Numériques Sciences Informatiques

Classe de Première

-Premier trimestre-

professeurgibaud.ovh

Table des matières

Chapitre 1
Algorithmique et programmation 1

I. Introduction

1. Définition

À l'ère du numérique, des smartphones et des réseaux sociaux, qui n'a pas entendu parler des algorithmes ? Et pour cause, ils sont omniprésents dans notre quotidien. Ils sont à la base de nos applis et de nos ordinateurs, de la télécommunication et d'internet. Sans eux, la technologie contemporaine ne pourrait exister.

Pourtant, si tout le monde a entendu parler des algorithmes, très peu sont capables de dire ce qu'est un algorithme. Cette méconnaissance entraîne tantôt de la fascination, tantôt de la crainte, alors que la notion d'algorithme n'est peut-être pas aussi complexe qu'on pourrait le penser. D'ailleurs, beaucoup de personnes utilisent directement des algorithmes sans le savoir. Peut-être seriez vous capable de citer des exemples d'algorithmes que vous utilisez vous-même régulièrement ?

Si les algorithmes sont à la base de la programmation informatique, et donc du fonctionnement de nos applis, leur première utilisation est bien antérieure à l'invention de l'ordinateur. D'ailleurs, celle-ci précède même la vie du mathématicien persan Muhammad Ibn Mūsā al-Khuwārizmī (circa 780-850) dont le nom sous sa forme latinisée, Algorizmi, est à l'origine du mot algorithme. En effet, la plus ancienne trace retrouvée de la description d'un algorithme date de plus de 4500 ans. Il s'agit d'une tablette d'argile sumérienne qui détaille un algorithme de division.

Définition 1.1.
Un ***algorithme*** est une séquence ***finie*** et ***univoque***[a] d'instructions permettant de résoudre une classe de problèmes[b].

a. Qui ne souffre pas d'ambiguïté.
b. N'importe quel problème d'un certain type.

Exemple 1.1.
L'addition avec retenue est un algorithme que vous connaissez et utilisez depuis le primaire ! Il permet d'additionner deux entiers positifs quels qu'ils soient. ■

2. Le code

Les algorithmes sont importants en mathématiques comme en informatique parce qu'ils définissent des protocoles très précis permettant d'accomplir une tâche particulière autant de fois que nécessaire, sans jamais se tromper, ni avoir à se demander comment faire : il suffit de suivre les instructions. C'est un peu comme une recette de

cuisine mais pour laquelle on est certain que le plat sera réussi à chaque fois si l'on suit correctement les étapes. Pour garantir cette réussite systématique, l'algorithme doit être présenté dans un langage simplifié qui ne tolère pas les double-sens ou les approximations : le ***code***. C'est en cela qu'un algorithme diffère d'une recette de cuisine ou d'un mode d'emploi qui sont eux écrits en langage naturel.

Il existe de très nombreux langages différents pour écrire du code, notamment en informatique, où l'on parle de ***langage de programmation***. Lorsqu'un algorithme est écrit dans un langage de programmation particulier, on dit qu'il est ***implémenté*** dans ce langage (on utilise aussi le terme ***implémentation***). Dans ce cours, nous utiliserons le langage de programmation Python qui est au programme du lycée. Nous utiliserons également le ***pseudo-code*** qui est un langage intermédiaire entre le langage naturel et le code à proprement parler.

▪ Exemple 1.2.
Voici un exemple d'une séquence d'instructions en pseudo-code et de son implémentation en Python.

```
Pour k allant de 0 à 9
    Faire
    |Afficher k
Fin pour
```

```
for k in range(10):
    print(k)
```

Pour bien distinguer entre les deux, la séquence en pseudo-code et la séquence en Python sont présentées avec des styles graphiques différents :
— le pseudo-code dans un rectangle blanc à bords droits ;
— le code Python dans un rectangle gris à bords ronds.
Cette convention sera maintenue tout au long de ce cours afin d'éviter toute confusion entre les deux. ▪

3. Objectifs du chapitre

Dans ce chapitre, vous apprendrez les principes de base pour l'élaboration d'un algorithme, son écriture en pseudo-code ainsi que son implémentation en Python. Vous serez également amenés à écrire vos propres algorithmes. Enfin, la notion de simulation sera introduite en fin de chapitre 2.

Quelques exercices très simples ont été intégrées dans le cours, et vous êtes fortement encouragées à vérifier les réponses de ces exercices par vous mêmes, en exécutant les instructions correspondantes sur votre ordinateur. Ainsi vous pourrez pleinement appréhender les nouveaux objets que vous découvrirez au fur et à mesure de votre lecture de ce cours.

4. Exécuter un code Python en Console

Pour exécuter un code Python, il faut d'abord installer Python sur votre machine. Aujourd'hui, on peut installer Python sur un ordinateur, une tablette, un téléphone portable et même une calculatrice programmable. En fonction de votre machine et de votre système d'exploitation (Ubuntu/Linux, Windows, Mac OS X, ...), il existe de nombreuses façons d'installer Python et de nombreux ***environnements de développement***

différents qui permettent de visualiser le code que l'on programme avant de l'exécuter. Si vous vous y connaissez déjà en programmation, vous êtes libres de choisir l'environnement qui vous convient le mieux. Pour les autres, nous vous conseillons l'utilisation de l'environnement de développement Pyzo installé sur votre ordinateur via la distribution de base Python3 [1].

Pour installer Python, vous pouvez télécharger l'installateur à partir de la page web : https://www.python.org/downloads/. En principe, votre système d'exploitation sera reconnu automatiquement par le site internet, si ce n'est pas le cas choisissez l'onglet correspondant à votre système d'exploitation (Windows, macOs ou Linux). Ensuite, cliquez sur le lien de téléchargement correspondant à la version de Python 3 la plus récente. Une fois l'installateur téléchargé, exécutez le fichier et suivez les instructions correspondantes. Il vous suffira ensuite de lancer l'application Pyzo pour programmer et exécuter du code Python. Vous pourrez télécharger Pyzo à partir de la page web : https://pyzo.org/start.html.

Pour une prise en main rapide de l'environnement Pyzo, vous pouvez visionner l'un des nombreux tutoriels disponibles sur internet.

II. Variables et instructions élémentaires

1. Types de variables

En informatique, pour stocker les données, on utilise des ***variables***. Une variable est définie par un nom qui permet de l'identifier de manière unique. Ce nom permet à l'ordinateur de retrouver la donnée contenue dans sa mémoire qui correspond à la ***valeur*** de la variable. Chaque variable est caractérisée par un ***type*** qui correspond au type de donnée qu'elle contient. Il existe de nombreux types de variables différents en informatique mais nous n'en étudierons que quatre à ce stade.

Type booléen

Une variable de ***type booléen*** (ou ***variable booléenne***) est une variable qui prend l'une des deux valeurs : **vrai** ou **faux**. Le terme booléen est un hommage au grand mathématicien et logicien anglais George Boole (1815-1864) considéré comme le père de la logique moderne. En Python, le type booléen se note «`bool`» et les valeurs associées «`True`» et «`False`» respectivement.

