Les séquences (2)

- tuple, liste -

Qu'est-ce qu'une **séquence** ?

• **Une séquence est une suite finie de valeurs numérotées** (distinctes ou pas, de n'importe quel type). Par exemple, les éléments des types `range` et `str` sont des séquences (toutes deux *non mutables*).

• La fonction `pos()` du graphisme tortue retournait un couple de nombres :

```
>>> pos()
(28,-100)
```

• Un tel couple est un cas particulier de **tuple** (couple, triplet, etc). Les tuples sont pratiques pour les **fonctions à plusieurs résultats**.

• Les **listes** sont plus souples que les tuples pour stocker et modifier un grand nombre d'objets. Contrairement aux tuples, elles sont ***mutables***.

```
'Hello !'        (2,-1,'red',3.5)        [2,-1,'red',3.5]
((20,5),(10,0),(-1,7))                   [[2,-3],[1,5]]
```


Les **tuples** généralisent les couples

• Programmons la division euclidienne de l'entier a ≥ 0 par l'entier b > 0 qui devra retourner deux résultats q (le *quotient*) et r (le *reste*) :

$$a = bq + r \quad avec \quad 0 \le r < b$$

```
a | b          43 | 5
r | q           3 | 8
```

• Lorsqu'une fonction doit retourner plusieurs résultats, elle n'en retournera qu'un seul ! Par exemple sous la forme d'un tuple.

```
def division(a,b) :
    q = 0
    while a >= b :
        a = a - b
        q = q + 1
    return (q,a)
```

```
>>> d = division(43,5)
>>> d
(8, 3)
>>> type(d)
<class 'tuple'>
>>> len(d)
2
>>> d[0] = 1     # TypeError
'tuple' object does not
support item assignment
```

• Les tuples ne sont **pas mutables** ⟶

```
>>> (2,3) + (3,4,5)
(2, 3, 3, 4, 5)
```


Les **listes** généralisent les tuples

• Programmons la fonction `diviseurs(n)` qui retourne la **liste** des **diviseurs non triviaux** (distincts de 1 et n) d'un entier n ≥ 2. Ne connaissant pas à l'avance le nombre d'éléments du résultat, on utilise une liste plutôt qu'un tuple (pas de méthode `append` dans les tuples).

```
def diviseurs(n) :              # dans [2,n-1]
    res = []                    # la liste vide
    for d in range(2,n) :
        if n % d == 0 : res.append(d)
    return res
```

```
>>> d = diviseurs(1000)
>>> (d, len(d))
([2, 4, 5, 8, 10, 20, 25, 40, 50, 100, 125, 200, 250, 500], 14)
```

• La méthode `append` est **efficace** et propre aux listes : `res.append(d)` demande à la liste `res` d'accepter un nouvel élément en `d` **queue**.

Accès aux éléments d'une séquence

1. Accès par rang

• L'accès est commun aux chaînes, tuples et listes avec la notation indexée par **crochets**.

```
>>> str = 'Hello!'
>>> len(str)
6
>>> (str[0], str[-1])
('H', '!')
```

• Python vérifie la légalité :

```
>>> str[6]
Error : string index out of range
```

```
>>> t = (6,3,'red',-1,True)
>>> len(t)
5
>>> (t[0], t[-1])
(6, True)
```

• Les **indices négatifs** facilitent l'accès aux éléments en queue.

```
>>> (str[-1], str[-6])
('!', 'H')
```

H	e	l	l	o	!
0	1	2	3	4	5
-6	-5	-4	-3	-2	-1

```
>>> lis = [6,3,'red',-1,True]
>>> len(lis)
5
>>> (lis[0], lis[-1])
(6, True)
```


2. Accès par contenu

• La méthode `index` permet de connaître le rang d'un élément d'une séquence donnée :

```
>>> lis
[3, 2, 5, 4, 5, 1]
>>> lis.index(5)
2
>>> lis.index(10)
Error : 10 is not in list
```

```
>>> str
'foobar'
>>> str.index('b')
3
>>> [3,5,7,9].index(7)
2
```

• Avec un second argument, on peut décider où démarre la recherche.

```
>>> lis.index(5,3)
4
```

je cherche 5 à partir de l'indice 3

• EXO : comment utiliser `index` pour programmer une fonction `positions(x,L)` retournant la liste des indices `i` tels que `L[i] == x` ?...

