

Cours1

February 17, 2020

0.1 Premier pas, l'interpréteur une super calculette!

0.1.1 Addition, soustraction

```
In [97]: # Addition  
1 + 3
```

```
Out[97]: 4
```

```
In [98]: # Soustraction  
2 - 4
```

```
Out[98]: -2
```

0.1.2 Multiplications

```
In [26]: # Multiplication  
10 * 5
```

```
Out[26]: 50
```

```
In [27]: #Puissance  
10 ** 5
```

```
Out[27]: 100000
```

```
In [28]: # 10^5  
1e5
```

```
Out[28]: 100000.0
```

0.1.3 Divisions

```
In [29]: # Division  
10 / 3
```

```
Out[29]: 3.3333333333333335
```

```
In [30]: # Division entière  
10 // 3
```

```
Out[30]: 3
```

```
In [31]: # Reste de la division entière  
10 % 3
```

```
Out[31]: 1
```

0.1.4 Comparaisons

```
In [32]: 10 > 5
```

```
Out[32]: True
```

```
In [33]: 10 < 5
```

```
Out[33]: False
```

```
In [34]: 10 <= 5
```

```
Out[34]: False
```

```
In [35]: 10 >= 5
```

```
Out[35]: True
```

```
In [36]: 10 == 5
```

```
Out[36]: False
```

```
In [37]: 10 != 5
```

```
Out[37]: True
```

0.2 Affectation

On accède à une donnée dans la mémoire grâce à un NOM que l'on choisit (ex. age). Pour stocker une données, on utilise l'opérateur =

Attention le = ne correspond pas au = (égalité) des mathématiques. Pour tester si deux variables sont égales, il faut utiliser ==

```
In [38]: # Affectation  
age = 20
```

```
In [39]: # Vérification  
age
```

```
Out[39]: 20
```

Il est possible de modifier la valeur de la variable (**de façon irréversible**)

```
In [40]: # Réaffectation  
age = 40
```

```
In [41]: # Vérification  
age
```

```
Out[41]: 40
```

0.3 Opérateurs pratiques sur les variables

0.3.1 Permutation

```
In [42]: a = 5  
        b = 10  
        a,b = b,a
```

```
In [43]: a
```

```
Out[43]: 10
```

```
In [44]: b
```

```
Out[44]: 5
```

0.3.2 Incrémentations

```
In [59]: a = 1
```

```
In [60]: # Incrémentation simple  
        a = a + 1  
        a
```

```
Out[60]: 2
```

```
In [61]: # Incrémentation condensée  
        a += 1  
        a
```

```
Out[61]: 3
```

```
In [62]: # Rentracher une valeur  
        a-=1  
        a
```

```
Out[62]: 2
```

```
In [63]: # Multiplier par une valeur  
        a *= 10  
        a
```

```
Out[63]: 20
```

```
In [64]: # Diviser par une valeur  
        a /= 2  
        a
```

```
Out[64]: 10.0
```

0.4 Les types de variables

0.4.1 Les nombres entiers (int)

```
In [65]: a = 10  
        b = -15  
        c = 3e8
```

0.4.2 Les nombres réels (float)

```
In [66]: a = 3.14159  
        b = -12.4e-5  
        c = 3.
```

0.4.3 Les chaînes de caractères (str)

```
In [68]: phrase = "Il fait beau !"  
        phrase2 = ""Il fait beau !""
```

0.4.4 Les booléens (bool)

```
In [69]: test = True  
        test2 = False  
        test3 = 3 > 4
```

Attention lorsque l'on fait des opérations sur des variables de type différent. Ex : integer + string ?

```
In [70]: a = 10  
        b = "toto"  
        a + b
```

```
-----  
TypeError                                Traceback (most recent call last)
```

```
<ipython-input-70-04ab7bb24f21> in <module>
```

```
1 a = 10
```

```
2 b = "toto"
```

```
----> 3 a + b
```

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

0.5 Les fonctions standards

Fonction : suite d'instruction déjà enregistrées. pour l'exécuter, il faut connaître son nom et lui donner les arguments (informations) nécessaires

```
nom_de_la_fonction(argument1, argument2,
```

Librairie : ensemble de fonctions prêtes à être utilisées (déjà compilées)

0.5.1 La fonction `print(variable)`

affiche la valeur d'une variable!

```
In [71]: phrase = "Il fait beau"
        print(phrase)
```

Il fait beau

```
In [72]: suite = "aujourd'hui !!!"
        print(phrase,suite)
```

Il fait beau aujourd'hui !!!

0.5.2 La fonction `type(variable)`

affiche le type d'une variable

```
In [73]: a = 10
        type(a)
```

Out[73]: int

0.5.3 La fonction `help(nom_de_la_fonction)`

affiche laide (en anglais) de la fonction!

Avec jupyter-notebook, nous pouvez aussi utiliser ?

```
In [81]: help(print)
```

Help on built-in function print in module builtins:

```
print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

Prints the values to a stream, or to sys.stdout by default.

Optional keyword arguments:

file: a file-like object (stream); defaults to the current sys.stdout.

sep: string inserted between values, default a space.

end: string appended after the last value, default a newline.

flush: whether to forcibly flush the stream.

```
In [82]: ?print
```

0.5.4 La fonction `exit()`

pour quitter python

On ne va pas le faire dans jupyter-notebook

0.5.5 Les fonctions pour changer le type d'une variables `float(variable)`, `int(variable)`, `str(variable)`

```
In [84]: a = 10.5  
        b = int(a)  
        print(b)  
        type(b)
```

10

Out[84]: int

```
In [85]: a = 1000.5  
        b = str(a)  
        print(b)  
        type(b)
```

1000.5

Out[85]: str

0.5.6 La fonction `input()`

Demande à l'utilisateur d'entrer une valeur

```
In [86]: phrase = input()
```

Bonjour à tous.

```
In [87]: print(phrase)
```

Bonjour à tous.

```
In [88]: note = input()
```

10

```
In [89]: type(note)
```

Out[89]: str

```
In [90]: note = int(input())
```

```
10
```

```
In [91]: type(note)
```

```
Out[91]: int
```

```
In [92]: note = input("Entrer notre note :")
```

```
Entrer notre note :10
```

0.5.7 Fonction mathématiques usuelles

Pour avoir les fonctions mathématiques usuelles dans python, il est nécessaire de charger une bibliothèque externe. Nous allons utiliser la librairie numpy déjà installée.

Pour charger la librairie

```
In [93]: from numpy import *
```

On peut alors appeler les fonctions mathématiques classiques (sqrt, cos, sin, tan, log...)

```
In [94]: tan(1.23)
```

```
Out[94]: 2.8198157342681518
```

```
In [95]: exp(10)
```

```
Out[95]: 22026.465794806718
```

```
In [96]: arcsin(0.2)
```

```
Out[96]: 0.2013579207903308
```

Attention pour les fonctions trigonométriques, il faut utiliser les radians

TD2

February 17, 2020

1 Les structures conditionnelles (if, elif, else)

Les structures conditionnelles permettent d'effectuer des opérations lorsqu'une ou plusieurs conditions sont remplies.

1.1 Structure simple (if):

Le principe est de dire si (if) telle condition est vérifiée alors telles commandes sont effectuées.

Voici la syntaxe :

```
if (condition) :  
    commande1  
    commande2
```

- **Attention devant chaque commande, il faut placer 4 caractères vides (ou une tabulation). On appelle cela indenter.** Tant que les commandes sont indentées, elle ne seront exécutées que si la condition est satisfaite. Si l'on retire l'indentation, le programme reprend son cours normalement.
- **Attention de ne pas oublier les : à la fin de la commande if**

Voici un exemple, exécuter le plusieurs fois en modifiant la valeur de la note afin de voir le comportement du if:

```
In [1]: note = 10
```

```
if (note > 10 ) :  
    print("J'ai mon UE !")  
  
print("Ma note est : ", note)
```

```
Ma note est : 10
```

1.2 Structure complexe à deux possibilités (if...else)

La structure conditionnelle peut être étendue. Si (if) la condition est respectée, je fais cela, sinon (else) je fais autre chose.

```
if (condition) :  
    commande1  
else :  
    commande2
```

Voici un exemple, exécuter le plusieurs fois en modifiant la valeur de la note pour voir le comportement du if...else:

```
In [2]: note = 9  
  
print ("Ma note est :",note)  
  
if (note > 10 ) :  
    print("J'ai mon UE !")  
else :  
    print("Je dois repasser en session 2.")
```

```
Ma note est : 9  
Je dois repasser en session 2.
```

1.3 Structure complexe à plusieurs possibilités (if...elif...else)

La structure conditionnelle peut encore être étendue. Il est possible d'ajouter autant de conditions que l'on souhaite en ajoutant le mot clé elif, contraction de else et if, qu'on pourrait traduire par "sinon si".

