python

```
print("Hello, world!")
```

FORMATION PYTHON : Le TRI en Python

Ce chapitre a pour but d'initier aux algorithmes de tri en Python.

https://sites.google.com/site/formateuranissaid/

Introduction

- Les méthodes de tri sont évidemment fondamentales.
- On dispose de nombreux algorithmes qui se distinguent par leurs complexités en temps et en place mémoire.
- Dans ce cours nous allons illustrer quelques-uns de ces algorithmes avec des listes ou des tableaux d'entiers (ou de flottants).
- Lorsque le programme contient une boucle while, il ne faudra pas oublier de s'assurer qu'elle se termine.
- Nous chercherons aussi à estimer le nombre d'opérations que son exécution entraine : On parle de calcul de complexité.

Introduction

- On considère ici des tableaux ou listes d'entiers ou de flottants.
- En Python, on peut trier une liste à l'aide de la méthode sort : si a est une liste d'entiers ou de flottants, a.sort() modifie la liste en la liste triée.
- En revanche, la fonction sorted(a) est une fonction qui prenant une liste ou un tableau renvoie la liste (ou le tableau) des éléments triés par ordre croissant.
- Ainsi : a = [4,1,3,2] ; a.sort() # modifie a en [1, 2, 3, 4]

  sorted(a) # renvoie la liste [1, 2, 3, 4]; a inchangée !

Introduction

- Une fois triée, une liste peut servir pour d'autres traitements comme :
- L'insertion d'une nouvelle donnée à sa bonne position en prenant en considération l'ordre établi ;
- Les algorithmes dichotomiques qui utilisent les listes triées pour accélérer la recherche…
- On distingue deux grandes catégories de tri :
  - Tris *lents* (par sélection, à bulles, par insertion…)
  - et les tris *rapides* (par fusion, quicksort,…)

Les Tris en Python

1. TRI PAR SELECTION
2. TRI PAR INSERTION
3. TRI RAPIDE
4. TRI FUSION
5. EXERCICES (TD)

1. TRI PAR SELECTION

1. Tri par sélection : Principe

- Commencer à chercher l'indice du minimum de la liste **L** ;
- Permuter le **1er** élément de la liste avec l'élément minimum trouvé ;
- Chercher l'indice du minimum des éléments de la sous liste **L[1 :]** et le permuter avec le 2ème élément…
- Continuer ce principe jusqu'à ce que la liste devienne triée.
- D'une manière générale, on échange l'élément à la position **i** avec le minimum de la sous liste **L[i+1 :]**