Type entier

Le ***type entier*** désigne, comme son nom l'indique, des variables entières (qui appartiennent donc à l'ensemble $\mathbb{N}$) [2]. En Python, le type entier se note «`int`» (contraction de «*integer*» qui signifie entier en anglais).

1. Une distribution contient à la fois Python et un ensemble de bibliothèques Python (voir section 2.)

2. Attention toutefois, un ordinateur est un système fini et ne peut donc stocker des variables sans limite de taille. En pratique, la taille maximale d'une variable de type entière varie d'un langage de programmation à un autre.

Type flottant

Une variable de ***type flottant*** (ou ***variable flottante***) contient un nombre qui s'écrit avec un nombre fini de chiffres après la virgule[3]. Le type flottant se note «`float`» en Python (qui signifie flotter en anglais). Attention, la virgule est remplacée par un point dans l'écriture anglo-saxonne des nombres et donc, en particulier, dans le langage Python.

Type chaîne de caractères

Une variable de ***type chaîne de caractères*** contient du texte. Le terme ***caractère*** désigne les caractères typographiques que sont les lettres (minuscules ou majuscules, avec ou sans accents), les chiffres, les signes de ponctuation, etc. Dans le langage Python, ce type est noté «`str`» (contraction de «*string*» qui signifie fil ou enchaînement en anglais). Les chaînes de caractères s'écrivent en Python par du texte entre guillemets simples (par exemple : `'Ceci est un string en Python.'`) ou guillemets doubles (par exemple : `"Ceci est également un string en Python!"`).

2. Affectation

Définition 1.2.
L'***affectation*** d'une valeur à une variable est l'action de donner une valeur à cette variable.

Pour indiquer que l'on affecte la valeur `x` à la variable `a` en pseudo-code, on note :

```
a ← x
```

En Python, on note :

```
a = x
```

Attention, le signe « = » n'a donc pas le même sens en Python qu'en mathématiques ! En particulier, en Python, « `a = x` » n'a pas le même sens que « `x = a` ».

En programmation, il est possible d'affecter et de ***réaffecter*** des variables. Dans ce cas, c'est la dernière affectation qui prévaut sur les autres. On peut également mettre à jour une variable en lui réaffectant une valeur qui dépend d'elle-même.

▪ **Exercice 1.1.**
Le code Python affecte plusieurs valeurs successives à `a` en réalisant plusieurs affichages (la fonction «`print`» permet de réaliser un affichage en Python).

3. L'adjectif flottant qualifie en réalité la virgule dont la position n'est pas fixée (et peut donc «*flotter*») en mémoire pour ce type de variables.

```
a = -3
a = 1
print(a)
a = a+1
print(a)
```

Que va renvoyer la Console Python lorsque l'on exécute ce code ? ■

3. Opérations élémentaires

À chaque type est associé un ensemble d'opérations. Vous trouverez ici la liste des opérations les plus élémentaires associées aux quatre types décrits précédemment. Tous les types admettent également deux opérations universelles qui sont l'opération de ***test d'égalité*** et l'opération de ***test de différence***. L'opération de test d'égalité, notée « = » en pseudo-code et « == » en Python, permet de vérifier si deux variables sont égales. L'opération de test de différence, notée « ≠ » en pseudo-code et « != » en Python, permet de vérifier si deux variables sont différentes.

■ **Exercice 1.2.**
Le code Python suivant affecte la valeur 2 à la variable `a` et la valeur −3 à la variable `b`.

```
a = 2
b = -3
print(a == b)
print(a != b)
```

Que va renvoyer la Console Python lorsque l'on exécute ce code ? ■

Opérations sur les flottants

Pour les flottants, on peut utiliser les opérations classiques entre deux nombres : l'addition, la soustraction, la multiplication, la division, l'élévation à la puissance ; ainsi que les comparateurs d'ordre : inférieur ou égal, inférieur strict, supérieur ou égal, supérieur strict. La liste des symboles utilisés en Python figure dans le tableau ci-dessous :

Opération mathématique	Expression Python
$a+b$	`a + b`
$a-b$	`a - b`
$a\times b$	`a * b`
$a\div b$	`a / b`
a^b	`a ** b`
$a\leq b$	`a <= b`
$a<b$	`a < b`
$a\geq b$	`a >= b`
$a>b$	`a > b`

Opérations sur les entiers

Parmi les opérations standards entre deux entiers, on retrouve toutes les opérations décrites pour les flottants auxquelles on peut ajouter le quotient et le reste par la division euclidienne (c'est-à-dire la division entière).

Opération mathématique	Expression Python
Quotient de la division entière de a par b	`a // b`
Reste de la division entière de a par b	`a % b`

▪ Exercice 1.3.
On considère le code Python suivant.

```
a = 3 - 1
b = 1 + 2
c = a ** 3
d = c / 5
e = c // 5
f = (a <= b)
```

Que valent les variables `a`, `b`, `c`, `d`, `e` et `f` ? ▪

Opérations sur les booléens

Les opérations standards sur les booléens sont :
— la ***négation*** associée au mot-clé «**non**» ;
— la ***conjonction*** associée au mot-clé «**et**» ;
— la ***disjonction*** associée au mot-clé «**ou**».

Elles sont implémentées en Python à l'aide des opérateurs «`not`», «`and`» et «`or`» respectivement. Si `a` et `b` sont deux variables booléennes, on a :
— **non**`(a)` vaut **vrai** si et seulement si `a` vaut **faux** ;
— `a` **et** `b` vaut **vrai** si et seulement si les deux variables `a` et `b` sont égales à **vrai** ;
— `a` **ou** `b` vaut **vrai** si et seulement si l'une au moins des deux variables `a` ou `b` vaut **vrai**.

▪ Exercice 1.4.
Analysez le code Python qui suit.

```
a = (0 < 1) or (0 > 2)
b = not (1 < 2)
c = a and b
```

Quelles valeurs prennent les variables booléennes `a`, `b` et `c` ? ▪

Opérations sur les chaînes de caractère

Il existe un certain nombre d'opérations sur les chaînes de caractère en Python mais nous n'en considérons qu'une ici : la ***concaténation***. La concaténation permet de créer une nouvelle chaîne de caractère à partir de deux chaînes de caractère `a` et `b`, en mettant les caractères de `b` à la suite des caractères de `a`. La concaténation de `a` et `b` en Python se note `a + b`.

▪ **Exercice 1.5.**
On considère le code Python suivant.

```
a = "math"
b = a + "ematiques"
c = "ha"
c = c + c
```

Que valent les variables `a`, `b` et `c` ? ▪

4. Instructions conditionnelles

R Un père dit à son enfant : "**Si** tu finis tes légumes, **Alors** tu auras un dessert". L'enfant ne finit pas ses légumes mais le père donne quand même à son enfant un dessert. Le père a-t-il menti ?

