Structures de données et Fondamentaux de Python

Ben Jannet Azza

1 Structures de données :

En Python, il existe des structures données utilisées pour ranger plusieurs données au même endroit. Ces structures de données sont réparties en deux catégories : les séquences, où l'ordre des éléments a une importance, et les collections, où l'ordre n'est pas retenu (donc, on ne peut pas accéder à un élément particulier à l'aide d'un index de position). De plus, chaque type de structure de données peut être mutable ou non mutable.

1.1 Les séquences :

Les séquences (ensembles ordonnés) sont des structures de données où l'ordre des éléments est important et est retenu par le programme. Dans une séquence, il est possible d'accéder à un élément particulier si l'on connait sa position (son numéro d'index). Parmis les principaux types de séquences en Python, on trouve les listes, les tuples et les ranges.

1.1.1 Les listes :

Création et manipulation des listes

```
#Créer une liste vide
liste_vide = []
#Créer une liste avec des éléments
liste = [10, 20, 30, 40, 50]
print(type(liste)) #<class 'list'>
#Ajouter un élément à la fin
liste.append(60)
#Insérer un élément à l'indice 2
liste.insert(2, 25)
```

Les listes peuvent contenir des objets de différentes nature.

Exemple :

```
Liste2 = [3, 8.2, "Bonjour", complex(1,2)]
print(liste2)
```

Les listes sont des objets **mutables**, donc il est possible d'en modifer le contenu sans créer un nouvel objet. Si deux variables pointent sur la même liste, et que l'on modifie la liste à travers l'une des variables, la seconde "verra" le changement.
Exemple :

```
a = [1, 2, 3, 4]
b = a
#On modifie la liste a en rajoutant un élément à la fin
a.append(5)
#Affichons la liste b pour vérifier qu'elle a été modifiée
print(f"La liste b vaut {b}")
```

Ces opérations (Tableau 1) ne peuvent être effectuées que sur des séquences mutables, comme les listes, car elles modiffient le contenu de la liste sur laquelle elles s'exécutent..

Tableau 1: Opérations sur les listes en Python

Opération	**Description**
`s.append(x)`	Rajoute l'élément `x` à la fin de la séquence `s`.
`s.extend(m)`	Rajoute la liste `m` à la fin de la séquence `s`.
`s.insert(k, e)`	Insère l'élément `e` dans la séquence `s` à la position `k`. Si la position `k` est en dehors de la séquence, alors `e` est inséré à la fin de la séquence `s`.
`s.remove(e)`	Retire la première occurrence de l'élément `e` dans la liste `s`.
`del s[n]`	Supprime l'élément à la position `n` de la séquence `s`. En général, l'opérateur `del` supprime tout objet qui est écrit à sa droite.
`s.reverse()`	Renverse la séquence `s`, en plaçant les derniers éléments au début, et vice-versa.
`s.sort()`	Trie la séquence `s`, par ordre croissant. Un second paramètre permet de trier par ordre décroissant. Cette fonction produira une exception si tous les éléments de `s` ne sont pas du même type (ils doivent être comparables).

Voici un exemple de code Python faisant appel à ces opérations sur une liste :

```
#On initialise une liste
l=[34, 45, 12, 8, 90, 45]
#On ajoute un élément à la liste puis on l'affiche
l.append(42)
print(l)
#On ajoute une liste à la liste puis on l'affiche
l.extend([51, 87, 4])
print(l)
#On ajoute un élément (100) dans la liste l à la position 5, puis on
    l'affiche
```

```
l.insert(5, 100)
print(l)
#On retire la première occurrence de l'élément 45
l.remove(45)
print(l)
#On supprime l'élément à la position 4
del l[4]
print(l)
#On renverse la liste
l.reverse()
print(l)
#On trie la liste
l.sort()
print(l)
```

On remarquera que la liste `l` évolue au fur et à mesure de l'exécution du programme : les fonctions utilisées modiffient directement la liste, et n'en créent pas une nouvelle. Ceci explique pourquoi ces fonctions ne sont pas disponibles pour des séquences non mutables.

Supprimer un élément de la liste :

```
#Supprimer le dernier élément
liste.pop() #syntaxe: liste.pop(indice)
#Supprimer l'élément à l'indice 2 (troisième élément) de la liste
liste.pop(2)
#Supprimer un élément à l'aide du mot clé del
del liste[2] #Syntaxe: del liste[indice]
#Supprimer un élément d'une liste
liste.remove(10) #syntaxe: liste.remove(élément)
```

Remarque : La fonction `pop()` renverra une erreur d'index lorsque vous essayez d'extraire un élément d'une liste en utilisant un index qui n'existe pas dans la liste (`IndexError: pop index out of range`). La fonction `remove()` renverra une erreur de valeur si un élément n'est pas dans la liste (`ValueError: list.remove(x): x not in list`).

Les listes imbriquées : Une liste (ou toute autre structure de données) étant un objet, il est tout à fait possible de placer une liste à l'intérieur d'une autre liste.
Exemple :

```
#On déclare une liste contenant des listes
a = [[1, 2, 3], [3, 7, 9, 10], 8]
print(a)
#On affiche la taille de la liste
print(len(a))
#On affiche le second élément de la liste (qui est lui-même une liste)
print(a[1])
```

Voici un exemple de code Python faisant appel à des opérations sur une liste (voir tableau 2) :

```
#On initialise une liste
l=[34, 45, 12, 8, 90, 45]
#On affiche la liste
print(l)
#On affiche un élément de l à la position 3
print(l[3])
#On affiche un autre élément de l à la position -2 (avant dernier)
print(l[-2])
#On affiche la taille de l
print(f'La taille de l est {len(l)}')
#On concatène deux listes ensemble
p = l + [2, 89, 31]
print(p)
#On essaye la multiplication (la concatenation de l avec lui-même 3
    fois)
p = 3*l
print(p)
#Le maximum et le minimum
print(f'Le minimum de l vaut {min(l)} tandis que le maximum
    vaut {max(l)}')
#La somme des éléments de l
print(sum(l))
#La recherche de la position de l'élément 45
print(l.index(45))
#Le dénombrement d'éléments
print(l.count(45))
print(92 in l)
#Le tri (ordre décroissant)
p = sorted(l, reverse=True)
print(p)
```