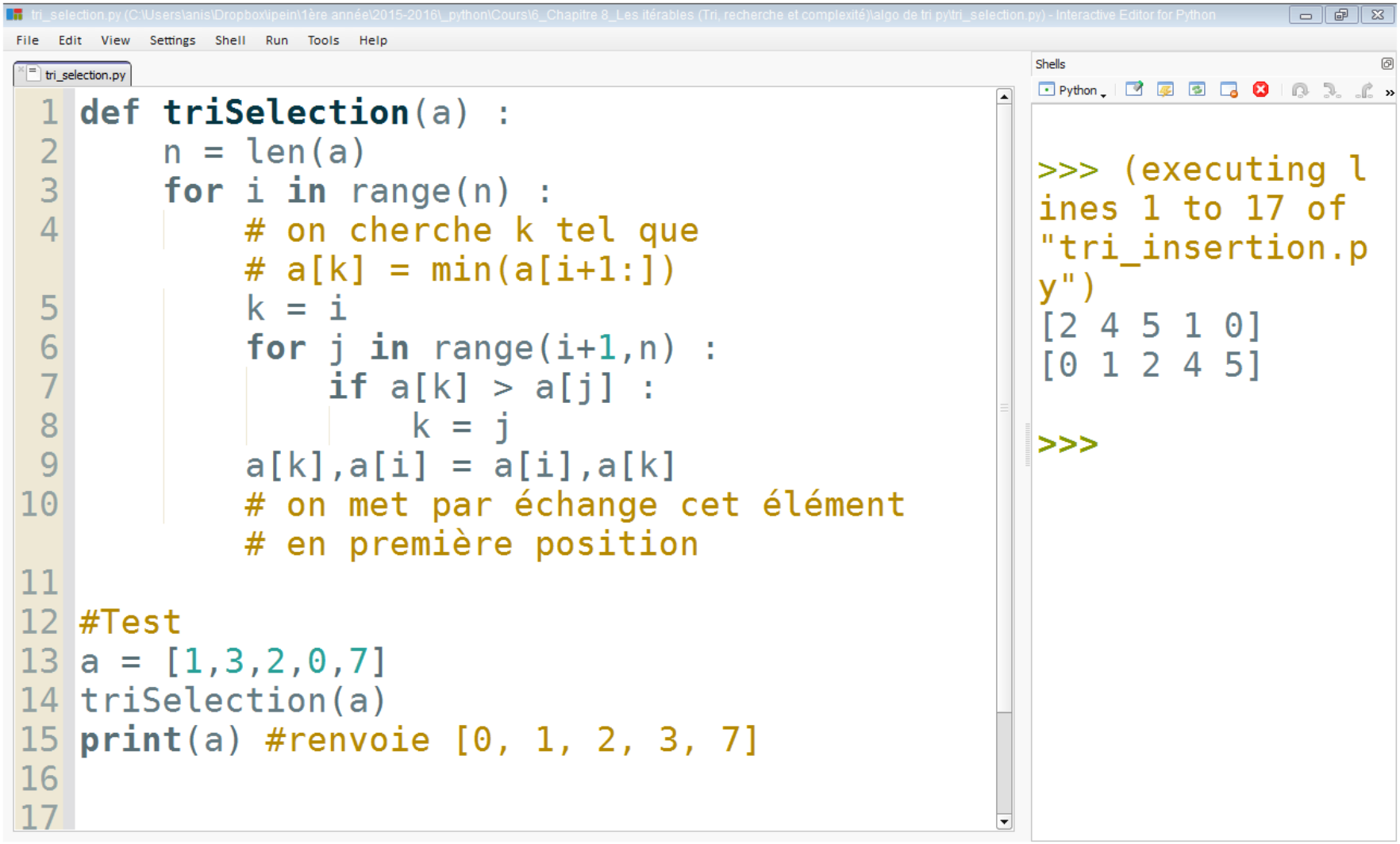
1. Tri par sélection : Exemple

Exemple :

L=[4,5,1,-6,2]

- Etape N°1 :
  Le minimum de L est -6, on le permute avec 4 : L=[**-6**,5,1,**4**,2] ;
- Etape N°2 :
  Le minimum de [5,1,4,2] est 1, on le permute avec 5 :
  L=[-6,**1**, **5**,4,2]
- Etape N°3 :
  Le minimum de [5,4,2] est 2, on le permute avec 5 :
  L=[-6,1,2,4,5]
- Etape N°4 : Le minimum de [4,5] est 4, il est à sa bonne place.

1. Tri sélection : Version itérative



1. Tri sélection : Version itérative

- Complexité

Algorithme	Unités de temps
`def triselect(L):`	`T(n)=`
`  n=len(L)`	`1 + 1`
`  for i in range(n-1):`	`n-2 boucles`
`  imin=i`	`1`
`  for j in range(i+1,n):`	`(n-i-1) boucles`
`  if L[j]<L[imin]:`	`1`
`  imin=j`	`1`
`  if i!=imin:`	`1`
`  L[i],L[imin]=L[imin],L[i]`	`2`

Le nombre de comparaison est donné par (*n-i-1*)

