Les dictionnaires, type `dict`

Introduction

"Point de cours 1"

Un **dictionnaire** en Python est une structure de données qui permet d'associer des **clés** à des **valeurs**. Contrairement aux listes, les éléments d'un dictionnaire ne sont pas ordonnés et sont accessibles par leur clé et non via leur position dans le dictionnaire.

En Python, un dictionnaire est de type `dict`, il est délimité par des accolades et les associations entre `clef` et `valeur` sont notées `clef: valeur` et les différentes associations sont séparées par une virgule.

```
eleve = {'nom': 'Dupont', 'prénom': 'Jean-Louis', 'Âge': 50}
# accès au prénom
print(eleve['prénom'])
```

"Pourquoi utiliser les dictionnaires ?"

Les dictionnaires sont des **p-uplets nommés**, permettant d'accéder aux éléments via une clé plutôt que par un indice, ce qui rend l'accès aux données plus **expressif** et **compréhensible**.

Comparons un tuple et un dictionnaire contenant les mêmes informations :

"Utilisation d'un tuple"

```
personne_tuple = ("Dupont", "Jean", "06.12.34.56.78")
print(personne_tuple[2])  # Pas très explicite
```

"Utilisation d'un dictionnaire"

```
personne_dico = {"nom": "Dupont", "prénom": "Jean",
                 "téléphone": "06.12.34.56.78"}
print(personne_dico["téléphone"])  # Plus clair et lisible
```

"Attention"

Un dictionnaire est l'analogue d'une fonction en mathématiques.

En mathématiques, une fonction f n'associe à un x fixé qu'une seule image possible $f(x)$. En revanche, un même y peut être l'image $y = f(x_i)$ de plusieurs x_i distincts appelés antécédents.

De même, une clef ne peut apparaître qu'une seule fois dans un dictionnaire mais une même valeur peut être associée à plusieurs clefs distinctes.

Mathématiques	Programmation
Fonction	Dictionnaire
Antécédent	Clef
Image	Valeur

"Cas d'utilisation des dictionnaires"

Les dictionnaires sont particulièrement utiles dans plusieurs cas :

- **Métadonnées EXIF** : Stocker des informations sur les images (taille, date, appareil photo utilisé).
- **Extraction de fichiers CSV** : Associer des en-têtes de colonnes à leurs valeurs pour un accès rapide.
- **Indexation de données** : Associer un identifiant unique à une information (ex : numéros d'étudiants et notes).
- **Stockage de configurations** : Enregistrer des paramètres d'une application.

Dictionnaires en Python

"Création d'un dictionnaire"

Plusieurs méthodes sont possibles pour créer un dictionnaire de type `dict` :

```
# Création par extension
dico = {"nom": "Alice", "âge": 25, "ville": "Paris"}

# Création avec le constructeur dict
dico2 = dict(nom="Bob", âge=30, ville="Lyon")

# Création par compréhension
carres = {x: x**2 for x in range(1, 6)}

# création d'un dictionnaire vide
dico_vide1 = {}  # avec délimiteurs
dico_vide2 = dict()  # avec constructeur
```

"Accès, modification et suppression"

Les dictionnaires sont des objets mutables en Python, ils sont accessibles en lecture ou écriture comme les objets de type `list` . Pour rappel les objets de type `tuple` sont juste accessibles en lecture.

```
# Accéder à une valeur
print(dico["nom"])  # Affiche "Alice"

# Modifier ou ajouter une valeur
dico["âge"] = 26  # Modification
dico["pays"] = "France"  # Ajout

# Supprimer une entrée
del dico["ville"]
```

"Différences entre liste et dictionnaire"

Les méthodes `append` et `insert` n'existent pas pour un dictionnaire car il n'y a pas de notion d'ordre d'insertion. En revanche la méthode `pop` existe.

La fonction `sorted` ne trie que les clefs et la méthode `sort` n'est pas définie. Enfin, les fonctions `min` et `max` n'opèrent de même que sur les clefs.