L'opération de découpage des structures de données (slicing) : En Python, il est possible d'extraire des sous listes de listes (ou autres séquences), c'est à dire extraire un sous ensemble d'éléments d'une séquence pour former une nouvelle séquence. Voici un exemple de slicing :

```
l = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
#On souhaite extraire de la liste tous les nombres de la position 3 à 7
    :
l2 = l[3:8]
print(l2)
```

La syntaxe générale d'un slicing est la suivante :
`nom_sequence[debut du parcours : fin du parcours (exclu) : pas d'incrémentation]`
Nous noterons plusieurs choses concernant le slicing :
- L'indice du début de parcours est inclus, tandis que celui de fin du parcours

est toujours exclu.
- Les indices de début ou de fin peuvent être négatifs, afin de faire référence à des positions relatives à la fin de la séquence.
- Le pas d'incrémentation peut-être négatif si l'on souhaite extraire des éléments dans le sens contraire de la séquence d'origine.
- Si le découpage réalisé sort de la liste (indice de fin trop élevé, ou indice de départ trop bas), les parties "en dehors" de la liste sont ignorées.
- Il est possible d'ignorer la valeur d'indice de début ou de fin dans un découpage. Dans ce cas, c'est le tout début ou la toute fin de la liste qui sera prise en compte.

Le tableau 3 dans la section Annexe récapitule des exemples d'opérations de découpage (slicing) sur une liste.

1.1.2 Les tuples :

En Python, les tuples sont des structures de données ordonnées et non mutables. Nous pouvons donc les voir comme des listes, sans la possibilité d'en modifier le contenu une fois la déclaration faite.

Initialisation d'un tuple en Python:
Pour initialiser un tuple en Python, il suffit de déclarer une liste, mais en utilisant des parenthèses à la place des crochets :

```
t = (3, 8, 7, 2)
print(type(t)) #<class 'tuple'>
```

Tout comme pour une liste, il est possible de ranger des objets de différents types dans un même tuple :

```
t2 = (3, 8.2, "Bonjour", complex(1,2))
print(t2)
```

Les tuples ne sont pas mutables :
Les tuples étant des structures de données non mutables, il n'est pas possible d'en modifier le contenu, et une erreur surviendra si' l'on tente de le faire :

```
t = (3, 9, 11, 5)
t[2] = 10
```

```
TypeError: 'tuple' object does not support item assignment
```

Cependant, il est possible de rajouter des éléments mutables à un tuple, permettant ainsi de le modifier indirectement plus tard.
Exemple :

```
l = [32, 90] #Déclaration d'une liste
t = (3, 9, 1, 5) #Déclaration du tuple
print(t)
l[0]=0 #On modifie la liste l, pour tenter de modifier le contenu du
    tuple
print(t)
```

On voit que le premier élément de l est modifié, et donc le tuple est modifié.

Opérations sur les tuples :

Les tuples étant des séquences, toutes les opérations listées dans le tableau 2 peuvent être effectuées.

1.1.3 Les ranges :

Les ranges sont comparables à des tuples : ce sont des séquences non mutables.

Initialisation d'un range en Python :

Les ranges sont créés à l'aide de la fonction range, permettant de générer une suite de nombre suivant une progression arithmétique. La fonction range prend trois paramètres :

1. `start` est le premier nombre de la séquence qui sera générée.
2. `stop` est la limite exclue de la séquence. Autrement dit, tout nombre de la séquence sera strictement inférieur à stop si `start < stop` (suite croissante), et strictement supérieur à stop si `start > stop` (suite décroissante).
3. `step` est le pas d'incrémentation entre deux éléments consécutifs de la suite arithmétique.

```
#On génère une suite arithmétique croissante, de premier terme 1, et de
    raison 13
r = range(1, 100, 13)
print(f"r est de type {type(r)}") #r est de type <class 'range'>
```

Remarque : Il n'est cependant pas possible d'afficher le contenu d'un range avec la fonction `print`.

```
r = range(1, 100, 13)
#Afficher le contenu de r
for i in r:
        print(i, end=' ')

#Créer une liste avec range
range_liste = list(r)
print(range_liste)
print(type(range_liste)) #<class 'list'>
```

On utilisera donc un range lorsque l'on aura besoin d'une séquence de nombres suivant une progression simple. En général, les ranges sont utilisés profusément avec les boucle for (que l'on verra plus tard).

1.2 Les collections :

1.2.1 Les ensembles :

Les ensembles, nommées set en Python, sont des collections mutables, ce qui signifie que l'ordre d'ajout des éléments n'est pas préservé ni enregistré. Le principe des ensembles est qu'il ne peut y avoir deux fois le même élément.

Initialisation d'un set en Python :
Pour déclarer un set en Python, il suffit d'écrire un ensemble d'éléments entre accolades :

```
s = {"rouge", "bleu", "vert", "jaune", "orange", "marron", "
    rose"}
print(type(s))
print(s)
```

Un set est une collection, c'est à dire que l'ordre d'insertion des éléments n'est pas mémorisé par Python car il n'a pas d'importance. On voit, dans l'exemple précédent, que, lors de l'affichage du set, les couleurs apparaissent dans un ordre différent de celui dans lequel elles étaient déclarées au début. Exécuter plusieurs fois ce code produira différents ordres d'affichage des couleurs. Ce principe empêche d'indexer les éléments d'un set, par exemple `print(s[2])` renvoie `TypeError: 'set' object is not subscriptable`.

Comme dit précédemment, un set n'acceptera pas d'éléments dupliqués, donc si l'on met plusieurs fois le même objet dans un même set, ce dernier n'apparaîtra qu'une seule fois :

```
s = {8, 34, 87, 56, 90, 34, 101}
print(s)
```

Sortie : {34, 101, 8, 87, 56, 90}

1.2.2 Les dictionnaires :

Les dictionnaires sont, depuis la version 3.7 de Python, des séquences et non des collections, c'est à dire que l'ordre d'ajout des éléments est mémorisé. Néanmoins, les dictionnaires sont le plus souvent utilisés comme des collections, aussi en parlons-nous dans cette section. Le principe des dictionnaires est d'associer deux objets ensemble : l'un des objets est la valeur que l'on souhaite stocker, tandis que l'autre sera une clef permettant de retrouver le premier objet.