1. Tri sélection : Version itérative

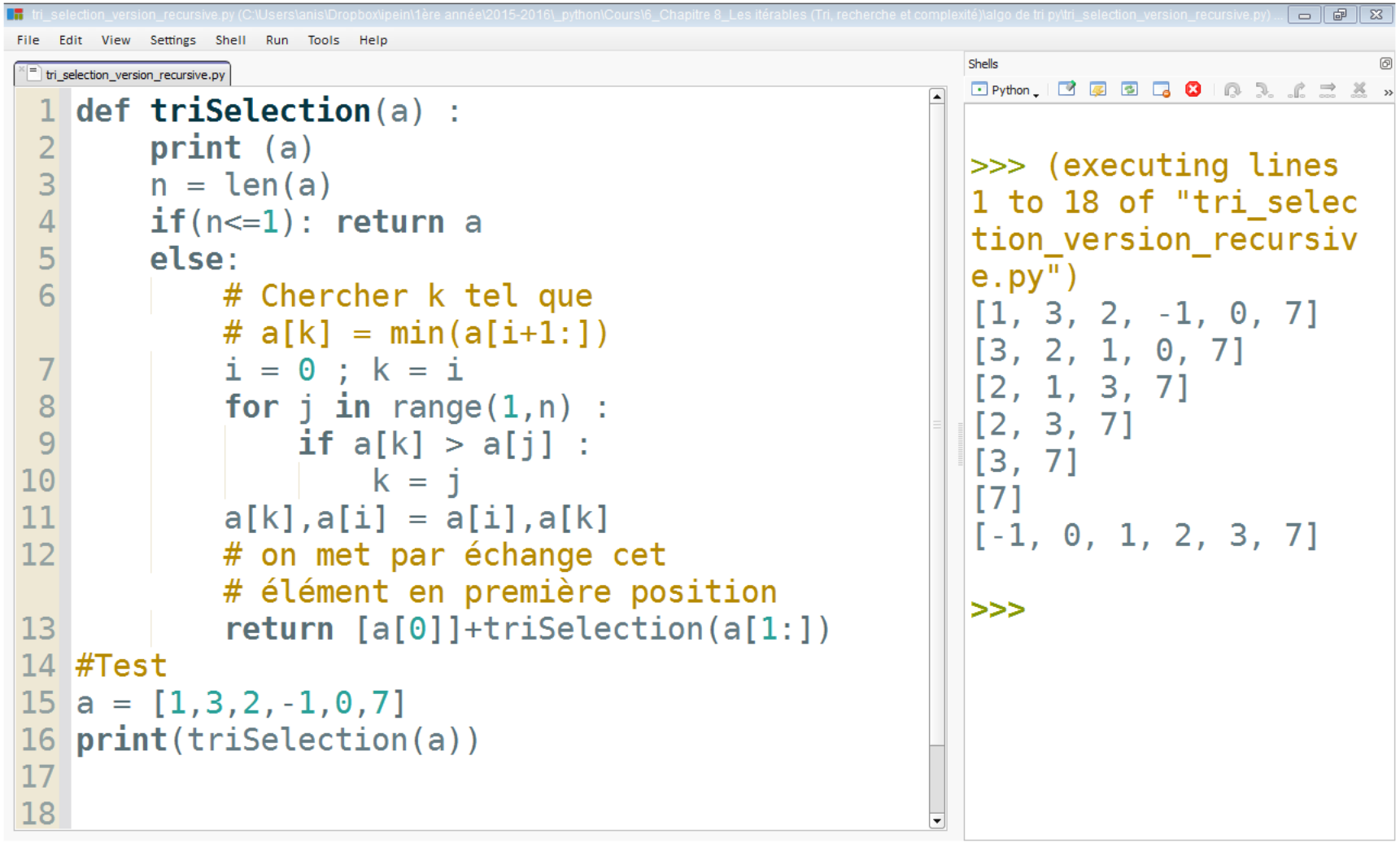
$$T(n) = (n-1) + (n-2) + \cdots + 2 + 1 = \sum_{i=0}^{n-2} (n-i-1)$$

- **T(n)** peut être vu comme la somme de **n-1** termes d'une suite arithmétique de premier terme **1** et de raison **1** :

$$T(n) = \frac{(n-1)\big(1+(n-1)\big)}{2} = \frac{(n-1)n}{2} = \frac{n^2}{2} - \frac{n}{2}$$

- O(T(n))=O(n²) **:** L'algorithme du tri est quadratique, pas très pratique quand la taille des données s'accroit.

