Cours1

March 4, 2020

0.1 Premier pas, l'interpréteur une super calculette!

0.1.1 Addition, soustraction

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

[97]: 4

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

[98]: -2

0.1.2 Multiplications

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

[26]: 50

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

[27]: 100000

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

[28]: 100000.0

0.1.3 Divisions

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

[29]: 3.3333333333333335

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

[30]: 3

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

[31]: 1

0.1.4 Comparaisons

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

[32]: True

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

[33]: False

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

[34]: False

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

[35]: True

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

[36]: False

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

[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 ==

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

[39]:
```
# Vérification
age
```

[39]: 20

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

[40]:
```
# Réaffactation
age = 40
```

[41]:
```
# Vérification
age
```

[41]: 40

0.3 Opérateurs pratiques sur les variables

0.3.1 Permutation

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

[43]:
```
a
```

[43]: 10

[44]:
```
b
```

[44]: 5

0.3.2 Incrémentations

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

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

[60]: 2

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

[61]: 3

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

[62]: 2

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

[63]: 20

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

[64]: 10.0

0.4 Les types de variables

0.4.1 Les nombres entiers (`int`)

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

0.4.2 Les nombres réels (`float`)

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

0.4.3 Les chaînes de caractères (`str`)

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

0.4.4 Les bouléens (`bool`)

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

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

[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 connaitre 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!

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

```
Il fait beau
```

[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

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

[73]:
```
int
```

0.5.3 La fonction `help(nom_de_la_fonction)`

affiche l'aide (en anglais) de la fonction!

Avec jupyter-notebook, nous pouvez aussi utiliser ?

[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.
```

[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)`

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

```
10
```

[84]:
```
int
```

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

```
1000.5
```

[85]:
```
str
```

0.5.6 La fonction `input()`

`Demande à l'utilisateur d'entrer une valeur`

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

```
Bonjour à tous.
```

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

```
Bonjour à tous.
```

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

```
10
```

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

[89]:
```
str
```

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

```
10
```

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

[91]:
```
int
```

[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

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

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

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

[94]:
```
2.8198157342681518
```

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

[95]:
```
22026.465794806718
```

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

[96]:
```
0.2013579207903308
```

Attention pour les fonctions trigonométriques, il faut utiliser les radians

TD2

March 4, 2020

1 1. Vrai ou faux en python (`True` et `False`)

Python affecte une valeur vraie ou fausse à une affirmation, par exemple

[1]:
```
print(5>1)
```

```
True
```

[2]:
```
print(5<1)
```

```
False
```

On peut combiner les affirmations avec des “et” et des “ou”, comme on le fait dans le langage courant : - l'affirmation « A et B » est vraie uniquement si A et B sont tous deux vraies. - L'affirmation « A ou B » est vraie si au moins une des deux affirmations A ou B est vraie.

Par exemple la commande suivante teste la valeur de vérité de l'affirmation “5 est supérieur à 1 et 5 est inférieur à 10”.

[3]:
```
print(5>1 and 5<10)
```

```
True
```

la commande suivante teste la valeur de vérité de l'affirmation “5 est supérieur à 1 et 5 est inférieur à 2”.

[4]:
```
print(5>1 and 5<2)
```

```
False
```

1.1 Exercice 1.1

Avant d'exécuter le programme suivant, devinez si les résultats sont `True` ou `False`.

[ ]:
```
print(5>1 or 5<10)
print(5>1 or 5<2)
print(5<1 or 5<2)
print((5<1 or 5<10) and 5>3)
```

1.2 Test d'une égalité

Si on veut tester qu'un nombre est égal à un autre, il faut utiliser l'opérateur `==` et non `=`, qui est réservé à l'affectation d'une variable. Par exemple,

[11]:
```
print(5==5)
print(5==3)
```

```
True
False
```

L'exemple suivant illustre la différence entre `=` et `==`. Le premier `print` affiche la valeur de la variable `a`, tandis que les deux suivants affichent la valeur des tests `a==2` et `a==5`.

[12]:
```
a = 5
print(a)
print(a==2)
print(a==5)
```

```
5
False
True
```

1.3 Exercice 1.2

Dans l'exemple suivant, remplacer `==` par `=`, exécuter et commenter le résultat.

[14]:
```
a=5
print(a==2)
```

```
False
```

1.4 Variables booléennes

On peut affecter une variable à un test, cette variable ne peut prendre que la valeur `True` ou `False`. On l'appelle une variable booléenne. Par exemple

[6]:
```
macondition = (5>1 or 5<10)
print(macondition)
```

```
True
```

1.5 Exercice 1.3

Le programme suivant défini deux variables booléennes `vrai` et `faux`. Avant de l'éxécuter, deviner si les résultats des opérations proposées sont True ou False.

[ ]:
```
vrai = True
faux = False
print("vrai et vrai : ", vrai and vrai)
print("vrai et faux : ", vrai and faux)
print("faux et faux : ", faux and faux)
print("vrai ou vrai : ", vrai or vrai)
print("vrai ou faux : ", vrai or faux)
print("faux ou faux : ", faux or faux)
```

2 3. Les structures conditionnelles (`if`, `elif`, `else`)

Les structures conditionnelles permettent d'effectuer des opérations lorsqu'une ou plusieurs conditions sont remplies, c'est-à-dire lorsqu'un test du type précédent est `True`.

2.1 Structure simple (`if`):

Le principe est de dire que 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`:

[ ]:
```
note = 10

if (note > 10 ) :
    print("J'ai mon UE !")

print("Ma note est : ", note)
```

2.2 Exercice 2.1

Écrire un programme qui écrit un message "note incorrecte" si la variable note n'est pas comprise entre 0 et 20.

[ ]:
```
note = 22
```

2.3 Exercice 2.2

Écrire un programme qui écrit “ce nombre est pair” si la variable note est un nombre pair.

[ ]:
```
note = 12
```

2.4 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`:

[ ]:
```
note = 9

print ("Ma note est :",note)

if (note > 10 ) :
    print("J'ai mon UE !")
else :
    print("Je dois repasser en session 2.")
```

2.5 Exercice 3.1

Écrire un programme qui écrit “ce nombre est pair” si la variable note est un nombre pair et qui écrit “ce nombre est impair” dans le cas contraire

[ ]:
```
note = 17
```

2.6 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, exécutez-le plusieurs fois en modifiant la valeur de la note pour voir le comportement du `if…elif…else`:

[ ]:
```
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")
```

2.7 Conditions composées (`and`, `or`)

Il est possible de tester des conditions plus complexes, en combinant des conditions comme nous l'avons vu au début. Grâce à la commande `and`, on peut tester si deux conditions sont vérifiées en même temps. Voici un exemple :

[ ]:
```
note = 13
if (note >= 12) and (note < 14) :
    print("J'ai la mention assez bien.")
```

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

[ ]:
```
note = 9
if (note < 12) or (note >= 14) :
    print("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`, vue plus haut.

[15]:
```
note = 9
condition = (note < 12) or (note >= 14)
if  condition :
    print("Je n'ai pas la mention assez bien.")
```

```
Je n'ai pas la mention assez bien.
```

2.8 Exercice 3.2 : mention à un examen

Écrire un programme qui écrit la mention associée à la note définie à la première ligne, dans la variable `note` (assez bien pour $[12, 14[$, bien pour $[14, 16[$, très bien pour $[16, 20]$)

[ ]:
```
note = 17
```

2.9 Exercice 3.3 : discriminant d'un polynôme de degré 2

Écrire un programme qui calcule le discriminant `delta` d'un polynôme $ax^2 + bx + c$ et qui écrit "les solutions sont réelles" lorsque `delta` est positif ou nul, ou "les solutions sont complexes" dans le cas contraire.

On ne cherchera pas ici à calculer les solutions (c'est l'exercice suivant).

[ ]:
```
a = 1
b = 2
c = 3

delta =
```

2.10 Exercice 3.4 : racines d'un polynôme de degré 2

Reprendre le programme écrit à l'exercice 3.3, en calculant puis en affichant les racines du polynôme dans le cas où elles sont réelles. On rappelle que les deux racines sont alors données par :

$$x_1 = \frac{-b - \sqrt{\Delta}}{2a}$$

et

$$x_2 = \frac{-b + \sqrt{\Delta}}{2a}$$

[ ]:
```
a = 1
b = 2
c = 3
```

2.11 Exercice 3.5 : racines d'un polynôme de degré 2

Reprendre le programme précédent, en calculant puis en affichant les racines du polynôme dans tous les cas. Pour vous aider, voici un schéma logique du programme.

[ ]:
```

```

2.12 Exercice 3.6 : année bissextile

Vérifier si une année est bissextile.

On rappelle qu'une année est bissextile si l'une des deux conditions suivantes est vérifiée : - elle est un multiple de 4 mais pas de 100 - elle est un multiple de 400

[ ]:

2.13 Exercice 3.7 : franchise d'assurance auto

Votre assurance auto vous rembourse 10% des frais de réparation suite à un accident à condition que ce remboursement soit supérieur de 150 euros (la franchise). Par ailleurs, votre contrat limite le remboursement à 1000 euros (si le remboursement calculé est plus grand, vous ne touchez que 1000 euros). Écrivez un programme qui affiche le montant du remboursement en fonction du montant des travaux (variable `montant`*). Modifiez la valeur de* `montant` *pour tester que le programme fonctionne comme attendu.*

[ ]:
```
montant = 2000
```

2.14 Exercice 3.8 : jour de la semaine

À partir d'une date quelconque de l'année 2019 (ex. 15/09), calculer le nombre de jours qui se sont écoulés depuis le début de l'année (1/01).

[ ]:

2.15 4. Partie facultative

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

2.15.1 Exercice 4 (facultatif)

Reprendre l'exercice 3.5 (racines d'un polynôme de second degré) avec les complexes, mais sans `if`

[ ]:

TD3

March 4, 2020

1 Les boucles

Les boucles permettent de répéter plusieurs fois un ensemble d'opérations, de façon légèrement différente. C'est ce qui fait la puissance des programmes informatiques : ils ne se lassent pas de répéter une opération un million de fois.

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

[2]:
```
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)
```

```
1^2 =  1
2^2 =  4
3^2 =  9
4^2 =  16
5^2 =  25
6^2 =  36
7^2 =  49
8^2 =  64
9^2 =  81
10^2 =  100
```

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

Si on regarde bien, on réalise en fait plusieurs fois la même opération en modifiant un paramètre. 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 (au sens vu dans le TD précédent) 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.
- Un des instructions au moins doit donc avoir un effet sur la condition et la faire passer à `False`
- Les conditions peuvent être multiples.
- Il est possible de répéter le bloc sans savoir à l'avance combien de fois on va le répéter.

