

Formation Python ILL

Structures de données et algorithmique

Matthieu Moy
Transparents originaux : Ahmed El Rheddane

Ensimag, Grenoble INP

octobre 2016

Sommaire

- 1 Programmation multi-fichiers : Imports
- 2 Types des expressions
- 3 Références vs. valeurs
- 4 Structures de données
- 5 Fonctions récursives
- 6 Exercices
- 7 Tris de liste
- 8 Quelques mots sur la programmation objet
- 9 Si le temps le permet ...

Sommaire

- 1 Programmation multi-fichiers : Imports
- 2 Types des expressions
- 3 Références vs. valeurs
- 4 Structures de données
- 5 Fonctions récursives
- 6 Exercices
- 7 Tris de liste
- 8 Quelques mots sur la programmation objet
- 9 Si le temps le permet ...

Programmation multi-fichiers

- Créer un fichier `bonjour.py`, y définir la fonction :

```
def dire_bonjour(nom) :  
    print "Bonjour", nom
```

- Créer un fichier `principal.py` dans le même répertoire, et écrire le programme :

```
print "Je vais dire bonjour"  
dire_bonjour("à tous")
```

⇒ Ça ne marche pas, Python ne sait pas où trouver la fonction `dire_bonjour`.

- Modifier `principal.py` :

```
import bonjour  
  
print "Je vais dire bonjour"  
bonjour.dire_bonjour("à tous")
```

Deux façons d'utiliser `import`

- Celui qu'on vient de voir :

```
import bonjour
```

```
# ``La fonction dire_bonjour du module bonjour``  
bonjour.dire_bonjour("à tous")
```

- Petit raccourci (pratique, mais pas forcément recommandable) :

```
from bonjour import *  
# Ou bien: from bonjour import dire_bonjour
```

```
# ``La fonction dire_bonjour,  
#   là où Python la trouvera``  
dire_bonjour("à tous") # Pas besoin de préciser
```

- Utilisations :

- ▶ structurer un programme
- ▶ réutiliser des bibliothèques

Petit échauffement

- Combien y a-t-il de zéros dans factorielle 2016 ?

Petit échauffement

- Combien y a-t-il de zéros dans factorielle 2016 ?
- Faut-il encore recoder la fonction factorielle ?

Petit échauffement

- Combien y a-t-il de zéros dans factorielle 2016 ?
- Faut-il encore recoder la fonction factorielle ?
 - ▶ Non, on peut la réutiliser si on l'a déjà dans un fichier `.py`, ou utiliser la bibliothèque `math`, qui contient une fonction `factorial`.
- Indice supplémentaire :
 - ▶ Utiliser la fonction `count` des chaînes de caractères

Solution : combien de zeros dans 2016 !

- Réponse :

```
>>> from math import factorial
>>> str(factorial(2016)).count("0")
1006
```

- Des entiers non-bornés !

```
>>> 2 ** 32
4294967296
>>> 2 ** 32 + 1
4294967297
>>> 2 ** 64
18446744073709551616
>>> 2 ** 64 + 1
18446744073709551617
>>> 2 ** 129
680564733841876926926749214863536422912
```

Sommaire

- 1 Programmation multi-fichiers : Imports
- 2 **Types des expressions**
- 3 Références vs. valeurs
- 4 Structures de données
- 5 Fonctions récursives
- 6 Exercices
- 7 Tris de liste
- 8 Quelques mots sur la programmation objet
- 9 Si le temps le permet ...

Types en Python

- Toute expression Python a un type :

```
>>> a = [0, 1, 1.]  
>>> type(a)  
<class 'list'>  
>>> type(a[1] + a[2])  
<class 'float'>
```

Types en Python

- Toute expression Python a un type :

```
>>> a = [0, 1, 1.]  
>>> type(a)  
<class 'list'>  
>>> type(a[1] + a[2])  
<class 'float'>
```

- Même une fonction qui ne retourne pas de valeur !

```
>>> def affichage():  
...     print "Bonjour"  
...  
>>> type(affichage())  
Bonjour  
<class 'NoneType'>
```

Sommaire

- 1 Programmation multi-fichiers : Imports
- 2 Types des expressions
- 3 Références vs. valeurs**
- 4 Structures de données
- 5 Fonctions récursives
- 6 Exercices
- 7 Tris de liste
- 8 Quelques mots sur la programmation objet
- 9 Si le temps le permet ...