La réponse est non. Et ce n'est pas pour des raisons d'éducation ou de morale. Le père a dit ce qu'il ferait si son enfant finit son dessert mais il n'a rien dit si il ne finissait pas son dessert.

Dans un algorithme, on peut choisir qu'une instruction ne s'exécute que si certaines conditions sont remplies. En pseudo-code, on peut utiliser les mots-clés **si** et **alors** pour indiquer une condition et la séquence d'instructions à exécuter si la condition est remplie. On peut également utiliser le mot-clé **Fin si** pour indiquer la fin de la séquence d'instructions conditionnelles. Une instruction conditionnelle peut donc s'écrire en pseudo-code sous la forme suivante :

```
Si la condition est vraie
    Alors faire
    |instructions
Fin Si
```

En Python, on utilise la commande «`if`» associé à l'utilisation des deux points «`:`» et d'une ***indentation*** (décalage vers la droite des lignes d'instructions [4]), selon la forme suivante :

```
if condition :
    instructions
```

Attention, les règles pour les deux points et l'indentation ne sont pas facultatives et le code ne fonctionnera pas correctement si elles ne sont pas respectées. Par ailleurs, l'indentation doit courir sur l'ensemble des instructions couvertes par la condition (donc éventuellement sur plusieurs lignes).

4. L'indentation en Python peut correspondre à une tabulation ou à un nombre fixe d'espaces. Toutefois, il est important qu'une même indentation soit utilisée pour un même groupe d'instructions. Dans l'usage, une indentation formée de 4 espaces est priviligiée.

On peut également différencier entre un certain nombre de cas en utilisant les mots-clés « **sinon si** » et « **sinon** » en pseudo-code ou les commandes « `elif` » et « `else` » en Python, selon le schéma qui suit :

```
Si la condition1 est vraie
    Alors faire
    |instructions1
Si la condition2 est vraie
    Alors faire
    |instructions2
Si la condition3 est vraie
    Alors faire
    |instructions3
Sinon
    Faire
    |instructions4
Fin si
```

```
if condition1 :
    instructions1
elif condition2 :
    instruction2
elif condition3 :
    instructions3
else :
    instructions4
```

▪ Exemple 1.3.
Deux joueurs s'affrontent aux dés. Le joueur qui obtient la plus grande valeur a gagné. Le programme suivant récupère la valeur obtenue par chacun des joueurs puis annonce qui a gagné la partie (on utilise pour cela les fonctions «`int`» et «`input`» de Python qui seront présentées en détails lors de la semaine 2).

```
d1 = int(input("Entrez la valeur obtenue par le joueur 1."))
d2 = int(input("Entrez la valeur obtenue par le joueur 2."))
if d1 > d2 :
    print("Le joueur 1 a gagne.")
elif d2 > d1 :
    print("Le joueur 2 a gagne.")
else:
    print("Match nul!")
```

▪

5. Boucles

En algorithmique, les ***boucles*** permettent de répéter une séquence d'instructions sans avoir à réécrire la séquence. On distingue deux formes de boucles :

— les ***boucles bornées*** «**pour**», pour lesquelles la répétition de la séquence d'instruction correspond au parcourt de tous les éléments d'un ensemble fini par une variable ;

— les ***boucles non bornées*** «**tant que**», pour lesquelles la répétition de la séquence d'instruction est soumise à condition de répétition à chaque tour de la boucle.

Les boucles bornées sont appelées ainsi parce qu'elles s'arrêtent quand on arrive au bout de l'ensemble fini correspondant. Les boucles non bornées sont appelées ainsi parce qu'elles ne s'arrêtent que si la condition de répétition est fausse.

La boucle bornée «pour»

Dans ce cours, on ne considère que les boucles «**pour**» où la répétition de la séquence correspond au parcours par une variable d'un intervalle d'entiers (par exemple, $\{4,5,6,7,8,9\}$) et, le plus souvent, un intervalle d'entiers débutant en 0 (par exemple, $\{0,1,2,3,4,5,\}$). En pseudo-code, on pourra utiliser le mot-clé «**pour**» en début de boucle et indiquer la fin de cette boucle par un «**fin pour**».

```
1 Pour i allant de 1 à n
2     Faire
3     |instructions
4 Fin pour
```

En Python, on utilise la commande «`for`», associée à une instruction de la forme «`i in range(n)`» qui signifie que `i` parcourt l'intervalle $\{0,1,2,...,n-1\}$, et suivie d'un deux points. Une indentation permet ensuite d'indiquer les instructions correspondant à la séquence d'instructions à répéter.

```
1 for i in range(n) :
2     instructions
```

La boucle non bornée «tant que»

Dans la boucle «**tant que**», la boucle se répète tant qu'une condition de répétition est vraie. En pseudo-code, on peut indiquer le commencement d'une telle boucle par «**tant que**» et la fin des instructions de la boucle par «**fin tant que**».

```
1 Tant que la condition est vraie
2     Faire
3     |instructions
4 Fin tant que
```

En Python, la boucle «**tant que**» passe par la commande «`while`», associée à une condition, et suivie d'un deux points. Comme usuellement en Python, une indentation permet d'indiquer les instructions correspondant à la séquence d'instructions à répéter.

```
1 while condition :
2     instructions
```

Attention, si la condition est toujours vérifiée, la boucle ne s'arrêtera jamais. On dit que le programme tourne en ***boucle infinie***. Généralement, ce n'est pas souhaitable. Il faut donc s'assurer que la condition de répétition puisse passer de la valeur **vrai** à la valeur **faux** lors de la séquence d'instructions de la boucle.

▪ **Exercice 1.6.**
Dans le programme Python qui suit, l'une des deux boucles tourne en boucle infinie. Laquelle ?

```
1 i = 0
2 while i >= 0 :
3     i = i + 1
4     print(i)
```

```
1 i = 10
2 while i >= 0 :
3     i = i - 1
4     print(i)
```

Décrire pas-à-pas les opérations qui sont effectuées lors de l'exécution de chacune de ces boucles. ▪

On remarquera que dans l'exercice précédent, la variable `i` est affectée à une valeur avant le début de la boucle. Ceci est un schéma classique en programmation qui s'appelle l'***initialisation***. On dit, par exemple, que la variable `i` est initialisée à 0 avant la première boucle et initialisée à 10 avant la deuxième boucle.

Exercices non à soumettre

■ **Exercice 1.7.**
On considère les trois séquences d'instructions suivantes en pseudo-code.

```
1 x ← x+1
2 b ← x²
3 a ← x-1
4 c ← a²
5 x ← b-c
```

```
1 x ← x-1
2 a ← x²
3 x ← x+2
4 b ← x²
5 x ← a+b
```

```
1 a ← x-1
2 b ← a²
3 c ← x+1
4 d ← c²
5 x ← b+d
```

1. On suppose que la variable `x` contient la valeur 2 avant l'exécution de la séquence. Dans chacun des cas, déterminez la valeur dans `x` après exécution de la séquence.
2. Identifiez la ou les séquences pour lesquelles, si `x` est initialisée à `a` avant l'exécution de la séquence, alors `x` contient $(a-1)^2 + (a+1)^2$ après exécution de la séquence.

