



Python - Dictionnaires

[Mise à jour le : 30/6/2021]

- **Sources**

- **Documentation** sur Python.org : [référence du langage](#), [dictionnaire](#), [fonctions natives](#) (built-in)

- **Lectures connexes**

- **Real Python**
 - [OrderedDict vs dict in Python: The Right Tool for the Job](#)
 - [Python's collections: A Buffet of Specialized Data Types](#)

- **Mots-clés** : dictionnaire, parcours de dictionnaires.



Les mots ci-dessous sont dits “réservés”. Ils ne peuvent pas être utilisés comme nom de variable. Les mots soulignés sont une nouveauté de Python 3. Les mots en **gras** sont utilisés dans cette page.

and	continue	finally	is	raise
as	def	for	lambda	return
assert	del	from	<u>None</u>	<u>True</u>
<u>async</u>	elif	global	<u>nonlocal</u>	try
<u>await</u>	else	if	not	while
break	except	import	or	with
class	<u>False</u>	in	pass	yield

- **Fonctions natives (built-in)**¹⁾ utilisées dans les exemples : **dict()**²⁾, **del()**, **print()**, **range()**, **zip()**.

1. Introduction

Le dictionnaire est une implémentation de table de hash. Il permet l'accès, l'insertion et le test d'appartenance indépendamment du nombre d'éléments. Le dictionnaire est un objet conteneur. À la différence des séquences, qui sont indexées par des nombres, les dictionnaires sont indexés par des **clés**, qui peuvent être de n'importe quel type immuable ; les chaînes de caractères et les nombres peuvent toujours être des clés.



Les **dictionnaires** sont des objets **mutables**. Leur structure n'est pas ordonnée (ceci est dû à l'action de la fonction de hachage). Les **clés** doivent avoir un **type immuable**.

2. Création



On utilise l'expression suivante : `nom_dictionnaire = dict()` ou `nom_dictionnaire = {}` pour créer des dictionnaires vides.

Exemples

*.py

```
# Première méthode
dico = {} # dictionnaire vide
dico = {'nom': 'Martin', 'prenom': 'Pierre-Emile'} # création en extension
print(dico) # Résultat : {'nom': 'Martin', 'prenom': 'Pierre-Emile'}

# Deuxième méthode
dico = dict() # dictionnaire vide
# La fonction native dict() construit un dictionnaire directement à
# partir d'une liste
# de paires clé-valeur stockées sous la forme de tuples.
dico = dict([('nom', 'Martin'), ('prenom', 'Pierre-Emile')])
print(dico) # Résultat : {'nom': 'Martin', 'prenom': 'Pierre-Emile'}
# ou
# Construction à partir d'une liste de tuples et de la fonction native
dict()
l = [('nom', 'Martin'), ('prenom', 'Pierre-Emile')]
dico = dict(l)
print(dico) # Résultat : {'nom': 'Martin', 'prenom': 'Pierre-Emile'}

# Troisième méthode
dico = dict(nom='Martin', prenom='Pierre-Emile')
print(dico) # Résultat : {'nom': 'Martin', 'prenom': 'Pierre-Emile'}
```



Les **accolades** délimitent les **dictionnaires**.

3. Lecture de la valeur d'un élément

On accède à une valeur à partir de sa **clé** par `nom_dictionnaire[clé]`. Si la clé n'existe pas, une exception de type **KeyError** sera levée.

Exemple

*.py

```
# En faisant un test d'appartenance on évite la génération d'une erreur
```

```
KeyError si la clé est absente
if 'nom' in dico:
    print(dico["nom"]) # résultat : Martin
```

4. Ajout ou modification d'un élément



Le dictionnaire est un type **mutable**, et donc on peut **modifier la valeur** associée à une clé. On utilise l'expression suivante : `nom_dictionnaire[clé] = valeur`. Et de la même façon, **ajouter une entrée**.

Exemples

*.py

```
dico = {}
# La clé peut être une chaîne de caractères
dico['nom'] = 'Martin'
dico['prenom'] = 'Pierre-Emile'
print(dico) # résultat : {'nom': 'Martin', 'prenom': 'Pierre-Emile'}

# La clé peut être un tuple
echiquier = {}
echiquier[('a', 1)] = 'tour blanche'
print(echiquier) # résultat : {('a', 1): 'tour blanche'}
```

5. Suppression d'éléments



On utilise `del(nom_dictionnaire[clé])` ou `nom_dictionnaire.pop("clé")`. `pop` renvoie la valeur.