Initialisation d'un dictionnaire en Python :
Un dictionnaire est initialisé en écrivant, entre accolades, des clefs suivis, à l'aide de deux points, de valeurs :

```
a = {'DF-723-HK' : 'Renault', 'AB-733-JH' : 'Peugeot', 'ER
    -310-JU' : 'Toyota'}
print(type(a)) #<class 'dict'>
```

Dans cet exemple, les clefs sont les numéro de plaque d'immatriculation, tandis que les valeurs associées sont les modèles de voiture.
Même si l'on peut accéder aux éléments du dictionnaire à l'aide d'un index, car ce n'est plus une collection depuis peu mais une séquence, il est préférable

d'accéder aux éléments d'un dictionnaire à l'aide de leur clef. Pour ce faire, il existe la méthode `get` :

```
print(a.get('AB-733-JH'))  #Sortie :  Peugeot
```

Une clef spécifique ne peut pas apparaître deux fois dans un même dictionnaire. Dans le cas d'une clef dupliquée, la dernière valeur associée à la clef sera prise en compte :

```
a = {'DF-723-HK' : 'Renault', 'AB-733-JH' : 'Peugeot', 'ER
    -310-JU' : 'Toyota', 'DF-723-HK' : 'BMW'}
for key in a:
        print(f'{key} : {a[key]}')
```

Sortie :

```
DF-723-HK : BMW
AB-733-JH : Peugeot
ER-310-JU : Toyota
```

On peut rajouter une nouvelle clef à un dictionnaire simplement en attribuant une valeur à cette clef :

```
a['HJ-997-KL'] = 'Kia'
for key in a:
        print(f'{key} : {a[key]}')
```

Sortie :

```
DF-723-HK : BMW
AB-733-JH : Peugeot
ER-310-JU : Toyota
HJ-997-KL : Kia
```

2 Boucles et conditions

2.1 Les blocs d'instructions et l'indentation

En Python, un **bloc d'instructions** est un groupe de lignes de code qui est exécuté ensemble sous certaines conditions, comme dans une boucle ou une condition `if`. Les blocs d'instructions sont délimités par l'**indentation**, c'est-à-dire un espace ou une tabulation en début de ligne. Contrairement à d'autres langages de programmation qui utilisent des accolades ({}) pour délimiter les blocs, Python repose entièrement sur l'indentation pour structurer le code.
De plus, lorsque vous utilisez des instructions comme `if`, `elif`, `else`, `for`, ou `while`, il est obligatoire de terminer la ligne par deux points « : ». Cela indique que les instructions qui suivent font partie du bloc d'instructions associé. Par exemple, les instructions suivantes font partie du même bloc car elles sont indentées de la même manière :

```
if x > 5:
        print("x est plus grand que 5")
        print("Ceci est dans le meme bloc")
```

Il est important de veiller à une indentation cohérente dans tout le programme pour éviter des erreurs de syntaxe. Aussi, l'absence des deux points après ces mots-clés (`if`, `elif`, `else`, `for`, `while`) entraînera une erreur de syntaxe. Ces deux points sont essentiels pour signaler à Python le début d'un bloc d'instructions.

2.2 Les conditions en Python

Lorsque vous souhaitez exécuter du code uniquement si une certaine condition est remplie, vous pouvez utiliser une déclaration conditionnelle :

- **Un simple "if"**, par exemple :

```
x = 10

if x == 10:  # condition
        print("x vaut 10")  # Exécuté si la condition est vraie.
```

- **Une série de déclarations "if"**, par exemple :

```
x = 10

if x > 5:  # condition 1
        print("x est superieur a 5")  # Exécuté si la condition 1
            est vraie.

if x < 10:  # condition 2
        print("x est inferieur a 10")  # Exécuté si la condition
            2 est vraie.

if x == 10:  # condition 3
        print("x est egale a 10")  # Exécuté si la condition 3
            est vraie.
```

Chaque déclaration "if" est testée séparément.

- **Une déclaration "if-else"**, par exemple :

```
x = 10

if x < 10:  # condition
        print("x est inferieur a 10")  # Exécuté si la condition
            est vraie.
else:
        print("x est superieur ou egal a 10")  # Exécuté si la
            condition est fausse.
```

- **Une série de déclarations "if" suivies d'un "else"**, par exemple :

```
x = 10

if x > 5:  # condition 1
        print("x est superieur a 5")  # Exécuté si la condition 1
            est vraie.

if x < 10:  # condition 2
        print("x est inferieur a 10")  # Exécuté si la condition
            2 est vraie.
else :
        print(x est egale a 10) # Exécuté si la condition 2 est
            fausse.
```

Chaque "if" est testé séparément. Le bloc "else" est exécuté si le dernier "if" est faux.

- **La déclaration "if-elif-else"**, par exemple :

```
x = 10

if x == 10:  # Vrai
        print("x == 10")

if x > 15:  # Faux
        print("x > 15")

elif x > 10:  # Faux
        print("x > 10")

elif x > 5:  # Vrai
        print("x > 5")

else:
        print("else ne sera pas execute")
```

Si la condition du "if" est fausse, le programme vérifie les conditions des blocs "elif" suivants. Le premier "elif" dont la condition est vraie est exécuté. Si toutes les conditions sont fausses, le bloc "else" sera exécuté.

- **Les conditions imbriquées**, par exemple :

```
x = 10

if x > 5:  # Vrai
        if x == 6:  # Faux
                print("x == 6")
        elif x == 10:  # Vrai
                print("x == 10")
```

```
        else:
                print("else")
else:
        print("else")
```

2.3 Les boucles en Python

2.3.1 Les types de boucles

Il existe deux types de boucles en Python : **while** et **for**.

