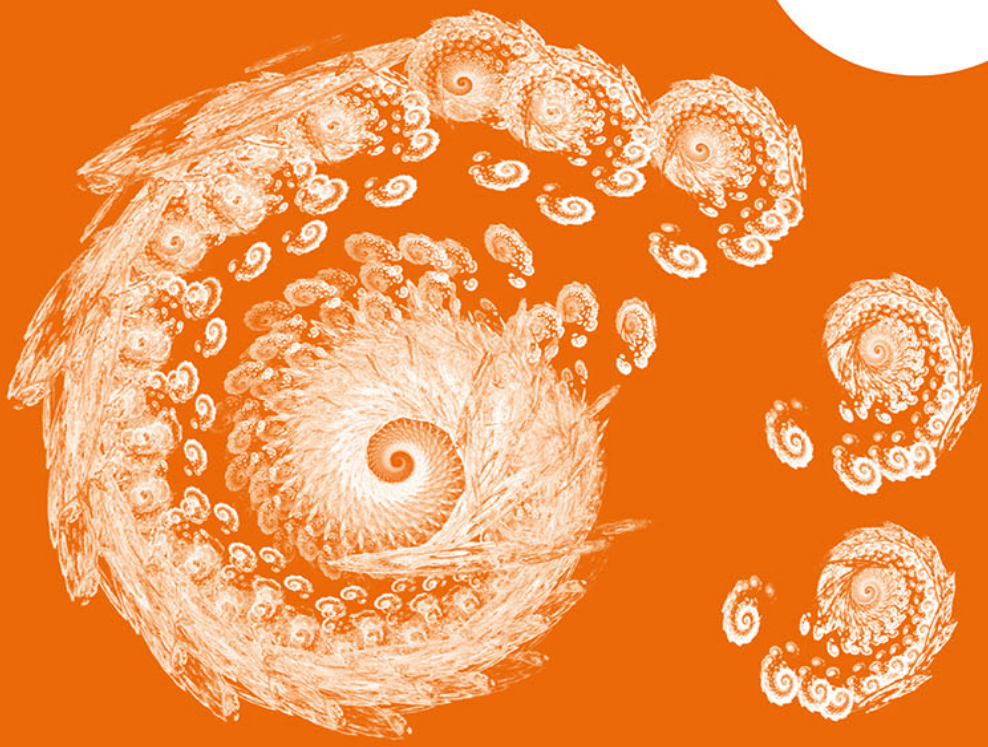
2e année **prépa scientifique**

Informatique

avec **Python**

Serge **Bays**



ellipses

Informatique avec Python

Informatique avec Python

Serge **Bays**

2e année
prépa scientifique

ellipses

Du même auteur chez le même éditeur :

Introduction à l'informatique - 1re année prépa scientifique
288 pages, 2016

ISBN 9782340-051218

32, rue Bargue 75740 Paris cedex 15

www.editions-ellipses.fr

Avant-propos

Cet ouvrage est destiné principalement aux étudiants en seconde année de classes préparatoires aux Grandes Ecoles. Les outils de base ont été étudiés en première année et les prérequis nécessaires sont tous rassemblés dans un précédent ouvrage consacré au programme d'informatique des deux premiers semestres. Le programme d'informatique du troisième semestre, commun aux différentes voies de la filière scientifique, est traité dans son intégralité. Des compléments abordent certains thèmes proposés en exercices dans le programme ou des prolongements du cours. Ils contribuent à la préparation aux concours.

Le langage de programmation utilisé est Python en version 3 avec, pour les représentations graphiques, la bibliothèque Matplotlib. Les bibliothèques NumPy et SciPy sont utilisées seulement quand elles permettent une simplification ou une amélioration du code.

Le contenu de l'ouvrage correspond en grande partie à un enseignement qui est dispensé en classe depuis la rentrée 2014.

Les trois premiers chapitres traitent le contenu théorique du programme et il est recommandé de les travailler dans l'ordre. Une bonne maîtrise de la structure de pile permet de mieux comprendre la notion de récursivité qui occupe une position centrale dans certains algorithmes de tri présentés ici.

