

V – Les conteneurs

1/ Généralités

Définition Les *conteneurs* sont des collections d'objets quelconques.
Ils sont itérables et parfaitement adaptés au parcours via une boucle `for`.

Nous allons étudier les listes, les t-uplets et les chaînes de caractères (conteneurs ordonnés ou **séquences**) ainsi que les ensembles et les dictionnaires (conteneurs non ordonnés).

Nous ne verrons que les fonctions et méthodes essentielles sur ces différents types d'objets. Bien entendu, Python dispose de plein de fonctions sophistiquées (recherche de maximum, insertion d'élément, etc...) mais leur usage est en général prohibé au concours : il faut donc être capable de les programmer tout seul. C'est de surcroît très instructif.

Opérations sur les conteneurs

Voici deux fonctions communes à tous les types de conteneurs.

<code>len(c)</code>	renvoie le nombre d'éléments dans le conteneur <code>c</code> .
<code>x in c</code>	teste d'appartenance d'un élément <code>x</code> au conteneur <code>c</code> .

Exemples : `len("Quatre")` donne 6 ; `"o" in "Eau"` donne `False`.

2/ Les séquences

2.1) Généralités

Définition Une *séquence* est un conteneur ordonné. Les positions de ses éléments sont repérées à l'aide d'*indices entiers*, qui partent de 0 : on parle alors de *conteneur indexable*.

Exemples : `[7, "oui", True, 2]` ; `("non ?", 45, False, 4)` ; `"Au travail !"`.

Opérations sur les séquences

Ces opérations de bases fournissent de nouvelles séquences.

<code>+, *</code>	opérations de concaténation (ou mise bout à bout).
<code>reversed(s)</code>	dans une boucle, permet de parcourir la séquence <code>s</code> à l'envers.

Exemples : `"jam"+"bon"` donne `"jambon"` ; `2*"cou"` donne `"coucou"`.

Manipulations de séquences

On accède aux éléments d'une séquence `s` en précisant leurs indices entre crochets.

<code>s[0], s[1], ...</code>	premier élément, deuxième élément, ... de <code>s</code> .
<code>s[-1], s[-2], ...</code>	dernier élément, avant-dernier élément, ... de <code>s</code> .
<code>s[a:b]</code>	sous-séquence allant de l'indice a inclus à l'indice b exclu .
<code>s[a:] / s[:b]</code>	même chose en allant jusqu'à la fin / en partant du début.
<code>s[a:b:pas]</code>	parcourt les indices avec un pas (éventuellement négatif).

Exemples :

- ❶ Prenons `s=[7, "a", 12, 4, True]`. On a alors `s[4]=True` et `s[1:3]=["a", 12]`.
On peut noter que `s[5]` renvoie une erreur, alors que `s[5:8]` fonctionne bien.
- ❷ L'instruction `s[:]` fournit une copie de `s`. Avec `s[::2]`, on ne prend que les termes d'indices pairs et avec `s[::-1]`, on parcourt `s` à l'envers.

2.2) Les listes

Définition

Une **liste** est une collection, délimitée par des crochets, d'éléments séparés par des virgules. C'est un objet **mutable** : on peut modifier ses éléments, en supprimer ou bien en rajouter. Enfin, c'est une séquence du type `list`.

Exemples : la liste `l=[7,4,"oh!",78,"et alors ?"]` ; la liste vide `l=[]`.

On peut modifier les éléments existants à l'aide d'affectations, comme pour des variables. Notons que la modification d'une sous-liste permet d'étirer ou de contracter la liste. Par contre, il n'est pas possible d'ajouter des éléments à une liste par simple affectation.

Exemple : on dispose de la liste `l=[2,3,5,7,11,13]`. Tester séparément les instructions `l[2]=8`, `l[3:]=[10]`, `l[3:5]=[6,7,8,9]`, `l[4:4]=[6,8,10]`, `l[6]=17` et `l[6:]=[17]`.

Opérations sur les listes

Voici quelques opérations utiles spécifiques aux listes.

<code>list(iterable)</code>	convertit un objet itérable en liste
<code>l.append(objet)</code>	ajoute l'élément <code>objet</code> à la fin de la liste <code>l</code> .
<code>l.pop()</code>	renvoie le dernier élément de <code>l</code> et l'enlève de la liste.
<code>l.remove(objet)</code>	supprime la première occurrence de l'élément <code>objet</code> .
<code>del l[i] / del l[i:j]</code>	supprime l'élément <code>l[i]</code> / la sous-liste <code>l[i:j]</code> .
<code>l.reverse()</code>	renverse l'ordre des éléments de <code>l</code> .

