



# Les Dictionnaires et Fichiers JSON en Python

Maîtrisez les structures de données clés et la manipulation de fichiers JSON pour vos projets SLAM

Cours de David DONISA , Enseignant BTS SIO

# Introduction aux Dictionnaires Python

## Qu'est-ce qu'un dictionnaire ?

Un dictionnaire est une structure de données Python qui stocke des paires **clé-valeur**. C'est l'équivalent d'un objet JavaScript ou d'une HashMap en Java. Chaque clé est unique et permet d'accéder rapidement à sa valeur associée.

Les dictionnaires sont **mutables, non ordonnés** (avant Python 3.7) et extrêmement performants pour la recherche de données. Ils constituent un outil fondamental pour tout développeur Python.

## Caractéristiques essentielles

- Clés uniques (pas de doublons)
- Clés immuables (int, str, tuple)
- Valeurs de tout type
- Accès en temps constant  $O(1)$
- Syntaxe avec accolades {}



# Créer et Manipuler des Dictionnaires

01

## Création

Utilisez les accolades {} pour créer un dictionnaire vide ou avec des données initiales

02

## Accès aux valeurs

Utilisez la notation entre crochets dict[clé] pour récupérer une valeur

03

## Ajout d'éléments

Assignez une valeur à une nouvelle clé : dict['nouvelle\_clé'] = valeur

04

## Modification

Réaffectez simplement une valeur à une clé existante pour la modifier

```
mon_dict = {}
mon_dict_a = {'test': 'Texte de test', 1: 25}
print(mon_dict_a[1]) # Affiche: 25
mon_dict_a['blabla'] = 'Nouvelle valeur'
```

# Méthodes Essentielles des Dictionnaires



.keys()

Retourne une vue itérable de toutes les clés du dictionnaire. Parfait pour parcourir uniquement les clés dans une boucle `for`.

```
for k in mon_dict.keys():
    print(k)
```



.values()

Retourne une vue itérable de toutes les valeurs. Utilisez-la quand seules les valeurs vous intéressent, sans les clés.

```
print(mon_dict.values())
```



.items()

Retourne des tuples (clé, valeur) pour chaque paire. La méthode la plus utilisée pour itérer sur un dictionnaire complet.

```
for key, value in dict.items():
    print(f'{key}: {value}')
```

# Supprimer et Fusionner des Éléments

## Méthodes de suppression

Python offre deux méthodes principales pour supprimer des éléments d'un dictionnaire :

**L'instruction** `del` supprime directement une clé. Attention : elle lève une exception `KeyError` si la clé n'existe pas.

```
del mon_dict_a['test']
```

**La méthode** `.pop()` supprime et retourne la valeur associée à la clé. Vous pouvez fournir une valeur par défaut pour éviter les erreurs.

```
valeur = mon_dict_a.pop(1)  
# Avec défaut:  
val = dict.pop('clé', None)
```

## Fusion de dictionnaires

La méthode `.update()` fusionne deux dictionnaires. Les valeurs du dictionnaire source écrasent celles du dictionnaire cible en cas de clés identiques.

```
mon_dict_a.update({2: "Valeur"})
```



### ⚠️ Attention aux exceptions

Toujours vérifier l'existence d'une clé avec `if 'clé' in dict:` avant d'utiliser `del` ou préférer `.pop()` avec une valeur par défaut.



# Introduction au Format JSON

## Qu'est-ce que JSON ?

**JSON** (JavaScript Object Notation) est un format de données textuelles léger et lisible. Il est devenu le standard pour l'échange de données entre applications web, API REST et systèmes distribués.

JSON structure les données avec des paires clé-valeur, similaires aux dictionnaires Python. Cette correspondance naturelle facilite grandement la manipulation de données JSON en Python.

