Bases de la programmation : Cours

1ère année

IUT de Villetaneuse.

Hanène Azzag,
Frédérique Bassino,
Pierre Gérard,
Bouchaïb Khafif,
Mathieu Lacroix,
Mustapha Lebbah,
François Lévy,
Guillaume Santini.

28 février 2012

Chapitre 1

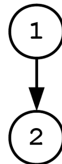
Séquentialité et variables

1.1 Introduction

Un ordinateur est une machine et en tant que telle, elle ne fait que ce qu'on lui demande. Les programmeurs ont pour tâche de faire faire à ces machines exactement ce qu'ils veulent qu'elles fassent et dans le bon ordre ni plus, ni moins, ni autre chose. Donner à un ordinateur une séquence d'ordres à effectuer, c'est écrire un programme qu'il devra exécuter.

Les problèmes à résoudre étant parfois complexes, la tâche principale est décomposée en plusieurs sous-problèmes plus simples résolus les uns après les autres. En définitive, pour résoudre un problème, le programmeur conçoit une série d'instructions, chacune donnant un ordre à l'ordinateur.

L'instruction `print(1)`, par exemple, correspond à l'ordre donné à l'ordinateur d'afficher 1 à l'écran. Il s'agit d'un ordre simple, mais qui peut être combiné à d'autres pour produire des résultats (un peu) plus complexes. Par exemple, cette instruction suivie de `print(2)` forme un programme avec deux instructions (dans le cercle : les numéros de ligne du programme) :




```
print(1)
print(2)
```

qui affichera 1 puis affichera 2.

Cet exemple est évidemment simplisme et peut paraître très éloigné des programmes que l'utilisateur contemporain utilise habituellement. Mais même pour réaliser des applications complexes (animation et rendu 3D, par exemple), ce sont toujours les mêmes principes simples qui sont à l'œuvre.

Au lieu d'instructions qui se contentent d'afficher un nombre à l'écran, on utilisera des instructions toujours aussi simples à écrire, mais dont les effets seront plus spectaculaires. Le travail du programmeur n'est pas plus complexe pour autant. Ainsi, au lieu de

```
print(1)
print(2)
```

On aura des instructions comme

```
pivoter(0.2)   # pour faire tourner une scène 3D de 0.2 degrés
afficher(0.1)  # pour demander un rendu 3D de la scène en moins de 0.1 seconde
```

dont l'impact visuel sera bien plus important, mais dont la complexité pour le programmeur reste comparable.

Construire des programmes informatiques, c'est toujours se demander quelles instructions exécuter, et dans quel ordre.

1.2 Séquences d'instructions

Un programme est une série d'instructions effectuées une par une, de la première à la dernière, en séquence. Chaque instruction est un ordre simple donné à l'ordinateur.

Chaque programme commence par exécuter sa première instruction. Ce n'est que lorsqu'elle est terminée qu'il passe à la seconde. S'il y a une troisième instruction, il attend d'avoir terminé la deuxième pour l'exécuter.

▶ Sur ce thème : **Exercice 1, TD1**

1.3 Variables

1.3.1 Noms de variables

Les instructions d'un programme permettent de traiter des données (quantité de vie restante, age d'un client, nombre de munitions, *etc.*) qui doivent pouvoir être ajustées au fur et à mesure que le programme se déroule. L'odinateur stocke donc chacune de ces informations dans sa mémoire à un endroit bien identifié, grâce à des variables. On utilise le terme *variables* parce que ces données sont susceptibles d'être modifiées, par opposition à des *constantes*).

Les variables associent un nom à une valeur. Une variable nommée `x` peut par exemple prendre la valeur 1. Chaque variable mobilise un espace mémoire accessible par un nom, et qui peut contenir une valeur. La valeur d'une variable peut être changée au cours de l'exécution d'un programme.

`x` | `1` |

Selon la syntaxe utilisée dans ce cours, le nom d'une variable commence par une lettre minuscule (a à z) ou majuscule (A à Z), ou bien par le caractère souligné (_). Pour la suite de son nom, on peut utiliser des lettres minuscules ou majuscules, des soulignés et des chiffres (0 à 9). *Le nom d'une variable ne contient donc pas d'espace.*

▶ Sur ce thème : **Exercice 2, TD1**

1.3.2 Affectation

L'opération principale mettant en œuvre les variables est l'affectation, qui permet de changer la valeur d'une variable en utilisant son nom. Une affectation est notée avec le signe « = ». A gauche du signe « = », on place le nom de la variable dont on veut changer la valeur, et à droite on définit la valeur que doit prendre la variable. Ce symbole n'est pas du tout l'égalité mathématique.

Par exemple, l'instruction

```
x=2
```

change la valeur de x. Notez bien que `2=x` est incorrect. 2 est un nombre constant et ne change pas de valeur. Ce n'est pas une variable et on ne peut donc pas le mettre à gauche de l'opérateur d'affectation.

Dans cet exemple, la nouvelle valeur est directement spécifiée. On aurait également pu placer à droite du symbole « = » une expression qui représente un calcul, comme par exemple dans

```
x=1+2
```

L'opérateur d'affectation commence toujours par évaluer la valeur à droite du signe « = » (quelle que soit la complexité du calcul) avant de placer le résultat dans l'espace mémoire réservé à la variable.

Ainsi, l'instruction

```
x=(1+2)/3
```

calcule d'abord la valeur de l'expression `(1+2)/3` avant d'affecter le résultat du calcul à la variable nommée x, qui prend finalement la valeur 1.

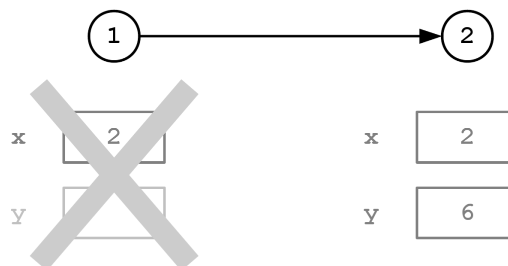
x 1

1.3.3 Utilisation des variables

Dans l'évaluation d'une expression, un nom de variable est remplacé par sa valeur. Si `x` vaut 2, par exemple, alors `(x*3)` vaut 6. Ainsi, à l'issue des deux instructions consécutives suivantes :

```
x=2
y=x*3
```

`x` vaut 2 et `y` vaut 6.



Cette évaluation des variables grâce à leur valeur vaut également pour les sorties écran. L'instruction `print(1+2)` est correcte, mais `print(x+y)` l'est également, et donnera le même résultat si par exemple `x` et `y` valent respectivement 1 et 2 juste avant son exécution.

▶ Sur ce thème : **Exercice 3, TD1**

Il est important de noter qu'une variable ne peut être utilisée que si elle a été définie plus haut. Ainsi, le programme suivant est correct :

```
x = 0
y = 2
print((x+y)+2)
```

alors que le suivant le l'est pas

```
x = 1
z = x+y
print(z)
```

parce qu'à la deuxième instruction, la variable `y` n'avait pas encore été définie. Selon le principe de séquentialité, ajouter une ligne à la fin comme dans le le programme suivant :

```
x = 1
z = x+y
print(z)
y = 3
```

n'aurait d'ailleurs pas résolu le problème.

Pour définir une variable, il faut lui affecter une valeur.

Une source d'erreurs importante en programmation est la non-maîtrise de la séquentialité des instructions. Les instructions s'exécutent dans l'ordre où elles sont écrites, les éventuelles modifications de variables se font dans le même ordre.

▶ Sur ce thème : **Exercice 4, TD1**
▶ Sur ce thème : **Exercice 5, TD1**
▶ Sur ce thème : **Exercice 6, TD1**

1.4 Types de données, opérations et conversions

1.4.1 Types de nombres et opérations permises

Il existe deux types de nombres :

- Les nombres entiers, appelés *integer*, ou en abrégé `int` : ils sont signés (1, 2, -3).
- Les nombres à virgule flotante, appelés *floating point numbers*, ou en abrégé `float` : ils sont positifs ou négatifs et on les note avec un point pour la virgule (1.0, 2.3, -17.32, 3.141516).

Pour définir une variable de type `int`, on lui affecte une valeur entière. Pour définir une variable de type `float`, on lui affecte un nombre à virgule. Par exemple :

- `a = 0` définit la variable `a` comme un `int` de valeur nulle,
- `a = 0.0` définit la variable `a` comme un `float` de valeur nulle.

Le tableau suivant récapitule succinctement les principales opérations possibles sur des nombres :

Expression	Résultat, `x` et `y` étant des nombres (`int` ou `float`)
`x + y`	somme de `x` et de `y`
`x - y`	différence de `x` et de `y`
`x * y`	produit de `x` et de `y`
`x / y`	division de `x` par `y`
`x % y`	reste de la division entière de `x` par `y`
`-x`	négation de `x`
`x ** y`	`x` à la puissance `y`
`abs(x)`	valeur absolue de `x`

Il faut être très vigilant quand au type des nombres quand on applique les opérateurs ci-dessus : si une opération implique deux nombres de même type, alors le résultat reste du même type. Si une opération implique à la fois une valeur de type `int` et une autre de type `float`, alors le résultat sera de type `float` (`float` « absorbe » `int`). Notons en particulier que *le résultat de la division de deux nombres entiers ne donne pas un `float`*. Typiquement, 3/2 vaut 1 et pas 1.5.

1.4.2 Chaines de caractères et opérations permises

Jusqu'ici, toutes les données saisies, calculées et affichées étaient numériques. Il existe en fait d'autres types de données comme par exemple les chaînes de caractères (« string » en anglais).

Par exemple, le programme

```
1 print('hello world')
```

affiche « hello world » à l'écran (sans les guillemets), `'hello world'` étant une chaîne de caractères alpha-numériques.

Les valeurs des chaînes de caractères sont définies entre simples quotes « ' » (apostrophes). Attention, chaque caractère compte, ainsi que la casse (majuscules/minuscules). Ainsi, les chaînes de caractères suivantes sont toutes différentes les unes des autres :

– 'HelloWorld'
– 'Hello World'
– 'hello world'
– 'helo world'
– 'halo world'
– 'halo 1 world 2'

On peut définir des variables de type chaînes de caractères de la même manière que celles de type numérique. Ainsi, le programme suivant :

```
1 str = 'Hello world'
2 print(str)
```

définit une variable de nom `str` de type chaîne de caractères, pour ensuite l'afficher.

str | 'hello world'

La chaîne de caractères vide (sans un seul caractère dedans) se note par deux simples quotes accolées « '' ». Par exemple, `str = ''` définit une chaîne de caractères et lui donne la valeur d'une chaîne vide. *Attention, il s'agit bien ici de deux simples quotes (deux apostrophes à la suite)*, et pas d'une double quote (guillemets anglais).

Il est possible d'appliquer quelques opérations sur les chaînes de caractères. Ces opérations sont résumées dans le tableau suivant :

Expression	Résultat, `s` et `t` étant des chaînes de caractères
`s + t`	concaténation de `s` et de `t` (mises bout à bout)
`len(s)`	longueur de `s`
`lower(s)`	`s` mis en minuscules
`upper(s)`	`s` mis en majuscules

Notez que certaines opérations peuvent être interprétées différemment selon le type des données auxquelles elles s'appliquent. C'est par exemple le cas de l'opérateur « + ». Ainsi `print(1+2)` affichera 3 (le + étant interprété comme une somme de deux nombres), alors que `print('hello' + 'world')` affichera `helloworld` (le + étant interprété comme une opération de concaténation de deux chaînes de caractères).

Si l'opérateur `+` peut être utilisé pour des nombres ou des chaînes de caractères, il n'est par contre pas possible d'utiliser les deux à la fois. En effet, on ne peut pas additionner un nombre avec une chaîne de caractères.

1.4.3 Conversion de type de données

Il est possible de transformer une chaîne de caractères en un nombre et un nombre en chaîne de caractères. De même, il est possible de transformer un `int` en `float` et inversement. Ceci se fait à l'aide des trois opérations de conversion données ci-dessous.

Expression	Résultat
`int(val)`	`val` est transfomée en un nombre de type `int`
`float(val)`	`val` est transfomée en un nombre de type `float`
`str(val)`	`val` est transfomée en une chaîne de caractères

Le tableau suivant indique alors les résultats des opérations de conversion `int(val)`, `float(val)` et `str(val)` sur une variable `val`. La valeur de `val` avant conversion (et donc son type) est donnée à la première ligne.

valeur de val	10	2.5	'2'	'2.5'	'12a34'
int(val)	10	2	2	ERREUR	ERREUR
float(val)	10.0	2.5	2.0	2.5	ERREUR
str(val)	'10'	'2.5'	'2'	'2.5'	

On remarque que l'opération `int(val)` supprime ce qui se trouve après la virgule si `val` est de type `float`. Ceci revient alors à prendre la partie entière inférieure de la variable `val`.

On remarque aussi que les opérations `int(val)` et `float(val)` génèrent une erreur si `val` est une chaîne de caractères qui ne correspond pas à un entier ou un nombre à virgule, respectivement.

▶ Sur ce thème : **Exercice 7, TD1**
▶ Sur ce thème : **Exercice 8, TD1**
▶ Sur ce thème : **Exercice 9, TD1**

1.5 Entrées / sorties

La fonction `print()` permet d'afficher une valeur connue à l'écran. Elle permet d'afficher indifféremment des nombres ou des chaînes de caractères. C'est une fonction qui réalise une sortie car elle fait « sortir » des valeurs de l'ordinateur vers l'utilisateur.

Il existe également des fonctions d'entrée qui permettent d'entrer des valeurs dans l'ordinateur. La plus simple est `raw_input()` qui s'utilise à droite d'un opérateur d'affectation « = ». Par exemple, le programme suivant demande une chaîne de caractères à l'utilisateur puis l'affiche :

```
1 z = raw_input()
2 print(z)
```

La première instruction est une affectation. Le programme va donc en premier lieu chercher à évaluer la valeur de ce qui est à droite du signe « = ». Pour ce faire, l'ordinateur invite l'utilisateur à saisir une valeur au clavier. Une pression de la touche « Entrée » validera sa saisie et celle-ci sera prise en compte par l'ordinateur.

Notons bien que la fonction `raw_input()` permet uniquement de saisir **des chaînes de caractères**. Si on veut saisir un nombre, il convient de procéder en deux temps : procéder d'abord à la saisie d'une chaîne de caractère, puis la convertir en un nombre.

▶ Sur ce thème : **Exercice 10, TD1**
▶ Sur ce thème : **Exercice 11, TD1**

Le programme suivant permet de demander à l'utilisateur les deux côtés d'un rectangle, et d'en calculer la surface avant de l'afficher :

```
1 print('Quelle est la longueur du premier côté ?')
2 cote1 = float(raw_input())
3 print('Quelle est la longueur du deuxième côté ?')
4 cote2 = float(raw_input())
5 surface = cote1 * cote2
6 print('La surface du rectangle ainsi formé est' + str(surface))
```

L'utilisation de variables pour stocker les longueurs des côtés saisies permet ici d'écrire un programme générique permettant de calculer la surface de n'importe quel rectangle.

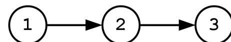
▶ Sur ce thème : **Exercice 12, TD1**
▶ Sur ce thème : **Exercice 13, TD1**
▶ Sur ce thème : **Exercice 14, TD1**

TD1 : Séquentialité et variables

Exercice 1 : Séquences d'instructions

Qu'affichent les programmes suivants ?

```
print(1)
print(2)
print(3)
```




```
print(2)
print(1)
print(3)
```

Avec les mêmes instructions, ces deux programmes ne font pas la même chose. Tous les programmes exécutent donc en premier les premières instructions : ils se « déroulent » du haut vers le bas.

Exercice 2 : Noms de variables

Parmi ces exemples, seuls certains sont des noms de variable valides. Lesquels ?

- `x`
- `x1`
- `X1`
- `toto`
- `éric`
- `_eric`
- `t_42`
- `24_t`

Exercice 3 : Séquentialité et modifications successives des valeurs des variables

Qu'affiche le programme suivant ?

```
toto = 4
print(toto)
toto = 5 + 5
print(toto)
tata = toto + 4
print(tata)
tata = tata + 5
print(toto)
tata = tata + (toto*2)
print(tata)
```

Pour répondre à la question, dessiner les cases mémoire à chaque étape de l'exécution du programme.

Exercice 4 : Séquentialité et valeur des variables

Qu'affiche le programme suivant ?

```
a=1
b=2
a=b
b=a
print(a)
print(b)
```

Exercice 5 : Séquentialité et valeur des variables

Qu'affiche le programme suivant ?

```
a=1
b=2
b=a
a=b
print(a)
print(b)
```

Exercice 6 : Echange des valeurs de deux variables

Que manque-t-il aux programmes précédents pour réaliser un échange entre les valeurs des variables a et b ? Écrire un programme qui réalise un tel échange.

Exercice 7 : Types numériques

A chaque ligne du programme suivant, donnez la valeur et le type des variables à gauche de l'opérateur d'affectation

```
a = 1.0
b = 2
c = a + 1
d = b + 3
c = float(d)
a = c / b
a = int(c) / b
```

Exercice 8 : Types et addition

Donnez la valeur des expressions suivantes :

- `123 + 123`
- `'123' + '123'`
- `123 + '123'`
- `'123' + 123`

Exercice 9 : Concaténation de chaînes de caractères

Considérons le programme suivant :

```
cp = 93
nom = 'Seine Saint-Denis'
phrase = ?????
print(phrase)
```

Que mettre à la place des ? ? ? ? ? pour que le programme affiche « Le code postal de la Seine Saint-Denis est 93 » ?

