Notes de cours
Semaine 4

Cours Turing

Introduction

Lors de cette quatrième semaine de cours, nous allons aborder des aspects importants de la programmation en Python qui vont au-delà du langage de programmation. Jusqu'à présent, nous avons appris les bases de la syntaxe Python, les types de données fondamentaux, les structures de contrôle et les fonctions. Nous allons nous intéresser aujourd'hui non pas à de nouvelles constructions du langage, mais plutôt à des pratiques et des conventions qui permettent d'écrire du code de meilleure qualité.

Nous allons aborder les sujets suivants :

— Guide de style Python
— Nomenclature
— Documentation en Python (commentaires, docstrings, annotations de type)
— Tests unitaires

Ces différents sujets devraient vous permettre de découvrir un aspect plus pratique de la programmation en Python. Ils vous aideront à écrire du code plus lisible, plus maintenable et plus fiable. Ils sont particulièrement importants lorsque vous travaillez en équipe ou sur des projets de grande envergure ou complexité. Je vous encourage vivement à appliquer ces bonnes pratiques dans vos projets de programmation futurs.

1 Guide de style Python

Un guide de style est un ensemble de conventions et de recommandations visant à améliorer la lisibilité, la cohérence et la maintenabilité du code. En Python, le guide de style le plus largement adopté est le PEP 8 : `https://peps.python.org/pep-0008/`. Le PEP 8 couvre divers aspects du style de code, notamment la mise en forme du code, la nomenclature, les conventions de codage et les bonnes pratiques. Je vous invite à aller le consulter pour plus de détails.

Voici quelques recommandations clés du PEP 8 :

- **Indentation :** Utiliser 4 espaces par niveau d'indentation. Éviter les tabulations.
- **Longueur des lignes :** Limiter la longueur des lignes à 79 caractères pour le code et 72 caractères pour les commentaires et les docstrings.
- **Espaces dans le code :** Utiliser des espaces autour des opérateurs (par exemple, `x = 1 + 2`) et après les virgules. Éviter les espaces inutiles à l'intérieur des parenthèses, crochets ou accolades.
- **Nomenclature :** Utiliser des noms explicites et descriptifs pour les variables.
- **Commentaires :** Utiliser des commentaires pour expliquer le code, mais éviter les commentaires évidents. Les commentaires doivent être clairs et concis.
- **Docstrings :** Utiliser des docstrings pour documenter les fonctions. Les docstrings doivent décrire brièvement ce que fait la fonction documentée, ses paramètres et sa valeur de retour.
- **Imports :** Grouper les imports en trois sections : imports standard, imports de bibliothèques tierces, et imports locaux. Chaque section doit être séparée par une ligne vide.

Le but d'un guide de style est de garantir que le code est facile à lire et à comprendre, même pour les développeurs qui ne l'ont pas écrit. En suivant un guide de style, on facilite la collaboration entre plusieurs personnes travaillant sur le même projet. Certains des points mentionnés ci-dessus seront abordés plus en détail dans les sections suivantes.

2 Nomenclature

La nomenclature est le terme utilisé pour décrire la façon de nommer les variables, fonctions et autres éléments dans le code. Une bonne nomenclature est essentielle pour rendre le code lisible et compréhensible.

2.1 Variables

Les noms de variables doivent être descriptifs et refléter clairement leur contenu ou leur rôle dans le programme.

2.1.1 Syntaxe

En Python, les noms de variables doivent respecter les règles suivantes :
— Ils doivent commencer par une lettre (a-z, A-Z) ou un underscore (_).
— Ils peuvent contenir des lettres, des chiffres (0-9) et des underscores.
— Ils sont sensibles à la casse (par exemple, `variable` et `Variable` sont deux variables différentes).
— Ils ne doivent pas être des mots réservés de Python (comme `if`, `for`, `while`, etc.).

2.1.2 Conventions

En Python, la convention de nommage recommandée pour les variables est le *snake_ case*, où les mots sont en minuscules et séparés par des underscores. Par exemple : `total_price`, `user_name`, `item_count`.

La plupart du temps, on préférera des noms de variables formés de mots complets plutôt que des abréviations. Par exemple, on utilisera `total_price` au lieu de `tp`.

Cependant, pour des variables temporaires ou dans des contextes très locaux, des noms courts comme `i`, `j` ou `temp` peuvent être acceptables. Ainsi, dans une boucle `for`, il est courant d'utiliser `i` comme variable d'itération :

```
for i in range(10):
    print(i)
```

De même, pour des variables représentant des valeurs abstraites ou mathématiques, des noms comme `x`, `y` ou `z` sont couramment utilisés.

```
def calculate_distance(p1, p2):
    x1, y1 = p1
    x2, y2 = p2
    return ((x2 - x1) ** 2 + (y2 - y1) ** 2) ** 0.5

point_a = (1, 2)
point_b = (4, 6)
distance = calculate_distance(point_a, point_b)
print("Distance:", distance)
```

Dans le cas de variables contenant des collections (listes, dictionnaires, ensembles), il est souvent utile d'utiliser des noms au pluriel pour indiquer qu'elles contiennent plusieurs éléments. Par exemple : `users`, `items`, `prices`. Cela permet de clarifier le type de données que la variable contient. Cela permet aussi d'utiliser des noms au singulier pour les éléments individuels lors du parcours de la collection à l'aide d'une boucle.

```
users = ["Alice", "Bob", "Charlie"]
for user in users:
    print("Hello,", user)
```

De même, pour des collections abstraites, on peut utiliser des noms comme `values` ou même `xs`, `ys` et `zs`.

```
def map_list(f, xs):
    result = []
    for x in xs:
        result.append(f(x))
    return result
```

2.2 Fonctions

Les fonctions doivent également avoir des noms descriptifs qui indiquent clairement leur objectif ou leur action.

2.2.1 Syntaxe

Les noms de fonctions suivent les mêmes règles que les noms de variables.

2.2.2 Conventions

En Python, la convention de nommage recommandée pour les fonctions est également le *snake_ case*. Par exemple : `calculate_total`, `get_user_info`, `process_data`.

Les noms de fonctions doivent généralement être des verbes ou des phrases verbales qui décrivent l'action effectuée par la fonction. Par exemple : `send_email`, `fetch_data`, `save_record`. Cela est très courant pour les fonctions avec des effets de bord (modification d'une variable globale, écriture dans un fichier, affichage à l'écran, etc.).

Pour les fonctions qui retournent une valeur sans effet de bord, on utilisera souvent un nom qui décrit la valeur retournée. Par exemple : `rectangle`, `sum`, `max_value`.

Dans le cas de fonctions qui retournent un booléen, il est courant d'utiliser des préfixes comme `is_`, `has_`, `can_` ou `should_`. Par exemple : `is_valid`, `has_permission`, `can_execute`.

Concernant les paramètres de fonctions, les mêmes conventions de nommage que pour les variables s'appliquent.

3 Documentation en Python

Documenter son code est une pratique essentielle en programmation. Une bonne documentation facilite la compréhension, la maintenance et la collaboration sur le code. En Python, il existe plusieurs façons de documenter son code. Nous allons aborder deux méthodes principales : les commentaires et les *docstrings*.

3.1 Commentaires

Les commentaires sont des lignes de texte dans le code qui ne sont pas exécutées par l'interpréteur Python. Ils sont utilisés pour expliquer le fonctionnement du code, fournir des informations supplémentaires ou marquer des sections importantes.

3.1.1 Syntaxe

En Python, les commentaires commencent par le symbole # et s'étendent jusqu'à la fin de la ligne. Les commentaires peuvent être placés sur une ligne à part ou à la fin d'une ligne de code. Lorsqu'un commentaire est placé à la fin d'une ligne de code, il est généralement précédé d'au moins deux espaces pour améliorer la lisibilité.

```
# Ceci est un commentaire sur une seule ligne

x = 10  # Ceci est un commentaire à la fin d'une ligne de code
```

Pour faire un commentaire sur plusieurs lignes, on place le symbole # au début de chaque ligne.

```
# Ceci est un commentaire
# sur plusieurs lignes
# Chaque ligne commence par un #
```

Les éditeurs de code, tels que VSCode, offrent souvent des raccourcis pour commenter ou décommenter rapidement des blocs de code.