- La boucle **while** exécute une ou plusieurs instructions tant qu'une condition booléenne spécifiée est vraie, par exemple :

```
# Exemple 1
while True:
        print("boucle infinie.")

# Exemple 2
compteur = 5
while compteur > 2:
        print(compteur)
        compteur -= 1
```

- La boucle **for** exécute un ensemble d'instructions plusieurs fois. Elle est utilisée pour itérer sur une séquence (par exemple, une liste, un dictionnaire, un tuple ou un ensemble) ou d'autres objets itérables (par exemple, des chaînes de caractères). Vous pouvez utiliser la boucle **for** pour itérer sur une séquence de nombres en utilisant la fonction intégrée `range`. Voici quelques exemples :

```
# Exemple 1
mot = "Python"
for lettre in mot:
        print(lettre, end="*")

# Exemple 2
for i in range(1, 10):
        if i % 2 == 0:
        print(i)
```

2.3.2 Les instructions `break` et `continue`

Vous pouvez utiliser les instructions `break` et `continue` pour modifier le flux d'exécution d'une boucle :

- `break` est utilisé pour sortir d'une boucle, par exemple :

```
texte = "OpenEDG Python Institute"
for lettre in texte:
        if lettre == "P":
                break
        print(lettre, end="")
```

- `continue` est utilisé pour passer à l'itération suivante en sautant l'itération en cours, par exemple :

```
texte = "pyxpyxpyx"
for lettre in texte:
        if lettre == "x":
                continue
        print(lettre, end="")
```

2.3.3 Les clauses `else` dans les boucles

Les boucles `while` et `for` peuvent également comporter une clause `else` en Python. La clause `else` est exécutée après la fin de la boucle, tant que celle-ci n'a pas été interrompue par une instruction `break`. Voici quelques exemples :

```
n = 0

while n != 3:
        print(n)
        n += 1
else:
        print(n, "else")

print()

for i in range(0, 3):
        print(i)
else:
        print(i, "else")
```

3 Les fonctions en Python

3.1 Qu'est-ce qu'une fonction ?

Une fonction est un bloc de code qui exécute une tâche spécifique lorsqu'elle est appelée (invoquée). Vous pouvez utiliser les fonctions pour rendre votre code réutilisable, mieux organisé et plus lisible. Les fonctions peuvent avoir des paramètres et retourner des valeurs.

3.2 Les types de fonctions en Python

Il existe au moins quatre types de fonctions en Python :

- **Les fonctions intégrées** : elles font partie intégrante de Python (comme la fonction `print()`). Vous pouvez consulter la liste complète des fonctions intégrées de Python à l'adresse suivante : `https://docs.python.org/3/library/functions.html`.
- **Les fonctions provenant de modules préinstallés** : comme les modules `math, numpy, matplotlib`.
- **Les fonctions définies par l'utilisateur** : il s'agit des fonctions que vous écrivez vous-même et que vous pouvez utiliser librement dans votre code.
- **Les fonctions lambda** : Les fonctions *lambda* en Python sont des fonctions anonymes, c'est-à-dire des fonctions qui n'ont pas de nom explicite comme celles définies avec `def`. Elles sont utiles pour des tâches simples et rapides, généralement définies en une seule ligne. Les fonctions *lambda* peuvent prendre un nombre quelconque d'arguments, mais ne peuvent contenir qu'une seule expression.
  Syntaxe d'une fonction *lambda*

```
lambda arguments: expression
```

  Exemple d'utilisation :

```
# Définition d'une fonction lambda qui double un nombre
double = lambda x: x * 2

# Appel de la fonction lambda
print(double(5))  # Résultat :  10
```

  Ici, `lambda x: x * 2` définit une fonction qui prend un argument `x` et retourne `x` multiplié par 2.

3.3 Comment définir une fonction ?

Vous pouvez définir votre propre fonction en utilisant le mot-clé `def` et la syntaxe suivante :

```
def votre_fonction(parametres_optionnels):
        # le corps de la fonction
```

3.3.1 Une fonction sans arguments

Vous pouvez définir une fonction qui ne prend aucun argument, par exemple :

```
def message():     # définition d'une fonction
        print("Bonjour")     # corps de la fonction

message()     # appel de la fonction
```

Dans cet exemple, la fonction `message` affiche simplement le mot "Bonjour" lorsque la fonction est appelée.

3.3.2 Une fonction avec des arguments

Vous pouvez également définir une fonction qui prend des arguments, comme dans l'exemple suivant d'une fonction avec un paramètre :

```
def bonjour(nom):     # définition d'une fonction
        print("Bonjour,", nom)     # corps de la fonction

nom = input("Entrez votre nom : ")

bonjour(nom)     # appel de la fonction
```

Dans cet exemple, la fonction `bonjour` prend un argument `nom` et affiche un message personnalisé avec ce nom.

Exemples de définition de fonctions avec plusieurs arguments

```
# Exemple 1 :  Fonction avec deux arguments
def hi_all(name_1, name_2):
        print("Salut,", name_2)
        print("Salut,", name_1)

hi_all("Sebastian", "Konrad")  # Appel avec deux arguments

# Exemple 2 :  Fonction avec trois arguments
def address(street, city, postal_code):
        print("Votre adresse est :", street, "St.,", city,
            postal_code)

s = input("Rue : ")
p_c = input("Code Postal : ")
c = input("Ville : ")
address(s, c, p_c)  # Appel avec trois arguments
```

Types d'arguments : Il existe plusieurs manières de passer des arguments à une fonction :

- **Passage d'arguments positionnels**, où l'ordre des arguments compte. (Exemple 1)

- **Passage d'arguments par mots-clés (nommés)**, où l'ordre des arguments n'a pas d'importance. (Exemple 2)
- **Mixage des deux types d'arguments**, avec des arguments positionnels suivis d'arguments nommés. (Exemple 3)

Exemples

```
# Exemple 1 : Arguments positionnels
def subtra(a, b):
        print(a - b)

subtra(5, 2)     # Résultat : 3
subtra(2, 5)     # Résultat : -3

# Exemple 2 : Arguments nommés
def subtra(a, b):
        print(a - b)

subtra(a=5, b=2)     # Résultat : 3
subtra(b=2, a=5)     # Résultat : 3

# Exemple 3 : Mixage des deux
def subtra(a, b):
        print(a - b)

subtra(5, b=2)     # Résultat : 3
subtra(5, 2)     # Résultat : 3

# Erreur si argument positionnel après un argument nommé
subtra(a=5, 2)     # SyntaxError
```

Il est important de se rappeler que les arguments positionnels ne doivent pas suivre les arguments nommés.

