



## Révisions

- Modules
- Chaines de caractères
- Fichiers
- Dictionnaires



## Modules

Un module est un bout de code que l'on a enfermé dans un fichier.

On **emprisonne** ainsi des **fonctions et des variables ayant toutes un rapport entre elles**.

Ainsi, si l'on veut travailler avec les fonctionnalités prévues par le module (celles qui ont été enfermées dans le module), il n'y a qu'à **importer le module** et utiliser ensuite toutes les fonctions et variables prévues.

Il existe un grand nombre de modules disponibles avec Python sans qu'il soit nécessaire d'installer des bibliothèques supplémentaires.

Par exemple, le **module math contient**, comme son nom l'indique, des **fonctions mathématiques**.

## La méthode import

Lorsque vous ouvrez l'interpréteur Python, les fonctionnalités du module math ne sont pas incluses.

Il s'agit d'un module, il vous appartient donc de l'importer si vous avez besoin de fonctions mathématiques.

première syntaxe d'importation.

```
import math
```

- le mot-clé import, qui signifie « importer » en anglais,
- suivi du nom du module, ici math.

Après l'exécution de cette instruction, Python importe le module math. Toutes les fonctions mathématiques contenues dans ce module sont maintenant accessibles.

## Appeler une fonction du module

Pour appeler une fonction du module, il faut taper

- le nom du module
- suivi d'un point « . »
- le nom de la fonction.

```
import math  
print(math.sqrt(16))
```

4.0

la fonction sqrt du module math renvoie la racine carrée du nombre passé en paramètre.

## Une autre méthode d'importation : from ... import ...

Il existe une autre méthode d'importation, qui nous permet **d'importer que la fonction dont nous avons besoin**, au lieu d'importer tout le module.

Dans le module math , si nous avons uniquement besoin, dans notre programme, de la fonction renvoyant la valeur absolue d'une variable.

```
from math import fabs  
print(fabs(-5))
```

5.0

on ne met plus le préfixe math. devant le nom de la fonction : nous l'avons importée avec la méthode from qui charge la fonction depuis le module indiqué et la place dans l'interpréteur au même plan que les fonctions existantes.

fabs est donc au même niveau que les fonctions principales

## Une autre méthode d'importation : from ... import ...

Vous pouvez appeler toutes les variables et fonctions d'un module en tapant « \* » à la place du nom de la fonction à importer

```
from math import *  
print(sqrt(4))  
print(fabs(5))
```

```
2.0  
5.0
```

À la ligne 1 de notre programme, l'interpréteur a parcouru toutes les fonctions et variables du module math et les a importées directement

Avantage : on économise la saisie systématique du nom du module en préfixe de chaque fonction.

Inconvénient : si on utilise plusieurs modules de cette manière et qu'il existe dans deux modules différents deux fonctions portant le même nom, l'interpréteur ne conservera que la dernière fonction appelée (il ne peut y avoir deux variables ou fonctions portant le même nom)



# Chaines de caractères



Voici un exemple de chaînes de caractères :

```
chaîne = "CECI EST UNE CHAÎNE"  
print(type(chaîne))  
print(chaîne.lower()) # chaîne en minuscule
```

```
<class 'str'>  
ceci est une chaîne
```

Si vous tapez `type(chaîne)` dans l'interpréteur, vous obtenez `<class 'str'>`

Les développeurs de Python ont créé le type **str** qui est utilisée pour créer des chaînes de caractères

Le type **str**, possède **plusieurs méthodes**, comme `lower`.

La fonction `lower` est propre aux chaînes de caractères. Toutes les chaînes peuvent y faire appel.

## Mettre en forme une chaîne

- `chaine.lower()` : renvoie la chaîne en minuscules mais ne modifie pas la chaîne
- `str()` : crée un objet *chaîne de caractères*
- `chaine.upper()` : renvoie la chaîne en majuscules
- `chaine.capitalize()` : renvoie La première lettre en majuscule
- `chaine.strip()` : retire les espaces au début et à la fin de la chaîne
- `chaine.center(20)` : On lui passe en paramètre la taille de la chaîne que l'on souhaite obtenir. La méthode va ajouter alternativement un espace au début et à la fin de la chaîne, jusqu'à obtenir la longueur demandée

On peut « chaîner » les méthodes : Exemple :

```
titre = "introduction"  
print(titre.upper().center(20))
```

```
' INTRODUCTION '
```

Vous pouvez en voir la liste exhaustive des méthodes dans l'aide, en tapant, dans l'interpréteur : `help(str)`

## La concaténation de chaînes

La concaténation cherche à regrouper deux chaînes en une, en mettant la seconde à la suite de la première.

