BEST SELLER

Python:

100 points de connaissance essentiels pour les débutants

Édition 2024

Écrit par Hiroko Takahashi

Index

Indentation significative
Typage dynamique
Gestion automatique de la mémoire
Tout est un objet
Les fonctions anonymes (lambda)
Compréhensions de liste, de dictionnaire, et d'ensemble
Générateurs et itérateurs
Décorateurs
Gestion des exceptions
Modules et paquets
Tests d'adhésion avec in
Concaténation avec +
Slicing sur les séquences
Unpacking de séquences
Arguments nommés dans les fonctions
Arguments par défaut dans les fonctions
Arguments variables dans les fonctions
Typage statique optionnel
Les assertions en Python
La portée des variables en Python: LEGB
Descripteurs et propriétés
Méta-classes
Polymorphisme en Python
Programmation fonctionnelle: map, filter, reduce
Expressions régulières
Gestion de contexte avec with
Protocole d'itération
Multiparadigme : impératif, orienté objet, fonctionnel
Encodage par défaut en UTF-8
Les compréhensions de liste sont plus rapides que les boucles
Utiliser is pour comparer avec None
Les chaînes sont immuables
Python 3.x contre Python 2.x
pip pour la gestion des paquets
Gestion des environnements avec virtualenv
Respect de PEP 8 pour un style de codage cohérent
IPython/Jupyter pour l'expérimentation interactive
Debugging avec PDB
Tests unitaires en Python
Docstrings en Python
Typage dynamique mais fort
Utilisation de _ pour ignorer des valeurs

F-strings pour le formatage de chaînes
Les ensembles (set) pour des opérations uniques et rapides
Le Duck Typing
La bibliothèque standard riche
Les compréhensions de générateur
asyncio pour la programmation asynchrone
Extension Python avec des fichiers .pyd
Annotations de type en Python
L'opérateur Walrus
Pattern Matching
Utilisation de enumerate dans les boucles
Les tuples nommés avec namedtuple
Les classes de données en Python
L'ordre des dictionnaires en Python
Opérateur Morse pour les affectations dans les expressions
L'assignation multiple pour échanger des valeurs
str vs repr en Python
Les listes liées en Python
Gestion des paquets avec conda
La manipulation de données avec pandas
Le calcul scientifique avec numpy
La visualisation de données avec matplotlib
Développement Web avec Django
Tests d'intégration continue avec GitHub Actions
Le formatage automatique avec black
La vérification du type avec mypy
Verrou Global de l'Interpréteur (GIL)
Utilisation de threads et multiprocessing
L'environnement Jupyter pour les notebooks interactifs
L'utilisation de l'indexation négative dans les séquences
Le slicing avec indices négatifs
Le ramasse-miettes et le comptage de référence
Gestion des ressources avec les context managers
Syntaxe de l'opérateur ternaire
Les énumérations avec Enum
L'importation conditionnelle de modules
La bibliothèque standard Python et ses modules
Manipulation des chemins de fichiers avec pathlib
Interpolation de chaînes en Python
Décorateurs en Python
La programmation par contrat avec les assertions
La sérialisation avec pickle et json
Les générateurs pour la création de séquences paresseuses
L'héritage multiple

Les slots de classe pour économiser de la mémoire
La copie profonde et la copie superficielle avec copy
Les expressions lambda
La bibliothèque asyncio
Utilisation avancée des listes et dictionnaires en Python
Les namespaces et la portée des variables en Python
Manipulation des dates et heures avec datetime
Programmation réseau avec sockets
Manipulation de chaînes de caractères en Python
Surcharge des opérateurs en Python
Utilisation des bibliothèques tierces
Écriture de scripts exécutables
La documentation et les commentaires pour la clarté
La manipulation d'erreurs et les exceptions personnalisées
Les tests unitaires pour la fiabilité du code
Le profiling et le debugging pour la performance
Les conventions de nommage pour la lisibilité
L'internationalisation et la localisation

Introduction

Dans ce monde en constante évolution, la programmation est devenue une compétence inc ontournable pour de nombreux professionnels. Au cœur de cette révolution numérique, Pyth on s'impose comme un langage essentiel, grâce à sa simplicité et à sa polyvalence.