1. Tri sélection : Version récursive

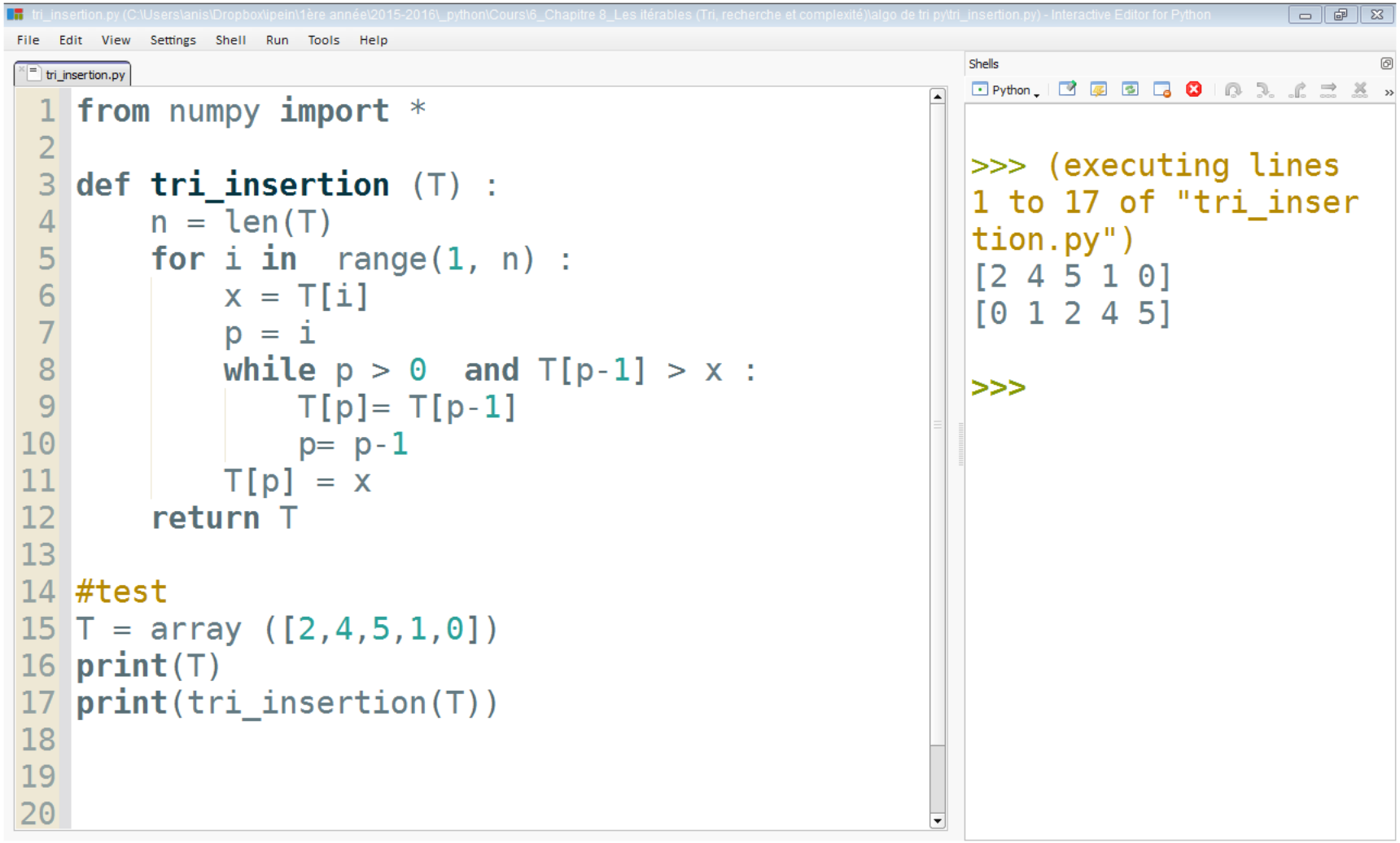


2. TRI PAR INSERTION

2. Tri par insertion : Principe *

- On considère une liste ou un tableau dont les premiers éléments d'indices 0, 1, …, i - 1 sont déjà triés.
- On cherche à placer l'élément d'indice i à sa bonne place parmi les éléments déjà triés, pour cela on procède à le comparer aux précédents jusqu'à ce que la place de T[i] soit trouvée.