• NB : la méthode `s1.find(s2)` de la classe `str` retourne le même résultat que `find` mais retourne -1 en cas d'absence (cf TP3 exo 7).

Construction d'une nouvelle séquence

1. Construction en extension

• Pour les **tuples** : une suite d'éléments entre parenthèses, séparés par une virgule : `()` `(2,)` `(1,'rouge',3.5,[2,5,3])`

• Pour les **listes** : une suite d'éléments entre crochets, séparés par une virgule : `[]` `[2]` `[1,'rouge',3.5,[2,5,3]]`

2. Concaténation de deux séquences

• On peut coller côte à côte (*concaténer*) deux séquences pour construire une nouvelle séquence, avec l'opérateur +

```
>>> 'bon' + 'jour'
'bonjour'
```

```
>>> (1,'a') + (2,'b')
(1, 'a', 2, 'b')
```

```
>>> [1,5] + [5,6]
[1, 5, 5, 6]
```

• Si s_1 et s_2 sont deux séquences : `len(s1 + s2) = len(s1) + len(s2)`

3. Extraction d'une tranche de séquence

`s[i:j]` dans `[i,j-1]`

• La notation des **tranches** (*slices*) est valide pour toute séquence.

```
>>> s = 'Hello!'
>>> lis = [3,2,5,1,4]
>>> sol = (28,-5,'red')
```

```
>>> s[1:5]
'ello'
>>> lis[:3]
[3, 2, 5]
>>> sol[1:]
(-5, 'red')
```

• RAPPEL : les éléments d'une séquence peuvent être :
  - de type quelconque
  - en nombre quelconque.

• En particulier, on peut emboîter les séquences. Un élément de $\mathbf{R}^4$ sera vu comme `(a,b,c,d)` et un élément de $\mathbf{R}^2 \times \mathbf{R}^2$ comme `((a,b),(c,d))` par exemple.

• Une matrice carrée d'ordre 3 pourra être architecturée sur la forme d'une liste :

`[[a,b,c],[d,e,f],[g,h,i]]`

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$$

4. Construction d'une liste ou d'un tuple en compréhension

• Une **construction par compréhension (de liste)** consiste à imiter la notation mathématique *{f(x) tels que x ∈ E et P(x)}* en fournissant une propriété P de x.

```
>>> [x*x for x in range(6)]
[0, 1, 4, 9, 16, 25]
>>> [c for c in 'azerty']
['a', 'z', 'e', 'r', 't', 'y']
```

[f(x) for x in <iterable>]

• Une expression conditionnelle `if` suivant le `for` permet de filtrer les éléments souhaités :

```
>>> [x*x for x in range(10) if x % 2 == 0 and x*x > 20]
[36, 64]
```

• Limitez-vous svp à ces deux constructions :

`[f(x) for x in ...]` `[f(x) for x in ... if ...]`

Modification d'une liste

1. Ajout d'un élément à une liste

• Seules les **listes** sont **mutables**, avec les méthodes `append` (en queue), `insert` (à un rang donné), `remove` (un élément), `pop`, etc.

```
>>> lis = [2,6,5]
>>> lis[1] = 4
>>> lis
[2, 4, 5]
```

```
>>> lis.append(1)
>>> lis
[2, 4, 5, 1]
```

```
>>> lis.insert(2,10)
>>> lis
[2, 4, 10, 5, 1]
```

rang = 2

• **Les chaînes de caractères et les tuples ne sont PAS mutables !**
Dans une chaîne `s`, l'élément `s[i]` n'est pas une variable, mais une constante. Par contre la variable `s` peut être affectée :

```
>>> s = 'Hello!'
>>> s[1] = 'a'
```

