Chapitre 1
Python, outils et mises au point

Avertissement :

Ce cours d'informatique pour tous reprend (c'est sûr) et complète (peut-être) le cours de première année de MPSI. Il ne doit pas être considéré comme complet et exhaustif.

La qualité de votre apprentissage dépendra fortement de votre investissement personnel au travers des cours et des TD proposés, mais également de votre travail de recherche en classe, le plus souvent sous forme de petits problèmes à résoudre. Bon travail à tous !

1. Modularité

La modularité est une notion fondamentale en informatique qui permet le réemploi de portions de code déjà développées. Il est donc possible d'utiliser en Python des modules préexistants (beaucoup sont installés par défaut), et d'en écrire de nouveaux.

Un module Python :

- est donc un fichier contenant du code python ;
- permet de factoriser du code entre différentes applications ;
- permet de ne charger en mémoire que ce dont on a besoin.

L'utilisation d'un module nécessite le mot clé `import` de la manière suivante `import <nom_Module>`

2. Typage et transtypage (cast)

Le typage est dynamique en Python (VS statique en JAVA) :

- le type de la variable n'est pas précisé lors de sa création, le typage des variables s'effectue « à la volée ».

Pour lever le doute sur le type supposé d'une variable, on peut utiliser la fonction `type(maVariable)`

- le *transtypage* (*cast* en anglais) permet la conversion d'une valeur d'un certain type vers un autre type.

On distingue le transtypage de type et de format.

3. Transtypage de type

Transtyper, c'est changer le type d'une variable : très pratique, surtout quand l'on doit afficher un nombre dans une chaîne de caractères ou encore effectuer un travail sur un type donné de variables.

On détermine le type d'une variable avec la méthode `type(maVariable)` :

```
>>> a
2
>>> str(a)
'2'
>>> float(a)
2.0
>>> int(a)
2
```

```
>>> type(a)
<class 'int'>
>>> type(a)==int
True
>>> b=str(a)
>>> type(b)
<class 'str'>
```

4. Transtypage de format

Le transtypage de format permet de convertir un nombre binaire en hexadécimal. Par exemple, il suffit de taper le format désiré puis la variable à transtyper entre parenthèses :

```
>>> a
2
>>> bin(a)
'0b10'
```

```
>>> hex(a)
'0x2'
```

5. Types mutables et non mutables

- Une variable de type mutable peut être modifiée après sa création : liste, tableau (array), ce qui peut être très intéressant.
- Les variables non mutables ne peuvent plus être modifiées après leur création : int, float, string, boolean, tuple...

Que penser de la variable a qui suit ?

```
>>> a=1
>>> a=2
```

6. La PEP 8 (Python Extension Proposal) en bref

La PEP 8 a pour objectif de définir des règles de développement communes entre développeurs. En effet, le langage Python étant assez flexible, il semble coexister différentes manières d'écrire une même instruction.

Par exemple, il est possible d'utiliser 2 espaces ou 4 pour l'indentation. Vous pouvez également ne pas mettre d'espace entre les opérateurs. Le code fonctionnera toujours ! Mais ce n'est pas parce que vous **pouvez** que vous **devez** le faire !

Que se passera-t-il le jour où une autre personne reprendra votre projet ?

Guido van Rossum insiste d'ailleurs sur le fait qu'un développeur passe plus de temps à lire du code qu'à l'écrire. C'est pourquoi votre code doit avant tout être facile à comprendre. Autrement dit, **lisible**.

Voici quelques règles communes que nous utiliserons cette année :

Mise en page

- L'indentation doit être de 4 espaces.
- Séparer chaque fonction par une ligne vide.
- Les noms (variable, fonction, classe, ...) ne doivent pas contenir d'accent. Que des lettres ou des chiffres !

Espaces dans les instructions

Les espaces suivent la syntaxe anglo-saxonne et non française. De manière plus générale, elle s'axe sur la lisibilité tout en supprimant les espaces superflus.

- Pas d'espace avant : mais un après. Exemple : `{oeufs: 2}`
- Opérateurs : un espace avant et un après. Exemple : `i = i + 1`
- Aucun espace avant et après un signe = lorsque vous assignez la valeur par défaut du paramètre d'une fonction. Exemple : `def elephant(trompe=True, pattes=4)`
- Une instruction par ligne. `{oeufs: 2}`

Commentaires

Néanmoins, si on doit faire le choix entre commenter et bien nommer, le nommage doit avoir priorité. Le commentaire est important, mais c'est le dernier recours d'un code qui n'est pas explicite. Avoir un code clair doit être l'objectif premier. Toujours préférer l'explicite à l'implicite.

- Le commentaire doit être cohérent avec le code.
- Il doit suivre la même indentation que le code qu'il commente.
- Évitez d'enfoncer des portes ouvertes : ne décrivez pas le code, expliquez plutôt à quoi il sert.
- Il doit être **en anglais**.

Conventions de nommage

- **fonctions** : minuscules et tiret du bas : `my_function()`

- **variables** : identique aux fonctions.
- **constantes** : tout en majuscules avec des tirets si nécessaire. `I_WILL_NEVER_CHANGE`
- **privé** : précédé de deux tirets du bas : `__i_am_private`
- **protégé** : précédé d'un tiret du bas : `_i_am_protected`

On essaye ? On essaye.

7. Portée d'une variable

On distingue deux sortes de variables : les variables **globales** et les variables **locales**. La portée d'une variable est la zone du code où on peut l'utiliser, autrement dit, sa zone de visibilité.

Quelques points remarquables :

- La portée d'une variable s'étend de la déclaration de la variable à la fin du bloc de déclaration.
- Dans un même bloc, il ne peut pas y avoir deux variables de même nom.
- L'usage d'une variable fait référence à la déclaration la plus proche vers le haut du programme.
- Il convient d'utiliser des variables les plus locales possibles et de limiter l'usage de variables globales dans un programme.
- Les variables locales définies avant l'appel d'une fonction seront accessibles, depuis le corps de la fonction, en lecture seule.
- Une variable locale définie dans une fonction sera supprimée après l'exécution de cette fonction.
- Les variables globales se définissent à l'aide du mot-clé *global* suivi du nom de la variable préalablement créée. Mais c'est une méthode dangereuse et déconseillée car la variable pourrait donc être modifiée par n'importe quelle fonction du programme.

 Observons cet exemple :

```
a = 10
def ma_Fonction ():
    a = 20
    print(a)
    return
```

Que vont afficher les instructions suivantes, écrites dans cet ordre ?

```
>>>print(a)
...
>>> ma_Fonction()
...
>>> print(a)
```

8. Spécifications des fonctions

Pourquoi écrire une spécification ?

Une spécification est vue comme un contrat entre l'implémenteur d'une méthode et son utilisateur. L'implémenteur s'engage à réaliser la méthode conformément au contrat et l'utilisateur s'engage à faire appel à la méthode sans supposer quoi que ce soit en dehors du contrat explicitement spécifié. Cet aspect constitue une des conditions nécessaires pour la réussite d'un développement modulaire (voir section **1.Modularité**) d'un logiciel de taille importante.

L'ajout de commentaires avec le symbole # apparaît donc comme nécessaire dans le code proposé au concours, afin de clarifier les intentions du codeur mais aussi de bien spécifier la nature des entrées et des sorties.