Programmons un simple compteur affichant les nombres de 0 à 9 :

[3]:
```
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'avancée 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'incrémentation `nb = nb -1`

1.1.1 Exercice

Sans l'exécuter, indiquer ce qu'affichera le programme suivant, dans lequel on a inversé les deux dernières lignes du programme précédent. Dans un second temps, exécuter le programme pour vérifier.

[ ]:
```
nb = 0

while nb < 10 :
    nb = nb + 1
    print(nb)
```

1.1.2 Exercice : Erreur courante

Une erreur que l'on commet souvent en débutant (voire plus tard) est d'oublier d'indenter ou de mal indenter. Exécuter le programme suivant, fortement inspiré du précédent et expliquer ce que vous obtenez en sortie.

[2]:
```
nb = 0

while nb < 10 :
    nb = nb + 1
print(nb)
```

```
10
```

1.1.3 Exercice 1.0

Sans l'exécuter, examiner le programme suivant et écrire de la façon la plus précise possible, sur un papier, ce qui sera affiché lorsqu'il sera exécuté.

Dans un second temps, exécuter le programme pour vérifier votre prédiction

[ ]:
```
nb = 0

while nb < 100 :
    print(nb)
    nb = nb**2 + 1
```

1.1.4 Exercice 1.1 :

1) Écrire un programme qui écrit les carrés des 10 premiers entiers, avec une boucle `while`. L'éxécution du programme doit donner

   1^2 = 1 2^2 = 4 3^2 = 9 4^2 = 16 5^2 = 25 6^2 = 36 7^2 = 49 8^2 = 64 9^2 = 81 10^2 = 100

On pourra s'inspirer fortement du programme donné en exemple pour introduire la boucle `while`.

[ ]:
```

```

1.1.5 Exercice 1.2

Ajouter la possibilité de choisir facilement le nombre de valeurs à afficher (de 1 à 100, de 1 à 1000…).

[ ]:
```
```

1.1.6 Exercice 1.3

Adapter ensuite ce code pour que l'on puisse facilement changer l'exposant (écrire les cubes ou les puissances quatrièmes des entiers successifs)

[ ]:
```
```

1.1.7 La fonction `input()`

Dans la suite, l'utilisateur va devoir entrer un nombre. Ceci est réalisé par la commande `input`.

[ ]:
```
choix = input("Entrer un nombre : ")
print("Vous avez choisi : ", choix)
```

Attention, elle renvoie la chaîne de caractères entrés par l'utilisateur (réessayez le code précédent en entrant un mot plutôt qu'un nombre) et si on veut accéder à une valeur numérique, il faut la convertir en un type numérique, un `int` ou un `float`, selon l'usage qu'on veut en faire.

[ ]:
```
choix = input("Entrer un nombre : ")
print("Vous avez choisi : ", float(choix))
```

1.1.8 Exemples de boucle `while` qui ne connait 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.

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

L'exemple suivant montre une boucle potentiellement éternelle, la condition n'étant jamais fausse. Pour sortir de la boucle, il est alors possible d'utiliser la commande `break`.

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

1.2 La boucle `for`

L'autre structure permettant de faire des opérations répétitives est 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 crochets `[]` et de séparer les éléments avec une virgule `,`. 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 :

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

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

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 Exercice 1.4

Créer une liste contenant les entiers de 0 à 9 et créer une boucle qui les affiche un par un

[ ]:
```

```

1.2.3 Exercice 1.5

Écrire un programme qui écrit les carrés des 10 premiers entiers, avec une boucle `for`. L'éxécution du programme doit donner

```
1^2 =  1
2^2 =  4
3^2 =  9
4^2 =  16
5^2 =  25
6^2 =  36
7^2 =  49
8^2 =  64
9^2 =  81
10^2 =  100
```

On pourra s'inspirer fortement du programme précédent.

[ ]:

1.2.4 Les listes - 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 :

[ ]:
```
phrase = """Il fait 30°C."""

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

1.2.5 Les listes - Suites d'entier : fonction `range()`

Nous avons eu besoin plus haut d'une suite d'entiers successifs, par exemple $0, 1, 2, 3, 4....$ Nous l'avons fait de façon manuelle :

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

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

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

Mais si la liste devient très grande, ce n'est pas bien pratique. Python fournit la fonction `range()` qui permet de la générer automatiquement. 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 :

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

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

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

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

1.2.6 Exercice 1.6 : Reprenons l'exemple initial :

1) Reprendre l'exemple du début et remplacer ces 10 lignes par une boucle `for` équivalente.

[ ]:
```
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.

1.3 Accumulateurs

Nous présentons ici une technique extrêmement utile et courante. Pour calculer la somme des entiers de 1 à 10, on peut utiliser une boucle `for`, en créant une variable `somme` initialisée à 0, à laquelle on ajoute chacun des éléments de la liste grâce à la boucle. À la fin, la variable `somme` contient le résultat cherché.

[2]:
```
liste = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
somme = 0
for nombre in liste:
    somme = somme + nombre
print(somme)
```

```
55
```

1.3.1 Exercice 1.7

Calculer la somme des carrés des nombres compris entre 1 et 10 ($1^2 + 2^2 + 3^3 + 4^2 + 5^5 + 6^2 + 7^2 + 8^2 + 9^2 + 10^2$), avec une boucle `for`, sur le modèle précédent.

[ ]:
```
liste = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

2 Exercices d'application

2.0.1 Exercice 2.1 : 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. Pour votre confort, la liste est déjà fournie.

[1]:
```
notes = [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]
```

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.

[ ]:
```

```

2.0.2 Exercice 2.2 : Suite de Fibonacci

La suite de Fibonacci peut être considérée comme le tout premier modèle mathématique en dynamique des populations ! Elle 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< /*div* >< *divalign* = "*center*" > F_1 = 1< /*div* >< *divalign* = "*center*" > F_n = F_{n-1} + F_{n-2}$

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

[ ]:
```

```

2.0.3 Exercice 2.3 : Nombres premiers

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

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

[ ]:

TD4

March 4, 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 uselles `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 :

[1]:
```
def cube(x) :
    print(x*x*x)
```

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

[2]:
```
cube(4)
cube(10.1)
```

```
64
1030.301
```

1.1.1 Exercice

Définir une fonction (avec le nom que vous voulez) qui prend deux arguments `x` et `y` et qui écrit la somme de ses deux arguments. Tester son fonctionnement en l'appelant

[ ]:

1.1.2 Exercice

Sans l'exécuter, analyser le programme suivant pour indiquer, sur une feuille, ce qu'il affichera quand on l'exécutera

[3]:
```
def mafonction(x, y):
    print(x, y, x*y)

mafonction(2, 3)
mafonction(0, 1)
```

```
2 3 6
0 1 0
```

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 defaut.

[4]:
```
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 :

[5]:
```
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 un arguments `nombre`. Elle affiche la table de multiplication associée à `nombre` de 0 jusqu'à 5 :

[6]:
```
def TableMultiplication(nombre=2):
    for i in (0, 1, 2, 3, 4, 5):
        print(i,"*",nombre,"=",i*nombre)
```

[7]:
```
TableMultiplication(3)
```

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

Voici une version un peu plus élaborée. Cette fonction prend deux arguments `nombre` et `max`. Elle affiche la table de multiplication associée à `nombre` de 0 jusqu'à `max` :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` :

[8]:
```
def TableMultiplication(nombre=2, max=10):
    for i in range(0,max+1):
        print(i,"*",nombre,"=",i*nombre)
```

[9]:
```
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
```

[10]:
```
TableMultiplication(5) # on demande la table de 5. On ne spéccifie 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
```

[11]:
```
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
```

[12]:
```
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 ce 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 :

[13]:
```
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
```

Le corps de la fonction peut contenir des calculs, des tests, des boucles. Pour commencer par un exemple simple, voici une fonction qui renvoie la chaîne de caractère "pair" si le nombre est pair et "impair" sinon :

[14]:
```
def parite(i) :
    if (i%2==0):
        return "pair"
    else:
        return "impair"

nombre = 27
resultat = parite(nombre)
print("Le nombre", nombre,"est", resultat)
```

```
Le nombre 27 est impair
```

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. La fonction définie dans

le programme suivant affiche un nombre et les deux suivants.

[15]:
```
def deux_suivants(i) :
    return i, i+1, i+2

print(deux_suivants(27))
```

```
(27, 28, 29)
```

1.3.1 Exercice

Sans l'exécuter, analyser le programme suivant pour indiquer, sur une feuille, ce qu'il affichera quand on l'exécutera. Résumer en une phrase le rôle de cette fonction.

[16]:
```
def mafonction(x,y) :
    a,b =  x,y
    if a > b :
        a,b = b,a
    return a,b

mafonction(22,2)
```

[16]:
```
(2, 22)
```

1.4 Les librairies de fonctions

Nous avons déjà chargé une librairie de fonctions : *numpy*. Pour cela nous avions 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. Il est préférable d'appeler une librairie de la façon suivante

```
import numpy
```

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

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

C'est un peu fastifieux et la plupart des utilisateurs de python préfèrent utiliser un raccourci sous la forme :

```
import numpy as np
```

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

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

Ceci est un peu plus lourd que d'utiliser simplement `sqrt(12)`, mais maintenant lorsque nous appellons 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.

[17]:
```
import numpy as np
import math as mt

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

```
3.4641016151377544
3.4641016151377544
```

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

[18]:
```
nb = 10+10j
print(np.sqrt(nb))
print(mt.sqrt(nb))
```

```
(3.4743442276011565+1.4391204994250741j)
```

```
␣
↪---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-18-14805480af57> in <module>
  1 nb = 10+10j
  2 print(np.sqrt(nb))
----> 3 print(mt.sqrt(nb))


TypeError: can't convert complex to float
```

1.5 Exercice 1 : Arrondir un nombre

1) Ecrire une fonction qui tronque un nombre à n chiffres après la virgule.
2) Tester avec plusieurs exemples, si cette fonction fonctionne correctement.

3) Adapter ensuite la fonction pour qu'elle arrondisse correctement le nombre. Pensez au cas où le nombre est négatif.

[ ]:

1.6 Exercice 1 bis : calcul d'une somme

Ecrire une fonction `somme` qui, lorsqu'on l'appelle sous la forme `somme(n)` calcule la somme des entiers de 1 à n. Tester la fonction avec `somme(9)` qui vaut 45.

[ ]:

1.7 Problème 1 : Racine d'une fonction mathématique par dichotomie

Dans cet exercice, nous allons utiliser tout ce que nous avons appris jusqu'ici et apprendre un algorithme très utilisé en informatique: ***la dichotomie****. Nous allons le faire avec un exemple.*

Nous souhaitons trouver numériquement les racines du polynôme suivant :

$f(x) = x^3 + 3.6821627548\,x^2 - 3,7208387236\,x - 9,7979589711$

Nous voyons sur ce graphique qu'une racine est présente entre 0 et 3. Cela se voit car $f(0)$ est négatif et $f(3)$ est positif. Il existe donc une valeur entre les deux pour laquelle $f(x) = 0$.

On coupe alors l'intervalle en deux:

- si $f(1.5) > 0$ cela signifie que la racine est avant 1.5
- si $f(1.5) < 0$ cela signifie que la racine est après 1.5. C'est le cas dans notre exemple.

Nous savons donc maintenant que la racine appartient à $[1.5, 3]$. L'intervalle a été divisé par deux. Nous nous approchons de la racine.

