Notes de cours
Semaine 5

Cours Turing

Introduction

Cette semaine, nous allons nous intéresser à des sujets assez différents :
— La génération de nombres pseudo-aléatoires à l'aide du module `random` de Python.
— L'utilisation de l'aléatoire (ou pseudo-aléatoire) pour résoudre des problèmes.
— Les *dictionnaires*, une structure de données en Python.
— La définition de listes (et dictionnaires) par compréhension.

1 Génération de nombres pseudo-aléatoires avec `random`

Le module `random` de Python rassemble plusieurs fonctions en rapport avec l'aléatoire. Le module est inclus dans la bibliothèque standard de Python, il n'est donc pas nécessaire de l'installer séparément.

Dans cette section, nous allons voir comment utiliser ce module pour générer des nombres pseudo-aléatoires, choisir des éléments pseudo-aléatoires dans une liste, ou encore mélanger une liste. La documentation complète du module `random` est disponible à l'adresse suivante :

`https://docs.python.org/3/library/random.html`

1.1 Fonctions courantes

Le module `random` fournit plusieurs fonctions pour générer des nombres pseudo-aléatoires. Il contient aussi des fonctions pour mélanger des listes ou choisir des éléments aléatoires dans une séquence.

Voici quelques fonctions couramment utilisées :

— `random.random()` : génère un nombre flottant aléatoire dans l'intervalle $[0.0, 1.0)$.
— `random.randint(a, b)` : génère un entier aléatoire entre `a` et `b` (inclus).
— `random.choice(xs)` : choisit un élément aléatoire dans la séquence `xs`.
— `random.shuffle(xs)` : mélange la liste `xs`.

Exemple

Voici un exemple d'utilisation du module `random` :

```python
import random

# Affiche un nombre à virgule flottante aléatoire entre 0.0 et 1.0
print(random.random())

# Affiche un entier aléatoire entre 1 et 10 (compris)
print(random.randint(1, 10))

# Affiche un élément aléatoire dans une liste
fruits = ['pomme', 'banane', 'cerise']
print(random.choice(fruits))

# Mélange la liste
random.shuffle(fruits)

# Affiche la liste mélangée
print(fruits)
```

Le résultat de ce programme sera différent à chaque exécution. Essayez de l'exécuter plusieurs fois pour voir les différences !

1.2 Le *seed*

Les générateurs de nombres pseudo-aléatoires utilisent un algorithme déterministe pour produire une séquence de nombres qui semblent aléatoires. En partant d'une même valeur initiale, appelée *seed*, le générateur produira toujours la même séquence de nombres.

On peut définir la *seed* en utilisant la fonction `random.seed(value)`. Cela est utile pour reproduire des résultats lors de tests ou de débogage.

Exemple

Voici un exemple montrant l'effet du *seed* :

```
import random

# Définir le seed
random.seed(42)

# Générer une séquence de nombres aléatoires
for _ in range(5):
    print(random.randint(1, 100))

print("---- Réinitialisation du seed ----")

# Réinitialiser le seed
random.seed(42)
# Générer à nouveau la même séquence de nombres aléatoires
for _ in range(5):
    print(random.randint(1, 100))
```

Les deux boucles afficheront la même séquence de nombres :

```
82
15
4
95
36
---- Réinitialisation du seed ----
82
15
4
95
36
```

1.3 Autres distributions

Le module `random` fournit aussi des fonctions pour générer des nombres selon différentes distributions statistiques. Vous pouvez par exemple générer des nombres suivant une distribution uniforme, normale (gaussienne), exponentielle, etc. Pour plus de détails, consultez la documentation du module `random`.

2 Utilisation de l'aléatoire en informatique

L'aléatoire est un outil puissant en informatique qui trouve des applications dans des domaines très variés. Quelques exemples de domaines d'application de l'aléatoire sont discutés ci-dessous.

2.1 Simulation

L'aléatoire est souvent utilisé pour simuler des phénomènes complexes dans divers domaines, tels que la physique, la biologie, l'économie, etc. Par exemple, on peut utiliser des nombres pseudo-aléatoires pour modéliser le comportement de particules dans un gaz, la propagation d'une épidémie, ou encore les fluctuations du marché boursier. Des modèles probabilistes sont souvent employés pour représenter ces phénomènes, et la génération de nombres aléatoires permet de simuler des scénarios variés.