```
if (condition1) :  
    commande1  
elif (condition2) :  
    commande2  
elif (condition3) :  
    commande3  
else :  
    commande4
```

Voici un exemple, exécuter le plusieurs fois en modifiant la valeur de la note pour voir le comportement du if...elif...else:

```
In [3]: nombre = -1  
  
if (nombre<0) :  
    print("C'est un nombre négatif.")
```

```
elif (nombre==0) :  
    print("c'est un nombre nul.")  
else :  
    print("C'est un nombre positif")
```

C'est un nombre négatif.

1.4 Conditions composées (and,or)

Il est possible de vérifier des conditions plus complexes.

Grâce à la commande and, on peut vérifier que deux plusieurs conditions à la fois. Voici un exemple :

```
In [4]: note = 11  
        if (note >= 10) and (note < 12) :  
            print("J'ai la mention assez bien.")
```

J'ai la mention assez bien.

Avec la commande or on peut vérifier si une condition est vérifiée parmi un ensemble de conditions.

```
In [5]: note = 9  
        if (note < 10) or (note >= 12) :  
            print("Je n'ai pas la mention assez bien.")
```

Je n'ai pas la mention assez bien.

Il est également possible de mettre le résultat d'une condition composée dans une variable de type bool.

```
In [6]: note = 9  
        condition = (note < 10) or (note >= 12)  
        if condition :  
            print("Je n'ai pas la mention assez bien.")
```

Je n'ai pas la mention assez bien.

1.5 Exercice 1 : mention à un examen

Entrer un note et donner la mention associée.

```
In [ ]:
```

1.6 Exercice 2 : racines d'un polynôme de degré 2

Chercher les racines d'un polynôme du second degré : $ax^2 + bx + c = 0$

Pour vous aider, voici un schéma logique du programme.

In []:

1.7 Exercice 3 : année bissextile

Vérifier si une année est bissextile.

Un année est bissextile si : - elle est un multiple de 4 mais pas de 100 - elle est un multiple de 400

In []:

1.8 Exercice 4 : franchise d'assurance auto

Votre assurance auto vous rembourse 10% des frais de réparation suite à un accident avec un minimum de 150 euros et un maximum de 1000 euros. Calculer le remboursement en fonction du montant des travaux.

In []:

1.9 Exercice 5 : jour de la semaine

A partir d'une date quelconque de l'année 2019 (ex. 15/09), calculer le nombre de jours qui s'est écoulé depuis le début de l'année (1/01).

In []:

1.10 Partie facultative

En *python*, il existe un type de base pour traiter les nombres complexes.

1.10.1 Exercice 6 (facultatif)

Reprendre l'exercice 2 (racines d'un polynôme de second degré) avec les complexes, mais sans *if*

In []:

TD3

February 17, 2020

1 Les boucles

boucle : répéter un bloc d'opération autant de fois que nécessaire.

Imaginons que l'on veuille afficher les valeurs de x^2 pour $x = 1, 2, 3 \dots 10$. On pourrait écrire :

```
In [ ]: print("1^2 = ", 1*1)
        print("2^2 = ", 2*2)
        print("3^2 = ", 3*3)
        print("4^2 = ", 4*4)
        print("5^2 = ", 5*5)
        print("6^2 = ", 6*6)
        print("7^2 = ", 7*7)
        print("8^2 = ", 8*8)
        print("9^2 = ", 9*9)
        print("10^2 = ", 10*10)
```

Cela fait beaucoup de lignes de codes donc beaucoup de risque d'erreur. Et si je veux maintenant aller jusqu'à 1000 ou afficher x^3 ou X^4 , ce n'est pas très adapté.

Si on regarde bien, on réalise en fait plusieurs fois la même chose en modifiant des paramètres. Dans ce cas, nous allons pouvoir utiliser les boucles.

Il y a deux types de boucle : les boucles `while` et les boucles `for`.

1.1 La boucle `while`

On répète un bloc ***tant que*** (*while*) une condition est respectée.

Cela s'écrit de la façon suivante :

```
while condition :
    instruction 1
    instruction 2
    instruction 3...
```

- Il est nécessaire d'avoir une condition qui finit par ne pas être satisfaite. Sinon la boucle ne s'arrête jamais.
- Les instructions doivent donc agir sur la condition.

- Les conditions peuvent être multiples.
- Il est possible de répéter le bloc sans savoir à l'avance combien de fois on va le répéter.

Essayons de faire un simple compteur de 0 à 10 :

```
In [5]: nb = 0      # initialisation du compteur

        while nb < 10 :    # condition : tant que nb est inférieur à 10
            print(nb)      # on affiche la valeur de nb
            nb = nb + 1    # on incrémente le compteur, et donc on modifie la condition qui
                          # finira par ne plus être satisfaite

0
1
2
3
4
5
6
7
8
9
```

Cette boucle offre de nombreuses possibilités : - Si je veux compter jusqu'à 100, on modifie simplement la condition `nb < 100` - Si je veux compter de 2 en 2, on modifie l'avancé du compteur `nb = nb + 2` - Si je veux faire un compte à rebours, on initialise `nb = 10`, on modifie la condition `nb > 0` et l'incrément `nb = nb - 1`

1.1.1 Exercice 1 :

- 1) Reprenez l'exemple du début (copié ci-dessous) et remplacez ces 10 lignes par une boucle `while` équivalente.

```
In [ ]: print("1^2 = ",1*1)
        print("2^2 = ",2*2)
        print("3^2 = ",3*3)
        print("4^2 = ",4*4)
        print("5^2 = ",5*5)
        print("6^2 = ",6*6)
        print("7^2 = ",7*7)
        print("8^2 = ",8*8)
        print("9^2 = ",9*9)
        print("10^2 = ",10*10)
```

- 2) Ajouter la possibilité de choisir facilement le nombre de valeurs à afficher (de 1 à 100, de 1 à 1000...).
- 3) Adapter ensuite ce code pour que l'on puisse facilement changer l'exposant.

1.1.2 Exemples de boucle `while` qui ne connaît pas à l'avance le nombre de fois que la boucle va être exécutée

Dans la boucle suivante, l'utilisateur va devoir entrer un nombre. Tant que ce nombre est négatif, l'ordinateur redemande à l'utilisateur de rentrer un nombre. A vous d'essayer.

```
In [6]: valeur = -1
        while valeur < 0 :
            valeur = float(input("Entrez un nombre positif : "))
        print("Merci")
```

```
Entrez un nombre positif : -1
Entrez un nombre positif : -3
Entrez un nombre positif : 2
Merci
```

Dans l'exemple suivant, nous allons faire une boucle totalement infini. Pour sortir de la boucle, il est alors possible d'utiliser la commande `break`.

```
In [2]: while True :
        valeur = input("Pour quitter, entrez Q :")
        if (valeur == "Q") :
            break
```

```
Pour quitter, entrez Q :1
Pour quitter, entrez Q :2
Pour quitter, entrez Q :3
Pour quitter, entrez Q :Q
```

1.2 La boucle `for`

On répète un bloc d'instruction **en parcourant une liste**.

Cela s'écrit de la façon suivante :

```
for element in liste :
    instruction 1
    instruction 2
    instruction 3....
```

Dans cette boucle `for`, la variable `element` va prendre tour à tour toutes les valeurs de la variable `liste`. Et à chaque fois, les instructions vont être exécutées.

Mais attendez, nous n'avons pas encore vu ce qu'était une liste !!!!

Pour commencer, en voici trois types. Mais nous en rencontrerons de nouveaux au fur et à mesure du cours.

1.2.1 Les listes - Simples ([]):

Pour fabriquer une liste en python, c'est très simple. Il suffit de mettre les éléments de la liste entre crochet [] et de séparer les éléments avec une virgure ,. Voici un exemple :

```
liste = [1,"lundi",0.45]
```

- Il est possible de mélanger les types dans une liste, mais attention à ce que vous ferez après. Ici nous avons un int, un str entre "et un float.

L'exemple suivant présente l'utilisation de la boucle for avec la liste précédente :

```
In [7]: liste = [1,"lundi",0.45]
```

```
for element in liste :  
    print("La variable element vaut : ", element)
```

```
La variable element vaut : 1  
La variable element vaut : lundi  
La variable element vaut : 0.45
```

Dans cet exemple, la variable element prend tour à tour la valeur 1 puis "lundi" puis 0.45. A chaque fois, l'ensemble des instructions est exécuté.

1.2.2 Les liste - Chaînes de caractères (str) :

Les chaînes de caractères (str) sont des listes. Grâce à la boucle for nous pouvons parcourir tous les caractères qui composent une chaîne de caractères. Voyons un exemple, ce sera plus parlant :

```
In [8]: phrase = """Il fait 30°C."""
```

```
for lettre in phrase : # grace à cette ligne, on parcourt toutes les lettres de la phras  
    print("La variable lettre vaut : ", lettre)
```

```
La variable lettre vaut : I  
La variable lettre vaut : l  
La variable lettre vaut :  
La variable lettre vaut : f  
La variable lettre vaut : a  
La variable lettre vaut : i  
La variable lettre vaut : t  
La variable lettre vaut :  
La variable lettre vaut : 3  
La variable lettre vaut : 0  
La variable lettre vaut : °  
La variable lettre vaut : C  
La variable lettre vaut : .
```

Ici la variable lettre parcourt les caractères (espaces compris) de la variable phrase.

1.2.3 Les listes - Suites d'entier (fonction range())

Une liste bien pratique est une suite d'entier. Par exemple : 0,1,2,3,4... On peut le faire de façon manuelle :