Nous allons donc utiliser cela pour chercher de façon itérative (*i.e. qui est répétée plusieurs fois*) la position exacte de la racine. A chaque itération, nous allons diviser par deux l'intervalle.

Ici on part d'un intervalle de longueur 3. Après une itération, il ne fera plus que 1.5, puis 0.75… Après n itérations, il ne fera donc plus que $3/2^n$. Pour se rendre compte, au bout de 30 itérations l'intervalle fera 0.000000002793968. Nous serons donc très proche de la racine.

A vous d'écrire cet algorithme. On pourra tout d'abord écrire une fonction qui pour un x donné retourne $f(x)$. On peut ensuite écrire un code qui effectue la première itération, c'est à dire, ce qui est décrit au dessus. Il restera alors à répéter correctement la procédure n fois à l'aide d'une boucle.

[ ]:

TD5

March 4, 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 créer 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.

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

```
<class 'list'>
[]
```

1.2 Accèder à / modifier un élément de la liste

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

Par exemple :

[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'élement que l'on veut modifier :

[45]:
```
maliste[2]='toto'
print(maliste)
```

```
['a', 'b', 'toto', 'd', 'e', 'f', 'g']
```

Il est également possible d'extraire plusieurs éléments concécutifs. 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)

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

[47]:
```
['b', 'toto']
```

1.3 Ajouter un élément dans une liste

1.3.1 Méthodes et fonctions

*Nous avons vu précédemment plusieurs exemples de **fonctions**, qui peuvent avoir des arguments mais ne les modifient pas. Ici nous allons voir un nouveau concept, les **méthodes**, qui agissent sur un objet et le transforment. Nous allons le montrer avec un objet de type `list`. La syntaxe est différente de celle des fonctions.*

1.3.2 Exemple : la méthode `append()` pour ajouter un élément à la fin d'une liste

Pour utiliser une méthode, on indique le nom de la variable suivie d'un `.` puis le nom de la méthode avec ses arguments.

Par exemple, pour ajouter un élément à la fin d'une liste, on utilise la **méthode `append()` de la classe `list`**. Elle prend en argument la valeur à ajouter :

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

Voici un exemple, où l'on voit que la liste a été transformée.

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

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

1.3.3 Insertion d'un élément : `insert()`

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

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

Regardons ce que cela donne :

[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 élements d'indice supérieur à n, pour intercaler la valeur supplémentaire.

1.3.4 Exercice

Insérer le nombre qui semble manquer dans la liste suivante, puis afficher la liste complétée

```
 [2, 4, 6, 8, 12, 14, 16]
```

[3]:
```
maliste = [2, 4, 6, 8, 12, 14, 16]
```

1.3.5 Supprimer un élement : `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 :

[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']
```

1.3.6 Exercice

Déterminer ce que fait le programme suivant, sans l'exécuter, puis vérifier en l'exécutant

[ ]:
```
maliste = ['Maxwell', 'Einstein', 'Kepler', 'Galilée']
maliste.append('Curie')
maliste.remove('Maxwell')
maliste.insert(1, 'Meitner')
print(maliste)
```

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

1.3.7 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 :

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

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

1.3.8 Nombre d'éléments dans une liste : `len()`

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

```
len(maliste)
```

Exemple :

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

[16]:
```
3
```

1.4 Parcourir une liste :

1.4.1 Avec une 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 :

[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.

1.4.2 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 :

[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.

1.4.3 Avec une boucle `for` et la fonction `enumerate()` :

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

[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 élements de la liste pendant que la variable `i` va servir de compteur et prendre ainsi le numéro de l'indice courant. Techniquement, la fonction `enumerate()` renvoie une succession de couples de valeurs, le premier élément du couple est le compteur, le second l'élément de la liste.

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

1.5 Exercice 1 : Échange de valeurs

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

Rappel pour échanger les valeurs contenues dans `a` et `b` :

```
a = 10
b = 12
a,b = b,a
```

[8]:
```
liste = ['Meitner', 'Maxwell', 'Curie', 'Einstein', 'Kepler', 'Galilée']
```

1.6 Exercice 1 bis : Recherche de petits nombres

Ecrire un programme qui à partir d'une liste, par exemple (1, 13, 33, 2, 4, 40), fabrique une nouvelle liste ne contenant que les nombres supérieurs à 10.

Algorithme : - Parcourir et afficher un à un les termes de la liste - Afficher seulement les termes $<$ 10 - Placer ces termes dans une nouvelle liste puis l'afficher

1.7 Exercice 2 : Liste symétrique

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

Algorithme : - Parcourir et afficher les termes de la liste - Inverser le sens de parcours de la liste (avec la fonction `reversed`) - Créer la liste symétrique (à l'nevers) - Vérifier si la liste et la liste symétrique sont identiques avec un test

[ ]:
```

```

1.8 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]

Algorithme : - Parcourir et afficher les termes de la liste - Utiliser le modulo (`%`) pour rammener l'angle antre 0 et 360 - Tester (avec un `if`) si l'angle est supérieur à 180, dans ce cas retrancher 360 - Placer le résulat dans la liste à la place de l'angle initial

[ ]:

1.9 Exercice 4 : Trier une liste

Écrire un programme qui trie du plus petit au plus grand une liste composée de nombres quelconques. *(attention aux petits malins, on vous demande de créer votre propre programme et non pas d'utiliser une fonction toute faite).*

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

Algorithme : - Parcourir et afficher les termes de la liste - Lors du parcours, identifier le nombre le plus petit en faisant un test (`if`) - Ajouter ce minimum dans une nouvelle liste, et effacer ce dernier de la première liste - Réitérer ce processus à l'aide d'une boucle

[ ]:

Reprenons le cours ici.

1.10 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 :

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

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

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

```
['Fer', 26]
['Al', 13]
```

Mais il est possible de récupérer directement l'élement 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 :

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

```
26
L'élément Fer a 26 protons.
```

1.11 Les compréhensions de liste

On appelle **compréhension de liste** la technique suivante qui permet de modifier ou de filtrer une liste très facilement.

1.11.1 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)

[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 :

[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 élements 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 `=`

1.12 Filtres sur une liste

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

[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 élements de la liste `nombres` en ne considérant que les nombres positifs.
- pour chaque élement retenu, on retourne sa valeur pour former une nouvelle liste `nombres_positifs`.

1.13 Fonctions sur les listes :

1.13.1 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 :

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

print(sum(nombres)) # fait la somme des élements 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
```

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

[11]:
```
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
```

[11]:
```
['ananas', 'bonjour', 'girafe', 'schtroumpf', 'tagada', 'zoo']
```

Attention, `sorted(mots)` est une fonction (pas une méthode) et `mots` est son argument. La liste `mots` n'est pas modifiée par le programme précédent :

[12]:
```
print(mots)
```

```
['bonjour', 'girafe', 'schtroumpf', 'tagada', 'ananas', 'zoo']
```

1.13.2 Fonctions provenant de librairies

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

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

[65]:
```
4
```

1.13.3 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 graphe représentant des données.

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

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

Si sous linux, cette librairie n'est pas installée, alors 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 :

[15]:
```
for i in range(0,11):
    x = i
    y = i*i
```

ou encore, dans un style différent mais avec le même résultat,

[ ]:
```
x = range(0,11)
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 (nous introduirons dans un TD ultérieur la notion de tableau, qui permet ce genre de syntaxe, mais patience).

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

```
␣
↪---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)
```

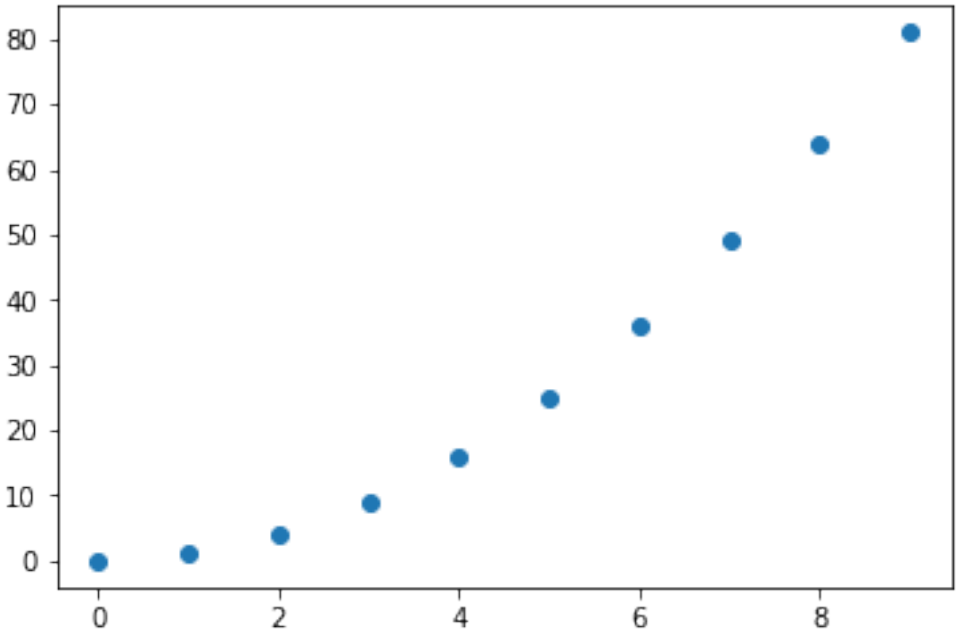
```
    <ipython-input-13-3c3ad855defc> in <module>
      1 x = range(0,10)
----> 2 y = x**2


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

Reprenons nos listes correctement construites.

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

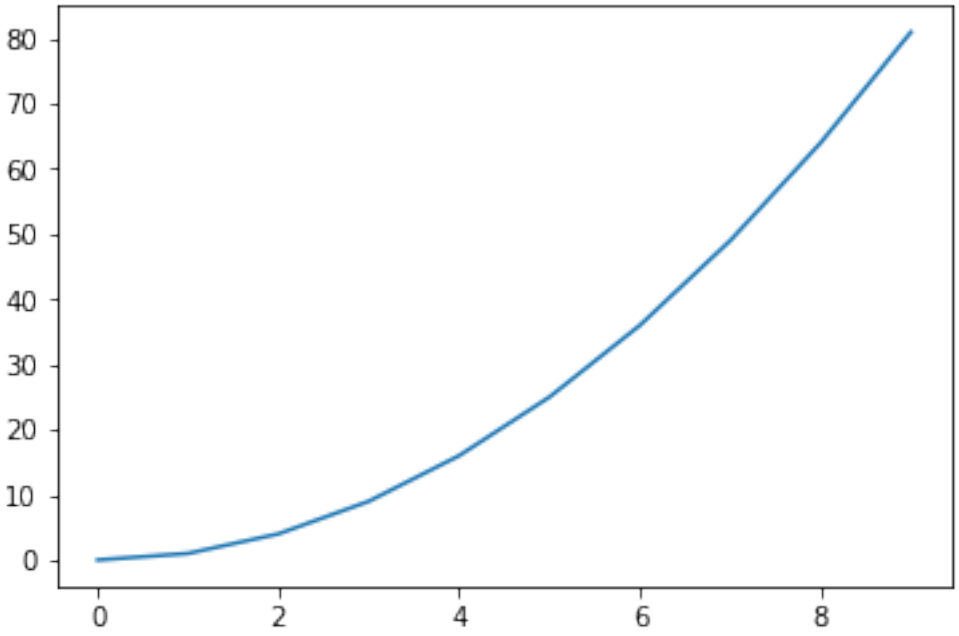
On peut alors afficher y en fonction de `x` à l'aide d'un nuage de points :

[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 reliant les points avec `plot` à la place de `scatter`. Les points sont alors reliés par des segments.