■

■ **Exercice 1.8.**
Aladin propose le code Python suivant pour échanger la valeur de deux variables `a` et `b` :

```
1 a = 42
2 b = 23
3 a = b
4 b = a
```

1. Expliquer pourquoi le code fourni par Aladin ne remplit pas ses objectifs.
2. Proposer une autre solution qui échange effectivement les deux variables.

■

■ **Exercice 1.9.**
On considère le code Python suivant :

```
x = int(input("Veuillez saisir un nombre entier."))
if x<100:
    print("Votre nombre est bien faible!")
else:
    print("Quel beau nombre!")
print("Au revoir!")
```

1. Que fait le programme si l'utilisateur rentre la valeur 15 ?
2. Et la valeur 1515 ?

■

■ **Exercice 1.10.**
À la fête foraine, l'accès aux montagnes russes est réservé aux personnes mesurant au moins 1m20 (compris) et au plus 2m10 (compris). Faites un programme Python qui demande à un utilisateur de rentrer sa taille en mètres et l'informe, en fonction de sa réponse, s'il :
— peut monter dans l'attraction ;
— est trop petit ;
— est trop grand.

■

■ **Exercice 1.11.**
Algorithme : Équation d'une droite passant par deux points donnés. L'objectif ici est de développer un programme Python qui renvoie l'équation d'une droite passant par deux points $A(x_A ; y_A)$ et $B(x_B ; y_B)$ dont les coordonnées dans un repère sont données.

1. Que se passe-t-il si $A = B$?
2. Quelle est l'équation de la droite dans le cas où $x_A = x_B$ et $y_A \neq y_B$?
3. Donnez l'équation de la droite dans le cas où $x_A \neq x_B$.
4. Utilisez une instruction conditionnelle pour réaliser un programme Python qui affiche l'équation de la droite passant par A et par B.

■

■ **Exercice 1.12.**
On considère le code Python suivant.

```
a = "A"
for k in range(2):
    a = a + "ha"
a = a +"!"
print(a)
```

1. Quelle valeur est contenu dans la variable a après exécution du programme.
2. Comment modifier le code pour produire un fou rire.

■

■ **Exercice 1.13.**
Pour chacun des programmes suivants, dire quelle valeur prend la variable d après exécution du programme.

```
a = 1
b = 1
c = 1
d = 0
for k in range(2):
    a = a+b
    b = b+a
    c = c+b
    d = d+c
```

```
a = 1
b = 1
c = 1
d = 0
for k in range(2):
    a = a+b
    b = b+a
    c = c+b
d = d+c
```

```
a = 1
b = 1
c = 1
d = 0
for k in range(2):
    a = a+b
    b = b+a
c = c+b
d = d+c
```

```
a = 1
b = 1
c = 1
d = 0
for k in range(2):
     a = a+b
b = b+a
c = c+b
d = d+c
```

■

■ **Exercice 1.14.**
On considère la séquence d'instructions en pseudo-code qui suit.

```
1  x ← 3
2  y ← 11
3  k ← 1
4  Tant que x < y
5      Faire
6      | x ← 3x + 2
7      | y ← 2y + 1
8      | k ←  k + 1
9  Fin tant que
10 Afficher k
```

1. Quelle valeur est affichée lorsque l'on exécute cette séquence ?
2. Implémentez cette séquence en Python et exécutez votre code afin de vérifier votre réponse.

■

■ **Exercice 1.15.**
Un étudiant fauché place 10€ sur son livret A en 2020, rémunéré 0,5% d'intérêt par an. À partir de quelle année cet étudiant aura-t-il au moins 20€ sur son livret ? Réalisez un code Python qui permette de répondre à la question. ■

■ **Exercice 1.16.**

1. En utilisant une boucle «**pour**», rédigez un code Python qui vous permettra de remplir le tableau ci-dessus.
2. Même question avec une boucle «**tant que**».

i	-5	-4	-3	-2	-1	0	1	2	3	4	5
$\frac{i^3-4}{2}$											

■

■ **Exercice 1.17.**
Algorithme : Test de divisibilité. On se propose ici de définir un algorithme pour tester si un nombre naturel b est divisible par un nombre naturel non nul a.

1. Expliquez que b est divisible par a si et seulement si b est un multiple de a.
2. Donnez les quatre premiers multiples positifs de a dans l'ordre croissant.
3. Décrivez un algorithme qui utilise une boucle pour vérifier tous les multiples de a dans l'ordre croissant, jusqu'à un certain point, pour savoir si b est un

multiple de a . À partir de quand la boucle peut-elle s'arrêter pour conclure sur la divisibilité de b par a.
4. Implémentez votre algorithme en Python et testez le pour voir s'il fonctionne correctement.
5. Proposez un second programme Python qui aboutit au même résultat, sans utiliser de boucle, en utilisant l'opérateur %.

■

■ Exercice 1.18.

Algorithme : Plus grand multiple de a inférieur ou égal à b.

1. En vous inspirant de l'algorithme dans l'exercice précédent, définissez un algorithme qui permet de trouver le plus grand multiple d'un nombre naturel a inférieur ou égal à un nombre naturel b.
2. Généralisez votre algorithme au cas où a et b sont des nombres relatifs.
3. Implémentez votre algorithme en Python.
4. Proposez un second programme Python qui aboutit au même résultat, sans utiliser de boucle, en vous servant de l'opérateur //.

■

■ Exercice 1.19.

text En utilisant le package random (taper *import random*) et la commande random.randint(a,b) qui tire un nombre au hasard en a et b. Faites une boucle conditionnelle tirant au hasard un nombre entre 0 et 50 jusqu'à être le numéro 18. On affiche le numéro dans la boucle. (Pour les NSI, cette boucle est elle infini et pourquoi ?) Faire une boucle qui tire des noms au hasard jusqu'à avoir un nombre pair. Faire une boucle où le nombre augmente de 3 en 3 jusqu'à dépasser le numéro 1802.

■

■ Exercice 1.20.

tt

1. Faire une boucle qui affiche les noms de toute votre famille
2. Faire une boucle qui compte tous les nombre de 1 à 1000
3. Faire une boucle qui dit le nombre de lettres pour le nom de chaque personne de votre famille (utiliser la fonction len())

■

Chapitre 2
Algorithmique et programmation 2

III. Les fonctions en Python

R

1. Comment un informaticien/mathématicien fait bouillir de l'eau avec une casserole vide, un robinet et une plaque chauffante ?
   Il remplit la casserole, il fait chauffer la casserole jusqu'à ébullition.
2. Comment un informaticien/mathématicien fait bouillir de l'eau avec une casserole vide, un robinet et une plaque chauffante ?
   Il vide la casserole et ré-applique la réponse de la question 1.

