TP Informatique n° 1
Prise en main de l'environnement de travail

I. L'environnement de travail

I.1. L'environnement de développement

L'environnement de développement est consitué de plusieurs fenêtres.

1) **Un éditeur de texte** destiné à la saisie des programmes qui offre par défaut un certain nombre de fonctionnalités qui améliore l'expérience utilisateur.

- Coloration syntaxique.
  Les mots clé du langage sont colorés ce qui améliore la lisibilité des programmes.
- Indentation automatique.
  Automatiquement, un espace est créé lors de l'écriture du corps d'un bloc.
- Complétion automatique.
  Lors de la saisie, différents termes permettant de compléter le mot sont proposés à l'utilisateur.

2) **Un interpréteur interactif** (appellé console) qui permet d'exécuter des instructions en ligne de commande.

Il suffit pour cela de taper du code directement à la suite de l'invite `>>>`.
L'instruction tapée est évaluée et le résultat est alors affiché dans la console.

3) **Un explorateur de variables**, qui permet de connaître les valeurs contenues dans les variables en cours d'utilisation.

4) **Un explorateur de fichiers** qui permet de se déplacer dans l'arborescence de fichier et d'en charger dans l'éditeur de texte.

I.2. L'espace de travail

Afin de pouvoir récupérer le travail effectué d'une séance à l'autre, on commence par créer un répertoire `Info_2a`. Dès cette séance, il faut créer dans ce répertoire un dossier `TP_1`. À chaque séance, on créera un nouveau dossier `TP_...` afin de sauvegarder tous les fichiers créés.

II. La philosophie du langage : le zen du Python

Le zen du **Python** est une série d'aphorismes donnnant les grands principes du **Python**.
On y accède en tapant `import this` dans la console.

- Accéder au zen du **Python**.
- Quelle idée sur l'explicite est promue dans ce listing ?

```
Explicit is better than implicit.
```

III. La notion de module

III.1. Introduction

- Les fichiers contenant les programmes sont enregistrés avec l'extension `.py`.
  Ils sont appelés **module**.
- Un module doit être compris comme une boîte à outil regroupant des fonctions.
- En regroupant différents modules, on forme ce que l'on appelle un **package**.
- Le terme **library** que l'on traduit par **bibliothèque** (même si on trouve parfois l'anglicisme « librairie ») est un terme générique utilisé pour désigner tout regroupement de code conçu dans le but de pouvoir être utilisé par d'autres utilisateurs. Ainsi, tout module ou package publié sera communément appelé bibliothèque.
- **Python** est fourni de base avec une bibliothèque standard. Elle regroupe des dizaines de modules et permet de faire un grand nombre de tâches (allant de petits calculs mathématiques jusqu'à des communications réseaux ou des accès à des bases de données).

III.2. Les bibliothèques classiques

- Il est à noter qu'il existe une grande communauté de développeurs autour du langage **Python**. On peut donc facilement trouver, en libre accès sur internet, des bibliothèques publiées mises à disposition par des développeurs. Il est donc probable que ce que l'on cherche à faire en **Python** a déjà été produit par d'autres. L'usage répandu dans cette communauté est de récupérer ces bibliothèques mises à disposition, de construire dessus (ce qui permet de gagner du temps dans le développement de votre projet) et d'éventuellement contribuer en publiant des ajouts ou des bibliothèques entières.
- Évidemment, le cadre du concours est un peu particulier. On s'attend à ce que vous ayez conçu tout votre code. On se limitera donc à charger quelques bibliothèques classiques. Plus précisément, on utilisera cette année les bibliothèques suivantes.
  - × `numpy` : essentiellement, ce module fournit des fonctions permettant la manipulation efficace de tableaux. Le comportement de ces tableaux est défini par la classe `ndarray` (que l'on connaît aussi sous l'alias `array`).
  - × `math` : ce module permet d'avoir accès aux fonctions mathématiques usuelles telles que cos, sin, tan, $\sqrt{}$, exp, ...
  - × `random` : module qui contient la fonction `random` qui implémente un générateur pseudo-aléatoire. On utilisera ce module dans le cadre de simultation de v.a.r.
  - × `matplotlib.pyplot` : module qui regroupe des fonctions permettant d'effectuer des graphiques en tout genre.
  - × `scipy` : package qui contient des outils scientifiques et numériques pour **Python**. Parmi ces outils, on trouve notamment un solveur d'équations différentielles, des fonctions pour de l'intégration numérique ou encore des outils de programmation parallèle. On utilisera essentiellement le module `stats` qui fournit les fonctions pour manipuler et simuler des v.a.r. suivant une loi normale.

III.3. Espace de nom et `import`

Les espaces de nom font partie de la philosophie de **Python**.
Pour s'en convaincre, intéressons-nous à ce qu'en dit le zen du **Python**.