Références en Python (1)

- Considérons les deux listes :

```
>>> a = [0, 1, 1]
```

```
>>> b = [0, 1, 1]
```

- a et b sont-elles égales ?

Références en Python (1)

- Considérons les deux listes :

```
>>> a = [0, 1, 1]
```

```
>>> b = [0, 1, 1]
```

- a et b sont-elles égales ?

```
>>> a == b
```

```
True
```

Références en Python (1)

- Considérons les deux listes :

```
>>> a = [0, 1, 1]
```

```
>>> b = [0, 1, 1]
```

- a et b sont-elles égales ?

```
>>> a == b
```

```
True
```

- Pourtant, a et b *réfèrent* deux *objets* différents.

```
>>> a[0] = -42
```

```
>>> b
```

```
[0, 1, 1]
```

- Pour visualiser ça :

<http://pythontutor.com/visualize.html>

Références en Python (2)

- Pour vérifier l'identité (l'égalité des références), on utilise `is`.

```
>>> a = [0, 1, 1]
```

```
>>> b = [0, 1, 1]
```

```
>>> a is b
```

```
False
```

```
>>> a = b
```

```
>>> a is b
```

```
True
```

Références en Python (2)

- Pour vérifier l'identité (l'égalité des références), on utilise `is`.

```
>>> a = [0, 1, 1]
```

```
>>> b = [0, 1, 1]
```

```
>>> a is b
```

```
False
```

```
>>> a = b
```

```
>>> a is b
```

```
True
```

- Maintenant que `a` et `b` désignent le même objet :

```
>>> a[0] = 666
```

```
>>> b
```

```
[666, 1, 1]
```

Références en Python (3)

- Considérons cet exemple :

```
>>> m = [[0]*2]*2
```

```
>>> m
```

```
[[0, 0], [0, 0]]
```

Références en Python (3)

- Considérons cet exemple :

```
>>> m = [[0]*2]*2
>>> m
[[0, 0], [0, 0]]
```

- Si, maintenant, on affectait la case (0,0) :

```
>>> m[0][0] = 1
>>> m
[[1, 0], [1, 0]]
```

- ▶ Argh ! Comment expliquer ça ?

Références en Python (3)

- Considérons cet exemple :

```
>>> m = [[0]*2]*2
>>> m
[[0, 0], [0, 0]]
```

- Si, maintenant, on affectait la case (0,0) :

```
>>> m[0][0] = 1
>>> m
[[1, 0], [1, 0]]
```

► Argh ! Comment expliquer ça ?

- Python va d'abord évaluer l'expression `[[0]*2]`, et comme c'est une liste va utiliser la référence du résultat pour l'opération suivante

Références et fonctions

- Que font les morceaux de code :

```
def ajoute_un(x) :  
    x = x + 1
```

```
a = 1  
ajoute_un(a)  
print a
```

```
def ajoute_un_l(liste) :  
    liste[0] = liste[0] + 1
```

```
li = [1]  
ajoute_un_l(li)  
print li[0]
```

- Expérimenter sur <http://pythontutor.com/> pour comprendre la différence.
- Du coup, comment faire la fonction `ajoute_un` en Python ?

Tout est référence, mais ...

- En Python, tout est référence.
- $a = b \Rightarrow a$ et b représentent le même objet (sémantique de **partage**)
- Mais on ne s'en rend pas compte si a et b sont non-mutable ! (e.g. entiers, flottants, chaînes)

Modification de référence ou modification de valeur ?

- Les deux morceaux de code suivant sont-ils équivalents ?

```
liste1 = [1, 2, 3]  
liste1.append(42)
```

```
liste2 = [1, 2, 3]  
liste2 = liste2 + [42]
```

- Utiliser l'opérateur `is` et/ou <http://pythontutor.com/> pour comprendre.

Sommaire

- 1 Programmation multi-fichiers : Imports
- 2 Types des expressions
- 3 Références vs. valeurs
- 4 Structures de données
- 5 Fonctions récursives
- 6 Exercices
- 7 Tris de liste
- 8 Quelques mots sur la programmation objet
- 9 Si le temps le permet ...

