Programmation avec Python - Automne 2016
Bruno Martin & Jean-Paul Roy
L1-Sciences

Cours n°6

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

Listes, complexité, exceptions

Rappels sur les listes

• Une **liste** est une suite finie **mutable** de valeurs numérotées (distinctes ou pas, de n'importe quel type).

```
>>> L = [3, 'foo', 12.05, (0, 1), True, [3, 11, 8]]
>>> (len(L),type(L))
(6, <class 'list'>)
>>> for x in L : print(x,' : ',type(x))

3  :  <class 'int'>
foo  :  <class 'str'>
12.05  :  <class 'float'>
(0, 1)  :  <class 'tuple'>
True  :  <class 'bool'>
[3, 11, 8]  :  <class 'list'>
```

```
>>> L[2] = L[0] + L[2]            # mutation de L !
>>> L
[3, 'foo', 15.05, (0, 1), True, [3, 11, 8]]
```


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.*

https://docs.python.org/3/tutorial/datastructures.html#more-on-lists

Un tri est-il performant ?

• Le tri prédéfini `sort` de Python est très efficace. Pour trier une liste aléatoire de n éléments, il a un coût en $O(n \log(n))$...

• Le tri par sélection (cours/TP 5) est peu efficace. Pour trier une liste aléatoire de n éléments, il a un coût proportionnel à n^2...

n	n log(n)	n^2
1000	6907	1000000
2000	15201	4000000

• Le programmeur s'efforcera toujours de minimiser le coût :

n^2 est meilleur que 2^n
n est meilleur que n log(n) qui est meilleur que n^2
log(n) est meilleur que n
1 = Cte est meilleur que log(n)

Un tri performant : quicksort

• Sans vouloir battre le tri `sort`, programmons un tri performant, dont le coût moyen est en $n\ log(n)$: le tri rapide (***quicksort***) ou ***tri par pivot***. Son principe illustre la stratégie DIVISER POUR RÉGNER.

• Il procède en trois phases successives :

1. On partage la liste L en deux sous-listes L1 et L2.
2. On trie séparément L1 et L2 pour obtenir LT1 et LT2.
3. On réunit les listes triées LT1 et LT2.

• **Etape 1** : stratégie du **pivot**. On choisit un élément p dans la liste (par exemple le premier si la liste est en vrac) et on pose L* = L-{p}.

`L1 = {x ∈ L* : x < p}` et `L2 = {x ∈ L* : x ≥ p}`

• **Etape 2** : la récurrence !

• **Etape 3** : on recolle les morceaux. LT = LT1 ⊔ {p} ⊔ LT2

• Exemple : `L = [5,3,8,1,7,2]` et le **pivot** `p = 5`. Alors :

`p` `L*`

1. **Séparer** L en deux **par pivotage** autour de 5

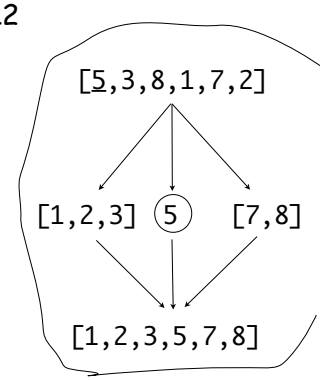
```
L1 = {x ∈ L* : x < 5} == [3,1,2]
L2 = {x ∈ L* : x ≥ 5} == [8,7]
```

2. **Trier** (**par hypothèses de récurrence**) L1 et L2

```
LT1 = quicksort(L1) == [1,2,3]
LT2 = quicksort(L2) == [7,8]
```

3. **Rassembler** les résultats obtenus

```
LT = [1,2,3] + [5] + [7,8]
LT == [1,2,3,5,7,8]
```



• Une **compréhension de listes** facilite l'écriture de la scission !

```
def quicksort(L) :
    '''Quicksort par récurrence'''
    if L == [] :
        return L
    p = L[0]     # le pivot
    L1 = [x for x in L[1:] if x < p]
    L2 = [x for x in L[1:] if x >= p]
    LT1 = quicksort(L1)
    LT2 = quicksort(L2)
    return LT1 + [p] + LT2
```

Trier

n	temps
10000	0.05
20000	0.12

1. Séparer

2. Trier par récurrence

3. Rassembler

• Le choix du premier élément L[0] comme **pivot** est excellent lorsque la liste est dans le désordre, mais devient pénalisant si elle est presque triée... Dans ce cas, on choisit le pivot au hasard et on le supprime aussitôt :

```
pivot = L.pop(randint(0,len(L)-1))
```


Pourquoi un coût en n log(n) ?

• La **complexité** peut porter sur le comportement d'un algorithme :

- dans le cas le plus favorable
- dans le pire des cas
- en moyenne

• Sans entrer dans les détails (L2 et L3-Info), développons des arguments intuitifs : une mathématique d'ingénieur, pas de mathématicien... Soit c_n le **coût** de trier une liste de longueur n (nous mesurons le nombre de fois que l'on ajoute un élément à une liste).