```
suite = [0,1,2,3,4,5,6,7,8,9]
```

La boucle for s'écrit alors :

```
In [9]: suite = [0,1,2,3,4,5,6,7,8,9]
```

```
for entier in suite :  
    print("La variable entier vaut : ", entier)
```

```
La variable entier vaut : 0  
La variable entier vaut : 1  
La variable entier vaut : 2  
La variable entier vaut : 3  
La variable entier vaut : 4  
La variable entier vaut : 5  
La variable entier vaut : 6  
La variable entier vaut : 7  
La variable entier vaut : 8  
La variable entier vaut : 9
```

Mais si la liste devient très grande, ce n'est pas bien pratique. On utilise donc la fonction range(). Cela fonctionne de la façon suivante :

```
range(debut,fin)
```

Une suite d'entier de 0 à 10 s'écrit ainsi : range(0,10).

La boucle for s'écrit alors :

```
In [10]: suite = range(0,10)
```

```
for entier in suite :  
    print("La variable entier vaut : ", entier)
```

```
La variable entier vaut : 0  
La variable entier vaut : 1  
La variable entier vaut : 2  
La variable entier vaut : 3  
La variable entier vaut : 4  
La variable entier vaut : 5  
La variable entier vaut : 6  
La variable entier vaut : 7  
La variable entier vaut : 8  
La variable entier vaut : 9
```


Il est également possible d'utiliser directement la fonction `range()` dans l'appel de la boucle `for`:

```
In [11]: for entier in range(0,10) :  
         print("La variable entier vaut : ", entier)
```

```
La variable entier vaut : 0  
La variable entier vaut : 1  
La variable entier vaut : 2  
La variable entier vaut : 3  
La variable entier vaut : 4  
La variable entier vaut : 5  
La variable entier vaut : 6  
La variable entier vaut : 7  
La variable entier vaut : 8  
La variable entier vaut : 9
```

1.2.4 Exercice 2 : Reprenons l'exemple initial :

- 1) Reprenez l'exemple du début et remplacez ces 10 lignes par une boucle `for` équivalente.

```
In [ ]: print("1^2 = ",1*1)  
        print("2^2 = ",2*2)  
        print("3^2 = ",3*3)  
        print("4^2 = ",4*4)  
        print("5^2 = ",5*5)  
        print("6^2 = ",6*6)  
        print("7^2 = ",7*7)  
        print("8^2 = ",8*8)  
        print("9^2 = ",9*9)  
        print("10^2 = ",10*10)
```

Vous savez tout ce dont nous aurons besoin sur les boucles `for` et `while`. N'oubliez pas que dans la boucle vous pouvez utiliser toutes les instructions que vous voulez. Par exemple, on peut mettre un `if` dans une boucle ou imbriquer plusieurs boucles. A vous de jouer.

1.2.5 Exercice 3 : Notes de contrôle

Lors d'un semestre, un lycéen a eu les notes suivantes :
10.5; 12.5; 19; 4.5; 10.5; 15; 8; 6.5; 14; 17; 13; 8.5; 12; 15; 5; 2; 7; 10; 15.5; 20; 19; 5; 1.5

- 1) Ecrire un code qui calcule la moyenne de ces notes.

```
In [ ]:
```

- 2) Modifiez ce code pour qu'il recherche en même temps la meilleure note que le lycéen a eu.
- 3) Modifiez ce code pour qu'il compte en même temps combien de notes sont au dessus de la moyenne.

1.2.6 Exercice 4 : Suite de Fibonacci

La suite de Fibonacci peut être considérée comme le tout premier modèle mathématique en dynamique des populations ! Elle y décrit la croissance d'une population de lapins sous des hypothèses très simplifiées, à savoir : chaque couple de lapins, dès son troisième mois d'existence, engendre chaque mois un nouveau couple de lapins, et ce indéfiniment.

Mathématiquement, la suite F_n s'écrit comme cela :

$F_0 = 0$ $F_1 = 1$ $F_n = F_{n-1} + F_{n-2}$

Ecrire un code pour déterminer combien de mois (i.e. la valeur n) pour avoir plus de 100 lapins.

In []:

1.2.7 Exercice 5 : Nombres premiers

Nombre premier : nombre qui ne peut être divisé que par lui-même et par 1.

Ecrire un programme qui affiche les nombres premiers inférieurs à 1000.

In []:

TD4

February 17, 2020

1 Les fonctions en python

Les fonctions permettent de préparer un bloc d'instructions que l'on pourra appeler et reappeler plus tard grâce à un nom de fonction. Nous avons déjà vu des fonctions usuelles `print()`, `input()`, `int()`...

1.1 Définir une fonction

Il est possible de créer ses propres fonctions. Lors de la création, il faut définir le nom de la fonction ainsi que les arguments qui seront nécessaires. Voici la syntaxe :

```
def nom_de_fonction(argument1,argument2) :  
    instruction 1  
    instruction 2  
    instruction 3...
```

- Le mot clé `def` permet à python de savoir que vous allez définir une fonction.
- Nous nous servirons ensuite du `nom_de_fonction` pour appeler la fonction.
- Les arguments seront à fournir pour que la fonction puissent opérer.

Voici un petit exemple :

```
In [23]: def cube(x) :  
         print(x*x*x)
```

Une fois définie, l'appel de la fonction se fait de la façon suivante :

```
In [24]: cube(4)  
         cube(10.1)
```

```
64  
1030.301
```

1.2 Valeurs par défaut des arguments

Il est souvent utile de préciser des valeurs par défaut à chaque paramètre. Pour cela nous allons à l'aide de = donner ces valeurs par défauts lors de la création de la fonction.

```
def nom_de_fonction(argument1 = valeur_defaut1, argument2 = valeur_defaut2) :  
    instruction 1  
    instruction 2  
    instruction 3...
```

Reprenons l'exemple précédent en précisant une valeur par défaut.

```
In [25]: def cube(x = 2) :  
         print(x*x*x)
```

La fonction donne les mêmes résultats que précédemment. Mais il est possible de l'appeler sans lui donner d'argument. Dans ce cas, nous obtiendrons toujours 2^3 :

```
In [26]: cube(4)  
         cube(10.1)  
         cube()           # Utilisation de la fonction avec les paramètres par défaut  
  
64  
1030.301  
8
```

Voici un deuxième exemple. Cette fonction prend deux arguments nombre et max. Elle affiche la table de multiplication associée à nombre de 0 jusqu'à max :

```
In [4]: def TableMultiplication(nombre=2, max=10):  
        for i in range(0,max+1):  
            print(i,"*",nombre,"=",i*nombre)
```

```
In [5]: TableMultiplication(2,5) # on demande la table de 2 de 0 à 5
```

```
0 * 2 = 0  
1 * 2 = 2  
2 * 2 = 4  
3 * 2 = 6  
4 * 2 = 8  
5 * 2 = 10
```

```
In [6]: TableMultiplication(5) # on demande la table de 5. On ne spécifie pas max.  
        # La table ira jusqu'à 10, la valeur par défaut
```

```
0 * 5 = 0  
1 * 5 = 5  
2 * 5 = 10
```

```
3 * 5 = 15
4 * 5 = 20
5 * 5 = 25
6 * 5 = 30
7 * 5 = 35
8 * 5 = 40
9 * 5 = 45
10 * 5 = 50
```

```
In [7]: TableMultiplication(max=5) # on peut préciser quel argument on veut modifier
```

```
0 * 2 = 0
1 * 2 = 2
2 * 2 = 4
3 * 2 = 6
4 * 2 = 8
5 * 2 = 10
```

```
In [9]: TableMultiplication(max=5, nombre=3) # et ce dans l'ordre que l'on veut
```

```
0 * 3 = 0
1 * 3 = 3
2 * 3 = 6
3 * 3 = 9
4 * 3 = 12
5 * 3 = 15
```

1.3 Sortie d'une fonction (return)

Les fonctions précédentes ne font qu'afficher des choses et il n'est pas possible de se servir d'un résultat obtenu en dehors de la fonction.

Afin de récupérer la *sortie* d'une fonction, on utilise la commande `return` puis on indique ce que l'on veut *sortir*. Reprenons la fonction `cube()`. On peut placer le résultat de cette fonction dans une variable :

```
In [1]: def cube(x = 2) :
        return x*x*x
```

```
resultat = cube(10) # On affecte le resultat de cube(10) dans la variable resultat
```

```
print("La variable résultat vaut : ", resultat)
```

```
La variable résultat vaut : 1000
```

Il est également possible de retourner plusieurs résultats en même temps. Le résultat est sous forme de `list`. Nous verrons dans le TD suivant ce qui nous pouvons en faire.

