Initiation Python/Jupyter TP1

Germain POULLOT

6 Octobre 2021

Table des matières

1 Objectifs

L'objectif de ces séances est d'obtenir les connaissances minimales et la maîtrise technique suffisante pour être à l'aise avec le langage Python (accompagné de la bibliothèque Sage) et l'environnement Jupyter afin de pouvoir sereinement aborder les cours d'option qui préparent à l'épreuve de modélisation numérique.

Il n'y a pas de prérequis (vraiment aucun) ! Nous adapterons la vitesse des TP selon les connaissances que vous possédez déjà et les buts que vous souhaitez atteindre. Il n'y a pas non plus d'évaluation.

Afin de vous permettre de mieux comprendre ce que vous faites, je vous proposerai quelques *points de cours*, qu'il n'est pas nécessaire de retenir mais qui permettent d'avoir du recul sur votre programmation.

Les objectifs de ce premier TP sont de :

0. Allumer l'ordinateur, se connecter avec son compte UPMC, etc.
1. Lancer Jupyter.
2. Écrire votre premier calcul et le lancer.
3. Ranger une valeur dans une variable.
4. Utiliser les structures `if`, `then`, `else` ; `for` ; `while`.
5. Écrire votre première fonction.
6. Savoir où se renseigner pour progresser.

Les termes en *italique* seront expliqués durant le TP. Les termes importants sont en *bleu*. Les passages en rouge sont des liens externes cliquables.

2 Qu'est-ce que Python, Jupyter et SageMath ?

2.1 Python

> Python est un *langage* de programmation *interprété*, multi-paradigme et multiplateformes. Il favorise la *programmation impérative structurée*, *fonctionnelle* et *orientée objet*. Il est doté d'un typage dynamique fort, d'une gestion automatique de la mémoire par ramasse-miettes et d'un système de gestion d'exceptions. [Il] est placé sous licence libre. (Wikipedia)

Python est ainsi la "langue" dans laquelle nous allons parler à l'ordinateur pour lui dicter les calculs à faire. L'usage de ce langage nécessite une syntaxe précise, l'emploi de certain mots clefs, d'une certaine disposition du texte, etc.

2.2 SageMath (anciennement appelé Sage)

> SageMath est un logiciel libre de mathématiques sous licence GPL. Il combine la puissance de nombreux programmes libres dans une interface commune basée sur le langage de programmation Python.

> SageMath permet de faire des mathématiques générales et avancées, pures et appliquées. Il couvre une vaste gamme de mathématiques, dont l'algèbre, l'analyse, la théorie des nombres, la cryptographie, l'analyse numérique, l'algèbre commutative, la théorie des groupes, la combinatoire, la théorie des graphes, l'algèbre linéaire formelle, etc... (SageMath - Français) [1]

SageMath, pour ce dont on va se servir, offre une *bibliothèque* Python (*librairy* en anglais), c'est-à-dire un ensemble de programmes déjà codés, prêt à l'emploi. Par exemple, pour calculer les valeurs propres d'une matrice, je n'ai pas besoin de ré-expliquer à l'ordinateur ce qu'est une matrice, ni de lui dire comment calculer les valeurs propres : j'entre *`ma_matrice`* dans le format de SageMath, je tape `eigenvalues(`*`ma_matrice`*`)` et j'ai immédiatement la solution.

Des exemples ici : `https://doc.sagemath.org/html/fr/a_tour_of_sage/`.

2.3 Jupyter

> The Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text. Uses include : data cleaning and transformation, numerical simulation, statistical modeling, data visualization, machine learning, and much more. (Jupyter website)

Jupyter permet de présenter correctement un code, par exemple en Python, de l'agrémenter de commentaires (en Markdown, ce que nous n'apprendrons pas ici), et de structure son code intelligemment et ergonomiquement. Il n'est pas nécessaire à l'utilisation de Python ou de SageMath, mais les facilite.

3 Bien commencer

3.1 Premières opérations

Les exercices **doivent** être faits dans l'ordre.

Exercise 3.1.1. Allumez l'ordinateur et lancer Jupyter : allez sur la plateforme `https://jupyter.math.upmc.fr` avec vos identifiants UPMC.

Créez un nouveau fichier en SageMath (`Nouveau` → `SageMath 9.1`).

Placez-vous dans une cellule et tapez :

```
5 + 3
```

Appuyez sur le bouton `Exécuter` (ou Crtl + Entrée pour aller plus vite).