Exercice 10 : Saisie de chaînes de caractères au clavier

Qu'affiche le programme suivant, à supposer que l'utilisateur saisisse 123 puis 456 ?

```
a = raw_input()
b = raw_input()
c = a+b
print(c)
```

Quels sont les types de a, b et c ?

Exercice 11 : Saisie de nombres au clavier

Comment faudrait-il modifier ce programme pour qu'il affiche 579, à supposer que l'utilisateur saisisse 123 puis 456 ? NB : 579 est la somme de 123 et de 456. Désormais, quels sont les types des différentes variables ?

Exercice 12 : Adaptation de programmes existants

Que faut-il modifier à ce programme pour que ce soit le périmètre qui soit calculé ?

```
print('Quelle est la longueur du premier cote ?')
cote1 = float(raw_input())
print('Quelle est la longueur du deuxieme cote ?')
cote2 = float(raw_input())
surface = cote1 * cote2
print('La surface du rectangle ainsi forme est' + str(surface))
```

Exercice 13 : Calcul de clôture

Écrire un programme qui demande à l'utilisateur les dimensions en mètres d'une zone rectangulaire à clôturer et qui affiche un message intelligible indiquant le nombre de piquets à utiliser, sachant qu'il faut au maximum 2 mètres entre chaque piquet. On suppose que les dimensions en mètres sont des nombres entiers.

Exercice 14 : Echange de nombres sans variable intermédiaire

Écrire un programme qui demande à l'utilisateur deux nombres, les affiche, les échange, et les ré-affiche après échange, mais sans utiliser de variable intermédiaire. On pourra utiliser une addition et une soustraction pour ne rien perdre.

Note : la méthode d'échange sans variable paraît plus intéressante que celle donnée en cours puisqu'elle n'utilise pas d'autre variable. (Elle nécessite cependant que l'ordinateur effectue des additions/soustractions en plus). On pourrait donc penser qu'elle est meilleure. Mais elle est aussi moins générale car elle suppose que les variables à échanger soient de type numérique : elle ne fonctionnerait pas sur des chaînes de caractères par exemple. D'une manière générale, il y a souvent plusieurs manières de résoudre un problème. Le rôle du programmeur consiste à identifier le meilleur algorithme (le meilleur enchaînement d'instructions), compte tenu des hypothèses qu'il lui semble raisonnable de faire.

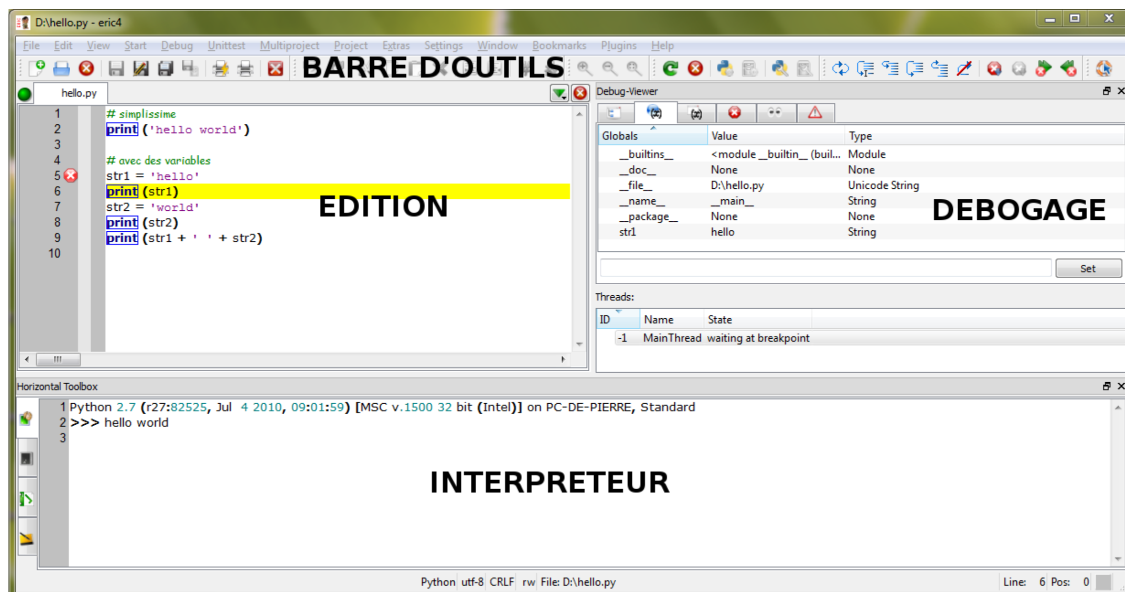
TP1 : Lancement d'Eric

Exercice 15 : Présentation d'Eric

Tous les TP de BdP se feront en utilisant le logiciel Eric, dans sa version 4. Eric est un outil de développement complet pour le langage de programmation Python. Il dispose d'un certain nombre d'options que nous n'utiliserons pas :

- Nous nous en tiendrons aux fonctions de base de Eric ;
- Nous n'exploiterons qu'une petite partie des possibilités offertes par le langage Python.
- La raison de ces simplifications est la mise en avant de ce qui est fondamental et commun à la plupart des langages de programmation. Ce cours n'est pas spécifiquement un cours de Python, mais aborde des notions bien plus générales que celles propres à un langage de programmation particulier. D'autres enseignements à l'IUT vous permettront d'acquérir des compétences spécialisées dans telles ou telles techniques.

Question 15.1 : Pour lancer Eric, suivez les instructions de votre enseignant. Vous devriez obtenir une fenêtre ressemblant à celle ci-dessous.



La fenêtre principale d'Eric se décompose en plusieurs parties :

- Le menu (tout en haut)
- La barre d'outils (juste en dessous)
- La fenêtre d'édition qui vous permettra d'écrire vos programmes (à gauche)
- La fenêtre de débogage qui vous permettra de contrôler pas à pas l'exécution de vos programmes (à droite)
- La fenêtre de l'interpréteur qui vous permettra a) de suivre l'exécution de vos programmes b) d'entrer manuellement et une à une des instructions à exécuter.

Il est possible que votre fenêtre principale comporte beaucoup plus (ou peut-être moins) de sous-fenêtres et de boutons dans la barre d'outils. Pour avoir quelque chose au même niveau de détail que ci-dessus, vous pouvez cliquer avec le bouton droit de la souris quelque part sur la barre d'état. Un menu contextuel vous permet alors de sélectionner les éléments apparaissant à l'écran.

Question 15.2 : Pour obtenir la même chose que ci-dessus, décochez tout sauf :

- Horizontal toolbox
- Debug viewer

- File
- Edit
- View
- Start
- Debug
- Help

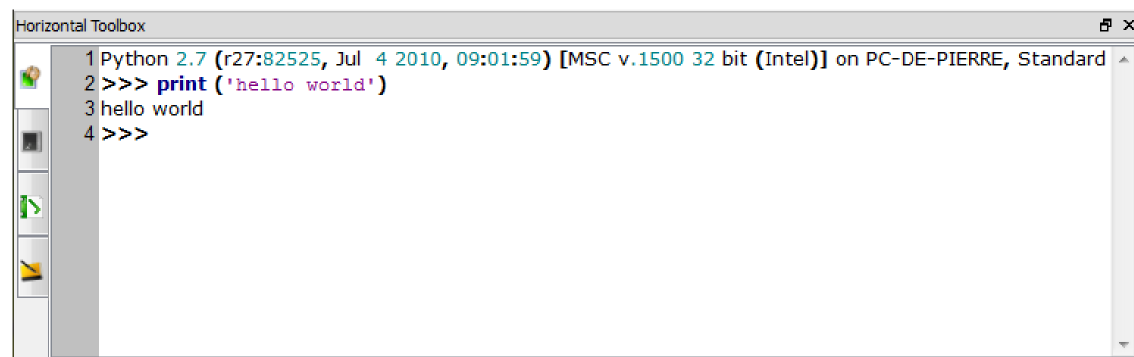
Exercice 16 : Utilisation de l'interpréteur

Dans la partie inférieure de la fenêtre principale, le premier onglet permet d'afficher l'interpréteur Python. C'est l'onglet affiché par défaut mais s'il arrivait que vous en changiez par inadvertance, resélectionnez simplement ce premier onglet.

L'interpréteur invite à saisir une commande lorsqu'un triple chevron >>> est affiché en début de ligne.

Question 16.1 : Saisissez `print('hello world')` et terminez par la touche « Entrée » du clavier.

L'instruction s'exécute et « hello world » s'affiche à la suite. Dans l'interpréteur, vous devriez voir :

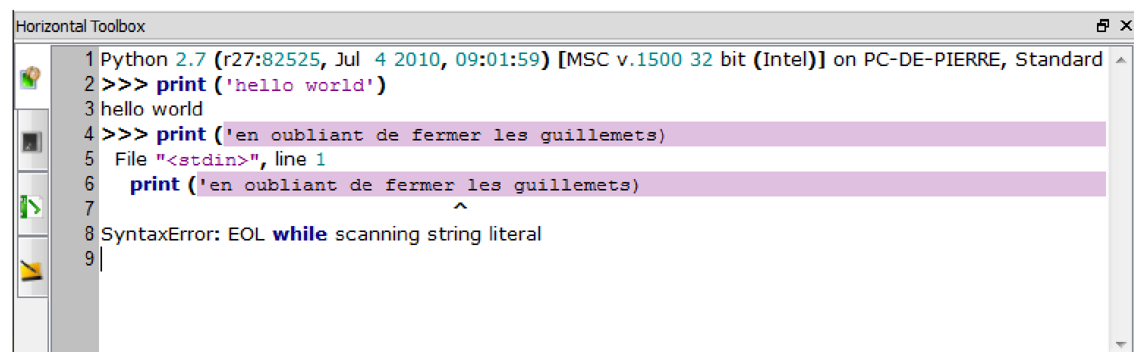


Après avoir interprété et exécuté l'instruction précédente, le triple chevron est à nouveau présent pour vous inviter à taper une seconde instruction.

Attention : quand vous donnez des instructions à l'ordinateur, il convient de respecter la syntaxe. Chaque caractère compte :

- `prunt` au lieu de `print` ne fonctionnera pas
- ouvrir une parenthèse sans la fermer provoquera une erreur
- oublier des simples quotes n'est pas correct
- ...

Question 16.2 : Faites volontairement l'erreur d'oublier la seconde simple quote terminant la chaîne de caractères. Vous obtenez un message d'erreur et l'interpréteur affichera ce qui suit.



Il y a différents types d'erreur, l'erreur de syntaxe (SyntaxError) est la plus courante, et résulte de fautes de frappe. Confrontés à une erreur, lisez toujours le message retourné par l'interpréteur, il vous donnera de précieux indices pour la résolution du problème.

Pour voir réapparaître les trois chevrons >>> appuyez simplement sur la touche « Entrée » et saisissez une nouvelle instruction, sans erreur cette fois.

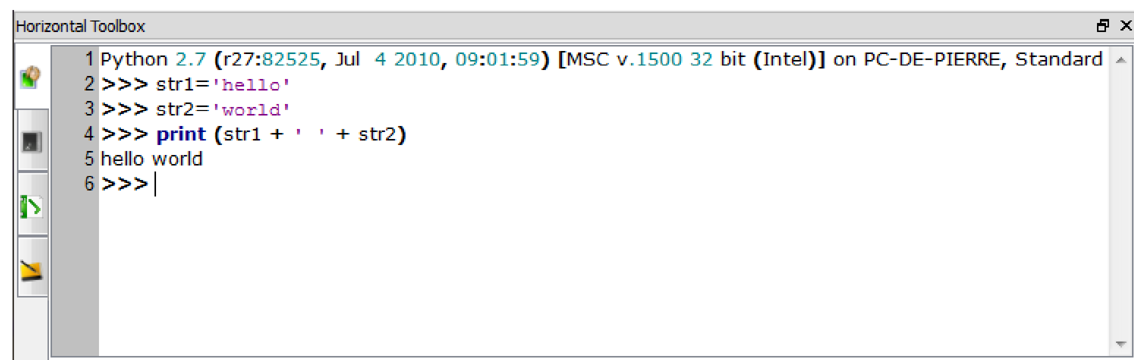
Exercice 17 : Séquence d'instructions dans l'interpréteur

Un programme est souvent plus complexe qu'une seule instruction. Un certain nombre d'instructions effectuées l'une après l'autre sont nécessaires à la résolution de problèmes plus complexes que le simple affichage d'un message amical.

Par exemple, en utilisant des variables, on pourrait donner plusieurs instructions à la suite :
- D'abord `str1 = 'hello'`.
- Puis `str2 = 'world'`
- Enfin `print(str1 + ' ' + str2)`

Question 17.1 : Entrez la première instruction, validez-la en appuyant sur « Entrée » : elle est exécutée. Entrez de la même manière les instructions suivantes.

L'interpréteur affiche « hello world ». Le résultat final est le suivant :



Notez bien que les deux premières instructions définissent des variables, et que l'interpréteur conserve leur valeur à mesure de l'exécution des instructions. Sans cela, la dernière instruction ne fonctionnerait pas.

Question 17.2 : Entrez d'ailleurs maintenant la suite d'instructions :

```
>>> str2 = 'dolly'
>>> print(str1 + ' ' + str2)
```

Vous constaterez que `str2` a changé de valeur, si bien que la dernière instruction affiche « hello dolly ».

Exercice 18 : Remise à zéro de l'interpréteur

Il peut arriver qu'à force de saisir des instructions, de multiples variables et autres éléments soient définis, qui pourraient troubler le bon fonctionnement de nouvelles instructions. Il est toujours possible de remettre l'interpréteur à zéro. Pour ce faire, cliquez avec le bouton droit quelque part sur la zone de l'interpréteur. Un menu contextuel apparaît et vous donne le choix entre plusieurs options :
- *Clear* efface simplement ce qu'il y a à l'écran
- *Reset* remet réellement l'interpréteur à zéro

Question 18.1 : Sélectionnez « Clear » puis entrez l'instruction `print(str1)`. Vous devez voir afficher « hello », preuve que la variable `str1` est toujours définie, même si les lignes précédentes ont été effacées. Clear n'est qu'une option d'affichage.

Question 18.2 : Sélectionnez « Reset » et entrez l'instruction print(str1).

Une erreur « NameError : name 'str1' is not defined » indique que `str1` n'est pas définie, preuve que Reset a bien mis à zéro l'ensemble des définitions de l'interpréteur. Après un Reset, vous redémarrez vraiment à zéro. Pour information, la capture d'écran ci après montre l'apparence de l'interpréteur après cette séquence d'actions.

```
1 Python 2.7 (r27:82525, Jul  4 2010, 09:01:59) [MSC v.1500 32 bit (Intel)] on PC-DE-PIERRE, Standard
2 >>> print (str1)
3 hello
4 >>> print (str1)
5 Traceback (innermost last):
6   File "<stdin>", line 1, in <module>
7 NameError: name 'str1' is not defined
8
```

Exercice 19 : Exécution d'un programme écrit dans un fichier externe

L'interpréteur est un moyen extrêmement pratique de tester rapidement des solutions à des problèmes rencontrés. Les programmeurs avertis s'en servent souvent abondamment, même s'il est évident qu'en fin de compte, on ne peut pas demander à l'utilisateur d'un programme de taper des instructions une à une. Les programmes sont donc écrits en entier dans des fichiers texte, et exécutés ensuite sans que l'utilisateur ne voit ces suites d'instructions (le code source).

Question 19.1 : Avec un éditeur de texte quelconque (gedit par exemple), saisissez le programme suivant :

```
# simplissime
print('hello world')

# avec des variables
str1 = 'hello'
print(str1)
str2 = 'world'
print(str2)
print(str1 + ' ' + str2)
```

Les lignes commençant par des # indiquent un commentaire : elles ne sont pas interprétées en tant qu'instructions mais permettent de rendre les programmes plus lisibles.

Question 19.2 : Enregistrez ce programme sous le nom « hello.py » dans le répertoire de votre choix. Par exemple, /bdp/tp1 convient parfaitement et vous permettra de le retrouver facilement. Si le dossier n'existe pas, créez-le.