En informatique, on essaie d'être assez fainéant mais productif, c'est à dire lorsque l'on a fait quelque chose on essaie de le réutiliser. Pour ce faire on utilise des fonctions. Comme en mathématiques, *une fonction c'est juste quelque chose en entrée et qui donne quelque chose en sortie.* En informatique parfois ce quelque chose c'est rien.

1. Généralités

En Python, il est possible de définir des ***fonctions***. Les fonctions en programmation permettent d'utiliser de multiple fois une séquence d'instructions qui dépendent d'un certain nombre de ***paramètres*** (on parle aussi d'***arguments***). La notion de fonction en informatique diffère légèrement de la notion de fonction en mathématiques qui est présentée la semaine suivante. D'ailleurs, elle peut varier d'un langage de programmation à l'autre. On se limite ici uniquement au cas des fonctions en Python.

En Python, la définition d'une fonction passe par la définition :

— du nom de la fonction ;
— de sa liste de paramètres en entrée (qui peut être vide) ;
— de la séquence d'instruction dans la fonction ;
— de la sortie de la fonction (qui est facultative en Python).

Elle prend forme suivante :

```
1 def nom_fonction(a,b,c):
2     instructions
3     return sortie
```

Ici, la première ligne comporte :

— la commande `def` qui indique qu'une fonction va être définie ;

— `nom_fonction` qui désigne le nom de la fonction (choisie par l'utilisateur) ;
— des variables `a`, `b` et `c` qui sont les paramètres de la fonction (dont le nom est choisi par l'utilisateur), ils sont placés entre parenthèses et séparés les uns des autres par des virgule (il peut y en avoir n'importe quel nombre entier, y compris 0, dans ce cas on écrit `nom_fonction()`) ;
— un deux points.

Cette première ligne est suivie, après indentation, de la séquence d'instructions de la fonction qui se termine généralement (mais pas nécessairement) par un ***renvoi*** de la fonction indiqué par la commande `return`. Le renvoi permet de définir une valeur renvoyée en sortie. Le code d'une fonction peut contenir plusieurs renvois correspondant à différents cas (en utilisant des instructions conditionnelles, par exemple). Toutefois, l'exécution de la fonction n'effectue jamais qu'un seul envoi.

▪ **Exemple 2.1.**
La fonction `f` prend un paramètre `x` et renvoie la valeur obtenue par l'instruction `2*x+3`. En mathématiques, elle correspond à une fonction affine de la forme $f : x \mapsto 2x+3$.

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

▪

▪ **Exemple 2.2.**
La fonction `puissance` prend deux paramètres `a` et `n` et renvoie en sortie `a` à la puissance `n`.

```
1 def puissance(a,n):
2     p = 1
3     for k in range(n):
4         p = p * a
5     return p
```

On remarque que l'indentation de la boucle s'ajoute à l'indentation de la définition de la fonction. ▪

▪ **Exemple 2.3.**
La fonction `mini` prend deux paramètres `a` et `b` et renvoie le plus petit des deux. Ici le renvoi dépend d'une instruction conditionnelle.

```
1 def mini(a,b):
2     if a<b:
3         return a
4     else:
5         return b
```

▪

▪ **Exemple 2.4.**
La fonction `bonjour` affiche `Bonjour!` à l'écran. Elle ne prend aucun paramètre et ne fait aucun renvoi [1].

```
def bonjour():
    print("Bonjour!")
```

▪

2. Fonctions prédéfinies

Dans Python, un certain nombre de fonctions sont déjà prédéfinies. Elles peuvent donc être utilisées directement sans avoir à les redéfinir. Quelques unes de ces fonctions ont déjà été évoquées dans ce cours (`print`,`input`). Le tableau suivant contient une liste de fonctions prédéfinies en Python qui pourront vous servir dans le cadre de ce cours. Cette liste n'est bien évidemment pas exhaustive et vous pourrez découvrir les autres fonctions prédéfinies en Python en consultant la documentation disponible via le lien suivant : https://docs.python.org/3/.

Fonction	Description
`print(x)`	Affiche la valeur de `x` dans la console.
`print(txt,x)`	Affiche le texte de la chaîne de caractères `txt` suivi de la valeur de `x` dans la console.
`input(txt)`	Affiche le texte de la chaîne de caractères `txt` dans la console. L'utilisateur doit alors taper une valeur dans la console qui sera renvoyée par la fonction `input` dès que l'utilisateur appuie sur la touche *Entrée* du clavier.
`int(x)`	Renvoie la valeur de `x` convertie en entier lorsque cela est possible.
`float(x)`	Renvoie la valeur de `x` convertie en flottant lorsque cela est possible.
`len(txt)`	Renvoie le nombre de caractères dans la chaîne de caractères `txt`.
`abs(x)`	Renvoie la valeur absolue de `x`.
`round(x,n)`	Renvoie la valeur arrondie de `x` à 10^{-n} près.

▪ **Exemple 2.5.**
Dans le code Python suivant, on utilise les fonctions `float`, `input`, `print` et `round`.

```
x = float(input("Veuillez rentrer une valeur pour x"))
print("La valeur approchee de x a 0,01 pres vaut",round(x,2))
```

1. attention à ne pas confondre affichage et renvoi!

Remarquez l'encapsulation de la fonction `input` dans la fonction `float`. Ceci est nécessaire car la fonction `input` ne permet que de récupérer le texte taper par l'utilisateur (même si ce texte comporte des nombres). Ainsi, si l'on n'utilise pas la fonction `float` et que l'utilisateur rentre la valeur `123.3` avant de valider par la touche Entrée, c'est la chaîne de caractère `"123.3"` qui sera stockée dans la variable `x`. La fonction `float` permet alors de convertir la chaîne de caractère `"123.3"` en `123.3` qui est un nombre flottant. ■

Bibliothèques

En plus des fonctions prédéfinies qui sont accessibles directement, Python propose également un certain nombres de ***bibliothèques*** (aussi appelés ***modules***) qui contiennent de nombreuses fonctions déjà programmées (ainsi que d'autres objets prédéfinis). Pour utiliser les fonctions d'une bibliothèque, il faut d'abord ***importer*** la bibliothèque. La manière la plus simple pour ce faire est d'utiliser la commande `import` suivi du nom de la bibliothèque.

■ **Exemple 2.6.**
L'instruction suivante permet d'importer la bibliothèque standard `math`.

```
1 import math
```

■

Pour utiliser une fonction d'une bibliothèque (ou n'importe quel autre type d'objet d'une bibliothèque), il suffit alors d'utiliser la syntaxe `nom_bibliotheque.nom_fonction`.