Exemple : si `l=[1,2,3]`, après `l.append(4)` on a `l=[1,2,3,4]`.

Pour créer des listes de nombres entiers, on dispose maintenant de `list(range(a,b,pas))`.

Compréhensions de listes

C'est un mécanisme à la fois simple et puissant qui permet de construire une liste à partir des résultats d'une opération sur les éléments d'un ou plusieurs itérables. La structure est

<code>liste=[expression(i) for i in iterable if condition]</code>

Exemples : créer ces trois listes dans votre console puis affichez-les

- 1) `l1=[x**2 for x in range(1,11) if x%2==0]`.
- 2) `l2=[10*i+j for i in range(1,10) for j in range(1,10)]`.
- 3) `l3=[[10*i+j for i in range(1,10)] for j in range(1,10)]`.

Énumérations de listes

Si l'on veut les indices des éléments lors d'un parcours de liste, on utilise l'instruction

<code>enumerate(l)</code>	fournit la liste de tous les couples <code>(i,l[i])</code>
---------------------------	--

comme dans le code suivant :

<pre>for (i,e) in enumerate(L): print(i,e)</pre>
--

2.3) Les tuples

Définition

<i>Un tuple est une collection, délimitée par des parenthèses, d'éléments séparés par des virgules. C'est en outre un objet non mutable ou immuable et une séquence du type tuple.</i>
--

Exemple : `t=(7,4,"oh",23,"ah")` ; le tuple vide `t=()` ; le tuple à un élément `t=(a,)`.

Les listes et les tuples sont des objets parfaitement adaptés aux itérations et aux boucles for. En pratique, on privilégiera l'usage des tuples pour deux raisons :

- on ne risque pas de les modifier par erreur en cours de programme ;
- ils occupent moins de place en mémoire que les listes.

Fonctions spéciales

On dispose de peu de fonctions sur les tuples puisqu'ils sont immuables.

<code>tuple(l)</code>	convertit une liste <i>l</i> en tuple
<code>zip(l_1,...,l_n)</code>	à partir de <i>n</i> listes de même longueur, crée une liste de tuple de longueur <i>n</i> en associant les éléments de même indice.

La fonction `zip()` permet de parcourir simultanément des listes de même longueur :

<pre>for (a,b) in zip(liste1,liste2): ... a b</pre>

Notons que l'on parcourt ici une liste de couples avec l'instruction

<code>for (a,b) in L</code>

2.4) Les chaînes de caractères

Définition

Une **chaîne de caractères** est une suite quelconque de caractères délimitée soit par :

- des apostrophes, ce qui permet d'insérer des guillemets ;
- des guillemets, ce qui permet d'insérer des apostrophes ;
- des triples guillemets, ce qui prend en compte les tabulations, les retours à la ligne ainsi que tous les caractères spéciaux.

C'est de plus un objet **immuable** et une séquence du type **str**.

Exemples : `a="L'apéro est servi"` et `b='C'est une astuce "diabolique" !'`.

Opérations sur les chaînes

Voici quelques méthodes spécifiques aux chaînes de caractères. Elles ne modifient pas les arguments fournis et renvoient un autre objet.

<code>str(x)</code>	convertit un objet en chaîne de caractères.
<code>eval(s)</code>	évalue l'expression littérale <code>s</code> (sans les guillemets).

Exemple : `ch="Vous me le copierez "+str(100)+" fois pour demain !"`

Remarque : transformer un objet en chaîne de caractères permet de l'insérer dans une chaîne, tandis que transformer une chaîne de caractères en liste permet de la modifier ensuite.

Utilisation du backslash

- ❶ Il sert à couper les lignes trop longues, comme dans

```
Salut="Bonjour les enfants, vous avez bien dormi ? \
J'espère que vous êtes prêts pour de nouvelles aventures !"
```

- ❷ On l'utilise pour obtenir des caractères spéciaux de mise en page, parmi lesquels

<code>\t</code> : tabulation horizontale	<code>\v</code> : tabulation verticale	<code>\'</code> : apostrophe
<code>\n</code> : saut de ligne	<code>\f</code> : saut de page	<code>\"</code> : guillemets

Exemple : `texte='\n"N\'est-ce pas ?"', dit-il. \n "Pitié !"`.