Structures de données (1)

- Les listes ([..., ...]) :

- ▶ Peuvent contenir des éléments de types différents.
- ▶ Peuvent être modifiées.

```
>>> l = [0, 1, 1., "deux"]
```

```
>>> l.append(True)
```

```
>>> l
```

```
[0, 1, 1., 'deux', True]
```

Structures de données (1)

● Les listes ([..., ...]) :

- ▶ Peuvent contenir des éléments de types différents.
- ▶ Peuvent être modifiées.

```
>>> l = [0, 1, 1., "deux"]
>>> l.append(True)
>>> l
[0, 1, 1., 'deux', True]
```

● Les tuples ((..., ...)) :

- ▶ Ne peuvent être modifiés.
- ▶ Utiles pour les affectations.

```
>>> t = (0, 1, 1)
>>> a = t[0]; b = t[1]; c = t[2]
>>> (a, b, c) = (0, 1, 1)
>>> a, b, c = 0, 1, 1
```

Structures de données (2)

- Les ensembles :

- ▶ Des listes non ordonnées.
- ▶ Ne contiennent pas de doublons.

```
>>> {"zero", 1, 1, "deux"}  
{1, 'zero', 'deux'}
```

Structures de données (2)

● Les ensembles :

- ▶ Des listes non ordonnées.
- ▶ Ne contiennent pas de doublons.

```
>>> {"zero", 1, 1, "deux"}  
{1, 'zero', 'deux'}
```

● Les dictionnaires

- ▶ Associent des valeurs à des clés.
- ▶ les clés doivent être de type non modifiable.

```
>>> d = {}  
>>> d["cle1"] = "valeur1"  
>>> d["cle2"] = "valeur2"  
>>> d["cle1"]  
'valeur1'  
>>> d  
{'cle1': 'valeur1', 'cle2': 'valeur2'}  
>>> "cle3" in d  
False
```

Sommaire

- 1 Programmation multi-fichiers : Imports
- 2 Types des expressions
- 3 Références vs. valeurs
- 4 Structures de données
- 5 Fonctions récursives**
- 6 Exercices
- 7 Tris de liste
- 8 Quelques mots sur la programmation objet
- 9 Si le temps le permet ...

Fonctions récursives

- Fonction récursive : fonction qui se rappelle elle-même.
- Attention à la condition d'arrêt !

```
def fact(n):  
    if n <= 1:  
        # condition d'arrêt  
        return 1  
    else:  
        # appel récursif  
        return n * fact(n - 1)
```

Sommaire

- 1 Programmation multi-fichiers : Imports
- 2 Types des expressions
- 3 Références vs. valeurs
- 4 Structures de données
- 5 Fonctions récursives
- 6 Exercices**
- 7 Tris de liste
- 8 Quelques mots sur la programmation objet
- 9 Si le temps le permet ...

Retour sur méthode de Héron (1/2)

- Version « mathématique » : $x_{n+1} = \frac{x_n + a/x_n}{2}$, $x_0 = a$
- Version itérative :

```
def heron1(x, n):  
    res = x  
    for i in range(n):  
        res = (res + x / res) / 2  
    return res
```

- Version récursive naïve (pourquoi est-ce si lent ?) :

```
def heron(x, n):  
    if n == 0:  
        return x  
    else:  
        return (heron(x, n - 1) +  
                x/heron(x, n - 1)) / 2  
  
print heron2(2, 23)
```

Retour sur méthode de Héron (2/2)

```
def heron3(x, n):  
    if n == 0:  
        return x  
    else:  
        # Un seul appel récursif ...  
        xn1 = heron3(x, n - 1)  
        # ... même si on utilise deux fois la valeur  
        return (xn1 + x/xn1) / 2
```

~> Ah, ça va plus vite !

Petite parenthèse : la complexité

- Combien d'opérations pour traiter une donnée de taille n ?
 - ▶ $O(n)$: on peut traiter n = plusieurs milliards
 - ▶ $O(n^2)$: on peut traiter n = plusieurs milliers
 - ▶ $O(2^n)$: lent avec $n = 30$, heures de calcul avec $n = 40$, quasi impossible d'atteindre $n = 80$.