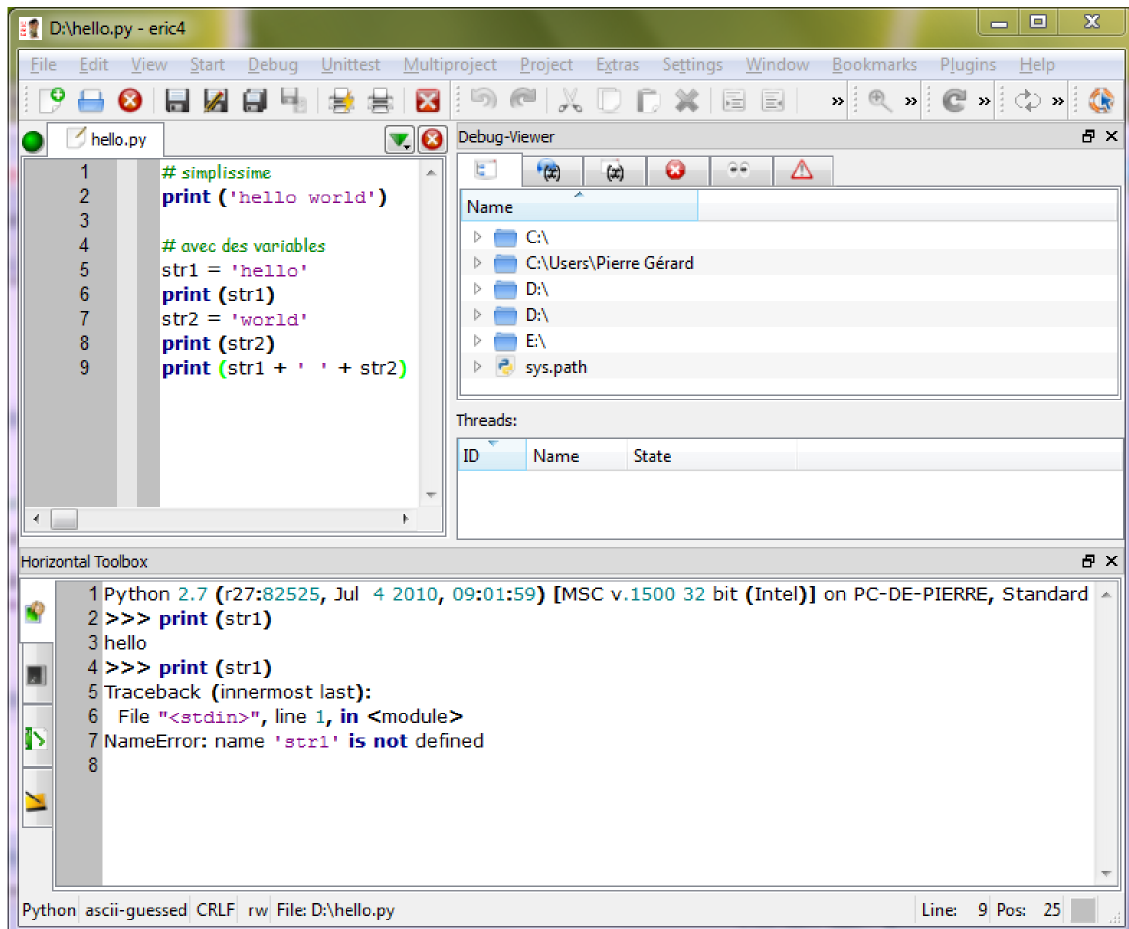
Question 19.3 : Après avoir enregistré le fichier, fermez l'éditeur de texte.

Question 19.4 : Revenez à Eric 4 (si vous l'avez fermé par inadvertance, relancez-le).

Question 19.5 : Soit en utilisant le deuxième bouton de la barre d'outils à gauche (Open), soit en utilisant le menu déroulant File > Open, soit en utilisant le raccourci clavier Ctrl+O (les touches « Ctr l » et « O » simultanément), ouvrez le fichier « hello.py ».

Si nécessaire, naviguez vers le bon répertoire. Si vous avez bien enregistré le fichier, et que vous êtes bien dans le bon répertoire, mais que le fichier n'apparaît pas, il s'agit peut-être d'un problème de filtre : sélectionnez « All Files *.* » dans la liste déroulante en bas à droite de la boîte de dialogue de sélection de fichiers.

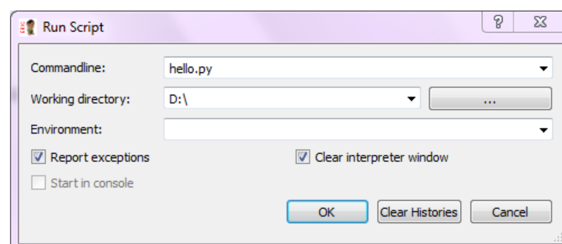
Ceci fait, votre fenêtre principale devrait avoir l'allure suivante :



Question 19.6 : Exécutez le programme. Vous pouvez soit :

- Utiliser le bouton de la barre d'outils « Run Script » (pour le trouver : laisser la souris un certain temps au dessus des boutons permet d'afficher un libellé en plus de l'image).
- Utiliser le raccourci clavier « F2 » (bien plus pratique)
- Utiliser le menu « Start > Run Script »

Une boîte de dialogue apparaît pour vous demander quel programme exécuter :



Question 19.7 : Renseignez le programme et dans quel répertoire il se trouve. Au lieu de « `D:` », mettez le répertoire dans lequel vous avez sauvegardé le fichier.

Question 19.8 : Appuyez sur OK et le programme s'exécute dans l'interpréteur, qui prend l'allure suivante :

```
1 Python 2.7 (r27:82525, Jul  4 2010, 09:01:59) [MSC v.1500 32 bit (Intel)] on PC-DE-PIERRE, Standard
2 >>> hello world
3 hello
4 world
5 hello world
6
```

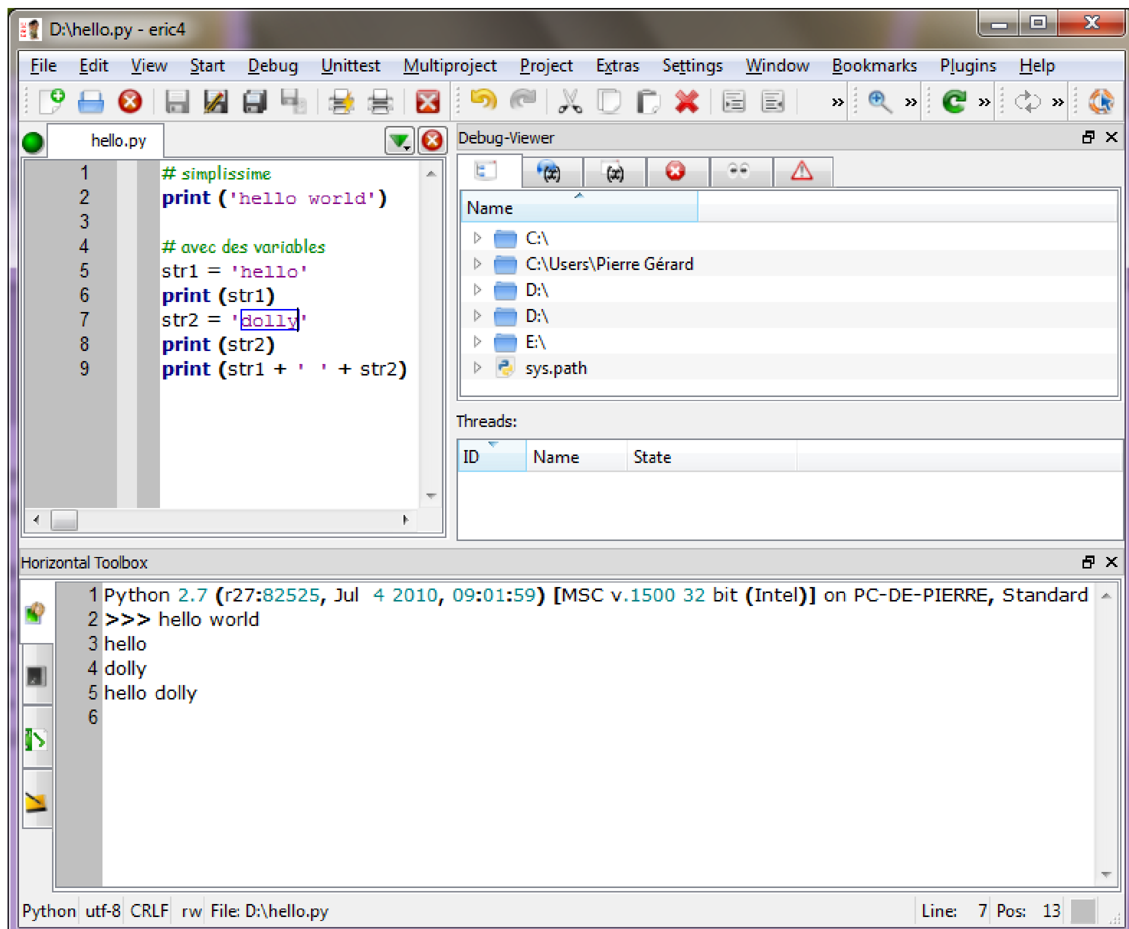
Le programme a été interprété en une seule fois, seuls les résultats des différentes utilisations de la fonction `print()` s'affichant à l'écran.

Question 19.9 : Dans la fenêtre d'édition, modifiez le programme en remplaçant la 7ème ligne « `str2 = 'world'` » par « `str2 = 'dolly'` ».

Question 19.10 : Sauvegardez le fichier :

- Bouton « Save » de la barre d'outils
- Raccourci clavier Ctrl+S
- Menu File > Save

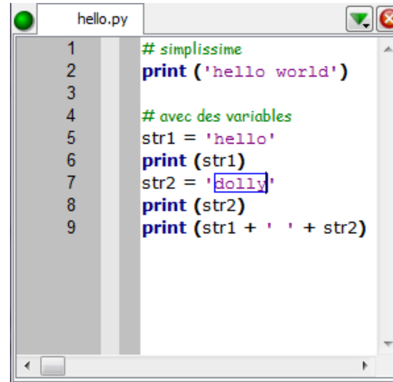
Question 19.11 : Exécutez à nouveau le script et constatez que le programme ayant été modifié, le résultat change.



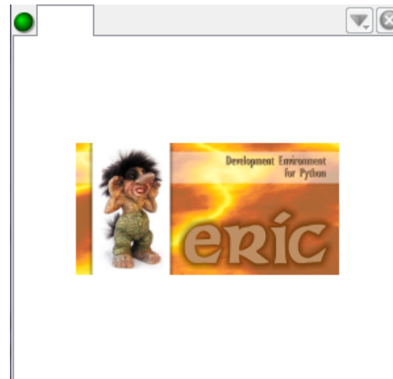
Exercice 20 : Écriture d'un programme directement à l'aide d'Eric

Dans la plupart des cas, Eric disposant d'un éditeur, vous serez amenés à écrire des programmes sans faire intervenir un autre éditeur. Nous allons donc utiliser Eric pour créer un nouveau fichier contenant un programme.

En premier lieu, fermons l'édition de « hello.py ». La partie éditeur de Eric ressemble à ceci :



Question 20.1 : Cliquez sur le bouton en croix tout en haut à droite. Ceci a pour effet de fermer l'édition de « hello.py »



Question 20.2 : Créez maintenant un nouveau fichier :

- Bouton « New » de la barre d'outils (premier à gauche)
- Raccourci clavier Ctrl+N
- Menu File > New

Question 20.3 : Le fichier et pour l'instant nommé « Untiled 1 ». Donnez lui un nom en le sauvegardant :

- Bouton « Save As »
- Raccourci clavier « Ctrl+Shift+S »
- Menu File > Save As

Appelez le nouveau fichier « echange.py » et placez-le dans le même répertoire que « hello.py ».

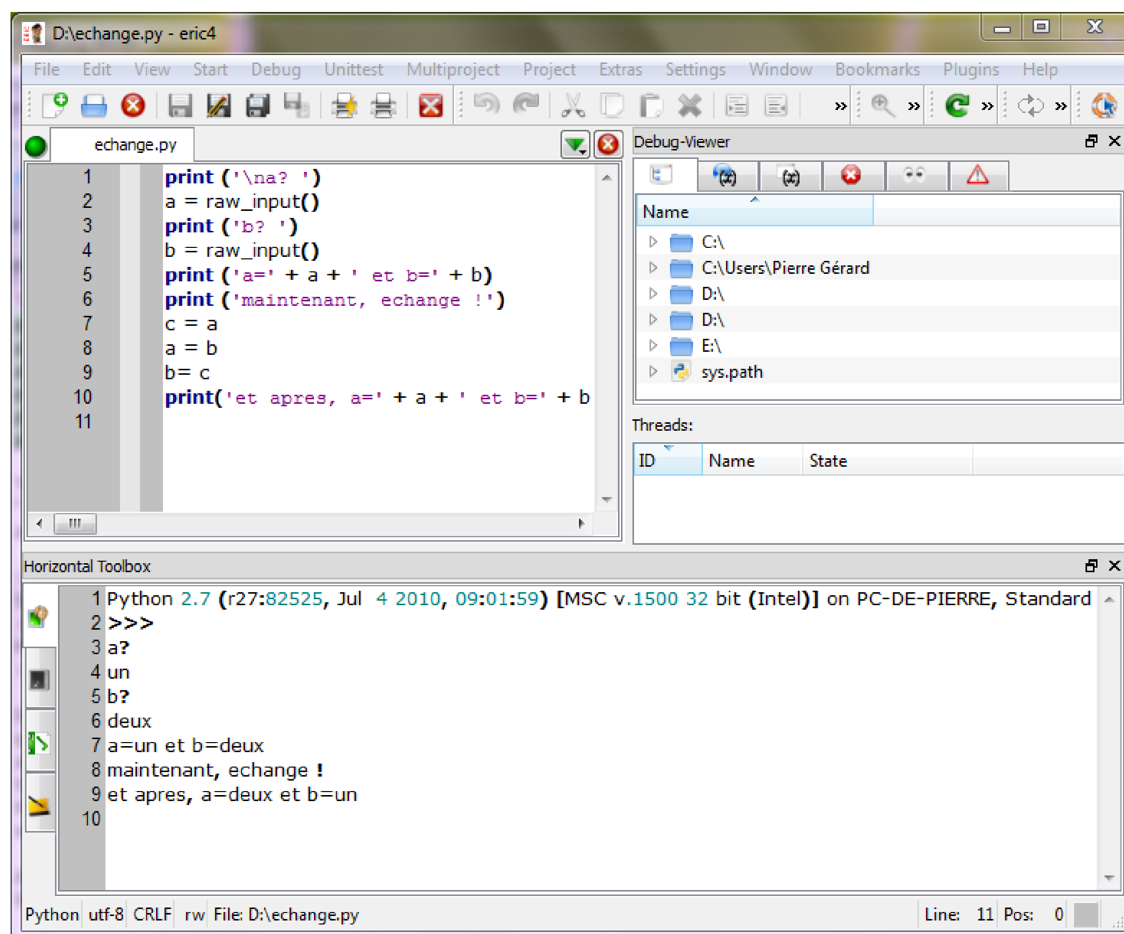
Question 20.4 : Saisissez le programme suivant puis sauvegardez :

```
1 print('\na? ')         # le \n permet de sauter une ligne
2 a = raw_input()
3 print('b? ')
4 b = raw_input()
5 print('a=' + a + ' et b=' + b)
6 print('maintenant, echange !')
7 c = a
```

```
a = b
b= c
print('et apres, a=' + a + ' et b=' + b)
```

Question 20.5 : Exécutez le programme. Quand les `raw_input` s'exécutent, tapez quelque chose dans l'interpréteur, et appuyez sur « Entrée » pour valider, le programme passera ensuite à l'instruction suivante.

En fin de compte, vous devriez obtenir une fenêtre comme :



Exercice 21 : Débogage : suspension de l'exécution d'un programme

Il est fréquent de commettre des erreurs et d'avoir des difficultés à les corriger. Les erreurs de syntaxe sont les plus simples à corriger, les plus difficiles étant souvent celles ayant trait à la structure même et à l'ordonnancement des instructions dans le programme. Pour corriger plus facilement ses erreurs, on peut s'aider d'un débogueur afin de bien contrôler l'évolution des valeurs des différentes variables au fur et à mesure de l'exécution du programme.

Pour ce faire, il faut

- Signifier au débogueur où s'arrêter
- Lui demander d'afficher l'état des variables souhaitées

Pour signifier au débogueur où suspendre l'exécution du programme le temps que le programmeur ait le temps de faire le point avant de poursuivre, on utilise des « points d'arrêt » (« *breakpoints* » en anglais). Nous allons ici suspendre l'exécution du programme à la ligne 5 puis à la ligne 10.

Question 21.1 : Pour définir un breakpoint à la ligne 5, mettez le curseur sur la ligne 5, puis soit :

- Dans le menu Debug > Toggle Breakpoint
- Utiliser le bouton correspondant dans la barre d'outils
- Utiliser le raccourci clavier Shift+F11

Un symbole avec une croix s'affiche à côté du numéro 5 de la ligne :

```
echange.py
1  print ('\na? ')
2  a = raw_input()
3  print ('b? ')
4  b = raw_input()
5  print ('a=' + a + ' et b=' + b)
6  print ('maintenant, echange !')
7  c = a
8  a = b
9  b= c
10 print('et apres, a=' + a + ' et b=' + b
11
```

Pour enlever le point d'arrêt, on utilise la même procédure. Pour le remettre si nécessaire, idem.