Ce guide est conçu spécifiquement pour ceux qui ont déjà une compréhension de base des concepts de programmation et souhaitent se concentrer exclusivement sur les connaissance s essentielles à maîtriser en Python. Notre objectif est de fournir un contenu précis et direct, permettant une acquisition rapide et efficace des compétences nécessaires.

Que vous soyez un débutant en Python cherchant à établir une solide fondation, ou un vétér an désireux de rafraîchir et d'actualiser vos connaissances sur les dernières évolutions du la ngage, ce livre est fait pour vous.

Nous espérons sincèrement que ce guide vous sera utile. Votre feedback est précieux pour nous et pour la communauté des développeurs qui, comme vous, cherchent à progresser. N ous vous invitons chaleureusement à laisser un commentaire ou une revue. Cela aide non s eulement à améliorer ce livre, mais aussi à le faire découvrir à d'autres ingénieurs qui pourra ient en bénéficier.

Merci de nous accompagner dans cette aventure d'apprentissage en Python.

1

Indentation significative

En Python, l'indentation n'est pas une question de style, mais elle est syntaxiquement obliga toire pour définir des blocs de code.

Considérons un exemple simple avec une condition if pour illustrer l'importance de l'indentati on en Python.

[Code]

```
if True:
    print("Ceci est vrai.")
else:
    print("Ceci est faux.")
```

[Result]

```
Ceci est vrai.
```

Dans d'autres langages comme C, Java, ou JavaScript, les blocs de code sont définis par d es accolades {}. En Python, ce sont les indentations qui définissent ces blocs. Une indentati on incorrecte peut entraîner des erreurs de syntaxe ou des comportements inattendus du pr ogramme. Par défaut, Python utilise 4 espaces pour une indentation, mais vous pouvez aus si utiliser des tabulations. Il est crucial que l'indentation soit cohérente dans tout le bloc de c ode. Cela rend également le code Python très lisible, mais cela signifie aussi que vous deve z faire attention à la manière dont vous structurez votre code.

[Trivia]
L'importance de l'indentation en Python découle de la philosophie du langage qui privilégie l a lisibilité et la clarté du code. Cela contraste avec d'autres langages où l'indentation est sou vent une question de préférence personnelle ou de convention d'équipe, sans impact sur le f onctionnement du code.

2

Typage dynamique

Python est un langage à typage dynamique, ce qui signifie que le type des variables est déte rminé à l'exécution et peut changer.

Illustrons cela avec un exemple où nous changeons le type d'une variable de int à str.

[Code]

```
x = 10
print(type(x))
x = "Bonjour"
print(type(x))
```

[Result]

```
<class 'int'>
<class 'str'>
```

Dans l'exemple ci-dessus, x commence comme un entier (int) avec la valeur 10. Ensuite, x e st réassigné à une chaîne de caractères (str) avec la valeur "Bonjour". Cela montre que le ty pe de x peut changer dynamiquement à l'exécution. En Python, vous n'avez pas besoin de d éclarer explicitement le type d'une variable lors de sa création; le langage détermine le type automatiquement basé sur la valeur assignée à la variable. Cette flexibilité facilite l'écriture d e code mais peut aussi mener à des bugs difficiles à trouver si les variables changent de typ e de manière inattendue. Il est donc important de bien comprendre les opérations que vous effectuez sur vos variables pour éviter des erreurs subtiles.

[Trivia]
Le typage dynamique de Python permet un développement rapide et une grande flexibilité d ans la manipulation des données. Cependant, cela peut aussi introduire des erreurs à l'exéc ution si les types ne sont pas gérés prudemment. Pour aider à éviter ces problèmes, Python 3.5 et versions ultérieures introduisent des annotations de type optionnelles, permettant aux développeurs de spécifier les types attendus des variables, bien que l'interpréteur Python ne les applique pas strictement.

3

Gestion automatique de la mémoire

En Python, la gestion de la mémoire est automatique. Le ramasse-miettes de Python s'occupe de libérer la mémoire non utilisée pour vous.

Exemple simple montrant la création et la suppression d'un objet en Python.