```
In [13]: def classer(x,y) :
          a,b = x,y
          if a > b :
              a,b = b,a
          return a,b

          classer(2,1)
```

```
Out[13]: (1, 2)
```

1.4 Les librairies de fonctions

Nous avons déjà chargé une librairie de fonctions : *numpy* Pour cela nous avons utilisé la commande :

```
from numpy import *
```

Pour que cela fonctionne, il faut naturellement que la librairie *numpy* soit installée. On peut alors utiliser les fonctions mathématiques de *numpy*, exemple :

```
sqrt(12)
tan(15)
```

Cet appel n'est en réalité pas très propre. La bonne façon d'appeler une librairie est la suivante :

```
import numpy as np
```

Pour utiliser la librairie *numpy*, il faut maintenant écrire :

```
np.sqrt(12)
np.tan(15)
```

Cela pourrait paraître plus fastidieux, mais maintenant lorsque nous appelons une fonction, nous savons dans quelle librairie nous allons la chercher. Lorsque l'on utilise plusieurs librairie, cela évite de se tromper et cela accélère *python*.

Voici un exemple d'erreur. Ici nous allons charger deux librairies mathématiques. *numpy* et *math*. Lorsque l'on appelle la fonction racine (`sqrt()`), nous voyons qu'il y a une différence.

```
In [7]: import numpy as np
          import math as mt

          print(mt.sqrt(12))
          print(np.sqrt(12))

3.4641016151377544
3.46410161514
```

Si maintenant, nous prenons la racine d'un complexe, on voit que *numpy* y arrive, mais pas *math*.

TD5

February 17, 2020

1 Les listes

Nous avons vu au TD3 que nous pouvions créer facilement des listes en python. Nous allons voir plus en détail ce qu'il est possible de faire avec ces listes.

list : objet qui peut contenir plusieurs objets de différents types

1.1 Création d'une liste :

1.1.1 Création simple [element1,element2...]

Pour fabriquer une liste en python, c'est très simple. Il suffit de mettre les éléments de la liste entre crochets [] et de séparer les éléments avec une virgule ,. Voici un exemple :

```
maliste = [1,"lundi",0.45]
```

- Il est possible de mélanger les types dans une liste, mais attention à ce que vous ferez après. Ici nous avons un int, un str entre "et un float.

1.1.2 Liste vide ([])

On peut aussi créer une liste vide, en n'indiquer aucun élément :

```
maliste = []
```

Ceci peut sembler surprenant et inutile, nous verrons plus loin que si, ça peut être utile.

```
In [2]: maliste=[] # On crée une liste vide
        type(maliste)
```

```
Out[2]: list
```

1.2 Accéder à / modifier un élément de la liste

On peut accéder aux éléments d'une liste grâce à son indice, entre crochets [] : - le premier élément a l'indice 0 - le second a l'indice 1 - le n^{ime} a l'indice $n - 1$

Ex:

```
In [44]: maliste = ['a','b','c','d','e','f','g']
         print(maliste[0])
         print(maliste[3])
```

a
d

Il est alors possible de modifier un élément de la liste. Pour cela on affecte (=) une nouvelle valeur à l'élément que l'on veut modifier :

```
In [45]: maliste[2]='toto'  
         print(maliste)  
  
['a', 'b', 'toto', 'd', 'e', 'f', 'g']
```

Il est également possible d'extraire plusieurs éléments consécutifs. On indique entre [] l'index du premier élément et celui auquel on s'arrête (non inclus), en les séparant par : (ceci s'appelle techniquement une tranche, ou une slice en anglais)

```
In [47]: maliste[1:3] # on extrait ici les élément d'index 1 et 2. Le 3 est exclu.  
  
Out[47]: ['b', 'toto']
```

1.3 Ajouter un élément dans une liste

1.3.1 Ajout à la fin (append())

Ici nous allons voir quelque chose de nouveau. Nous allons appliquer une fonction qui n'agit que sur un seul type (ici le type `list`). La syntaxe est donc différente de ce que l'on a vu précédemment.

Pour appeler une fonction associée à un type, on indique le nom de la variable suivie d'un `.` puis le nom de la fonction avec ses arguments.

Pour ajouter un élément à la fin d'une liste, on utilise la **fonction** `append()` **de la classe** `list`. Cette fonction prend en argument la valeur à ajouter :

```
maliste.append("h")
```

Voici un exemple :

```
In [10]: maliste = ['a','b','d','e','f','g']  
         maliste.append("h")  
         print(maliste)  
  
['a', 'b', 'd', 'e', 'f', 'g', 'h']
```

1.3.2 Insertion d'un élément (insert())

La fonction `insert()` est également **une fonction associée au type** `list`. Elle prend deux arguments : l'indice où l'on va insérer l'élément, et la valeur que l'on souhaite insérer :

```
maliste.insert(2,"c")
```

Regardons ce que cela donne :


```
In [11]: print(maliste)
          maliste.insert(2, "c")
          print(maliste)

['a', 'b', 'd', 'e', 'f', 'g', 'h']
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
```

Remarque : lorsque l'indice d'insertion vaut n , la méthode va décaler les éléments d'indice supérieur à n , pour intercaler la valeur supplémentaire.

1.4 Supprimer un élément (remove())

La fonction `remove()` associée au type `list` permet de supprimer un élément. Cette fonction prend en argument, **non pas l'indice de l'élément à supprimer, mais l'élément lui-même** :

```
maliste.remove("c")
```

Exemple :

```
In [1]: maliste = ['a', 'b', 'c', 'd', 'c', 'e']
          print(maliste)
          maliste.remove("c")
          print(maliste)

['a', 'b', 'c', 'd', 'c', 'e']
['a', 'b', 'd', 'c', 'e']
```

On note que s'il y a plusieurs fois le même élément, c'est le premier qui est supprimé.

1.5 Concaténer des listes (+)

Nous l'avons déjà beaucoup pratiqué sur les chaînes de caractères (il se trouve qu'il s'agit d'un type de `list`).

Il est possible de concaténer (de mettre bout à bout) des listes grâce à l'opérateur `+`:

```
maliste1+maliste2
```

exemple :

```
In [24]: maliste1 = ["a", "b", "c"]
          maliste2 = ["d", "e", "f"]
          maliste1 + maliste2

Out[24]: ['a', 'b', 'c', 'd', 'e', 'f']
```

1.6 Nombre d'éléments dans une liste (len())

La fonction len() permet de connaître le nombre d'éléments d'une liste :

```
len(maliste)
```

Exemple :

```
In [16]: maliste = ["a","b","c"]
         len(maliste)
```

```
Out[16]: 3
```

1.7 Parcourir une liste :

1.7.1 Avec un boucle while :

Grâce à la fonction len(), nous connaissons le nombre d'éléments de la liste. Il est alors facile d'écrire une boucle while pour parcourir notre liste :

```
In [18]: liste = ['a', 'b', 'c', 'd', 'e', 'f']
         i = 0 #initialisation du compteur

         while i < len(liste) :
             print(liste[i])
             i += 1      # ne pas oublier de faire avancer le compteur
```

```
a
b
c
d
e
f
```

Cette méthode n'est pas la plus élégante ni la plus naturelle en python : elle nécessite entre autre de créer à la main un compteur. On évitera de s'en servir en python. Cette méthode reste cependant la plus utilisée dans les autres langages.

1.8 Avec une boucle for :

Nous avons déjà vu cela dans le TD3 sur les boucles. Nous allons parcourir la liste en prenant chaque élément, l'un après l'autre :

```
In [19]: liste = ['a', 'b', 'c', 'd', 'e', 'f']

         for element in liste :
             print(element)
```

a
b
c
d
e
f

Ici pas besoin de connaître le nombre d'élément, ni de créer un compteur. C'est bien plus joli. Mais sans compteur, nous ne savons pas à quel élément nous en sommes, ce qui sera gênant dans certaines situations. Il y a donc une dernière solution.

1.9 Avec une boucle for et la fonction enumerate() :

Il est plus simple de vous présenter un exemple et de le commenter :

```
In [22]: liste = ['a', 'b', 'c', 'd', 'e', 'f']

        for i, element in enumerate(liste) :
            print("L'élément",i,"vaut :",element)
```

```
L'élément 0 vaut : a
L'élément 1 vaut : b
L'élément 2 vaut : c
L'élément 3 vaut : d
L'élément 4 vaut : e
L'élément 5 vaut : f
```

Durant cette boucle for, la variable `element` va parcourir tous les éléments de la liste pendant que la variable `i` va servir de compteur et prendre ainsi le numéro de l'indice courant.

Il reste encore plusieurs choses à voir sur les listes, mais faisons dès maintenant quelques applications, pour s'exercer.

1.10 Exercice 1 : Echange de valeurs

Ecrire un programme qui échange les valeurs du premier et du dernier élément d'une liste. Ce programme doit fonctionner quelle que soit la liste initiale.

```
In [ ]:
```

1.11 Exercice 1 bis : Recherche de petits nombres

Ecrire un programme qui, dans une liste de nombres, par exemple (1, 13, 33, 2, 4, 40), supprime tous ceux qui sont supérieurs à 10.

1.12 Exercice 2 : Liste symétrique

Ecrire un programme qui vérifie si une liste est symétrique (liste identique à la liste à l'envers)

In []:

1.13 Exercice 3 : Encadrement d'angles [-180,180]

Ecrire un code qui remplace les angles de la liste suivante par leurs équivalents entre $[-180, 180]$.

