Cours python 1ere

6 septembre 2021

1 Introduction : markdown

1.1 Syntaxe

1.1.1 Sous-sous titre

Ceci est un **texte** qui peut être *mis en forme*. On peut écrire des maths x^2. On peut écrire du code : `print("Hello")`

```
<h1>Ceci est du html</h1>
<a href="cours">Mon cours</a>
```

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nunc nunc odio, viverra maximus sodales sit amet, facilisis sit amet enim. Donec a scelerisque lacus. Nullam laoreet eget orci id ornare.

Il faut une ligne vide pour changer de paragraphe.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nunc nunc odio, viverra maximus odales sit amet, facilisis sit amet enim. Donec a scelerisque lacus. Nullam laoreet eget orci id ornare.

- une
- liste
- de
- choses
  - sous
  - liste

1. une
2. liste
3. numérotée

-------------

Après la ligne

Ecrire des maths : $x^2$ ou une formule séparée : $$\frac{x^2}{\sqrt{3}}$$
```

2 Les bases

2.1 Les types

2.1.1 Les nombres

2 types de nombres en python :

- les nombres entier (**int**)
- les nombres à virgule (**float**)

La fonction `type(donnee)` renvoie le type de la donnée en argument.

```
type(2)
```

```
int
```

```
type(2.)
```

```
float
```

Les opérations disponibles :

- addition : `+`
- soustraction : `-`
- multiplicaiton : `*`
- division : `/`
- division entière : `//`
- modulo (reste de la division) : `%`
- puissance : `**`

```
# <---- ceci est un commentaire
print(2+2) # ceci est une addition

# division
print(3/2)

# division entière
print(3//2)
# reste
print(3%2)
```

```
4
1.5
1
1
```

```
# Le python n'est pas limité dans la
# précision des entiers

print(42**150)
```

```
307180441096283624310834810220933473375225212052491651628878831787928561891324276922154623380102930966024074468487016106864224281931718647811320581112979155344834823272145421950205151779439580895845237852365648330865859384385910863109588149862 4
```

```
# Attention aux calculs avec des nombres à virgule
print(0.2+0.1)
```

```
0.30000000000000004
```

```
# Conversions de type

print(float(2))
print(int(3.14159))
print(int(3.9999))
```

```
2.0
3
3
```

Autres types (à voir plus tard)

- booléens (vrai/faux)
- chaînes de caractères (“bonjour”)
- listes & les n-uplets
- dictionnaires
- objets
- ...

3 Les variables

Une variable est un espace de mémoire nommé.

On **assigne** une valeur à une variable avec le signe égal (=)

```
a=3
```

met la valeur `3` dans la case mémoire nommée `a`

- si la case existe déjà, on remplace la valeur existante
- si la case n’existait pas, on la crée et on met la valeur dedans

Règles pour les noms des variables :

- en minuscules
- sans accents
- sans espaces (utiliser le _ tiretduhuit)
- commence par une lettre

Bon :

- a
- ma_variable
- mon_resultat

Pas bon :

- maVariable
- ma Variable
- mon Résultat

```
x=3
print(x)
```

```
3
```

```
# Raccourcis intéressants :


x += 3 # pareil que x = x + 3
x -= 3 # pareil que x = x - 3
x /= 2 # pareil que x = x / 2
x *= 5 # pareil que x = x * 5

# Assignation multiple
x = 0
y = 0
z = 0
x = y = z = 0
```

3.0.1 Exercices

3.0.1.1 Exercice 1

Avec a=1 et b=2, prévoir la valeur de a += (b-3)

```
a=1
b=2
a += (b-3) # a <---- a + (b-3)
print(a, b)
```

```
0 2
```

Attention : le signe = est une opération d'**affectation** (parfois écrite dans d'autres langages `<---` : de la droite vers la gauche)

3.0.1.2 Exercice 2

avec `a="toto"` et `b="pouet"`... inverser le contenu de `a` et de `b`

```
a="toto"
b="pouet"
print("au départ : ", a, b)

# a vous de jouer
# b <--- a
# a <--- b
```

```
c = a
print("après c=a", a,b,c)
a = b
print("après a=b : ", a,b,c)
b = c
print("après b=c : ", a,b,c)

print("résultat :", a, b)
```

```
au départ :  toto pouet
après c=a toto pouet toto
après a=b :  pouet pouet toto
après b=c :  pouet toto toto
résultat : pouet toto
```

3.1 Les fonctions

3.1.1 Les fonctions et les librairies

Par défaut, le python a très peu de fonctionnalités. Mais il existe une énorme **bibliothèque** de fonctions qu'on peut utiliser dans nos programmes. L'index de la bibliothèque se trouve https ://docs.python.org section Library Reference.

Pour utiliser une fonction d'une bibliothèque, il faut **l'importer** (= dire à Python de charger le code de la fonction)

```
import math  # bourrin : importe TOUTE la bibli
math.sqrt(2)
math.cos(3)