*.py

```
placard = {"chemise": 6, "pantalon": 4}
del(placard["chemise"])
print(placard) # Résultat : {'pantalon': 4}
n = placard.pop("pantalon")
print(placard) # Résultat : {}
print(n, "pantalon(s) donné(s)") # Résultat : 4 pantalon(s) donné(s)
```

6. Le parcours de dictionnaires

La méthode la plus fréquente pour parcourir tout un dictionnaire est à base de la méthode `items`.

6.1 Parcours des clés et valeurs simultanément



On utilise la méthode **items** de la classe **dict**. Elle renvoie une **liste**, contenant les couples **clé : valeur**, sous la forme d'un tuple.

*.py

```
placard = {"chemise": 6, "pantalon": 4, "chaussette": 10, "pull": 4}
# A chaque tour de boucle, items renvoie un tuple constitué de la clé
# et de la valeur
for cle, valeur in placard.items(): # notation de tuple unpacking
    print(cle, valeur)

# Résultat
chemise 6
pantalon 4
chaussette 10
pull 4
```

On peut obtenir séparément la liste des clés et des valeurs.

6.2 Parcours des clés



On utilise la méthode **keys()** de la classe **dict**.

*.py

```
placard = {"chemise": 6, "pantalon": 4, "chaussette": 10, "pull": 4}
for cle in placard.keys():
    print(cle)

# Résultat
chemise
pantalon
chaussette
pull
```



L'itérateur sur les dictionnaires itère directement sur les clés.

Exemple

*.py

```
# Dans l'exemple précédent, on obtient le même résultat sans préciser la
méthode keys()
placard = {"chemise": 6, "pantalon": 4, "chaussette": 10, "pull": 4}
for cle in placard:
    print(cle)

# Résultat
chemise
pantalon
chaussette
pull
```

6.3 Parcours des valeurs



On utilise la méthode **values()** de la classe **dict**.

*.py

```
placard = {"chemise": 6, "pantalon": 4, "chaussette": 10, "pull": 4}
for valeur in placard.values():
    print(valeur)

# Résultat
6
4
10
4
```



Les méthodes **keys()**, **values()** et **items()** retournent un objet particulier appelé : **une vue** (itérable et possédant le test d'appartenance). La caractéristique principale d'une vue est qu'elle est **mise à jour** en même temps que le dictionnaire.

Exemple

*.py

```
# Création du dictionnaire en extension
```

```
placard = {"chemise": 6, "pantalon": 4, "chaussette": 10, "pull": 4}
k = placard.keys() # Création d'une "vue" sur le dictionnaire placard
print(k) # Résultat : dict_keys(['chemise', 'pantalon', 'chaussette', 'pull'])
placard['short'] = 3 # Ajout d'un couple dans le dictionnaire placard
print(placard) # Résultat : {'chemise': 6, 'pantalon': 4, 'chaussette': 10, 'pull': 4, 'short': 3}
# La vue a été modifiée sans réaffectation
print(k) # Résultat : dict_keys(['chemise', 'pantalon', 'chaussette', 'pull', 'short'])

# Test d'appartenance sur une vue
'chemise' in k # Résultat : True
'ceinture' in k # Résultat : False
```

7. Formation d'un dictionnaire à partir de deux listes



On utilise l'expression suivante : `nom_dictionnaire = dict(zip(listeClés,listeVal))`

Exemples

*.py

```
# Formation du dictionnaire dico à partir de deux listes
cles=['a','b','c']
valeurs=[1,2,3]
dico=dict(zip(cles,valeurs))

# Affichage
print(dico) # résultat : {'a': 1, 'b': 2, 'c': 3}
```

8. Transformation d'un dictionnaire en paramètres nommés d'une fonction

Exemple

*.py

```
parametres = {"sep" : " >> ", "end" : " -\n"}
print("Voici", "un", "exemple", "d'appel", **parametres)
# Résultat : Voici >> un >> exemple >> d'appel -
```

9. Gérer des enregistrements

Un enregistrement est une donnée composite qui contient plusieurs champs (struct ou un record dans d'autres langages).