Type	Opération	Syntaxe
`list`	ajout d'un élément à la fin	`lis.append(element)`
`dict`	ajout d'une association `clef: valeur`	`dico[clef] = valeur`
`list`	extraire l'élément d'index `k`	`lis.pop(k)`
`dict`	extrait la valeur associée à la `clef`	`dico.pop(clef)`

Un petit exemple avec le nombre de titres de champions de France de Football en 2025 :

```
>>> titres = {'PSG': 12, 'ASSE': 10,
              'OM': 10,  'ASM': 8, 'OL': 7}
>>> titres['OM'] = titres['OM'] - 1
# Affaire OM/VA le titre de 1993 est retiré pour tricherie
>>> titres['FCNA'] = 8 # on avait oublié le FC Nantes !
>>> titres.pop('PSG') # exclu pour concurrence déloyale
12
>>> titres
{'ASSE': 10, 'OM': 9, 'FCNA': 8, 'ASM': 8, 'OL': 7}
>>> sorted(titres) # on ne trie que les clefs
['ASM', 'ASSE', 'FCNA', 'OL', 'OM']
```

"Objets mutables ou immuables"

Type	Propriété	Sens
`list` , `dict`	mutable	acessible en lecture et écriture
`tuple` , `int` , `bool` , `str`	immuable	accessible en lecture seule / non modifiable

Attention, dans les associations `clef: valeur` d'un dictionnaire, la `clef` ne peut être que d'un type **immuable** c'est-à-dire non modifiable en écriture : `tuple` , `int` , `bool` ,

`str` . Il n'y a pas de restriction sur le type de la `valeur` .

```
# un dictionnaire dont les valeurs sont des listes
carnet_notes = {'Alex' :[14, 11, 13], 'Sabri': [18, 15, 14]}
# valeur d'une position (ligne, colonne) dans un jeu de plateau
# on peut utiliser des tuples comme clefs car ils sont immuables
valeur = {(1, 2): 10, (4, 5): -1}
# on ne pourrait pas utiliser [1, 2] et [4, 5] comme clefs
```

"Parcours d'un dictionnaire"

Trois méthodes de parcours d'une dictionnaire sont possibles. Le parcours par index n'est pas possible car il n'y a pas d'index dans un dictionnaire mais des clefs.

```
# Parcours par les clés
for cle in dico.keys():
    print(cle)

# Syntaxe simplifiée pour le parcours par les clés
for cle in dico:
    print(cle)

# Parcours par les valeurs
for valeur in dico.values():
    print(valeur)

# Parcours par paires clé-valeur
for cle, valeur in dico.items():
    print(f"{cle}: {valeur}")
```

Complexité des opérations sur un dictionnaire

"Test d'appartenance"

- Une clef peut être présente au plus une fois dans un dictionnaire. Pour tester *si une clef est présente* dans un dictionnaire on utilisera l'opérateur `in` qui est très efficace (voir ci-

dessous) :

Test	Syntaxe
`clef` présente dans `dico`	`clef in dico`
`clef` pas présente dans `dico`	`clef not in dico`

- Une valeur peut être présente plusieurs fois dans un dictionnaire. Compter le nombre d'occurrences d'une valeur `cible` nécessite une boucle :

```
def nombre_occurrence(cible, dico):
    n = 0
    for clef, valeur in dico.items():
        if valeur == cible:
            n = n + 1
    return n
```

"Point de cours 2"

En Python un **dictionnaire** de type `dict` est implémenté par une table de hachage qui permet des opérations très performantes en **temps constant**. Pour rechercher si une clef est présente dans un dictionnaire, inutile de parcourir toutes les clefs comme on le ferait pour une recherche séquentielle dans une liste, la fonction de hachage permet de déterminer si la clef est présente grâce à un calcul reposant sur l'arithmétique modulaire (mathématiques expertes au lycée).