Les chapitres suivants présentent des thèmes utilisant les différentes notions du programme ou approfondissent certaines parties. Ces thèmes ont été choisis pour différentes raisons. La notion de classe est indispensable aux personnes souhaitant aller plus loin dans le domaine de la programmation. Le tracé de courbes et l'étude d'équations différentielles, abordés dans de nombreux sujets de concours, font le lien avec les problèmes de modélisation. Une bonne maîtrise des tableaux ou des matrices est absolument nécessaire. Le codage et le chiffrement sont l'occasion de parler d'information, de sa représentation et de sa transmission. Les probabilités liées à l'informatique sont très présentes dans de nombreux domaines de recherche.

Les objectifs principaux restent la préparation des étudiants aux concours et l'ouverture vers des horizons nouveaux pouvant motiver le lecteur à la pratique de l'informatique.

De nombreux exercices progressifs et variés sont proposés. Ils illustrent en particulier le cours dans les trois premiers chapitres. Ils sont tous corrigés et les réponses sont données sous forme de programmes. Naturellement, quand il s'agit de code, plusieurs écritures sont envisageables. La résolution d'un exercice peut se faire en trois temps : d'abord écrire un programme valide, ensuite essayer de l'améliorer en termes de simplicité et de clarté, puis éventuellement le rendre plus performant en terme de complexité. Certains programmes ne sont d'aucune utilité s'ils ne sont pas optimisés.

Dans la perspective des concours, l'écriture de programmes sur papier doit être une activité régulière. Une alternative peut être d'écrire de temps en temps des programmes dans un simple éditeur de texte, afin de ne pas bénéficier des aides logicielles comme la coloration syntaxique, l'indentation automatique et bien sûr la détection d'erreurs. Une pratique assidue sur machine est absolument nécessaire pour acquérir de bons réflexes.

L'environnement de développement utilisé pour cet ouvrage est "Idle", présent dans la distribution standard, qui rassemble les tâches courantes d'un éditeur de texte, l'interprétation d'un fichier source et l'exécution d'un programme. C'est un environnement simple, léger et amplement suffisant.

Si l'installation des bibliothèques Matplotlib, NumPy ou SciPy pose problème, c'est souvent lié aux versions utilisées. Ceci se règle généralement en changeant de version. Une distribution "portable" comme WinPython s'installe sans difficulté sur une machine ou sur une clé USB et contient tout le nécessaire.

Je remercie mes collègues enseignants pour les discussions animées en salle des professeurs, la direction du lycée pour sa confiance, et tous les élèves pour le temps passé ensemble. Leur réussite est toujours l'une de mes principales sources de motivation.

Merci à Alain, Daniel, Eric, Fabrice, Michel pour leurs conseils avisés, chacun dans sa spécialité. Merci à René pour la force de ses convictions.

Je tiens à remercier l'UPS qui gère la liste de discussion UPS-info, vivante, réactive et stimulante. Grâce aux collègues intervenant sur cette liste, la variété des questions et la clarté des réponses constituent une source continue d'idées et de réflexions.

Je remercie particulièrement Marie Virat, collègue de mathématiques, pour ses propositions de corrections et d'améliorations, ses conseils, son soutien, toujours avec le sourire !

Si le texte contient encore des erreurs malgré toutes les vérifications, j'en suis désolé. J'espère qu'elles ne perturberont pas trop la lecture et la compréhension.

Merci à mon épouse et mes deux filles pour leur présence.

Table des matières

Chapitre 1

Notion de pile

1.1 Introduction

Nous utilisons en Python des objets élémentaires de type **int**, **float**, **str**, **bool**, ou plus complexes comme les types **list** ou **tuple**. La construction et l'utilisation de ces objets, avec des propriétés, des opérations, des méthodes, des fonctions, sont déjà prévues par le langage.

Mais nous pouvons aussi construire et utiliser nos propres objets ou structures de données. Cela signifie dans ce cas que nous devons implémenter de manière explicite un ensemble d'objets avec toutes les fonctions utiles ou nécessaires, en particulier des opérations de construction, d'accès et de modification.

Dans ce chapitre, nous allons étudier les **piles**, ("stack" en anglais), qui seront construites en utilisant les objets de type **list**. Une liste a un début et une fin et il est possible d'accéder à chacun de ses éléments par l'intermédiaire d'un index. Les piles peuvent être considérées comme des listes avec des conditions d'utilisation restreintes : il est possible d'insérer ou de supprimer un élément uniquement à la fin, (en anglais "LIFO", acronyme de "Last In First Out" : dernier entré, premier sorti).