[Code]

```
class Fruit:
    def __init__(self, nom):
        self.nom = nom
    def __del__(self):
        print(f"{self.nom} a été supprimé de la mémoire.")
# Création d'un objet Fruit
fruit = Fruit("Pomme")
# Suppression explicite de l'objet
del fruit
```

[Result]

```
Pomme a été supprimé de la mémoire.
```

En Python, les objets sont créés et détruits automatiquement. Quand un objet n'est plus référencé par aucune variable ou qu'il est explicitement supprimé avec del, Python appelle automatiquement le destructeur de l'objet (__del__), libérant ainsi la mémoire. Cette gestion automatique simplifie le développement, car le programmeur n'a pas à se soucier de la libération manuelle de la mémoire, contrairement à des langages comme C ou C++ où la gestion de la mémoire est une préoccupation majeure. Le ramasse-miettes de Python travaille en coulisse pour nettoyer et réallouer efficacement la mémoire, réduisant ainsi les fuites de mémoire et les erreurs de segmentation.

[Trivia]
La gestion automatique de la mémoire en Python utilise principalement deux techniques: le comptage de références et le ramassage de cycles, ce qui lui permet de détecter et de libérer les références circulaires qui ne seraient pas libérées autrement.

4

Tout est un objet

En Python, tout est représenté comme un objet, y compris les fonctions et les classes.

Exemple montrant comment les fonctions sont des objets pouvant être assignés à des variables et passés comme arguments.

[Code]

```
def saluer():
    return "Bonjour tout le monde!"
# Assignation de la fonction à une variable
message = saluer
# Utilisation de cette variable pour appeler la fonction
print(message())
```

[Result]

```
Bonjour tout le monde!
```

Dans cet exemple, saluer est une fonction définie par l'utilisateur qui retourne un simple message. En Python, puisque tout est un objet, cette fonction peut être assignée à une variable (message dans ce cas) et être appelée à travers cette variable, comme n'importe quel objet. Cette propriété permet une grande flexibilité et puissance dans la conception de programmes, rendant Python extrêmement dynamique. Par exemple, les fonctions peuvent être passées en tant qu'arguments à d'autres fonctions, stockées dans des listes, et plus encore. Cela ouvre la porte à des concepts avancés comme les décorateurs et les fermetures, enrichissant ainsi les possibilités offertes par le langage.

[Trivia]
L'approche "tout est un objet" en Python facilite la réflexion (introspection) et la métaprogrammation, permettant aux développeurs de manipuler le comportement du programme de manière dynamique, d'examiner les types et les membres des objets à l'exécution, et de créer des frameworks et bibliothèques flexibles et puissants.

5

Les fonctions anonymes (lambda)

En Python, une fonction anonyme est une fonction définie sans nom, utilisant le mot-clé lam bda.

Voici comment créer et utiliser une fonction lambda pour additionner deux nombres.

[Code]

```
# Définition d'une fonction lambda pour additionner deux nombres
addition = lambda x, y: x + y
# Utilisation de cette fonction
resultat = addition(5, 3)
print(resultat)
```

[Result]

```
8
```

Une fonction lambda en Python est définie avec le mot-clé lambda, suivi par les arguments d e la fonction, un deux-points, et enfin l'expression qui est évaluée et retournée. Cette syntax e permet de créer des fonctions courtes et simples sans avoir besoin de la syntaxe plus lour de def. Les fonctions lambda sont souvent utilisées pour des opérations simples, comme pa sser une fonction en argument à une autre fonction. Par exemple, elles sont fréquemment ut ilisées avec des fonctions comme filter(), map(), et sorted() pour appliquer une opération sim ple à une collection d'éléments.

[Trivia]
Les fonctions lambda sont appelées "anonymes" parce qu'elles n'ont pas besoin d'être nom mées. Cela les rend utiles pour la création rapide de petites fonctions à utiliser à l'instant, sp écialement dans les cas où la fonction ne sera utilisée qu'une seule fois.

6

Compréhensions de liste, de dictionnaire, et d'ensemble

Les compréhensions en Python sont une manière concise de créer des listes, des dictionnaires, ou des ensembles basés sur des séquences existantes.

Voici des exemples de compréhensions de liste, de dictionnaire, et d'ensemble pour créer de nouvelles collections à partir de séquences existantes.

