TP n°1 - Initiation à Python

En MPSI, les algorithmes que nous allons étudier seront implémentés grâce au langage **Python**. Python est un langage de haut-niveau - c'est-à-dire un langage qui permet à l'utilisateur d'écrire des programmes informatique en terme de ce qu'il veut faire (et non pas en terme de ce que la machine sait faire) et sa syntaxe est très naturelle (si, comme pour tout langage, on maîtrise un anglais basique).
Python a vu sa première version sortir en 1991 et a été créé par Guido Van Rossum.
Il s'agit d'un langage facile a appréhender qui permet de nombreuses possibilités. On étend ses fonctionnalités grâce à des bibliothèques (pour simplifier : des collections de fonctions déjà codées) qui facilitent le travail des développeurs : quand on travail sur un sujet précis, on ne veut pas avoir à coder des fonctions basiques comme par exemple le cosinus ou la partie entière !

Mais alors, à terme, que peut-on faire avec Python :

— Dans un premier temps, programmer des petits scripts qui peuvent automatiser des tâches sur notre ordinateur (renommer des fichiers en masse par exemple)

— Puis une fois très à l'aise, créer des logiciels, des jeux, etc...

Alors maintenant, comment utiliser Python ?

Nous allons utiliser l'environnement de développement **EduPython** qui inclut entre autres :

— Python 3
— numpy : c'est une bibliothèque de fonctions pour effectuer des calculs numériques
— scipy : c'est une bibliothèque de fonctions pour effectuer du calcul scientifique
— matplotlib : c'est une bibliothèque de fonctions graphiques

Vous pouvez l'installer sur votre ordinateur personnel en vous rendant à l'adresse :

`https://edupython.tuxfamily.org/`.

Dans ce premier TP, nous allons nous familiariser avec l'environnement de développement EduPython et avec la syntaxe et les particularités du langage Python.

1. L'environnement de développement

On peut écrire et exécuter du code écrit en langage Python via :

— *La console interactive ou shell :* le code à évaluer est écrit dans l'interprète de commande après **l'invite de commande** ou **prompt** i.e. `>>>` ou `In` (selon l'environnement) ; Le **résultat** (s'il y en a un) du code est affiché en dessous sans préfixe dans le premier cas ou après `Out` dans le second. **Attention :** Toute commande inscrite via la console **ne peut pas être sauvegardée** ! On l'utilisera principalement pour effectuer des calculs ou des tests de parties de nos programmes mais **jamais** pour implémenter nos algorithmes !

— *L'éditeur :* il permet d'écrire des programmes et de les enregistrer dans un fichier. On peut alors exécuter le code inscrit dans l'éditeur par l'interpréteur Python : le résultat (s'il y en a un) de l'exécution est affiché directement dans la console interactive.

2. La console interactive

Pour le moment, nous allons nous familiariser avec la fenêtre qui se trouve dans le shell : *l'interprète de commande.*

a. Les bases de l'interprète de commande

Après les informations relatives à la version de Python, l'interprète de commande commence par une invite de commande (ou prompt). Comme il a été dit plus haut, elle est symbolisée par `>>>` ou `In`. On entre nos lignes de code une par une après chaque invite de commande en les validant avec la touche "Entrée". Le résultat (s'il y en a un) s'affiche alors sous chacune des lignes validées : on remarque que la ligne le contenant ne commence pas par l'invite de commande ; en effet, il s'agit d'un résultat, et non d'une instruction. Puis encore en dessous, une nouvelle invite apparaît, afin que l'on puisse de nouveau écrire une instruction.

On commence par répondre à la première invite par `1+1` et en validant par "Entrée" ; on doit obtenir :

code python

```
>>>1+1
2
>>>
```

L'interprète de commande a exécuté notre instruction et retourné le résultat de celle-ci. Une fois la tâche terminée, le prompt nous invite de nouveau à taper une instruction.

Attention, il convient de bien différencier ce que peut produire une instruction donnée à Python. Essayons l'instruction suivante : `print('Hello world ')`!. On obtient :

code python

```
>>>print('Hello world !')
Hello world !
>>>
```

Du point de vue de l'affichage, on ne voit pas de différence avec le code précédent, pourtant il y en a une !