Bonne nouvelle : Python sait faire une addition !

Exercise 3.1.2. Insérez une nouvelle cellule (bouton `+`).

Testez les opérations numériques classiques :

1. Et la théorie des polytopes aussi !

```
5 - 3
5 * 3
5 / 3
5 // 3
5 % 3
5^3
```

Python n'est pas sensible aux (simples) espaces : `5 + 3` ; `5+ 3` ; `5 +3` et `5+3` sont la même chose. À vous de voir comment vous souhaitez écrire, je préfère comme ci-dessus, les espaces permettent à l'oeil humain de mieux lire.

Par contre, si vous avez exécuté le code ci-dessus dans une cellule, il vous aura répondu `125` : toutes les lignes de votre code sont bien exécutées, mais seul le résultat final (celui de la dernière instruction) apparaît.

En fait, vous voulez demander à Python non seulement de calculer ces opérations mais aussi de vous **écrire** le résultat :

```
print(5 - 3)
print(5 * 3)
print(5 / 3)
print(5 // 3)
print(5 % 3)
print(5^3)
```

Ici, Python ne vous donne pas un résultat `Out[3]:...`, mais écrit sur votre page ce que vous lui demandez.

Q : Faites des tests pour comprendre à quoi correspondent les opérations que vous ne connaissez pas (notamment `//` et `%`).

Attention : Avec la bibliothèque SageMath, l'exponentiation s'écrit `5^3`. Avec Python seul, elle s'écrit `5**3`. Le symbole `**` a une autre utilité dans SageMath (que nous ne verrons pas).

Exercise 3.1.3. Dans l'exercice précédent, `5/3` donnait comme réponse `5/3` : SageMath considère que `5` et `3` sont des entiers, il cherche dont l'ensemble le plus adéquat pour vous donner une réponse. Or un ratio d'entiers est toujours un rationnel : il vous répond dans $\mathbb{Q}$.

Mais peut-être vouliez-vous voir $5/3$ comme un réel ? Tapez :

```
print(5/3)
print(5.0/3)
print(5./3)
print(RR(5/3))
print(RR(5)/3)
print(QQ(5.0)/3)
```

Écrire `5.0` permet de forcer le nombre 5 à être vu comme un réel. `5.` est un racourci pour `5.0`. `RR` représente $\mathbb{R}$: vous demandez à SageMath de convertir `5/3` en un réel. Essayez de comprendre les deux dernières opérations.

Beaucoup d'autres structures (mathématiques) sont déjà implémentées dans SageMath. **Attention :** Toutes n'existent pas nativement dans Python.

Exercise 3.1.4 (Facultatif)**.** Pour renter un peu plus dans le détail, demandons à SageMath comment il "voit" les données :

```
print(type(5))
print(type(5.0))
print(type(RR(5)))
print(type(5/3))
print(type(int(5)))
print(type(True))
print(type("a"))
print(type(print))
```

On a deux sortes de réponses : soit le `type` est en `sage....` (c'est un type qui existe grâce à SageMath), soit il est *natif*. N'allons pas plus loin dans les détails (et ne tapons pas `type(type)`...).

Vous aurez rarement à gérer les types, mais il est bon de savoir ce que c'est.

Exercise 3.1.5. Jouons avec un autre type : les booléens. Un booléen peut être `True` ou `False`. Certains opérateurs permettent de poser une question (logique du premier ordre). Le test d'égalité s'écrit `==`.

Mettre `print()` permet de sauter des lignes (une présentation lisible des résultat est importante) :

```
print(1 == 2)
print(5 + 3 == 8)
print()
print(5 * 3 > 10)
print(5 * 3 > 15)
print(5 * 3 >= 15)
print()
print("1" == 1)
print(5 == 5.0)
print()
print(3 == 2 + 1 and 7 == 8 - 1)
print(1 == 2 or 7 == 8 - 1)
print(not 1 == 2)
print()
print(3 in NN)
print(1.5 in ZZ)
print(sqrt(2) in QQ)
```

Q : Prenez le temps de lire les résultats et de noter ce qui vous surprend.