# mieux : importation sélective
from math import sqrt,cos
sqrt(2)
cos(3)

# PAS BIEN DU TOUT, NE JAMAIS FAIRE
from math import *
cos(3)
sinh(12)

from math import cos, sin, pi
print(sin(3.14))
print(sin(pi))
```

```
0.0015926529164868282
1.2246467991473532e-16
```

```
print(pi)
```

```
3.141592653589793
```

```
import math
math.sin(math.pi)
```

```
1.2246467991473532e-16
```

3.1.2 Exercices

3.1.2.1 Exercice 1

Calculer le PGCD de 4956 et 8052... (en anglais : GCD)

```
from math import gcd
gcd(4956, 8052)
```

```
12
```

3.1.2.2 Exercice 2

Calculer la racine carrée de `40!`.. idem, RTFM

```
# en anglais : factorial et square root
from math import factorial, sqrt
sqrt(factorial(40))
```

```
9.032802905233224e+23
```

3.2 Créer ses propres fonctions

Une fonction est un ensemble d'instructions avec un nom.

exemple : je veux

- multiplier un nombre par 5
- lui ajouter 12
- afficher "bip bip"

et je veux le faire plusieurs fois

```
a = 10
a = a * 5
a = a + 12
print("bip bip")

a = 20
a = a * 5
a = a + 12
print("bip bip")

a = 4
a = a * 5
a = a + 12
print("bip bip")

# ^^^^ trop de répétitions, on fait mieux

# définition de la fonction :
#   a est un argument de la fonction
def mafonction(a):
    a = a * 5
```

```
    a = a + 12
    print("bip bip")

# appel de la fonction
mafonction(10)
mafonction(20)
mafonction(4)
```

```
bip bip
bip bip
bip bip
bip bip
bip bip
bip bip
```

Une fonction en maths : $f(x) = 3x + 2$ (elle prend la valeur de x, la multiplie par 3 et ajoute 2) s'écrit en python :

```
def f(x):
    return 3*x+2
```

Plus précisément :

- le nom de la fonction = mêmes contraintes que pour le nom des variables (minuscules, pas accents ni espaces)
- on définit la fonction avec le mot clef **def**
- entre parenthèses, on fournit zéro ou plusieurs **arguments**
- **on n'oublie pas les deux points à la fin de la définition**
- le corps de la fonction (= les instructions qui font partie de la fonction) sont *indentées* (décalées par rapport à la marge)
- la réponse (éventuelle) de la fonction est donnée avec le mot clef **return**

```
def f(x):
    return 3*x+2
print(f(3))
print(f(f(1)))
print(f(f(f(f(f(f(f(f(3))))))))
```

```
11
17
8747
```

```
# Il faut fournir les arguments...
print(f())
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-55-04323a1e865c> in <module>
      1 # Il faut fournir les arguments...
----> 2 print(f())
```

```
TypeError: f() missing 1 required positional argument: 'x'
```

```python
# Une fonction peut avoir des arguments par défaut
# le b=0 signifie la valeur utilisée si
#  la fonction est appelée sans b
def affine(x, a, b=0):
    return a*x+b

print(affine(2,3,5))
print(affine(2,3,0))
print(affine(2,3))
```

```
11
6
6
```

```python
# Passage des arguments par position
# dans le même ordre que la définition
# Exemple
print(affine(2,3,5))

# Passage des arguments par nom
# ordre indifférent
print(affine(a=3,b=5,x=2))
print(affine(a=3,x=2))

# Mix de par position et par nom
# attention : les arguments par position sont en premier
print(affine(2,3,b=5))
```

```
11
11
6
11
```

```python
# Il y a des variables "internes" à la fonction (locales),
# il y a des arguments
# il y a des variables "externes" (définies en dehors de la fonction)

toto = 1
titi = 2
tata = 3


def mafonction(titi, tutu):
    toto = 4  # recrée une variable toto **locale** (existe pas dehors)
    print("toto = ", toto)
    print("titi = ", titi) # argument de la fonction (donné à l'appel)
    print("tata = ", tata) # tata : pas argument, pas défini dans la fonction
                           # ==> on va chercher au dessus dans le prog principal
    print("tutu = ", tutu) # argument (donné à l'appel) = local
```

```
mafonction(titi=7, tutu=8)
print("----------------------")
print("toto = ", toto)
print("titi = ", titi)
print("tata = ", tata)
# print("tutu = ", tutu) ne marche pas car tutu n'existe que dans la fonction


toto =  4
titi =  7
tata =  3
tutu =  8
----------------------
toto =  1
titi =  2
tata =  3

# Une fonction est un "objet" python comme les autres

def yo():
    print("yo")
def yi():
    print("yi")

print(type(yo))
# on peut la mettre dans une variable
ya = yo
ya()

# on peut donner une fonction en argument à une fonction
def double(f):
    print("Je dis")
    f()
    print("et puis")
    f()

double(f=yo)
double(yi)

<class 'function'>
yo
Je dis
yo
et puis
yo
Je dis
yi
et puis
yi
```

3.2.1 Exercices

3.2.1.1 Exercice 1

définir une fonction pythagore(cote1, cote2) qui renvoie l'hypothénuse quand on lui donne 2 côtés