- l'instruction `1+1` a comme résultat `2` et n'a pas d'effet sur l'environnement
- l'instruction `print('Hello world !')` a un résultat particulier qui n'est pas affiché (`None`) et a pour effet d'écrire une chaîne de caractère dans le shell.

Oulah, mais quelle est la différence entre résultat et effet sur l'environnement ?

Même si la console n'indique aucune différence, la voici : un effet est une interaction avec un périphérique de sortie ou une modification de la mémoire (on le verra avec la déclaration de variable et un résultat est le retour d'une instruction (on s'y attardera plus longuement dans l'étude des fonctions). Pour s'en convaincre, modifions le code précédent et tapons `print('Hello world !', file=open('essai.txt', 'w'))` ; on obtient :

code python

```
>>>print('Hello world !', file=open('effet.txt', 'w'))
>>>
```

Cette fois ci, plus rien n'apparaît dans le shell après notre instruction ! Le résultat `None` n'étant jamais affiché, et on remarque qu'un fichier effet.txt contenant la phrase *Hello world !* a été créé dans notre répertoire : c'est l'effet de notre instruction sur son environnement !

b. Python=calculatrice ?

Nous allons effectuer quelques tests dans la console en proposant comme instructions divers calculs arithmétiques simples.

Essayons les instructions suivantes ligne par ligne et observons le résultat.

code python

```
7+1
5-1
2*4
2**10
8/4
10/3
3//4
10//3
25%7
```

On comprend avec ces quelques exemples à quoi correspondent ces différents opérateurs :

+	l'addition usuelle
-	la soustraction usuelle
*	la multiplication usuelle
**	l'exponentiation (puissance)
/	la division usuelle
//	le quotient division euclidienne
%	le reste de la division euclidienne

La priorité des opérations est la même qu'en mathématiques. Pour s'assurer de la priorité d'une opération par rapport à une autre, on utilise, toujours comme en mathématiques, les parenthèses.

Si on veut effectuer des calculs plus poussés, Python n'est pas armé par défaut pour permettre l'utilisation de nos fonctions usuelles comme le cosinus, le logarithme ou la racine. Pour cela, il faut faire appel à une bibliothèque (ou module) nommée `math` qui définit ces fonctions pour nous. Une fois importée grâce à l'instruction **`import`**, on peut faire usage d'une multitude de fonctions supplémentaires pour nos calculs.

On importe la bibliothèque `math` et on lui donne le surnom `m` pour simplifier nos futurs appels de fonctions de cette bibliothèque :

```
import math as m
```

Le nom des fonctions dans la bibliothèque `math` est volontairement très proche du nom de leur pendant mathématique : on trouve par exemple `exp`, `cos`, `tan`, etc... Il faut cependant faire référence à la bibliothèque avant l'appel de la fonction : il faut indiquer à l'interpréteur où aller la chercher puisqu'elle n'existe pas pour python à la base !

Voici comment calculer la racine de 8 grâce à la bibliothèque `math` (que l'on a surnommée `m`) :

```
m.sqrt(8)
```

On peut également importer un module en utilisant la syntaxe suivante `from nomdumodule import *` ; dans le cas de `math` cela donne :

```
from math import *
```

L'avantage de cet appel est que l'on a plus besoin du "surnom", on appelle directement les fonctions du module :

```
sqrt(8)
```

Ce code donnera le même résultat que le précédent calcul.

Mais alors pourquoi s'embêter avec les "surnoms", ça alourdi considérablement la syntaxe ! En effet, mais un souci peut se poser si on importe plusieurs modules : si deux modules contiennent des fonctions ayant le même nom, python ne saura pas faire la différence, et il considérera que c'est le dernier module appelé qui imposera sa fonction. En fait, la première fonction est "écrasée" par la deuxième portant le même nom.

On prendra donc l'habitude, quand on importe plusieurs modules, d'utiliser la première méthode pour ne pas avoir de mauvaises surprises...

Revenons au module `math`. Comment savoir quelle fonctions sont inclus dans ce module ? Pour tout connaître d'un module/bibliothèque, il suffit de taper `help("nomdumodule")` dans l'invite de commande. Donc pour connaître le module `math` (après l'avoir importé), on tape :

```
help("math")
```

Voici un premier exercice pour appréhender les calculs numériques avec Python.