3/ Applications

3.1) Sequence unpacking

Avec des égalités de séquences, on peut extraire des valeurs d'une séquence de type quelconque et les affecter au même nombre de variables. Ainsi :

- $(a, b, c) = [5, 8, 2]$ affecte les valeurs 5, 8 et 2 aux variables a, b et c.
- $(x, y) = (y, x)$ échange les contenus des variables x et y.
- $(a, b, c) = \text{"zut"}$ affecte les valeurs 'z', 'u' et 't' aux variables a, b et c.

C'est une façon simple de gérer les affectations multiples ou de récupérer des données.

3.2) Les suites récurrentes

On a vu lors du chapitre IV comment calculer et afficher les termes successifs d'une suite récurrente. Grâce aux listes, on va pouvoir les stocker et y accéder à l'aide de leurs indices.

Calculons alors les N premiers termes de la suite $(u_n)_{n \in \mathbb{N}}$ définie par
$$\begin{cases} u_0 = a \\ u_{n+1} = f(u_n) \end{cases}$$

Version classique

On calcule les termes successifs de la suite $(u_n)_{n \in \mathbb{N}}$, en les ajoutant au fur et à mesure à une liste U initialement vide. De la sorte, le terme u_n est tout simplement $U[n]$.

<pre> U ← [] u ← a (c'est u₀) POUR n ALLANT DE 0 à N-1 ajouter u à la fin de U u ← f(u) (calcul de u_{n+1}) Afficher U </pre>	<pre> U=[] u=a for n in range(N): U.append(u) u=f(u) print(U) </pre>
--	--

Exemple : étudier de cette façon la suite $(u_n)_{n \in \mathbb{N}}$ définie par $u_0 = 1$ et $u_{n+1} = \cos(u_n)$.

Astuce mathématique

En créant une liste contenant initialement le bon nombre de termes, on peut invoquer $U[n]$ et $U[n+1]$ dans les calculs, ce qui permet de « coller » à la définition mathématique.

<pre> U ← [0,0,...,0] (liste de N termes) U[0] ← a (c'est u₀) POUR n ALLANT DE 0 à N-2 U[n+1] ← f(U[n]) (calcul de u_{n+1}) Afficher U </pre>	<pre> U=[0]*N U[0]=a for n in range(N-1): U[n+1]=f(U[n]) print(U) </pre>
---	--

Remarque : cette méthode est particulièrement intéressante lors de récurrences multiples.

4/ Les conteneurs non-ordonnés

Dans de tels conteneurs, on ne pourra plus retrouver deux fois le même élément.

4.1) Les ensembles

Définition

Un **ensemble** est une collection, délimitée par des accolades, d'éléments séparés par des virgules. C'est un objet **mutable** mais **non indexable** du type `set`.

Exemples : l'ensemble `s={4, 1, 3, 2}` ; l'ensemble vide `s=set()`.

Opérations sur les ensembles

Voici quelques méthodes spécifiques aux ensembles, dont les opérations classiques.

<code>set(iterable)</code>	construit un ensemble avec les éléments d'un itérable.
<code>s.add(objet)</code>	ajoute l'élément <code>objet</code> à l'ensemble <code>s</code> .
<code>s.remove(objet)</code>	enlève l'élément <code>objet</code> de l'ensemble <code>s</code> .
<code>s1.union(s2)</code>	renvoie l'ensemble $s_1 \cup s_2$.
<code>s1.intersection(s2)</code>	renvoie l'ensemble $s_1 \cap s_2$.
<code>s1.difference(s2)</code>	renvoie l'ensemble $s_1 \setminus s_2$.
<code>s1.isdisjoint(s2)</code>	renvoie <code>True</code> si $s_1 \cap s_2 = \emptyset$ et <code>False</code> sinon.

Remarque : les opérateurs `<`, `<=`, `>` et `>=` fournissent les tests d'inclusion, stricte ou pas.

Exemple : si `s1={1, 2, 3, 4}` et `s2={3, 4, 5}`, alors `s1.intersection(s2)` renvoie `{3, 4}` tandis que `s2.difference(s1)` renvoie `{5}`.

4.2) Les dictionnaires ou tableaux associatifs

Définition

Un **dictionnaire** est une collection, délimitée par des accolades, de paires **clef:valeur** séparées par des virgules. C'est un objet **mutable** mais **non indexable** du type `dict`.

Exemples : le dictionnaire `d={1: 'janvier', 'deux': 2}` ; le dictionnaire vide `d={}`.