2.2 Algorithmique

Certains algorithmes utilisent la génération de nombres pseudo-aléatoires pour obtenir des résultats approximatifs de manière plus rapide que des algorithmes déterministes. Par exemple, comme on le verra en exercices, il est possible d'estimer la valeur de π en utilisant une méthode probabiliste appelée la méthode de Monte Carlo.

Contenu avancé

Le hasard peut aussi être utilisé dans la résolution de jeux, comme les échecs ou le go, où des algorithmes comme Monte Carlo Tree Search (MCTS) explorent les coups possibles en simulant des parties aléatoires pour évaluer les positions. Dans ce contexte, l'aléatoire permet de sonder efficacement un grand espace de possibilités de parties.

Contenu avancé

D'autres algorithmes utilisent l'aléatoire pour éviter des situations pathologiques dans lesquelles un algorithme déterministe pourrait être inefficace. C'est le cas par exemple de l'algorithme de tri rapide (*quicksort*) qui choisit un pivot aléatoire pour diviser la liste à trier. Dans ce genre d'algorithme, le résultat n'est pas affecté par l'aléatoire, mais la performance de l'algorithme peut être améliorée en utilisant un choix aléatoire.

2.3 Optimisation

L'aléatoire est également utilisé dans des algorithmes d'optimisation, tels que les algorithmes génétiques ou le recuit simulé (*simulated annealing*). Ces algorithmes s'inspirent de processus naturels pour explorer l'espace des solutions possibles et trouver de bonnes solutions à des problèmes complexes. L'utilisation de l'aléatoire permet de diversifier la recherche et d'éviter de rester bloqué dans des optima locaux.

2.4 Cryptographie

Finalement, un autre domaine où l'aléatoire est crucial est la cryptographie. Les systèmes de chiffrement modernes reposent sur des clés secrètes qui doivent être générées de manière (pseudo-)aléatoire pour garantir la sécurité des communications. Vous aurez l'occasion d'en apprendre davantage sur ce sujet dans le cadre du prochain module de ce cours. Vous y verrez notamment des méthodes de génération de nombres pseudo-aléatoires.

Contenu avancé

Il est important de noter que pour des applications en cryptographie, les générateurs de nombres pseudo-aléatoires standards tels que ceux fournis par le module `random` de Python ne sont pas adaptés, car ils peuvent être prédits par un attaquant.
Des générateurs de nombres aléatoires cryptographiquement sécurisés, comme ceux disponibles dans le module `secrets` de Python, doivent être utilisés à la place.

3 Dictionnaires en Python

Un dictionnaire en Python est une structure de données qui permet de stocker des paires clé-valeur. Chaque clé est unique et est associée à une valeur.

3.1 Création de dictionnaires et accès aux valeurs

Voici comment créer un dictionnaire en Python :

```
# Création d'un dictionnaire vide
mon_dictionnaire = {}

# Création d'un dictionnaire avec des paires clé-valeur
mon_dictionnaire = {'clé1': 'valeur1', 'clé2': 'valeur2'}

# Accès à une valeur via sa clé
valeur = mon_dictionnaire['clé1']  # valeur sera 'valeur1'
```

La méthode `get` permet aussi d'accéder à une valeur en fournissant une clé. Contrairement à l'accès direct via les crochets, `get` ne génère pas d'erreur si la clé n'existe pas dans le dictionnaire. Il est possible de spécifier qui sera retournée si la clé n'existe pas. Si aucune valeur par défaut n'est fournie, la valeur `None` sera retournée en cas de clé inexistante.

```
mon_dictionnaire = {'clé1': 'valeur1', 'clé2': 'valeur2'}

# Accès à une valeur via sa clé avec get
valeur = mon_dictionnaire.get('clé1', 'valeur_par_défaut')
print(valeur)  # Affiche valeur1

valeur_inexistante = mon_dictionnaire.get('clé_inexistante', 'valeur_par_défaut')
print(valeur_inexistante)  # Affiche valeur_par_défaut
```