► Quel message le zen du **Python** fait-il passer sur les espaces de nom ?

```
Namespaces are one honking great idea - let's do more of those!
```

- Afin de pouvoir profiter des fonctionnalités offertes par une bibliothèque, il faudra la charger dans votre environnement de travail. Pour ce faire, on utilise le mot clé `import`.
  La syntaxe de base est la suivante :

```
import module
```

  ou, si le module se retrouve au sein d'un package :

```
import package.module
```

- Un module définit ce qu'on appelle un **espace de nom**. Une fois un module chargé, les variables et fonctions définies à l'intérieur d'un module sont alors accessibles via la syntaxe suivante.

```
module.variable
```

- Si la bibliothèque que l'on charge est un package, la syntaxe pour accéder à une variable au sein d'un module est donc la suivante.

```
package.module.variable
```

  Cette syntaxe étant un peu lourde, on utilise la notion d'**alias** qui permet, lors du chargement d'un module, de le renommer.

```
import package.module as mod
```

- L'intérêt d'un espace de nom est qu'il permet une isolation parfaite d'une variable au sein d'un module. Imaginons que l'on souhaite utiliser une variable `var` définie dans un module nommé `module1` et que cette variable soit aussi définie (autrement !) dans un autre module `module2` que l'on a chargé. Alors l'accès à cette variable se fera sans aucune ambiguïté : l'appel `module1.var` permettra d'accéder à la variable `var` définie au sein du premier module tandis que l'appel `module2.var` permet d'accéder à la variable au sein du deuxième module.

III.4. Le danger du `import *`

Les fonctions trigonométriques de la bibliothèque `math` prennent en paramètre des entrées que l'on doit exprimer en radian.

▶ Écrire une fonction `cos` prenant un paramètre `x` exprimé en degré et renvoyant la valeur du cosinus de `x`. Pour ce faire, on utilisera :

- × la fonction `cos` du module `math`,
- × la variable `pi` du module `math`.

Ce module sera chargé à l'aide de l'instruction : `import math`.

```
import math

def cos(x):
    return(math.cos(math.pi/180*x)

```

▶ Enregistrer ce module sous le nom `trigo.py`.

▶ On se propose maintenant de créer un nouveau module qui utilisera le module trigo.

- Pour ce faire, il faut commencer par ajouter le dossier `TP_1` comme endroit possible de stockage de modules. Ceci se fait à l'aide de la bibliothèque `sys`. Dans la console :
  - × exécuter l'instruction `import sys`,
  - × exécuter ensuite l'instruction `sys.path.append('mon_chemin')`
    où `mon_chemin` est le chemin d'accès du dossier `TP_1`.
    Par exemple : `sys.path.append('Documents\\TP\\TP_1')`
    *(on veillera à doubler les symboles \ qui introduisent des caractères particuliers en **Python**)*
- Dans l'éditeur de texte, ouvrir un nouvel onglet et recopier le programme suivant.

```
1 import trigo
2
3 print(trigo.cos(90))
```

Exécuter. Le résultat obtenu vous paraît-il cohérent ?

On obtient (approximativement !) 0 ce qui cohérent avec la définition de cosinus.

- On souhaite maintenant obtenir le résultat du calcul de $e^{\cos(90)}$.
Afin d'avoir accès à la fonction exp, on se propose de charger le module `math`.
Prévoir le résultat de l'exécution du programme suivant.

```
1 from trigo import *
2 from math import *
3
4 print(exp(trigo.cos(90)))
5 print(exp(math.cos(90)))
6 print(exp(cos(90))
```

- Le premier calcul va fournir approximativement 1.
  C'est la fonction cos avec angle en degré qui est utilisée.
  Le cosinus d'un angle de 90 degré vaut 0.
- Le deuxième calcul va rendre une valeur proche de $e^{-\frac{1}{2}}$.
  C'est la fonction cos avec angle en radian qui est utilisée.
  Or, 90 est congru à environ $\frac{2\pi}{3}$ modulo 2. Et $\cos(\frac{2\pi}{3}) = -\frac{1}{2}$.
- Pour le troisième affichage, il est plus difficile de se positionner. En effet, la fonction cos n'est plus appelée au sein d'un espace de nom. Elle peut donc correspondre à l'une ou l'autre des fonctions cos qui ont été précédemment chargées.

- Procéder à l'exécution et comparer à vos attentes.
Le résultat obtenu est-il le même si l'on change l'ordre des instructions `import` ?

- La troisième instruction renvoie le même résultat que la deuxième.
- On peut donc conclure que le dernier `import` a écrasé la définition de la première fonction cos chargée (celle qui prend un paramètre exprimé en degré) pour ne retenir que la définition de la deuxième fonction cos chargée (celle qui prend un paramètre exprimé en radian).
- En changeant l'odre des instructions `import`, la troisième instruction fournit alors le même résultat que la première puisque la dernière fonction cos chargée est celle dont le paramètre s'exprime en degré.

Conclusion
L'utilisation de l'instruction `import *` au sein d'un module est fortement déconseillée.
Plusieurs raisons à cela :

× cela peut provoquer des conflits de nom de fonctions ou de variables. En effet, cela ajoute certains objets dont on ne voulait pas a priori.

× cela peut s'avérer coûteux en temps. En effet, le nombre d'objets peut être très important.

× cela ne permet plus de documenter précisément l'origine des fonctions utilisées.

