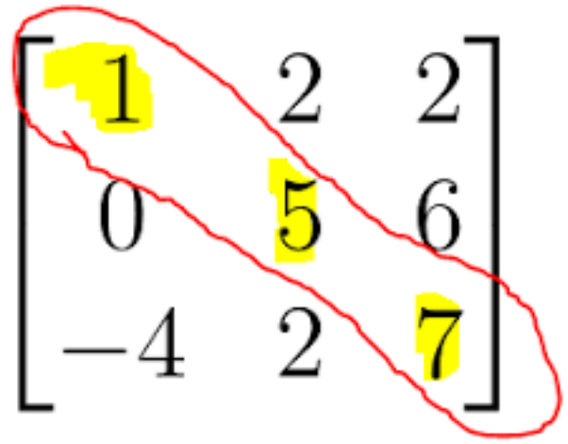
```
def matrice_identité(n) :
    res = matrice_nulle(n, n)
    for i in range(n) :
        res[i][i] = 1
    return res
```

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

Ou avec des listes par compréhensions et une **expression conditionnelle**

```
def matrice_identité(hauteur, largeur) :
    return [[1 if li==co else 0 for co in largeur] for li in hauteur]
```

Calculer la trace d'une matrice



Solution 1 : on utilise un accumulateur

```
def trace(M) :    # la trace d'une matrice carré
    acc = 0
    for i in range(len(M)) :
        acc = acc + M[i][i]    # le ième elt de la diagonale
    return acc
```

Solution 2 : on utilise sum

```
def trace(M) :          # la trace d'une matrice carrée
    return sum(M[i][i] for i in range(len(M))
```

Addition de matrices

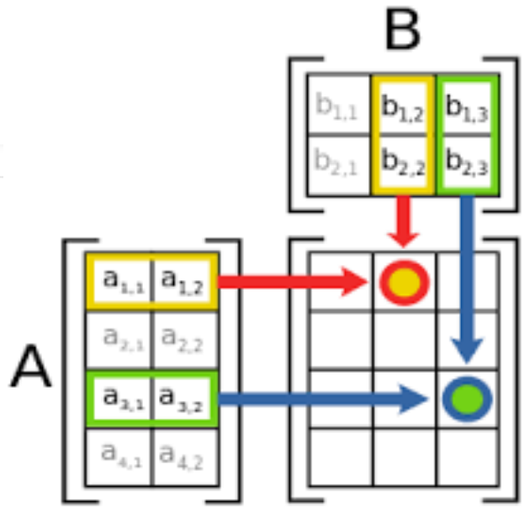
$$\begin{pmatrix} 0 & 1 \\ 1 & 2 \\ 2 & 3 \end{pmatrix} + \begin{pmatrix} 0 & 1 \\ 1 & 2 \\ 2 & 3 \end{pmatrix} = \begin{pmatrix} 0 & 2 \\ 2 & 4 \\ 4 & 6 \end{pmatrix}$$

```
def matrice_som(M1, M2) :
    (H1, L1) = (len(M1), len(M1[0]))
    (H2, L2) = (len(M2), len(M2[0]))

    assert(L1 == L2 and H1 == H2)  # M1, M2 de même dim

    res = matrice_nulle(H1, L1)

    for li in range(H1) :
        for co in range(L1) :
            res[li][co] = M1[li][co] + M2[li][co]
    return res
```

On omet la solution par listes par compréhensions; c'est bien de toute façon de décomposer les trois étapes :

1) vérifier la compatibilité des dimensions
2) préparer une matrice nulle aux bonnes dimensions pour contenir le résultat
3) calculer res[li][co] pour toutes les coordonnées (li,co) à considérer

Produit de matrices




```
def matrice_prod(A, B) :
    (HA, LA) = (len(A), len(A[0]))
    (HB, LB) = (len(B), len(B[0]))

    # étape 1 : condition de dimension
    assert(LA == HB)

    # étape 2 : on préparer la matrice qui contiendra le résultat
    res = matrice_nulle(HA, LB)

    # étape 3 : on remplit la matrice résultat
    for i in range(HA) :
        for j in range(LB) :
            res[i][j] = sum(A[i][k] * B[k][j] for k in range(LA))
    return res
```

$$M_{i,j} = \sum_{k} A_{i,k} \times B_{k,j}$$

Concaténation de matrices

Exemple : concaténation horizontale

$$\begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} \\ A_{1,0} & A_{1,1} & A_{1,2} \\ A_{2,0} & A_{2,1} & A_{2,2} \end{pmatrix} \oplus \begin{pmatrix} B_{0,0} & B_{0,1} \\ B_{1,0} & B_{1,1} \\ B_{2,0} & B_{2,1} \end{pmatrix} = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & B_{0,0} & B_{0,1} \\ A_{1,0} & A_{1,1} & A_{1,2} & B_{1,0} & B_{1,1} \\ A_{2,0} & A_{2,1} & A_{2,2} & B_{2,0} & B_{2,1} \end{pmatrix}$$

Cette opération n'est pas très courante en mathématique… mais en informatique, elle est très naturelle: elle correspond à mettre côté à côte deux images bitmaps

cf TP !