3.2 Modification de dictionnaires

Les dictionnaires sont des structures mutables, ce qui signifie que vous pouvez ajouter, modifier ou supprimer des paires clé-valeur après leur création.

```
# Création d'un dictionnaire
mon_dictionnaire = {'clé1': 'valeur1', 'clé2': 'valeur2'}

# Ajout d'une nouvelle paire clé-valeur
mon_dictionnaire['clé3'] = 'valeur3'

# Modification d'une valeur existante
mon_dictionnaire['clé1'] = 'nouvelle_valeur1'
```

```python
# Suppression d'une paire clé-valeur
del mon_dictionnaire['clé2']

# Affichage du dictionnaire
print(mon_dictionnaire)  # Affiche {'clé1': 'nouvelle_valeur1', 'clé3': 'valeur3'}
```

3.3 Itération sur les dictionnaires

Vous pouvez itérer sur les clés, les valeurs ou les paires clé-valeur d'un dictionnaire en utilisant des boucles `for`. Par défaut, itérer sur un dictionnaire parcourt ses clés. Il est aussi possible d'utiliser les méthodes `keys()`, `values()` et `items()` pour obtenir respectivement les clés, les valeurs et les paires clé-valeur.

```python
mon_dictionnaire = {'clé1': 'valeur1', 'clé2': 'valeur2'}

# Itération sur les clés
for cle in mon_dictionnaire:
    print(cle)

# Autre façon d'itérer sur les clés
for cle in mon_dictionnaire.keys():
    print(cle)

# Itération sur les valeurs
for valeur in mon_dictionnaire.values():
    print(valeur)

# Itération sur les paires clé-valeur
for cle, valeur in mon_dictionnaire.items():
    print(cle, valeur)
```

3.4 Applications des dictionnaires

Les dictionnaires sont très utiles pour représenter des données structurées, comme par exemple des informations sur des personnes, des produits, etc.

```python
# Représentation d'une personne
personne1 = {
    'nom': 'Martin',
    'prénom': 'Claire',
    'âge': 25,
}

personne2 = {
    'nom': 'Dupont',
    'prénom': 'Jean',
```

```
    'âge': 30,
}
```

Dans ce genre d'utilisation, chaque entrée du dictionnaire représente un attribut de l'object (la personne, le produit, etc.), avec la clé correspondant au nom de l'attribut et la valeur correspondant à la valeur de l'attribut.

Les dictionnaires sont aussi très utiles pour associer des données à des clés qui proviennent d'une collection de choses, comme par exemple par associer un numéro de téléphone à des noms, ou une population à des villes.

```
# Dictionnaire associant des noms à des numéros de téléphone
annuaire = {
    'Alice': '079 123 45 67',
    'Bob': '078 234 56 78',
}

# Dictionnaire associant des villes à leur population
populations = {
    'Berne': 146348,
    'Zurich': 448664,
    'Genève': 203856,
}
```

Les dictionnaires sont aussi très utilisés aussi lors de calculs, par exemple pour compter des occurrences d'éléments dans une liste.

```
# Compter les occurrences de chaque fruit dans une liste
fruits = ['pomme', 'banane', 'orange', 'pomme', 'orange', 'banane', 'pomme']
compteur = {}
for fruit in fruits:
    if fruit in compteur:
        compteur[fruit] += 1
    else:
        compteur[fruit] = 1
print(compteur)  # Affiche {'pomme': 3, 'banane': 2, 'orange': 2}
```

Contenu avancé

Les dictionnaires sont aussi très utilisés comme *caches* pour mémoriser des résultats de calculs coûteux. Voici un exemple simple :

```
# Fonction pour calculer le n-ième nombre de Fibonacci
cache = {}
def fibonacci(n):
    if n in cache:
        return cache[n]
    if n <= 1:
        return n
    result = fibonacci(n - 1) + fibonacci(n - 2)
    cache[n] = result
    return result

print(fibonacci(500))
```

Dans cet exemple, le dictionnaire `cache` est utilisé pour mémoriser les résultats des appels précédents à la fonction `fibonacci`. Cela permet d'éviter de recalculer plusieurs fois les mêmes valeurs, ce qui améliore considérablement les performances de la fonction.

4 Compréhensions de listes et de dictionnaires

Les compréhensions de listes et de dictionnaires sont des syntaxes concises pour créer des listes ou des dictionnaires en Python.

4.1 Compréhensions de listes

Une compréhension de liste permet de créer une nouvelle liste en appliquant une expression à chaque élément d'une séquence existante. Mais le mieux est peut-être de voir un exemple !

```
# Création d'une liste des carrés des nombres de 0 à 9
carrés = [x**2 for x in range(10)]
print(carrés)  # Affiche [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Il est aussi possible d'ajouter une condition pour filtrer les éléments de la séquence.