```
def pythagore(cote1, cote2):
    from math import sqrt
    return sqrt(cote1**2+cote2**2)
```

```
def pythagore(cote1, cote2):
    from math import sqrt
    a2 = cote1**2
    b2 = cote2**2
    c2 = a2+b2
    return sqrt(c2)
pythagore(3,4)
```

```
5.0
```

3.2.1.2 Exercice 2

Une équation $ax^2 + bx + c$. Faire une fonction deg2 prenant comme arguments a, b, c et x, avec b qui vaut 1 par défaut, c qui vaut 0 par défaut et qui renvoie le résultat de $ax^2 + bx + c$

```
def deg2(x,a,b=1,c=0):
    return a*x**2+b*x+c
deg2(1,1,c=3)
```

```
5
```

3.2.1.3 Exercice 3

le produit scalaire de 2 vecteurs (ux, uy) et (vx, vy) est défini par $u.v = ux * vx + uy * vy$

Définir une fonction produit_scalaire qui renvoie le produit scalaire de 2 nombres Si on ne donne pas une coordonnée de vecteur, par défaut elle est nulle

```
# Exo 2 : le produit scalaire de 2 vecteurs (ux, uy) et (vx, vy)
# est défini par u.v=ux*vx+uy*vy
# Définir une fonction produit_scalaire qui renvoie le produit
# scalaire de 2 nombres
# Si on ne donne pas une coordonnée de vecteur, par défaut elle
# est nulle

def produit_scalaire(ux=0,uy=0, vx=0, vy=0):
    return ux*vx+uy*vy

# La tester avec u=(1,0) v=(0,1)  et avec u=(3,4), v=(0,6)
print(produit_scalaire(1,0,0,1))
print(produit_scalaire(ux=1,vy=1))
print(produit_scalaire(3,4,0,6))
print(produit_scalaire())
```

```
0
0
24
0
```

4 Les structures du langage

4.1 Les booléens et les tests

Un booléen : valeur vraie ou fausse / 0 ou 1 / en python : `True` ou `False`

```
type(True)
```

```
bool
```

```
type(False)
```

```
bool
```

Les opérations sur le booléens sont :

- et = and
- ou = or
- non = not

```
print("vrai et vrai = ", True and True)
print("vrai ou faux = ", True or False)
print("non vrai = ", not True)
print("vrai non-et faux = ", not(True and False))
```

```
vrai et vrai =  True
vrai ou faux =  True
non vrai =  False
vrai non-et faux =  True
```

Les **tests** renvoient une valeur booléenne

“est-ce que machin est plus grand que truc ?” : vrai / faux

liste des tests disponibles :

- > supérieur
- < inférieur
- >= supérieur ou égal
- <= inférieur ou égal
- == égalité
- != différent
- `in` cf plus tard

```
print(3<4)
print(2>=2)
print(3!=2)
```

```
True
True
True
```

4.1.1 Exercices

4.1.1.1 Exercice 1

Ecrire un test permettant de vérifier que 3 est bien dans l'intervalle `[2,6[`

```
print(2<=3 and 3<6)
print(2<=3<6)
```

```
True
True
```

4.1.1.2 Exercice 2

Ecrire une fonction `intervalle` renvoyant vrai ou faux selon que l'argument x est à l'intérieur de l'intervalle `[a, b]` sachant que a et b sont deux arguments de la fonction, avec a=0 et b=100 par défaut

```
def intervalle(x, a=0, b=100):
    return x >= a and b >= x

# Une manière de tester automatiquement qu'une fonction
# fait ce qu'on veut :
assert intervalle(12, 5, 15) == True
assert intervalle(19, 5, 15) == False
```

4.2 Les structures conditionnelles

Plusieurs manières de faire des actions qui dépendent d'une condition :

```
SI condition
ALORS on fait machin

SI condition
ALORS on fait machin
SINON on fait truc

SI condition
ALORS on fait machin
SINON SI une autre condition
ALORS on fait truc
SINON SI une encore autre condition
ALORS on fait bidule
SINON on fait chouette
```

Au niveau de la syntaxe, le python ne précise pas le "ALORS".

```
# Exemple complet

age = 56
if age < 6:
    print("Regarde thcoupi")
elif age < 9:
```

```
    print("Regarde power rangers")
elif age < 12:
    print("Joue avec ta PS4")
elif age < 16:
    print("Quitte le téléphone et fais tes devoirs")
elif age < 80:
    print("Abonne toi à Netflix")
else:
    print("Regarde Rex et Derrick")
```

```
Abonne toi à Netflix
```

4.2.1 Exercice 1 : comparaison

Ecrire une fonction `compare(a, b)` qui **affiche** (print)

- “plus petit” si a < b
- “plus grand” si a > b
- “pareil” si a egal b

```
def compare(a,b):
    if a < b:
        print("plus petit")
    elif a > b:
        print("plus grand")
    elif a == b:
        print("pareil")
    # else:
    #    print("pareil")

compare(2,3)
compare(3,2)
compare(3,3)
```

```
plus petit
plus grand
pareil
```

4.2.1.1 Exercice 2 : factorielle

La factorielle $n!$ d’un nombre est le produit de $1 \times 2 \times 3 ... \times n$

Ecrire une fonction factorielle(n) qui renvoie la factorielle du nombre n d’après l’algorithme suivant :

```
SI n est égal à 1
ALORS renvoie 1
SINON renvoie n * factorielle(n-1)
```

```
def factorielle(n):
    if n == 1:
        return 1
    else:
        return n*factorielle(n-1)
```

```python
def factorielle(n):
    if n == 1:
        return 1
    return n*factorielle(n-1)

factorielle(5)
```

```
120
```

4.3 Les boucles (épisode 1)

```
TANT QUE une condition
REPETE des instructions
```

La condition est évaluée au début de chaque itération.

En python :

```python
while CONDITION:
    instruction1
    instruction2
endehorsduwhile
```

Les instructions de la boucle sont exécutées **TANT QUE** la condition est **VRAIE**

```python
# Exemple : compter jusqu'à 10
compteur = 1
while compteur <= 10:
    print(compteur, end=" ")
    compteur = compteur + 1 # compteur += 1
print("Fini...")
```

```
1 2 3 4 5 6 7 8 9 10 Fini...
```

```python
# boucle infinie = pas bien
# mais si on doit l'utiliser, il faut avoir un moyen d'en sortir
# : instruction break
compteur = 0
while True:
    print("yo", end=" > ")
    compteur += 1
    if compteur >= 5:
        break
```

```
yo > yo > yo > yo > yo >
```

L'instruction `continue` permet de passer directement à l'itération suivante

```python
compteur = 1
while compteur <= 10:
    compteur += 1
    # si le reste de la division par 2
    # de compteur est nul = compteur est
    # un nombre pair
    if compteur % 2 == 0:
        print("je suis pair")
        continue
    print("je suis impair")

print("FINI")
```

```
je suis pair
je suis impair
je suis pair
je suis impair
je suis pair
je suis impair
je suis pair
je suis impair
je suis pair
je suis impair
FINI
```

4.3.1 Exercices

4.3.1.1 Exercice 1 : compter de différentes manières

Ecrire plusieurs fonctions `multiples_de(n)` qui compte de 0 jusqu'à 1000 en affichant uniquement les multiples de `n`.

Exemple `multiples_de(3)` : 0, 3, 6, 9, 12...

```python
# Exercice 1

def multiples_de(n):
    i = 0
    while i<=1000:
        # est ce que i est divisible par n
        if i % n == 0:
            print(i, end=" ")
        i += 1

def multiples_de2(n):
    i = 0
    while i<=1000:
        # est ce que i est divisible par n
        print(i, end=" ")
        i += n

def multiples_de3(n):
    i = 0
    while i*n<=1000:
```

```
        # est ce que i est divisible par n
        print(i*n, end=" ")
        i += 1

multiples_de(13)
```

```
0 13 26 39 52 65 78 91 104 117 130 143 156 169 182 195 208 221 234 247
260 273 286 299 312 325 338 351 364 377 390 403 416 429 442 455 468 481
494 507 520 533 546 559 572 585 598 611 624 637 650 663 676 689 702 715
728 741 754 767 780 793 806 819 832 845 858 871 884 897 910 923 936 949
962 975 988
```

```
multiples_de2(13)
```

```
0 13 26 39 52 65 78 91 104 117 130 143 156 169 182 195 208 221 234 247
260 273 286 299 312 325 338 351 364 377 390 403 416 429 442 455 468 481
494 507 520 533 546 559 572 585 598 611 624 637 650 663 676 689 702 715
728 741 754 767 780 793 806 819 832 845 858 871 884 897 910 923 936 949
962 975 988
```

```
multiples_de2(13)
```

```
0 13 26 39 52 65 78 91 104 117 130 143 156 169 182 195 208 221 234 247
260 273 286 299 312 325 338 351 364 377 390 403 416 429 442 455 468 481
494 507 520 533 546 559 572 585 598 611 624 637 650 663 676 689 702 715
728 741 754 767 780 793 806 819 832 845 858 871 884 897 910 923 936 949
962 975 988
```

4.3.1.2 Exercice 2 : factorielle

Calculer la factorielle de `n` avec une boucle `while`

```
def factorielle(n):
    i = 1
    etape = 1
    while i <=n:
        etape *= i  # etape = etape * i
        i += 1
    return etape

# On teste ici le résultat de la fonction
# syntaxe : assert UN TEST QUI DOIT RENVOYER TRUE, message d'erreur


assert factorielle(5) == 1*2*3*4*5, "non ça marche pô"
assert factorielle(1) == 1, "non ça marche pô"
assert factorielle(10) == 1*2*3*4*5*6*7*8*9*10, "non ça marche pô"
```

4.4 Listes et n-uplets

Une liste est une structure de données qui permet de stocker des valeurs les unes à la suite des autres, de manière *séquentielle*.

Chaque élément est associé à une *position* dans la liste. **Attention : les positions ne commencent pas à 1 mais à 0** car les positions sont en fait des **décalages** par rapport au début de la liste.

On écrit une liste entre crochets `[]`. Exemple : `[1, 2, 3]`