Arguments avec valeurs par défaut : Il est également possible de prédéfinir une valeur pour un argument :

```
def name(first_name, last_name="Dupont"):
        print(first_name, last_name)

name("Andy")     # Résultat : Andy Dupont
name("Betty", "Johnson")     # Résultat : Betty Johnson
```

3.4 Retourner un résultat d'une fonction

Vous pouvez utiliser le mot-clé `return` pour indiquer à une fonction de renvoyer une valeur. L'instruction `return` permet également de sortir de la fonction. Par exemple :

```
def multiply(a, b):
        return a * b

print(multiply(3, 4))  # Affiche :  12
```

Si aucune valeur n'est spécifiée après `return`, la fonction renverra `None` :

```
def multiply(a, b):
        return

print(multiply(3, 4))  # Affiche :  None
```

Le résultat d'une fonction peut être facilement affecté à une variable, comme dans cet exemple :

```
def wishes():
        return "Joyeux Anniversaire !"

w = wishes()
print(w)  # Affiche :  Joyeux Anniversaire !
```

Voici deux exemples montrant la différence entre un appel de fonction qui renvoie une valeur et un autre qui affiche directement un résultat :

Exemple 1 :

```
def wishes():
        print("Mes Voeux")
        return "Joyeux Anniversaire"

wishes()  # Affiche :  Mes Voeux
```

Exemple 2 :

```
def wishes():
        print("Mes Voeux")
        return "Joyeux Anniversaire"

print(wishes())
# Affiche :
# Mes Voeux
# Joyeux Anniversaire
```

Vous pouvez également passer une liste en tant qu'argument d'une fonction :

```
def hi_everybody(my_list):
        for name in my_list:
                print("Salut,", name)

hi_everybody(["Adam", "Jean", "Lucie"])
```

De plus, une liste peut aussi être le résultat retourné par une fonction :

```
def create_list(n):
        my_list = []
        for i in range(n):
                my_list.append(i)
        return my_list

print(create_list(5))  # Affiche :  [0, 1, 2, 3, 4]
```

3.5 Portées en Python

1. Une variable qui existe en dehors d'une fonction a une portée à l'intérieur du corps de la fonction (Exemple 1) à moins que la fonction ne définisse une variable du même nom (Exemple 2 et Exemple 3) :

Exemple 1 :

```
var = 2

def mult_by_var(x):
        return x * var

print(mult_by_var(7))  # Affiche :  14
```

Exemple 2 :

```
def mult(x):
        var = 5
        return x * var

print(mult(7))  # Affiche :  35
```

Exemple 3 :

```
def mult(x):
        var = 7
        return x * var

var = 3
print(mult(7))  # Affiche :  49
```

2. Une variable qui existe à l'intérieur d'une fonction a une portée à l'intérieur du corps de la fonction (Exemple 4) :

Exemple 4 :

```
def adding(x):
        var = 7
        return x + var

print(adding(4))  # Affiche :  11
print(var)        # NameError
```

3. Vous pouvez utiliser le mot-clé `global` suivi du nom d'une variable pour rendre la portée de la variable globale :

```
var = 2
print(var)  # Affiche :  2

def return_var():
        global var
        var = 5
        return var

print(return_var())  # Affiche :  5
print(var)           # Affiche :  5
```

3.6 Récursivité en Python

1. Une fonction peut appeler d'autres fonctions, ou même elle-même. Lorsqu'une fonction s'appelle elle-même, cette situation est connue sous le nom de récursivité. La fonction qui s'appelle elle-même et contient une condition d'arrêt spécifiée (c'est-à-dire le cas de base — une condition qui ne demande pas à la fonction de faire d'autres appels à cette fonction) est appelée une fonction récursive.
2. Vous pouvez utiliser des fonctions récursives en Python pour écrire un code propre et élégant, et le diviser en morceaux organisés. D'un autre côté, il faut être très prudent, car il est facile de commettre une erreur et de créer une fonction qui ne se termine jamais. Vous devez également vous rappeler que les appels récursifs consomment beaucoup de mémoire et peuvent donc parfois être inefficaces.

Lorsque vous utilisez la récursivité, il est important de prendre en compte tous ses avantages et inconvénients.
La fonction factorielle est un exemple classique de la façon dont le concept de récursivité peut être mis en pratique. Voici un exemple d'une fonction factorielle sans et avec récursivité :

```
#Implémentation itérative de la fonction factorielle
def factorial_iterative(n):
        result = 1
        for i in range(1, n + 1):
                result *= i
                return result

print(factorial_iterative(4))  # Affiche :  24 (4 * 3 * 2 * 1)

# Implémentation récursive de la fonction factorielle.
def factorial(n):
        if n == 1:     # Le cas de base (condition d'arrêt).
                return 1
        else:
                return n * factorial(n - 1)
```

```
print(factorial(4))  # Affiche :  24 (4 * 3 * 2 * 1)
```

Dans cet exemple, la version récursive de la fonction factorielle utilise des appels de fonction pour calculer le produit, tandis que la version itérative utilise une boucle. La récursivité permet souvent d'écrire du code plus concis et plus lisible, mais elle peut entraîner une utilisation plus élevée de la mémoire et des risques d'appels infinis si la condition d'arrêt n'est pas correctement définie.

4 Modules en Python

4.1 Importation des modules

1. Si vous souhaitez importer un module dans son intégralité, vous pouvez le faire en utilisant l'instruction `import module_name`. Vous pouvez importer plusieurs modules en même temps en utilisant une liste séparée par des virgules. Par exemple :

```
import mod1
import mod2, mod3, mod4
```

Bien que cette dernière forme ne soit pas recommandée pour des raisons de style, il est préférable et plus esthétique d'exprimer la même intention de manière plus explicite, comme ceci :

```
import mod2
import mod3
import mod4
```

2. Si un module est importé de cette manière et que vous souhaitez accéder à l'une de ses entités, vous devez préfixer le nom de l'entité en utilisant la notation par point. Par exemple :

```
import my_module

result = my_module.my_function(my_module.my_data)
```

Le code ci-dessus utilise deux entités provenant du module `my_module` : une fonction nommée `my_function()` et une variable nommée `my_data`. Les deux noms doivent être préfixés par `my_module`. Aucun des noms des entités importées ne doit entrer en conflit avec les noms identiques existant dans l'espace de noms de votre code.