Le signe « + » est le signe utilisé pour **concaténer** deux chaînes

```
prenom = "Jean"  
message = "Bonjour"  
chaine_complete = message + " " + prenom # On utilise le symbole '+' pour concaténer deux chaînes  
print(chaine_complete)
```

**Bonjour Jean**

**Si vous voulez concaténer des chaînes et des nombres, il faudra convertir** les nombres en chaînes pour pouvoir le concaténer aux autres chaînes, sinon vous obtiendrez une erreur

```
age = 21  
message = "J'ai " + str(age) + " ans."  
print(message)
```

**J'ai 21 ans.**

## Vérifier si un élément existe dans une chaîne de caractères

Pour déterminer si un élément spécifié est présent dans une chaîne de caractères, utilisez le mot-clé *in* :

```
prenom = "Jean"  
if "p" in prenom :  
    print("Oui, 'p' est dans le prénom")
```

Oui, 'p' est dans le prénom

## Parcours de chaînes

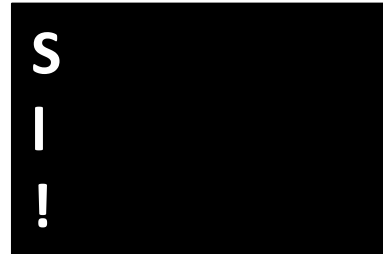
### Accéder aux caractères d'une chaîne

Pour accéder aux lettres constituant une chaîne, nous allons utiliser un indice : On précise entre crochets [] l'indice, c'est-à-dire la position du caractère auquel on souhaite accéder.

```
chaîne[position_dans_la_chaîne]
```

Par exemple, nous souhaitons sélectionner la première lettre d'une chaîne.

```
chaîne = "Salut !"
print(chaîne[0]) # Première lettre de la chaîne
print(chaîne[2]) # Troisième lettre de la chaîne
print(chaîne[-1]) # Dernière lettre de la chaîne
```



S  
l  
!

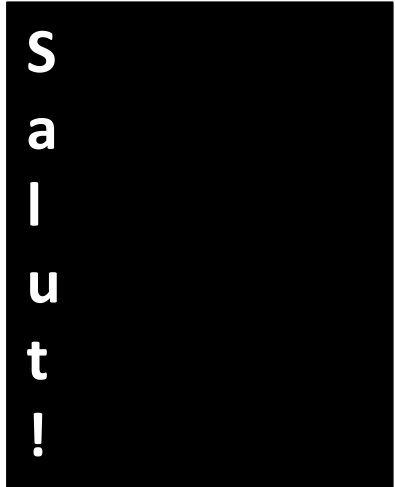
- On commence à compter à partir de 0. La première lettre est donc à l'indice 0.
- On peut accéder aux lettres en partant de la fin à l'aide d'un indice négatif. Quand vous tapez chaîne[-1], vous accédez ainsi à la dernière lettre de la chaîne.

## Parcours de chaînes

### Méthode de parcours par while

on peut parcourir une chaîne grâce à la boucle while.

```
chaine = "Salut"  
i = 0 # On appelle l'indice 'i' par convention  
while i < len(chaine):  
    print(chaine[i]) # On affiche le caractère à chaque tour de boucle  
    i += 1
```



S  
a  
l  
u  
t  
!

N'oubliez pas d'incrémenter i.

Si vous essayez d'accéder à un indice qui n'existe pas (par exemple 25 alors que votre chaîne ne fait que 20 caractères de longueur), Python lèvera une exception de type **IndexError**.

## Parcours de chaînes

### Méthode de parcours par For

on peut parcourir une chaîne grâce à la boucle for.

Par exemple, nous allons parcourir la séquence "Bonjour les amis"

```
chaine = "Bonjour les amis"  
for lettre in chaine:  
    print(lettre)
```

1. **l'interpréteur va créer une variable lettre** qui contiendra le premier élément de la chaîne (autrement dit, la première lettre).
2. la variable lettre prend successivement la valeur de chaque lettre contenue dans la chaîne de caractères (d'abord B, puis o, puis n...).
3. On affiche ces valeurs avec print et cette fonction revient à la ligne après chaque message, ce qui fait que toutes les lettres sont sur une seule colonne.