[29]:
```
plt.plot(x,y) # on génère une courbe reliant les points
plt.show() # on affiche le graphique
```



1.14 Exercice 5 :

Produire la courbe représentative de la fonction $f(x) = \log(x)$ entre 0 et 30.

Algorithme : - Créer une liste `x` allant de 0 à 30 à l'aide de la fonction `range` - Parcourir cette liste et calculer $\log(x)$ - Ajouter ce résultat dans une liste `y` - Utiliser la fonction `scatter` ou `plot` de matplotlib pour afficher les points (`y` vs `x`)

[ ]:

1.15 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. Chaque liste contient les trois coordonnées du vecteur. On rappelle que le produit vectoriel est donné par :

[ ]:

Tester ensuite votre fonction avec $u = (1, 0, 0)$ et $u = (0, 1, 0)$

[ ]:

1.16 Exercice 7 : Tableau périodique

Chaque élément de la liste suivante est une liste à deux éléments, contenant le symbole et le numéro atomique Z d'une espèce atomique.

Atomes = [[“Fer”,26],[“Ag”,47],[“Ca”,20],[“Al”,13],[“Ne”,10],[“O”,8],]

Ajouter à cette liste l'or de symbole “Au” et Z=79 puis afficher la liste complétée.

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

Pour trier par ordre alphabétique du nom d'élément, on peut utiliser la fonction `sorted`

[55]:
```
sorted(Atomes)
```

[55]:
```
[['Ag', 47],
 ['Al', 13],
 ['Au', 79],
 ['Ca', 20],
 ['Fer', 26],
 ['Ne', 10],
 ['O', 8]]
```

Écrire un programme qui trie les élements chimiques précédents par ordre croissant de numéro atomique.

Algorithme : - Écrire un programme qui crée une liste dans laquelle numéro et nom sont inversés :

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

- Trier cette nouvelle liste avec la commande `sorted`

[61]:
```
Atomes = [["Fer",26],["Ag",47],["Ca",20],["Al",13],["Ne",10],["O",8],["Au",79]]
```

1.17 Exercice 7bis

Chaque élément de la liste suivante est une liste à quatre éléments donnant le prénom, le nom, le prénom, la date de naissance et la date de décès de différentes personnalités du monde de la physique.

```
liste=(('James Clerk', 'Maxwell', 1831, 1879), ('Albert', 'Einstein', 1879, 1955), ('Isaac', 'Newt
```

Ajouter un élément à la liste pour Lise Meitner (1878-1968) et afficher la liste complète.

[1]:
```
liste=[['James Clerk', 'Maxwell', 1831, 1879], ['Albert', 'Einstein', 1879,␣
↪1955], ['Isaac', 'Newton', 1643, 1727]]
```

Écrire un programme affichant la liste des noms par ordre de date de naissance croissante, sous la forme

```
Isaac Newton de 1878 à 1968
James Clerk Maxwell de 1831 à 1879
```

`(etc...)`

[ ]:

1.18 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

Algorithme : - Créer une liste `entiers` d'entiers de 0 à 100 (`range`) - Parcourir cette liste et créer une seconde liste `x` de nombres entre 0 et 10 par pas de 0.1 - Parcourir cette nouvelle liste et tester si la partie entière de chaque élément est paire - Utiliser cette condition pour remplir la liste `y` contenant 1 si la condition est réalisée et 0 sinon - Afficher la courbe (`y` vs `x`) obtenue

[ ]:

TD6

March 4, 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 *binaire*. Nous y reviendrons plus tard.

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

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

[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`.

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

[60]:
```
str
```

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

[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`.

[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.

[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()` :

[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.")
```

[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.

[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 plius 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 :

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

[77]:
```
'  mon texte  '
```

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

[78]:
```
'  MON TEXTE  '
```

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

[79]:
```
'  mon texte  '
```

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

[80]:
```
'mon TEXTE'
```

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

[87]:
```
'T'
```

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

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

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

[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 :

[72]:
```
```

```
texte = "Je m'appelle {prenom} et j'ai {age} ans." # deux labels {prenom} et␣
 ↪{age} sont spécifiés
print(texte) # on peut afficher la chaîne précédente.
```

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

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

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

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

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

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

Si l'on souhaite modifié la variable `texte` de façon définitive, on peut écrire :

[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 replacent 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 rrvi.e”*

1) Ouvrir le fichier et afficher le texte qu'il contient

2) Décoder le code

[ ]:
```

```

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`.

[5]:
```
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 bianaire.

Le fichier ne sera donc pas lisible par un humain, moais l'ordinateur pour y mettre des informations supplémaires 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.

[6]:
```
import numpy as np


fichier = open("fichiers/data.bin","wb")

x = []
y = []

for i in range(20):
    x.append(i*0.1)
    y.append(np.sin(i*0.1))

pickle.dump(x,fichier)
pickle.dump(y,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.

[7]:
```
fichier = open("fichiers/data.bin","rb")

x = pickle.load(fichier)
y = pickle.load(fichier)

print(x)
print(y)
```

```
[0.0, 0.1, 0.2, 0.30000000000000004, 0.4, 0.5, 0.6000000000000001,
0.7000000000000001, 0.8, 0.9, 1.0, 1.1, 1.2000000000000002, 1.3,
1.4000000000000001, 1.5, 1.6, 1.7000000000000002, 1.8, 1.9000000000000001]
[0.0, 0.09983341664682815, 0.19866933079506122, 0.2955202066613396,
0.3894183423086505, 0.479425538604203, 0.5646424733950355, 0.6442176872376911,
```

```
0.7173560908995228, 0.7833269096274833, 0.8414709848078965, 0.8912073600614354,
0.9320390859672264, 0.963558185417193, 0.9854497299884603, 0.9974949866040544,
0.9995736030415051, 0.9916648104524686, 0.9738476308781951, 0.9463000876874145]
```

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:

[8]:
```
Toto = pickle.load(fichier)
```

```
␣
↪---------------------------------------------------------------------------

EOFError                                  Traceback (most recent call last)

<ipython-input-8-fb57e1700de5> in <module>
----> 1 Toto = pickle.load(fichier)


EOFError: Ran out of input
```

N'oubliez pas de fermer le fichier.

[9]:
```
fichier.close()
```

1.8 Exercice 2 : PIB par pays

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

1) Charger le fichier, récuperer 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 ?

[ ]:
```

```

1.9 Problème 2 : Chûte libre

Un avions lâche une caisse de matériel d'une altitude H et une vitesse initiale horizontale $\vec{v_0}$. Nous allons étudier la trajetoire de la caisse.

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

- $a_x = 0$

- $a_z = -g$
- $v_x = v_0$
- $v_z = -g\,t$
- $x = v_0\,t$
- $z = -1/2g\,t^2 + H$

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

1) Tracer la trajectoire jusqu'au sol, c'est à dire z en fonction de x.
2) Cette chûte 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 précedemment.
3) D'où provient la différence observée ?

[ ]:

TD7

March 4, 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 impoter la librairie *numpy*, il suffit d'utiliser la fonction `array()` pour convertir une liste `list` en `ndarray` :

[2]:
```
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 :

[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 biens 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`) :

[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`) :

[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` :

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

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

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

[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 :

[52]:
```
x = np.random.randint(low=10, high=30, size=6) # 6 nombres tirés aléatoirement␣
↪entre 10 et 30
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 :

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

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

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

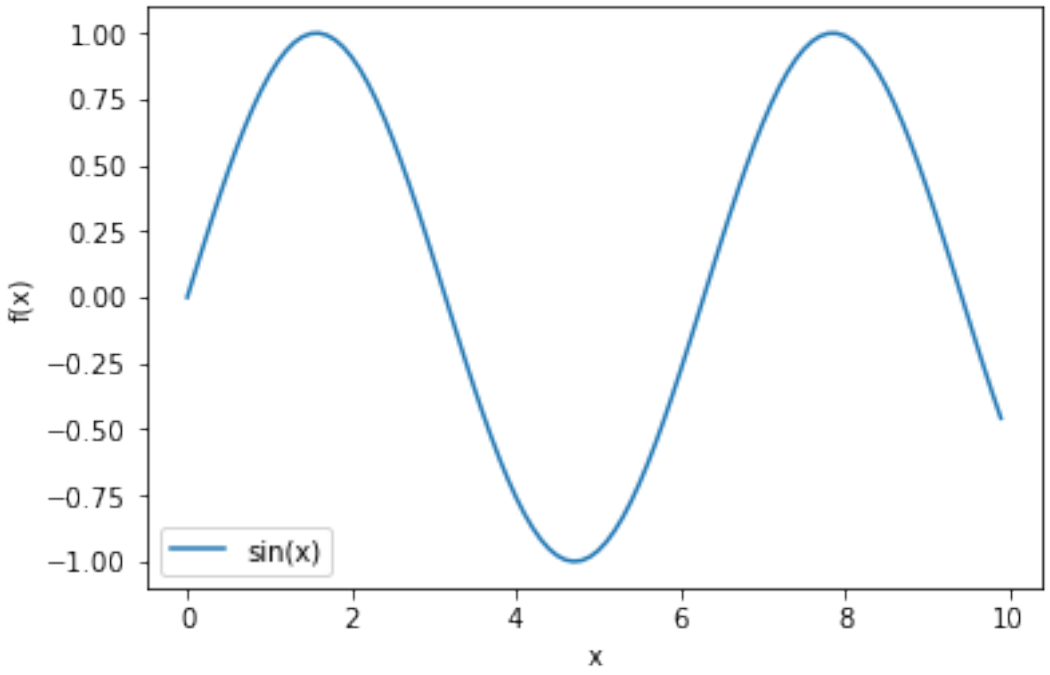
```
[ 0  3  6  9 12 15 18 21 24 27]
```

[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 :

[10]:
```
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 commes les autres :

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

[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 `nbarray`.