## Pourquoi utiliser JSON ?

- Format universel et interopérable
- Lisible par les humains et machines
- Léger et performant
- Support natif dans Python
- Idéal pour les API et configurations

# Le Module JSON en Python



## Import du module

Commencez par importer le module standard `json` qui contient toutes les fonctions nécessaires

```
import json
```



## Lecture de fichiers

Utilisez `.load()` pour charger un fichier JSON directement en dictionnaire Python

```
with open('file.json', 'r') as f:  
    data = json.load(f)
```



## Écriture de fichiers

Utilisez `.dump()` pour sauvegarder un dictionnaire dans un fichier JSON

```
with open('file.json', 'w') as f:  
    json.dump(mon_dict, f, indent=4)
```

Le paramètre `indent=4` rend le fichier JSON formaté et lisible. C'est une bonne pratique pour la maintenance et le débogage.

# Manipuler JSON : Fichiers et Chaînes

## Manipulation de fichiers JSON

Les fonctions `json.load()` et `json.dump()` permettent de lire et écrire directement dans des fichiers JSON.

### `json.load()`

Lit un fichier JSON et retourne un dictionnaire Python.

### `json.dump()`

Écrit un dictionnaire Python dans un fichier JSON.

Il existe deux principales méthodes pour gérer l'ouverture et la fermeture des fichiers :

### Méthode 1 : Utilisation manuelle de `open()` et `.close()`

Cette méthode offre un contrôle explicite, mais nécessite de fermer le fichier manuellement, ce qui peut entraîner des problèmes si la fermeture est oubliée ou en cas d'erreur d'exécution.

```
file = open('file.json', 'r')
data = json.load(file)
file.close()
```

```
# Exemple d'écriture
my_dict = {"name": "Alice", "age": 30}
file = open('output.json', 'w')
json.dump(my_dict, file, indent=4)
file.close()
```

### Méthode 2 : Utilisation du gestionnaire de contexte `with` (Recommandé)

Le mot-clé `with` assure que le fichier est automatiquement fermé après que le bloc de code soit exécuté, même en cas d'exceptions. C'est la méthode la plus sûre et recommandée pour la gestion des fichiers. Un gestionnaire de contexte est un mécanisme Python qui garantit l'exécution de code de nettoyage (comme la fermeture de fichiers) même en cas d'erreur.

```
with open('file.json', 'r') as f:
    data = json.load(f)
```

```
# Exemple d'écriture
my_dict = {"name": "Bob", "age": 25}
with open('output.json', 'w') as f:
    json.dump(my_dict, f, indent=4)
```

## Manipulation de chaînes JSON

Les fonctions `json.loads()` et `json.dumps()` sont utilisées pour convertir des chaînes de caractères JSON en dictionnaires Python et vice-versa.

### `json.dumps()`

Convertit un dictionnaire Python en chaîne JSON (le 's' signifie 'string'). Cette opération s'appelle la **sérialisation** : elle transforme un objet Python en une représentation textuelle qui peut être stockée ou transmise. Utilisé pour envoyer des données à une API web, sauvegarder en base de données, ou créer des logs structurés. La **sérialisation** permet de convertir des structures de données complexes en format texte pour le stockage ou la transmission, tandis que la **désérialisation** (avec `json.loads()`) fait l'opération inverse.

### `json.loads()`

Parse une chaîne JSON et retourne un dictionnaire Python. Parser signifie analyser et convertir une chaîne de caractères selon des règles syntaxiques précises. Utilisé pour traiter des réponses d'API, lire des configurations depuis des chaînes, ou analyser des données reçues par réseau.

```
my_dict = {"id": 1, "product": "Widget"}
json_str = json.dumps(my_dict, indent=4)
print(json_str)
# Output:
# {
#   "id": 1,
#   "product": "Widget"
# }
```

```
parsed_data = json.loads(json_str)
print(parsed_data)
print(type(parsed_data)) #
```

- Le paramètre `indent=4` ajoute une indentation de 4 espaces à chaque niveau de structure JSON, rendant le fichier lisible et bien formaté.