```
mes_courses = ["pain","lait","beurre","camembert","masques"]
```

Pour récupérer un élément d'une liste : on utilise la notation `[position]`. Exemple : `mes_courses[2]`

```
# beurre
```

```
print(mes_courses[2])
print([1,2,3][1])
print(["pain","lait","beurre","camembert","masques"][2])
```

```
beurre
2
beurre
```

Pour connaitre la longueur d'une liste : la fonction `len(liste)` renvoie le nombre d'éléments de la liste.

attention le dernier élément d'une liste est situé à la position `len(liste)-1`

On peut récupérer des éléments en partant de la fin de la liste en utilisant une position *négative*

- `[-1]` renvoie le dernier élément
- `[-2]` renvoie l'avant dernier élément

```
assert mes_courses[3] == "camembert"
assert mes_courses[-2] == "camembert"
assert mes_courses[len(mes_courses)-2] == "camembert"
```

Lorsqu'on essaie de récupérer un élément à une position qui n'existe pas , on déclenche une erreur (une exception) **IndexError**

```
mes_courses[999]
```

```
---------------------------------------------------------------------------

IndexError                                Traceback (most recent call last)

<ipython-input-161-a8e2444d3ef1> in <module>
----> 1 mes_courses[999]


IndexError: list index out of range
```

Un n-uplet est une liste **non modifiable**. On l'écrit entre parenthèses, voire sans aucune parenthèse si le contexte le permet

```
alphabet1 = ("A", "B", "C", "D")
alphabet2 = "A", "B", "C", "D"
print(type(alphabet2))
print(type([1,2,3]))
```

```
<class 'tuple'>
<class 'list'>
```

On peut avoir des listes dans des listes

```
morpion = [
    [" ", "x", "o"],
    ["x", " ", "W"],
    ["x", " ", " "]
]
print(morpion[1])
print(morpion[1][2])
```

```
['x', ' ', 'W']
W
```

```
tableau = [[3,4,5,6], [1,2,42]]
assert 4 == tableau[0][1]
assert 42 == tableau[1][2]
assert 6 == tableau[0][3]
assert 1 == tableau[1][0]
```

```
tableau = [1,2, [3,4, [5,6, [7,8]]]]
assert 1 == tableau[0]
assert 2 == tableau[1]
assert 3 == tableau[2][0]
assert 4 == tableau[2][1]
assert 5 == tableau[2][2][0]
assert 6 == tableau[2][2][1]
assert 7 == tableau[2][2][2][0]
assert 8 == tableau[2][2][2][1]
```

4.4.1 Assignation

4.4.1.1 Remplacer dans un tableau

Assigner une valeur dans une liste, c'est écrire la valeur à la position donnée. La position doit déjà exister.

```
tableau = ["A", "B", "W", "D"]
tableau[2] = "C"
print(tableau)
```

```
['A', 'B', 'C', 'D']
```

```
tableau[43]="braaa"
```

```
---------------------------------------------------------------------------

IndexError                                Traceback (most recent call last)

<ipython-input-176-6ead7e28e1e9> in <module>
----> 1 tableau[43]="braaa"


IndexError: list assignment index out of range
```

```
tableau[-1] = "fini"
print(tableau)
```

```
['A', 'B', 'C', 'fini']
```

```
tableau = [["A","B"], ["C", "W"]]
tableau[-1][-1]="D"
print(tableau)

tableau[0]="bouuu"
print(tableau)
```

```
[['A', 'B'], ['C', 'D']]
['bouuu', ['C', 'D']]
```

4.4.1.2 Autres opérations

Pour **ajouter** un élément à la fin d'un tableau : on utilise "append"

```
tableau = [1,2,3]
tableau.append(42)
print(tableau)
```

```
[1, 2, 3, 42]
```

Pour **insérer** un élément dans un tableau : on utilise `insert(position, element)`

```
tableau = [1,2,3]
tableau.insert(1,42)
print(tableau)
```

```
[1, 42, 2, 3]
```

Pour retirer un élément dans un tableau :

- on utilise `del tableau[pos]`
- on utilise `tableau.pop(pos)`

```
tableau = [1,2,3,4,5,6]
print(tableau)
del tableau[3]
print(tableau)
tableau.pop(1)
print(tableau)
```

```
[1, 2, 3, 4, 5, 6]
[1, 2, 3, 5, 6]
[1, 3, 5, 6]
```

4.4.1.3 Les slices / morceaux de tableau

Pour récupérer juste une partie de tableau, on utilise la notation des slices : `tableau[debut:fin:pas]` ; le début est inclus, la fin est exclue