```
# Création d'une liste des carrés des nombres pairs de 0 à 9
carrés_pairs = [x**2 for x in range(10) if x % 2 == 0]
print(carrés_pairs)  # Affiche [0, 4, 16, 36, 64]
```

Les compréhensions de listes sont souvent plus concises et lisibles que les boucles traditionnelles pour créer des listes. Comparez par exemple avec la version utilisant une boucle `for` :

```
# Version avec compréhension de liste
puissances_de_2 = [2**x for x in range(10)]
print(puissances_de_2)  # Affiche [1, 2, 4, 8, 16, 32, 64, 128, 256, 512]

# Version avec boucle for
puissances_de_2 = []
for x in range(10):
    puissances_de_2.append(2**x)
print(puissances_de_2)  # Affiche [1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

Bien entendu, l'utilisation de `range` dans les exemples ci-dessus n'est pas obligatoire. On peut utiliser n'importe quelle séquence, comme une liste ou une chaîne de caractères.

```
# Création d'une liste des lettres en majuscules dans une chaîne de caractères
chaine = "Quelle Belle Journée"

print([c.lower() for c in chaine if c.isupper()])  # Affiche ['q', 'b', 'j']
```

Les compréhensions de listes sont souvent utilisées en conjonction avec les différentes fonctions de la bibliothèque standard de Python pour manipuler (`enumerate`, `zip`, etc.) ou réduire des séquences (`sum`, `all`, `any`, etc.). Le code résultant est souvent très concis et lisible.

```
prix = [20, 10, 5, 15, 30]
quantites = [2, 0, 1, 4, 3]

total = sum([p * q for p, q in zip(prix, quantites)])
print(total)  # Affiche 195
```

Contenu avancé

Les compréhensions de listes peuvent également être imbriquées pour créer des listes à partir de plusieurs séquences. Voici un exemple :

```
# Création d'une liste de tous
# la chaine de caractères i + j
# avec i dans ["0", "1", "2"]
# et j dans ['A', 'B']
produits = [
    str(i) + j
    for i in range(3)
    for j in "AB"
]
print(produits)  # Affiche ['0A', '0B', '1A', '1B', '2A', '2B']
```

Il est aussi possible d'ajouter des conditions dans les compréhensions imbriquées.

```
# Création d'une liste de tous
# les couples (i, j)
# avec i dans [0, 1, 2]
# et j dans [0, 1, 2]
# où i + j est pair
somme_paires = [
    (i, j)
    for i in range(3)
    for j in range(3)
    if (i + j) % 2 == 0
]
print(somme_paires)  # Affiche [(0, 0), (0, 2), (1, 1), (2, 0), (2, 2)]

# Création d'une liste de tous
# les couples (i, j)
# avec i dans [0, 1, 2]
# où i est pair,
# et j dans [0, 1, 2]
# où j est pair
deux_pairs = [
    (i, j)
    for i in range(3)
    if i % 2 == 0
    for j in range(3)
    if j % 2 == 0
]
print(deux_pairs)  # Affiche [(0, 0), (0, 2), (2, 0), (2, 2)]
```

4.2 Compréhensions de dictionnaires

Les compréhensions de dictionnaires permettent de créer des dictionnaires de manière concise, similaire aux compréhensions de listes.

Voici un exemple :

```
# Création d'un dictionnaire des carrés des nombres de 0 à 3
carrés_dict = {x: x**2 for x in range(4)}
print(carrés_dict)  # Affiche {0: 0, 1: 1, 2: 4, 3: 9}
```

Tout comme pour les compréhensions de listes, il est possible d'ajouter une condition pour filtrer les éléments.

```
# Création d'un dictionnaire des carrés des nombres pairs de 0 à 3
carrés_pairs_dict = {x: x**2 for x in range(4) if x % 2 == 0}
print(carrés_pairs_dict)  # Affiche {0: 0, 2: 4}
```

La compréhensions de dictionnaires permettent de facilement convertir des listes de paires clé-valeur en dictionnaires.

```
cles = ['a', 'b', 'c']
valeurs = [1, 2, 3]
cles_valeurs = zip(cles, valeurs)
dictionnaire = {k: v for k, v in cles_valeurs}
print(dictionnaire)  # Affiche {'a': 1, 'b': 2, 'c': 3}
```