# Méthode 1 : Exemple avec `open()` et `.close()`

Cette méthode offre un contrôle explicite, mais nécessite de fermer le fichier manuellement, ce qui peut entraîner des problèmes si la fermeture est oubliée ou en cas d'erreur d'exécution.

```
import os, json # Pour manipuler le JSON, il nous faut le module JSON

file_path = './file.json'
mon_dict = {'people': ['Albert', 'Martin', 'Louis'], 'mes Chiens': [1, 2, 4, 5]}

# Pour manipuler les JSON fichiers, il nous faut
# accéder aux deux méthodes ci-dessous
if os.path.exists(file_path):
    file = open(file_path, 'r')
    # Une fois le chargement du JSON,
    # on obtient ici une liste de dictionnaire
    # car le JSON contient plusieurs éléments dans un tableau

    # Pour charger un fichier dans un dictionnaire,
    # il nous faut la méthode .load()
    data = json.load(file)
    file.close()
    print(data)
else:
    file = open(file_path, 'w')

    # Pour sauvegarder un objet dans un JSON, il nous
    # faut la méthode .dump() (indent sert à avoir
    # une présentation plus esthétique)
    json.dump(mon_dict, file, indent=4)
    file.close()

# Pour obtenir la variable string qui va être la représentation textuelle d'un objet,
# on peut se servir de la méthode .dumps() (Avec indent=XXX pour
# l'esthétique ) qui va retourner un string
json_str = json.dumps(mon_dict, indent=4)
print(json_str)
print(type(json_str))

# Pour transformer une chaîne de caractère au format JSON en
# un dictionnaire, il existe la méthode .loads() qui
# va retourner un dictionnaire
data = json.loads(json_str)
print(data)
print(type(data))
print(data['people'])
```

## Risques et Bonnes Pratiques

L'utilisation de `open()` sans `with` nécessite de toujours appeler `.close()` explicitement. En cas d'erreur avant `.close()` est appelée, le fichier pourrait rester ouvert, consommant des ressources et potentiellement corrompant des données. C'est pourquoi la méthode `with open()` (comme vu dans l'exemple précédent) est fortement recommandée pour une gestion de fichiers plus robuste et sécurisée.

# Méthode 2 : Exemple avec le gestionnaire de contexte `with`

Cette méthode est la manière idiomatique et recommandée de manipuler les fichiers en Python. Elle utilise le gestionnaire de contexte `with` pour s'assurer que les fichiers sont correctement ouverts et fermés, même en cas d'erreurs, offrant ainsi un code plus propre et plus sûr.

```
import os, json

file_path = './file.json'
mon_dict = {'people': ['Albert', 'Martin', 'Louis'],
            'mes_chiens': [1, 2, 4, 5]}

# Vérification de l'existence du fichier et manipulation
if os.path.exists(file_path):
    # Le fichier existe : on le lit
    with open(file_path, 'r') as file:
        data = json.load(file)
        print(data)
        print(data['people']) # Accès aux données
else:
    # Le fichier n'existe pas : on le crée
    with open(file_path, 'w') as file:
        json.dump(mon_dict, file, indent=4)
        print("Fichier créé avec succès")

# Conversion dict → JSON string
json_str = json.dumps(mon_dict, indent=4)
print(json_str)
print(type(json_str))

# Conversion JSON string → dict
nouveau_dict = json.loads(json_str)
print(nouveau_dict)
print(type(nouveau_dict))
print(nouveau_dict['people'])
```

## □ Avantages de `with open()`

L'utilisation de `with open()` garantit la **fermeture automatique** du fichier dès que le bloc de code est exécuté, même si une erreur survient. Cela **prévient les fuites de ressources** et rend le code **plus robuste et plus facile à lire**. C'est la méthode la plus sûre et recommandée pour la gestion des fichiers en Python.