• Si la liste L est longue, en vrac et le pivot choisi au hasard, L1 et L2 sont à peu près de même longueur. Donc chaque appel par récurrence (lignes 8-9) a un coût de $c_{n/2}$. Enfin, le coût du recollement est de n.

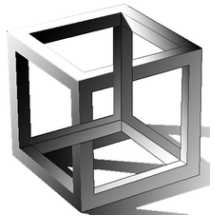
• Arguments un peu rapides, une véritable analyse requiert des arguments probabilistes plus compliqués, mais l'idée est là !

		coût
1	def quicksort(L) :	c_n
2	"""Quicksort par récurrence"""	0
3	if L == [ ] : return [ ]	0
4	else:	0
5	p = L[0]	0
6	L1 = [x for x in L[1:] if x < p]	n
7	L2 = [x for x in L[1:] if x >= p]	n
8	LT1 = quicksort(L1)	$c_{n/2}$
9	LT2 = quicksort(L2)	$c_{n/2}$
10	return LT1 + [p] + LT2	n

• Equations de complexité : $\begin{cases} c_0 = c_1 = 0 \\ c_n = 2c_{n/2} + 3n \end{cases}$

• Vous verrez en TD comment calculer le terme général de cette suite et vous montrerez que c_n est d'ordre n log(n)...

Sur la recherche d'un élément dans une liste, et les phénomènes exceptionnels auxquels elle peut conduire...



Recherche séquentielle dans une liste

• Problème fréquent en programmation : **rechercher la position** d'un élément dans une séquence (ici une liste).

• Soit L une liste de nombres, mais **sans ordre** : les nombres sont donnés *en vrac*. Il peut y avoir des **répétitions**. Par exemple :

```
L = [3, 2, 8, 6, 2, 4, 7, 6, 5]
```

• Je cherche l'élément x, par exemple x = 6. Il peut apparaître plusieurs fois. Je propose de chercher de la gauche vers la droite. Si x ∉ L, je retourne -1, sinon je retourne le premier i tel que L[i] == x.

```
def chercher(x, L):
  for i in range(len(L)):
    if L[i] == x :
       return i
  return -1
```

```
>>> chercher(6,L)
3
>>> chercher(9,L)
-1
```

```
>>> L.index(6)
3
>>> L.index(9)
ValueError
```

Coût en O(n), si n = len(L)

Recherche dichotomique dans une liste triée

• Lorsque la liste est dans le **désordre**, la recherche est **séquentielle** et coûte **O(n)** si n = len(L) : on *paye n* dans le pire des cas...

• Lorsque la liste est **triée**, donc dans l'ordre (plusieurs ordres possibles), on peut **accélérer la recherche**. La stratégie consiste à **abandonner la moitié de la liste chaque fois** !

1	2	5	7	8	10	12	13	15

0 *m*

• En pseudo-langage, en supposant la liste croissante ⇨

N.B. Si l'élément x est répété, je ne garantis plus de trouver la première apparition !

• soit m l'indice du milieu de la liste
• si x == L[m], alors :
le résultat est m
sinon :
si x < L[m], alors :
continuer dans la moitié gauche
sinon :
continuer dans la moitié droite

• **Complexité dans le pire des cas** : combien de fois puis-je diviser n = len(L) par 2 ? Exemple avec n = 1000 :

n=1000, 500, 250, 125, 62, 31, 15, 7, 3, 1, 0

$\longleftrightarrow$

k

• En remontant de la droite vers la gauche, cela revient à effectuer k multiplications par 2, donc $n \simeq 2^k$, d'où $k \simeq \log_2(n)$.

```
>>> log(1000) / log(2)
9.965784284662087
```

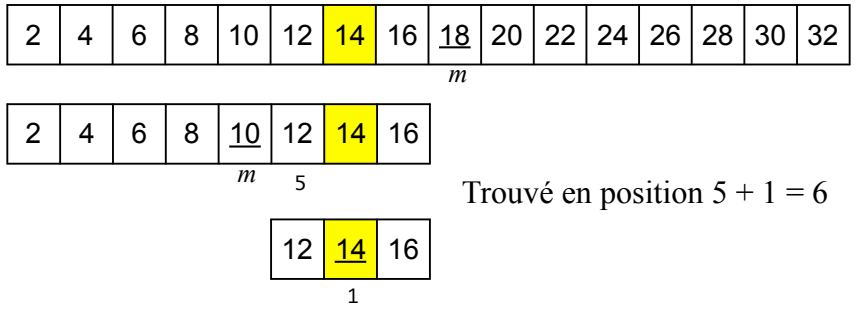
• Donc **bien meilleur que la recherche séquentielle** en O(n)...

```
def rech_dicho(x,L):      # retourne un i tel que L[i] == x
    if L == [] : return -1
    m = len(L) // 2
    if x == L[m] : return m
    if x < L[m] : return rech_dicho(x,L[:m])
    rechdroite = rech_dicho(x,L[m+1:])
    if rechdroite == -1 : return -1
    else : return m + 1 + rechdroite       # <- attention !
```



```
>>> L = [2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34]
>>> rech_dicho(14,L)
6
>>> rech_dicho(7,L)
-1
```

• Etapes de la recherche de x = 14 :



N.B. Inconvénient du programme : beaucoup de calculs de tranches ! Vous éliminerez toutes ces tranches en TD6 (exo 4)...