Notez bien qu'il est **inutile d'incrémenter la variable lettre**.

Python se charge de l'incrémentation, c'est l'un des grands avantages de l'instruction for.

B  
o  
n  
j  
o  
u  
r  
  
l  
e  
s  
  
a  
m  
i  
s

## Sélection de chaînes

La sélection consiste à extraire une partie de la chaîne.

Cette opération renvoie le morceau de la chaîne sélectionné, sans modifier la chaîne d'origine.

```
chaîne[indice_debut:indice_fin]
```

Par exemple

```
presentation = "salut"
```

```
presentation[0:2] # On sélectionne les deux premières lettres : 'sa'
```

```
presentation[2:len(presentation)] # On sélectionne la chaîne sauf les deux 1ères lettres : 'lut'
```

on peut sélectionner du début de la chaîne jusqu'à un indice, ou d'un indice jusqu'à la fin de la chaîne, sans préciser autant d'informations :

```
presentation[:2] # Du début jusqu'à la troisième lettre non comprise : 'sa'
```

```
presentation[2:] # De la troisième lettre (comprise) à la fin : 'lut'
```



## Sélection de chaînes

on peut sélectionner à partir de la fin de la chaîne:

```
presentation[-1] # 't'
```

On peut constituer une nouvelle chaîne, en remplaçant une lettre par une autre :

```
mot = "lac"  
mot = "b" + mot[1:]  
print(mot)
```

**bac**

Pour rechercher/remplacer des lettres, nous avons aussi à notre disposition les méthodes **count**, **find** et **replace** de la classe str, à savoir « compter », « rechercher » et « remplacer ».



## Fichiers

## chemins relatifs et absolus

Pour décrire l'arborescence d'un système, on a deux possibilités :

### 1- Le chemin absolu

on **décrit l'intégralité du chemin (la suite des répertoires) menant au fichier.**

- Sous Windows, on partira du nom de volume (C:/,D:/...). Par exemple : C:/test/fic.txt
- Sous les systèmes Unix, ce sera depuis la racine /.

### 2- Le chemin relatif

on **tient compte du dossier dans lequel on se trouve.**

- Ainsi, si on se trouve dans le dossier *C:/test* et que l'on souhaite accéder au fichier *fic.txt* contenu dans ce même dossier, le chemin relatif menant à ce fichier sera *fic.txt*.  
Mais si on se trouve dans le dossier *C:*, le chemin relatif sera *test/fic.txt*.
- Quand on décrit un chemin relatif, on utilise parfois le symbole *..* qui désigne le répertoire parent. Si le répertoire de travail courant est *test* et que l'on souhaite accéder à *fic2.txt* qui se trouve dans un répertoire *test2*, notre chemin relatif sera *..\test2\fic2.txt*.

## Nom des fichiers

Choisissez de préférence des noms de fichiers courts.

Évitez dans toute la mesure du possible les caractères accentués, les espaces et les signes typographiques spéciaux.

Dans les environnements de travail de type Unix (MacOS, Linux, ...), il est souvent recommandé aussi de n'utiliser que des caractères minuscules.

## Ouverture du fichier

Pour ouvrir un fichier avec Python, on utilise la fonction **open**, disponible sans avoir besoin de rien importer.

Elle prend en paramètre :

- le **chemin** (absolu ou relatif) menant au fichier à ouvrir ;
- le **mode d'ouverture**.

Le mode est donné sous la forme d'une chaîne de caractères :

- 'r': ouverture en lecture (Read). C'est le mode par défaut donc le paramètre peut être omis.
- 'w': ouverture en écriture (Write). Le contenu du fichier est écrasé. Si le fichier n'existe pas, il est créé.
- 'a': ouverture en écriture en mode ajout (Append). On écrit à la fin du fichier sans écraser l'ancien contenu du fichier. Si le fichier n'existe pas, il est créé.

```
mon_fichier = open("fic.txt", "r")
```

Ici nous souhaitons lire le fichier fic.txt. Nous avons donc utilisé le mode 'r'.

## Fermeture de fichier

N'oubliez pas de fermer un fichier après l'avoir ouvert.