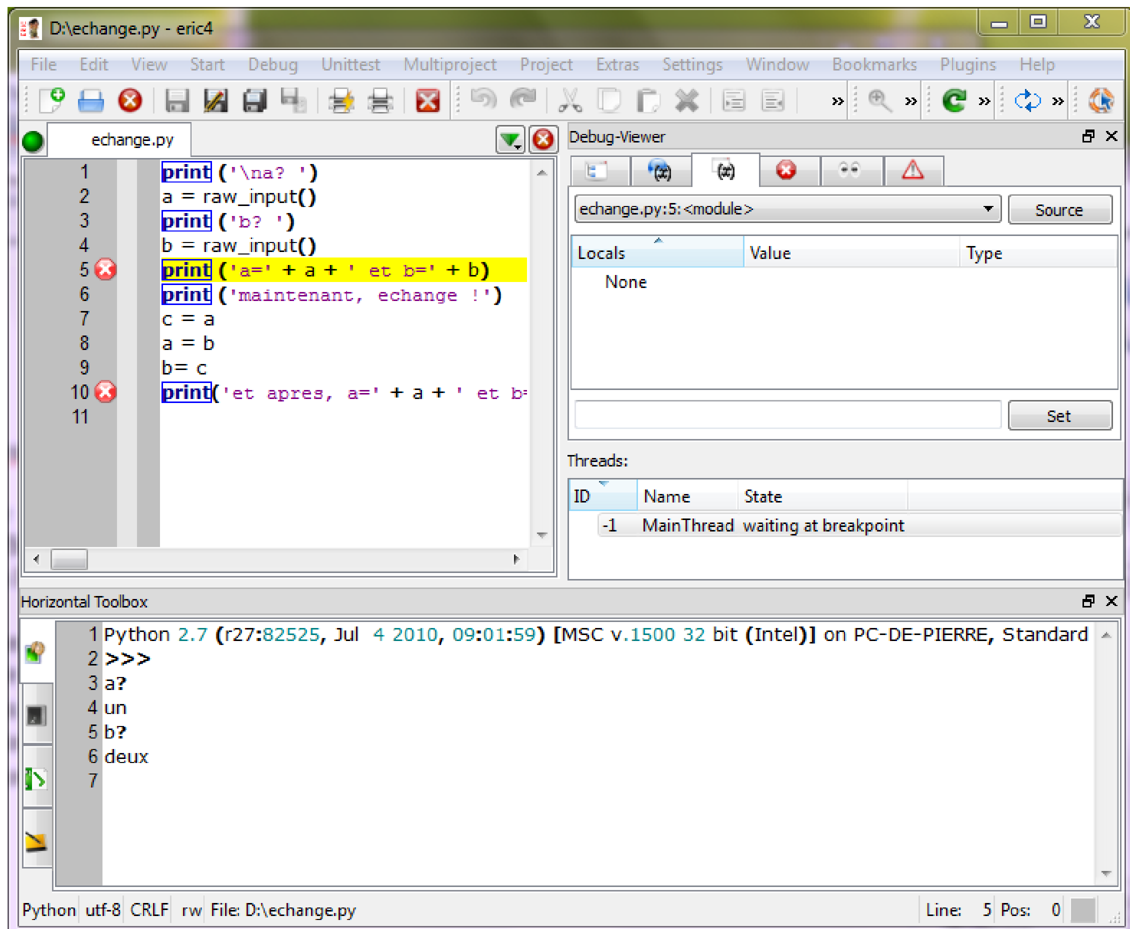
On peut procéder encore plus simplement que le raccourci clavier pour activer/désactiver un point d'arrêt : il suffit de cliquer là où se trouve le symbole avec la croix.

Question 21.2 : Procédez ainsi pour définir le point d'arrêt de la ligne 10.

Question 21.3 : Pour exécuter le programme en mode débogage de manière à ce que les points d'arrêt soient pris en compte, lancez le programme d'une manière différente :

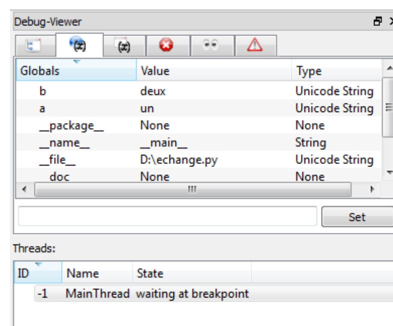
- Par le menu Start > Debug Script
- Par le bouton correspondant dans la barre d'outils
- Par le raccourci clavier « F5 »

Question 21.4 : Entrez des valeurs pour a et b, puis le programme s'interrompt à la ligne 5 (avant de l'exécuter). La ligne 5 est surlignée de jaune, en attente d'un ordre pour poursuivre l'exécution.



Exercice 22 : Débogage : valeurs des variables

Question 22.1 : A ce point, dans la partie débogage de la fenêtre principale d'Eric, sélectionnez le deuxième onglet pour voir apparaître toutes les variables définies, avec leur valeur et leur type.



Notez bien qu'à ce point, `c` n'est pas encore définie puisque cette variable ne le sera qu'à la ligne 7 du programme.

Question 22.2 : Poursuivez l'exécution du programme :

- Par le menu Debug > Continue
- Par le bouton correspondant dans la barre d'outils
- Par le raccourci clavier F6

De nouvelles choses s'affichent dans l'interpréteur, et le programme s'interrompt à la ligne 10, en surbrillance. La variable `c` est maintenant définie et les valeurs de `a` et de `b` ont été échangées.

Question 22.3 : Poursuivez l'exécution jusqu'à la fin.

Exercice 23 : Exécution pas à pas d'un programme

Une fois que vous avez défini un point d'arrêt, vous pouvez lancer l'exécution en mode débogage. Après la première interruption, si vous souhaitez suivre la valeur des variables après chaque instruction dans exception, il n'est pas nécessaire de définir des points d'arrêt pour chaque ligne de code.

Question 23.1 : Enlevez les points d'arrêt précédents et n'en placez qu'un seul, après l'exécution de la première instruction.

Question 23.2 : Lancez l'exécution en mode débogage

Le programme s'arrête avant d'effecuer la seconde instruction. Dans le deuxième onglet de la fenêtre de débogage, on remarque qu'aucune variable n'est définie pour l'instant.

Question 23.3 : Poursuivez l'exécution instruction par instruction :

- Par le menu Debug > Single step
- Par le bouton correspondant dans la barre d'outils
- Par le raccourci clavier « F7 »

La valeur de `a` vous est demandée ; `a` est définie comme une variable de valeur « un ».

Question 23.4 : Poursuivez l'exécution du programme pas à pas jusqu'à son terme par des appuis répétés sur « F7 », en regardant bien l'évolution de la valeur des variables, en particulier entre les lignes 7 et 9, où l'échange proprement dit se produit.

Une fois que le programme a été arrêté une fois à un point d'arrêt, il est possible de poursuivre l'exécution de différentes manières : instruction par instruction comme ci-dessus, ou encore jusqu'au curseur, ou d'autres façons encore qui prendront leur sens lors des séances suivantes.

Chapitre 2

Alternatives

2.1 Séquentialité

2.1.1 Rappels

Les instructions sont exécutées les unes à la suite des autres. À la fin de chaque instruction exécutée, on passe à la suivante, et ainsi de suite sans possibilité de revenir en arrière (au moins de prime abord). Considérons, par exemple, les instructions suivantes :

```
b=5
a=b
```

Nous pouvons dessiner le graphe de séquence suivant :

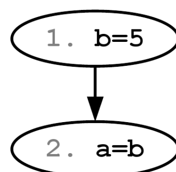


Figure 2.1 – Rappels sur la séquentialité

Dans l'exemple de la Figure 2.1, on affecte la valeur *5* à la variable *b*, puis on affecte le contenu de la variable *b* à la variable *a*.

Remarque : L'affectation de la ligne 2 écrase le contenu de `a` mais ne modifie pas le contenu de `b`.

▶ Sur ce thème : **Exercice 1, TD2**

2.1.2 Un peu de réflexion...

D'après vous est-il possible de quitter une séquence pour en emprunter une autre ? Si oui, comment décider du chemin à suivre ?

2.2 Manipuler les booléens ou comment modéliser une affirmation ?

2.2.1 Qu'est ce qu'un booléen ?

Un booléen permet de modéliser une "affirmation" qui peut être vraie (c'est à dire avérée) ou fausse. Pour cela nous disposons de deux valeurs (uniquement) possibles :

– *True* est la valeur qui indique une valeur de vérité *vraie*,
– *False* est la valeur qui indique une valeur de vérité *fausse*.

Par exemple,
– A la proposition, "La terre est plate ?", la valeur booléenne est *False*,
– et à la proposition "Une vache est un mammifère ?", la valeur booléenne est *True*.

2.2.2 Expressions booléennes simples

Une expression booléenne simple détermine la vérité *True* ou *False* d'une affirmation. Exemple :

```
print(5>15)
```

Le code précédent affiche **la valeur `False`** ; car à l'affirmation : "5 est strictement supérieur à 15", la réponse est *False*.

Il serait judicieux de pouvoir stocker une valeur booléenne dans un emplacement mémoire afin de suivre l'évolution d'une situation. Par exemple un booléen pourrait suivre l'évolution d'une température et répondre à l'affirmation "la température est-elle supérieure à 100°C ?"...

2.2.3 Variables booléennes

Introduction

Une variable booléenne est un emplacement mémoire qui permet de stocker soit la valeur *True*, soit la valeur *False*. Comme vous l'avez vu dans le cours sur la séquentialité, cet emplacement est appelé *variable* et dispose d'un nom appelé *nom de variable*.

Exemple :

```
a=True  # On affecte à la variable booléenne a la valeur True
b=False # On affecte à la variable booléenne b la valeur False
c= 15>2    # On affecte à la variable booléenne c la valeur True
print(c)
```

Avec le code précédent, nous pouvons déduire le schéma ci-dessous précisant la dynamique des variables :

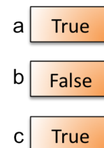


FIGURE 2.2 – Exemple d'utilisation de variables booléennes

Opérations booléennes simples

Il est possible d'utiliser les opérateurs booléens : $<$, $<=$, $>$, $>=$, $==$, $!=$ pour comparer 2 expressions (de **même type** ou de **types compatibles**). Le traitement se fait en deux étapes :

1. Les expressions sont évaluées,
2. les résultats sont comparés.

Exemple :

```
x=20
y=13
z= x+3>y
print(z)
```

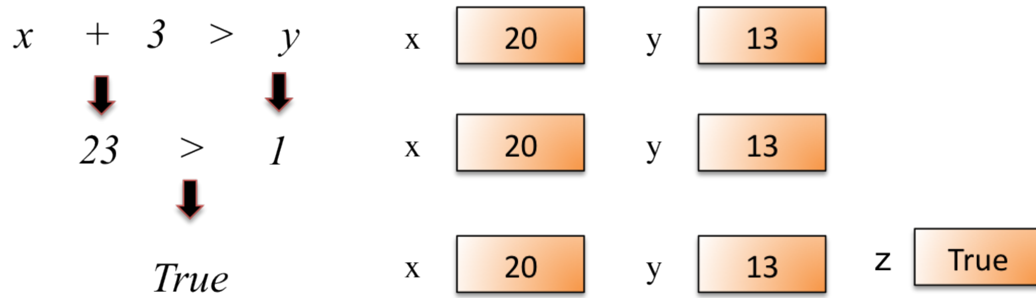


FIGURE 2.3 – Dynamique des variables lors de l'évaluation d'une opération booléenne simple

L'exécution de l'instruction de la ligne 3 du listing précédent est représenté par la figure 2.3.
L'affichage sera donc : *True*

Opérations booléennes composites

C'est une opération booléenne composée de plusieurs opérations booléennes simples. L'évaluation de ce type d'expression booléenne complexe se fait en deux étapes :

1. chaque opération booléenne simple est évaluée,
2. l'expression booléenne composite résultante est évaluée suivant les tables de vérité suivantes :

exp1	exp2	not(exp2)	exp1 and exp2	exp1 or exp2
True	True	False	True	True
True	False	True	False	True
False	True		False	True
False	False		False	False

FIGURE 2.4 – Principaux opérateurs logiques

2.2.4 Opérations booléennes équivalentes

Ce sont des expressions qui ont toujours la même valeur de vérité :

– Expressions booléennes quelconques :

e1==True	<->	e1
e1==False	<->	not(e1)
not(e1 and e2)	<->	not(e1) or not(e2)
not(e1 or e2)	<->	not(e1) and not(e2)

– Relations avec les opérations booléennes :

not(e1<e2)	<->	e1>=e2
not(e1<=e2)	<->	e1>e2
not(e1>e2)	<->	e1<=e2
not(e1>=e2)	<->	e1<e2
not(e1==e2)	<->	e1 !=e2
not(e1 !=e2)	<->	e1==e2

▶ Sur ce thème : **EXERCICE 2, TD2**
▶ Sur ce thème : **EXERCICE 3, TD2**

▶ Sur ce thème : **Exercice 4, TD2**
▶ Sur ce thème : **Exercice 5, TD2**

2.3 Comment faire des choix au sein d'un algorithme ?

L'exécution d'une séquence d'instructions peut être conditionnée par le résultat de l'évaluation d'une expression booléenne.

2.3.1 Structure de contrôle conditionnelle `if`

La structure de contrôle conditionnelle `if` permet que certaines instructions soient exécutées uniquement si une expression booléenne est vraie. Sa syntaxe est la suivante :

```
I1
I2
if condition :
       I4
       I5
I6
I7
```

Les instructions des lignes 1 et 2 sont exécutées. Puis, si *condition* est vérifiée (évaluation de la condition à *True*), alors les instructions des lignes 4 et 5 sont exécutées. Dans tous les cas, le programme exécute ensuite les instructions des lignes 6 et 7. On dit que l'exécution de la séquence des lignes 4 et 5 est *conditionnelle*. Le traitement de la structure `if` lors de l'exécution du programme est représenté dans la figure 2.5. Les numéros de la figure correspondent aux numéros de lignes du listing de code précédent.

Remarque importante : L'*indentation*, c'est-à-dire les espaces insérés avant une instruction, des lignes 4 et 5 est obligatoire. Cela permet à l'interpréteur Python de distinguer le bloc conditionnel. Ce bloc peut contenir autant d'instructions que nécessaire.

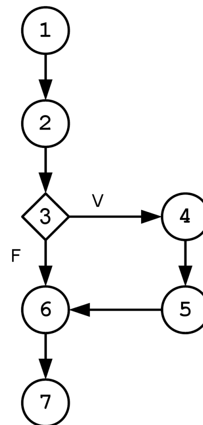


Figure 2.5 – Principe de la structure `if`

Exemple :

```
temperature = int(raw_input())
pression=100

if temperature > 55 :
    print('alerte')          #indentation obligatoire
    pression = pression - 5  #indentation obligatoire
```

```
print(pression)                # cette instruction ne fait pas partie du "if"
```

Dans cet exemple, le programme affichera à l'écran le message `alerte` et diminuera la pression de 5 uniquement si la température saisie par l'utilisateur est strictement supérieure à 55.

▶ Sur ce thème : **Exercice 6, TD2**

2.3.2 Structure de contrôle conditionnelle `if, else`

Cette structure de contrôle permet d'exécuter une séquence d'instructions si une *condition* est remplie. Cependant, si cette condition n'est pas remplie, une autre séquence d'instructions est exécutée.

```
if condition :
       I1
       I2
else:                          # not(condition)
       I3
       I4
I5
I6
```

Si la condition `condition` est vérifiée (évaluation de la condition à *True*), alors les instructions des lignes 2 et 3 sont exécutées, sinon (c'est-à-dire si la condition est évaluée à *False*) la séquence d'instructions des lignes 5 et 6 est exécutée. Dans tous les cas, le programme exécute ensuite les instructions des lignes 7 et 8. Le traitement de la structure `if, else` lors de l'exécution du programme est représenté dans la figure 2.6. Les numéros de la figure correspondent aux numéros de lignes du listing de code précédent.

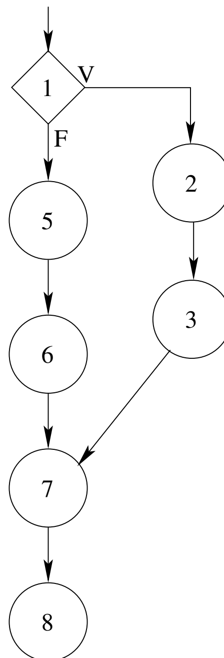


Figure 2.6 – Principe de la structure `if, else`

Exemple :

```
temperature = int(raw_input())
if temperature > 55 :
```

```
3      print('alerte')          #indentation obligatoire
4  else :
5      print('Tout va bien au niveau de la température') #indentation obligatoire
```

Dans cet exemple, le programme affichera à l'écran le message `alerte` si la température saisie par l'utilisateur est strictement supérieure à 55. Dans le cas contraire, il affichera à l'écran le message `Tout va bien au niveau de la température`.

2.3.3 Structure de contrôle conditionnelle `if, elif, else`

Les alternatives peuvent être imbriquées pour exprimer des choix "complexes" et exclusifs les uns des autres, ce qui permet d'affiner le traitement selon un contexte donné.