Exercice 1.

Écrire en Python les calculs mathématiques suivants et afficher le résultat :

1. le quotient et le reste de la division euclidienne de 10000 par 123 ;
2. $\sqrt{4064256}$;
3. $\cos(\frac{2\pi}{3})$
   Remarque : le nombre π est implémenté sous le nom `pi` dans le module `math` que nous avons surnommé `m` : on retrouve donc π en Python (enfin une approximation, bien sûr) avec `m.pi`.
4. $\mathrm{ch}(0)$
5. $3 \times 2^2 + \frac{5+11^4}{20} - (64-5)^{-2} + 3.8$ (Attention la virgule pour les nombres décimaux devient en point en Python).

c. Python=grapheur ?

On peut utiliser python pour tracer des graphes de fonctions. Pour cela, nous avons besoin d'importer deux bibliothèques, il s'agit ici de `numpy` et `matplotlib` (et son module `pyplot`).
Le module `numpy` permet de gérer des listes de points et d'appliquer des fonctions à ces listes.

On importe `numpy` en le surnommant `np`

```
1 import numpy as np
```

Remarque 1.

Si un code d'erreur apparaît, c'est que `numpy` n'est pas installé : il suffit alors de faire appel à un installeur de module. Le code suivant permet d'installer un module quelconque (enfin, pas tous, mais beaucoup de modules) avec `pip` (ou `conda` selon ce qui est présent sur la machine) :

```
1 pip install nomdumodule
```

Importons le module `pyplot` de `matplotlib`

```
1 import matplotlib.pyplot as plt
```

Voyons alors un exemple de ce qu'il est possible de faire grâce à ces bibliothèques ; validons ligne par ligne le code suivant :

```
1 x=np.linspace(0,10,100)
```

→ Crée une liste de 100 nombres régulièrement espacés entre 0 et 1 et la stocke dans la variable x (on y reviendra tout à l'heure !)

```
1 y=np.sqrt(x)
```

→ Crée une liste contenant la racine de chaque nombre contenu dans la liste x et la stocke dans y.

On veut alors tracer une courbe qui relie les points du plan donnés par les couples de nombres contenus dans le couple (x, y). On doit alors utiliser le module `pyplot` et sa fonction `plot` qui fait le lien entre les abscisses (ici, les nombres contenus dans x) et les ordonnées (ici, les nombres contenus dans y) de la courbe de la fonction à tracer :

```
1 plt.plot(x,y)
```

Nous utiliserons et comprendrons ces fonctionnalités dans de prochains TP !

3. L'éditeur

L'éditeur - la fenêtre à côté de la shell - permet d'écrire plusieurs lignes de code à la fois ! Mais celle-ci ne sont pas interprétées directement : il faut exécuter le code inscrit dans l'éditeur pour qu'il soit interpréter dans le shell. Mais attention, quand on exécute un code provenant de l'éditeur, le résultat ne s'affiche pas dans la console, seuls les effets des instructions peuvent permettre d'y afficher quelque chose après exécution.
L'intérêt principal de l'éditeur n'est donc pas d'effectuer des calculs : il nous permettra d'écrire des programmes, qui sont des successions d'instructions ; et surtout, il nous permettra d'enregistrer ces programmes : l'éditeur offre la possibilité de sauvegarder dans un fichier d'extension .py (pour être directement reconnu et utilisable comme module par Python). C'est ce que nous ferons lorsque nous écrirons nos premiers programmes.

Nous étudierons plus en détail l'éditeur dans le prochain T.P.

4. Les bases du langage Python

Dans les langages de programmation, le concept de **type** est très important. En Python, tous les objets (entiers, fonctions,...) ont un type : ce type caractérise la manière dont l'objet est représenté en mémoire. Le type représente la nature de l'objet : c'est essentiel pour l'ordinateur de savoir quelle est la nature des objets qu'il manipule. Pour nous, humains, on sait très bien qu'on ne peut pas ajouter une pomme et une voiture : ces objets sont de type complètement différents. Mais pour l'ordinateur, tous les objets sont représentés en mémoire par une suite de 0 et de 1 : il pourrait ajouter sans problème deux telles suites, et ce, qu'elles représentent de la musique et une image par exemple !
C'est pourquoi la notion de type est essentielle : elle permet à l'ordinateur de différencier les objets manipulés, et en cas de mauvaise manipulation par l'utilisateur, au lieu d'ajouter inutilement des objets de natures différentes, il lui renvoie une erreur !