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 :

[12]:
```
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 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 :

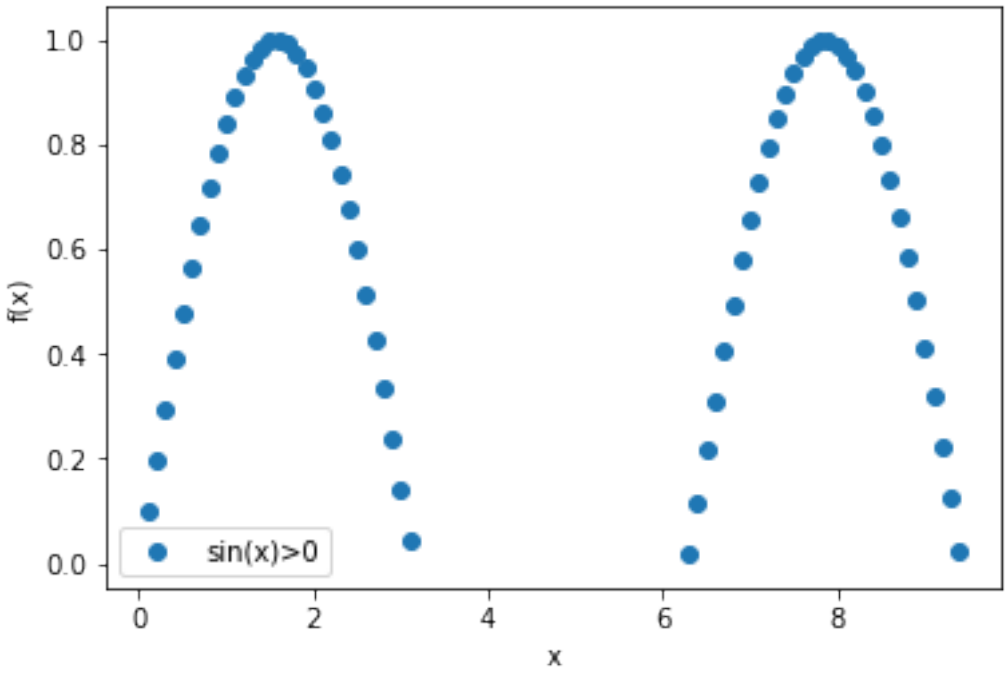
[13]:
```
print(y[z])
```

```
[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 :

[16]:
```
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 pairs par des -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).

[ ]:

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.

[ ]:

Cours1

March 4, 2020

0.1 Premier pas, l'interpréteur une super calculette!

0.1.1 Addition, soustraction

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

[97]: 4

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

[98]: -2

0.1.2 Multiplications

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

[26]: 50

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

[27]: 100000

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

[28]: 100000.0

0.1.3 Divisions

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

[29]: 3.3333333333333335

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

[30]: 3

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

[31]: 1

0.1.4 Comparaisons

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

[32]: True

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

[33]: False

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

[34]: False

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

[35]: True

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

[36]: False

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

[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 ==

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

[39]:
```
# Vérification
age
```

[39]: 20

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

[40]:
```
# Réaffactation
age = 40
```

[41]:
```
# Vérification
age
```

[41]: 40

0.3 Opérateurs pratiques sur les variables

0.3.1 Permutation

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

[43]:
```
a
```

[43]: 10

[44]:
```
b
```

[44]: 5

0.3.2 Incrémentations

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

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

[60]: 2

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

[61]: 3

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

[62]: 2

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

[63]: 20

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

[64]: 10.0

0.4 Les types de variables

0.4.1 Les nombres entiers (`int`)

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

0.4.2 Les nombres réels (`float`)

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

0.4.3 Les chaînes de caractères (`str`)

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

0.4.4 Les bouléens (`bool`)

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

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

[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 connaitre 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!`

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

```
Il fait beau
```

[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`

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

[73]:
```
int
```

0.5.3 La fonction `help(nom_de_la_fonction)`

`affiche l'aide (en anglais) de la fonction!`

Avec jupyter-notebook, nous pouvez aussi utiliser ?

[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.
```

[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)`

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

```
10
```

[84]:
```
int
```

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

```
1000.5
```

[85]:
```
str
```

0.5.6 La fonction `input()`

`Demande à l'utilisateur d'entrer une valeur`

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

```
Bonjour à tous.
```

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

```
Bonjour à tous.
```

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

```
10
```

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

[89]:
```
str
```

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

```
10
```

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

[91]:
```
int
```

[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

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

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

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

[94]:
```
2.8198157342681518
```

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

[95]:
```
22026.465794806718
```

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

[96]:
```
0.2013579207903308
```

Attention pour les fonctions trigonométriques, il faut utiliser les radians

TD2

March 4, 2020

1 1. Vrai ou faux en python (`True` et `False`)

Python affecte une valeur vraie ou fausse à une affirmation, par exemple

[1]:
```
print(5>1)
```

```
True
```

[2]:
```
print(5<1)
```

```
False
```

On peut combiner les affirmations avec des “et” et des “ou”, comme on le fait dans le langage courant : - l’affirmation « A et B » est vraie uniquement si A et B sont tous deux vraies. - L’affirmation « A ou B » est vraie si au moins une des deux affirmations A ou B est vraie.

Par exemple la commande suivante teste la valeur de vérité de l’affirmation “5 est supérieur à 1 et 5 est inférieur à 10”.

[3]:
```
print(5>1 and 5<10)
```

```
True
```

la commande suivante teste la valeur de vérité de l’affirmation “5 est supérieur à 1 et 5 est inférieur à 2”.

[4]:
```
print(5>1 and 5<2)
```

```
False
```

1.1 Exercice 1.1

Avant d’exécuter le programme suivant, devinez si les résultats sont `True` ou `False`.

[ ]:
```
print(5>1 or 5<10)
print(5>1 or 5<2)
print(5<1 or 5<2)
print((5<1 or 5<10) and 5>3)
```

1.2 Test d'une égalité

Si on veut tester qu'un nombre est égal à un autre, il faut utiliser l'opérateur `==` et non `=`, qui est réservé à l'affectation d'une variable. Par exemple,

[11]:
```
print(5==5)
print(5==3)
```

```
True
False
```

L'exemple suivant illustre la différence entre `=` et `==`. Le premier `print` affiche la valeur de la variable `a`, tandis que les deux suivants affichent la valeur des tests `a==2` et `a==5`.

[12]:
```
a = 5
print(a)
print(a==2)
print(a==5)
```

```
5
False
True
```

1.3 Exercice 1.2

Dans l'exemple suivant, remplacer `==` par `=`, exécuter et commenter le résultat.

[14]:
```
a=5
print(a==2)
```

```
False
```

1.4 Variables booléennes

On peut affecter une variable à un test, cette variable ne peut prendre que la valeur `True` ou `False`. On l'appelle une variable booléenne. Par exemple

[6]:
```
macondition = (5>1 or 5<10)
print(macondition)
```

```
True
```

1.5 Exercice 1.3

Le programme suivant défini deux variables booléennes `vrai` et `faux`. Avant de l'éxécuter, deviner si les résultats des opérations proposées sont True ou False.

[ ]:
```
vrai = True
faux = False
print("vrai et vrai : ", vrai and vrai)
print("vrai et faux : ", vrai and faux)
print("faux et faux : ", faux and faux)
print("vrai ou vrai : ", vrai or vrai)
print("vrai ou faux : ", vrai or faux)
print("faux ou faux : ", faux or faux)
```

2 3. Les structures conditionnelles (`if`, `elif`, `else`)

Les structures conditionnelles permettent d'effectuer des opérations lorsqu'une ou plusieurs conditions sont remplies, c'est-à-dire lorsqu'un test du type précédent est `True`.

2.1 Structure simple (`if`):

Le principe est de dire que 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`:

[ ]:
```
note = 10

if (note > 10 ) :
    print("J'ai mon UE !")

print("Ma note est : ", note)
```

2.2 Exercice 2.1

Écrire un programme qui écrit un message "note incorrecte" si la variable note n'est pas comprise entre 0 et 20.

[ ]:
```
note = 22
```

2.3 Exercice 2.2

Écrire un programme qui écrit “ce nombre est pair” si la variable note est un nombre pair.

[ ]:
```
note = 12
```

2.4 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`:

[ ]:
```
note = 9

print ("Ma note est :",note)

if (note > 10 ) :
    print("J'ai mon UE !")
else :
    print("Je dois repasser en session 2.")
```

2.5 Exercice 3.1

Écrire un programme qui écrit “ce nombre est pair” si la variable note est un nombre pair et qui écrit “ce nombre est impair” dans le cas contraire

[ ]:
```
note = 17
```

2.6 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, exécutez-le plusieurs fois en modifiant la valeur de la note pour voir le comportement du `if…elif…else`:

[ ]:
```
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")
```

2.7 Conditions composées (`and`, `or`)

Il est possible de tester des conditions plus complexes, en combinant des conditions comme nous l'avons vu au début. Grâce à la commande `and`, on peut tester si deux conditions sont vérifiées en même temps. Voici un exemple :

[ ]:
```
note = 13
if (note >= 12) and (note < 14) :
    print("J'ai la mention assez bien.")
```

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

[ ]:
```
note = 9
if (note < 12) or (note >= 14) :
    print("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`, vue plus haut.

[15]:
```
note = 9
condition = (note < 12) or (note >= 14)
if  condition :
    print("Je n'ai pas la mention assez bien.")
```

```
Je n'ai pas la mention assez bien.
```

2.8 Exercice 3.2 : mention à un examen

Écrire un programme qui écrit la mention associée à la note définie à la première ligne, dans la variable `note` (assez bien pour $[12, 14[$, bien pour $[14, 16[$, très bien pour $[16, 20]$)

[ ]:
```
note = 17
```

2.9 Exercice 3.3 : discriminant d'un polynôme de degré 2

Écrire un programme qui calcule le discriminant `delta` d'un polynôme $ax^2 + bx + c$ et qui écrit "les solutions sont réelles" lorsque `delta` est positif ou nul, ou "les solutions sont complexes" dans le cas contraire.

On ne cherchera pas ici à calculer les solutions (c'est l'exercice suivant).

[ ]:
```
a = 1
b = 2
c = 3

delta =
```

2.10 Exercice 3.4 : racines d'un polynôme de degré 2

Reprendre le programme écrit à l'exercice 3.3, en calculant puis en affichant les racines du polynôme dans le cas où elles sont réelles. On rappelle que les deux racines sont alors données par :

$$x_1 = \frac{-b - \sqrt{\Delta}}{2a}$$

et

$$x_2 = \frac{-b + \sqrt{\Delta}}{2a}$$

[ ]:
```
a = 1
b = 2
c = 3
```

2.11 Exercice 3.5 : racines d'un polynôme de degré 2

Reprendre le programme précédent, en calculant puis en affichant les racines du polynôme dans tous les cas. Pour vous aider, voici un schéma logique du programme.

[ ]:
```

```

2.12 Exercice 3.6 : année bissextile

Vérifier si une année est bissextile.

On rappelle qu'une année est bissextile si l'une des deux conditions suivantes est vérifiée : - elle est un multiple de 4 mais pas de 100 - elle est un multiple de 400

[ ]:

2.13 Exercice 3.7 : franchise d'assurance auto

Votre assurance auto vous rembourse 10% des frais de réparation suite à un accident à condition que ce remboursement soit supérieur de 150 euros (la franchise). Par ailleurs, votre contrat limite le remboursement à 1000 euros (si le remboursement calculé est plus grand, vous ne touchez que 1000 euros). Écrivez un programme qui affiche le montant du remboursement en fonction du montant des travaux (variable `montant`*). Modifiez la valeur de* `montant` *pour tester que le programme fonctionne comme attendu.*

[ ]:
```
montant = 2000
```

2.14 Exercice 3.8 : jour de la semaine

À partir d'une date quelconque de l'année 2019 (ex. 15/09), calculer le nombre de jours qui se sont écoulés depuis le début de l'année (1/01).

[ ]:

2.15 4. Partie facultative

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

2.15.1 Exercice 4 (facultatif)

Reprendre l'exercice 3.5 (racines d'un polynôme de second degré) avec les complexes, mais sans `if`

[ ]:

TD3

March 4, 2020

1 Les boucles

Les boucles permettent de répéter plusieurs fois un ensemble d'opérations, de façon légèrement différente. C'est ce qui fait la puissance des programmes informatiques : ils ne se lassent pas de répéter une opération un million de fois.

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

[2]:
```
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)
```