```
tableau = [1,2,3,4,5,6]
print(tableau[2:3])
print(tableau[2:5])
print(tableau[0:6])
print(tableau[0:])
print(tableau[0:-1])
print(tableau[:])
```

```
[3]
[3, 4, 5]
[1, 2, 3, 4, 5, 6]
[1, 2, 3, 4, 5, 6]
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5, 6]
```

```
# Bonus pas duuuu tout au programme
tableau = [1,2,3,4,5,6]
print(tableau[2:3:2])
print(tableau[2:5:2])
print(tableau[0:6:2])
print(tableau[0::2])
print(tableau[0:-1:2])
print(tableau[::2])
```

```
[3]
[3, 5]
[1, 3, 5]
[1, 3, 5]
[1, 3, 5]
[1, 3, 5]
```

```
# Pour la culture : pour renverser un tableau
tableau = [1,2,3,4]
print(tableau[::-1])
```

```
[4, 3, 2, 1]
```

Pour tester si un élément est **dans** un tableau : l'opérateur `in`

```
tableau = [1,2,3,4]
print(12 in tableau)
print(3 in tableau)
```

```
False
True
```

Pour trier un tableau, on peut utiliser la méthode `tableau.sort` ou la fonction `sorted`

```
tableau = [2,4,1,3]
tableau.sort() # trié en place
print(tableau)

tableau = [2,4,1,3]
tableautrie = sorted(tableau) # renvoie une copie du tableau
print(tableautrie, tableau)
```

```
[1, 2, 3, 4]
[1, 2, 3, 4] [2, 4, 1, 3]
```

4.4.1.4 Exercice 1 - morpion python

Créer un tableau `morpion` qui contient 3 lignes de 3 cases. Lorsque une case est vide, la valeur vaut 0, lorsqu'elle a été jouée par le J1 elle vaut 1, et elle vaut 2 pour le J2.

Jouer une partie de morpion en utilisant la notation `morpion[][]=`

```
morpion = [[0,0,0],
           [0,0,0],
           [0,0,0]]
morpion[1][1] = 1
morpion[0][0] = 2
morpion[2][0] = 1
morpion[0][2] = 2
morpion[0][1] = 1
morpion[2][1] = 2
print(morpion)
```

```
[[2, 1, 2], [0, 1, 0], [1, 2, 0]]
```

4.5 Les boucles for

Les listes sont très utiles à partir du moment où on peut accéder à tous leurs éléments de manière itérative.

Pour accéder à tous les éléments d'une liste, la structure est la suivante :

```
for variable_destination in liste_source:
    instruction
    instruction


maliste = ["un", "deux", "trois", "quatre"]

for truc in maliste:
    print("Je sais compter jusqu'à", truc)
print("fini")
```

```
Je sais compter jusqu'à un
Je sais compter jusqu'à deux
Je sais compter jusqu'à trois
Je sais compter jusqu'à quatre
fini
```

On peut avoir besoin de connaitre non seulement la valeur, mais aussi la position dans la liste à chaque itération. Dans ce cas on peut **énumérer** la liste.

```
maliste = ["un", "deux", "trois", "quatre"]
for machin, truc in enumerate(maliste):
    print(machin, truc)
```

```
0 un
1 deux
2 trois
3 quatre
```

Pour les boucles for, on a aussi les instructions `break` et `continue`

```
maliste = ["un", "deux", "trois", "quatre"]
for pos, val in enumerate(maliste):
    # dans la première varialbe : la position
    # dans la deuxième variable : la valeur
    print("Position ", pos)
    if val == "deux":
        continue
    print("Valeur", val)
    if val == "trois":
        break
```

```
Position  0
Valeur un
Position  1
Position  2
Valeur trois
```

Une fonction très pratique : la fonction `range(debut, fin, pas)`, ou `range(debut, fin)` ou `range(fin)` : elle renvoie une liste de nombre, de début inclus, jusqu'à fin **non inclus**, avec un pas

```
for i in range(0,10,3):
    print(i, end=", ")
```

```
0, 3, 6, 9,
```

```
for i in range(10):
    print(i, end=", ")
```

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

```
for i in range(2,10):
    print(i, end=", ")
```

```
2, 3, 4, 5, 6, 7, 8, 9,
```

4.5.1 Exercices

4.5.1.1 Exercice 1 : Ajouter à une liste

Ecrire une fonction `bonus(notes, valeur)` qui ajoute `valeur` à chaque note de la liste `notes` et calcule la moyenne de l'élève.

Exemple : notes : 12, 14, 13 / valeur : 3 moyenne

$$\frac{(12+3)+(14+3)+(13+3)}{3}$$

```
def bonus(notes, valeur):
    total = 0 # sert d'accumulateur
    for n in notes:
        total += n
        total += valeur
    return total/len(notes)

bonus([12,14,13], 3)
```

```
16.0
```

4.5.1.2 Exercice 1 bis

Ecrire une fonction `moyenne(notes)` qui renvoie la moyenne des notes données dans la liste `notes`