Remarques :

- ❶ Les clefs d'un dictionnaire doivent être des objets non mutables.
- ❷ Lors d'un parcours avec une boucle `for`, les éléments d'un dictionnaire sont les clefs.
- ❸ On accède aux valeurs d'un dictionnaire `d` grâce à ses clefs en tapant `d[clef]` : c'est la même syntaxe que pour une liste, les clefs remplaçant les indices. On retrouve de même les fonctions `del` et `pop()` ainsi que les créations par compréhension.
- ❹ Attention, si la clef est absente du dictionnaire, alors `d[clef]` provoque une erreur.

Opérations sur les dictionnaires

Voici quelques méthodes spécifiques aux dictionnaires.

<code>dict(liste)</code>	convertit une liste de couples en dictionnaire.
<code>d1.update(d2)</code>	ajoute le dictionnaire d_2 au dictionnaire d_1 .
<code>d[clef]=valeur</code>	modifie ou crée une entrée dans le dictionnaire d .
<code>d.get(clef)</code>	renvoie la valeur associée à la <code>clef</code> si elle est présente dans le dictionnaire, et rien sinon (pas d'erreur!).
<code>d.get(clef, default)</code>	même chose en renvoyant la valeur <code>default</code> si la <code>clef</code> est absente du dictionnaire d .
<code>d.keys()</code> / <code>d.values()</code>	renvoie la liste des clefs / des valeurs du dictionnaire d .
<code>d.items()</code>	renvoie la liste des couples (<code>clef</code> , <code>valeur</code>) de d .

Exemple : comptons de deux façons les lettres d'un texte donné, en faisant attention à ce que l'absence d'une clef ne provoque pas d'erreur.

<pre>d={} for c in texte: if c in d: d[c]=d[c]+1 else: d[c]=1</pre>	<pre>d={} for c in texte: d[c]=d.get(c,0)+1</pre>
---	---

Remarque : les dictionnaires sont des outils puissants et optimisés pour l'appartenance (l'accès est rapide). On les utilisera fréquemment pour stocker des informations dans des algorithmes d'exploration ou de recherche (occurrences de lettres dans un texte, issues d'une expérience aléatoire, parcours de graphes, etc...)

5/ Problèmes rencontrés lors des copies

Les copies d'objets mutables peuvent provoquer des effets surprenants. Il va donc falloir être particulièrement vigilants lorsque l'on copie des listes ou des dictionnaires.

5.1) Effets produits par les égalités de variables

Considérons un objet `objet` de type quelconque. À l'issue des deux instructions

```
x = objet
y = x
```

les deux variables pointent vers le même objet : elles indiquent la même adresse en mémoire, ce qui peut se vérifier à l'aide des commandes `id(x)` et `id(y)`. De même, l'instruction

```
x = objet2
```

affecte un nouvel objet à `x` sans modifier `y`, qui ne pointe plus vers le même objet.

Par contre, si objet est un objet mutable et que l'on modifie x sans la réaffecter, cela va impacter y car les variables x et y pointent toujours vers le même objet.

Exemples :

- ❶ En observant ce qui se passe, taper dans une console `x=[1,2,3]`, `y=x` puis `y[0]=8`.

En fait, l'instruction `y=x` ne duplique pas la liste en mémoire mais copie les références.

Du coup, toute modification d'une liste se répercute sur l'autre liste.

- ❷ De même, taper dans une console `v=[0,1,2]`, `m=[v,v,v]` puis `m[0][1]=17`.

Ainsi, la liste v et les trois lignes de la matrice m partagent la même représentation en mémoire.

Une modification d'un seul élément de m se répercutera sur toutes ses lignes ... et même sur v.

- ❸ Taper dans une console `n=[[0]*3]*4` puis `n[0][1]=1`.

Les lignes sont interdépendantes car on a créé à notre insu quatre copies d'une même liste.

- ❹ Cette fois, taper `d={1:[0,1]}`, `e=d` puis `d['zut']='flûte !'` et `e[1][0]=8`.

Toute modification d'un des dictionnaires se répercute sur l'autre.

Par conséquent, il faudra proscrire l'instruction `y=x` pour effectuer des copies d'objets mutables comme les listes ou les dictionnaires.

5.2) Bonne pratique des copies

Voici comment procéder pour éviter les problèmes que l'on vient de rencontrer.

- 1) Pour copier un tableau à une dimension x, on tapera `y=x[:]` ou `y=list(x)`.
- 2) Pour copier un dictionnaire d d'objets non mutables, on tapera `e=dict(d)`.
- 3) Pour copier un tableau à deux dimensions m, on tapera `n=[v[:] for v in m]`.
Mais ça devient plus compliqué et on privilégiera la méthode suivante.
- 4) Pour des listes ou dictionnaires quelconques, on utilisera de manière systématique la fonction `deepcopy` de la bibliothèque `copy`. Par exemple : `L2=copy.deepcopy(L1)`.