L'instruction `elif`, contraction de "else if" (sinon si), permet d'exécuter un bloc d'instructions si la condition associée est évaluée à `True`. Cependant, cette condition ne sera évaluée que si toutes les conditions du `if` et des `elif` précédents ont été évaluées à `False`. Dans une structure `if, elif, else`, on peut avoir autant d'instructions `elif` que nécessaire. Le traitement de la structure `if, elif, else` lors de l'exécution du programme est représenté dans la figure 2.7. Les numéros de la figure correspondent aux numéros de lignes du listing de code suivant.

```
1  I1
2  if condition1 :
3         I2
4  elif condition2 :
5         I3
6  elif condition3 :
7         I4
8  else :
9         I5
10 I6
```

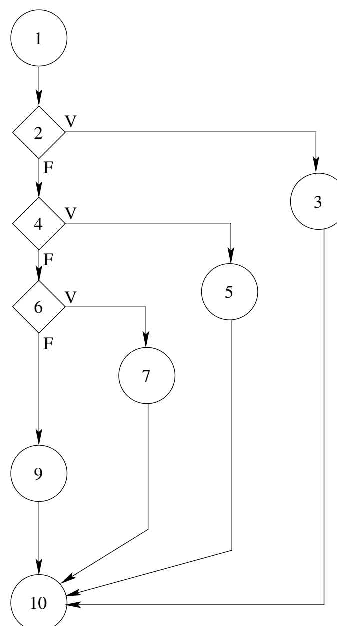


Figure 2.7 – Principe de la structure `if, elif, else`

Dans l'exemple suivant, un message approprié est affiché suivant la valeur de la température. On remarque qu'il y a trois messages possibles qui peuvent être affichés (`danger d'explosion`, `alerte`, `régime normal`).

```
temperature = int(raw_input())

if temperature > 75:  # temperature > 75
  print('danger d\'explosion')
elif  temperature > 55: # temperature > 55    et <= 75
  print('alerte')
else:                   # temperature <= 55
  print('régime normal')
```

- ▶ Sur ce thème : **Exercice 7, TD2**
- ▶ Sur ce thème : **Exercice 8, TD2**
- ▶ Sur ce thème : **Exercice 9, TD2**
- ▶ Sur ce thème : **Exercice 10, TD2**
- ▶ Sur ce thème : **Exercice 11, TD2**
- ▶ Sur ce thème : **Exercice 12, TD2**

TD3 : Alternatives

Exercice 1 : Séquences d'instructions

Que fait la séquence suivante ?

```
x = 1
y = 2
print(x)
z = x+y
print(z)
```

Important : Vous prendrez soin de suivre pas à pas l'évolution des variables.

Exercice 2 : Evaluation d'expressions booléennes

Prévoir le résultat des expressions booléennes (1), (2), (3) et (4) dans l'algorithme suivant :

```
x = 12
test = x>12
test = x<11 or (x>40 and x<100)
test = x!=9
test = not(x>10 and x<=12) and x%2==0
test = x>=10 and test
```

Important : Vous prendrez soin de suivre pas à pas l'évolution des variables.

Exercice 3 : Exprimer des formules booléennes

Donner la formule booléenne (partie *italique* du texte) correspondante à la condition de l'alternative pour les énoncés suivants :

- s'il fait beau ou que je suis en forme j'irai me promener *beau* et *enForme* sont des booléens
- si la moyenne est comprise entre 12 et 14 alors on délivre la mention assez bien ;*moyenne* est un réel.
- s'il fait trop chaud et qu'il ne pleut pas alors ouvrir la fenêtre ;*tropChaud* et *pleuvoir* sont des booléens.
- s'il ne fait pas trop chaud ou qu'il pleut alors fermer la fenêtre ;*tropChaud* et *pleuvoir* sont des booléens.
- si je ne suis pas fatigué et que j'ai du courage j'irai courir ;*fatigue*, *courage* sont des booléens
- si je ne suis pas fatigué et que j'ai du courage j'irai courir sauf s'il pleut ou que la température n'est pas comprise entre 12 et 25 degrés ;*fatigue*, *courage*, *pleuvoir* sont des booléens, *température* est un entier.

Exercice 4 : Tic, tac !

Que fait le programme suivant ? Pensez à suivre le contenu des variables.

```
flag = False
flag = not(flag)
flag = not(flag)
flag = not(flag)
flag = not(flag)
flag = not(flag)
```

Exercice 5 :

1. Saisir une valeur entière. Si cette dernière est paire et positive ou si cette valeur est impaire et comprise entre 5 (inclus) et 25 (inclus), alors l'expression booléenne vaut True sinon False. De plus vous afficherez le résultat. Remarque : Il existe un opérateur appelé *modulo* dont le symbole est % qui permet de calculer le reste de la division euclidienne. Par exemple, 3%2 vaut 1.
2. Donner un jeu d'essai (4 tests significatifs) et prévoir les résultats.

Exercice 6 : Affichage de la valeur absolue d'un nombre

Écrire un programme qui calcule et affiche la valeur absolue d'un nombre.

Exercice 7 : Programme mystère

Que fait le programme suivant ?

Important : Suivez l'évolution du contenu des variables sur papier :

```
1  interrupteur=False
2  intensite=1
3
4  interrupteur =False
5  intensite=0
6  interrupteur =not(interrupteur)
7  if interrupteur ==True:
8      print('lampe allumee')
9      intensite=intensite+1
10 else:
11         print('lampe eteinte')
12
13 interrupteur =not(interrupteur)
14
15 if interrupteur ==True:
16         print('lampe allumee')
17         intensite=intensite+1
18 else:
19         print('lampe eteinte')
20
21 interrupteur =not(interrupteur)
22
23 if interrupteur ==True:
24         print('lampe allumee')
25         intensite=intensite+1
26 else:
27         print('lampe eteinte')
28
29
30 interrupteur =not(interrupteur)
31
32 if interrupteur ==True:
33         print('lampe allumee')
34         intensite=intensite+1
35 else:
36         print('lampe eteinte')
37
38 interrupteur =not(interrupteur)
39
40 if interrupteur ==True:
41         print('lampe allumee')
42         intensite=intensite+1
43 else:
```

```
44        print('lampe eteinte')
```

Exercice 8 :

Écrire un algorithme qui demande deux nombres entiers à l'utilisateur et qui calcule la différence du plus grand nombre avec le plus petit nombre, quel que soit l'ordre de saisie.

Exercice 9 : Calcul de gabarit

On définit le *gabarit* d'un objet en fonction de sa *taille*. Le gabarit peut prendre les valeurs `'Grand'`, `'Moyen'` ou `'Petit'` (qui sont des chaînes de caractères) selon que la *taille*, qui est un nombre entier, est respectivement supérieure ou égale à 10, comprise entre 4 (inclus) et 10 (non inclus) ou strictement inférieure à 4.

Écrire un programme qui demande à l'utilisateur la taille d'un objet et affiche après l'avoir déterminé le garabit correspondant. On s'attachera à ne pas faire de test inutile.

Exercice 10 :

Écrire un algorithme qui demande à l'utilisateur l'heure, les minutes et les secondes, et il affichera l'heure qu'il sera une seconde plus tard.
Par exemple, si l'utilisateur tape 21 puis 32 puis 8, l'algorithme doit répondre : "Dans une seconde, il sera 21 heure(s) 32 minute(s) et 9 secondes".
NB : on suppose que l'utilisateur entre une heure valide. Pas besoin donc de la vérifier.

Exercice 11 :

Écrire un algorithme qui demande 3 nombres entiers à l'utilisateur et qui les trie par ordre croissant. Un affichage devra être prévu.

Exemple : Si à un instant donné trois variables sont initialisées telles que :

```
1  a=4
2  b=7
3  c=1
```

alors après le traitement, les instructions suivantes :

```
1  print('a='+str(a)+' b='+str(b)+' c='+str(c))
```

afficheront :

```
1  a=1 b=4 c=7
```

Exercice 12 :

Écrire un algorithme qui calcule et affiche les racines réelles d'un polynôme du second degré.

Prenez soin de bien analyser le problème, et posez-vous la question de savoir ce qui est demandé très précisément pour répertorier les données du problème.

Rappel : Ce sont les racines réelles de l'équation a*x*x + b*x + c = 0

TP3 : Les alternatives

Exercice 13 : Eric le retour...

Dans cet exercice nous allons :
- suivre l'évolution des variables booléennes d'un algorithme à l'aide du débogueur de Eric,
- tester la validité des relations booléennes en changeant les données en entrée.

Pour cela nous allons utiliser les Exercices 2 et 7.

Question 13.1 : Saisir l'algorithme de l'Exercice 2 et faire une exécution pas à pas comme vous l'avez appris lors du premier TP. Visualiser le contenu de la variable *test*. Changer la valeur de x avec des valeurs significatives et refaire les essais.

Valeurs significatives : Valeurs que le programmeur devra choisir de façon pertinente afin de valider un algorithme.

Question 13.2 : Saisir l'algorithme de l'Exercice 7 et faire une exécution pas à pas comme vous l'avez appris lors du premier TP. Visualiser le contenu des variables et répondre aux questions suivantes :

1. A quoi sert la variable *interrupteur* ?
2. Comment le programme agit-il sur la variable *interrupteur* ?
3. Quand la variable *intensite* est-elle modifiée ?
4. En conclusion, à quoi sert ce programme ?

Exercice 14 : Facture pour la reprographie

Un magasin de reprographie facture 0,10 Euro les dix premières photocopies, 0,09 Euro les vingt suivantes et 0,08 Euro au-delà. Ecrivez un algorithme qui demande à l'utilisateur le nombre de photocopies effectuées et qui affiche la facture correspondante. Vous veillerez à faire le moins possible de tests.

Exercice 15 : L'impôt au pays de *Zorglubland*

Les habitants de *Zorglubland* paient l'impôt selon les règles suivantes :
- Les hommes de plus de 20 ans paient l'impôt,
- les femmes paient l'impôt si elles ont entre 18 et 35 ans,
- les autres ne paient pas d'impôt,
- le programme demandera donc l'âge et le sexe du *Zorglubien*, et se prononcera donc ensuite sur le fait que l'habitant est imposable ou pas.

Chapitre 3

Boucles Simples

3.1 Introduction

Les boucles sont un élément central de la programmation. Elles permettent d'*effectuer un même ensemble d'instructions* (affichages, affectations, calculs) *plusieurs fois de suite*. Les boucles permettent donc d'éviter le copier/coller d'un ensemble d'instructions. Ainsi, si je dois afficher à l'écran 100 fois la phrase "hello world", je peux écrire une seule fois l'instruction `print('hello world')` et spécifier que cette instruction doit être effectuée 100 fois par l'ordinateur. Le code est ainsi plus concis et beaucoup plus rapide à écrire. De plus, avec les boucles, le nombre de fois que ces instructions doivent être répétées peut ne pas être connu au moment de l'écriture du code (contrairement au copier/coller). Ainsi, il est possible par exemple de spécifier que certaines instructions doivent être répétées autant de fois qu'indiqué par une valeur saisie par l'utilisateur.

Les boucles sont un raisonnement logique indépendant de la programmation et sont utilisées dans la vie quotidienne. En effet, lorsque l'on est arrêté à un feu tricolore en voiture, le code de la route spécifie "tant que le feu est rouge, je reste à l'arrêt". Cette spécification peut être vue comme une boucle. L'instruction "je reste à l'arrêt" est effectuée autant de fois que nécessaire, et ce, jusqu'à ce que la condition "le feu est rouge" soit fausse.

Le type de boucles présenté ici reprend le même schéma de raisonnement que l'exemple cidessus. Cette boucle *while* (while signifiant "tant que" en anglais), s'écrit de la manière suivante :

```
1 Instruction 0
2 while(condition):
3     Instruction 1
4     Instruction 2
5     ...
6     Instruction n
7 Instruction n+1
```

Remarque importante : L'indentation des instructions 1 à n est obligatoire. Cela permet à l'interpréteur Python de distinguer le bloc d'instructions associé à la boucle while.

Ce code indique que les instructions indentées par rapport au mot-clé "while" (*i.e.*, les instructions 1 à n) sont répétées tant que la "condition" est vraie. Ces instructions peuvent correspondre à du calcul, des affectations, des alternatives ou à d'autres boucles. L'ordre dans lequel ces instructions sont exécutées par l'ordinateur est représenté dans la figure 3.1. Lorsque le programme arrive à l'instruction while, il évalue la condition associée. Si cette dernière est vraie, alors les instructions 1 à n sont exécutées puis la condition est de nouveau évaluée. Lorsque la condition est fausse, le programme exécute directement l'instruction n+1.

Chaque exécution des instructions 1 à n associées au while est appelée *itération*. La condition est très souvent donnée par la valeur d'une variable dont on modifie la valeur à chaque itération.

1. Les numéros de la figure correspondent aux numéros de lignes du code while précédent.

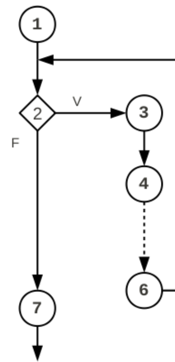


FIGURE 3.1 – Principe de la boucle while [1]

Par exemple, si je souhaite afficher 4 fois la phrase "hello world", je peux donc considérer une variable, dont la valeur est fixée initialement à 0, et incrémenter cette variable de un à chaque itération. Le bloc d'instructions associé au while devra alors être exécuté tant que la variable est inférieure strictement à 4. On obtient alors le code suivant :

```
print ('********************')
i = 0
while(i <4 ):
    print 'hello world'
    i += 1
print ('********************')
```

L'exécution de ce code produit alors l'affichage suivant :

```
********************
hello world
hello world
hello world
hello world
********************
```

Pour comprendre exactement ce qui se passe durant l'exécution, exécutons à la main l'exemple précédent. Les instructions sont alors les suivantes :
instruction 1 : affichage de `'********************'`
instruction 2 : définition d'une variable entière de valeur 0
instruction 3 : test ($i < 4$) avec $i = 0$ est vrai => exécuter les instructions 4 et 5
instruction 4 : affichage de 'hello world'
instruction 5 : modifie la valeur de la variable i : i vaut 1
boucle instruction 3 : test ($i < 4$) avec $i = 1$ est vrai => exécuter les instructions 4 et 5
instruction 4 : affichage de 'hello world'
instruction 5 : modifie la valeur de la variable i : i vaut 2
boucle instruction 3 : test ($i < 4$) avec $i = 2$ est vrai => exécuter les instructions 4 et 5
instruction 4 : affichage de 'hello world'
instruction 5 : modifie la valeur de la variable i : i vaut 3
boucle instruction 3 : test ($i < 4$) avec $i = 3$ est vrai => exécuter les instructions 4 et 5
instruction 4 : affichage de 'hello world'
instruction 5 : modifie la valeur de la variable i : i vaut 4
boucle instruction 3 : test ($i < 4$) avec $i = 4$ est faux => exécuter l'instruction 6 (et suivantes)
instruction 6 : affichage de `'********************'`'

► Sur ce thème : **EXERCICE 1, TD3**

▶ Sur ce thème : **Exercice 2, TD3**

3.2 Construire une boucle : exemple simple

Dans cette partie, nous montrons le raisonnement relatif à l'élaboration d'une boucle simple. Nous nous concentrons sur l'écriture d'un programme permettant d'afficher tous les nombres pairs entre 1 et 10 inclus. Le premier code venant à l'esprit est donc le suivant :

```
print(2)
print(4)
print(6)
print(8)
print(10)
```

Cependant, c'est loin d'être la meilleure technique ! De plus, si la question consiste à afficher tous les nombres pairs entre 1 et 10000 inclus, cette technique necessitera plusieurs heures pour écrire le programme ! Par ailleurs, comme il est clair que l'on répète plus ou moins la même instruction, à savoir afficher un nombre, il est possible (en modifiant légèrement le code), d'utiliser une boucle pour afficher ces nombres pairs.

Afin de pouvoir créer une boucle, il faut que les instructions effectuées à chaque itération soient identiques. Pour cela, il convient tout d'abord de modifier le programme afin de faire en sorte que les affichages soient strictement identiques. On utilise alors une variable, disons i, pour stocker la partie qui ne se répète pas (2, 4,...,10). Ainsi, l'affichage devient identique pour tous les nombres, à savoir `print(i)`. Le code est alors le suivant :

```
i = 2
print(i)
i = 4
print(i)
i = 6
print(i)
i = 8
print(i)
i = 10
print(i)
```

On répète bien 5 fois la même instruction `print(i)`. Par contre, on effectue 5 affectations différentes de la variable i. Mais on se rend finalement compte que l'affectation consiste à ajouter deux à la valeur de i. On peut donc remplacer les 5 affectations différentes par l'affectation `i+=2`. Pour que cela soit toujours vrai, il faut par contre que i soit initialement égal à 0. On obtient alors le code suivant :

```
i = 0

i += 2
print(i)

i += 2
print(i)

i += 2
print(i)

i += 2
print(i)

i += 2
```

```
16 print(i)
```

Dans ce code, il y a bien un bloc de deux instructions répété 5 fois. Avant le dernier bloc d'instructions (ligne 14), la variable i est égale à 8. On répète donc ces instructions tant que la variable i est inférieure ou égale à 8. Nous pouvons donc remplacer ce code par la boucle :

```
1 i = 0
2 while (i <= 8) :
3       i += 2
4       print(i)
```

La difficulté de l'élaboration d'une boucle vient du fait qu'il faut être capable de modifier les instructions afin qu'elles soient identiques. L'exemple présenté ici est simple si l'on comprend que la variable i est incrémentée de deux à chaque itération. Cette "transformation" du code (en instructions identiques) peut cependant être difficile si l'on n'a pas la bonne "intuition".