Ex. 1 : Fibonacci itérative

- Implémentez une fonction `fib2(n)` itérative qui retourne pour chaque n , F_n tel que :
 - ▶ $F_0 = 0$.
 - ▶ $F_1 = 1$.
 - ▶ $F_n = F_{n-1} + F_{n-2}$ pour tout $n > 1$.
- Indices :
 - ▶ Traitez les cas particuliers à part.
 - ▶ Calculez les F_2, F_3, \dots, F_n successivement.

Ex. 1 : Solution

```
def fib1(n):  
    if n == 0:  
        return 0  
    if n == 1:  
        return 1  
    a = 0  
    b = 1  
    for i in range(1, n):  
        a, b = b, b + a  
  
    return b
```

- On dit que le coût de cette fonction est *linéaire* (ou $O(n)$ ¹).

1. Oui, le même qu'en maths !

Ex. 2 : Fibonacci récursive

- Implémentez maintenant la fonction `fib2(n)`, la version récursive de `fib1(n)`.
- Indice :
 - ▶ N'oubliez pas les conditions d'arrêts !
- Combien de fois appelle-t-on `fib2(0)` pour calculer `fib2(n)` ?

Ex. 2 : Solution

```
def fib2(n):  
    if n == 0:  
        return 0  
    if n == 1:  
        return 1  
    return fib2(n - 1) + fib2(n - 2)
```

- On dit que le coût de cette fonction est *exponentiel* ($O(2^n)$).

Sommaire

- 1 Programmation multi-fichiers : Imports
- 2 Types des expressions
- 3 Références vs. valeurs
- 4 Structures de données
- 5 Fonctions récursives
- 6 Exercices
- 7 Tris de liste**
- 8 Quelques mots sur la programmation objet
- 9 Si le temps le permet ...

Trier une liste

- Entrée : [1, 3, 2, 5, 4, -2]
- Sortie : [-2, 1, 2, 3, 4, 5]
- Beaucoup d'algorithmes (≈ 65 sur wikipedia)
- Premier exemple : tri bulle (peu efficace, mais simple)
 - ▶ Parcourir le tableau
 - ▶ Pour chaque paire d'éléments consécutifs dans le mauvais ordre, inverser les deux.
 - ▶ Recommencer autant de fois que nécessaire

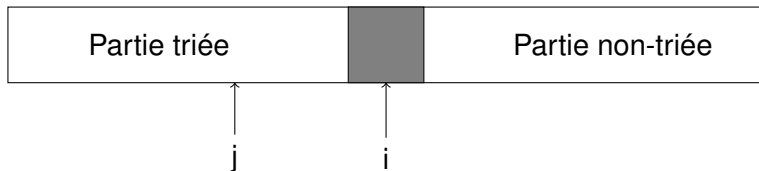
Tri Bulle : une solution (très naïve)

```
def tri_bulle(liste):  
    done = False  
    while not done:  
        done = True  
        for i in range(len(liste) - 1):  
            if liste[i] > liste[i + 1]:  
                done = False  
                liste[i], liste[i + 1] = \  
                    liste[i + 1], liste[i]  
    return liste  
  
print tri_bulle([1, 3, 2, 5, 4, -2])
```

- Combien d'opérations ?
- Comment améliorer ?

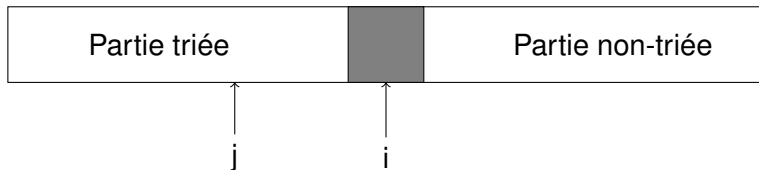
Tri par insertion

- Tri d'un paquet de cartes :
 - ▶ Partie triée en main gauche
 - ▶ Partie non-triée en main droite
 - ▶ On insère la première carte de la main droite au bon endroit
- Idem avec une liste / un tableau :



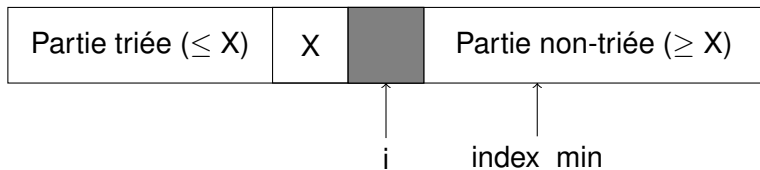
- ▶ i = indice de la valeur à insérer
- ▶ j = indice où $l[i]$ doit être inséré