En modifiant la condition d'insertion et de suppression, nous pouvons obtenir une **file** ("queue" en anglais). C'est encore un objet qui peut être considéré comme la restriction d'une liste. Une file autorise l'insertion d'un côté et la suppression de l'autre (en anglais "FIFO", acronyme de "First In First Out" : premier entré, premier sorti). Ceci correspond par exemple à la file d'attente à une caisse.

Il est important d'avoir toujours en tête l'image d'une pile concrète pour bien comprendre ce qu'il est possible de faire. Si nous disposons d'assiettes que nous pouvons manier seulement une par une, nous commençons par choisir un endroit où les poser : c'est la création d'une **pile vide**. Ensuite, nous posons une première assiette à l'endroit choisi, puis une deuxième sur la première et ainsi de suite. Nous empilons les assiettes une par une en les posant sur la pile : ce sera le rôle de la fonction **empiler**. Nous pouvons les reprendre dans l'ordre inverse en commençant

par la dernière ajoutée, toujours une par une : ce sera le rôle de la fonction **dépiler**. Si les assiettes sont empilées dans un placard, la **capacité** est limitée par la hauteur du placard. Si nous sommes en extérieur, il n'y a pas de limites, à part la taille de l'échelle dont nous aurons besoin pour empiler ou dépiler et un problème de stabilité ! Enfin, nous pouvons constater si une pile est vide ou pas et nous pouvons compter le nombre d'assiettes empilées pour obtenir la **taille** de la pile.

Cette structure de pile est utilisée par la plupart des microprocesseurs : la pile correspond à une zone de la mémoire, et le processeur retient l'adresse du dernier élément. Nous la rencontrerons également dans le chapitre sur la récursivité.

La structure de file est elle aussi utilisée dans un ordinateur pour mémoriser temporairement une suite d'actions en attente qui seront traitées suivant l'ordre d'arrivée des demandes. C'est le cas, par exemple, avec les mémoires tampons ("buffers" en anglais), les serveurs d'impression et les moteurs multitâches d'un système d'exploitation.

1.2 Rappels et compléments sur les listes

Les objets de type **list**, que nous appelons des listes, sont étudiés en première année. Ils sont très souvent utilisés et il est bon de faire quelques rappels.

1.2.1 Définition

Une liste est un ensemble ordonné d'éléments éventuellement hétérogènes dont les valeurs peuvent être modifiées.

Les éléments d'une liste sont séparés par des virgules et entourés de crochets.

1.2.2 Création d'une liste

```
liste1=[]  # une liste vide
# ou liste1=list()
liste2=['a']  # une liste contenant un unique élément
liste3=['a','bonjour',17]
liste4=['d','e']
liste3[1]='b'  # modification d'un élément
print(liste3) # affiche ['a','b',17]
liste5=liste3+liste4
print(liste5) # affiche ['a','b',17,'d','e']
liste6=3*liste4 # soit ['d','e','d','e','d','e']
```

La fonction **list()** permet de convertir certains objets en listes. Nous pouvons aussi l'utiliser avec la fonction **range** pour initialiser une liste d'entiers.

```
liste=list('bonjour') # ['b','o','n','j','o','u','r']
liste1=list(range(4)) # [0,1,2,3]
liste2=list(range(1,4)) # [1,2,3]
liste3=list(range(2,14,3)) # [2,5,8,11]
```

Construction par compréhension

```
liste=[2*x-1 for x in range(1,5)] # construit [1,3,5,7]
```

Nous pouvons construire une autre liste en itérant sur les éléments d'une liste.

```
liste2=[2*x for x in liste] # construit [2,6,10,14]
```

Copie d'une liste

Pour créer une nouvelle liste, copie d'une liste existante, le code est le suivant :

```
liste1=[0,2,4,6,8]
liste2=list(liste1)
# ou liste2=liste1[:]
```

Attention : ce code peut poser problème si les éléments de la liste sont eux-mêmes des listes.

```
liste1=[[0,1],[2,3],[4,5]]
liste2=list(liste1)
liste2[1][0]=8 # modifie aussi liste1
print(liste1) # affiche [[0,1],[8,3],[4,5]]
```

Pour éviter cela, nous devons plutôt écrire :

```
liste1=[[0,1],[2,3],[4,5]]
liste2=[list(x) for x in liste1]
```