Vous découvrez peu à peu des mots en gras en couleur (vert dans Jupyter, bleu ici) : ce sont des *mots clefs*, des mots qui ont un sens spécifique en Python et ne peuvent pas être utilisés à autre chose qu'à leur dessein préconçu. Ils n'ont pas besoin de parenthèses pour fonctionner. En voici la liste (vous apprendrez

à utiliser ceux en bleu a priori) :

```
False, await, else, import, pass, None, break, except, in, raise,
True, class, finally, is, return, and, continue, for, lambda, try,
as, def, from, nonlocal, while, assert, del, global, not, with, async,
elif, if, or, yield
```

3.2 Structures conditionnelles et boucles

Exercise 3.2.1. On sait faire un test, on peut faire quelque chose selon l'issue :

```
if 1 + 1 == 2:
    print("Tout va bien.")

if 1 > 2:
    print("Huston, nous avons un probleme !")
```

L'*indentation* est cruciale ici ! Voyons un exemple plus touffu pour comprendre (il n'est pas nécessaire de comprendre ce que fait le programme) :

```
C1 = {1,2,3,4,5,0}
C2 = {2,3,4,5,6,7}
Ind1, Ind2, Shared = set(), set(), set()
for x in C1:
    if not x in C2:
        Ind1.add(x)
        if len(Ind1) >= 2:
            break
    else:
        Shared.add(x)
Ind2 = C2.difference(Shared)
print(Ind1, Ind2, Shared)
```

Regardez le premier `if`. Il est suivi d'un bloc indenté : Python va calculer le booléen qui suit `if`, c'est-à-dire `not x in C2`, s'il est vrai, on exécutera les instructions du bloc indenté, sinon on passera directement à la ligne qui a la même indentation que notre `if` : celle de `else`.

Pour le second `if`, si `len(Ind1) >= 2`, alors Python exécutera `break`. Dans le cas contraire (ou bien après avoir exécuté `break`), il sautera à la fin du bloc, jusqu'à `else`.

Essayons un petit programme :

```
x = random()
print(x)
if x > 0.5:
    print(+1)
else:
    print(-1)
```

Q : L'instruction `x = random()` permet de tirer un nombre aléatoire (uniforme) dans $[0,1[$ et de l'appeler `x`. Que fait ce programme ?

Q : Écrivez un programme qui tire deux variables aléatoires (uniformes) dans $[0,1[$ et qui retourne le maximum des deux. Idem avec trois variables. (Interdit d'utiliser la fonction `max`, bien évidemment !)

Il existe un mot clef `elif` qui est la contraction de `else` et `if` : à vous de trouver comment s'en servir.

Exercise 3.2.2. Outre les structures conditionnelles, Python permet aussi de faire des boucles. Deux formes de boucles sont possibles : les boucles `for` et les boucles `while`.

```
for x in {1,2,3,4,5}:
    print(x^2)
    print(x % 2 == 0)
print()
print(x)
```

Ici, Python "parcourt" l'ensemble `{1,2,3,4,5}`, et pour chaque valeur possible, effectue les opérations indentées. On constate qu'à la fin, `x` vaut bien 5, la dernière valeur utilisée dans la boucle.

Q : Je lance deux dés à six faces : affichez toutes les différences possibles (comment en déduire la probabilité que cette différence fasse 3 ?).

Q : Faites la liste de tous les entiers pairs. **Attention :** pour arrêter un programme, utilisez le bouton ■.

Q : Pour ne pas taper des ensembles de nombres, on utilise `range(a,b)` : c'est l'ensemble des nombres de a à $b-1$ (compris). Par convention `range(n)` est l'ensemble des nombres de 0 à $n-1$. Dans toutes vos fonctions, remplacez vos ensembles de nombres par des `range(n)` ou `range(a,b)`.

Exercise 3.2.3. Avant d'attaquer les boucles `while`, il faut parler des variables. Vous en avez déjà vu ! Le symbole `=` permet de d'affecter une valeur à une variable.

```
x = 3
print(x)
x = 5
print(x)
y = 4
print(x,y)
y = 2*x + y
print(x,y)
z = x==1
print(x,y,z)
x = 1
print(x,y,z)
```

Lorsque vous affectez une variable, si le contenu est issu d'un calcul, ce calcul est fait sur le moment : par exemple `z = x==1` met `False` dans `z`, modifier `x` par

la suite ne change pas la valeur dans `z`. **Attention :** Cette idée peut devenir fausse pour des structures de données plus compliquées...

Vous pouvez donner n'importe quel nom à vos variables, aussi long que vous voulez. Donner des noms intelligents permet de s'y retrouver : une matrice qu'on veut diagonaliser peut être appelée `matrix_to_be_diagonalized` (pas d'accents, pas d'espaces, pas de mot clef, les nombres oui).