# Les 4 Méthodes de Gestion des Fichiers JSON

Le module `os` fournit des fonctions pour interagir avec le système d'exploitation de manière traditionnelle et procédurale, tandis que le module `pathlib` introduit une approche plus moderne et orientée objet pour manipuler les chemins de fichiers, rendant le code souvent plus lisible et moins sujet aux erreurs.

Ces deux modules peuvent être combinés avec les méthodes de lecture/écriture de fichiers, soit via `open()/close()` pour un contrôle manuel, soit avec le gestionnaire de contexte `with` pour une gestion automatique et sécurisée des ressources.

Module	Méthode d'ouverture	Exemple de syntaxe
<code>os</code>	<code>open()/close()</code>	<pre>import os, json file_path = './data.json' file = open(file_path, 'r') data = json.load(file) file.close()</pre>
<code>os</code>	<code>with open()</code>	<pre>import os, json file_path = './data.json' with open(file_path, 'r') as file:     data = json.load(file)</pre>
<code>pathlib</code>	<code>Path().open()/close()</code>	<pre>from pathlib import Path import json file_path = Path('./data.json') file = file_path.open('r') data = json.load(file) file.close()</pre>
<code>pathlib</code>	<code>with Path.open()</code>	<pre>from pathlib import Path import json file_path = Path('./data.json') with file_path.open('r') as file:     data = json.load(file)</pre>



# Chemins Relatifs vs Absolus

## Chemins relatifs

Les chemins relatifs décrivent l'emplacement d'un fichier ou d'un répertoire par rapport à la position actuelle (le répertoire de travail). Ils sont flexibles mais peuvent devenir ambigus si le point de référence change.

- `.` → dossier courant
- `..` → dossier parent
- `./dossier` → dans le dossier courant
- `../dossier` → dans le dossier au-dessus

## Chemins absolus

Les chemins absolus spécifient l'emplacement exact d'un fichier ou d'un répertoire à partir de la racine du système de fichiers. Ils sont infaillibles, quelle que soit la position actuelle du programme.

Obtention avec :

- `os.path.abspath("./data_os")`
- `pathlib_path.resolve()`

Les chemins absolu sont particulièrement utiles pour éviter les erreurs de localisation, car ils garantissent que votre programme trouve toujours les fichiers et répertoires corrects, quel que soit le dossier à partir duquel le script est exécuté. Cela apporte une robustesse essentielle aux applications Python.



# Fonctions Essentielles du Module os

## `os.path.join(a, b)`

- Assemble proprement un chemin selon l'OS
- Évite les erreurs de / et \ selon l'OS
- Exemple : `os.path.join("dossier", "fichier.json")`

## `os.makedirs(path, exist_ok=True)`

- Crée un dossier, même s'il existe déjà
- `exist_ok=True` évite les erreurs si le dossier existe
- Exemple pratique avec création de structure de dossiers

## `os.path.abspath(chemin)`

- Convertit un chemin relatif en chemin absolu
- Utile pour déboguer et localiser précisément les fichiers

# Fonctions Essentielles du Module `pathlib`



`Path("nom_dossier")`

- Crée un objet chemin moderne et orienté objet
- Plus lisible et moins sujet aux erreurs que `os.path`



`path.resolve()`

- Renvoie le chemin ABSOLU complet
- Équivalent moderne de `os.path.abspath()`



`path.mkdir(parents=True, exist_ok=True)`

- `parents=True` → crée les dossiers parents si besoin
- `exist_ok=True` → ne génère pas d'erreur si le dossier existe déjà
- Équivalent moderne de `os.makedirs()`

Le module `pathlib` offre une approche moderne et orientée objet pour manipuler les chemins de fichiers, rendant le code souvent plus lisible et moins sujet aux erreurs. C'est l'approche recommandée pour les nouveaux projets Python.