3. Vous pouvez non seulement importer un module dans son intégralité, mais aussi importer uniquement des entités individuelles à partir de celui-ci. Dans ce cas, les entités importées ne doivent pas être préfixées lors de leur utilisation. Par exemple :

```
from module import my_function, my_data

result = my_function(my_data)
```

Cette méthode, bien qu'attirante, n'est pas recommandée en raison du risque de conflits avec des noms provenant de l'espace de noms du code.
4. La forme la plus générale de la déclaration ci-dessus vous permet d'importer toutes les entités offertes par un module :

```
from my_module import *

result = my_function(my_data)
```

Notez que cette variante d'importation n'est pas recommandée pour les mêmes raisons que précédemment (le risque de conflit de noms est encore plus dangereux ici).
5. Vous pouvez changer le nom de l'entité importée "à la volée" en utilisant la phrase `as` de l'importation. Par exemple :

```
from module import my_function as fun, my_data as dat

result = fun(dat)
```

4.2 Fonctions sélectionnées du module math

Commençons par un aperçu rapide de certaines des fonctions fournies par le module `math`. Nous les avons choisies de manière arbitraire, mais cela ne signifie pas que les fonctions que nous n'avons pas mentionnées ici sont moins importantes.

Le premier groupe de fonctions du module `math` est lié à la trigonométrie :

- `sin(x)` → le sinus de x ;
- `cos(x)` → le cosinus de x ;
- `tan(x)` → la tangente de x.

Toutes ces fonctions prennent un argument (une mesure d'angle exprimée en radians) et retournent le résultat approprié (attention avec `tan()` - tous les arguments ne sont pas acceptés).

Bien sûr, il existe également leurs versions inversées :

- `asin(x)` → l'arcsinus de x ;
- `acos(x)` → l'arccosinus de x ;
- `atan(x)` → l'arctangente de x.

Ces fonctions prennent un argument et retournent une mesure d'angle en radians.

Pour travailler efficacement avec les mesures d'angle, le module `math` vous fournit les entités suivantes :

- `pi` → une constante dont la valeur est une approximation de π ;

- `radians(x)` → une fonction qui convertit x des degrés en radians ;
- `degrees(x)` → agissant dans l'autre sens (des radians vers les degrés).

Prédir les résultats de ce code :

```
from math import pi, radians, degrees, sin, cos, tan, asin

ad = 90
ar = radians(ad)
ad = degrees(ar)

print(ad == 90.)
print(ar == pi / 2.)
print(sin(ar) / cos(ar) == tan(ar))
print(asin(sin(ar)) == ar)
```

En plus des fonctions circulaires (listées ci-dessus), le module `math` contient également un ensemble de leurs analogues hyperboliques :

- `sinh(x)` → le sinus hyperbolique ;
- `cosh(x)` → le cosinus hyperbolique ;
- `tanh(x)` → la tangente hyperbolique ;
- `asinh(x)` → l'arcsinus hyperbolique ;
- `acosh(x)` → l'arccosinus hyperbolique ;
- `atanh(x)` → l'arctangente hyperbolique.

Un autre groupe de fonctions du module `math` est formé par des fonctions liées à l'exponentiation :

- `e` → une constante dont la valeur est une approximation du nombre d'Euler (e) ;
- `exp(x)` → trouver la valeur de e^x ;
- `log(x)` → le logarithme naturel de x ;
- `log(x, b)` → le logarithme de x à la base b ;
- `log10(x)` → le logarithme décimal de x (plus précis que `log(x, 10)`) ;
- `log2(x)` → le logarithme binaire de x (plus précis que `log(x, 2)`).

Remarque : la fonction `pow()` :

- `pow(x, y)` → trouver la valeur de x^y.

C'est une fonction intégrée et n'a pas besoin d'être importée.

```
from math import e, exp, log

print(pow(e, 1) == exp(log(e)))
print(pow(2, 2) == exp(2 * log(2)))
print(log(e, e) == exp(0))
```

Le dernier groupe comprend quelques fonctions à usage général telles que :

- `ceil(x)` → le plafond de x (le plus petit entier supérieur ou égal à x) ;
- `floor(x)` → le plancher de x (le plus grand entier inférieur ou égal à x) ;
- `trunc(x)` → la valeur de x tronquée à un entier (attention - ce n'est pas équivalent ni à `ceil` ni à `floor`) ;
- `factorial(x)` → retourne $x!$ (x doit être un entier et non négatif) ;
- `hypot(x, y)` → retourne la longueur de l'hypoténuse d'un triangle rectangle dont les longueurs des côtés sont égales à x et y (équivalent à `sqrt(pow(x, 2) + pow(y, 2))` mais plus précis).

Examinez l'exemple suivant :

```
from math import ceil, floor, trunc

x = 1.4
y = 2.6

print(floor(x), floor(y))
print(floor(-x), floor(-y))
print(ceil(x), ceil(y))
print(ceil(-x), ceil(-y))
print(trunc(x), trunc(y))
print(trunc(-x), trunc(-y))
```

Ce programme démontre les différences fondamentales entre `ceil()`, `floor()` et `trunc()`.

4.3 Fonctions sélectionnées du module NumPy

Le module `NumPy` est l'une des bibliothèques les plus populaires pour le calcul numérique en Python. Il fournit une multitude de fonctions puissantes pour effectuer des opérations sur des tableaux multidimensionnels et des matrices. Explorons quelques-unes des fonctions et caractéristiques les plus utiles de `NumPy`.

L'une des principales caractéristiques de `NumPy` est sa capacité à créer et manipuler des tableaux. Voici quelques fonctions couramment utilisées pour créer des tableaux :

- `numpy.array(object)` → crée un tableau à partir d'une liste ou d'une autre structure de données.