Si d'autres applications, ou d'autres morceaux de votre propre code, souhaitent accéder à ce fichier, ils ne pourront pas car le fichier sera déjà ouvert.

C'est surtout vrai en écriture, mais prenez de bonnes habitudes.

La méthode à utiliser est **close** :

```
mon_fichier.close()
```

## Lecture de l'intégralité du fichier

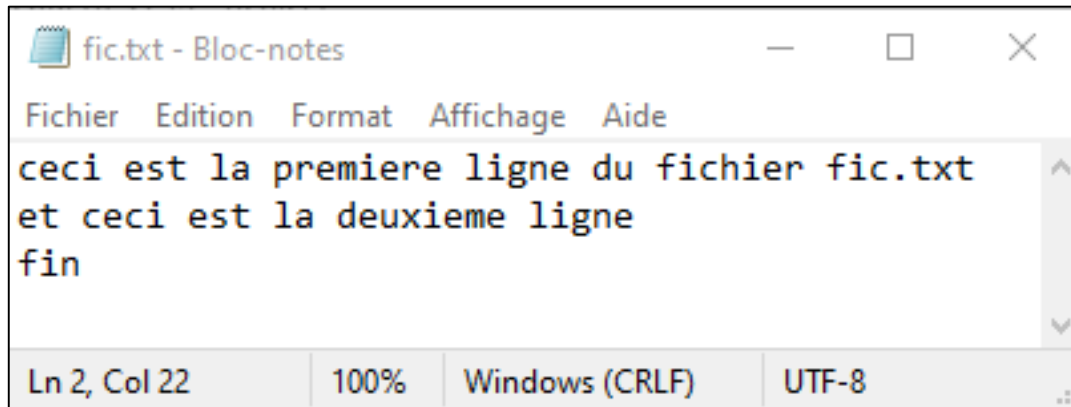
Pour ce faire, on utilise la méthode **read** de la classe TextIOWrapper.

Elle renvoie tout le contenu du fichier, que l'on capture dans une chaîne de caractères :

```
mon_fichier = open("fic.txt", "r")  
contenu = mon_fichier.read()  
print(contenu)  
mon_fichier.close()
```

Code.py

On part du principe que le fichier *fic.txt* se trouve au même endroit que le code *Code.py*, je peux donc indiquer le chemin relatif "fic.txt"



fic.txt

**ceci est la premiere ligne du fichier fic.txt  
et ceci est la deuxieme ligne  
fin**

Avec la méthode **read**, on peut limiter le nombre des caractères lus, par exemple:

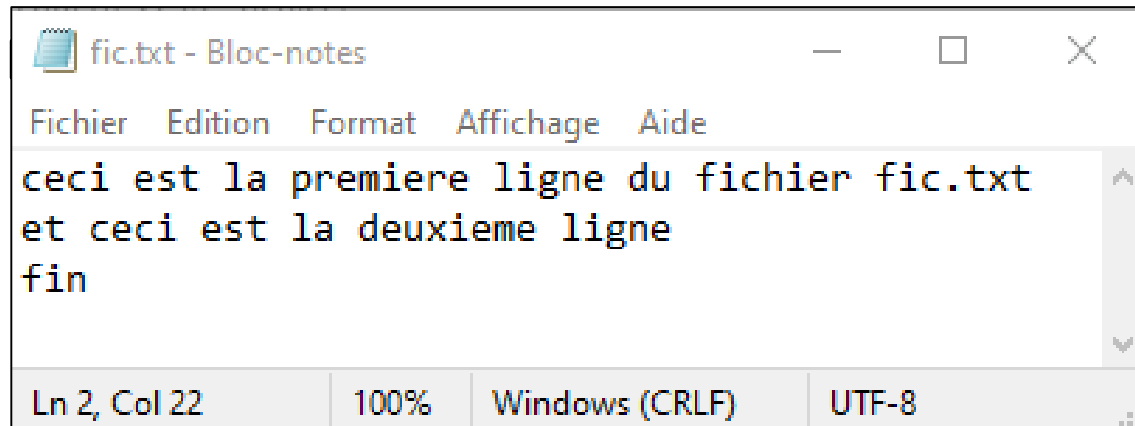
```
suivant10= f.read(10) # lit au plus 10 caractères à partir de la position courante
```

## Lecture de chaque ligne du fichier

Si on veut lire chaque ligne du fichier, il suffit d'ouvrir le fichier puis d'utiliser une boucle

```
mon_fichier = open("fic.txt", "r")  
for ligne in mon_fichier :  
    print(ligne)  
mon_fichier.close()
```

Si chaque fin de ligne contient un saut de ligne, vous aurez dans votre `print(ligne)` une ligne vide traduisant un saut de ligne.