# JSON : Paramètres d'Encodage Importants

## `encoding="utf-8"`

- S'applique AU FICHIER
- Permet d'écrire/lire correctement les accents
- Standard officiel JSON (RFC 8259)
- Compatible Windows/Linux/Mac

## `ensure_ascii=False`

- S'applique AU CONTENU JSON
- Permet de laisser les accents visibles au lieu de les convertir
- Exemple : "é" reste "é" au lieu de "\u00e9"

## Tableau Comparatif des Paramètres d'Encodage

Paramètre	Application	Effet Principal	Comportement des Accents
<code>encoding="utf-8"</code>	Au fichier lors de l'ouverture ( <code>open()</code> )	Définit comment les caractères sont stockés/lus physiquement dans le fichier.	Assure que les accents sont correctement interprétés et écrits comme des caractères UTF-8.
<code>ensure_ascii=False</code>	Au contenu JSON lors de l'encodage ( <code>json.dump()</code> ou <code>json.dumps()</code> )	Contrôle la représentation des caractères non-ASCII dans la chaîne JSON.	Laisse les accents et autres caractères non-ASCII visibles et non échappés (par ex. "é" au lieu de "\u00e9").

Il est important de comprendre que ces deux paramètres sont complémentaires et jouent des rôles distincts. Le paramètre `encoding="utf-8"` garantit que le fichier lui-même est traité correctement en UTF-8, tandis que `ensure_ascii=False`, utilisé avec `json.dump()` ou `json.dumps()`, optimise la lisibilité du contenu JSON en conservant les caractères non-ASCII tels quels, sans les convertir en séquences d'échappement.

# Exemple Complet : Les 4 Méthodes en Action

Ce script illustre les différentes manières de gérer des fichiers JSON en Python, en utilisant les modules `os` et `pathlib`, combinés avec les gestionnaires de fichiers `open()/close()` et `with open()`, ainsi que les paramètres d'encodage importants.

## 1. Préparation des données

```
import json
import os
from pathlib import Path

# 1. Données exemple
data_to_save = {
    "nom": "Jean-Luc Picard",
    "grade": "Capitaine",
    "vaisseau": "USS Enterprise-D",
    "capacites": ["diplomatie", "commandement", "archéologie"],
    "coordonnees": {
        "latitude": 40.7128,
        "longitude": -74.0060
    },
    "actif": True,
    "notes_accents": "Ceci est un texte avec des caractères accentués : éàçù."
}

# Définir le nom du fichier JSON
file_name = "data_exemple.json"

# Chemin du dossier de sortie (créé avec os.makedirs pour la démonstration)
output_dir_os = "./output_os"
output_dir_pathlib = Path("./output_pathlib")

# Créer les dossiers de sortie pour les exemples
os.makedirs(output_dir_os, exist_ok=True)
output_dir_pathlib.mkdir(parents=True, exist_ok=True)

# Conversion des données en string JSON pour dumps()/loads()
json_string = json.dumps(data_to_save, ensure_ascii=False, indent=4)
print(f"Chaîne JSON préparée :\n{json_string}\n")
```

## 2. Méthode 1 : os + open()/close()

```
# Chemin complet du fichier
file_path_os_manual = os.path.join(output_dir_os, "methode1_os_manual.json")

# Écriture du fichier
print(f"Écriture avec os + open()/close() dans {file_path_os_manual}")
file_os_manual_write = open(file_path_os_manual, 'w', encoding="utf-8")
json.dump(data_to_save, file_os_manual_write, ensure_ascii=False, indent=4)
file_os_manual_write.close()

# Lecture du fichier
file_os_manual_read = open(file_path_os_manual, 'r', encoding="utf-8")
data_read_os_manual = json.load(file_os_manual_read)
file_os_manual_read.close()
print(f"Données lues (Méthode 1) : {data_read_os_manual['nom']} - {data_read_os_manual['notes_accents']}\n")
```