```
def moyenne(notes):
    tot = 0
    for note in notes:
        tot = tot + note
    return tot/len(notes)

moyenne([10, 12, 14, 16])
```

```
13.0
```

4.5.1.3 Exercice 1 ter

Ecrire une fonction `mini(notes)` qui renvoie la plus petite de toutes les notes données dans `notes`

Ecrire une fonction `maxi(notes)` qui renvoie la plus grande de toutes les notes données dans `notes`

```
def mini(notes):
    derniermini = 100000000
    for note in notes:
        if note < derniermini:
            derniermini = note
    return derniermini

print(mini([14, 11, 13]))

def maxi(notes):
    derniermaxi = -100000000
    for note in notes:
```

```
        if note > derniermaxi:
            derniermaxi = note
    return derniermaxi

print(maxi([12, 14, 13]))
```

```
11
14
```

4.5.1.4 Exercice 2 : Moyenne coefficientée

Ecrire une fonction `moyenne(notes)` qui calcule la moyenne d'une liste de notes avec leur barème et leur coefficient (chaque note est donnée sous forme `[12, 20, 1]` : note, barème, coefficient

```
def moyenne(notes):
    A = 0
    B = 0
    for n in notes:
        A = A + n[0]*n[2]
        B = B + n[1]*n[2]
    return A*20/B
moyenne([[3, 5, 1], [2, 10, 3], [16, 20, 2]])
```

```
10.933333333333334
```

4.5.1.5 Exercice 3 : Morpion (v2)

Faire une fonction `verif(morpion)` qui vérifie si une grille de morpion est gagnante ou non, en utilisant des boucles for et des boucles whiles

```
def verif(morpion):
    # morpion[ligne][colonne]
    # Est-ce que le joueur 1 a gagné ?
    for ligne in range(3):
        for colonne in range(3):
            print("l",ligne, "c",colonne)

verif([[0,0,0], [0,0,0], [1,1,1]])
```

```
l 0 c 0
l 0 c 1
l 0 c 2
l 1 c 0
l 1 c 1
l 1 c 2
l 2 c 0
l 2 c 1
l 2 c 2
```

4.5.1.6 Exercice 4 : Crible d'Eratosthène

Programmer le crible d'Eratosthène d'après l'algorithme donné sur : https ://fr.wikipedia.org/wiki/Crible_d%27%C3%

4.6 Arguments par référence ou par valeur

Les listes sont représentées par une “adresse” (une référence) en mémoire. Lorsqu’on assigne une liste à une variable, la variable ne contient qu’une “référence” vers cette liste. Si la liste est assignée à plusieurs variables, elle n’existe réellement qu’une seule fois.

Si on la modifie dans une des variables, elle sera modifiée partout.

```
maliste = [0,1,2,3]
taliste = maliste
taliste[0] = 42
print(maliste)
print(taliste)
```

```
[42, 1, 2, 3]
[42, 1, 2, 3]
```

```
def mafonction(toto):
    # toto: argument passé par référence
    toto[1]=42
mafonction(maliste)
print(maliste)
```

```
[42, 42, 2, 3]
```

```
# solution
# Pour une liste: on peut "copier la liste" (solution correcte..)
maliste = [0,1,2,3]
taliste = maliste[:]
taliste[0] = 42
print(maliste)
print(taliste)
```

```
[0, 1, 2, 3]
[42, 1, 2, 3]
```

```
## Contre exemple
# solution
# Pour une liste: on peut "copier la liste" (solution correcte..)
maliste = [[0,1,2,3],[4,5,6,7]]
taliste = maliste[:]
taliste[0][0] = 42
print(maliste)
print(taliste)
```

```
[[42, 1, 2, 3], [4, 5, 6, 7]]
[[42, 1, 2, 3], [4, 5, 6, 7]]
```

```
# Solution propre
from copy import deepcopy
maliste = [[0,1,2,3],[4,5,6,7]]
taliste = deepcopy(maliste)
taliste[0][0] = 42
print(maliste)
print(taliste)
```

```
[[0, 1, 2, 3], [4, 5, 6, 7]]
[[42, 1, 2, 3], [4, 5, 6, 7]]
```

4.7 Les “compréhensions” de liste

On se retrouve souvent à faire des opérations du genre :

```
carres = []
for nombre in range(1,11):
    carres.append(nombre**2)
print(carres)
```

on peut le réécrire en “plus court” et moins compréhensible :

```
carres = [nombre ** 2 for nombre in range(1,11)]
```

On peut même mixer des conditions dans les compréhensions de liste, mais attention aux excès qui enlèvent de la lisibilité au programme

```
carres_pairs = [nombre**2 for nombre in range(1,20) if nombre % 2 == 0 ]
print(carres_pairs)
```

```
[4, 16, 36, 64, 100, 144, 196, 256, 324]
```

4.8 Les dictionnaires

Les listes permettent de stocker les objets les uns à la suite des autres, les dictionnaires permettent d’associer des clefs avec des valeurs.

exemple : un contact doit stocker nom, prénom, adresse, tel, email…

```
moi = {"nom": "BARBIER", "prenom": "Jean-Matthieu",
       "mail": "jm.barbier@solidev.net", "clef": "valeur"}
```

Les clefs peuvent être des chaînes de caractères (le plus souvent), mais aussi (presque) n’importe quel type de données. Idem pour les valeurs, qui peuvent prendre n’importe quel type.