Tri par insertion : corrigé



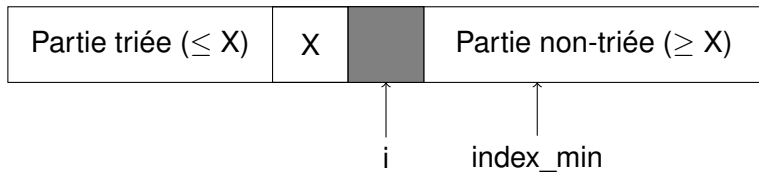
```
def insertion_sort(lst):  
    for i in range(1, len(lst)):  
        value = lst[i]  
        j = i  
        while j > 0 and lst[j - 1] > value:  
            lst[j] = lst[j - 1]  
            j = j - 1  
        lst[j] = value  
        # print(lst)
```

Tri par sélection du minimum



- Recherche du minimum dans la partie non-triée (`index_min`)
- Échange de `lst[index_min]` et `lst[i]`
- Incrément de `i` et on recommence !
- À vous !

Tri par sélection du minimum : corrigé



```
def selection_sort(lst):  
    for i in range(len(lst) - 1):  
        min = lst[i]  
        i_min = i  
        for j in range(i, len(lst)):  
            if lst[j] < min:  
                min = lst[j]  
                i_min = j  
        lst[i], lst[i_min] = lst[i_min], lst[i]
```

Sommaire

- 1 Programmation multi-fichiers : Imports
- 2 Types des expressions
- 3 Références vs. valeurs
- 4 Structures de données
- 5 Fonctions récursives
- 6 Exercices
- 7 Tris de liste
- 8 Quelques mots sur la programmation objet
- 9 Si le temps le permet ...

Besoin numéro 1 : structurer les données

- Exemple : représenter une personne \leadsto nom, prénom, âge.
Comment faire ?

Besoin numéro 1 : structurer les données

- Exemple : représenter une personne \leadsto nom, prénom, âge.
Comment faire ?
- Version 1 : avec une liste.
 - ▶ Pierre Dupont, 23 ans \leadsto ['Pierre', 'Dupont', 23]
 - ▶ Obtenir l'âge de $x \leadsto x[2]$
 - ▶ Problèmes : facile de se mélanger les pinceaux entre les indices !

Besoin numéro 1 : structurer les données

- Exemple : représenter une personne \leadsto nom, prénom, âge.
Comment faire ?
- Version 1 : avec une liste.
 - ▶ Pierre Dupont, 23 ans \leadsto ['Pierre', 'Dupont', 23]
 - ▶ Obtenir l'âge de `x` \leadsto `x[2]`
 - ▶ Problèmes : facile de se mélanger les pincesaux entre les indices !
- Version 2 : avec une liste + des fonctions
 - ▶ Exemple (cf. `person_struct.py`):

```
def build_person(first, last, age):  
    return [first, last, age]  
  
def get_first(x):  
    return x[0]
```
 - ▶ Plus lisible, moins d'erreurs

Besoin numéro 1 : structurer les données

- Exemple : représenter une personne \leadsto nom, prénom, âge.
Comment faire ?
- Version 1 : avec une liste.
 - ▶ Pierre Dupont, 23 ans \leadsto ['Pierre', 'Dupont', 23]
 - ▶ Obtenir l'âge de `x` \leadsto `x[2]`
 - ▶ Problèmes : facile de se mélanger les pincesaux entre les indices !
- Version 2 : avec une liste + des fonctions
 - ▶ Exemple (cf. `person_struct.py`):

```
def build_person(first, last, age):  
    return [first, last, age]  
  
def get_first(x):  
    return x[0]
```
 - ▶ Plus lisible, moins d'erreurs
- Version 3 : en programmation objet

Un objet : des variables mises ensemble

- 1 classe = type de données, possiblement défini par l'utilisateur (contient 1 ou plusieurs « champs »)
- 1 objet = instance de classe
- Exemple (cf. `person_object.py`) :

```
# La classe "Person"
class Person(object):
    def __init__(self, first, last, age):
        self.first = first
        self.last = last
        self.age = age

# On crée une instance de la classe (un objet)
# et on l'affecte à x :
x = Person('Pierre', 'Dupont', 23)

print x.first, x.last, "a", x.age, "ans."
```