3.1.2 Bonnes pratiques

Les commentaires doivent être utilisés judicieusement pour améliorer la compréhension du code. Les commentaires ne doivent pas être utilisés pour expliquer des choses évidentes ou redondantes. Par exemple, le commentaire suivant est inutile :

```
x = 10  # On stocke la valeur 10 dans la variable x
```

Il est préférable d'utiliser des commentaires pour expliquer le pourquoi du code, les décisions de conception ou les parties complexes. Par exemple :

```
frames = []

# Phase de chute
```

```
for i in range(0, 101):
    p = (i / 100) ** 2  # Progression quadratique (accélération)
    frames.append(frame(p, red, black))

# Phase de rebond
for i in range(99, 0, -1):  # 100 et 0 sont exclus (déjà présents)
    p = (i / 100) ** 2
    frames.append(frame(p, red, black))
```

Écrire des commentaires utiles n'est pas toujours facile et demande de la pratique. Je vous encourage vivement à écrire des commentaires dans votre code. Pour chaque commentaire, questionnez-vous sur son utilité. Évitez les paraphrases et privilégiez les explications qui éclairent les lecteurs sur les aspects importants du code. C'est en pratiquant cet exercice que vous deviendrez plus à l'aise avec l'écriture de commentaires pertinents.

3.2 Docstrings

En Python, les docstrings sont des chaînes de caractères utilisées pour documenter notamment les fonctions. Elles sont placées immédiatement après la définition de l'élément qu'elles documentent et sont délimitées par des triples guillemets (généralement doubles).

```
def ma_fonction(param1, param2):
    """Description de ce que fait la fonction.

    Plus de détails sur le fonctionnement de la fonction.

    Paramètres:
        param1: Description du paramètre 1.
        param2: Description du paramètre 2.

    Retourne:
        Description de la valeur de retour.
    """
    # Corps de la fonction
    pass
```

Les docstrings sont accessibles via l'attribut `.__doc__` de l'objet documenté. Par exemple :

```
print(ma_fonction.__doc__)
```

Les docstrings sont particulièrement utiles pour fournir des informations sur l'utilisation des fonctions, leurs paramètres, leurs valeurs de retour et tout autre détail pertinent. Elles sont également utilisées par des outils de documentation automatique pour générer des documents à partir du code source.

Il n'existe pas de format strictement imposé pour les docstrings, mais il est recommandé de suivre des conventions cohérentes. Les conventions changent suivant les différentes communautés, équipes, projets ou encore outils de documentation.

Par exemple, voici la docstring de la fonction `rectangle` de PyTamaro, telle qu'elle apparait dans le code source [1] :

```
def rectangle(width: float, height: float, color: Color) -> Graphic:
    """
    Creates a rectangle of the given size, filled with a color.

    :param width: width of the rectangle
    :param height: height of the rectangle
    :param color: the color to be used to fill the rectangle
    :returns: the specified rectangle as a graphic
    """
    # Reste du code...
```

Notez au passage l'utilisation d'*annotations de type* dans la définition de la fonction. Nous abordons ce sujet dans la section suivante.

3.3 Annotations de type

Les annotations de type en Python sont une fonctionnalité qui permet de spécifier les types de données attendus pour les paramètres d'une fonction et pour sa valeur de retour. Elles sont introduites en Python 3.5 et sont principalement utilisées pour améliorer la lisibilité du code et faciliter la détection d'erreurs potentielles lors du développement.