Le module **copy** propose la fonction **deepcopy** pour effectuer une vraie copie en profondeur (voir chapitre 10).

```
from copy import deepcopy
liste1=[[0,1],[2,3],[4,5]]
liste2=deepcopy(liste1)
```

Insertion et extraction

La méthode **append** permet d'ajouter un élément à la fin d'une liste.

```
liste=['a','b']
liste.append('c')  # (liste=['a','b','c'])
```

La méthode **insert** permet d'insérer un objet dans une liste.

```
liste=['a','b','d','e']
liste.insert(2,'c') # l'élément 'c' est inséré à l'index 2
print(liste) # affiche ['a','b','c','d','e'],
# 'd' et 'e' sont décalés vers la droite
```

La syntaxe pour extraire une sous-liste est la suivante :

```
liste=['a','b','c','d','e','f']
liste2=liste[1:4] # liste2 est la liste ['b','c','d']
# liste[-1] est le dernier élément soit 'f'
```

Suppression d'un élément

Pour supprimer et récupérer un élément d'une liste, nous utilisons la méthode **pop** qui supprime l'élément dont l'indice est passé en paramètre et le renvoie.

```
liste=['a','b','c','d']
x=liste.pop(2)  # l'élément d'indice 2 est affecté dans x
#  et il est supprimé de la liste
print(liste) # affiche ['a','b','d']
print(x) # affiche 'c'
```

La valeur par défaut du paramètre, (l'indice), est -1. Donc à l'exécution de l'instruction `liste.pop()`, l'élément en fin de liste est supprimé et renvoyé.

La méthode **remove** permet de supprimer un élément de valeur donnée.

```
liste=['a','b','c','d','c','e']
liste.remove('c')  # l'élément 'c' est supprimé
# seul le premier rencontré !
print(liste) # affiche ['a','b','d','c','e'],
# 'd', 'c' et 'e' sont décalés vers la gauche
```

1.3 Création et manipulation d'une pile

Nous allons écrire des fonctions dont les plus importantes ont déjà été décrites dans l'introduction.

Trois fonctions sont absolument nécessaires.

- Une fonction "créer une pile" qui crée une pile vide et éventuellement lui alloue une certaine capacité.
- Une fonction "empiler" qui ajoute un élément sur la pile ; le terme anglais est "push". Dans le cas d'une pile à capacité finie, nous devons vérifier avant que la pile n'est pas pleine.
- Une fonction "dépiler" qui enlève le "sommet" de la pile et le renvoie ; le terme anglais est "pop". Nous devons vérifier ici que la pile n'est pas vide.

D'autres fonctions peuvent nous être utiles, en voici quelques-unes parmi les plus courantes.

- Une fonction "pile vide ?" qui renvoie vrai si la pile est vide et faux sinon.
- Une fonction "taille de la pile" qui renvoie le nombre d'éléments stockés dans la pile.
- Une fonction "sommet de la pile" qui renvoie le sommet de la pile sans le dépiler ; le terme anglais est "peek".
- Une fonction "vider la pile" qui permet de dépiler tous les éléments ; le terme anglais est "clear".

L'écriture de ces fonctions dépend du mode de représentation choisi pour une pile et nous allons voir qu'il existe plusieurs possibilités avec pour chacune des avantages et des inconvénients.

La construction d'une véritable structure de donnée, une classe, n'est pas au programme mais est néanmoins présentée au chapitre 5. Nous allons simplement utiliser ici des objets connus avec certaines restrictions.

Parmi les types d'objets connus, le seul pouvant convenir est le type **list**. En pratique, une pile a une capacité finie (quand elle est pleine, la tentative d'empiler

un élément renvoie une erreur) et plusieurs possibilités s'offrent à nous pour créer et manipuler une pile de capacité c.

Voici quatre exemples de représentations d'une pile de capacité cinq contenant trois éléments.

- Une liste de deux éléments dont le premier est la capacité et le second une liste contenant dans l'ordre les éléments empilés : p = [5, ['A', 'B', 'C']].
- Une liste dont le premier élément est la capacité, contenant ensuite dans l'ordre les éléments empilés : p = [5, 'A', 'B', 'C'].
- Une liste contenant dans l'ordre les éléments empilés et dont la longueur est la capacité : p = ['A', 'B', 'C', None, None].
- une liste dont le premier élément est la taille de la pile, contenant ensuite dans l'ordre les éléments empilés et de longueur c+1, où c est la capacité de la pile : p = [3, 'A', 'B', 'C', None, None].