Testons le type de différents objets pour Python grâce à la fonction ... `type` et observons le résultat :

```
type(12)
type('Bonjour monde !')
type(3.7)
type(4+3j)
type(None)
type(m.pi)
type(m.sqrt)
type(type)
```

Nous verrons les types `str` et `builtin_function_or_method` plus tard même s'il est aisé de comprendre quelle est la nature des objets qui ont ces types. Intéressons nous aux types des nombres :

a. Représentation des nombres en Python

Python reconnaît trois types de nombres :
— Le type `int` : les nombres entiers tels que 6 par exemple ;
— Le type `float` : les nombres décimaux tels que 6.12 par exemple ;
— Le type `complex` : les nombres entiers tels que $5 + 3j$ par exemple ;

Un même nombre, 3 par exemple, a trois représentations mémoires différentes selon son type : 3 pour le type `int` ; 3.0 pour le type `float` et $3 + 0j$ pour le type `complex`.

Dans certains langages, dits fortement typés, il est impossible de faire une opération avec des nombres

de types différents !! En Python, heureusement, ce n'est pas le cas : si on multiplie un `int` et un `float` par exemple, Python aura "la présence d'esprit" de convertir les types automatiquement. La conversion se fait toujours de `int` vers `float` vers `complex`. Observons sur ces exemples :

```
2+2
2+2.0
2+(2+0j)
2.0+(2+0j)
```

On peut également demander manuellement demander la conversion d'un type de nombre vers un autre. Testons les codes suivant et voyons ce qui est possible ou pas :

```
float(34)
int(45.97)
complex(3)
complex(2.3)
float(2+0j)
```

b. Les Booléens

Comme en Mathématiques, la logique est un aspect essentiel de l'Informatique. En Python, il existe un type pour désigner les "V" (=vrai), "F" (=faux) utilisés en logique mathématique. Il s'agit du type `bool` et ce type ne contient que deux objets : `True` et `False` : **Vrai** et **Faux** donc ! Trois opérateurs sont associés à ce type : `not`, `or` et `and` qui correspondent exactement au "*non*", "*ou*" et "*et*" logique. On a donc par exemple : `not True = False` et `not False = True` - Faisons le test dans le shell !

Exercice 2.

Donner les tables des opérations `or` et `and` sur les deux objets de type `bool`.

Il existe d'autres opérateurs, dit opérateurs de comparaison, qui permettent de comparer de objets de type similaire et qui renvoient un objet de type `bool` : `True` ou `False`. Les voici, soit `o`, `o'` des objets de même type (ou du moins de types qui peuvent être convertis vers l'un ou l'autre)

`o == o'`	retourne `True` si o=o' ; `False` sinon
`o >= o'`	retourne `True` si o>=o' ; `False` sinon
`o <= o'`	retourne `True` si o<=o' ; `False` sinon
`o > o'`	retourne `True` si o>o' ; `False` sinon
`o < o'`	retourne `True` si o<o' ; `False` sinon
`o != o'`	retourne `True` si o≠o' ; `False` sinon

Testons tout ceci :

```
4+5 == 3+6
5 == 3
2+1 == 3 or 6>=8
'avion'>='fusee'
'a'<'b' and not 'd'<'c'
```

```
6 (1==2) != (2==2)
```

c. Les Variables

Pour mettre nos données en mémoire, nous aurons besoin des **variables**. Une variable est définie par un nom auquel on associe une valeur grâce à l'opérateur d'affectation =. Attention ! = n'a pas la même signification que == :

- le premier permet de **définir** ou **redéfinir** une variable en lui associant une valeur ;
- le deuxième est un "test" : c'est, comme on l'a vu plus haut, un opérateur qui à deux objets associe une booléen : `True` s'ils sont égaux ; `False` sinon.