Error : 'str' object does not support item assignment

```
>>> s = s[0]+'a'+s[2:]
>>> s
'Hallo!'
```


Méthodes usuelles sur les listes

`list.append(x)`	*Add an item to the end of the list.*
`list.extend(L)`	*Extend the list by appending all the items in the given list.*
`list.insert(i,x)`	*Insert an item at a given position. The first argument is the index of the element before which to insert.*
`list.remove(x)`	*Remove the first item from the list whose value is x. It is an error if there is no such item.*
`list.pop(i)`	*Remove the item at the given position in the list, and return it. If no index is specified, a.pop() removes and returns the last item in the list.*
`list.index(x)`	*Return the index in the list of the first item whose value is x. It is an error if there is no such item.*
`list.count(x)`	*Return the number of times x appears in the list.*
`list.sort()`	*Sort the items of the list, in place.*
`list.reverse()`	*Reverse the elements of the list, in place.*

• La méthode `append` est très efficace, mais pas la méthode `insert` !

http://docs.python.org/release/3.4/tutorial/datastructures.html

Déstructuration d'un tuple ou d'une liste

• Lorsqu'une fonction retourne un tuple ou une liste, comment récupérer son résultat ? Deux manières :

- sous la forme d'une seule valeur de type tuple/liste:

```
d = division(43,5)
```

```
>>> d
(8, 3)
```

```
lis = [x*x for x in range(6,8)]
```

```
>>> lis
[36, 49]
```

- en *déstructurant* le tuple/liste en composantes variables (même structure des deux côtés) :

```
(a,b) = division(43,5)
```

```
>>> a
8
```

```
>>> b
3
```

```
[u,v] = [x*x for x in range(6,8)]
```

```
>>> u
36
```

```
>>> v
49
```

Premiers algorithmes avec les listes

Liste des chiffres dans une chaîne

• Soit à calculer la liste des chiffres d'une chaîne s. Nous utilisons la méthode `isdigit()` de la classe `str`.

```
>>> '456'.isdigit()
True
```

```
>>> '4d5'.isdigit()
False
```

• La première méthode est classique, facilement transportable dans d'autres langages de programmation :

```
def chiffres(s) :
    """Retourne la liste des chiffres d'une chaîne s"""
    res = []
    for c in s :
        if c.isdigit() : res.append(c)
    return res
```

```
>>> chiffres('les 2 ou 3 bateaux à 430 euros')
['2', '3', '4', '3', '0']
```


• La seconde méthode est plus *pythonesque* donc plus courte et passe par une **compréhension de liste** :

```
def chiffres(s) :
    """Retourne la liste des chiffres d'une chaîne s"""
    return [car for car in s if car.isdigit()]
```

• La troisième méthode est encore plus *pythonesque* et passe par `filter` dont vous avez pu trouver le nom dans la doc. En effet `filter(f,L)` permet de **filtrer les éléments d'une liste** `L` répondant `True` à la fonction `f`. MAIS : le résultat `f` n'est pas une liste, c'est un **objet itérable** !

Hors Programme !

```
>>> f = filter(lambda c : c.isdigit(), 'a1b2c3')
>>> f
<filter object at 0x108c26a10>
>>> for x in f : print(x,end=' ')
1 2 3
# mais maintenant l'objet f est épuisé !
```

au besoin :
`f = list(f)`

Liste de nombres premiers

• Soit à calculer la liste des nombres premiers de `[2,n]`. Reprenons les idées du cours 2 en commençant par programmer la fonction `ppdiv(n)` retournant le plus petit diviseur de l'entier `n`.