Nous avons choisi pour la suite la dernière représentation mais c'est un bon exercice d'écrire des fonctions **creer_pile**, **empiler** et **depiler** correspondant aux autres représentations et voir ainsi les avantages et inconvénients de chacune.

1.3.1 Pile à capacité finie

Nous créons une liste p, dont la longueur est la capacité plus un. L'élément p[0] représente la taille (le nombre d'éléments) de la pile.

La fonction **creer_pile** prend en paramètre la capacité de la pile et retourne une pile vide.

```
def creer_pile(c): # crée une pile de capacité c
    p=(c+1)*[None] # la pile est vide
    p[0]=0 # le nombre d'éléments de la pile
    return p
```

Attention à ne pas confondre la capacité qui est le nombre d'éléments que peut contenir une pile et la taille qui est le nombre d'éléments effectivement contenus dans une pile.

Définissons maintenant les fonctions **empiler** et **dépiler** :

```
def empiler(p,x): # ajout de l'élément x sur la pile
    assert p[0]<len(p)-1 # la pile n'est pas pleine ?
    p[0]=p[0]+1 # la taille augmente d'une unité
    p[p[0]]=x # le nouveau sommet est placé sur la pile
```

L'un des intérêts de la représentation choisie est que p[0] est à la fois la taille de la pile et l'index du sommet.

```
def depiler(p):
    assert p[0]>0 # la pile n'est pas vide ?
    x=p[p[0]] # le sommet de la pile
    p[p[0]]=None # le sommet est retiré de la pile
    p[0]=p[0]-1 # la taille diminue d'une unité
    return x
```

Voici quelques fonctions utiles dont les définitions sont rendues très simples grâce à la représentation utilisée :

```
def pile_vide(p):
    return p[0]==0 # renvoie True si p[0]==0 sinon False

def taille(p):
    return p[0]

def sommet(p):
    assert p[0]>0
    return p[p[0]]
```

Remarque : il est important de bien comprendre que même si nous utilisons des listes pour lesquelles nous disposons de nombreux moyens d'actions (fonctions ou méthodes), nous manipulons une pile uniquement avec les fonctions spécifiques que nous créons, indépendamment de la représentation choisie.

Par exemple, nous nous interdisons d'écrire `pile[i]` pour obtenir un élément de la pile, même si cette instruction ne renvoie évidemment aucune erreur !

1.3.2 Pile à capacité non bornée

En théorie, nous pouvons utiliser des piles dont la capacité n'est pas bornée. Une pile est alors représentée par une liste formée des éléments de la pile.

Pour définir nos fonctions spécifiques aux piles, nous utilisons donc ici les objets de type **list** avec les méthodes **append** et **pop** et la fonction **len**.

Remarque : il n'y a pas d'instruction `return` pour la fonction **empiler**.

```
def creer_pile():
    return []
```

```
def empiler(p,x):
    p.append(x)

def depiler(p):
    assert len(p)>0
    return p.pop()

def pile_vide(p):
    return p==[]

def taille(p):
    return len(p)

def sommet(p):
    assert len(p)>0
    return p[-1]
```

1.4 Applications

1.4.1 Des applications courantes

Dans un navigateur web, une pile peut servir à mémoriser les pages visitées. L'adresse de chaque nouvelle page visitée est empilée et en cliquant sur un bouton "Afficher la page précédente", l'utilisateur dépile l'adresse de la page précédente.

Dans un logiciel de traitement de texte, les modifications apportées au texte sont mémorisées dans une pile, et une fonction "annuler la frappe" ("undo" en anglais) permet de revenir en arrière pas à pas.

L'évaluation des expressions mathématiques en notation postfixée, ou notation polonaise inverse, utilise une pile.

1.4.2 La notation polonaise inverse

La notation polonaise inverse, (NPI), est une manière d'écrire les formules arithmétiques sans utiliser de parenthèses ; pour cela les opérandes sont présentés avant les opérateurs.

Par exemple, l'expression " 3 + 4 " s'écrit en NPI : " 3 4 + " , et l'expression " $7 \times (12 + 3)$ " peut s'écrire sous la forme " 7 12 3 + $\times$ ".

La réalisation d'une calculatrice NPI est basée sur l'utilisation d'une pile : à la lecture de l'expression, seuls les opérandes sont empilés un à un, ainsi que les résultats des calculs intermédiaires qui sont retournés en haut de la pile.