■ **Exemple 2.7.**
Le code suivant permet d'importer la bibliothèque `math` dans Python pour utiliser la fonction `sqrt` qui permet de calculer une racine carrée[2]. Ici, on affiche la valeur de $\sqrt{2}$.

```
1 import math
2 print(math.sqrt(2))
```

■

Une autre façon d'importer les fonctions et objets d'une bibliothèque est d'utiliser une instruction suivant la syntaxe ci-dessous :

```
1 from nom_bibliotheque import *
```

Cette instruction permet d'importer toutes les fonctions et objets d'une bibliothèque de manière à ce qu'ils puissent être utilisés sans rappeler le nom de la bibliothèque.

■ **Exemple 2.8.**
Ici, on importe `math` pour utiliser la valeur approchée `pi` de π qui est enregistrée dans la bibliothèque `math`.

2. `sqrt` est une abréviation de *square root* qui signifie *racine carrée* en anglais.

```
from math import *
print(pi)
```

■

Il existe un certain nombre d'autres façons d'importer des bibliothèques en Python (en entier ou partiellement) qui ne seront pas détaillées ici. Vous pourrez les retrouver dans la documentation Python.

En plus de la bibliothèque `math`, on utilise également dans ce cours la bibliothèque `random` (pour faire des simulations aléatoires) et la bibliothèque `mathplotlib.pyplot` (pour faire des représentations graphiques). Les fonctions des bibliothèques seront présentées au fur et à mesure de leur utilisation dans le cours ou les exercices.

3. Fonctions aléatoires

Le langage Python permet également la définition de ***fonctions aléatoires***. En informatique, une fonction aléatoire est une fonction dont la valeur renvoyée est le résultat d'un tirage aléatoire. Cette valeur n'est donc pas déterminée de manière unique par la valeur des paramètres en entrée de la fonction. La bibliothèque `random` en Python contient un certain nombre de fonctions aléatoires prédéfinies et notamment la fonction `randint` qui renvoie un entier compris entre deux valeurs fournies en paramètres de manière aléatoire et selon une loi de probabilité uniforme[3].

■ **Exemple 2.9.**
Dans le code suivant, on importe les fonctions de la bibliothèque `random` afin de tirer au hasard soit 0, soit 1.

```
from random import *
print(randint(0,1))
```

■

Simulation

Une ***simulation*** est un processus qui calque un phénomène réel dans le but d'en prédire l'issue. Le résultat d'une simulation peut ainsi informer sur le résultat possible du phénomène. On utilise généralement les simulations en informatique pour recréer un phénomène qui coûterait trop cher à réaliser physiquement (comme pour une simulation de crash-test d'avion) ou qui est tout simplement impossible à réaliser physiquement (comme pour une simulation de l'évolution climatique du globe terrestre).

■ **Exemple 2.10.**
On peut utiliser la fonction `randint` comme précédemment pour simuler le lancer d'une pièce qui peut tomber sur pile (que l'on fait correspondre au 0) ou face (que l'on fait correspondre au 1). Si un lancer de pièce est facile à réaliser physiquement, le lancer d'un million de pièces par exemple devient plus compliqué alors qu'en simulation cela ne met que quelques secondes sur un ordinateur moderne. Dans le code Python

3. Comme on le verra au cours de la semaine **??**, cela signifie que toutes les valeurs peuvent sortir avec la même probabilité

suivant, une simulation permet de comptabiliser combien de fois une pièce tombe sur face en un million de lancers. C'est la variable p qui permet de stocker cette valeur.

```
from random import *
p=0
for k in range(1000000):
    p = p + randint(0,1)
```

■

Exercices non à soumettre

Exercice 2.1.
Déterminez les renvois des fonctions `f` et g définies ci-dessous pour les appels `f(2)`, `f(0)`, `g(1)` et `g(3)`

```
def f(x):
    y = x**2-3
    return y
```

```
def g(x):
    y = x**2-3
    z = 3*x-y
    y = z + x
    return z
```

Exercice 2.2.
On considère les trois fonctions Python suivantes.

```
def f1(x,y):
    z = 0
    while z<20:
        z = x+y
        x = 2*x
        y = 3*y
        z = y*z
    return z
```

```
def f2(x,y):
    z = 0
    while z<20:
        z = x+y
        x = 2*x
        y = 3*y
    z = y*z
    return z
```

```
def f3(x,y):
    z = 0
    while z<20:
        z = x+y
        x = 2*x
    y = 3*y
    z = y*z
    return z
```

Détaillez les calculs de `f1(5,1)`, `f2(5,1)` et `f3(5,1)`. Vous listerez l'ensemble de toutes les affectations qui sont réalisées en indiquant si celles-ci ont lieu :
— avant le début de la boucle ;
— dans la boucle et, dans ce cas, au combientième tour de la boucle ;
— après la sortie de la boucle.

Exercice 2.3.

Le tableau suivant donne la mention en fonction de la note N obtenue à la première session du baccalauréat.

Condition	Mention
$0 \leq N < 8$	AJOURNE(E)
$8 \leq N < 10$	RATTRAPAGE
$10 \leq N < 12$	SANS MENTION
$12 \leq N < 14$	ASSEZ BIEN
$14 \leq N < 16$	BIEN
$16 \leq N$	TRES BIEN

Définissez une fonction Python qui prend une note en paramètre et renvoie la mention correspondante sous forme de chaîne de caractère. ■

■ **Exercice 2.4.**
À quelle notion mathématique correspond la fonction Python définie ci-dessous ?

```
1 from math import *
2
3 def fonction_mystere(a,b,c,d):
4     return sqrt((a-c)**2+(b-d)**2)
```

■

■ **Exercice 2.5.**
On considère la fonction Python suivante qui prend comme paramètres six variables qui correspondent aux coordonnées des sommets d'un triangle ABC dans un repère orthonormé et renvoie un couple de valeurs (les couples sont des objets qui existent en Python et sont notés à l'aide de parenthèses comme en mathématiques).

```
1 def G(xA,yA,xB,yB,xC,yC):
2     xG=(xA+xB+xC)/3
3     yG=(yA+yB+yC)/3
4     return (xG,yG)
```

1. On appelle cette fonction en rentrant les paramètres correspondant aux points $A(2;1)$, $B(-2;5)$ et $C(0;-3)$. Que renvoie-t-elle ?
2. On considère maintenant que A, B et C sont trois points quelconques du plan. Montrez que le point G correspondant au renvoi de la fonction `G` en Python est défini par l'équation vectorielle suivante :

$$3\overrightarrow{OG} = \overrightarrow{OA} + \overrightarrow{OB} + \overrightarrow{OC}$$

3. Montrez que cette équation vectorielle est équivalente à :

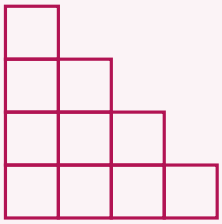
$$\overrightarrow{GA} + \overrightarrow{GB} + \overrightarrow{GC} = \overrightarrow{0}$$

4. À quoi sert cette fonction Python ?