Une fois déclarée, on peut récupérer la valeur d'une variable en faisant appel au nom de cette variable. Voyons comment cela fonctionne sur un exemple - on remarquera au passage que la casse est importante, une minuscule est différente d'une majuscule dans le nom d'une variable :

```
1 l = 5
2 L = 12
3 p = 2*(5+12)
4 print("Le périmètre d'un rectangle de largeur",l,"et de longueur",L,"est égal à",p)
```

Essayons de comprendre la ligne suivante en observant le résultat de l'appel de la variable :

```
1 L=L*2
2 L
```

On remarque que lors de l'affectation d'une nouvelle valeur à une variable, Python garde en mémoire (et c'est le cas de le dire) l'ancienne valeur de la variable et n'associe la nouvelle qu'après avoir effectué les calculs ! C'est extrêmement pratique !

Au niveau de la syntaxe, on a des écritures raccourcis pour certaines déclarations auto-référentes, ce qui permet un gain de temps lors de l'écriture de nos programmes :

- `L=L*2` est équivalent à `L*=2`
- `L=L+1` est équivalent à `L+=1`
- `L=L/4` est équivalent à .....
- `L=L*2` est équivalent à .....

Petit problème concernant les variables :

Exercice 3.

On déclare deux variables `x` et `y`. Donner une suite d'instructions qui permet d'échanger leur valeur.

On verra au tableau que Python permet aisément ce type d'affectations simultanée sans perdre de place en mémoire avec l'introduction de variables annexes.

d. Les chaînes de caractères

Comme on l'a vu pendant nos tests de la fonction `type`, les données alphanumériques - c'est-à dire des suites de lettres/nombres/symboles - sont de type `str` en Python. On appelle ce type de données **chaînes de caractères**. En Python, il existe plusieurs moyens de définir une chaînes de caractères ; voici les deux principaux :

Pour définir une chaînes de caractères en Python, on encadre les caractères voulus par :

— des guillemets simples ' (touche 4 du clavier en mode minuscule), ou

— des guillemets doubles " (touche 3 du clavier en mode minuscule).

D'accord, mais comment choisir ? Et bien cela dépend du contenu de la chaîne considérée : si celle-ci contient un symbole ', on l'encadrera par des guillemets doubles ", et réciproquement :
Testons les codes suivants dans la console :

```
1 "Le TP d'Informatique"
2 'est vraiment "cool"'
3 'n'est-ce pas ?'
```

Que se passe-t-il lorsqu'on exécute la troisième ligne ? Comment l'éviter ?

D'accord, mais si ma chaîne de caractères contient les deux sortes de guillemets ? Alors on les *échappe* grâce au *caractère d'échappement*, le *backslash* : \.

Testons :

```
1 'n\'est-ce pas ? "oui !"'
2 "\"Citation\""
```

Le backslash sert également à définir des caractères spéciaux comme le \n qui est interprété comme un passage à la ligne :

```
1 print("Allez, on passe\n à la ligne")
```

On peut réaliser diverses opérations sur les chaînes de caractères. Testons les instructions suivantes pour déterminer le comportement des opérations sur les chaînes.

Au passage, vous remarquerez l'utilisation dans le code suivant du caractère # suivi d'une explication : il s'agit d'un commentaire du code en Python. Tout se qui se trouve après un # ne sera pas interprété par Python. Lorsque nous écrirons des programmes dans l'éditeur, nous prendrons l'habitude de commenter nos instructions de manière à rendre compréhensible (sans trop s'appesantir) nos différentes lignes de code pour une autre personne (ou pour soi-même pour une future relecture).

```
1 'Infor'+'matique'
2 s="Sainte Croix "
3 s+="Saint Euverte " # équivalent à s=s+"Saint Euverte "
4 print(s)
5 s*5                 #équivalent à s+s+s+s+s
```

L'opération + pour des chaînes de caractères n'a pas la même signification que pour les objets de type `int` pour lesquels il représente l'addition. Pour les chaînes de caractères, comme on a pu s'en rendre compte grâce aux instructions précédentes, le symbole + correspond à la **concaténation** de deux chaînes

i.e. + met bout-à-bout les deux chaînes qu'on lui donne.