```
1^2 =  1
2^2 =  4
3^2 =  9
4^2 =  16
5^2 =  25
6^2 =  36
7^2 =  49
8^2 =  64
9^2 =  81
10^2 =  100
```

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

Si on regarde bien, on réalise en fait plusieurs fois la même opération en modifiant un paramètre. 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 répètre un bloc **tant que** (`while`) une condition (au sens vu dans le TD précédent) 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.
- Un des instructions au moins doit donc avoir un effet sur la condition et la faire passer à `False`
- Les conditions peuvent être multiples.
- Il est possible de répéter le bloc sans savoir à l'avance combien de fois on va le répéter.

Programmons un simple compteur affichant les nombres de 0 à 9 :

[3]:
```
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'avancée 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'incrémentation `nb = nb -1`

1.1.1 Exercice

Sans l'exécuter, indiquer ce qu'affichera le programme suivant, dans lequel on a inversé les deux dernières lignes du programme précédent. Dans un second temps, exécuter le programme pour vérifier.

[ ]:
```
nb = 0

while nb < 10 :
    nb = nb + 1
    print(nb)
```

1.1.2 Exercice : Erreur courante

Une erreur que l'on commet souvent en débutant (voire plus tard) est d'oublier d'indenter ou de mal indenter. Exécuter le programme suivant, fortement inspiré du précédent et expliquer ce que vous obtenez en sortie.

[2]:
```
nb = 0

while nb < 10 :
    nb = nb + 1
print(nb)
```

```
10
```

1.1.3 Exercice 1.0

Sans l'exécuter, examiner le programme suivant et écrire de la façon la plus précise possible, sur un papier, ce qui sera affiché lorsqu'il sera exécuté.

Dans un second temps, exécuter le programme pour vérifier votre prédiction

[ ]:
```
nb = 0

while nb < 100 :
    print(nb)
    nb = nb**2 + 1
```

1.1.4 Exercice 1.1 :

1) Écrire un programme qui écrit les carrés des 10 premiers entiers, avec une boucle `while`. L'éxécution du programme doit donner

   1^2 = 1 2^2 = 4 3^2 = 9 4^2 = 16 5^2 = 25 6^2 = 36 7^2 = 49 8^2 = 64 9^2 = 81 10^2 = 100

On pourra s'inspirer fortement du programme donné en exemple pour introduire la boucle `while`.

[ ]:
```

```

1.1.5 Exercice 1.2

Ajouter la possibilité de choisir facilement le nombre de valeurs à afficher (de 1 à 100, de 1 à 1000...).

[ ]:
```
```

1.1.6 Exercice 1.3

Adapter ensuite ce code pour que l'on puisse facilement changer l'exposant (écrire les cubes ou les puissances quatrièmes des entiers successifs)

[ ]:
```
```

1.1.7 La fonction `input()`

Dans la suite, l'utilisateur va devoir entrer un nombre. Ceci est réalisé par la commande `input`.

[ ]:
```
choix = input("Entrer un nombre : ")
print("Vous avez choisi : ", choix)
```

Attention, elle renvoie la chaîne de caractères entrés par l'utilisateur (réessayez le code précédent en entrant un mot plutôt qu'un nombre) et si on veut accéder à une valeur numérique, il faut la convertir en un type numérique, un `int` ou un `float`, selon l'usage qu'on veut en faire.

[ ]:
```
choix = input("Entrer un nombre : ")
print("Vous avez choisi : ", float(choix))
```

1.1.8 Exemples de boucle `while` qui ne connait 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.

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

L'exemple suivant montre une boucle potentiellement éternelle, la condition n'étant jamais fausse. Pour sortir de la boucle, il est alors possible d'utiliser la commande `break`.

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

1.2 La boucle `for`

L'autre structure permettant de faire des opérations répétitives est 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 crochets `[]` et de séparer les éléments avec une virgule `,`. 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 :

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

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

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 Exercice 1.4

Créer une liste contenant les entiers de 0 à 9 et créer une boucle qui les affiche un par un

[ ]:
```

```

1.2.3 Exercice 1.5

Écrire un programme qui écrit les carrés des 10 premiers entiers, avec une boucle `for`. L'éxécution du programme doit donner

```
1^2 =  1
2^2 =  4
3^2 =  9
4^2 =  16
5^2 =  25
6^2 =  36
7^2 =  49
8^2 =  64
9^2 =  81
10^2 =  100
```

On pourra s'inspirer fortement du programme précédent.

[ ]:

1.2.4 Les listes - 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 :

[ ]:
```
phrase = """Il fait 30°C."""

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

1.2.5 Les listes - Suites d'entier : fonction `range()`

Nous avons eu besoin plus haut d'une suite d'entiers successifs, par exemple $0, 1, 2, 3, 4....$ Nous l'avons fait de façon manuelle :

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

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

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

Mais si la liste devient très grande, ce n'est pas bien pratique. Python fournit la fonction `range()` qui permet de la générer automatiquement. 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 :

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

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

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

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

1.2.6 Exercice 1.6 : Reprenons l'exemple initial :

1) Reprendre l'exemple du début et remplacer ces 10 lignes par une boucle `for` équivalente.

[ ]:
```
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.

1.3 Accumulateurs

Nous présentons ici une technique extrêmement utile et courante. Pour calculer la somme des entiers de 1 à 10, on peut utiliser une boucle `for`, en créant une variable `somme` initialisée à 0, à laquelle on ajoute chacun des éléments de la liste grâce à la boucle. À la fin, la variable `somme` contient le résultat cherché.

[2]:
```
liste = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
somme = 0
for nombre in liste:
    somme = somme + nombre
print(somme)
```

```
55
```

1.3.1 Exercice 1.7

Calculer la somme des carrés des nombres compris entre 1 et 10 ($1^2+2^2+3^3+4^2+5^5+6^2+7^2+8^2+9^2+10^2$), avec une boucle `for`, sur le modèle précédent.

[ ]:
```
liste = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

2 Exercices d'application

2.0.1 Exercice 2.1 : 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. Pour votre confort, la liste est déjà fournie.

[1]:
```
notes = [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]
```

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.

[ ]:
```

```

2.0.2 Exercice 2.2 : Suite de Fibonacci

La suite de Fibonacci peut être considérée comme le tout premier modèle mathématique en dynamique des populations ! Elle 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< /div >< divalign = "center" > F_1 = 1< /div >< divalign = "center" > F_n = F_{n-1} + F_{n-2}$

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

[ ]:
```

```

2.0.3 Exercice 2.3 : Nombres premiers

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

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

[ ]:

TD4

March 4, 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 uselles `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 :

[1]:
```
def cube(x) :
    print(x*x*x)
```

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

[2]:
```
cube(4)
cube(10.1)
```

```
64
1030.301
```

1.1.1 Exercice

Définir une fonction (avec le nom que vous voulez) qui prend deux arguments `x` et `y` et qui écrit la somme de ses deux arguments. Tester son fonctionnement en l'appelant

[ ]:

1.1.2 Exercice

Sans l'exécuter, analyser le programme suivant pour indiquer, sur une feuille, ce qu'il affichera quand on l'exécutera

[3]:
```
def mafonction(x, y):
    print(x, y, x*y)

mafonction(2, 3)
mafonction(0, 1)
```

```
2 3 6
0 1 0
```

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 defaut.

[4]:
```
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 :

[5]:
```
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 un arguments `nombre`. Elle affiche la table de multiplication associée à `nombre` de 0 jusqu'à 5 :

[6]:
```
def TableMultiplication(nombre=2):
    for i in (0, 1, 2, 3, 4, 5):
        print(i,"*",nombre,"=",i*nombre)
```

[7]:
```
TableMultiplication(3)
```

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

Voici une version un peu plus élaborée. Cette fonction prend deux arguments `nombre` et `max`. Elle affiche la table de multiplication associée à `nombre` de 0 jusqu'à `max` :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` :

[8]:
```
def TableMultiplication(nombre=2, max=10):
    for i in range(0,max+1):
        print(i,"*",nombre,"=",i*nombre)
```

[9]:
```
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
```

[10]:
```
TableMultiplication(5) # on demande la table de 5. On ne spéccifie 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
```

[11]:
```
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
```

[12]:
```
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 ce 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 :

[13]:
```
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
```

Le corps de la fonction peut contenir des calculs, des tests, des boucles. Pour commencer par un exemple simple, voici une fonction qui renvoie la chaîne de caractère "pair" si le nombre est pair et "impair" sinon :

[14]:
```
def parite(i) :
    if (i%2==0):
        return "pair"
    else:
        return "impair"

nombre = 27
resultat = parite(nombre)
print("Le nombre", nombre,"est", resultat)
```

```
Le nombre 27 est impair
```

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. La fonction définie dans

le programme suivant affiche un nombre et les deux suivants.

[15]:
```
def deux_suivants(i) :
    return i, i+1, i+2

print(deux_suivants(27))
```

```
(27, 28, 29)
```

1.3.1 Exercice

Sans l'exécuter, analyser le programme suivant pour indiquer, sur une feuille, ce qu'il affichera quand on l'exécutera. Résumer en une phrase le rôle de cette fonction.

[16]:
```
def mafonction(x,y) :
    a,b =  x,y
    if a > b :
        a,b = b,a
    return a,b

mafonction(22,2)
```

[16]:
```
(2, 22)
```

1.4 Les librairies de fonctions

Nous avons déjà chargé une librairie de fonctions : *numpy*. Pour cela nous avions 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. Il est préférable d'appeler une librairie de la façon suivante

```
import numpy
```

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

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

C'est un peu fastifieux et la plupart des utilisateurs de python préfèrent utiliser un raccourci sous la forme :

```
import numpy as np
```

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

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

Ceci est un peu plus lourd que d'utiliser simplement `sqrt(12)`, mais maintenant lorsque nous appellons 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.