## 3. Méthode 2 : os + with open()

```
# Chemin complet du fichier
file_path_os_with = os.path.join(output_dir_os, "methode2_os_with.json")

# Écriture du fichier (utilisation de 'with' pour une gestion automatique de la fermeture)
print(f"Écriture avec os + with open() dans {file_path_os_with}")
with open(file_path_os_with, 'w', encoding="utf-8") as file_os_with_write:
    json.dump(data_to_save, file_os_with_write, ensure_ascii=False, indent=4)

# Lecture du fichier
with open(file_path_os_with, 'r', encoding="utf-8") as file_os_with_read:
    data_read_os_with = json.load(file_os_with_read)
print(f"Données lues (Méthode 2) : {data_read_os_with['nom']} - {data_read_os_with['notes_accents']}\n")
```

## 4. Méthode 3 : pathlib + open()/close()

```
# Chemin complet du fichier (objet Path)
file_path_pathlib_manual = output_dir_pathlib / "methode3_pathlib_manual.json"

# Écriture du fichier
print(f"Écriture avec pathlib + Path().open()/close() dans {file_path_pathlib_manual}")
file_pathlib_manual_write = file_path_pathlib_manual.open('w', encoding="utf-8")
json.dump(data_to_save, file_pathlib_manual_write, ensure_ascii=False, indent=4)
file_pathlib_manual_write.close()

# Lecture du fichier
file_pathlib_manual_read = file_path_pathlib_manual.open('r', encoding="utf-8")
data_read_pathlib_manual = json.load(file_pathlib_manual_read)
file_pathlib_manual_read.close()
print(f"Données lues (Méthode 3) : {data_read_pathlib_manual['nom']} - {data_read_pathlib_manual['notes_accents']}\n")
```

## 5. Méthode 4 : pathlib + with .open()

```
# Chemin complet du fichier (objet Path)
file_path_pathlib_with = output_dir_pathlib / "methode4_pathlib_with.json"

# Écriture du fichier
print(f"Écriture avec pathlib + with Path().open() dans {file_path_pathlib_with}")
with file_path_pathlib_with.open('w', encoding="utf-8") as file_pathlib_with_write:
    json.dump(data_to_save, file_pathlib_with_write, ensure_ascii=False, indent=4)

# Lecture du fichier
with file_path_pathlib_with.open('r', encoding="utf-8") as file_pathlib_with_read:
    data_read_pathlib_with = json.load(file_pathlib_with_read)
print(f"Données lues (Méthode 4) : {data_read_pathlib_with['nom']} - {data_read_pathlib_with['notes_accents']}\n")
```

## 6. Bonus : dumps()/loads()

```
# Utilisation de json.dumps pour convertir un dictionnaire Python en chaîne JSON
# ensure_ascii=False pour préserver les caractères accentués
json_output_string = json.dumps(data_to_save, ensure_ascii=False, indent=4)
print(f"Sortie de json.dumps() avec ensure_ascii=False :\n{json_output_string}\n")

# Utilisation de json.loads pour convertir une chaîne JSON en dictionnaire Python
data_from_string = json.loads(json_output_string)
print(f"Données lues via json.loads() : {data_from_string['nom']} - {data_from_string['notes_accents']}\n")
```



# Points Clés à Retenir

## 🔑 Dictionnaires

- Structure clé-valeur native Python
- Méthodes essentielles : `.keys()`, `.values()`, `.items()`
- Suppression : `del` ou `.pop()`
- Fusion : `.update()`

## 📄 Format JSON

- Standard universel d'échange de données
- Correspondance naturelle avec les dictionnaires Python
- Léger, lisible et performant
- Module `json` intégré à Python

## 🔧 Manipulation JSON

- `load()` / `dump()` : fichiers
- `loads()` / `dumps()` : chaînes
- Paramètre `indent` pour la lisibilité
- Gestion des erreurs avec `os.path.exists()`

Vous maîtrisez maintenant les fondamentaux des dictionnaires et de JSON en Python ! Ces compétences sont essentielles pour vos projets SLAM : API REST, configuration d'applications, manipulation de données structurées. Pratiquez régulièrement avec des exercices concrets pour consolider ces acquis.