Pour l'expression " 3 + 4 " qui s'écrit en NPI " 3 4 + ", on empile les opérandes 3 puis 4. Ensuite l'opérateur " + " apparaît, donc on dépile 4 puis 3, on les ajoute et on empile le résultat 7.

Pour l'expression " $7 \times (12 + 3)$ " qui peut s'écrire en NPI " $7\ 12\ 3\ +\ \times$ ", on empile 7 puis 12 puis 3. L'opérateur " + " apparaît, donc on dépile 3 puis 12, on les ajoute et le résultat 15 est placé sur la pile. La pile contient alors 7 et 15. Ensuite, l'opérateur " $\times$ " apparaît. Donc on dépile 15 puis 7, on les multiplie et on empile le résultat 105. La pile contient finalement l'élément 105.

1.5 Exercices corrigés

1.5.1 Les listes

Exercice 1.5.1
Deviner la valeur de la variable a définie par l'instruction qui suit.

```
a=[8,5,4,9,5,6,8,3,1,7,6,5][2:7][3]
```

Corrigé
Nous pouvons décomposer le code en trois instructions.
- c = [8, 5, 4, 9, 5, 6, 8, 3, 1, 7, 6, 5].
- b = c[2 :7] ; b est un extrait de c et b = [4, 9, 5, 6, 8].
- a = b[3], donc la valeur de a est le nombre entier 6.

Exercice 1.5.2
Deviner ce que va afficher le code suivant.

```
a=[]
for i in range(1,10):
    if i%2==0:
        a.append(i//2)
    else:
        a.append(2*i)
print(a)
```

Corrigé
Le nombre i décrit les valeurs entières de 1 à 9. Si i est pair il est divisé par deux et ajouté à la fin de la liste a. S'il est impair, c'est son double qui est ajouté. Nous obtenons donc la liste a = [2, 1, 6, 2, 10, 3, 14, 4, 18].

Exercice 1.5.3
Ecrire un programme qui permet de constituer une liste de nombres entiers, les nombres étant entrés au clavier par un utilisateur.

Corrigé

Il y a de nombreuses solutions et quatre possibilités sont présentées ci-dessous.

```
# avec une boucle while
liste=[]
test=True
while test:
    n=input("Entrer un entier ou taper f pour terminer: ")
    if n=='f':
        test=False
    else:
        liste.append(int(n))

# avec une boucle for
liste=[]
n=int(input("Combien d'entiers ? "))
for i in range(n):
    p=int(input("Entrer un entier: "))
    liste.append(p)

# avec une liste définie en compréhension
n=int(input("Combien d'entiers ? "))
liste=[int(input("Entrer un entier: ")) for i in range(n)]

# avec une chaîne de caractères
chaine=input("Entrer les nombres séparés par une virgule")
liste=chaine.split(",") # renvoie une liste de str
liste=[int(c) for c in liste]
```

Complément : le langage Python dispose d'une fonction **eval**.

Tester les instructions `a=eval('[3,2]')` et `a=eval('3,2')`.

Dans le premier cas a est bien la liste [3, 2], donc l'utilisateur pourrait entrer toute sa liste de nombres en une seule fois à condition de bien écrire les crochets au début et à la fin et de séparer les nombres par une virgule. S'il oublie les crochets, le résultat obtenu est de type tuple, il suffit alors de le convertir en type list avec : `a=list(eval('3,2'))`.

Exercice 1.5.4

Ecrire une fonction **mini** qui prend en argument une liste de mots et renvoie le mot le plus court avec son indice dans la liste.

Si des mots ont le même nombre de lettres la fonction renvoie le premier de ces mots rencontrés dans la liste.

Tester la fonction avec la liste : m = ['cinq', 'six', 'sept', 'huit', 'neuf', 'dix'].

Comment modifier simplement la fonction pour qu'elle renvoie le dernier des mots rencontrés dans la liste si des mots ont le même nombre de lettres ?

Dans le premier cas la fonction doit renvoyer le mot 'six' avec l'indice 1, dans le second cas le mot 'dix' avec l'indice 5.