```
liste_angle = [1234,17345,-31243,23,245,456,3600]
```

In []:

1.14 Exercice 4 : Trier une liste

Ecrire un programme qui trie une liste composée de nombres quelconques du plus petit au plus grand nombre (*attention aux petits malins, on vous demande de créer votre propre programme*).

```
liste = [435,324,456,56,567,-45,546,0,345,2,-5]
```

In []:

Reprenons le cours ici.

1.15 Les listes de listes

Si l'on relit la définition d'une liste, *objet qui peut contenir plusieurs objets*, rien ne nous empêche de placer des listes dans des listes. Par exemple :

```
In [18]: maliste = [{"Fer",26}, {"Ag",47}, {"Ca",20}, {"Al",13}]
```

Afin de récupérer une sous-liste, on utilise son index :

```
In [20]: print(maliste[0])
         print(maliste[3])
```

```
['Fer', 26]
```

```
['Al', 13]
```

Mais il est possible de récupérer directement l'élément d'une sous-liste, en utilisant un premier index pour sélectionner la sous-liste, puis un second pour sélectionner l'élément de la sous-liste :

```
In [23]: print(maliste[0][1])
         print("L'élément",maliste[0][0],"a",maliste[0][1],"protons.")
```

```
26
```

```
L'élément Fer a 26 protons.
```

1.16 Les compréhensions de liste :

The list comprehension permet de modifier ou de filtrer une liste très facilement.

1.16.1 Opérations simples :

Imaginons que nous voulons créer une liste en mettant au carré chaque élément d'une liste d'origine. L'idée qui vient tout de suite à l'esprit est de faire une boucle comme ceci (on remarque au passage l'intérêt de créer une liste vide, ici appelée *carre* : ceci permet d'utiliser la fonction `append()` sur l'objet *carre*, qui existe bien)

```
In [4]: liste = [0,1,3,-1,5, 6] # initialisation
        carre = [] #initialisation de la liste contenant les carrés

        for i,elt in enumerate(liste) :
            carre.append(liste[i]**2)
        print(carre)
```

```
[0, 1, 9, 1, 25, 36]
```

```
In [2]: liste = [0,1,3,-1,5, 6] # initialisation
        carre = [] #initialisation de la liste contenant les carrés

        for elt in liste :
            carre.append(elt**2)
        print(carre)
```

```
[0, 1, 9, 1, 25, 36]
```

Avec *python*, il est possible de faire cette même boucle en une ligne de commande, que nous commenterons après :

```
In [1]: liste = [0,1,3,-1,5, 6] # initialisation

        carre = [elt**2 for elt in liste]
        print(carre)
```

```
[0, 1, 9, 1, 25, 36]
```

Parcourons la seconde ligne de droite à gauche :

- la variable `elt` parcourt les éléments de la liste, grâce à la commande `for elt in liste`.
- pour chaque élément, on calcule `elt**2`.
- les `[]` indiquent que les résultats précédents vont créer une liste
- liste que l'on affecte à la variable `carre` grâce à l'opérateur `=`

1.17 Filtres sur une liste

Il est également possible d'ajouter une condition pour ne sélectionner qu'une partie des éléments d'une liste.

```
In [62]: nombres = [-11,10,9,-8,12,-4,20]

         nombres_positifs = [elt for elt in nombres if elt > 0]
         print(nombres_positifs)

[10, 9, 12, 20]
```

- ici elt parcourt les éléments de la liste nombres en ne considérant que les nombres positifs.
- pour chaque élément retenu, on retourne sa valeur pour former une nouvelle liste nombres_positifs.

1.18 Fonctions sur les listes :

1.18.1 Fonctions usuelles

Il existe plusieurs fonctions usuelles bien pratiques à connaître. Vous allez les découvrir au fur et à mesure. En voici quelques unes. Leur nom est assez explicite :

```
In [12]: nombres = [-11,10,9,-8,12,-4,20]

         print(sum(nombres)) # fait la somme des éléments d'une liste
         print(max(nombres)) # retourne le maximum d'une liste
         print(min(nombres)) # retourne le minimum d'une liste
         sorted(nombres) # ordonne du plus petit au plus grand les éléments d'une liste

28
20
-11
```

```
Out[12]: [-11, -8, -4, 9, 10, 12, 20]
```

```
In [10]: mots = ['bonjour','girafe','schtroumpf','tagada', 'ananas', 'zoo']

         print(max(mots)) # retourne le maximum d'une liste
         print(min(mots)) # retourne le minimum d'une liste
         sorted(mots) # ordonne du plus petit au plus grand les éléments d'une liste

zoo
ananas
```

```
Out[10]: ['ananas', 'bonjour', 'girafe', 'schtroumpf', 'tagada', 'zoo']
```

1.18.2 Fonctions provenant de bibliothèques

Il existe beaucoup de bibliothèques contenant des fonctions s'appliquant aux listes. A vous de les chercher en fonction de vos besoins. En voici une :

```
In [65]: from statistics import mean, median
         median(nombres) # retourne la valeur médiane
         mean(nombres) # retourne la valeur moyenne
```

Out[65]: 4

1.18.3 Visualiser des données numériques

Nous arrivons ici à un point ultra important pour un scientifique, la visualisation. Ici nous allons voir une première façon d'afficher un graphique représentant des données.

Nous allons utiliser la bibliothèque **matplotlib** pour afficher notre première courbe à partir de deux listes. Importons cette bibliothèque :

```
In [34]: import matplotlib.pyplot as plt # on importe la bibliothèque
```

Si sous linux, cette bibliothèque n'est pas installée, lancer un Terminal puis taper : `python3 -m pip install -U matplotlib --user`

Imaginons que nous voulons afficher la fonction x^2 entre 0 et 10. Nous allons créer deux listes, une contenant les abscisses, l'autre les ordonnées :

```
In [78]: x = range(0,10)
         y = [i*i for i in x]
```

Remarque : vous auriez peut-être voulu écrire quelque chose du type suivant, et non ça ne marche pas, python ne sait pas ce que signifie le carré d'une liste.

```
In [13]: x = range(0,10)
         y = x**2
```

```
-----
TypeError                                Traceback (most recent call last)

<ipython-input-13-3c3ad855defc> in <module>
      1 x = range(0,10)
----> 2 y = x**2

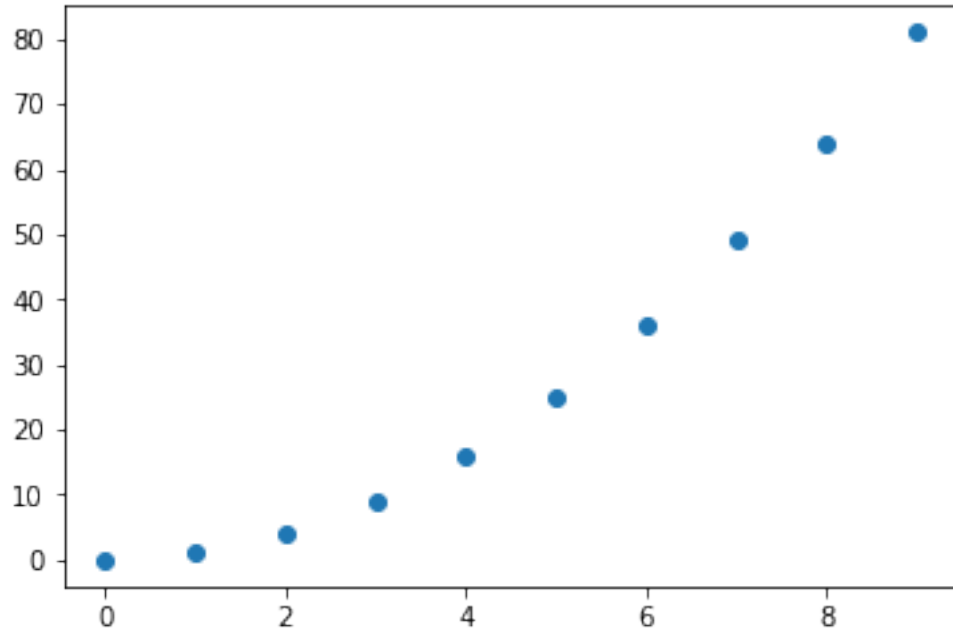
TypeError: unsupported operand type(s) for ** or pow(): 'range' and 'int'
```

Reprenons nos listes correctement construites.