■

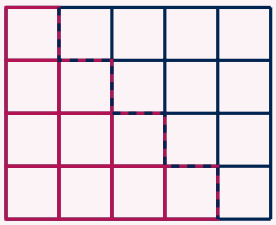
■ **Exercice 2.6.**

On considère un escalier (en deux dimensions) formés de carré comme dans la représentation graphique ci-dessous.



On appelle n le nombre de carrés situés à la base de l'escalier. Le but de l'exercice est de déterminer le nombre total m de carrés qui composent l'escalier en fonction de n. Dans l'exemple, $n = 4$ et $m = 10$.

1. Si on part du sommet pour arriver à la base, chaque étage possède un carré supplémentaire par rapport au précédent. En vous appuyant sur cette idée, écrivez une fonction Python qui permet de calculer m en fonction de n en parcourant une boucle.
2. On complète le schéma précédent en rajoutant un escalier inversé de la même forme que le premier de la manière suivante :



Déduisez-en une expression algébrique de m en fonction de n.

3. Quelle approche privilégieriez-vous entre les deux ? Justifiez votre réponse.

■

■ **Exercice 2.7.**

Algorithme : Première puissance de a inférieure (ou supérieure) à b. On cherche ici à définir un algorithme qui permette d'obtenir la première puissance entière positive d'un nombre réel positif a inférieure ou égale à un autre nombre réel positif b. Autrement dit, quelle est la plus petite valeur de $n \in \mathbb{N}^*$ telle que $a^n \leq b$.

1. Quelle valeur de n convient si $a \leq b$?
2. On considère par la suite que $b < a$.

(a) En considérant, $a = 4$ et $b = \frac{1}{2}$, montrez qu'un tel n n'est pas forcément défini.
(b) Discutez en fonction de la position relative de a par rapport à 1 de l'existence d'un tel n.
(c) On suppose de plus que $a < 1$. Complétez le programme Python suivant qui permet de déterminer la valeur souhaitée dans le cas où $a = 0,9$ et $b = 0,02$.

```
a = 0.9
b = 0.02

n = 1
p = a
while p > b:
    n = ...
    p = ...
```

3. En utilisant les différents éléments que vous avez trouvés, écrivez une fonction Python qui permettent de répondre à l'énoncé du problème.
4. Écrivez une fonction pour le cas où l'on recherche la première puissance de a supérieure à b.

■

■ Exercice 2.8.

Algorithme : Test de primalité. Le but de cet exercice est de déterminer si un entier $p > 1$ est premier ou non.

1. Rappelez la définition d'un nombre premier.
2. Que vaut `(p%k==0)` lorsque `k` est un diviseur de `p` ? Et sinon ?
3. Expliquez pourquoi le code suivant permet bien de déterminer si un nombre est premier.

```
def premier1(p):
    for k in range(2,p):
        if (p%k == 0):
            return False
    return True
```

4. Montrez que l'on peut arrêter la boucle après avoir testé tous les diviseurs potentiels de p inférieurs ou égaux à $\sqrt{p}$.
5. Complétez le code suivant pour intégrer cette information. On pourra utiliser la fonction `sqrt` du module `math` et faisant attention au fait que `range` doit prendre en entrée des paramètres de type `int`.

```
def premier2(p):
        n = ...
    for k in range(2,n):
        if (p%k == 0):
            return False
    return True
```

6. Testez chacune des fonctions pour déterminer si 999999937 est premier. Vous devriez trouver que c'est un nombre premier dans les deux cas : c'est le plus grand nombre premier à 9 chiffres.
7. Les deux fonctions ne mettent pas le même temps à répondre. Estimez le nombre de tours parcourus dans la boucle dans chacun des cas.
8. Testez maintenant les fonctions pour déterminer si 999999938 est premier. Cette fois-ci les fonctions ont mis à peu près le même temps. Pourquoi?

■

■ **Exercice 2.9.**

1. Complétez le code suivant de la fonction `lancer` pour qu'elle simule le lancer d'un dé équilibré à six faces.

```
from random import *

def lancer():
    return randint(...,...)
```

2. On souhaite réaliser une simulation du lancer de 1000 dés. Complétez le code suivant et exécutez le à la suite du code précédent afin de pouvoir remplir le tableau des effectifs correspondant à la série statistique de la simulation.

```
n = 1000
n1 = 0
n2 = 0
n3 = 0
n4 = 0
n5 = 0
n6 = 0

for k in range(n):
    x = lancer()
    if x == 1:
        n1 = n1 + 1
    elif x == 2:
        n2 = n2 + 1
    ...
    else:
        n6 = n6 + 1
```

Face	1	2	3	4	5	6
Effectif						

3. Écrivez une fonction `moyenne` qui calcule la moyenne de la valeur obtenue lors d'un lancer à partir des valeurs obtenues lors de la simulation.

■

■ **Exercice 2.10.**

Dans cet exercice, on réalise la ***simulation du saut de puce***. C'est un problème classique en mathématiques qui appartient à une classe de problèmes très importants connus sous le nom de ***marches aléatoires***. Les marches aléatoires ont des applications multiples dans de nombreux domaines allant de l'informatique à l'économie en passant par la biologie. Elles sont beaucoup utilisées, par exemple, pour faire se déplacer des personnages dans des jeux vidéos. Les marches aléatoires peuvent être définies de manière très complexe mais on ne considère ici que la version la plus simple qu'est le saut de puce. On imagine qu'une puce se déplace en faisant des sauts le long d'un axe gradué dans un sens ou dans l'autre. On appelle x la position de la puce sur l'axe gradué. La puce saute toutes les secondes exactement et le saut de la puce mesure toujours exactement la même longueur d'unité 1. Si l'on suppose que la puce commence à la position $x = 0$ de l'axe gradué, après 1 seconde, elle sera soit à $x = -1$, soit à $x = 1$. Enfin, le sens dans lequel la puce saute à chaque fois est parfaitement aléatoire : la puce a autant de chance de sauter vers la droite que vers la gauche.

1. Complétez le code Python suivant où l'on définit une fonction `saut` qui prend en paramètre correspondant à la position de la puce avant un saut et renvoie la position de la puce après le saut.

```
from random import *

def saut(x):
    a = randint(0,1)
    if a == 1:
        return ...
    else:
        return ...
```

2. On suppose que la puce commence toujours à la position $x = 0$. Réalisez une fonction `deplacement` qui simule le déplacement de la puce pendant n secondes et renvoie sa position finale (c'est-à-dire après n sauts). Vous pourrez utiliser la fonction `saut` déjà définie.
3. Montrez qu'après 4 sauts, la position x de la puce appartient nécessairement à $\{-4;-2;0;2;4\}$.
4. Exécutez la commande `deplacement(4)` vingt fois de suite et relevez les effectifs correspondant à l'occurrence de chacune des positions finales possibles dans le tableau suivant.

x	-4	-2	0	2	4
Effectif					

5. Quelle est la valeur moyenne de la position finale donnée par cette série statistique?
6. On voudrait réaliser une série statistique similaire mais, cette fois-ci, avec un effectif total beaucoup plus important : 1,000,000 par exemple. Faire un programme Python qui génère une telle série statistique et déterminez les fréquences obtenues dans ce cas.
7. Quelle est la valeur moyenne de cette nouvelle série statistique?