9.1 Implémenter un enregistrement comme un dictionnaire

Exemple

*.py

```
# Enregistrement
personnes = [
    {'nom': 'Pierre', 'age': 25, 'email': 'pierre@example.com'},
    {'nom': 'Paul', 'age': 18, 'email': 'paul@example.com'},
    {'nom': 'Jacques', 'age': 52, 'email': 'jacques@example.com'},
]
# Pour l'anniversaire de Pierre on fera :
personnes[0]['age'] += 1
# Affichage
for personne in personnes:
    print(10*"=")
    for info, valeur in personne.items():
        print(f"{info} -> {valeur}")
# Résultat
# =====
# nom -> Pierre
# age -> 26
# email -> pierre@example.com
# =====
# nom -> Paul
# age -> 18
# email -> paul@example.com
# =====
# nom -> Jacques
# age -> 52
# email -> jacques@example.com
```



Problème : l'accès à un enregistrement suppose ici que l'on connaisse sa position.

9.2 Un dictionnaire pour indexer les enregistrements



Pour modéliser ces informations, il est plus adapté d'utiliser, non pas une liste, mais



un dictionnaire de dictionnaires.

Exemple

*.py

```
personnes = [
    {'nom': 'Pierre', 'age': 25, 'email': 'pierre@example.com'},
    {'nom': 'Paul', 'age': 18, 'email': 'paul@example.com'},
    {'nom': 'Jacques', 'age': 52, 'email': 'jacques@example.com'},
]
# on crée un index permettant de retrouver rapidement une personne dans
# la liste
index_par_nom = {personne['nom']: personne for personne in personnes}
index_par_nom # Résultat : {'Pierre': {'nom': 'Pierre', 'age': 26,
'email': 'pierre@example.com'},
#                               'Paul': {'nom': 'Paul', 'age': 18, 'email':
'paul@example.com'},
#                               'Jacques': {'nom': 'Jacques', 'age': 52,
'email': 'jacques@example.com'}}
# On accède à l'age de Pierre par
index_par_nom['Pierre']['age'] # Résultat : 26
# au lieu de personnes[0]['age'], ce qui est plus pertinent car un
# dictionnaire n'est pas ordonné
```

9. Les méthodes de la classe dict

- Source [w3schools.com](https://www.w3schools.com/python/python_dictionaries_methods.asp)

Fonction	Paramètres	Effet	Structure
clear()		Supprime tous les éléments du dictionnaire.	d.clear()
copy()		Renvoie une copie du dictionnaire.	d.copy()
fromkeys()	clés,valeur	Crée un dictionnaire à partir d'une liste de clés prenant la même valeur ou None .	d.fromkeys(keys,value)
get()	clé,default	Renvoie la valeur de l'élément avec la clé spécifiée ou la valeur par défaut si celle-ci est absente.	d.get(clé, default)
items()		Renvoie les paires clé-valeur du dictionnaire.	d.items()
keys()		Renvoie la liste des clés du dictionnaire.	d.keys()
popitem()		Renvoie et supprime le dernier élément du dictionnaire.	d.popitem()
pop()	clé	Renvoie et supprime l'élément correspondant à la clé.	d.pop(clé)
setdefault()	clé,valeur	Renvoie l'élément correspondant à la clé. S'il n'est pas présent, insère la clé avec la valeur dans le dictionnaire.	d.setdefault(clé,valeur)
update()	iterable	Met à jour le dictionnaire.	d.update(iterable)

Fonction	Paramètres	Effet	Structure
values()		Renvoie les valeurs du dictionnaire .	d.values()

Résumé

- Un dictionnaire est un objet conteneur associant des clés à des valeurs.
- Pour créer un dictionnaire en extension, on utilise la syntaxe **dictionnaire = {clé1:valeur1,clé2:valeur2,clén:valeurn}**.
- On ajoute ou on remplace un élément dans un dictionnaire par **dictionnaire[clé] = valeur**.
- On supprime une clé et sa valeur avec le mot-clé **del** ou la méthode **pop**.
- On parcourt un dictionnaire avec les méthodes **keys()**, **values()** et **items()**.
- On capture les paramètres nommés passés à une fonction avec la syntaxe **def nom_fonction(**parametres_nommes** : (les paramètres nommés se trouvent dans le dictionnaire *parametres_nommes*).



Quiz

- [Python Dictionaries Quiz](#)
- [Python Dictionary Iteration Quiz](#)

1)

Fonctions toujours disponibles.

2)

Constructeur : un constructeur est, en programmation orientée objet, une fonction particulière appelée lors de l'instanciation. Elle permet d'allouer la mémoire nécessaire à l'objet et d'initialiser ses attributs.

From:

<https://webge.fr/dokuwiki/> - **WEBGE Wikis**

Permanent link:

<https://webge.fr/dokuwiki/doku.php?id=python:bases:dictionnaires&rev=1628697948>

Last update: **2021/08/11 18:05**