```
In [14]: x = range(0,10)
         y = [i*i for i in x]
```

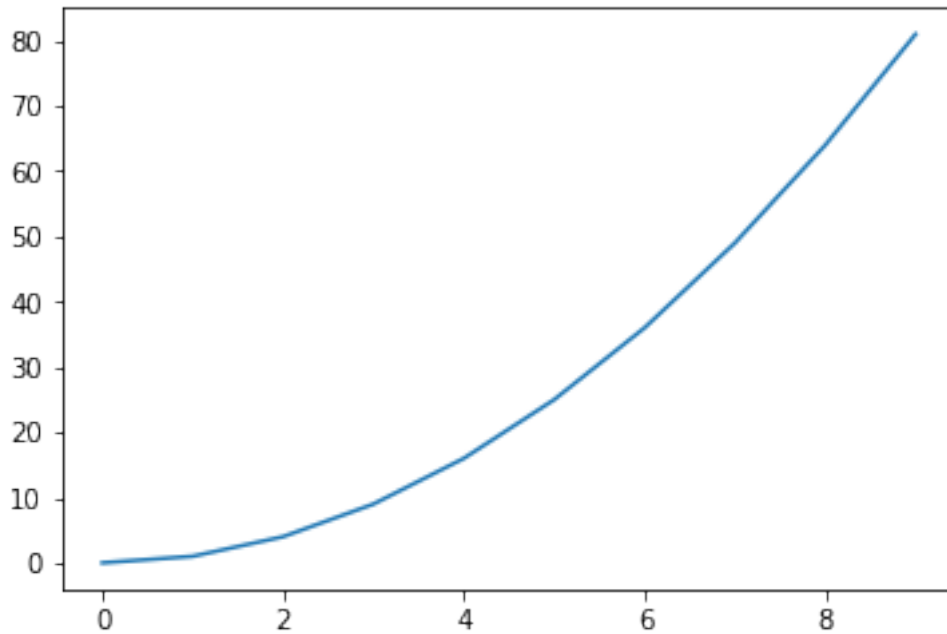
On peut alors afficher y en fonction de x à l'aide d'un nuage de points :

```
In [79]: plt.scatter(x,y) # on génère un graphique point par point  
plt.show() # on affiche le graphique
```



On peut également afficher une courbe en remplaçant scatter par plot. Les points sont alors reliés par des segments.

```
In [80]: plt.plot(x,y) # on génère une courbe  
plt.show() # on affiche le graphique
```

1.19 Exercice 5 :

Produire la courbe représentative de la fonction $f(x) = \sin(x)$ entre 0 et 10π .

In []:

1.20 Exercice 6 :

Ecrire une fonction qui calcule le produit vectoriel de deux vecteurs. Les paramètres d'entrée seront deux list (vec1,vec2) et le résultat sera également une liste. On rappelle que le produit vectoriel est donné par :

In []:

1.21 Exercice 7 : Tableau périodique

Atomes = [[`"Fer"`,26],[`"Ag"`,47],[`"Ca"`,20],[`"Al"`,13],[`"Ne"`,10],[`"O"`,8],[`"Au"`,79]]

Ecrire un programme qui trie les éléments chimiques précédents par ordre croissant de numéro atomique.

In []:

1.22 Exercice 8 : fonction créneaux

Tracer une fonction $f(x)$ entre 0 et 10 avec des points tous les 0.1, telle que :

- $f(x) = 1$ si la partie entière de x est paire
- $f(x) = 0$ si la partie entière de x est impaire

In []:

TD6

February 17, 2020

1 Lire et écrire un fichier

Dans ce TD, nous allons voir comment ouvrir, lire et écrire un fichier. Ce TD requière que le fichier *fable.txt* soit placé dans un répertoire nommé *fichiers*.

1.1 Ouvrir un fichier (open())

Pour ouvrir un fichier, on utilise la fonction `open()` en lui indiquant le chemin (relatif ou absolu) du fichier ainsi que le mode d'ouverture :

- `r` pour *read* : le fichier sera accessible en lecture seule
- `w` pour *write* : le fichier sera ouvert en écriture et le contenu sera écrasé.
- `a` pour *append* : le fichier sera ouvert en écriture. L'écriture se fera en fin de fichier et le contenu ne sera pas perdu
- `b` pour *binary* : cette option peut s'ajouter au précédente. Elle permet de spécifier que le fichier est un fichier *binnaire*. Nous y reviendrons plus tard.

L'appel de la fonction se fait de la façon suivante :

```
In [59]: mon_fichier = open("fichiers/fable.txt","r") # ouverture en mode lecture seule
         type(mon_fichier)
```

```
Out[59]: _io.TextIOWrapper
```

La fonction `open()` retourne un objet de type `TextIOWrapper`. Même si nous ne regarderons pas en détail ce type d'objet, nous allons voir comment s'en servir. La fonction associée `read()` retourne le contenu du fichier sous forme d'un *gros* `str`.

```
In [60]: contenu = mon_fichier.read()
         type(contenu)
```

```
Out[60]: str
```

On peut donc utiliser tout ce que l'on sait sur les `str`.

```
In [61]: print(contenu)
```

Maître Corbeau, sur un arbre perché,
Tenait en son bec un fromage.
Maître Renard, par l'odeur alléché,
Lui tint à peu près ce langage :
Et bonjour, Monsieur du Corbeau,
Que vous êtes joli ! que vous me semblez beau !

1.2 Fermer un fichier (close())

Pour fermer un fichier ouvert, on utilise la fonction associée `close()` sur l'objet de type `TextIOWrapper`.

```
In [62]: mon_fichier.close()
```

1.3 Ecrire des str dans un fichier (write())

Pour écrire dans un fichier, il faut tout d'abord l'ouvrir. On peut ouvrir un fichier existant, mais aussi ouvrir un fichier qui n'existe pas encore. Dans ce cas il sera créé.

```
In [63]: mon_fichier = open("fichiers/nouveau.txt", "w") # création du fichier nouveau.txt
```

On peut alors ajouter écrire du texte dans le fichier sous forme de `str` avec la fonction associée `write()` :

```
In [64]: mon_fichier.write(contenu)
          mon_fichier.write("Sans mentir, si votre ramage")
          mon_fichier.write("Se rapporte à votre plumage,")
          mon_fichier.write("Vous êtes le Phénix des hôtes de ces bois.")
```

```
Out[64]: 42
```

La fonction `write()` renvoie le nombre de caractères ajoutés. Ici le 42 correspond à la dernière commande `write()`.

Il ne reste plus qu'à fermer le fichier.

```
In [65]: mon_fichier.close()
```

Vous pouvez vérifier que dans le répertoire `fichiers`, le fichier `nouveau.txt` a été créé et qu'il contient le texte précédent. Vous pouvez utiliser la commande shell `cat` pour afficher le contenu du fichier.

1.4 Fonctions associées aux str

Jusqu'à présent, nous n'avons pas vraiment regardé les fonctions associées aux `str`. La lecture et l'écriture de `str` dans un fichier est l'occasion de revenir sur plusieurs fonction qui peuvent être utiles. Nous ne serons pas exhaustif. N'hésitez pas à chercher sur internet...

Avant d'aller plus loin, Nous rappelons que les chaînes de caractères sont des listes. Vous pouvez donc utiliser toutes les méthodes que nous avons vues dans le TD 5.

1.4.1 Fonctions simples :

```
In [77]: texte = " mon TEXTE "
         texte.lower() # met tout en minuscule

Out[77]: ' mon texte '

In [78]: texte.upper() # met tout en majuscule

Out[78]: ' MON TEXTE '

In [79]: texte.capitalize() # met une majuscule en début de phrase et le reste en minuscule

Out[79]: ' mon texte '

In [80]: texte.strip() # retire les espaces en début et fin de chaîne

Out[80]: 'mon TEXTE'

In [87]: texte.find("TEXTE") # cherche une chaîne de caractères
         # et renvoie l'index du début de la chaîne (ici 6).
         texte[6]

Out[87]: 'T'

In [70]: texte = "La la la la la !!!"
         texte.replace("la","ho") # remplace une chaîne par une autre

Out[70]: 'La ho ho ho ho !!!'

In [71]: texte = "La la la la la !!!"
         texte.replace("la","ho",2) # remplace une chaîne par une autre,
         # un nombre de fois spécifié

Out[71]: 'La ho ho la la !!!'
```

1.4.2 Fonction associée format()

Cette fonction est très puissante. Elle permet de créer facilement des chaînes de caractères dynamique. Lors de la création de la chaîne de caractère, on place des *labels* entre {} qui seront remplacés par des valeurs spécifiées dans la fonction format(). Ok, regardons un exemple, ce sera plus parlant :

```
In [72]: texte = "Je m'appelle {prenom} et j'ai {age} ans." # deux labels {prenom} et {age} sont
         print(texte) # on peut afficher la chaîne précédente.
```

Je m'appelle {prenom} et j'ai {age} ans.