**ceci est la premiere ligne du fichier fic.txt**

**et ceci est la deuxieme ligne**

**Fin**



## Écriture dans un fichier

il faut ouvrir le fichier avant tout avec open :

- mode w : écrase le contenu éventuel du fichier

Puis, pour écrire dans un fichier, on utilise la méthode **write** en lui passant **en paramètre la chaîne à écrire** dans le fichier.

Elle renvoie le nombre de caractères qui ont été écrits.

```
mon_fichier = open("fic.txt", "w")  
mon_fichier.write("Premier test d'écriture\n")  
mon_fichier.close()
```

Attention :

- La méthode write n'accepte en paramètre que des chaînes de caractères. Si vous voulez écrire dans votre fichier des nombres, il faudra les convertir en chaîne avant de les écrire *str()* et les convertir en entier après les avoir lus *int()*.
- La méthode write n'ajoute pas d'elle-même le retour à la ligne. Si on le veut, il faut insérer le caractère saut de ligne « \n » après chaque écriture.

## Ajout dans un fichier

il faut ouvrir le fichier avant tout avec open :

- mode a : ajoute ce que l'on écrit à la fin du fichier, sans l'écraser comme c'est le cas dans le mode 'w'.

Si le fichier demandé n'existe pas, il sera créé.

Puis, pour écrire dans un fichier, on utilise également la méthode **write**

```
mon_fichier = open("fic.txt", "a")  
mon_fichier.write("ajout de texte\n")  
mon_fichier.close()
```

## Ouverture du fichier

Deuxième méthode d'ouverture de fichier :

```
with open("fic.txt", 'r') as f:  
    for ligne in f:  
        # traitement prévu
```

à la fin du bloc with open(), le fichier fic.txt est automatiquement fermé



# Dictionnaires

- ☐ Les dictionnaires sont des objets pouvant en contenir d'autres, comme les listes.
- ☐ Cependant, au lieu d'héberger des informations dans un ordre précis, ils **associent chaque objet contenu à une clé** (la plupart du temps, une chaîne de caractères)
- ☐ Donc pour accéder aux objets contenus dans le dictionnaire, on n'utilise pas des indices mais des clés, qui peuvent être de bien des types distincts

## Créer un dictionnaire

première méthode d'instanciation d'un dictionnaire vide :

```
mon_dictionnaire = dict()
```

deuxième méthode d'instanciation d'un dictionnaire vide :

```
mon_dictionnaire = {}
```

Si on résume :

- les crochets délimitent les listes
- les accolades {} délimitent les dictionnaires.

## Créer un dictionnaire

La classe sur laquelle se construit un dictionnaire est **dict**.

```
mon_dictionnaire = {}  
print(type(mon_dictionnaire))  
print(mon_dictionnaire)
```

```
<class 'dict'>  
{}
```

## Ajouter des éléments dans un dictionnaire

Pour ajouter des clés et valeurs dans notre dictionnaire vide :

```
mon_dictionnaire = {}  
mon_dictionnaire["pseudo"] = "Jean"  
mon_dictionnaire["mot de passe"] = "****"  
print(mon_dictionnaire)
```

```
{'mot de passe': '****', 'pseudo': 'Jean'}
```

Nous indiquons entre crochets la clé à laquelle nous souhaitons accéder (pseudo par exemple)

Si la clé n'existe pas, elle est ajoutée au dictionnaire avec la valeur spécifiée après le signe = (Jean par exemple)



## Ajouter des éléments dans un dictionnaire

Concernant les clés, on peut utiliser quasiment tous les types comme clés.