Dans le code précédent, à chaque itération, on effectue d'abord l'incrémentation de la variable i puis l'affichage de i. L'ordre dans lequel ces deux instructions est fait est complètement arbitraire. On pourrait, tout aussi bien considérer qu'à chaque itération, on affiche d'abord i avant de l'incrémenter. Comme on souhaite d'abord afficher 2, il faut par contre affecter la valeur 2 à i avant la boucle while. De plus, comme le dernier entier à afficher est 10, il est nécessaire d'effectuer le bloc d'instruction tant que i est inférieur ou égal à 10. On obtient alors le code suivant :

```
1 i = 2
2 while (i <= 10) :
3     print(i)
4     i += 2
```

▶ Sur ce thème : **Exercice 3, TD3**
▶ Sur ce thème : **Exercice 4, TD3**

3.3 Construire une boucle : autre exemple

Jusqu'à maintenant, dans les différents exemples de boucles que nous avons vus, le nombre d'itérations était connu à l'avance. Cependant, ceci n'est pas toujours vrai. Dans ce cas, il n'est plus possible de choisir une variable et de la modifier jusqu'à ce qu'elle atteigne une certaine valeur. D'autres types de conditions associées au while sont nécessaires. Pour illustrer ceci, nous nous intéressons à l'écriture d'un programme permettant de calculer la moyenne d'un étudiant. Le nombre de notes d'un étudiant n'est pas connu à l'avance. L'utilisateur doit saisir au clavier les différentes notes (nombres réels compris entre 0.0 et 20.0). Le programme doit arrêter la saisie lorsque le nombre saisi par l'utilisateur ne correspond pas à une note. Il doit alors afficher la moyenne des notes saisies.

Comme l'utilisateur peut saisir plusieurs notes (par exemple 10), il y a forcément une répétition de l'instruction de saisie d'un nombre, c'est-à-dire, de l'instruction `nb = float(raw_input())`. De plus, cette saisie doit continuer tant que le nombre saisi correspond à une note, c'est-à-dire, tant que la variable nb est supérieure ou égale à 0.0 et inférieure ou égale à 20.0. Une ébauche de la boucle est alors la suivante :

```
1 while ((nb >= 0) and (nb <= 20)) :
2     nb = float(raw_input())
```

La saisie de nombre ne suffit pas. Afin de calculer la moyenne de plusieurs notes, il faut connaître le nombre total de notes ainsi que la somme de ces notes. Il faut donc deux nouvelles variables, une correspondant au nombre de notes que l'on appelera `compteur`, et une correspondant à la somme des notes (variable `somme`). À chaque itération, il faut donc incrémenter la variable `compteur`, ajouter à `somme` la variable `nb` et demander à nouveau que l'utilisateur saisisse un nombre. La boucle devient alors

```
while ((nb >= 0) and (nb <= 20)) :
    compteur += 1
    somme += nb
    nb = float(raw_input())
```

Attention, l'ordre est très important ! Comme l'utilisateur peut saisir un nombre qui ne corresponde pas à une note, il est nécessaire que l'on vérifie que `nb` est une note avant de l'ajouter à `somme`. Pour cela, la saisie de `nb` est la dernière instruction du while. Si ce nombre est une note, autrement dit, si la condition du while est vraie, alors on incrémente compteur de un et on ajoute la dernière note saisie à `somme` à l'itération suivante.

Il faut alors maintenant considérer les valeurs initiales des variables. En effet, lors du premier test du while, la variable `nb` n'est pas définie. Elle doit correspondre à la première note saisie par l'utilisateur. Il faut donc ajouter avant la boucle l'instruction `nb = float(raw_input())`. Il faut également donner des valeurs initiales pour `compteur` et `somme`. Afin que le résultat soit correct, il faut les initialiser à 0. On a alors

```
nb = float(raw_input())
compteur = 0
somme = 0
while ((nb >= 0) and (nb <= 20)) :
    compteur += 1
    somme += nb
    nb = float(raw_input())
```

À la fin de cette boucle, les variables `somme` et `compteur` correspondent respectivement à la somme des notes saisies et à leur nombre. Pour afficher la moyenne, il faut donc afficher `print('La moyenne est ' + str(somme/compteur))`. Cependant, il faut impérativement que le programme fonctionne quelles que soient les notes saisies par l'utilisateur. Il est donc possible que l'utilisateur ne saisisse aucune note. Dans ce cas, `compteur` est nul et l'affichage engendre une division par 0, générant alors une erreur. Un test sur le nombre de notes saisies est alors nécessaire. Le code après la boucle est alors

```
if(compteur==0) :
      print('Aucune note n\'a ete saisie. On ne peut donc pas calculer la moyenne!!!')
else :
      print('La moyenne est ' + str(somme/compteur))
```

- ▶ Sur ce thème : **EXERCICE 5, TD3**
- ▶ Sur ce thème : **EXERCICE 6, TD3**
- ▶ Sur ce thème : **EXERCICE 7, TD3**

3.4 Boucles infinies

Les boucles infinies sont des boucles dont les instructions sont exécutées une infinité de fois. De telles boucles interviennent lorsque la condition du while est toujours vraie. Elles correspondent à une erreur de programmation et ne sont pas détectées par l'ordinateur. De telles erreurs sont dues soit à une mauvaise initialisation de la ou des variables intervenant dans la condition, soit à un mauvais test. Voici un exemple de boucle infinie :

```
    i = 1
    while (i != 10) :
       i += 2
       print(i)
```

Un peu (ou beaucoup !) de reflexion avant l'écriture de la boucle permet d'éviter de telles boucles infinies.

TD4 : Boucles simples

Exercice 1 :

Effectuer la trace de ce programme :

```
i = 10
nb = 1
while (i<5) :
    i += 1
    nb += 2 * i
print ('nb = ' + str(nb))
```

Même question lorsque la première instruction est respectivement remplacée par l'instruction `i = 0` et `i = 3`.

Exercice 2 :

Effectuer la trace de ce programme :

```
i = 12
j = 43
print ('i = ' + str(i) + ', j = ' + str(j))
while ((i<25) and (j > 3 * i)) :
    i += 3
    j -= i - 18
    print ('i = ' + str(i) + ', j = ' + str(j))
```

Exercice 3 :

Afficher les entiers de 1 à 20 dans l'ordre décroissant.

Exercice 4 :

Afficher la table de multiplication de 7 jusqu'à 20 comme suit :
1 * 7 = 7
2 * 7 = 14
...
19 * 7 = 133
20 * 7 = 140

Exercice 5 :

Écrire un programme demandant à l'utilisateur de saisir un nombre entier positif. La saisie sera répétée jusqu'à ce que le nombre soit positif. Même question pour un nombre entier positif et multiple de 3.

Exercice 6 :

Écrire un programme permettant de vérifier si un nombre saisi par l'utilisateur est premier. *Rappel :* Un nombre est dit *premier* si ses deux seuls diviseurs entiers positifs sont 1 ou lui-même. Le nombre 13 est donc un nombre premier, alors que 6 ne l'est pas puisque $6 = 2 \times 3$.

Exercice 7 :

(Exercice basé sur un exercice donné en IUT de mesures physiques à Orsay.)

Écrire un programme demandant à l'utilisateur un nombre compris entre 1 et 50 et affichant ce nombre en chiffres romains.

TP4 : Boucles et dessins

Exercice 8 :

Écrire un programme simulant le lancé de 5 dés. Afficher le résultat de chaque dé ainsi que la somme des 5 lancés obtenus. Pour simuler un lancer de dé, on génère un nombre aléatoire compris entre 1 et 6 inclus. Cela s'effectue en python à l'aide de l'instruction `tmp = randint(1,6)` qui affecte à la variable tmp un nombre entier aléatoire entre 1 et 6 inclus. Afin de pouvoir utiliser cette instruction, il est nécessaire d'ajouter en début de programme l'instruction `from random import *`.

Exercice 9 :

Écrire un programme permettant de calculer la factorielle d'un nombre saisi par l'utilisateur. *Rappel :* si n est un entier positif, alors la *factorielle* de n, notée $n!$, est égale à :

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$$

Par convention, $0! = 1$.

Exercice 10 :

Écrire un programme qui, pour tout entier compris entre 1 et 10, affiche sur une même ligne, les valeurs de cet entier, de son carré et de son cube. L'affichage doit donc être équivalent à :

```
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

Afin d'obtenir l'affichage correct, il est possible de spécifier en python le nombre de caractères à utiliser pour afficher une chaîne de caractères, cette chaîne étant alors justifiée à droite. Pour cela, il suffit d'utiliser l'instruction `rjust(str,n)` où `str` est une chaîne de caractères et `n` est le nombre de caractères à utiliser pour l'affichage de str. Afin de pouvoir utiliser cette instruction, il est nécessaire d'ajouter en début de programme l'instruction `from string import *`. À titre d'exemple, les instructions suivantes

```
from string import *
print ('x')
print (rjust('x',2))
print (rjust('x',3))
nb = 1
print (rjust(str(nb),3))
nb = 10
print (rjust(str(nb),3))
nb = 100
print (str(nb))
```

correspondent à l'affichage :

```
x
 x
  x
  1
 10
100
```

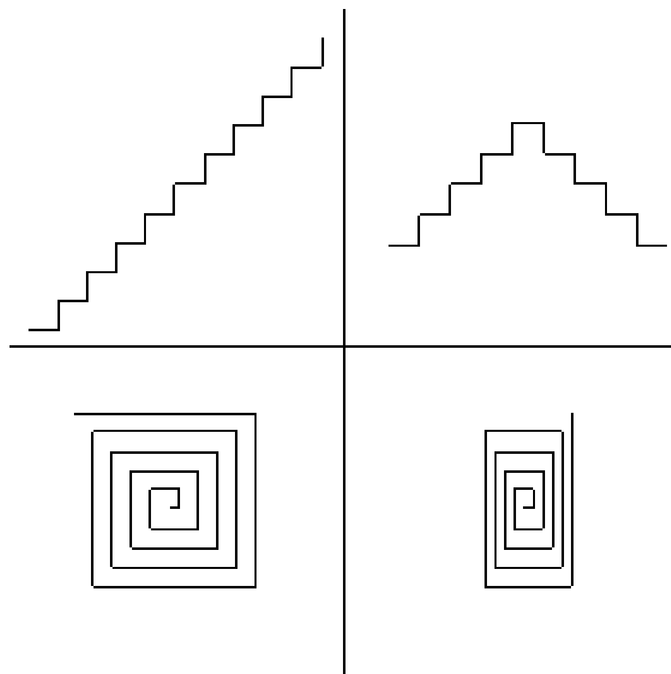
Exercice 11 :

En utilisant l'instruction `rjust(n)` expliquée précédemment, écrire un programme reproduisant l'affichage suivant :

```
 *
   *
     *
       *
         *
           *
             *
               *
                 *
                   *
                     *
```

Exercice 12 : Utilisation de Turtle

Reproduire à l'aide du module turtle les figures ci-dessous. Le module Turtle permet de dessiner



à l'écran des figures. Afin de pouvoir uiliser ce module, il faut ajouter, en début de programme, l'instruction : `from turtle import *`. On peut alors utiliser les instructions suivantes pour dessiner :

– `goto(x,y)` : aller à l'endroit de coordonnées x et y,

- `forward(distance)` : avancer d'une distance donnée,
- `up()` : relever le crayon (pour pouvoir avancer sans dessiner),
- `down()` : abaisser le crayon (pour pouvoir recommencer à dessiner),
- `left(angle)` : tourner à gauche d'un angle donné (exprimé en degré),
- `right(angle)` : tourner à droite.

À titre d'exemple, regarder ce que donne le code :

```
from turtle import *
forward(100)
left(90)
forward(50)
goto(0, 0)
```

Exercice 13 : Utilisation de Turtle (suite)

Écrire un programme demandant un nombre compris entre 3 et 10 à l'utilisateur et affichant, à l'aide du module turtle, un polygone ayant un nombre de côtés égal à la valeur saisie par l'utilisateur.

Chapitre 4

Tableaux

Les variables que nous avons utilisées jusqu'à maintenant ne permettent de stocker qu'une seule donnée par variable. Ainsi, si je dois stocker le score de deux joueurs, je dois donc créer deux variables, par exemple `scoreJoueur1` et `scoreJoueur2`. Si ce moyen est possible pour quelques données, il n'est par contre pas possible d'utiliser une variable par donnée à stocker lorsque le nombre de données est très grand. En effet, si l'on doit stocker le score de 1000 joueurs, il est impossible d'écrire un algorithme utilisant 1000 variables, chacune stockant le score d'un des joueurs. Dans ce cas, il est nécessaire d'utiliser les tableaux.

Un tableau est un type de données contenant plusieurs valeurs. Ces valeurs peuvent être de différents types (entiers, réels, chaînes de caractères). Un tableau est défini par les symboles `[` et `]`. Entre ces crochets sont définies les valeurs contenues dans le tableau. Ces valeurs sont séparées par une virgule et sont appelées *éléments du tableau*. Le tableau `['abc', True, 12.3, 3]` contient 4 éléments, respectivement de type chaîne de caractères, booléen, réel et entier.

4.1 Variables de type tableau et accès aux éléments

Comme pour les autres types de données, les tableaux peuvent être stockés dans une variable. Ceci se fait en affectant à une variable un tableau. Ainsi, l'instruction

```
1 tab = ['bonjour','au revoir','salut']
```

définit un tableau contenant trois chaînes de caractères qui est affecté à la variable de nom `tab`.

On peut accéder aux éléments d'un tableau affecté à une variable par leur numéro de position dans celui-ci ; on appelle ce numéro *indice* du tableau. Il suffit pour cela de donner le nom de la variable de type tableau suivi de l'indice entre crochets. On obtient alors `variableTableau[indice]`. Une chose est cependant à retenir : l'indice d'un tableau de n éléments commence à 0 et se termine à $n-1$. On peut accéder à un élément d'un tableau pour connaître sa valeur ou la modifier. L'exécution du code (où `tab` est la variable définie précédemment)

```
1 print('le premier élément du tableau tab est : ' + tab[0])
2 print('le deuxième élément du tableau tab est : ' + tab[1])
3 print('le troisième élément du tableau tab est : ' + tab[2])
4 tab[0] = 'hello'
5 print('le premier élément du tableau tab est maintenant : ' + tab[0])
```

produira l'affichage suivant

```
1 le premier élément du tableau tab est : bonjour
2 le deuxième élément du tableau tab est : au revoir
3 le troisième élément du tableau tab est : salut
4 le premier élément du tableau tab est maintenant : hello
```

Il est également possible de définir la position de l'élément par rapport à la fin du tableau. Pour accéder au ième élément du tableau à partir de la fin, on utilise alors `variableTableau[-i]`. On parle alors d'*indice négatif*. Attention :

- afin qu'il n'y ait pas d'ambuguïté avec les indices, l'indice négatif d'un tableau de n éléments commence à -1 et se termine à$-n$
- l'existence d'indices négatifs est une spécificité de Python. La plupart des langages n'acceptent qu'un seul indice par position du tableau.

L'exécution du code (où `tab` est la variable définie précédemment)

```
1 print('le dernier élément du tableau tab est : ' + tab[-1])
2 print('l\'avant dernier élément du tableau tab est : ' + tab[-2])
3 print('le premier élément du tableau tab est : ' + tab[-3])
```

produira l'affichage suivant

```
1 le dernier élément du tableau tab est : salut
2 l'avant dernier élément du tableau tab est : au revoir
3 le premier élément du tableau tab est : bonjour
```

▶ Sur ce thème : **Exercice 1, TD4**

4.2 Affichage d'un tableau

Comme les autres types de données, il est possible d'afficher un tableau à l'aide de la fonction `print()`. Ainsi, l'exécution du code

```
1 animaux = ['girafe','hippopotame','singe',chat']
2 print(animaux)
```

affichera à l'écran `['girafe','hippopotame','singe',chat']`.

Il est également possible d'utiliser la fonction `str()` afin de transformer le tableau en une chaîne de caractères. Cette chaîne est alors identique au résultat de la fonction `print()`. En effet, en plus des éléments, la chaîne de caractères contient les crochets et les virgules. De plus, les éléments du tableau correspondant à des chaînes de caractères sont encadrés par des apostrophes. L'intérêt de la fonction `str()` est de pouvoir afficher des chaînes de caractères contenant la valeur d'un tableau. Ainsi, l'instruction `print('le tableau vaut : ' + str(animaux))` affichera `le tableau vaut : ['girafe','hippopotame','singe',chat']`.

4.3 Nombre d'éléments et parcours d'un tableau

La fonction `len()` permet de donner le nombre d'éléments d'un tableau. Il faut par contre donner entre les parenthèses le nom du tableau dont on veut connaître le nombre d'éléments. L'exécution de l'instruction `print(len(tab))` affichera alors 3 puisque le tableau `tab` contient 3 éléments, à savoir 'bonjour', 'au revoir', et 'salut'.

Il est très souvent nécessaire d'accéder à tous les éléments d'un tableau, que ce soit pour calculer la somme des éléments du tableau (si ces derniers sont des nombres), déterminer le plus grand élément ou pour savoir si une valeur apparaît dans un tableau. Pour accéder à tous les éléments d'un tableau, une façon consiste à utiliser une boucle `while` en faisant varier l'indice de 0 à la longueur du tableau - 1. Ainsi, le code suivant permet d'afficher tous les éléments d'un tableau, les uns en dessous des autres.