```
In [73]: texte.format(prenom="Thomas",age=20) # la fonction format remplace ici les balises par
```

```
Out[73]: "Je m'appelle Thomas et j'ai 20 ans."
```

Notez qu'ici la variable *texte* n'est pas modifiée :

```
In [74]: print(texte)
```

Je m'appelle {prenom} et j'ai {age} ans.

Si l'on souhaite modifier la variable *texte* de façon définitive, on peut écrire :

```
In [75]: texte = texte.format(prenom="Thomas",age=20)
        print(texte)
```

Je m'appelle Thomas et j'ai 20 ans.

1.5 Exercice 1 : Tableaux périodiques

```
Atomes = [["Fer",26],["Ag",47],["Ca",20],["Al",13],["Ne",10],["O",8],["Au",79]]
```

- 1) Ecrire un programme qui parcourt la liste précédente et affiche pour chaque élément :
"L'élément XXX a pour numéro atomique YYY."
- 2) Modifier ce programme pour que le texte affiché soit maintenant sauvegardé dans un fichier.

1.6 Problème 1 : Fichier codé

Récupérer le fichier *code.txt* et placer un sous répertoire *fichiers* dans votre répertoire de travail.

Ce fichier est codé. Il va falloir le décoder. Le code est le suivant : - les chiffres 0,1,2,3,4,5,6,7,8,9 remplacent respectivement a,c,e,i,l,n,o,r,s,t - Chaque caractère (espace compris) a été échangé avec son voisin, exemple : "Le train arrive." -> "eLt arnia rroi.e"

- 1) Ouvrir le fichier et afficher le texte qu'il contient
- 2) Décoder le code

```
In [ ]:
```

1.7 Ecrire des objets dans un fichier (pickle)

Il est également possible d'enregistrer des *objets* comme des listes dans des fichiers et de les récupérer plus tard. Pour cela, nous allons utiliser la librairie *pickle*.

```
In [16]: import pickle
```

Comme précédemment, on ouvre en écriture (w) le fichier que l'on veut créer en ajoutant l'option b pour préciser que le fichier sera au format binaire.

Le fichier ne sera donc pas lisible par un humain, mais l'ordinateur pour y mettre des informations supplémentaires pour y stocker des objets.

Une fois le fichier ouvert, on utilise la fonction associée `pickle.dump(objet,fichier)` pour ajouter un objet dans le fichier. Il est possible d'ajouter plusieurs objets. Il ne reste plus qu'à fermer le fichier.

```
In [17]: fichier = open("fichiers/data.bin","wb")

Atomes = [ ["Fer",26], ["Ag",47], ["Ca",20], ["Al",13], ["Ne",10], ["O",8], ["Au",79]]
Nombre = [1,2,3,4]

pickle.dump(Atomes,fichier)
pickle.dump(Nombre,fichier)

fichier.close()
```

Pour récupérer plus tard, ce que nous avons mis dans le fichier, il faut réouvrir le fichier avec les options rb, puis charger un à un les objets sauvegardés.

```
In [18]: fichier = open("fichiers/data.bin","rb")

Atomes = pickle.load(fichier)
Nombre = pickle.load(fichier)

print(Atomes)
print(Nombre)

[['Fer', 26], ['Ag', 47], ['Ca', 20], ['Al', 13], ['Ne', 10], ['O', 8], ['Au', 79]]
[1, 2, 3, 4]
```

Noter que **vous devez savoir ce qu'il y a dans le fichier**. S'il y a deux objets et que vous en chargez trois, il y aura une erreur:

```
In [20]: Toto = pickle.load(fichier)
```

```
-----
EOFError                                Traceback (most recent call last)

<ipython-input-20-fb57e1700de5> in <module>
----> 1 Toto = pickle.load(fichier)

EOFError: Ran out of input
```

N'oubliez pas de fermer le fichier.

```
In [21]: fichier.close()
```

1.8 Exercice 2 : PIB par pays

Le fichier *PIB.bin* contient une liste d'éléments. Chaque élément est constitué du nom d'un pays, de son PIB par habitant et de son nombre d'habitants.

- 1) Charger le fichier, récupérer la liste.
- 2) Afficher à l'aide d'un nuage de point, le PIB par habitant en fonction du nombre d'habitants.
- 3) Calculer le PIB total de chaque pays. Quel Pays a le PIB total le plus important ?

```
In [ ]:
```

1.9 Problème 2 : Chûte libre

Un avion lâche une caisse de matériel d'une altitude H et une vitesse initiale horizontale \vec{v}_0 . Nous allons étudier la trajectoire de la caisse.

Si l'on néglige les frottements, la trajectoire s'obtient à partir du principe fondamental de la dynamique. Ici il n'y a que le poids \vec{p} qui agit donc :

- $a_x = 0$
- $a_z = -g$
- $v_x = v_0$
- $v_z = -gt$
- $x = v_0 t$
- $z = -1/2gt^2 + H$

On prendra $H = 10000 \text{ m}$, $g = 9.81 \text{ m.s}^{-2}$ et $v_0 = 100 \text{ m.s}^{-1}$

- 1) Tracer la trajectoire jusqu'au sol, c'est à dire z en fonction de x .
- 2) Cette chute a été enregistrée par une caméra. Le fichier *chute.bin* contient la trajectoire enregistrée sous forme de deux listes : la première correspond à x , la seconde à z . Tracer un même schéma la trajectoire enregistrée et celle calculée précédemment.
- 3) D'où provient la différence observée ?

```
In [ ]:
```


TD7

February 17, 2020

1 La simulation *pas à pas*

Nous ne verrons ici que le début du sujet car il faudrait bien plus de temps pour aborder ce domaine. Vous le ferez sûrement dans le futur.

Reprenons le dernier exercice comme exemple :

Un avions lâche une caisse de matériel d'une altitude H et une vitesse initiale horizontale \vec{v}_0 . Nous allons étudier la trajectoire de la caisse.

On prendra $H = 10000\text{ m}$, $g = 9.81\text{ m.s}^{-2}$ et $v_0 = 100\text{ m.s}^{-1}$

Le code ci-dessous affiche les données du fichier *chute.bin* avec quelques petits cosmétiques supplémentaires.

```
In [1]: # Ici on charge les données
```

```
import pickle
```

```
fichier = open("fichiers/chute.bin", "rb")
```

```
data_x = pickle.load(fichier)
```

```
data_z = pickle.load(fichier)
```

```
fichier.close()
```

```
In [4]: # Ici on affiche les données
```

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

```
fig, ax = plt.subplots()
```

```
ax.set_xlim((0, 5000))
```

```
ax.set_ylim((0, 12000))
```

```
#ax.set_aspect('equal') # Permet de faire que l'echelle des x et des z soient les mêmes
```

```
plt.arrow(0, 10000, 1000, 0, head_width=200.00, head_length=100.0, fc='r', ec='r', length=1000)
```

```
plt.scatter(data_x, data_z)
```

```
plt.show()
```

TD8

February 17, 2020

1 Numpy Array

Ce petit TP concerne une type numérique très pratique, les ndarray fournis par *numpy*.

De façon rapide, un ndarray est une liste qui ne contient qu'un seul type de variable (que des float, que des int...). L'avantage est qu'il permet des manipulations numériques que ne permet pas les listes.

1.1 Créer un ndarray

Après avoir importer la librairie *numpy*, il suffit d'utiliser la fonction `array()` pour convertir une liste `list` en ndarray :

```
In [4]: import numpy as np

        liste = [1,2,3,4]

        ndliste = np.array(liste)
        print(ndliste,type(ndliste))

[1 2 3 4] <class 'numpy.ndarray'>
```

La fonction `zeros(n)` permet de créer un ndarray composé de `n` valeurs nulles :

```
In [8]: nul = np.zeros(10)
        print(nul)

[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
```

Il est également possible de créer des ndarray à partir de fonctions bien pratiques :

La fonction `arange(debut,fin,pas)` (analogue à `range()`) un ndarray des valeurs réparties les deux bornes (`debut,fin`) avec un pas fixé (`pas`) :

```
In [7]: nb = np.arange(1,50,3)
        print(nb)

[ 1  4  7 10 13 16 19 22 25 28 31 34 37 40 43 46 49]
```

La fonction `linspace(debut,fin,n)` génère un `ndarray` de `n` nombre de valeurs uniformément réparties entre les bornes `(debut,fin)` :

```
In [9]: nb = np.linspace(0,50,4)
        print(nb)

[ 0.          16.66666667  33.33333333  50.          ]
```

La fonction `zeros.like(array)` permet de créer un `ndarray` de 0 ayant la même taille que `array` :

```
In [10]: nul = np.zeros_like(nb)
         print(nul)

[ 0.  0.  0.  0.]
```

Il est possible d'utiliser ces mêmes fonctions pour créer des matrices ($n \times m$) :

```
In [12]: mat = np.zeros((2,3))
         print(mat)

[[ 0.  0.  0.]
 [ 0.  0.  0.]]
```

Enfin, il est possible de créer des `ndarray` avec des nombres aléatoires. Il existe plusieurs fonctions pour effectuer le tirage :

```
In [52]: x = np.random.randint(low=10, high=30, size=6) # 6 nombres tirés aléatoirement entre 10
         x = np.random.normal(size=5) # 5 Nombres sur une loi normal...
         print(x)

[-0.45432073  1.19188413 -1.1259179   0.54479408 -1.7294817  ]
```

1.2 Opérations mathématiques :

Si deux `ndarray` ont la même taille, il est possible de faire des opérations mathématiques :

```
In [19]: tab = np.arange(0,10,1)
         print(tab)
         tab2 = tab *2 # on multiplie tous les éléments par 2
         print(tab2)