On comprend mieux la différence entre la concaténation de chaînes et l'addition de nombres grâce aux tests suivants :

```
'78'+'83'
78+83
'78'+83    #produit une erreur : les types sont incompatibles.
```

On peut accéder aux caractères d'une chaîne individuellement : chaque caractère est accessible grâce à son rang (son indice) dans la chaîne. On utilise les crochets pour désigner le rang d'un caractère dans un chaîne.

Attention : l'indice du 1er caractère d'une chaîne est 0 ! L'indice du 2eme caractère est 1, etc...

Testons :

```
chaine="Mathématiques Supérieures"
chaine[0]                                  # retourne le 1er caractère de chaine, ici "M"
chaine[6]                                  # retourne le 7eme caractère de chaine, ici "a
    "
chaine[13]                                 # un espace compte comme un caractère !
chaine[14]+chaine[15]+chaine[16]           # concatène les 15,16 et 17eme caractères
chaine[0]+chaine[1]+chaine[18]+chaine[14]
```

On peut aussi accéder aux caractères dans le sens inverse de la chaîne (i.e. de droite à gauche) en utilisant des indices négatifs : la dernier caractère a pour indice -1 ; l'avant dernier -2 ; etc...

```
chaine[-1]                # retourne le dernier caractère de chaine, ici "s"
chaine[-5]                # 5eme caractère de chaîne en partant de la droite, ici "e"
chaine[-25]*2+chaine[-11]
```

Exercice 4.

En utilisant la variable `chaine="Mathématiques Supérieures"`, écrire des instructions qui permettent de renvoyer les chaînes de caractères suivantes :

1. `"Maths"` ;
2. `"Maman"` ;
3. `"Super"` ;
4. `"Maaaaaaaaaaaaaaaaaaaaaaaaaths"` ;

On peut également accéder à la longueur d'une chaîne c'est-à-dire le nombre de caractères dans la chaîne grâce à la fonction **`len`** :

```
len(chaine) # retourne la longueur de chaine
```

Exercice 5.

Expliquer ce que va retourner l'instruction suivante pour une chaîne de caractères affectée à une variable `chn` :

```
1 chn[len(chn)-1]
```

On peut aller plus loin dans l'accès individuel aux caractères : on utilise la technique du **slicing** (découpage en tranches) pour extraire une suite de caractères dans une chaîne.

On doit préciser l'indice de début `m` et l'indice de fin `n` de la tranche à extraire, séparer par un double point `:`. Attention, l'indice `n` de fin est toujours **exclu** de la tranche ! Ainsi le nombre de caractères affichés par la tranche `[m:n]` est égal à `n-m`.

```
1 chaine[5:8]  # retourne du 6eme (indice 5) jusqu'au 8eme (indice 7) caractère
2 chaine[1:-1] # retourne du 2eme (indice 1) jusqu'à l'avant dernier (indice -2) caractère
```

De plus, si l'un des indices de début ou de fin de tranche n'est pas indiqué, Python considérera que tous les indices avant (ou après selon le cas) sont demandés. Observons :

```
1 chaine[3:] # retourne la chaîne du 4eme (indice 3) jusqu'au dernier caractère
2 chaine[:8] # retourne la chaîne du 1er jusqu'au 8eme (indice 7) caractère
3 chaine[:4]+chaine[-1]+chaine[13:17]
```

Exercice 6.

On considère la variable `chn="J'aime pas l'Info !"`. En utilisant le slicing sur la variable `chn`, écrire une instruction qui renvoie la chaîne de caractères :

`"J'aime l'Info "`

Exercice 7.

Comment retourner les 5 derniers caractères de `chaine` ? Ou plus généralement, comment afficher les k derniers caractères d'une chaîne ?

Et pour finir notre étude de l'accès individuel des caractères d'une chaîne, on notera une dernière méthode de slicing plus poussée : `chaine[m:n:p]` retourne tous les caractères dont les indices sont compris entre `m` (inclus) et `n` (exclu) et séparés d'un pas de `p`. Par exemple :

```
1 chaine[2:9:3] # retourne la chaîne du 3,6 et 9eme caractères (indices 2,5,8)
2 chaine[:14:5]
3 chaine[::2]   # expliquer cette instruction !
```

En mettant une valeur `k` de pas négative, on peut inverser simplement la chaîne de caractères :

```
1 chaine[::-1] # Inverse l'ordre des caractères de chaine
```