2. Tri par insertion : Version itérative



2. Tri par insertion : Version itérative

Complexité dans le pire de cas :

- Dénombrons les comparaisons entre éléments du tableau
- Dans tous les cas on a n-1 passages dans la boucle for
- Dans la boucle while, le nombre d'itérations est maximal lorsque T est trié dans l'ordre décroissant, ce qui conduit à i-1 comparaisons.
- Soit en tout, le temps d'exécution est donné par :

$$T(n) = \sum_{i=1}^{n-1} i = 1 + 2 + \cdots + n - 1 = \frac{n(n-1)}{2} \approx \frac{n^2}{2}$$

- $O(T(n))=O(n^2)$

2. Tri par insertion : Version récursive

tri_insertion_version _récursive.py (C:\Users\anis\Dropbox\ipein\1ère année\2015-2016_python\Cours\6_Chapitre 8_Les itérables (Tri, recherche et complexité)\algo de tri py\tri_insertion_version _récursive.py)...

File Edit View Settings Shell Run Tools Help

tri_insertion_version _récursive.py

```
1  import numpy as np
2  def insert_elt(x,T):
3      n=len(T)
4      p=n-1
5      e=T[p]
6      while p > 0  and T[p-1] > x :
7          T[p]= T[p-1]
8          p= p-1
9      T[p] = x
10     T=np.append(T,e)
11     return T
12
13 def tri_insertion(T,i):
14     n=len(T)
15     if n==0 or i==n:
16         return T
17     else:
18         print("In/ i={0}, T={1}".format(i,T))
19         R=insert_elt(T[i],T[:i])
20         T=np.append(R,T[i+1:])
21         i+=1
22         return tri_insertion(T,i)
23
24 #Test
25 T=np.array([12,5,-12,7,6])
26 print("before :",T)
27 T=tri_insertion(T,1)
28 print("after :",T)
29
30
```

Shells

Python

```
>>> (executing lines 1 to 28 of "tri_inserti
on_version _récursive.py")
before : [ 12   5 -12   7   6]
In/ i=1, T=[ 12   5 -12   7   6]
In/ i=2, T=[  5  12 -12   7   6]
In/ i=3, T=[-12   5  12   7   6]
In/ i=4, T=[-12   5   7  12   6]
after : [-12   5   6   7  12]

>>>
```