```
moi = {"nom": "BARBIER", "prenom": "Jean-Matthieu",
       "mail": "jm.barbier@solidev.net",
       "adresse": { "numero": 4, "rue": "clos d'ivry",
                    "ville": "houlbec", "cp": "27120"}}
print(moi)
print(moi["nom"])
print(moi["adresse"])
print(moi["adresse"]["cp"])
```

```
{'nom': 'BARBIER', 'prenom': 'Jean-Matthieu',
 'mail': 'jm.barbier@solidev.net',
 'adresse': {
    'numero': 4,
    'rue': "clos d'ivry",
    'ville': 'houlbec',
    'cp': '27120'
```

```
  }
}
BARBIER
{'numero': 4, 'rue': "clos d'ivry", 'ville': 'houlbec', 'cp': '27120'}
27120
```

Pour rajouter un élément dans un dictionnaire, il suffit d'assigner une valeur à une clef.

- si la clef n'existe pas dans le dictionnaire, elle est créée
- si elle existe, la valeur correspondante est modifiée

```
moi = {}
moi["mail"] = "jm.barbier@solidev.net"
print(moi)
moi["nom"] = "BARBIER"
print(moi)
moi["prenom"] = "Toto"
print(moi)
moi["prenom"] = "Jean-Matthieu"
print(moi)
```

```
{'mail': 'jm.barbier@solidev.net'}
{'mail': 'jm.barbier@solidev.net', 'nom': 'BARBIER'}
{'mail': 'jm.barbier@solidev.net', 'nom': 'BARBIER', 'prenom': 'Toto'}
{'mail': 'jm.barbier@solidev.net', 'nom': 'BARBIER', 'prenom': 'Jean-Matthieu'}
```

```
# Pour supprimer une valeur
moi["portnawak"] = 42
print(moi)
del moi["portnawak"]
print(moi)
```

```
{'mail': 'jm.barbier@solidev.net', 'nom': 'BARBIER',
 'prenom': 'Jean-Matthieu', 'portnawak': 42}
{'mail': 'jm.barbier@solidev.net', 'nom': 'BARBIER',
 'prenom': 'Jean-Matthieu'}
```

On peut parcourir un tableau de 3 manières :

- en utilisant ses clefs
- en utilisant ses valeur
- en utilisant clefs et valeurs

```
# En utilisant les clefs
for aazeaze in moi:
    print(aazeaze)
```

```
mail
nom
prenom
```

```
# En utilisant les valeurs
for sdsdf in moi.values():
    print(sdsdf)
```

```
jm.barbier@solidev.net
BARBIER
Jean-Matthieu

# En utilisant les clefs et les valeurs
for clef, valeur in moi.items():
    print(clef, valeur)

mail jm.barbier@solidev.net
nom BARBIER
prenom Jean-Matthieu
```

4.9 Divers

```
# Imprimer joliment
moi = {"nom": "BARBIER", "prenom": "Jean-Matthieu",
       "mail": "jm.barbier@solidev.net",
       "adresse": { "numéro": 4, "rue": "clos d'évry",
                    "ville": "houlbec", "cp": "27120"}}
import pprint
pprint.pprint(moi)

{'adresse': {'cp': '27120',
             'numéro': 4,
             'rue': "clos d'évry",
             'ville': 'houlbec'},
 'mail': 'jm.barbier@solidev.net',
 'nom': 'BARBIER',
 'prenom': 'Jean-Matthieu'}

import yaml
print(yaml.dump(moi, allow_unicode=True))

adresse:
  cp: '27120'
  numéro: 4
  rue: clos d'évry
  ville: houlbec
mail: jm.barbier@solidev.net
nom: BARBIER
prenom: Jean-Matthieu
```

Intéractivité : fonction `input(message) -> str`

```
reponse = input("Quel est votre nom ?")
print("Votre nom est : ", reponse)

Quel est votre nom ? BARBIER


Votre nom est :  BARBIER
```

4.9.0.1 Exercice 1 : Création d'une fiche de carnet d'adresse

Créer une fonction qui demande à l'utilisateur son nom, son prénom et son email, et qui renvoie les infos dans un dictionnaire

4.9.0.2 Exercice 2 : Création d'un carnet d'adresse

Créer une fonction qui demande à l'utilisateur combien de personnes il veut entrer, et qui ensuite demande autant de fois que désiré le nom, prénom et email à ajouter, et qui stocke ces fiches stockées à la suite les unes des autres dans une liste

4.9.0.3 Exercice 3 : Recherche dans un carnet d'adresse

Créer une fonction prenant comme arguments un carnet d'adresses et un nom, et qui renvoie la fiche dont le nom correspond si elle existe, et ne renvoie rien si rien n'est trouvé.