Exercise 3.2.4. Maintenant, faisons une boucle `while` :

```
i = 0
while i <= 5:
    print(i^2)
    print(i % 2 == 0)
    i = i+1
print()
print(i)
```

Dans cette boucle, la valeur de `i` **n'est pas** mise à jour automatiquement à chaque passage, mais une condition est testée. Les boucles `while` sont donc plus souples que les boucles `for` (on peut toujours recoder une boucle `for` par une `while`), mais les boucles `for` sont plus faciles à utiliser.

Q : Codez la suite de Syracuse pour 14 : affichez tous les termes de la suite de Syracuse avec $u_0 = 14$, $u_{n+1} = 3u_n + 1$ si u_n est impair, et $u_{n+1} = {}^{u_n}/_2$ si u_n est pair. On n'affichera les termes tant qu'on n'a pas rencontré de 1.

Q : Changez votre programme pour pouvoir commencer à n'importe quel entier.

Q : N'affichez plus les termes consécutifs, mais le nombre de termes avant d'atteindre 1. Affichez ce "temps de vol" pour les 20 premiers entiers.

3.3 Créer une fonction

Q : **Exercise 3.3.1.** Écrivez un programme dont la première ligne est `n = 123`, et dont les lignes suivantes affichent les diviseurs de `n` (méthode naïve).

Q : Modifiez ce programme pour qu'il affiche le nombre de diviseurs de `n`.

On veut maintenant encapsuler ce programme pour pouvoir l'utiliser dans d'autres situations sans avoir à recopier toutes les lignes à chaque fois. Pour ce faire, on utilise le mots clefs `def` et `return` et créer une fonction :

```
def nombre_de_diviseurs(n):
    premiere ligne de mon programme
    deuxieme ligne de mon programme
    ...
    derniere ligne de mon programme
    return le_resultat
```

Grâce à cette fonction, on va pouvoir en créer d'autres !

Q : Créez une fonction qui détermine si un entier est un nombre premier (elle doit donc prendre en entrée un nombre `n` et retourner un booléen).

Q : Créez une fonction qui calcule le plus grand diviseur premier d'un nombre.

3.4 Où se documenter ?

Premièrement, je vous conseille de lire la page du concours sur l'épreuve de modélisation : https://agreg.org/index.php?id=modelisation.

Si vous voulez savoir ce que fait quelque chose dans Python, vous pouvez taper `help(...)`. Par exemple, tapez `help(range)` et essayez de trouver l'endroit où est l'explication de ce que fait `range`.

Pour SageMath, il suffit de taper "Sagemath + *votre question*" sur Google pour trouver de très bonnes informations. Les pages de référence sont typiquement en "*sujet* - Sage 9.4 Reference Manual".
Par exemple, on trouve une excellente fiche récapitulative sur les matrices : https://wiki.sagemath.org/quickref?action=AttachFile&do=get&target=quickref-linalg.pdf

Si vous avez une question sur un objet issu de Python, tapez "Python + *votre question*" sur Google. Les informations de la documentation sont souvent très technique et beaucoup trop détaillées (essayez quand même de les lire), mais les discussions des forums aident à résoudre le problème (Geeksforgeeks, W3Schools, Programiz,...).

Jupyter permet de faire des cellules de texte en plus du code. Pour ce faire, aller dans `Cellule > Type de cellule > Markdown` sur Jupyter. Le texte que vous y taperez est en Markdown, un langage au-dessus du html (donc vous pouvez taper en html ou en LaTeX si vous voulez). Pour débuter en Markdown, c'est ici : https://learnxinyminutes.com/docs/fr-fr/markdown-fr/

Il existe des centaines de cours de Python directement en ligne, gratuits ou payants, pour n'importe quel niveau et de n'importe quelle qualité. Le site du Zéro en faisait des bons, mais ce site a été refondu dans OpenClassroom qui est bien mais mal rangé. Sinon, le site suivant est plutôt (trop) exhaustif : https://courspython.com/bases-python.html
Par contre, vous ne trouverez que difficilement des cours pour SageMath : le manuel de référence est suffisamment fourni (normalement).

Il est **indispensable** de lire les messages d'erreur quand votre code échoue ! Notamment le type de l'erreur (au début, souvent `SyntaxError`), sa localisation, et sa description. C'est en faisant des erreurs qu'on apprend !