https://sites.google.com/site/formateuranissaid/

3.
TRI RAPIDE

3. Tri rapide : Principe *

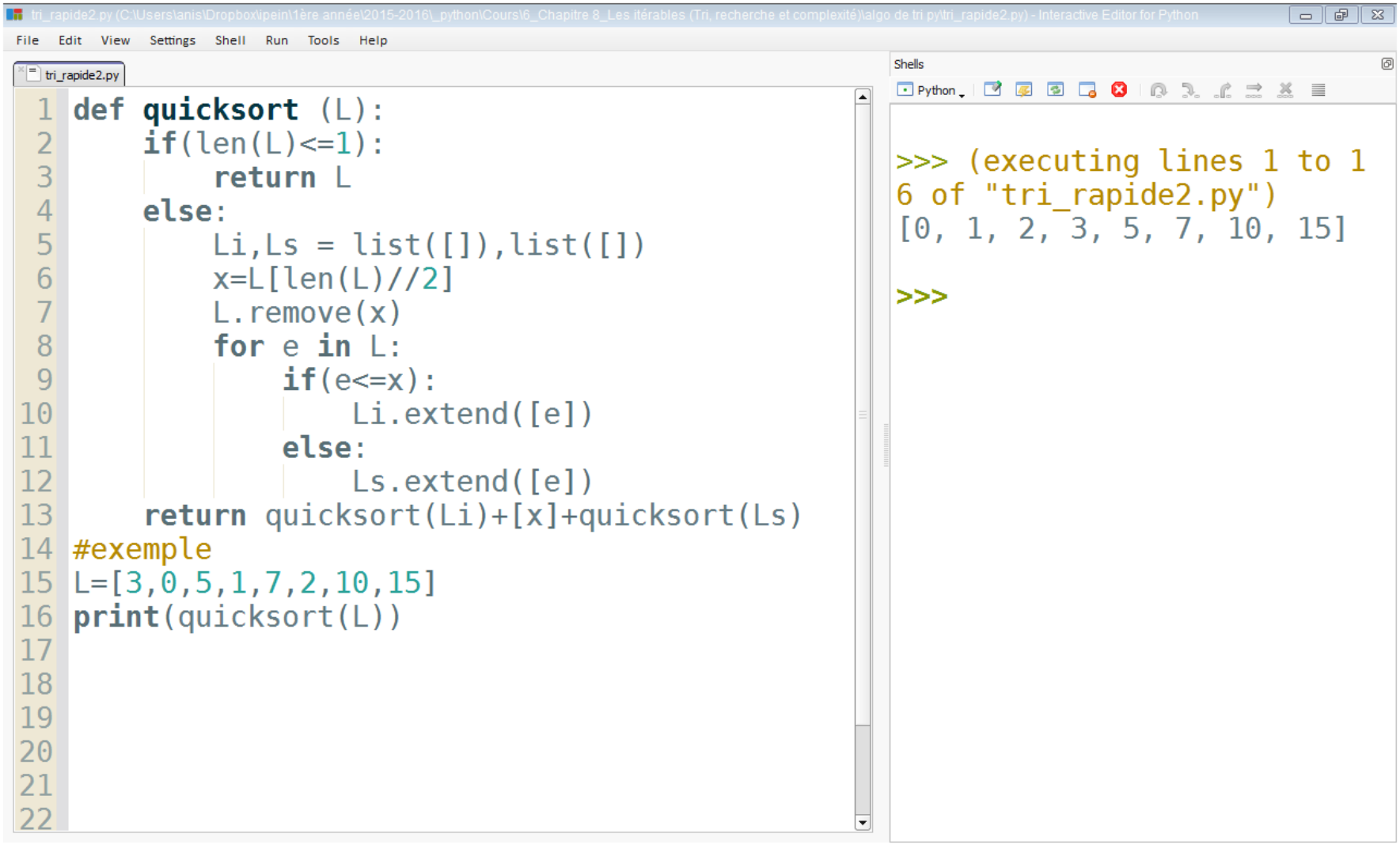
Tri rapide (quicksort) : diviser pour régner

Le principe de ce tri est le suivant :

Pour trier une liste L,

- Si L est vide ou admet un seul élément, elle est triée (c'est notre critère d'arrêt)
- Sinon :
  - On choisit un élément e ϵ L, quelconque, que l'on retire de L ;
  - On construit deux sous listes :
    - Li formée des éléments inférieurs à e
    - Ls formée des éléments plus grands que e
  - Le résultat est la concaténation de Li récursivement triée, de [e], et de Ls qui est aussi récursivement triée.

3. Tri rapide : Version récursive



3. Tri rapide : Version récursive

Complexité :

- Nous chercherons naturellement à évaluer le nombre des comparaisons entre les éléments de la liste.
- Considérons un appel principal avec une liste de longueur n.
- Afin d'identifier le pire de cas, on suppose que l'on choisisse systématiquement le plus grand élément de la liste.
- Nous aurons à chaque fois une de deux listes est vide len(Li)=0 et l'autre liste contenant len(Ls)-1 comparaisons

3. Tri rapide : Version récursive

- Les appels récursifs iront deux par deux avec une liste privé d'un élément par rapport à l'argument d'appel et une liste vide.
- L'arbre des appels sera la plus haute possible
- Les appels seront les plus couteux en nombre de comparaisons :

$$(\mathrm{n}-1)+(n-2)+\cdots+1=\frac{n(n-1)}{2!}=O(n^2)$$

$O(n^2)$: complexité au pire de cas

$O(\,n\ln n)$: complexité moyenne

3. Tri rapide : Conclusion

- **Question :**

Qu’est-ce que garantit que le programme se termine ?