[17]:
```
import numpy as np
import math as mt

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

```
3.4641016151377544
3.4641016151377544
```

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

[18]:
```
nb = 10+10j
print(np.sqrt(nb))
print(mt.sqrt(nb))
```

```
(3.4743442276011565+1.4391204994250741j)
```

```
␣
↪---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-18-14805480af57> in <module>
  1 nb = 10+10j
  2 print(np.sqrt(nb))
----> 3 print(mt.sqrt(nb))


TypeError: can't convert complex to float
```

1.5 Exercice 1 : Arrondir un nombre

1) Ecrire une fonction qui tronque un nombre à n chiffres après la virgule.
2) Tester avec plusieurs exemples, si cette fonction fonctionne correctement.

3) Adapter ensuite la fonction pour qu'elle arrondisse correctement le nombre. Pensez au cas où le nombre est négatif.

[ ]:

1.6 Exercice 1 bis : calcul d'une somme

Ecrire une fonction `somme` qui, lorsqu'on l'appelle sous la forme `somme(n)` calcule la somme des entiers de 1 à n. Tester la fonction avec `somme(9)` qui vaut 45.

[ ]:

1.7 Problème 1 : Racine d'une fonction mathématique par dichotomie

Dans cet exercice, nous allons utiliser tout ce que nous avons appris jusqu'ici et apprendre un algorithme très utilisé en informatique: ***la dichotomie****. Nous allons le faire avec un exemple.*

Nous souhaitons trouver numériquement les racines du polynôme suivant :

$f(x) = x^3 + 3.6821627548\,x^2 - 3,7208387236\,x - 9,7979589711$

Nous voyons sur ce graphique qu'une racine est présente entre 0 et 3. Cela se voit car $f(0)$ est négatif et $f(3)$ est positif. Il existe donc une valeur entre les deux pour laquelle $f(x) = 0$.

On coupe alors l'intervalle en deux:

- si $f(1.5) > 0$ cela signifie que la racine est avant 1.5
- si $f(1.5) < 0$ cela signifie que la racine est après 1.5. C'est le cas dans notre exemple.

Nous savons donc maintenant que la racine appartient à $[1.5, 3]$. L'intervalle a été divisé par deux. Nous nous approchons de la racine.

Nous allons donc utiliser cela pour chercher de façon itérative (*i.e. qui est répétée plusieurs fois*) la position exacte de la racine. A chaque itération, nous allons diviser par deux l'intervalle.

Ici on part d'un intervalle de longueur 3. Après une itération, il ne fera plus que 1.5, puis 0.75... Après n itérations, il ne fera donc plus que $3/2^n$. Pour se rendre compte, au bout de 30 itérations l'intervalle fera 0.000000002793968. Nous serons donc très proche de la racine.

A vous d'écrire cet algorithme. On pourra tout d'abord écrire une fonction qui pour un x donné retourne $f(x)$. On peut ensuite écrire un code qui effectue la première itération, c'est à dire, ce qui est décrit au dessus. Il restera alors à répéter correctement la procédure n fois à l'aide d'une boucle.

[ ]:

TD5

March 4, 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 créer 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.

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

```
<class 'list'>
[]
```

1.2 Accèder à / modifier un élément de la liste

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

Par exemple :

[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'élement que l'on veut modifier :

[45]:
```
maliste[2]='toto'
print(maliste)
```

```
['a', 'b', 'toto', 'd', 'e', 'f', 'g']
```

Il est également possible d'extraire plusieurs éléments concécutifs. 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)

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

[47]:
```
['b', 'toto']
```

1.3 Ajouter un élément dans une liste

1.3.1 Méthodes et fonctions

*Nous avons vu précédemment plusieurs exemples de **fonctions**, qui peuvent avoir des arguments mais ne les modifient pas. Ici nous allons voir un nouveau concept, les **méthodes**, qui agissent sur un objet et le transforment. Nous allons le montrer avec un objet de type `list`. La syntaxe est différente de celle des fonctions.*

1.3.2 Exemple : la méthode `append()` pour ajouter un élément à la fin d'une liste

Pour utiliser une méthode, on indique le nom de la variable suivie d'un `.` puis le nom de la méthode avec ses arguments.

Par exemple, pour ajouter un élément à la fin d'une liste, on utilise la **méthode `append()` de la classe `list`**. Elle prend en argument la valeur à ajouter :

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

Voici un exemple, où l'on voit que la liste a été transformée.

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

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

1.3.3 Insertion d'un élément : `insert()`

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

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

Regardons ce que cela donne :

[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 élements d'indice supérieur à n, pour intercaler la valeur supplémentaire.

1.3.4 Exercice

Insérer le nombre qui semble manquer dans la liste suivante, puis afficher la liste complétée

```
[2, 4, 6, 8, 12, 14, 16]
```

[3]:
```
maliste = [2, 4, 6, 8, 12, 14, 16]
```

1.3.5 Supprimer un élement : `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 :

[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']
```

1.3.6 Exercice

Déterminer ce que fait le programme suivant, sans l'exécuter, puis vérifier en l'exécutant

[ ]:
```
maliste = ['Maxwell', 'Einstein', 'Kepler', 'Galilée']
maliste.append('Curie')
maliste.remove('Maxwell')
maliste.insert(1, 'Meitner')
print(maliste)
```

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

1.3.7 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 :

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

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

1.3.8 Nombre d'éléments dans une liste : `len()`

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

`len(maliste)`

Exemple :

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

[16]:
```
3
```

1.4 Parcourir une liste :

1.4.1 Avec une 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 :

[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.

1.4.2 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 :

[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.

1.4.3 Avec une boucle `for` et la fonction `enumerate()` :

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

[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 élements de la liste pendant que la variable `i` va servir de compteur et prendre ainsi le numéro de l'indice courant. Techniquement, la fonction `enumerate()` renvoie une succession de couples de valeurs, le premier élément du couple est le compteur, le second l'élément de la liste.

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

1.5 Exercice 1 : Échange de valeurs

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

Rappel pour échanger les valeurs contenues dans `a` et `b` :

```
a = 10
b = 12
a,b = b,a
```

[8]:
```
liste = ['Meitner', 'Maxwell', 'Curie', 'Einstein', 'Kepler', 'Galilée']
```

1.6 Exercice 1 bis : Recherche de petits nombres

Ecrire un programme qui à partir d'une liste, par exemple (1, 13, 33, 2, 4, 40), fabrique une nouvelle liste ne contenant que les nombres supérieurs à 10.

Algorithme : - Parcourir et afficher un à un les termes de la liste - Afficher seulement les termes $<$ 10 - Placer ces termes dans une nouvelle liste puis l'afficher

1.7 Exercice 2 : Liste symétrique

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

Algorithme : - Parcourir et afficher les termes de la liste - Inverser le sens de parcours de la liste (avec la fonction `reversed`) - Créer la liste symétrique (à l'nevers) - Vérifier si la liste et la liste symétrique sont identiques avec un test

[ ]:
```

```

1.8 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]

Algorithme : - Parcourir et afficher les termes de la liste - Utiliser le modulo (`%`) pour rammener l'angle antre 0 et 360 - Tester (avec un `if`) si l'angle est supérieur à 180, dans ce cas retrancher 360 - Placer le résulat dans la liste à la place de l'angle initial

[ ]:

1.9 Exercice 4 : Trier une liste

Écrire un programme qui trie du plus petit au plus grand une liste composée de nombres quelconques. *(attention aux petits malins, on vous demande de créer votre propre programme et non pas d'utiliser une fonction toute faite).*

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

Algorithme : - Parcourir et afficher les termes de la liste - Lors du parcours, identifier le nombre le plus petit en faisant un test (`if`) - Ajouter ce minimum dans une nouvelle liste, et effacer ce dernier de la première liste - Réitérer ce processus à l'aide d'une boucle

[ ]:

Reprenons le cours ici.

1.10 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 :

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

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

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

```
['Fer', 26]
['Al', 13]
```

Mais il est possible de récupérer directement l'élement 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 :

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

```
26
L'élément Fer a 26 protons.
```

1.11 Les compréhensions de liste

On appelle **compréhension de liste** la technique suivante qui permet de modifier ou de filtrer une liste très facilement.

1.11.1 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)

[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 :

[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 élements 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 `=`

1.12 Filtres sur une liste

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

[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 élements de la liste `nombres` en ne considérant que les nombres positifs.
- pour chaque élement retenu, on retourne sa valeur pour former une nouvelle liste `nombres_positifs`.

1.13 Fonctions sur les listes :

1.13.1 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 :

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

print(sum(nombres)) # fait la somme des élements 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
```

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

[11]:
```
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
```

[11]:
```
['ananas', 'bonjour', 'girafe', 'schtroumpf', 'tagada', 'zoo']
```

Attention, `sorted(mots)` est une fonction (pas une méthode) et `mots` est son argument. La liste `mots` n'est pas modifiée par le programme précédent :

[12]:
```
print(mots)
```

```
['bonjour', 'girafe', 'schtroumpf', 'tagada', 'ananas', 'zoo']
```

1.13.2 Fonctions provenant de librairies

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

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

[65]:
```
4
```

1.13.3 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 graphe représentant des données.

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

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

Si sous linux, cette librairie n'est pas installée, alors 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 :

[15]:
```
for i in range(0,11):
    x = i
    y = i*i
```

ou encore, dans un style différent mais avec le même résultat,

[ ]:
```
x = range(0,11)
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 (nous introduirons dans un TD ultérieur la notion de tableau, qui permet ce genre de syntaxe, mais patience).

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

```
␣
↪---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)
```