```
1 animaux = ['girafe','hippopotame','singe','chat']
2 print('Voici les animaux contenus dans le tableau')
3 indice = 0
```

```
while indice < len(animaux) :
       print(animaux[indice])
      indice = indice + 1
```

▶ Sur ce thème : **Exercices 2, 3 et 4, TD4**

4.4 Opérateurs + et *

Les tableaux, tout comme les chaînes, supportent les opérations de concaténation (opérateur +) et de répétition (opérateur *). L'opération de concaténation `t1 + t2`, lorsque `t1` et `t2` sont des tableaux, crée un nouveau tableau dont les éléments correspondent aux éléments de `t1` suivis des éléments de `t2`. L'opération de répétition `t * n`, où `t` est un tableau et `n` est un entier positif ou nul, crée un nouveau tableau de taille `n * len(t)` dont les éléments correspondent aux éléments de `t` répétés `n` fois. Ainsi, l'exécution du code

```
print(['a',1,-3] + ['coucou',24,12])
print([0,1] * 6)
```

produit l'affichage

```
['a', 1, -3, 'coucou', 24, 12]
[0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1]
```

L'opération de répétition est notamment utile pour créer un tableau dont tous les éléments ont la même valeur.

▶ Sur ce thème : **Exercice 5, TD4**

4.5 Affectation d'une variable de type tableau à une autre variable de type tableau

Le type tableau modifie quelque peu l'opérateur d'affectation lorsque l'opérande de droite est une variable de type tableau. En effet, si l'on a l'instruction `t2 = t`, où `t` est une variable de type tableau, la variable `t2` contient alors exactement le même tableau que la variable `t` ! Ceci implique que si l'on modifie un élément de `t`, on modifie en même temps le tableau t2, et vice versa. Ainsi, l'exécution du code

```
t = [1,2,3]
t2 = t
t[0] = 18
print(t)
print(t2)
```

produit l'affichage

```
[18, 2, 3]
[18, 2, 3]
```

Ceci n'est vrai que si l'on affecte à la variable `t2` une variable de type tableau. Si l'on affecte à `t2` un autre tableau (même si ce nouveau tableau est identique à `t`), le "problème" n'apparaît plus. Par exemple, le code

```
t = [1,2,3]
t2 = [1,2,3] #ou t2 = t * 1 ou t2 = t + []
t[0] = 18
print(t)
print(t2)
```

produira, après exécution, l'affichage

```
[18, 2, 3]
[1, 2, 3]
```

▶ Sur ce thème : **Exercice 6, TD4**

4.6 Opérateur in

L'opérateur `in` permet de tester si une valeur apparaît dans un tableau. Il s'utilise selon le modèle `valeur in tableau` et renvoie `True` si la valeur apparaît dans le tableau et `False` sinon. Par exemple, en exécutant le code

```
liste = [18.4,'ab',33,12]
if 19 in liste :
       print('19 est une valeur du tableau')
else :
       print('19 n\'est pas une valeur du tableau')
if 33 in liste :
       print('33 est une valeur du tableau')
else :
       print('33 n\'est pas une valeur du tableau')
nb = 12
if nb in liste :
       print('Un élément du tableau a la même valeur que nb')
```

On obtient alors

```
19 n'est pas une valeur du tableau
33 est une valeur du tableau
Un élément du tableau a la même valeur que nb
```

4.7 Boucle for

Nous avons vu précédemment comment parcourir un tableau avec une boucle `while` en faisant varier l'indice. Il existe cependant une autre façon de parcourir le tableau, qui consiste à affecter successivement à une variable la valeur de tous les éléments du tableau. Ce type de parcours se fait à l'aide de l'instruction `for val in tab :`, où `val` est une variable et `tab` est un tableau. Cette instruction peut se traduire par "pour la variable `val` prenant successivement toutes les valeurs apparaissant dans le tableau `tab`, faire". L'exemple suivant permet d'afficher tous les éléments d'un tableau, les uns en dessous des autres. (On remarque que le code donne exactement le même résultat que dans l'exemple donné avec la boucle `while`.)

```
animaux = ['girafe','hippopotame','singe','chat']
print('Voici les animaux contenus dans le tableau')
for animal in animaux :
       print(animal)
```

L'utilisation de la boucle `for` permet d'écrire un code plus clair et plus rapide pour le parcours d'un tableau. Attention, il convient de bien comprendre la différence entre la boucle `while` et la boucle `for` décrites ci-dessus. La première utilise dans le test une variable correspondant à l'indice du tableau et accède aux éléments grâce à `tableau[indice]` alors dans la boucle `for`, la variable utilisée par la boucle prend "directement" la valeur des éléments du tableau.

Attention, dans la boucle `for`, la variable de boucle prend successivement la valeur de tous les éléments du tableau, mais elle ne représente pas l'élément lui-même. Ceci implique que même si l'on modifie la variable à l'intérieur de la boucle, le tableau n'est pas modifié. Par exemple, à la fin de l'exécution du code

```
t = [1,2,3,4]
for nb in t :
      nb = nb +10
```

le tableau affecté à la variable `t` n'a pas été modifié. Il est toujours égal à `[1,2,3,4]`.

▶ Sur ce thème : **Exercice 7, TD4**

4.8 Fonctions de modifications d'un tableau

Il existe différentes fonctions permettant de modifier un tableau. (Ces fonctions ne créent pas un nouveau tableau.) Les noms de ces fonctions respectent le modèle `list.nomFonction` car elles s'appliquent aux tableaux (traduction de `list`).

- `list.insert(tab,ind,val)` : insère dans le tableau `tab` un élément de valeur `val` à l'indice `ind`
- `list.pop(tab,ind)` : supprime dans le tableau l'élément qui se trouve à l'indice `ind`
- `list.sort(tab)` : trie le tableau `tab`
- `list.reverse(tab)` : inverse l'ordre des éléments du tableau `tab`

L'exécution du code

```
x = [132,18,24]
print(x)
list.insert(x,0,36)
print(x)
list.insert(x,1,-6)
print(x)
list.insert(x,1,'ze')
print(x)
list.pop(x,3)
print(x)
list.pop(x,3)
print(x)
list.sort(x)
print(x)
list.reverse(x)
print(x)
```

donne l'affichage suivant

```
[132, 18, 24]
[36, 132, 18, 24]
[36, -6, 132, 18, 24]
[36, 'ze', -6, 132, 18, 24]
[36, 'ze', -6, 18, 24]
[36, 'ze', -6, 24]
[-6, 24, 36, 'ze']
['ze', 36, 24, -6]
```

▶ Sur ce thème : **Exercice 8, TD4**

4.9 Création automatique de tableaux d'entiers

La fonction `range()` permet de créer des tableaux d'entiers (et d'entiers uniquement) de manière simple et rapide. Elle fonctionne sur le modèle : `range([début,] fin[, pas])`. Elle crée alors un tableau contenant tous les entiers entre `début` compris et `fin` non compris, selon le pas `pas`. Les arguments entre crochets sont optionnels. Si l'argument `début` n'est pas spécifié, il est alors considéré comme égal à 0. Si le pas `pas` n'est pas spécifié, il est alors considéré comme égal

à 1. Attention, si seulement deux arguments sont spécifiés, ils correspondent alors aux arguments `début` et `fin`. En exécutant le code

```
print(range(0,10))
print(range(10))
print(range(15,21))
print(range(0,1000,100))
print(range(2,-5,-1))
```

on obtient l'affichage

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[15, 16, 17, 18, 19, 20]
[0, 100, 200, 300, 400, 500, 600, 700, 800, 900]
[2, 1, 0, -1, -2, -3, -4]
```

La fonction `range()` est aussi utile pour créer une boucle `for`. Ainsi, pour exécuter 10 fois une instruction, par exemple afficher bonjour, il est possible d'utiliser les instructions

```
for nb in range(10) :
      print('bonjour')
```

En effet, l'instruction `print('bonjour')` sera effectuée pour chaque valeur prise par la variable `nb`. Or, dans ce cas, `nb` prend successivement comme valeur tous les éléments du tableau `range(10)`, à savoir : 0, 1, ..., 9.

De la même façon, la fonction `range()` permet de créer une boucle `for` qui parcourt les indices d'un tableau. L'exemple suivant permet d'afficher tous les éléments d'un tableau, les uns en dessous des autres. (On remarque que ce code donne exactement le même résultat que dans les deux exemples donnés précédemment pour la boucle `while` et la boucle `for`.)

```
animaux = ['girafe','hippopotame','singe','chat']
print('Voici les animaux contenus dans le tableau')
for indice in range(len(animaux)) :
      print(animaux[indice])
```

En effet, `range(len(animaux))` correspond au tableau [0,1,2,3]. Les différents éléments de ce tableau correspondent aux différents indices des éléments de `animaux`. La variable `indice` prend alors successivement les valeurs 0, 1, 2 puis 3. Pour chacune des valeurs de `indice`, on affiche l'élément du tableau `animaux` qui apparaît à l'indice `indice`.

► Sur ce thème : **Exercices 9 et 10, TD4**

4.10 Chaînes de caractères

Les chaînes de caractères peuvent être considérées comme des tableaux. Il est donc possible d'accéder aux caractères grâce à leurs indices et de créer de nouvelles chaînes de caractères à l'aide des tranches et des opérations de concaténation et de répétition. Cependant, ces tableaux sont **constants**, c'est-à-dire qu'il n'est pas possible de modifier les éléments (*i.e.*, les caractères). L'instruction `ch[0] = 'a'` n'est donc par exemple pas permise si `ch` est une chaîne de caractères. Par ailleurs, la fonction `print()` n'affiche pas la chaîne sous forme de tableau (L'affichage ne contient donc ni les crochets, ni les apostrophes). L'exécution du code

```
ch = 'bonjour'
print(ch)
print(ch[0])
print(ch[3] + ch[-1])
```

affiche alors

```
bonjour
b
jr
```

Comme les chaînes de caractères sont des tableaux constants, si l'on souhaite modifier un caractère, il faut alors construire une nouvelle chaîne. Par exemple, si l'on souhaite afficher la chaîne `ch` en remplaçant le caractère à la position 3 par la lettre 'k', on peut alors par exemple utiliser l'instruction `print(ch[0] + ch[1] + ch[2] +'k' + ch[4] + ch[5] + ch[6])`.

▶ Sur ce thème : **Exercices 11, 12 et 13, TD4**

TD5 : Tableaux

Exercice 1 :

Affecter à une variable appelée `semaine` un tableau contenant les jours de la semaine. Afficher le premier élément et le dernier élément de `semaine` en utilisant successivement les indices et les indices négatifs.

Exercice 2 :

Reprendre la variable `semaine` définie dans l'exercice 1. Afficher les jours de la semaine en indiquant devant le numéro du jour. L'affichage doit être équivalent à :

```
1 - lundi
2 - mardi
3 - mercredi
4 - jeudi
5 - vendredi
6 - samedi
7 - dimanche
```

Inverser ensuite l'ordre d'affichage (commencer par le dimanche pour finir par le lundi) (deux parcours d'indices différents avec une boucle while).

Exercice 3 :

Affecter à la variable `tab` le tableau `[5,6,8,14,17]`. Calculer ensuite la somme des nombres de ce tableau et l'afficher.

Exercice 4 :

Écrire un programme qui recherche le plus grand élément présent dans un tableau donné. Par exemple, si le tableau est égal à `[32, 5, 12, 8, 3, 75, 2, 15]`, ce programme doit afficher : `le plus grand élément du tableau a la valeur 75`.

Exercice 5 :

Écrire un programme qui permet à un utilisateur de saisir les notes d'un étudiant et de les stocker dans un tableau. La saisie s'arrête lorsque l'utilisateur a saisi un nombre strictement inférieur à 0 ou strictement supérieur à 20. Afficher alors le tableau si celui-ci n'est pas vide.

Exercice 6 :

Donner la valeur des variables `t1`, `t2` et `t3` après exécution du code

```
t1= ['a', 'b', 'c', 'd']
t2 = [34, 31, 957]
t3 = t1
t3[1] = -3
t1 = t2
t2[0] = 'abc'
t3 = t2 + []
t2[0] = 'bonjour'
t3[2] = 'oui'
```

Exercice 7 :

Modifier le code des exercices 3 et 4 en utilisant la boucle `for` à la place de la boucle `while`.

Exercice 8 :

Soit la variable de type tableau `x = [34,-2,0,39,0,78,0,0,46]`. Supprimer toutes les valeurs nulles de ce tableau puis l'afficher dans l'ordre décroissant.

Exercice 9 :

Utiliser la fonction `range()` pour afficher :
- les entiers de 0 à 3
- les entiers de 4 à 7
- les entiers de 2 à 8 par pas de 2

Exercice 10 :

Modifier le code de la première question de l'exercice 2 en utilisant la boucle `for` à la place de la boucle `while`.

Exercice 11 :

Considérons la chaîne `s = 'bonjour'`. Quel est le nombre d'éléments de `[s]` ?

Écrire un programme qui crée, à partir d'une chaîne de caractères, disons `s = 'bonjour'`, un tableau `tab` dont les éléments sont les lettres de la chaîne de caractères `s`. Créer ensuite une chaîne de caractères `s2`, dont les lettres sont les éléments de `tab`. Les chaînes `s` et `s2` doivent être équivalentes. Donner le résultat produit par l'affichage de `s`, `tab` et `s2`.

Exercice 12 :

Saisir une chaîne de caractères et l'afficher en remplaçant toutes les voyelles par un tiret. Ainsi, si la chaîne saisie est `Il fait beau aujourd'hui`, le programme doit afficher `-l f-t b---j-rd'h-`

Exercice 13 :

Faire saisir à l'utilisateur une chaîne de caractères et tester si cette chaîne correspond à un entier positif ou négatif.

TP5 : Tableaux

Exercice 14 :

Voici les notes d'un étudiant sous forme d'un tableau : `[8, 4, 12, 11, 9, 14, 19, 11, 19, 4, 18, 12, 19, 3, 5]`. Écrire un programme qui affiche la note maximum, la note minimum et et la moyenne de cet étudiant. Améliorer le programme ajoutant la mention : redoublement (< 10), passable (entre 10 et 12), assez-bien (entre 12 et 14), bien (entre 14 et 16) et très bien (supérieur ou égal à 16).

Exercice 15 :

Considérons les variables suivantes suivantes :

- `joursMois = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]`
- `nomMois = ['Janvier', 'Février', 'Mars', 'Avril', 'Mai', 'Juin', 'Juillet', 'Août', 'Septembre', 'Octobre', 'Novembre', 'Décembre']`

Écrire un programme qui crée un nouveau tableau. Celui-ci devra contenir tous les éléments des deux tableaux en les alternant, de telle manière que chaque nom de mois soit suivi du nombre de jours correspondant : `['Janvier',31,'Février',28,'Mars',31, etc...]`.

Exercice 16 :

Ecrivez un programme qui permet d'afficher les tailles des prénoms suivant : Jean-Michel, Marc, Vanessa, Anne, Maximilien, Alexandre-Benoit, Louise

Exercice 17 :

Écrire un programme qui crée, à partir d'un tableau de chaînes de caractères, deux nouveaux tableaux. L'un contiendra les mots comportant moins de 6 caractères, l'autre les mots comportant 6 caractères ou davantage. Si le tableau donné est `['Jean', 'Maximilien', 'Brigitte', 'Sonia', 'Jean-Pierre', 'Sandra']`, les deux tableaux créés doivent être : `['Jean', 'Sonia']` et `['Maximilien', 'Brigitte', 'Jean-Pierre', 'Sandra']`.

Exercice 18 :

Écrire un programme permettant à l'utilisateur de saisir un mot et indiquant si ce mot apparaît dans le tableau `['girafe', 'chien', 'hippopotame', 'singe', 'chat' , 'chien']`. Indiquer également le nombre d'occurences de ce mot dans le tableau.

À titre d'exemple, si l'utilisateur saisit `chien`, alors le programme doit afficher `chien apparaît 2 fois`. Si l'utilisateur saisit `lapin`, le programme doit afficher `lapin n'apparaît pas dans la liste`.

Exercice 19 :

Ecrivez un programme qui permet de tester si un mot est un palindrome, c'est-à-dire si sa lecture de gauche à droite donne le même résultat que sa lecture de droite à gauche. Par exemple, radar et rotor sont des palindromes.

Chapitre 5

Fonctions

5.1 Introduction

Avec les fonctions nous abordons un autre point essentiel de la conception des programmes.

Nous avons vu dans les cours précédents :
- *les séquences* d'instructions qui sont à la base d'un programme ; elles indiquent l'enchaînement des instructions, *instruction après instruction*,
- *les alternatives* qui permettent de choisir d'exécuter ou pas une séquence d'instruction(s) suivant le résultat de l'évaluation d'une condition (expression booléenne),
- *les boucles* qui permettent de répéter une séquence d'instruction(s) tant qu'une condition est vérifiée (expression booléenne).

Il s'agit dans les trois cas de *tronçons de code* qui ne sont utilisables qu'une fois au sein du programme ; dès que ces tronçons sont exécutés le programme se poursuit pour exécuter les instructions suivantes. On dit que ces *tronçons logiciels* ne sont pas réutilisables.

Les **fonctions** permettent, elles, de réaliser plusieurs fois le même tronçon de code au sein d'un programme. Vous connaissez, par exemple, les fonctions `raw_input()` et `print()` qui permettent respectivement de saisir ou d'afficher une chaîne de caractères à l'écran.

```
1 print('indiquer un prenom')  # affiche 'indiquer un prenom' à l'écran
2 mot=raw_input() #affecte le résultat de la saisie dans la chaine mot
3 print('le prenom est ' + mot) # affiche ' le prenom est' suivi de ce qui a été saisi 1' à l'écran
```

Dans ce qui suit, nous allons voir comment :

1. créer une fonction,
2. comment donner corps au traitement que doit exécuter la fonction,
3. comment utiliser ces fonctions.