Un objet : des variables et des fonctions

- En plus des « champs » (données), on va ajouter des méthodes (ou « fonctions membres ») pour manipuler ces données.

```
# La classe "Person"
```

```
class Person(object):  
    def __init__(self, first, last, age):  
        self.first = first  
        self.last = last  
        self.age = age  
  
    def display(self):  
        print self.first, self.last, \  
            "a", self.age, "ans."
```

```
x = Person('Pierre', 'Dupont', 23)
```

```
# Appel de la méthode "display"  
x.display()
```

Héritage : construire une classe en partant d'une autre

- Sujet complexe mais fondamental en objet (pas le temps de l'expliquer vraiment ici)
- `Teacher` **dérive de** `Person` : « `Teacher` sait faire tout ce que `Person` sait faire, plus d'autres choses ».
- Appel de méthodes : Python va choisir la méthode de la bonne classe à l'exécution (mots clés : liaison dynamique ou polymorphisme dynamique).
- Exemple : `person_hierarchie.py`

Héritage : construire une classe en partant d'une autre

- Sujet complexe mais fondamental en objet (pas le temps de l'expliquer vraiment ici)
- `Teacher` **dérive de** `Person` : « `Teacher` sait faire tout ce que `Person` sait faire, plus d'autres choses ».
- Appel de méthodes : Python va choisir la méthode de la bonne classe à l'exécution (mots clés : liaison dynamique ou polymorphisme dynamique).
- Exemple : `person_hierarchie.py`
- Résumé pour nous :
 - ▶ `f(x)` = appel de fonction `f` sur `x`
 - ▶ `x.f()` = appel de méthode `f` sur `x`
 - ▶ On a déjà rencontré les deux (exemple : `liste.append(val)`)

Sommaire

- 1 Programmation multi-fichiers : Imports
- 2 Types des expressions
- 3 Références vs. valeurs
- 4 Structures de données
- 5 Fonctions récursives
- 6 Exercices
- 7 Tris de liste
- 8 Quelques mots sur la programmation objet
- 9 Si le temps le permet ...

Ex. 3 : Récursivité améliorée

- Il se trouve que :
 - ▶ $F_{2n-1} = F_n^2 + F_{n-1}^2$.
 - ▶ $F_{2n} = (2F_{n-1} + F_n)F_n$.
- Utilisez ces formules pour implémenter `fib3(n)`, une version récursive améliorée de Fibonacci.
- Indices :
 - ▶ L'opérateur modulo en Python est `%`.
- En quoi cette version est-elle meilleure que `fib2(n)` ?

Ex. 3 : Solution

```
def fib3(n):  
    if n == 0:  
        return 0  
    if n == 1:  
        return 1  
    if n % 2 == 0:  
        a = fib3(n // 2)  
        b = fib3(n // 2 - 1)  
        return (2 * b + a) * a  
    else:  
        a = fib3((n + 1) // 2)  
        b = fib3((n + 1) // 2 - 1)  
        return a ** 2 + b ** 2
```

Ex. 4 : Fibonacci ultime !

- Au tour de la mémoïsation :
 - ▶ sauvegarder les résultats (dans un dictionnaire), afin d'éviter les calculs redondants.
- Implémentez la fonction `fib4(n, dico)`, sur la base de `fib3(n)` et du principe de mémoïsation.
- Indices :
 - ▶ Avant de retourner un résultat, pensez à l'insérer dans le dictionnaire.
 - ▶ Avant de calculer un résultat, vérifiez s'il n'est pas déjà dans le dictionnaire.
- Cette fonction est de coût *logarithmique*, coût optimal pour le calcul de Fibonacci.

Ex. 4 : Solution

```
def fib4(n, d):  
    if n in d:  
        return d[n]  
    if n <= 1:  
        d[n] = n; return n  
    if n % 2 == 0:  
        a = fib4(n // 2, d)  
        b = fib4(n // 2 - 1, d)  
        r = (2 * b + a) * a  
        d[n] = r  
        return r  
    a = fib4((n + 1) // 2, d)  
    b = fib4((n + 1) // 2 - 1, d)  
    r = a ** 2 + b ** 2  
    d[n] = r; return r
```