3.3.1 Syntaxe

Les annotations de type sont ajoutées à la définition d'une fonction en utilisant le symbole `:` après le nom du paramètre, suivi du type attendu.

```
def ma_fonction(param1: int, param2: str):
    # Corps de la fonction
    pass
```

Dans cet exemple, `param1` est annoté comme un entier (`int`) et `param2` comme une chaîne de caractères (`str`).

Pour annoter le type de retour d'une fonction, on utilise le symbole `->` suivi du type de retour attendu, placé avant les deux-points de la définition de la fonction.

```
def addition(a: int, b: int) -> int:
    return a + b
```

Ici, la fonction `addition` prend deux entiers en paramètres et retourne également un entier.

1. Accessible ici : `https://github.com/LuCEresearchlab/pytamaro/blob/main/pytamaro/primitives.py`

3.3.2 Utilisation

Les principales raisons d'utiliser des annotations de type sont les suivantes :

— **Documentation :** Elles servent de documentation intégrée au code, aidant les développeurs à comprendre rapidement les types de données attendus par une fonction.

— **Éditeurs de code :** De nombreux éditeurs de code (comme VSCode) utilisent les annotations de type pour fournir des fonctionnalités avancées, telles que l'autocomplétion, la détection d'erreurs en temps réel et la navigation dans le code.

— **Outils de vérification de type :** Des outils externes, comme *mypy*, peuvent être utilisés pour analyser le code et vérifier que les types utilisés sont cohérents avec les annotations. Ils peuvent être intégrés dans l'éditeur de code ou exécutés séparément.

Il est important de noter que les annotations de type en Python sont facultatives et n'affectent pas l'exécution du code.

3.3.3 Types courants

Voici quelques types couramment utilisés dans les annotations de type en Python :

— `int` : Entier
— `float` : Nombre à virgule flottante
— `str` : Chaîne de caractères
— `bool` : Booléen (True ou False)
— `list` : Liste
— `dict` : Dictionnaire
— `set` : Ensemble
— `tuple` : Tuple (une liste immuable)
— `None` : Absence de valeur utile

Les noms de *classes* peuvent également être utilisés comme types. Par exemple, dans PyTamaro, on peut annoter une fonction pour indiquer qu'elle retourne un objet de type `Graphic` ou qu'elle prend un paramètre de type `Color`.

```
from pytamaro import Graphic, Color

def square(size: float, color: Color) -> Graphic:
    return rectangle(size, size, color)
```

3.3.4 Types de collections

Pour les collections, comme les listes, il est possible de spécifier le type des éléments qu'elles contiennent en utilisant la syntaxe `list[Type]`. Par exemple, une liste d'entiers peut être annotée comme `list[int]`.

```
def my_sum(numbers: list[int]) -> int:
    return sum(numbers)
```

Pour les tuples, on peut spécifier les types de chaque élément en utilisant la syntaxe `tuple[Type1, Type2, ...]`. Par exemple, un tuple contenant un entier et une chaîne de caractères peut être annoté comme `tuple[int, str]`.

3.3.5 Unions de types

Il est possible d'indiquer qu'un paramètre ou une valeur de retour peut être de plusieurs types en utilisant la syntaxe `Type1 | Type2`. Par exemple, une fonction qui peut retourner soit un entier, soit une chaîne de caractères peut être annotée comme `int | str`.

```
def process(value: int | str) -> None:
    if isinstance(value, int):
        print("C'est un entier :", value)
    else:
        print("C'est une chaîne de caractères :", value)
```

3.3.6 Types optionnels

Parfois, on souhaite indiquer qu'un paramètre est optionnel, ou alors qu'une valeur de retour peut ne pas être présente. Dans les deux cas, on utilise généralement la valeur spéciale `None` pour représenter l'absence de valeur. Pour annoter un paramètre ou une valeur de retour qui peut être d'un certain type ou `None`, on utilise la syntaxe `Type | None`.

4 Tests

Les tests sont une partie essentielle du développement logiciel. Ils permettent de vérifier que le code fonctionne comme prévu et de détecter les erreurs. Il existe plusieurs types de tests, mais nous allons nous concentrer sur les tests *unitaires*, qui consistent à tester des unités individuelles de code. Dans notre cas, les unités de code que nous allons tester sont les *fonctions*.