Corrigé

```
def mini(liste):
    nblettres=len(liste[0])
    ind=0
    for i in range(len(liste)):
        k=len(liste[i])
        if k<nblettres:
            nblettres,ind=k,i
    return liste[ind],ind

# test
m=['cinq', 'six', 'sept', 'huit', 'neuf', 'dix']
print(mini(m))
```

Pour le deuxième cas, il suffit de remplacer dans le test l'inégalité stricte par une inégalité large : `k<=nblettres`.

Exercice 1.5.5

1. Que vaut l'expression `[1,2]+[3,4]` ?
2. Que valent les expressions `2*['a','b']` et `2*[1,2]` ?
3. Ecrire une fonction **multiplication** qui prend en paramètres un nombre et une liste de nombres et renvoie une nouvelle liste obtenue en multipliant chaque élément de la liste par le nombre.
4. (a) Ecrire une fonction **somme1** qui prend en paramètres deux listes de nombres de même longueur et qui renvoie une nouvelle liste obtenue en ajoutant terme à terme les nombres des deux listes.
   (b) Ecrire une fonction **somme2** qui traite le cas où les deux listes de nombres sont de longueurs quelconques. Si une liste est plus courte que l'autre, les éléments manquants sont considérés égaux à 0.

Corrigé

1. L'expression vaut `[1,2,3,4]`. C'est la concaténation de listes.

2. La première expression vaut `['a','b','a','b']`.
La seconde expression vaut `[1,2,1,2]`.

3. Il y a bien entendu plusieurs possibilités pour écrire cette fonction.

```
def multiplication(n,liste):
    return [n*x for x in liste]

# ou bien
def multiplication(n,liste):
    return [n*liste[i] for i in range(len(liste))]

# ou bien
def multiplication(n,liste):
    prod=[]
    for i in range(len(liste)):
        prod.append(n*liste[i])
    return prod
```

4. (a) Les longueurs des deux listes sont supposées égales.

```
def somme1(list1,list2):
    return [list1[i]+list2[i] for i in range(len(list1))]
```

4. (b) Il est nécessaire ici de déterminer la longueur de la liste la plus courte.

```
def somme2(list1,list2):
    n1,n2=len(list1),len(list2)
    mini=min(n1,n2)
    som=[list1[i]+list2[i] for i in range(mini)]
    if n1>mini:
        som+=list1[mini:n1]
    else:
        som+=list2[mini:n2]
    return som
```

Remarque : si n1 = n2 = mini, l'instruction `list2[mini:n2]` renvoie la liste vide.

Le code précédent peut être simplifié de la manière qui suit.

```
def somme2(list1,list2):
    n1,n2=len(list1),len(list2)
```

Ce livre est principalement destiné aux étudiants en deuxième année de classes préparatoires scientifiques. Il traite entièrement le programme d'informatique de ces classes et le complète par des thèmes préparant aux différents concours des Grandes Écoles. Il fait suite à un précédent ouvrage centré sur le programme de première année.

Le contenu théorique du programme est traité dans les trois premiers chapitres qui constituent un bon tiers de ce livre. Il est recommandé de les travailler dans l'ordre : pour commencer, la notion de pile, puis la récursivité et enfin quelques algorithmes de tris.

Les chapitres suivants, en grande partie indépendants, présentent des sujets qu'il semblait important de placer dans ce cadre. Le concept de classe est abordé.

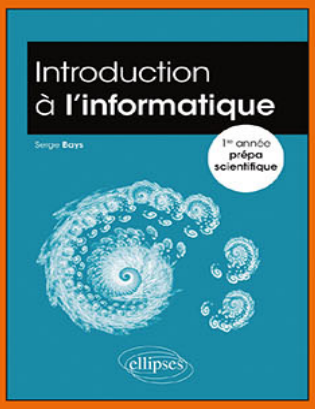
Des développements sur les courbes, les matrices, les équations différentielles et les probabilités, sont exposés. Certains points font le lien avec le programme de mathématiques. Quelques méthodes utilisées pour la transmission de données sont présentées. Ces chapitres se concluent par de nombreux exercices corrigés.

Un complément apporte des précisions sur la gestion des erreurs, l'utilisation des listes et la portée des variables.

Cet ouvrage pourra aussi intéresser les étudiants à l'université ou en écoles d'ingénieurs, ainsi que toute personne déjà initiée à la programmation en Python.

Serge Bays est agrégé de mathématiques et enseigne l'informatique en classes préparatoires au lycée Les Eucalyptus à Nice.

Du même auteur :



9 782340 019461

www.editions-ellipses.fr