Remarque : si les dimensions et profondeurs sont connues, les compréhensions restent efficaces. Ainsi, on peut créer proprement un tableau nul à deux dimensions en tapant

```
n=[[0 for j in range(3)] for i in range(4)]
```

Exercices

Exercice 1.

On considère la liste $s=[7, 14, 5, 9, 8, 3]$. Que renvoient les instructions suivantes ?

$s[4]$ $s[-3]$ $s[2:4]$ $s[::3]$ $s[-1::-2]$

Exercice 2.

Indiquer les résultats des suites d'instructions suivantes :

<code>liste=[17,89,19,59]</code>	<code>liste2=list(range(2,19,2))</code>
<code>liste.pop()</code>	<code>del liste2[3]</code>
<code>liste.pop()</code>	<code>liste2.pop(10)</code>
<code>liste</code>	<code>liste2</code>

Exercice 3.

Donner le contenu des listes suivantes :

- 1) `[x**2 for x in [7,15,41,2,3] if x%2!=0]`
- 2) `[n + p for n in [2, 4] for p in [10, 20, 30]]`
- 3) `[[n + p for n in [2, 4]] for p in [10, 20, 30]]`

Exercice 4.

On dispose en mémoire d'une liste L de longueur 20, d'un objet x et d'un entier $n \in \llbracket 0;10 \rrbracket$. Comment faire pour insérer x dans L à la position d'indice i ?

Exercice 5.

On désire écrire un programme qui rentre des données dans une liste. Pour cela, on va :

- 1) demander en boucle des entrées à l'utilisateur ;
- 2) s'arrêter quand l'utilisateur tape simplement <Entrée> ;
- 3) afficher alors la liste obtenue.

Exercice 6. Créer les listes des 50 premiers termes :

- 1) d'une suite arithmétique de raison 3 et de premier terme 4 ;
- 2) d'une suite géométrique de raison 0.4 et de premier terme 5.

Exercice 7.

Afficher proprement les éléments d'une séquence précédemment créée.

Exercice 8.

Créer la liste des nombres pairs et celle des nombres impairs d'une liste d'entiers.

Exercice 9. Afficher le minimum et le maximum d'une liste de nombres.

Exercice 10. Écrire un programme qui :

- 1) renvoie la liste des chiffres d'un nombre en base 10 ;
- 2) renvoie un nombre à partir de la liste de ses chiffres en base 10

Exercice 11. On dispose des deux listes suivantes :

`l1=[31,28,31,30,31,30,31,31,30,31,30,31]`

`l2=['Janvier','Février','Mars',...,'Novembre','Décembre']`

Créer alors la liste `l3=['Janvier',31,'Février',28,...,'Décembre',31]`.

Exercice 12. Créer un programme qui :

- 1) demande deux nombres entiers l'utilisateur ;
- 2) donne la liste des multiples de 3 ou de 5 compris entre ces nombres.

Exercice 13.

Stocker dans une liste les 50 premiers termes de la suite définie par $u_0 = 3$ et $u_{n+1} = 2 + \frac{1}{u_n}$

Exercice 14. On définit la suite de Fibonacci par $u_0 = 0$, $u_1 = 1$ et $u_{n+2} = u_{n+1} + u_n$ pour $n \in \mathbb{N}$.

- 1) Stocker dans une liste les 50 premiers termes de cette suite.
- 2) Dans une nouvelle liste, stocker les valeurs des quotients u_{n+1}/u_n . Qu'observe-t-on ?

Exercice 15.

Stocker dans deux variables X et Y deux vecteurs (càd listes de nombres) de même longueur. En utilisant la fonction `zip()`, écrire un programme qui calcule leur produit scalaire.

Exercice 16. On dispose en mémoire d'une chaîne de caractères s.

- 1) Déterminer si la chaîne s contient un 'e'.
- 2) Compter le nombre de 'e' présents dans s.
- 3) Trouver la première occurrence de 'e' dans s.

Exercice 17.

Créer un programme qui demande une phrase à l'utilisateur puis recense le nombre de caractères de chaque type à l'aide d'un dictionnaire.

Exercice 18.

Utiliser un dictionnaire pour crypter / décrypter un texte par échange de lettres. On prendra un texte sans accents ni ponctuation pour faire simple.