- `numpy.zeros(shape)` → crée un tableau rempli de zéros de la forme spécifiée.
- `numpy.ones(shape)` → crée un tableau rempli de uns de la forme spécifiée.
- `numpy.arange(start, stop, step)` → crée un tableau contenant une séquence de nombres, en spécifiant la valeur de départ, d'arrêt et l'incrément.
- `numpy.linspace(start, stop, num)` → génère un tableau contenant un nombre spécifié de valeurs également espacées entre les valeurs de départ et d'arrêt.

Voici un exemple de création de tableaux avec `NumPy` :

```
import numpy as np

# Création d'un tableau à partir d'une liste
a = np.array([1, 2, 3, 4])
print(a)

# Création d'un tableau rempli de zéros
b = np.zeros((2, 3))
print(b)

# Création d'un tableau rempli de uns
c = np.ones((2, 2))
print(c)

# Création d'un tableau avec une séquence de nombres
d = np.arange(0, 10, 2)
print(d)

# Création d'un tableau avec des valeurs également espacées
e = np.linspace(0, 1, 5)
print(e)
```

Une autre fonctionnalité puissante de `NumPy` est la capacité de réaliser des opérations mathématiques sur des tableaux. Voici quelques-unes des fonctions mathématiques les plus utiles :

- `numpy.add(x1, x2)` → additionne deux tableaux élément par élément.
- `numpy.subtract(x1, x2)` → soustrait le deuxième tableau du premier élément par élément.
- `numpy.multiply(x1, x2)` → multiplie deux tableaux élément par élément.
- `numpy.divide(x1, x2)` → divise le premier tableau par le deuxième élément par élément.
- `numpy.mean(array)` → calcule la moyenne des éléments d'un tableau.

- `numpy.sum(array)` → calcule la somme des éléments d'un tableau.
- `numpy.std(array)` → calcule l'écart type des éléments d'un tableau.

Voici un exemple illustrant certaines de ces opérations :

```
# Définition de deux tableaux
x = np.array([1, 2, 3])
y = np.array([4, 5, 6])

# Addition des tableaux
addition = np.add(x, y)
print('Addition:', addition)

# Soustraction des tableaux
soustraction = np.subtract(x, y)
print('Soustraction:', soustraction)

# Multiplication des tableaux
multiplication = np.multiply(x, y)
print('Multiplication:', multiplication)

# Division des tableaux
division = np.divide(x, y)
print('Division:', division)

# Calcul de la moyenne
moyenne = np.mean(x)
print('Moyenne de x:', moyenne)

# Calcul de la somme
somme = np.sum(x)
print('Somme de x:', somme)

# Calcul de l'écart type
ecart_type = np.std(x)
print('Ecart type de x:', ecart_type)
```

En plus de ces fonctions, `NumPy` offre également des outils pour effectuer des opérations sur des matrices. Par exemple :

- `numpy.dot(a, b)` → calcule le produit matriciel de a et b.
- `numpy.transpose(a)` → renvoie la transposée du tableau a.
- `numpy.linalg.inv(a)` → calcule l'inverse de la matrice a (si elle est inversible).

Voici un exemple d'opérations sur des matrices :

```
# Création de deux matrices
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])

# Produit matriciel
produit_mat = np.dot(A, B)
print('Produit matriciel A * B:\n', produit_mat)

# Transposée de la matrice A
transpose_A = np.transpose(A)
print('Transposee de A:\n', transpose_A)

# Inverse de la matrice A
inverse_A = np.linalg.inv(A)
print('Inverse de A:\n', inverse_A)
```

Le module `NumPy` offre une grande variété de fonctionnalités qui facilitent le calcul numérique et l'analyse de données. Pour une exploration plus approfondie, consultez la documentation officielle de `NumPy` et découvrez d'autres fonctions qui peuvent répondre à vos besoins spécifiques.

4.4 Le module `matplotlib`

Le module `matplotlib` est une bibliothèque essentielle pour créer des visualisations de données en Python. Elle permet de générer des graphiques en 2D de manière simple et efficace. `Matplotlib` est très polyvalente et peut être utilisée pour produire des figures, des graphiques en lignes, des histogrammes, des diagrammes en nuage de points, et bien plus encore.

La sous-bibliothèque la plus couramment utilisée de `matplotlib` est `pyplot`, qui fournit une interface simple pour créer et manipuler des graphiques.

Voici un exemple simple montrant comment utiliser `matplotlib` pour tracer un graphique linéaire :

```
import matplotlib.pyplot as plt
import numpy as np

# Données de l'exemple
x = np.linspace(0, 10, 100)
y = np.sin(x)

# Création de la figure et de l'axe
plt.plot(x, y, label="sin(x)")

# Ajout du titre et des légendes
plt.title("Graphique de la fonction sinus")
plt.xlabel("x")
```

```
plt.ylabel("sin(x)")
plt.legend()

# Affichage du graphique
plt.show()
```

Dans cet exemple :

- `plt.plot(x, y)` trace la courbe de la fonction `sin(x)` sur l'intervalle défini par `x`.
- `plt.title()` ajoute un titre au graphique.
- `plt.xlabel()` et `plt.ylabel()` définissent les étiquettes des axes.
- `plt.legend()` affiche la légende correspondant à la courbe tracée.
- `plt.show()` affiche la figure générée.

`Matplotlib` offre également la possibilité de personnaliser les couleurs, les types de lignes, les marques sur les points de données, et bien plus encore. Vous pouvez, par exemple, changer la couleur et le style de ligne avec :

```
plt.plot(x, y, color='red', linestyle='--', label="sin(x)")
```

D'autres types de graphiques peuvent également être créés, tels que les histogrammes, les diagrammes en barres, ou les nuages de points :

```
# Histogramme
data = np.random.randn(1000)
plt.hist(data, bins=30, alpha=0.5)
plt.show()

# Nuage de points
x = np.random.rand(50)
y = np.random.rand(50)
plt.scatter(x, y)
plt.show()
```

Enfin, `matplotlib` permet d'enregistrer les graphiques dans différents formats (PNG, PDF, etc.) avec `plt.savefig()` :

```
plt.savefig("graphique.png")
```

4.5 Installation des bibliothèques dans Jupyter Notebook

Pour utiliser des bibliothèques comme `matplotlib` ou `numpy` dans un `Jupyter Notebook`, vous devez d'abord vous assurer que ces bibliothèques sont installées dans votre environnement Python.