Chapitre 3
Une Introduction à la Programmation Orientée Objet : Les Classes et les Objets

I. Concept : Programmation Orientée Objet

Il consiste en la définition et l'interaction de briques logicielles appelées objets ; un objet représente un concept, une idée ou toute entité du monde physique, comme une voiture, une personne ou encore une page d'un livre. Il possède une structure interne et un comportement, et il sait interagir avec ses pairs. Il s'agit donc de représenter ces objets et leurs relations ; l'interaction entre les objets via leurs relations permet de concevoir et réaliser les fonctionnalités attendues, de mieux résoudre le ou les problèmes. Dès lors, l'étape de modélisation revêt une importance majeure et nécessaire pour la POO. C'est elle qui permet de transcrire les éléments du réel sous forme virtuelle.

Beaucoup des types que nous découvrirons dans le prochain chapitre sont considérés comme des **Classes** en Python. Par exemple, les listes, dictionnaires, chaîne de caractère.
À la place de manipuler des Classes/types directement donnés par Python. On peut créer directement les objets/types qui nous intéressent.

Un bel exemple d'utilisation de Programmation Orientée Objet est la simulation de foule. Une vidéo de la chaîne Fouloscopie (https://www.youtube.com/watch?v=w-Oy4TYDnoQ) vulgarise bien ce concept.

II. Création d'une Classe

Pour créer une nouvelle Classe, on fait : (la nouvelle classe est Eleve)

```
1 class Eleve:
```

III. Attributs

Comme dit précédemment, **les Classes/Objets** sont utiles pour réaliser des modélisations. On va prendre le problème de modélisation suivant : on veut modéliser des

élèves de Seconde qui veulent partir en Première générale, afin de pouvoir faire un programme qui propose des répartitions d'élèves dans des classes.

Première question : Que faut-il pour caractériser un tel élève ... On posera qu'il faut connaître :

— Son nom
— Son sexe
— Ses voeux de spécialité.

Ces caractéristiques sont appelés **Attributs** de la Classe. Pour ajouter des attributs, on aura besoin d'un constructeur. Ce constructeur est la fonction/méthode : **__init__**.

```
class Eleve:
        def __init__(self):
                self.nom = "Mettre un Nom"
                self.sexe = "Adolescent"
                self.specialite = []
```

Définition 3.1.
Un objet est une instance d'une classe *i.e.* tout comme "babar" est une instance d'une chaîne de caractère. Un objet sera une instance d'une classe.

On veut créer un objet élève avec les attributs : "Kévin", "Homme", ["Math","NSI","SES"] on va affecter à une variable NouvelEleve un objet de type Eleve. et affecter à chaque attribut la valeur correspondante. Tout d'abord on crée l'objet NouvelEleve.

```
PremierEleve ← Eleve()
```

```
PremierEleve = Eleve()
```

Regardons maintenant ses attributs. Les attributs d'un objet sont des variables, don cpour les observer On entre dans la console : (les >>> représente ce que la console renvoie)

```
PremierEleve.nom
>>> "Mettre un Nom"
PremierEleve.sexe
>>> "Adolescent"
PremierEleve.specialite
>>> []
```

Pour modifier cet attribut, on entre dans la console :

```
PremierEleve ← Eleve()
PremierEleve.nom ← "Kevin"
PremierEleve.sexe ← "Homme"
PremierEleve.specialite ← ["Math","NSI","SES"]
```

On sait que **__init__** est une fonction. Donc elle prend une entrée. Ici l'entrée est *self*. Ce dernier représente l'objet.

Pour voir les attribut de l'Objet PremierEleve maintenant, j'entre de nouveau dans la console :

```
PremierEleve.nom
>>> "Kevin"
PremierEleve.sexe
>>> "Homme"
PremierEleve.specialite
>>> ["Math","NSI","SES"]
```

▪ **Exercice 3.1.**
L'objectif de cet exercice est de faire une classe Prof et de créer deux Objets : Gorce et Gibaud avec les bons attributs.

1. Trouvez les caractéristiques d'un professeur de lycée (sur papier).
2. Définir la classe Professeur (avec son constructeur)
3. Créer deux variables de type Professeur. Une variable sera Gibaud, l'autre Gorce.
4. Changer les attributs de ces deux fonctions pour que Gibaud et Gorce aient les bons attributs

▪

IV. Les Méthodes

Toute la beauté de la programmation orientée objet est que les objets ont :
— des caractéristiques appelés **Attributs**
— des actions/fonctions appelés **Méthodes**

Les méthodes sont des fonctions internes à une Classe. Cela permet aux objets d'agir et d'interagir entre eux.

Pour faire une méthode (ici DirePresent ou __init__)on entre dans la console :

```
class Eleve:
        def __init__(self):
                self.nom = "Entrer un nom"
                self.sexe = "Adolescent"
                self.voeux = []

        def DirePresent(self):
                print(self.nom + " Present !")
```

R **Toutes les méthodes prennent au moins self en entrée**

Pour appeler cette méthode, on doit déjà créer l'objet puis appeler la méthode. (on va changer l'attribut d'abord). On entre alors dans la console, l'appel de la méthode est en ligne 5 :

```
1 SecondEleve = Eleve()
2 SecondeEleve.nom = "Isma"
3 SecondEleve.sexe = "Femme"
4 SecondEleve.voeux = ["Math","NSI","HLP"]
5 SecondEleve.DirePresent()
6 >>> "Isma Present !"
```

R Pour qu'un objet appelle une méthode on met un • entre l'objet et la méthode. Comme la méthode est une fonction on met des parenthèses avec les arguments après la méthode. Si la méthode ne prend que *self* on met des parenthèses vides.

Cependant les méthodes peuvent être plus compliquées et faire des actions plus complexes. Par exemple **__init__** est une méthode ou **append** qui est une méthode pour les liste et qui permet d'ajouter un élément à la fin d'une liste.

On pourrait avoir le suivant :

```
1 class Eleve:
2         def __init__(self):
3                 self.nom = "Entrer un nom"
4                 self.sexe = "Adolescent"
5                 self.voeux = []
6         def DirePresent(self,Phrase):
7                 print(Phrase)
```

On aurait alors comme appel de DirePresent :

```
1 SecondEleve = Eleve()
2 SecondeEleve.nom = "Isma"
3 SecondEleve.sexe = "Femme"
4 SecondEleve.voeux = ["Math","NSI","HLP"]
5 SecondEleve.DirePresent("Je suis ici, Monsieur."
    )
6 >>> "Je suis ici, Monsieur"
```

On remarque cette fois DirePresent prend un argument en entrée. Cet argument est le *Phrase* de *def DirePresent(self,Phrase)*