4.1 Décomposition en fonctions

La décomposition en fonctions est une pratique de programmation qui consiste à diviser un programme en fonctions indépendantes et réutilisables. Chaque fonction doit avoir une responsabilité claire et accomplir une tâche spécifique.

En plus de faciliter la lecture et la maintenance du code, la décomposition en fonctions facilite également le processus de test du code. En effet, en isolant des parties spécifiques du code dans des fonctions, on peut tester chaque fonction individuellement, ce qui permet de détecter plus facilement les erreurs et de s'assurer que chaque partie du code fonctionne correctement.

4.2 Le module `unittest`

Python fournit un module intégré appelé `unittest` pour écrire et exécuter des tests unitaires. Voici un exemple simple de test unitaire utilisant le module `unittest` :

```
import unittest

from my_module import addition  # Importer la fonction à tester

# La classe de test
class TestAddition(unittest.TestCase):
    def test_add_positive_numbers(self):
        self.assertEqual(addition(2, 3), 5)

    def test_add_negative_numbers(self):
        self.assertEqual(addition(-2, -3), -5)

if __name__ == '__main__':
    unittest.main()
```

Dans cet exemple, nous avons une fonction `addition` que nous voulons tester et qui est définie dans un module appelé `my_module`. La classe `TestAddition` hérite de `unittest.TestCase` et contient deux méthodes de test : `test_add_positive_numbers` et `test_add_negative_numbers`. [2]

2. Le module `unittest` utilise des concepts de programmation orientée objet comme les classes et l'héritage. Nous n'avons pas encore abordé ces concepts dans le cours. Ne vous inquiétez pas si vous ne comprenez pas tout de suite comment cela fonctionne. L'important est de comprendre l'idée générale des tests unitaires et la façon dont ils sont écrits.

Lorsque vous exécutez ce script, le module `unittest` exécute automatiquement toutes les méthodes de test définies dans la classe `TestAddition`. Chaque méthode de test utilise des *assertions* pour vérifier que le résultat de la fonction `addition` est correct. Si une assertion échoue, le test est considéré comme ayant échoué.

Écrire des tests unitaires est une bonne pratique qui permet de détecter des erreurs rapidement. Cependant, il n'est pas toujours évident de trouver de bons tests à écrire. Quels cas de figure faut-il tester ? Il n'existe malheureusement pas de recette miracle. Voici quelques conseils pour vous aider à écrire de bons tests :

— Testez les cas normaux : Commencez par tester les cas d'utilisation de votre fonction anticipés comme les plus courants.
— Testez les cas limites : Pensez aux cas extrêmes ou aux valeurs limites qui pourraient poser problème. Par exemple, si votre fonction prend un nombre en entrée, testez les valeurs négatives, zéro et les très grands nombres. Si votre fonction prend une liste en entrée, testez les listes vides ou les listes avec un seul élément.
— Testez les erreurs : Si votre fonction peut lever des exceptions, écrivez des tests pour vérifier que les exceptions sont bien levées dans les situations appropriées.

Aussi, certaines fonctions sont plus faciles à tester que d'autres. Essayez dans votre pratique de **découper votre code en fonctions testables** et d'écrire des tests pour ces fonctions.

Conclusion

Dans cette quatrième semaine de cours, nous avons exploré des aspects importants de la programmation en Python qui vont au-delà des constructions de base du langage.

Les différents éléments que nous avons abordés aujourd'hui sont essentiels pour écrire du code de qualité. Ils contribuent à rendre le code plus compréhensible et facilitent la collaboration entre plusieurs développeurs. Je vous encourage vivement à appliquer ces bonnes pratiques dans vos projets de programmation. Comme pour toute compétence en programmation, la maîtrise de ces aspects vient avec la pratique. N'hésitez pas à expérimenter, à écrire du code, à le documenter et à le tester régulièrement.