- **Réponse :**

Les appels récursifs ont pour arguments des listes de longueurs strictement plus petites que len(L).

La condition d’arrêt sera remplie dans tous les cas en un temps fini (l’arbre des appels est au plus de hauteur len(L))

3. Tri rapide : Conclusion

- **Question :**

Qu'est-ce que garantit que la liste sera finalement triée ?

- **Réponse :**

Si len(L) = 0 ou len(L)=1, le résultat est clair ;

Sinon,

Supposons que la propriété est vraie pour les listes de longueurs 0,1, …, n.

Les deux appels récursifs ont comme paramètre des sous-listes de L\{e}

Elles retournent donc par hypothèse de récurrence deux copies triées de Li et Ls.

La concaténation de trois listes Li,[e] et Ls est donc triée.

Avec e (le pivot) est plus grand que les éléments de Li et plus petit que les éléments de Ls.

4.
TRI FUSION

4. Tri fusion : Principe *

Le tri fusion est défini de la façon suivante :

- Le critère d'arrêt : Le tableau a au plus un élément (il est déjà trié)
- Si le tableau a plus d'un élément, on appelle récursivement la fonction sur chacun des sous-tableaux et on interclasse leurs copies triées
- Pour rassembler deux tableaux T1 et T2 déjà triés, on peut les interclasser de la façon suivante :
  - On compare les plus petits éléments de chacun d'eux, on place le plus petit d'eux dans un nouveau tableau T.
  - On poursuit en comparent successivement les deux plus petits éléments non encore trié de ces tableaux jusqu'à l'épuisement de l'un d'eux, il ne reste plus qu'ajouter les éléments du tableau non vidé.

4. Tri fusion : Version récursive

tri_fusion1.py (C:\Users\anis\Dropbox\ipein\1ère année\2015-2016_python\Cours\6_Chapitre 8_Les itérables (Tri, recherche et complexité)\algo de tri py\tri_fusion1.py) - Interactive Editor for Python

File Edit View Settings Shell Run Tools Help

tri_fusion1.py

```
1  import numpy as np
2  def interclassement(T1,T2):
3      i=j=k=0
4      n1,n2,n=len(T1),len(T2),len(T1)+len(T2)
5      T = np.ones(n,dtype=type(T1[0]))
6      while i < n1 and j < n2:
7          if(T1[i]<=T2[j]):
8              T[k],i=T1[i],i+1
9          else:
10             T[k],j=T2[j],j+1
11         k+=1
12     if i==n1:
13         while j<n2:
14             T[k],j,k=T2[j],j+1,k+1
15     else:
16         while i<n1:
17             T[k],i,k=T1[i],i+1,k+1
18     return T
19 def tri_fusion(T):
20     print("Appel avec T=",T)
21     if(len(T)<=1): return T
22     else:
23         n= len(T) ; m=n//2
24         T1 = tri_fusion(T[0:m])
25         T2 = tri_fusion(T[m:n])
26         return interclassement(T1,T2)
27
28 #Test
29 T=np.array([2,4,0,4,7,5,1])
30 print(tri_fusion(T))
31
```

Shells

Python

```
>>> (executing lines 1 to
30 of "<tmp 5>")
Appel avec T= [2 4 0 4 7 5
1]
Appel avec T= [2 4 0]
Appel avec T= [2]
Appel avec T= [4 0]
Appel avec T= [4]
Appel avec T= [0]
Appel avec T= [4 7 5 1]
Appel avec T= [4 7]
Appel avec T= [4]
Appel avec T= [7]
Appel avec T= [5 1]
Appel avec T= [5]
Appel avec T= [1]
[0 1 2 4 4 5 7]

>>>
```

4. Tri fusion : Conclusion

Inconvénients:

- C’est n’est pas un tri en place, car il réserve un nouveau tableau de même taille que le tableau à trier.
- Un tri qui n’utilise pas de mémoire supplémentaire est un tri en place (puisque les éléments sont échangés dans le tableau ou la liste, sur place)