Attention : un coût sera-t-il amorti ?

• Donc deux types de recherches dans une liste :

Liste triée	Liste non triée
Recherche dichotomique	Recherche séquentielle
Coût en O(log n)	Coût linéaire en O(n)
Rapide !	Lent.

• Si une liste n'est pas triée, j'ai donc deux possibilités :

- faire une recherche séquentielle ⇒ coût = O(n)
- la trier d'abord puis faire une recherche dichotomique. ⇒ coût = O(n log(n) + log(n)) = O (n log(n))

• Alors, quelle est la meilleure méthode ?

• Si je n'effectue la recherche qu'**une seule fois**, le tri n'est pas intéressant car $n\ log(n)$ est plus cher que n.

• Ceci nous conduit à l'idée d'**amortissement** qui prend aussi en compte le nombre de fois que je vais utiliser un programme !

- si je procède k fois à une recherche séquentielle (sans tri), j'aurais un coût de **kn**.
- si je procède k fois à une recherche dichotomique (avec un seul tri), j'aurais un coût total de **n log(n) + k log(n)**.

• Mais si n est fixe et k assez grand (combien ?), il vaut mieux payer **n log(n) + k log(n)** que **kn**...

MORALE : lorsqu'on s'intéresse à la complexité, il faut à la fois tenir compte de l'algorithme et du nombre de fois qu'on l'utilisera. Un algorithme peut être un ***one shot*** !

Les phénomènes exceptionnels

• Vous savez qu'il peut y avoir des **erreurs** dans un calcul, dues à la mauvaise gestion des cas particuliers :

```
>>> L = [6, 4, 7, 2, 1, 8]
>>> L[6]
IndexError: list index out of range
```

IndexError

```
>>> L.index(5)
ValueError: 5 is not in list
```

ValueError

```
>>> 2 / 0
ZeroDivisionError: division by zero
```

ZeroDivisionError

etc.

• En fait, ces erreurs portent le nom d'**exceptions**.

• Le programmeur peut gérer ces cas exceptionnels et **reprendre la main** pour proposer un traitement exceptionnel !

• A la page 11, nous avions programmé la fonction `chercher(x,L)` qui retournait la position de la première occurrence de x dans L, ou bien -1.

```
def chercher(x, L):
  for i in range(len(L)):
    if L[i] == x :
      return i
  return -1
```

```
>>> L = [3,2,8,6,2,4,7,6,5]
>>> chercher(6,L)
3
```

• Or Python fournit la méthode `index` qui demande à une liste L la position d'un élément. Hélas elle ne renvoie pas -1 mais **lève une exception** en cas d'échec ! Comment transformer cette erreur en -1 ?

```
>>> L.index(6)
3
>>> L.index(9)
ValueError
```

• Réponse : en traitant l'exception, en essayant de la capturer au passage pour proposer un traitement de ce cas exceptionnel.

• Retenez : **il n'y a pas d'erreur, seulement des EXCEPTIONS !**

• Si j'invoque brutalement la méthode index en demandant `L.index(x)`, je risque de provoquer une exception. Je vais simplement **ESSAYER** de faire le calcul et regarder ensuite si j'ai fait une erreur...

• En Python, on utilise la construction ***try: ... except: ...***

```
def cherche(x,L) :
    try :
        return L.index(x)
    except :
        return -1
```

```
try :
    <instr1>
    ...
except :
    <instr2>
    ...
```

- Si au cours de l'exécution du bloc d'instructions <*instr*$_1$>... une exception se produit, l'exécution du bloc est abandonnée et le bloc <*instr*$_2$>... est exécuté.
- Si l'exécution du bloc <*instr*$_1$>... s'est déroulée normalement, le bloc `except` n'est pas utilisé. Est-ce assez clair ?

• Autre exemple. Je souhaite convertir une chaîne comme `"5634"` en un entier, ou bien `"45.8084"` en un flottant. Facile, Python me propose les fonctions `int(...)` et `float(...)`. Elles provoquent des exceptions :

```
>>> float('56.78')
56.78
>>> float('trois')
ValueError: could not convert string to float: 'trois'
```

• Au final, les exceptions permettent d'exprimer le comportement général d'un calcul tout en sachant qu'il pourra y avoir quelques cas d'erreurs exceptionnelles qui seront traités à part.

• Il y a différentes exceptions : `ValueError`, `ZeroDivisionError`, `KeyError`, etc. Un bloc `except:` les attrape toutes. Il est possible de n'attraper que certaines exceptions, ou d'envisager des traitements spécialisés pour chaque exception possible...

```
try :
    ....
except ValueError :
    ...
```