• Avec deux optimisations :

- On teste le diviseur 2 à part, de sorte à ne parcourir que des entiers impairs ≥ 3 ensuite.
- Si on ne trouve aucun diviseur $d \leq \sqrt{n}$, il n'y en aura pas d'autre avant n lui-même : n est un nombre premier *[pourquoi ?]*.

```
def ppdiv(n) :                          # n >= 2
    if n % 2 == 0 : return 2
    for d in range(3,int(sqrt(n))+1,2) :
        if n % d == 0 : return d
    return n
```

```
>>> ppdiv(1003)
17
>>> ppdiv(2003)
2003
```

2003 est donc premier

• Il est alors facile de tester si un nombre entier n est premier :

```
def estPremier(n) :
    return n >= 2 and n == ppdiv(n)
```

```
>>> estPremier(1003)
False
>>> estPremier(2003)
True
```

• et de construire la liste des nombres premiers jusqu'à n :

```
def premiers(n) :            # n >= 2
    return [p for p in range(2,n+1) if estPremier(p)]
```

```
>>> premiers(50)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
>>> t1 = time(); n = len(premiers(100000)); t2 = time()
>>> (n, t2-t1)
(9592, 0.2213151454925537)    ; il y en 9592, en 0.2 seconde !
```

• Exo : Ecrire la fonction premiers(n) sans compréhension de liste !

Tri d'une liste en ordre croissant

• Python possède deux primitives pour trier une liste L de nombres en ordre croissant :

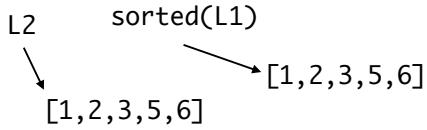
- la **fonction** sorted(L) qui construit une ***nouvelle copie*** triée de L.
- la **méthode** L.sort() qui trie la liste L ***sur place*** (sans créer de nouvelle liste, donc en modifiant le contenu de L).

```
>>> L1 = [3,1,6,5,2]
>>> sorted(L1)
[1, 2, 3, 5, 6]
>>> L1
[3, 1, 6, 5, 2]
```

```
>>> L2 = [3,1,6,5,2]
>>> L2.sort()
>>> L2
[1, 2, 3, 5, 6]
```

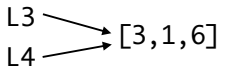
• Les contenus de deux listes peuvent être comparés par égalité :

```
>>> L2 == sorted(L1)
True
>>> L2 is sorted(L1)
False
```



• Ne confondez pas les opérateurs == et is sur les listes. Vous aurez rarement besoin de is : **L3 is L4** exprime que L3 et L4 ne sont que des noms différents pour la même liste en mémoire !

```
>>> L3 = [3,1,6]
>>> L4 = L3
>>> L4 is L3
True
```



• Le temps de calcul de cet algorithme pour trier une liste à n éléments est proportionnel à **n log(n)**. Petite expérience :

```
def chrono_sort() :
    L1 = [randint(1,100) for i in range(100000)]
    L2 = [randint(1,100) for i in range(200000)]
    t = time(); L1.sort(); t = time() - t
    print('Temps pour 100000 :',t,'ms')      ⟶  0.035 ms
    t = time(); L2.sort(); t = time() - t
    print('Temps pour 200000 :',t,'ms')      ⟶  0.079 ms
```

• On voit qu'en multipliant le nombre d'éléments par 2, on fait un peu plus que doubler le temps de calcul. Ceci nous rassure sur le **n log(n)**...

Comment trier une liste ? Exemple du tri par sélection

• Il existe beaucoup d'algorithmes de tri. En voici un moins efficace que sort, le **tri par sélection**. Il consiste à ramener successivement tous les minima en tête.

• Il est moins efficace que sorted. Vous constaterez en TP que son coût est proportionnel à n^2...

```
>>> L = [5,6,7,2,4]
>>> tri_sel(L)
L = [2, 6, 7, 5, 4]
L = [2, 4, 7, 5, 6]
L = [2, 4, 5, 7, 6]
L = [2, 4, 5, 6, 7]
>>> L
[2, 4, 5, 6, 7]
```

```
def tri_sel(L) :
    """tri par sélection d'une liste de nombres, sur place"""
    n = len(L)
    for i in range(n-1) :
        m = imin(L,i)          # indice du min à partir de L[i]
        (L[i],L[m]) = (L[m],L[i])    # échange !
        print('L =',L)                # juste pour voir...
```

imin en TP !

• Aucun résultat. Pas de return.