En revanche, ce qu'on gagne en temps, on le perd en mémoire car la structure de données d'une table de hachage est complexe : l'empreinte mémoire de l'implémentation d'un objet de type `dict` est beaucoup plus importante que pour un objet de type `list` .

Structure de données	Taille de l'entrée	Coût de la recherche d'une clef dans le pire cas	Notation mathématique
Liste/tableau	n	linéaire, proportionnel à n	$O(n)$
Dictionnaire	n	constant, ne dépend pas de n	$O(1)$

```python
# Recherche dans une liste (O(n))
liste_noms = ["Nom1", "Nom2", ..., "Nom9999"]
nom_recherche = "Nom10000"
print(nom_recherche in liste_noms)  # Parcours potentiel de tous les éléments

# Recherche dans un dictionnaire (O(1))
dico_noms = {"Nom1": 1, "Nom2": 2, ..., "Nom9999": 9999}
print("Nom10000" in dico_noms)
# Recherche immédiate grâce au hachage
```

Structures de données imbriquées

"Dictionnaires imbriqués"

- On utilise souvent des listes de dictionnaires pour représenter les données d'un fichier CSV sous forme de table comme dans un tableur.

"Exemple avec un fichier CSV"

Fichier `etudiants.csv` :

```
nom,âge,note
Alice,20,15
Bob,22,12
Charlie,19,17
```

Représentation en liste de dictionnaires en Python :

```python
etudiants = [
    {"nom": "Alice", "âge": 20, "note": 15},
    {"nom": "Bob", "âge": 22, "note": 12},
    {"nom": "Charlie", "âge": 19, "note": 17}
]
print(etudiants[1]["nom"])  # Affiche "Bob"
```

- Pour les données hiérarchisées, le format de fichier JSON est très courant sur les plateformes d'Open Data. On peut naturellement représenter des données au format JSON en Python sous la forme d'un dictionnaire de dictionnaires

"Exemple avec un fichier JSON"

Fichier `villes.json` :

```
{
    "Paris": {"latitude": 48.8566, "longitude": 2.3522},
    "Londres": {"latitude": 51.5074, "longitude": -0.1278},
    "New York": {"latitude": 40.7128, "longitude": -74.0060}
}
```

Représentation en dictionnaire de dictionnaires Python :

```
villes = {
    "Paris": {"latitude": 48.8566, "longitude": 2.3522},
    "Londres": {"latitude": 51.5074, "longitude": -0.1278},
    "New York": {"latitude": 40.7128, "longitude": -74.0060}
}
print(villes["Paris"]["latitude"])  # Affiche 48.8566
```

Mutabilité et partage de référence

"Pièges de la mutabilité"

Comme les listes, les dictionnaires en Python sont des objets **mutables**, ce qui signifie qu'ils peuvent être modifiés après leur création. Il faut être vigilant lorsqu'on réalise des copies de dictionnaires, car une simple affectation partage la même référence en mémoire.

"Partage de référence"

L'affectation d'un dictionnaire à un nouveau nom de variable ne fait pas une copie mais crée un *alias*, un nouveau nom, qui partage la référence vers le même dictionnaire.

Exemple sur Pythontutor.

```
dico1 = {"nom": "Alice", "âge": 25}
dico2 = dico1  # dico2 référence le même objet que dico1
dico2["âge"] = 30
print(dico1["âge"])
# Affiche 30, car dico1 et dico2 pointent vers le même objet
```

"Copie indépendante"

La fonction `deepcopy` du module `copy` permet de faire une *copie en profondeur* d'un dictionnaire en déréférençant tous les objets mutables imbriqués dans la structure.

Exemple sur Pythontutor

```
import copy
dico1 = {"nom": "Alice", "âge": 25}
dico2 = copy.deepcopy(dico1)  # Création d'une copie indépendante
dico2["âge"] = 30
print(dico1["âge"])
# Affiche 25, car dico1 et dico2 sont maintenant distincts
```

QCM de synthèse

Chaque question comporte une seule bonne réponse parmi les quatre proposées.