Au lieu d'utiliser des chaînes de caractères, vous pouvez utiliser des entiers :

```
mon_dictionnaire = {}  
mon_dictionnaire[0] = "a"  
mon_dictionnaire[1] = "e"  
mon_dictionnaire[2] = "i"  
mon_dictionnaire[3] = "o"  
mon_dictionnaire[4] = "u"  
print(mon_dictionnaire)
```

```
{0: 'a', 1: 'e', 2: 'i', 3: 'o', 4: 'u'}
```

## Ajouter des éléments dans un dictionnaire

On peut aussi créer des dictionnaires déjà remplis :

```
panier = {"pomme":3, "poire":6, "fraise":7}
```

On précise :

- la clé entre guillemets ,
- le signe deux points « : »
- la valeur correspondante.

On sépare les différents couples clé : valeur par une virgule.

C'est d'ailleurs comme cela que Python vous affiche un dictionnaire quand vous le lui demandez.

## Modifier des éléments dans un dictionnaire

Un dictionnaire **ne peut pas contenir deux clés identiques** (mais rien n'empêche d'avoir deux valeurs identiques).

L'ancienne valeur à l'emplacement indiqué est remplacée par la nouvelle :

```
mon_dictionnaire = {}  
mon_dictionnaire["pseudo"] = "Jean"  
mon_dictionnaire["mot de passe"] = "****"  
mon_dictionnaire ["pseudo"] = "Arthur"  
print(mon_dictionnaire)
```

```
{'mot de passe': '****', 'pseudo': 'Arthur'}
```

La valeur 'Jean' pointée par la clé 'pseudo' a été remplacée, à la ligne 4, par la valeur 'Arthur'.

## Accéder à un élément dans un dictionnaire

Pour accéder à la valeur d'une clé précise :

```
mon_dictionnaire["mot de passe"]  
print(mon_dictionnaire)
```

```
****
```

Si la clé n'existe pas dans le dictionnaire, une exception de type **KeyError** sera levée.

## Supprimer des clés d'un dictionnaire

Vous avez deux possibilités :

- le mot-clé del;

```
panier = {"pomme":3, "poire":6, "fraise":7}  
del panier["pomme"]  
print(mon_dictionnaire)
```

```
{"poire":6, "fraise":7}
```

- la méthode de dictionnaire pop.

```
panier = {"pomme":3, "pantalon":6, "tee shirt":7}  
print(panier.pop("pomme"))
```

```
3
```

La méthode pop supprime la clé précisée, mais elle renvoie en plus la valeur supprimée :

NB : Un dictionnaire n'a pas de structure ordonnée : si vous supprimez l'indice 2, le dictionnaire, contrairement aux listes, ne va pas décaler toutes les clés d'indice supérieur à l'indice supprimé

## Parcourir un dictionnaire

### Parcours des clés

Pour parcourir les clés d'un dictionnaire, on a deux solutions :

```
fruits = {"pommes":21, "melons":3, "poires":31}  
for cle in fruits:  
    print(cle)
```

```
fruits = {"pommes":21, "melons":3, "poires":31}  
for cle in fruits.keys():  
    print(cle)
```

```
melons  
poires  
pommes
```

On parcourt la liste des clés contenues dans le dictionnaire.

les clés ne s'affichent pas dans l'ordre dans lequel on les a entrées, car les dictionnaires n'ont pas de structure ordonnée.

## Parcourir un dictionnaire

### Parcours des valeurs

On peut parcourir les valeurs contenues dans un dictionnaire.

Pour ce faire, on utilise la méthode **values** (« valeurs » en anglais).

```
fruits = {"pommes":21, "melons":3, "poires":31}  
for valeur in fruits.values():  
    print(valeur)
```

3

31

21

## Parcourir un dictionnaire

### Parcours des clés et valeurs simultanément

Pour avoir en même temps les indices et les objets d'un dictionnaire, on utilise la méthode **items**.

Elle renvoie une liste, contenant les couples clé : valeur, sous la forme d'un tuple :

```
fruits = {"pommes":21, "melons":3, "poires":31}
for cle, valeur in fruits.items():
    print("La clé ", cle , " contient la valeur ", valeur)
```

**La clé melons contient la valeur 3.**

**La clé poires contient la valeur 31.**

**La clé pommes contient la valeur 21.**



**Vérifier l'existence d'un élément dans un dictionnaire**  
on utilise une condition :

```
fruits = {"pommes":21, "melons":3, "poires":31}  
if 21 in fruits.values():  
    print("Un des fruits se trouve dans la quantité 21.")
```

**Un des fruits se trouve dans la quantité 21.**