Si vous travaillez dans un `Jupyter Notebook`, vous pouvez installer des bibliothèques directement depuis une cellule en utilisant la commande suivante :

```
!pip install matplotlib numpy
```

Cela installera les bibliothèques `matplotlib` et `numpy` dans l'environnement utilisé par le notebook. Vous pouvez remplacer `matplotlib` et `numpy` par d'autres bibliothèques que vous souhaitez installer.

Après l'installation, vous pouvez importer les bibliothèques dans vos cellules et les utiliser immédiatement :

```
import matplotlib.pyplot as plt
import numpy as np
```

Si les bibliothèques sont déjà installées dans votre environnement Python, vous n'avez pas besoin de répéter l'installation et vous pouvez directement commencer à les utiliser dans votre code.

5 Annexe

Nous présentons des tableaux récapitulatifs des différentes opérations que l'on peut effectuer sur les listes ou les autres structures de données représentant des séquences, les sets ainsi que sur les dictionnaires.

Tableau 2: Opérations sur les séquences en Python

Opération	**Description**
`s[i]`	Accéder à l'élément `i` de la séquence `s` (la numérotation commence à 0). Si `i` est un nombre négatif, on accède aux éléments par la fin de la séquence (-1 est l'indice du dernier élément).
`len(s)`	Renvoie le nombre d'éléments de la séquence `s`.
`s + m`	Concatène les séquences `s` et `m` et crée une nouvelle séquence.
`s * n`	Le paramètre `n` doit être un entier ; dans ce cas, itère l'opération d'addition (concaténation) `n` fois sur la séquence, c'est-à-dire que la séquence `s` est concaténée `n` fois avec elle-même.
`min(s)`, `max(s)`	Retourne la valeur du plus petit/grand élément de la séquence `s`.
`sum(s)`	Calcule la somme des éléments de `s`.
`s.index(k)`	Recherche l'élément `k` dans la séquence `s`, et renvoie la position de sa première occurrence.
`s.count(k)`	Compte le nombre d'occurrences de `k` dans la séquence `s`.
`k in s`	Renvoie vrai ou faux selon que l'élément `k` soit dans la séquence `s` ou non.
`sorted(s)`	Renvoie une nouvelle séquence triée de façon croissante, composée des éléments de `s`. Un second paramètre permet de trier dans l'ordre décroissant.

Tableau 3: Exemples d'opération de découpage (slicing) des listes

Code	**Description**	**Résultat sur la liste `l`=[0, 1, ..., 15]**
`l[3:8]`	Extraction des éléments d'indice 3 à 7	[3, 4, 5, 6, 7]
`l[:5]`	Extraction des cinq premiers éléments de la liste	[0, 1, 2, 3, 4]
`l[5:]`	Extraction de tous les éléments sauf les 5 premiers	[5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
`l[:-5]`	Extraction de tous les éléments jusqu'aux 5 derniers	[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
`l[-5:]`	Extraction des 5 derniers éléments	[11, 12, 13, 14, 15]
`l[3:12:3]`	Extraction d'un élément sur 3 à partir de l'indice 3, jusqu'à l'indice 11	[3, 6, 9]
`l[::2]`	Extraction d'un élément sur deux à partir du premier	[0, 2, 4, 6, 8, 10, 12, 14]
`l[::-2]`	Extraction d'un élément sur deux à partir de la fin	[15, 13, 11, 9, 7, 5, 3, 1]
`l[10::-2]`	Extraction d'un élément sur deux à partir de l'indice 10	[10, 8, 6, 4, 2, 0]
`l[:5:-2]`	Extraction d'un élément sur deux à partir de la fin, jusqu'à l'indice 5 (exclu)	[15, 13, 11, 9, 7]
`l[::-1]`	Inversion de la séquence	[15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

Tableau 4: Opérations sur les sets

Code	**Description**
`s.add(15)`	Ajoute l'élément 15 à la collection. Cette fonction ne fait rien si l'élément est déjà présent.
`s.discard(8)`	Retire l'élément 8 de la collection. Cette fonction ne fait rien si l'élément n'est pas présent.
`s.remove(8)`	Réalise la même opération que `discard`, mais produit une erreur si l'élément n'est pas présent.
`s.clear()`	Vide la collection de tous ses éléments.
`s | f` ou `s.union(f)`	Réalise l'union des collections `s` et `f`.
`s & f` ou `s.intersection(f)`	Réalise l'intersection des collections `s` et `f`.
`s - f` ou `s.difference(f)`	Retire de la collection `s` tous les éléments de `f`.
`s ^ f` ou `s.symmetric_difference(f)`	L'ensemble des éléments dans `s` ou `f`, mais pas dans `s` et `f`.
`k in s`	Renvoie vrai ou faux selon que l'élément `k` soit dans l'ensemble `s` ou non.

Tableau 5: Opérations sur les dictionnaires

Code	**Description**
`a.get(key)`	Récupère la valeur associée à la clef `key`, et renvoie `None` si cette clef n'existe pas. Se comporte comme `a[key]`, sauf que `get` ne produit pas d'exception en cas de clef absente.
`a.keys()`	Renvoie une liste de toutes les clefs présentes dans le dictionnaire.
`a.values()`	Renvoie une liste de toutes les valeurs présentes dans le dictionnaire.
`a.items()`	Renvoie une liste de toutes les paires (clef, valeur) présentes dans le dictionnaire. Chaque paire est représentée par un tuple.
`k in a`	Renvoie vrai ou faux selon que la clef `k` soit dans le dictionnaire `a` ou non.
`e & f` ou `e.intersection(f)`	Réalise l'intersection des collections `e` et `f`.
`e - f` ou `e.difference(f)`	Retire de la collection `e` tous les éléments de `f`.
`e ^ f` ou `e.symmetric_difference(f)`	L'ensemble des éléments dans `e` ou `f`, mais pas dans `e` et `f`.