Question 1

On considère une variable telle que :

```
mots = ["chat", "chien", "éléphant"]
```

qui contient une liste de mots de type `str`. Comment générer un dictionnaire `longueurs` qui associe à chaque mot de la liste sa longueur ?

On devrait avoir :

```
longueurs = {'chat': 4, 'chien': 5, 'éléphant': 8}
```

Le code doit fonctionner quelle que soit la valeur de la variable `mots`.

- [ ] **A)** `longueurs = {m for m in mots: len(m)}`
- [ ] **B)** `longueurs = {m: len(m) for m in range(len(mots))}`
- [ ] **C)** `longueurs = {m: len(m) for m in mots}`
- [ ] **D)** `longueurs = {m: len(m) for m, len(m) in mots.items()}`

Question 2

Soit le dictionnaire :

```
legumes = {'carotte': 3, 'tomate': 5, 'courgette': 2}
```

Comment accéder à la quantité de tomates ?

- [ ] **A)** `legumes['tomate']`
- [ ] **B)** `legumes[1]`
- [ ] **C)** `legumes('tomate')`
- [ ] **D)** `legumes(1)`

Question 3

On définit une variable `repertoire` ainsi :

```
repertoire = [{'nom': 'Francette', 'poste': 412},
              {'nom': 'Jeanne', 'poste': 222},
              {'nom': 'Éric', 'poste': 231}]
```

Quelle expression permet d'accéder au poste d'Éric ?

- [ ] **A)** `repertoire[2]['poste']`
- [ ] **B)** `repertoire['poste'][2]`
- [ ] **C)** `repertoire['Éric']['poste']`
- [ ] **D)** `repertoire['Éric']`

Question 4

Étant donné le dictionnaire :

```
res = {"Donald": 10, "Edger": 7, "Grace": 9, "Ada": 6}
```

Comment sélectionner les élèves ayant un score inférieur à 8 ?

- [ ] **A)**

```
sb = {nom: score if score < 8 for nom, score in res.items()}
```

- [ ] **B)**

```
sb = {nom: score for nom, score in res if score < 8}
```

- [ ] **C)**

```
sb = {nom: score for nom, score in res.items() if score < 8}
```

- [ ] **D)**

```
sb = {nom: score for nom, score in res.values() if score < 8}
```

Question 5

On considère le dictionnaire :

```
apprenants = {'Maxime': 15, 'Julie': 15, 'Alexandre': 17, 'Sophie': 16}
```

Comment ajouter `Thomas` , 16 ans ?

- [ ] **A)** `apprenants['Thomas'] = 16`
- [ ] **B)** `apprenants.append('Thomas', 16)`
- [ ] **C)** `apprenants.add('Thomas', 16)`
- [ ] **D)** `apprenants.insert('Thomas', 16)`

Question 6

Soit le dictionnaire :

```
monuments = {"France": "Notre-Dame", "Italie": "Colisée",
             "Corée du Nord": "Monument Mansudae", "Angleterre": "Big Ben"}
```

Vous souhaitez afficher chaque paire `pays-monument` sous la forme :

```
Le monument emblématique de [pays] est [monument].
```

en itérant sur le dictionnaire. Quelle est la syntaxe correcte pour réaliser cette opération ?

- [ ] **A)**

```
for pays in monuments.keys():
    print("Le monument emblématique de " + pays + " est " + monuments[pays])
```

- [ ] **B)**

```
for monument, pays in monuments.items():
    print("Le monument emblématique de " + pays + " est " + monument)
```

- [ ] **C)**

```
for pays in monuments.values():
    print("Le monument emblématique de " + pays + " est " + monuments[pays])
```

- [ ] **D)**

```
for pays, monument in monuments:
    print("Le monument emblématique de " + pays + " est " + monument)
```