[0 1 2 3 4 5 6 7 8 9]
[ 0  2  4  6  8 10 12 14 16 18]
```

```
In [20]: tab3 = tab + tab2 # on additionne deux ndarray  
print(tab3)
```

```
[ 0  3  6  9 12 15 18 21 24 27]
```

```
In [23]: tab3 = tab * tab2 # On multiplie deux ndarray  
print(tab3)
```

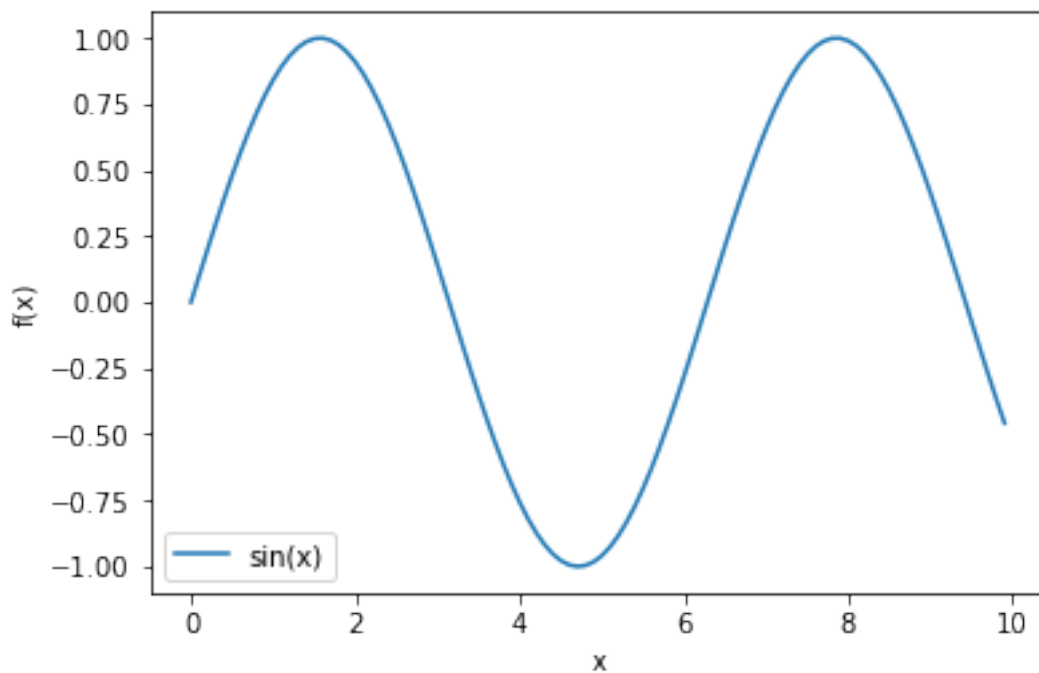
```
[ 0  2  8 18 32 50 72 98 128 162]
```

Petit exemple. Imaginons que l'on veuille afficher la fonction $\sin(x)$ entre 0 et 10 avec un pas de 0.1 :

```
In [35]: import matplotlib.pyplot as plt
```

```
x = np.arange(0,10,0.1)  
y = np.sin(x)
```

```
plt.plot(x,y,label="sin(x)")  
plt.legend()  
plt.xlabel("x")  
plt.ylabel("f(x)")  
plt.show()
```



C'est bien plus facile que les listes, non ?

1.3 Des listes comme les autres :

Les ndarray sont des listes comme les autres. Vous pouvez utiliser `for` , la compréhension de liste et récupérer un élément n avec `[n]` :

```
In [51]: nb = np.arange(0,10,1)
         for i in nb :
             print(i)

         print("La valeur en 1 :",nb[1])

         toto = [i*22 for i in nb]
         print(toto)

0
1
2
3
4
5
6
7
8
9
La valeur en 1 : 1
[0, 22, 44, 66, 88, 110, 132, 154, 176, 198]
```

1.4 Sélections par masque :

On peut facilement faire des sélections en appliquant un mask au ndarray.

Le plus simple est de voir un exemple. Imaginon que je souhaite dans l'exemple précédent sélectionner les points pour lesquels $\sin(x) > 0$. Nous allons créer un masque de bool qui vaut 1 lorsque $\sin(x) > 0$ et 0 sinon :

```
In [38]: z = (y>0)
         print(z)

[False  True  True  True  True  True  True  True  True  True  True  True
   True  True  True  True  True  True  True  True  True  True  True  True
   True  True  True  True  True  True  True  True False False False False
  False False False False False False False False False False False False
  False False False  True  True  True  True  True  True  True  True  True
   True  True  True  True  True  True  True  True  True  True  True  True
   True  True  True  True  True  True  True  True  True  True  True False
  False False False False]
```

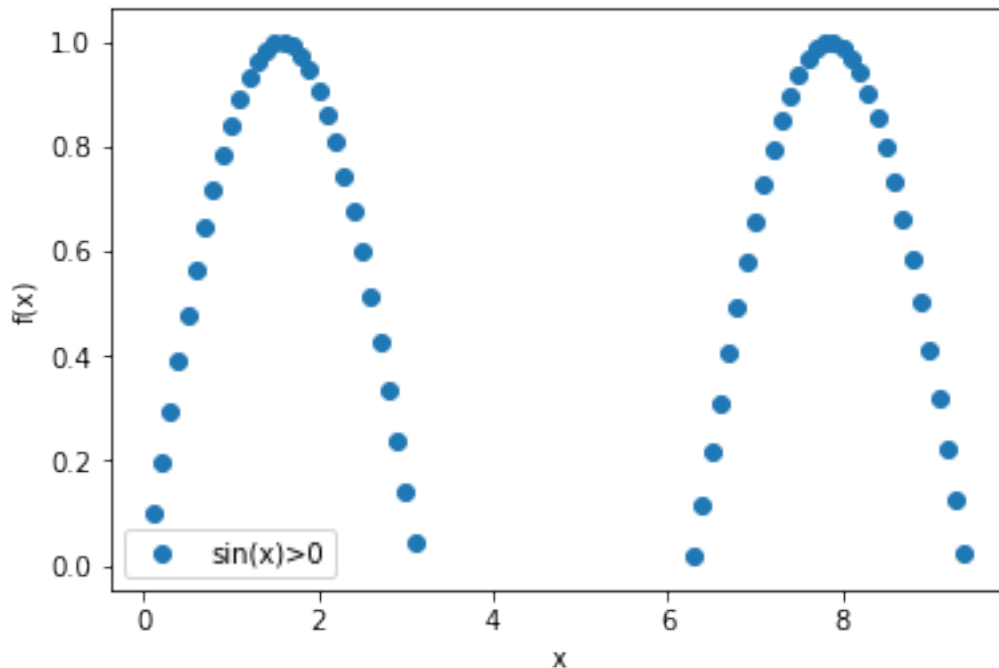
Pour appliquer notre masque `z` à `y`, il suffit alors de l'indiquer entre `[]` comme ceci :

```
In [39]: print(y[z])
```

```
[ 0.09983342  0.19866933  0.29552021  0.38941834  0.47942554  0.56464247
 0.64421769  0.71735609  0.78332691  0.84147098  0.89120736  0.93203909
 0.96355819  0.98544973  0.99749499  0.9995736  0.99166481  0.97384763
 0.94630009  0.90929743  0.86320937  0.8084964  0.74570521  0.67546318
 0.59847214  0.51550137  0.42737988  0.33498815  0.23924933  0.14112001
 0.04158066  0.0168139  0.1165492  0.21511999  0.31154136  0.40484992
 0.49411335  0.57843976  0.6569866  0.72896904  0.79366786  0.85043662
 0.8987081  0.93799998  0.96791967  0.98816823  0.99854335  0.99894134
 0.98935825  0.96988981  0.94073056  0.90217183  0.85459891  0.79848711
 0.7343971  0.66296923  0.58491719  0.50102086  0.41211849  0.31909836
 0.22288991  0.12445442  0.02477543]
```

On peut s'en servir même dans les plots :

```
In [41]: plt.scatter(x[z],y[z],label="sin(x)>0")
plt.legend()
plt.xlabel("x")
plt.ylabel("f(x)")
plt.show()
```



1.5 Exercice 1 : échauffement

Créer un ndarray d'entiers allant de 0 à 20, remplacer tous les nombres paires par des -1.

In [5] :

```
[-1  1 -1  3 -1  5 -1  7 -1  9 -1 11 -1 13 -1 15 -1 17 -1 19 -1]
```

1.6 Exercice 2 : Tracer des math

Utiliser *matplotlib* pour tracer sur un seul graphique la fonction $f(x) = e^{x/10} \sin(\pi x)$ et $g(x) = x e^{x/3}$ sur l'intervalle $[0, 10]$.

Ajouter les noms des abscisses et ordonnées ainsi que la légende des courbes.

Sauvegarder le graphique en png. A vous de chercher comment (*google* vient m'aider).

In [] :

1.7 Exercice 3 : Cardioïde

la fonction paramétrique d'un limaçon est donnée par :

$$r = r_0 + \cos(\theta)$$

$$x = r \cos(\theta)$$

$$y = r \sin(\theta)$$

Affichier cette fonction pour $r_0 = 0.8$, $r_0 = 1$ et $r_0 = 1.2$. Laquelle de ces courbes s'appelle un cardioïde.

Ajuster bien le nombre de points pour que ces courbes soient lisses.

In [] :