[Code]

```
# Compréhension de liste pour obtenir les carrés des nombres de 1 à 5
carres = [x**2 for x in range(1, 6)]
print(carres)
# Compréhension de dictionnaire pour associer chaque nombre à son carré
carres_dict = {x: x**2 for x in range(1, 6)}
print(carres_dict)
# Compréhension d'ensemble pour obtenir les carrés uniques des nombres de -5 à 5
carres_ensemble = {x**2 for x in range(-5, 6)}
print(carres_ensemble)
```

[Result]

```
[1, 4, 9, 16, 25]
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
{0, 1, 4, 9, 16, 25}
```

Les compréhensions sont une des fonctionnalités les plus puissantes et élégantes de Python, permettant de construire de nouvelles listes, dictionnaires, ou ensembles de manière très concise et lisible. La syntaxe de base est [expression for item in iterable if condition] pour les listes, {clé: valeur for item in iterable if condition} pour les dictionnaires, et {expression for item in iterable if condition} pour les ensembles. La partie if condition est optionnelle et permet de filtrer les éléments de la séquence source qui ne satisfont pas la condition spécifiée.

[Trivia]
L'utilisation de compréhensions peut rendre votre code plus Pythonique, c'est-à-dire plus clair, plus efficace, et plus aligné avec les conventions de style de Python. Elles peuvent souven

t remplacer des boucles for plus longues et plus complexes, rendant le code non seulement plus concis mais souvent aussi plus rapide.

7

Générateurs et itérateurs

Les générateurs sont des outils permettant de créer des itérateurs de manière très intuitive e t efficace en Python. Un itérateur, c'est un objet qui permet de parcourir, un par un, les élém ents d'un conteneur (comme une liste ou un dictionnaire), et cela, sans avoir besoin de les c harger tous en mémoire.

Imaginons que vous vouliez créer une suite de nombres. Au lieu de les générer tous en une fois et de les stocker dans une liste (ce qui pourrait consommer beaucoup de mémoire si la li ste est très longue), vous pouvez utiliser un générateur.

[Code]

```
def compte_jusqu_a(n):
    i = 0
    while i < n:
        yield i
        i += 1
# Création d'un générateur
mon_generateur = compte_jusqu_a(3)
# Itération sur le générateur
for nombre in mon_generateur:
    print(nombre)
```

[Result]

```
Copy code0
1
2
```

Dans cet exemple, compte_jusqu_a est une fonction génératrice. Lorsqu'elle est appelée, ell e ne s'exécute pas immédiatement. À la place, elle retourne un générateur qui, lui, peut être itéré. Lorsque l'on passe à l'élément suivant du générateur (avec la boucle for, par exemple), la fonction s'exécute jusqu'à ce qu'elle rencontre l'instruction yield. Cette instruction renvoie l a valeur à la boucle for, mais, très important, elle "met en pause" la fonction. La fonction ne s'arrête pas; elle attend juste le prochain appel pour continuer. Cela permet d'économiser de la mémoire, surtout avec de grandes quantités de données.

[Trivia]
Le mot-clé yield transforme une fonction normale en une fonction génératrice. L'utilisation de générateurs est particulièrement utile lorsque vous travaillez avec de grandes données ou des flux de données en temps réel, car ils permettent de réduire la consommation de mémoire.

8

Décorateurs

Les décorateurs sont un outil puissant en Python, permettant de modifier le comportement d'une fonction ou d'une méthode avant et après son exécution, sans en changer le code source.

Prenons un exemple simple de décorateur qui chronomètre le temps d'exécution d'une fonction. Cela peut être utile pour optimiser les performances de vos programmes.

[Code]

```
import time
# Décorateur pour mesurer le temps d'exécution d'une fonction
def chronometre(fonction):
    def enveloppe(*args, **kwargs):
        debut = time.time()
        resultat = fonction(*args, **kwargs)
        fin = time.time()
        print(f"Temps d'exécution: {fin - debut} secondes")
        return resultat
    return enveloppe
@chronometre
def somme(x, y):
    return x + y
# Test du décorateur
print(somme(5, 7))
```

[Result]

```
Temps d'exécution: 0.000001 secondes
12
```

Dans cet exemple, le décorateur @chronometre est appliqué à la fonction somme. Lorsque somme est appelée, au lieu d'exécuter directement somme, Python exécute chronometre(somme). Cela signifie que la fonction somme est passée en argument à la fonction chronometre, et c'est cette dernière qui est exécutée. Dans chronometre, nous définissons une fonction interne, enveloppe, qui exécute la fonction originale somme, tout en ajoutant avant et après