IV. Les variables

On s'intéresse dans ce qui suite à la notion de variable ainsi qu'à la gestion mémoire des variables.
On demande notamment par la suite de représenter des schémas variables / objets.
On pourra se rendre sur la page : http://pythontutor.com/visualize.html pour obtenir la construction de tels schéma lors de l'exécution d'un programme.

IV.1. La notion d'affectation

La notion de variable est essentielle en informatique. Elle s'articule autour de 3 principes :

× **stockage** d'une information,

× **accès** au contenu du stockage,

× **modification** de l'information stockée.

Dans la pratique, une variable est repérée par son **nom** auquel on associe une valeur **val**.
Ce mécanisme d'association **nom** = **val** est appelé **affectation**.

IV.2. Les références partagées

On s'intéresse tout d'abord au programme suivant.

```
1 a = [1, 2]
2 b = a
3 a[0] = 5
4 print(a)
5 print(b)
```

▶ Quelle est la valeur de `a` affcihée ? Quelle est la valeur de `b` affichée ? Expliquer.
On fera un schéma variables / objets.

On s'intéresse maintenant au programme suivant.

```
1 a = [1, 2]
2 b = a[:]
3 a[0] = 5
4 print(a)
5 print(b)
```

▶ Quelle est la valeur de `a` affcihée ? Quelle est la valeur de `b` affichée ? Expliquer. On fera un schéma variables / objets.

On s'intéresse enfin au programme suivant.

```
1 a = [1, [2]]
2 b = a[:]
3 a[1][0] = 5
4 print(a)
5 print(b)
```

▶ Quelle est la valeur de `a` affcihée ? Quelle est la valeur de `b` affichée ? Expliquer. On fera un schéma variables / objets.

▶ Comment peut-on empêcher que la modification de `a` modifie la variable `b` ?

Il faut utiliser la notion de **deep copy**. Pour ce faire, on charge le module `copy` et on écrit `b = copy.deepcopy(a)`.

- Faire le schéma variables / objets correspondant.

IV.3. La portée d'une variable dans les fonctions

La portée d'une variable est la zone de code dans laquelle la variable vit (on peut récupérer sa valeur, la mettre à jour). Illustrons cette notion, plus subtile qu'il n'y paraît, sur des exemples.

- On considère tout d'abord le programme suivant.

```
1  a = 1
2  b = 1
3  for i in range (10):
4      print(a)
5
```

Quel est le résultat produit par ce programme ? Commenter.

Les variables `a` et `b` sont des variables définies dans le bloc de code principal du programme. Elles sont dites **globales**. Elles sont accessibles partout dans le module.

- On ajoute alors les instructions suivantes.

```
6  def f():
7      b = 2
8      c = 3
9      print(a, b, c)
10
11 print(b)
```

Quel est la valeur de `b` affichée par ce programme ? Commenter.

La valeur affichée est 1. En effet, la définition `b = 2` est une définition locale à la fonction `f`. Cette définition n'est accessible qu'au sein du bloc de code de la fonction. Lors de l'appel à la fonction `f`, une case mémoire est créée pour stocker la valeur de `b`. Cette case est supprimée à la fin de l'exécution de `f`.

- On ajoute alors l'instruction suivante.

```
12  print(c)
```

Que produit l'exécution de ce programme ? Commenter.

> Un message d'erreur apparaît : `NameError: name 'c' is not defined`. L'explication est la même que la précédente. La variable `c` est une variable locale à la fonction `f`.

- Que se passe-t-il si on intercale l'instruction `a = a+1` entre la ligne 6 et la ligne 7 ? Commenter.

> Un message d'erreur apparaît : `UnboundLocalError: local variable 'a' referenced before assignment`. La variable `a`, locale à la fonction `f` nepeut être mise à jour car elle n'a pas été définie dans le bloc de `f`.

- On considère maintenant le programme suivant.

```
1   a = 1
2   b = 1
3   c = 1
4
5   def g():
6       b = 3
7       c = 4
8       def h():
9           c = 5
10          print(a, b, c)
11      h()
12
13  g()
```

Quel est le résultat de l'exécution de ce programme ? Commenter.

> Pour la variable `a`, le mécanisme est le suivant.
> - La fonction `g` réalise un appel à `h`, fonction qui réalise l'affichage de `a, b, c`.
> - On cherche alors la définition de `a` dans le bloc local *i.e.* dans le bloc défini par la fonction `h`.
> - La variable `a` n'est pas définie dans ce bloc. On cherche donc la définition de `a` dans le bloc de la fonction `g` qui englobe le bloc de `h`.
> - La variable `a` n'est pas définie dans ce bloc. Il n'y a pas de bloc de fonction englobant. On cherche alors `a` dans le bloc principal.
>   La variable `a` y est définie et vaut 1.
>
> Pour la variable `b`, le mécanisme est le suivant.
> - La variable `b` n'est pas définie localement dans `h`.
> - Par contre, elle est définie dans le bloc englobant de la fonction `g`. Elle est égale à 3.
>
> Pour la variable `c`, le mécanisme est le suivant.
> - La variable `c` est définie localement dans `h` et vaut 5.
>
> Ce programme produit donc l'affichage : `1,3,5`