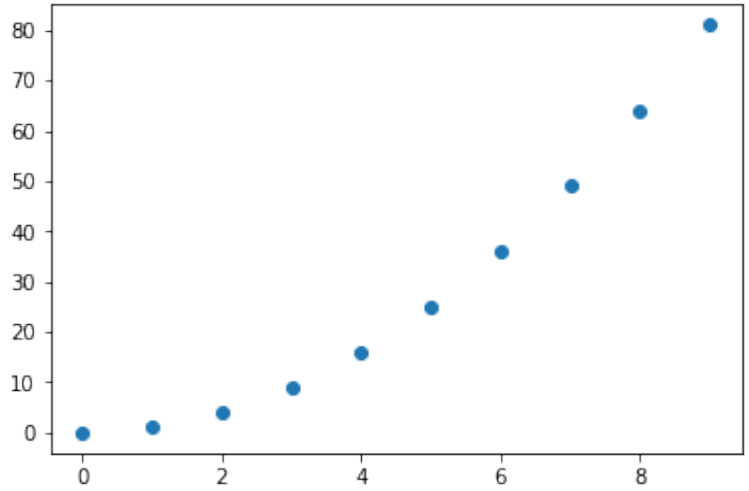
```
<ipython-input-13-3c3ad855defc> in <module>
      1 x = range(0,10)
----> 2 y = x**2


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

Reprenons nos listes correctement construites.

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

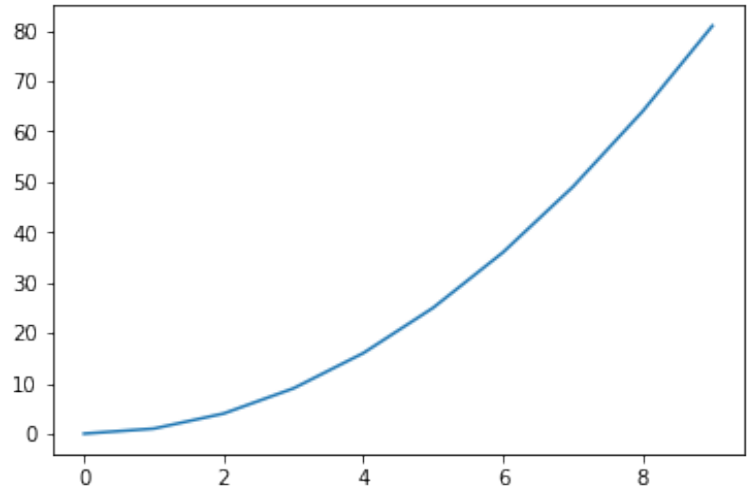
On peut alors afficher yen fonction de `x` à l'aide d'un nuage de points :

[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 reliant les points avec `plot` à la place de `scatter`. Les points sont alors reliés par des segments.

[29]:
```
plt.plot(x,y) # on génère une courbe reliant les points
plt.show() # on affiche le graphique
```



1.14 Exercice 5 :

Produire la courbe représentative de la fonction $f(x) = \log(x)$ entre 0 et 30.

Algorithme : - Créer une liste `x` allant de 0 à 30 à l'aide de la fonction `range` - Parcourir cette liste et calculer $\log(x)$ - Ajouter ce résultat dans une liste `y` - Utiliser la fonction `scatter` ou `plot` de matplotlib pour afficher les points (`y` vs `x`)

[ ]:

1.15 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. Chaque liste contient les trois coordonnées du vecteur. On rappelle que le produit vectoriel est donné par :

[ ]:

Tester ensuite votre fonction avec $u = (1, 0, 0)$ et $u = (0, 1, 0)$

[ ]:

1.16 Exercice 7 : Tableau périodique

Chaque élément de la liste suivante est une liste à deux éléments, contenant le symbole et le numéro atomique Z d'une espèce atomique.

Atomes = [[“Fer”,26],[“Ag”,47],[“Ca”,20],[“Al”,13],[“Ne”,10],[“O”,8],]

Ajouter à cette liste l'or de symbole “Au” et Z=79 puis afficher la liste complétée.

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

Pour trier par ordre alphabétique du nom d'élément, on peut utiliser la fonction `sorted`

[55]:
```
sorted(Atomes)
```

[55]:
```
[['Ag', 47],
 ['Al', 13],
 ['Au', 79],
 ['Ca', 20],
 ['Fer', 26],
 ['Ne', 10],
 ['O', 8]]
```

Écrire un programme qui trie les élements chimiques précédents par ordre croissant de numéro atomique.

Algorithme : - Écrire un programme qui crée une liste dans laquelle numéro et nom sont inversés :

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

- Trier cette nouvelle liste avec la commande `sorted`

[61]:
```
Atomes = [["Fer",26],["Ag",47],["Ca",20],["Al",13],["Ne",10],["O",8],["Au",79]]
```

1.17 Exercice 7bis

Chaque élément de la liste suivante est une liste à quatre éléments donnant le prénom, le nom, le prénom, la date de naissance et la date de décès de différentes personnalités du monde de la physique.

```
liste=(('James Clerk', 'Maxwell', 1831, 1879), ('Albert', 'Einstein', 1879, 1955), ('Isaac', 'Newt
```

Ajouter un élément à la liste pour Lise Meitner (1878-1968) et afficher la liste complète.

[1]:
```
liste=[['James Clerk', 'Maxwell', 1831, 1879], ['Albert', 'Einstein', 1879,␣
↪1955], ['Isaac', 'Newton', 1643, 1727]]
```

Écrire un programme affichant la liste des noms par ordre de date de naissance croissante, sous la forme

```
Isaac Newton de 1878 à 1968
James Clerk Maxwell de 1831 à 1879
```

`(etc...)`

[ ]:

1.18 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

Algorithme : - Créer une liste `entiers` d'entiers de 0 à 100 (`range`) - Parcourir cette liste et créer une seconde liste `x` de nombres entre 0 et 10 par pas de 0.1 - Parcourir cette nouvelle liste et tester si la partie entière de chaque élément est paire - Utiliser cette condition pour remplir la liste `y` contenant 1 si la condition est réalisée et 0 sinon - Afficher la courbe (`y` vs `x`) obtenue

[ ]:

TD6

March 4, 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 *binaire*. Nous y reviendrons plus tard.

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

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

[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`.

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

[60]:
```
str
```

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

[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`.

[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.

[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()` :

[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.")
```

[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.

[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 plius 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 :

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

[77]:
```
'  mon texte  '
```

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

[78]:
```
'  MON TEXTE  '
```

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

[79]:
```
'  mon texte  '
```

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

[80]:
```
'mon TEXTE'
```

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

[87]:
```
'T'
```

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

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

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

[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 :

[72]:
```
```

```
texte = "Je m'appelle {prenom} et j'ai {age} ans." # deux labels {prenom} et␣
 ↪{age} sont spécifiés
print(texte) # on peut afficher la chaîne précédente.
```

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

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

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

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

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

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

Si l'on souhaite modifié la variable `texte` de façon définitive, on peut écrire :

[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 replacent 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 rrvi.e”*

1) Ouvrir le fichier et afficher le texte qu'il contient

2) Décoder le code

[ ]:
```

```

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`.

[5]:
```
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 bianaire.

Le fichier ne sera donc pas lisible par un humain, moais l'ordinateur pour y mettre des informations supplémaires 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.

[6]:
```
import numpy as np


fichier = open("fichiers/data.bin","wb")

x = []
y = []

for i in range(20):
    x.append(i*0.1)
    y.append(np.sin(i*0.1))

pickle.dump(x,fichier)
pickle.dump(y,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.

[7]:
```
fichier = open("fichiers/data.bin","rb")

x = pickle.load(fichier)
y = pickle.load(fichier)

print(x)
print(y)
```

```
[0.0, 0.1, 0.2, 0.30000000000000004, 0.4, 0.5, 0.6000000000000001,
0.7000000000000001, 0.8, 0.9, 1.0, 1.1, 1.2000000000000002, 1.3,
1.4000000000000001, 1.5, 1.6, 1.7000000000000002, 1.8, 1.9000000000000001]
[0.0, 0.09983341664682815, 0.19866933079506122, 0.2955202066613396,
0.3894183423086505, 0.479425538604203, 0.5646424733950355, 0.6442176872376911,
```

```
0.7173560908995228, 0.7833269096274833, 0.8414709848078965, 0.8912073600614354,
0.9320390859672264, 0.963558185417193, 0.9854497299884603, 0.9974949866040544,
0.9995736030415051, 0.9916648104524686, 0.9738476308781951, 0.9463000876874145]
```

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:

[8]:
```
Toto = pickle.load(fichier)
```

```
␣
↪---------------------------------------------------------------------------

EOFError                                  Traceback (most recent call last)

<ipython-input-8-fb57e1700de5> in <module>
----> 1 Toto = pickle.load(fichier)


EOFError: Ran out of input
```

N'oubliez pas de fermer le fichier.

[9]:
```
fichier.close()
```

1.8 Exercice 2 : PIB par pays

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

1) Charger le fichier, récuperer 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 ?

[ ]:
```

```

1.9 Problème 2 : Chûte libre

Un avions lâche une caisse de matériel d'une altitude H et une vitesse initiale horizontale $\vec{v_0}$. Nous allons étudier la trajetoire de la caisse.

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

- $a_x = 0$

- $a_z = -g$
- $v_x = v_0$
- $v_z = -g\,t$
- $x = v_0\,t$
- $z = -1/2g\,t^2 + H$

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

1) Tracer la trajectoire jusqu'au sol, c'est à dire z en fonction de x.

2) Cette chûte 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 précedemment.

3) D'où provient la différence observée ?

[ ]:

TD7

March 4, 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 impoter la librairie *numpy*, il suffit d'utiliser la fonction `array()` pour convertir une liste `list` en `ndarray` :

[2]:
```
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 :

[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 biens 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`) :

[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`) :

[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` :

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

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

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

[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 :

[52]:
```
x = np.random.randint(low=10, high=30, size=6) # 6 nombres tirés aléatoirement␣
 ↪entre 10 et 30
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 :

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

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

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

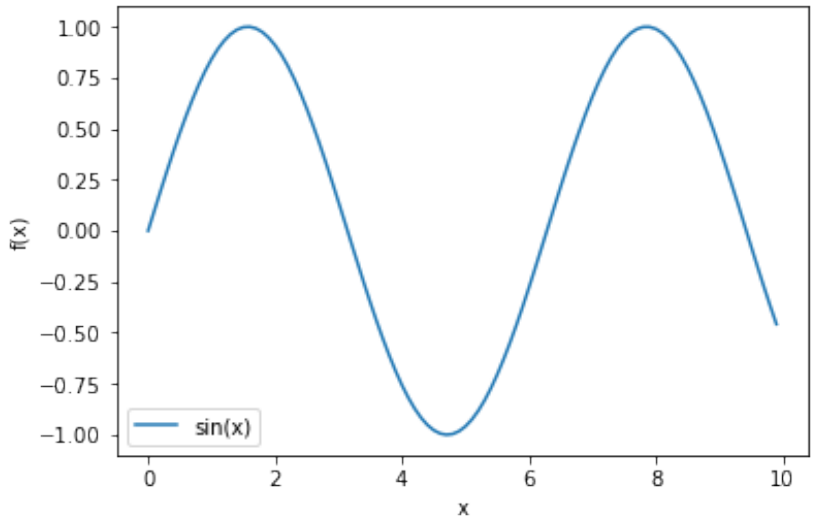
```
[ 0  3  6  9 12 15 18 21 24 27]
```

[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 :

[10]:
```
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 commes les autres :

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

[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 `nbarray`.

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 :

[12]:
```
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 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 :

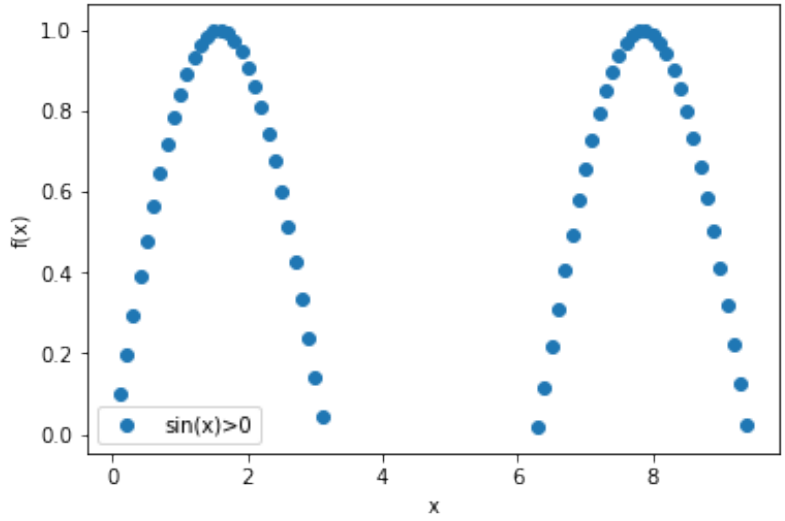
[13]:
```
print(y[z])
```

```
[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 :

[16]:
```
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 pairs par des -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).

[ ]:

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.

[ ]: