Prise en main de l'environnement de travail

I. Quelques mots sur l'outil

I.1. Une brève introduction

Python est un langage de programmation dont la naissance remonte aux années 1990, développé par Guido van Rossum. **Python** est placé sous une licence libre (et donc **gratuite**) qui fournit un puissant environnement de développement pour les applications scientifiques et l'ingénierie. **Python** est distribué sous la licence open source Python Software Foundation License, et est téléchargeable gratuitement. Il est disponible sous GNU/Linux, Mac OS X et Windows.

I.2. Installation de Python sur votre machine

Pour l'installation de Python :

1) on téléchargera directement **Anaconda** qui permet d'installer **Python** et toutes les bibliothèques nécessaires à son utilisation en classes préparatoires.
Normalement, la page web citée précédemment détecte automatiquement votre système d'exploitation (Linux, Mac OS ou Windows et sa version). Néanmoins, vous vérifierez bien que la version proposée à l'installation convient à votre machine.

2) on suivra les instruction de téléchargement.

II. Environnement de travail

II.1. Les fenêtres initiales

Lors du premier lancement d'**Anaconda**, la fenêtre suivante apparaît.

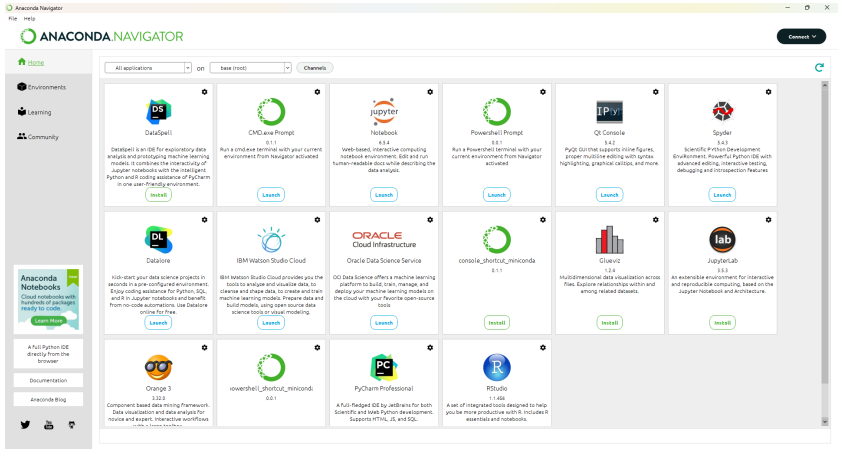

Fig. 1 Accueil Anaconda

Dans le cadre des cours et TP, nous utiliserons essentiellement **Spyder** et **Jupyter Notebook**. Ces deux applications sont accessibles depuis l'accueil **Anaconda** mais le sont aussi directement depuis votre menu d'Applications pour les systèmes d'exploitation Mac OS et Linux, ou votre menu Windows.

Intéressons nous plus spécifiquement à **Spyder**. Lors du premier lancement de **Spyder**, un groupement de 3 fenêtres apparaît. L'environnement de travail se compose du navigateur de fichiers, de la console, du navigateur de variables et de l'historique des commandes.

- L'*éditeur de texte*, à gauche, permet d'écrire et sauvegarder facilement son code.

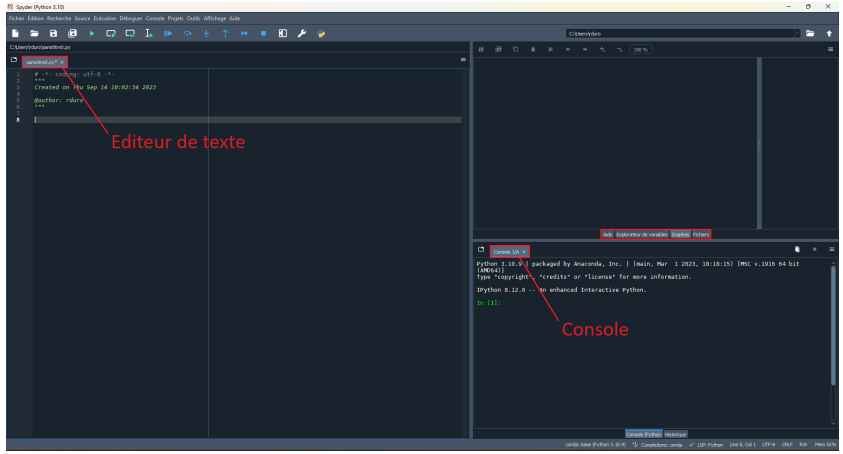

Fig. 2 Environnement de travail sous Spyder

- La *console*, en bas à droite, permet l'interprétation directe des commandes que l'on tape, ce qui correspond à une utilisation de type « calculatrice ». Pour ce faire, il suffit de saisir une commande à la suite du symbole `In [1]:` et de taper sur `ENTRÉE` pour que cette commande soit interprétée.
- La fenêtre en haut à droite est multi-tâche. Elle contient :
  - × un onglet d'*aide*

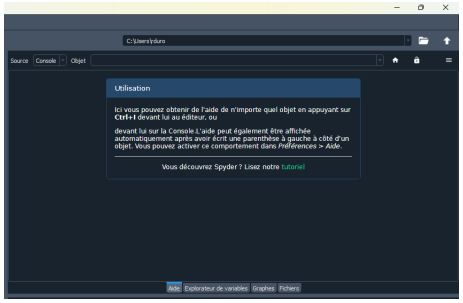

Fig. 3 Aide sous Spyder

  - × le *navigateur de variables* qui permet de visionner les valeur des variables.

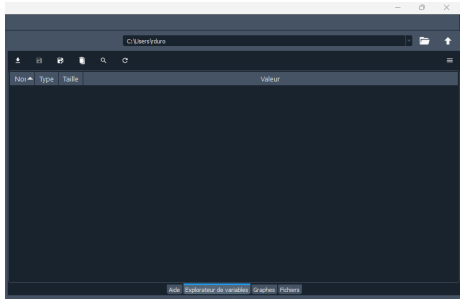

Fig. 4 Navigateur de variables

  - × un onglet qui permet d'afficher les *graphes* générés par son code **Python**.
  - × le *navigateur de fichiers* qui permet de se déplacer dans l'arborescence des fichiers de l'ordinateur.

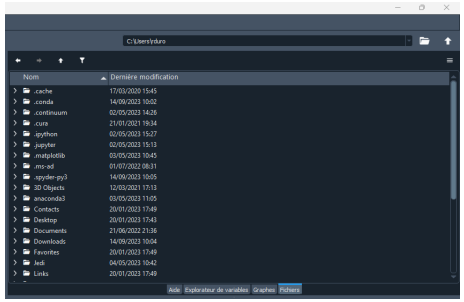

Fig. 5 Navigateur de fichiers

II.2. Un mot sur l'éditeur

L'éditeur de texte **Spyder** dispose de quelques fonctionnalités intéressantes :

- **coloration syntaxique** : les mots clés du langage apparaissent en couleur.
- **indentation** : on rappelle que le langage **Python** utilise l'indentation pour structurer la programmation. Celle-ci s'effectue automatiquement au début d'un bloc d'instructions (délimité par un double-point :).
- **autocomplétion** : lors de l'écriture d'un mot clé, l'éditeur peut automatiquement intégrer de nouveaux éléments (tels que des couples de parenthèses ou des mots clés associés au premier) dans le but de faire gagner du temps au programmeur. De plus, si l'on tape les premières lettres du nom d'une fonction prédéfinie, **Python** peut proposer une liste de noms de fonctions commençant par ces lettres. Pour ce faire, il suffit de taper les premières lettres d'une fonction et d'appuyer sur la touche `TAB`. Par exemple, si l'on tape `np.sq` suivi de `TAB`(après importation de la bibliothèque `numpy`, une boîte d'affichage s'ouvre, proposant notamment les noms de fonctions `sqrt`, `square` ou `squeeze`. On peut alors utiliser l'aide pour savoir ce qui est calculé par ces fonctions.
  La console intègre également un mécanisme d'autocomplétion.

II.3. Quelques raccourcis intéressants pour gagner du temps

Connaître certains raccourcis claviers permet de gagner en vitesse en évitant notamment l'utilisation un peu lourde de la souris. Voici quelques raccourcis qui pourront servir en **Python** (voire pour certains autres logiciels).

	Windows/Linux	Mac
Sélectionner tout	`CTRL-A`	`CMD-A`
Sauvegarder	`CTRL-S`	`CMD-S`
Ouvrir un nouvel onglet	`CTRL-N`	`CMD-N`
Exécuter	`F5`	`fn-F5`

III. La philosophie du langage : le zen du Python

Le zen du **Python** est une série d'aphorismes donnant 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.
```

IV. La notion de module

IV.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).

IV.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 pour votre TIPE par exemple. 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.

IV.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 :

```
1 import module
```

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

```
1 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.

```
1 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.

```
1 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.

```
1 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.

IV.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`.

```
1  import math
2
3  def cos(x) :
4      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 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.