Il ne faudra pas perdre de vue qu'une fonction permet de remplir **une tâche bien spécifique** ni plus, ni moins ; et à chaque fois que le programmeur aura besoin d'éxecuter cette tâche, il lui suffira d'appeler la fonction par son **nom** (au lieu de faire un copier coller des lignes d'instructions correspondantes). Le résultat est que le programme est **plus concis**, **plus lisible** et plus **facile à déboguer**. De plus, les fonctions développées pourront être **réutilisées** dans d'autres applications avec des **contextes différents**.

5.2 Concevoir une fonction

5.2.1 Définition d'une fonction

Quand on définit une fonction, on décide :

1. du **nom de la fonction** qui respecte les mêmes règles que les noms de variable,
2. des **arguments (donnés sous la forme d'une liste ordonnée de leurs noms éventuellement vide)** que la fonction doit recevoir,
3. du **traitement que doit effectuer la fonction**, elle utilisera pour cela les arguments éventuellement reçus.
4. du **renvoi ou non d'un résultat**.

.

La définition ressemblera un peu à cela :

```
def nomDeFonction (param1, param2, ...) :
    I1        # ici, on écrit du code utilisant les paramètres
    I2
    ....
    In
    return resultat # si la fonction renvoie un resulat
# le code de la fonction s'arrête à la première ligne non indentée
```

Pour définir une fonction en Python, on utilise le mot-clé `def`. Si l'on souhaite que cette fonction renvoie une valeur, on utilise le mot-clé `return` Notez que la syntaxe du `def` utilise les `:`, comme dans le cas boucles et des alternatives, un bloc d'instructions est donc attendu. L'indentation de ce bloc d'instructions, qui forme le *corps de la fonction*, est obligatoire.

Une fonction est appelée avec des **arguments** ou non : l'utilisateur fournit ou non des données à la fonction, et le déroulement de celle-ci ne sera pas le même suivant les données fournies.

Dans l'exemple de la fonction `print()` vu précédemment c'est le message à afficher qui représente l'argument et suivant la valeur de ce dernier la fonction `print()` n'affichera pas la même chose. On peut remarquer que la fonction `print()` n'attend qu'un seul argument.

Les noms de variables placés en argument pourront ainsi désigner à chaque appel des valeurs différentes ; on ne peut donc pas préjuger de leurs valeurs lors de l'écriture de la fonction c'est pour cela que ces arguments s'appellent des **paramètres formels** : **ils n'ont pas encore de contenu**.

Lors de l'utilisation d'une d'une fonction, l'utilisateur fournit les **valeurs des arguments**. Prenons un exemple :

```
def somme(x,y) :
    z = x + y
    return z

### Programme appelant #####
a = 7.3
b = 5.2
print(somme(a,b))      # Affiche 12.5
```

Le programme qui utilise la fonction s'appelle programme **appelant**.

Quelques commentaires à propos de l'exemple :

- La fonction ici s'appelle `somme` et possède deux paramètres formels `x` et `y`.
- Le corps de la fonction contient deux instructions dont le but est de calculer la somme des deux paramètres formels et de la renvoyer.
- Au cours du programme appelant, des arguments ont été fournis à la fonction `somme()` qui a retourné une valeur, ensuite affichée à l'écran. La valeur retournée par la fonction peut également être stockée dans une variable.

```
### Programme appelant #####
resultat=somme(7.3,5.2)
print(resultat) # Affiche 12.5
```

Notez qu'une fonction ne prend pas nécessairement d'argument et ne renvoie pas forcément une valeur. Toutefois il faut, lors de sa définition, mettre des parenthèses après le nom de la fonction. Comme cette fonction n'admet pas d'arguments, aucun nom de paramètre formel n'est donné entre les parenthèses. Par exemple :

```
def affiche_bonjour() :
    print('Bonjour')
```

Lors de l'appel d'une fonction sans argument on doit également taper les parenthèses.

Exemple d'utilisation :

```
affiche_bonjour()
```

L'affichage produit est le suivant :

```
Bonjour
```

Dans cet exemple, la fonction ne prend aucun argument et ne renvoie aucun résultat, se contentant d'afficher la chaîne de caractère 'Bonjour' à l'écran. cela n'a donc pas de sens de vouloir récupérer dans une variable le résultat renvoyé par une telle fonction.

Il est également possible de renvoyer une liste de résultats que le programme appelant affectera à autant de variables :

```
def quotientreste (a,b):
    q=a/b
    r=a%b
    return q,r

u,v = quotientreste(1253,87)
print('quotient : '+ str(u) +' reste :' +str(v) ) #affiche quotient : 14 reste : 35
```

▶ Sur ce thème : **Exercices 1 à 7, TD 5**

5.2.2 Portée des variables

Il est très important lorque l'on manipule des fonctions de connaître la portée des variables.

Variables locales

On peut créer des variables au sein d'une fonction qui ne seront pas visible à l'extérieur de celle-ci ; on les appelle **variables locales** Par exemple

```
def affiche_somme(x,y) :
    z = x + y
    print('la somme vaut '+ str(z))

### Programme appelant #####
affiche_(somme(7.3,5.2))
print(str(z))
```

L'affichage est alors le suivant :

```
la somme vaut 12.5
Traceback (most recent call last):
  File "<stdin>", line 7, in ?
Name error: name 'z' is not defined
```

Lorsque la fonction est exécutée, le contenu de la variable z est connu. De retour dans le programme appelant, la variable z n'est plus connue d'où le message d'erreur.

Deuxième ment lors de l'appel de la fonction, les variables passées en argument sont recopiées respectivement dans les paramètres formels et deviennent les **paramètres effectifs**. Il sont alors des variables locales. Mais même si la fonction modifie leurs valeurs, cela **ne modifiera pas** la valeur des variables `a` et `b` du programme appelant. Voici un autre exemple :

```
def incrementer(x):
    x=x+1              # la fonction modifie le contenu du paramètre formel
    print(str(x))      #
nbre=4
print(nbre)    # affiche 4
incrementer(nbre) # affiche 5
print(nbre) #  # affiche 4    , le paramètre effectif n'a pas été modifié !
```

Le cas des tableaux

Il est toutefois possible de modifier directement le contenu des éléments d'un tableau en utilisant directement le(s) paramètre(s) formel(s) ; cette modification s'appliquera au(x) paramètre(s) effectif(s). Pour cela, il faudra absolument utiliser la syntaxe `tab[indice] = expression`.

```
def changeEnPlace(t, ind):
    t[ind]= 'Jacques' # un élément de t est modifié, alors que t ne l'a pas été

txt=['bonsoir', 'Jean']
print(txt)     # affiche ['bonsoir', 'Jean']
changeEnPlace(txt, 1)
print(txt)     # affiche ['bonsoir', 'Jacques']
```

Le cas des chaînes de caractères

Les chaînes ressemblent aux tableaux parce qu'on peut lire les caractères de la même façon, avec `s[i]`, mais elles ne sont par définition pas modifiables : `s[i] = 'A'` est rejeté car une chaîne de caractères est considérée comme constante que ce soit dans une fonction ou pas. Si on a besoin de modifier le contenu d'une chaîne de caractères reçue en paramètre, il faudra créer une nouvelle chaîne de caractère(s) contenant le résultat et la renvoyer avec l'instruction `return`.

Voyons un exemple :

```
def bonjour(nom):
   s = 'bonjour ' + nom # création de la nouvelle chaîne de caractères
   return s                          # on renvoie la chaîne résultante

message = bonjour('Dupond')

print(message) # affichage : bonjour Dupond
```

5.2.3 Exemples de fonctions

Le **corps** de la fonction est la tâche que doit accomplir la fonction. Les **paramètres formels** y sont utilisés comme des **variables initialisées**. Les variables introduites dans le corps de la

fonction sont des **variables locales**, elles ne sont connues et utilisables que dans le corps de la fonction. Dès que la fonction se termine, **les paramètres formels et les variables locales sont détruits**.

Remarquez que, dans le corps d'une fonction, il est possible d'appeler d'autres fonctions ; dans ce cas le corps de la fonction jouera le rôle de programme appelant.

Pour écrire les fonctions, deux règles essentielles sont à respecter :

1. Concevoir la fonction pour qu'on puisse l'utiliser commodément dans un maximum de situations (contextes),
2. Bien expliquer et documenter ce qu'attend la fonction et ce qu'elle fait.

1. Nous souhaitons écrire une fonction qui trouve le plus grand élément d'un tableau de valeurs.
   (a) Comment appeler la fonction, quels sont ses paramètres formels, que renvoie la fonction ?
      i. le nom de la fonction doit indiquer ce que fait la fonction, choisissons le nom : `chercheMaxTab()`,
      ii. la fonction a besoin de parcourir un tableau afin de rechercher la plus grande valeur. A chaque appel de la fonction, le contenu et la taille du tableau peut changer. On en déduit qu'il y a **un paramètre formel de type tableau**. Il n'y a pas de paramètre pour la taille du tableau, parce qu'en Python, la fonction len() permet de l'obtenir.
      iii. après avoir trouvé la valeur maximale, la fonction doit communiquer cette valeur au programme appelant, donc la fonction doit se terminer par un `return` suivi du nom de la variable locale contenant le maximum.

   (b) Que doit faire la fonction ? C'est dans cette étape qu'on écrit le corps de la fonction. On pourra constater après analyse que nous utiliserons :
      – une variable locale `plusGd` qui contiendra la valeur la plus grande du tableau ; au départ elle sera initialisée avec la première valeur du tableau,
      – une autre variable locale `i` qui correspondra à l'indice courant du tableau, au départ elle sera initialisée à 1, car la case d'indice 0 a déjà été traitée (voir point précédent),
      – il faudra utiliser une itérative permettant de comparer chaque case du tableau avec la variable `plusGd` pour retenir à chaque fois le maximum.

   (c) Voici une version qui remplit le cahier des charges :

```
1  def chercheMaxTab(tab):
2      ###Renvoie le plus grand élément du tableau tab###
3      plusgd = tab[0]
4      i = 1
5      while i < len(tab):
6          if tab[i] > plusgd :
7              plusgd = tab[i]
8          i = i + 1
9      return plusgd
10
11 t = [3, 5, 18, 4, 7, -9]
12 res = chercheMaxTab(t)
13 print(res)     # affichage de 18
14
15 t = [34, -5, 3, -41, 47, 46, 9, 38]
16 res = chercheMaxTab(t)
17 print(res)     # affichage de 47
```

2. Voici un autre exemple qui permet de confirmer un choix
   Une des interactions les plus courantes avec l'utilisateur consiste à lui demander confirmation d'un choix ('répondez par oui ou par non') et à contrôler que sa réponse est bien une des réponses attendues. C'est si fréquent que cela vaut la peine d'en faire une fonction :

```
def ouiNon():
    #renvoie une réponse controlée, soit 'o' pour oui, soit 'n' pour non
    print('Repondez par oui ou par non (o/n) : ')
    rep = raw_input()
    while (rep != 'o' and rep != 'O' and rep !='n' and rep != 'N'):
        print('Repondez par oui ou par non (o/n) : ')
        rep = raw_input()
    return rep


print('Avez-vous révisé votre cours ?')
choix = ouiNon()

if choix=='o' :
        print('C\'est la clef de la reussite')
else :
        print('il n\'est jamais trop tard !')
```

5.3 Encore quelques informations utiles

5.3.1 Fonctions prédéfinies

Dans tous les langages, on peut utiliser des fonctions écrites par les autres, et qui sont disponibles dans des **bibliothèques**.

Pour chaque langage, il existe une bibliothèque standard, qui est accessible sans qu'il faille le préciser. En Python, les bibliothèques s'appellent des *modules* ; la bibliothèque standard est le *noyau.* La directive `import` est nécessaire pour utiliser les fonctions qui sont dans une bibliothèque autre que le noyau. Par exemple, pour avoir accès aux fonction de la librairie mathématique (`math`) , vous devez spécifier que cette librairie doit être utilisée, en écrivant l'instruction suivante au début de votre programme : `from math import *`.

Pour le langage C, ce sont des options lors de l'édition des liens qui permettent de les utiliser, associées à la directive `#include<descripteur de bibliothèque>` (voir cours sur le C qui sera abordé ultérieurement)).

Le programmeur peut créer ses propres bibliothèques pour permettre leur réutilisation par d'autres programmeurs.

5.3.2 Complexité des opérations

Un dernier point auquel il faut faire attention : **chaque fois** que l'on appelle une fonction, son code s'exécute, et cela peut comporter beaucoup d'opérations. Il vaut mieux en tenir compte quand on programme. Voici un exemple pour illustrer le problème.

On veut écrire une fonction qui normalise le tableau reçu en argument - c'est à dire qui divise tout par un même nombre pour que la somme du nouveau tableau soit 1 (c'est une fonction très banale). La fonction s'appelle normalise(t), voici une première façon de l'écrire.

```
def somme(t):
      s=0
      for n in t :
            s+=n
      return(s)

def normalise(t):
     i=0
     while(i<len(t)):
          t[i] = t[i] / somme(t)
```

Cette écriture est très maladroite : dans normalise(), à **chaque** boucle while on calcule somme(t), et le résultat est toujours le même. Il vaudrait mieux le calculer une fois pour toutes. Améliorer le codage est simple :

```
def somme(t):
      s=0
      for n in t :
            s+=n
      return(s)

def normalise(t):
     s=somme(t)
     i=0
     while(i<len(t)):
         t[i] = t[i] / s
```

Si vous regardez bien les opérations faites par les deux versions :
- le nombre d'opérations (additions) de somme() est la taille de t
- dans la première version, chaque boucle `while` fait un nombre d'opérations égal à 1 + le nombre d'opérations de somme()
- dans la deuxième version, chaque boucle while fait une opération
- le nombre de boucles while est le même dans les deux cas

Quand on travaille sur un tableau de 1000 nombres, ce calcul donne pour la première version 1001000 opérations, et pour la seconde 2000 opérations!!!

► Sur ce thème : **Exercices 8 à 10, TD 5**

TD6 : Fonctions

Exercice 1 : Test de parité

Question 1.1 : Écrire la fonction `estPair()` qui affiche si un nombre reçu en paramètre est pair ou non. Faire des essais en utilisant des entiers, des flottants et avec des chaînes de caractères. Prévoir les résultats.

Question 1.2 : Ré-écrire la fonction `estPair()` pour que cette dernière renvoie `True` si le nombre reçu en paramètre est pair, `False` sinon. Faire des essais en utilisant des entiers, des flottants et avec des chaînes de caractères. Prévoir les résultats.

Exercice 2 : Suite de carrés

Écrire une fonction qui permet d'afficher la suite des carrés jusqu'à n^2 où n est un entier choisi par l'utilisateur. L'affichage se fera sous la forme

$$0 - 1 - 4 - 9 - 16 - 25 - 36 - 49 - 64 - 81 - 100 \ldots$$

Dans l'exemple suivant l'entier n est égal à 6 : $0 - 1 - 4 - 9 - 16 - 25 - 36$.

Exercice 3 : Comparer deux nombres

Écrire la fonction `compare()` qui reçoit deux nombres en argument `a` et `b` et affiche un message adapté selon que `a` soit supérieur, inférieur ou égal à à `b`. Faire des essais en utilisant des entiers, des flottants.

Exercice 4 : Moyenne de deux nombres

Écrire la fonction qui reçoit deux nombres en arguments et qui calcule et renvoie leur moyenne. Quel est le problème si les deux nombres sont entiers ? quelle solution proposez-vous ?

Exercice 5 : Produit d'entiers

Écrire une fonction `produit()` qui calcule et renvoie le produit $n1*(n1+1)*...*n2$ (*1<=n1<= n2*) des entiers compris entre n_1 et et n_2 inclus.

Exercice 6 : Année bissextile

Écrire une fonction qui permet de determiner si une année est bissextile.

On rappelle qu'une année est bissextile si

- elle est divisible par 4
- mais n'est pas divible par 100
- sauf si elle est divisible par 400

Ainsi 2008 est bissextile, 1900 n'était pas bissextile et 2000 était bissextile.

Exercice 7 : Échange de variables

Écrire une fonction `swap(u,v)` qui sert à échanger les valeurs de 2 variables `u` et `v` passées en arguments. Cette fonction est-elle utile ? Á quels types s'applique-t-elle ?

Exercice 8 : Comptage des éléments d'un tableau

Écrire une fonction `nbPairImpair()` qui renvoie le nombre d'élément(s) pair(s) et le nombre d'élément(s) impair(s) dans le tableau reçu en argument.

Exercice 9 : Décalage des éléments d'un tableau à droite

Écrire une fonction `decaleCircDroite()` qui réalise le décalage circulaire vers la droite d'un

tableau d'entiers.

Voici un exemple d'affichage a obtenir :

```
Avant decalage circulaire a droite
[12, 21, 10, 11, 0, 1, 6, 8]
Apres decalage circulaire a droite
[8, 12, 21, 10, 11, 0, 1, 6]
```

Exercice 10 : Conversion entre binaire et décimal

Question 10.1 : Du binaire vers le décimal.

Écrire une fonction `bin2Dec()` qui permet de convertir une chaîne de caractères contenant la représentation binaire d'un nombre (codage entier naturel) en sa représentation décimale.

Exemple d'utilisation :

```
nBin='10000001'
nDec = bin2Dec(nBin)
print('Le nombre binaire (code entier naturel) '+ str(nBin)+' se convertit en base 10 : ' + str(n
```

Question 10.2 : Du décimal vers le binaire.

Écrire une fonction qui calcule l'écriture en base 2 d'un nombre entier positif passé en argument sous sa forme décimale.

Le résultat pour 5 sera 101.

TP6 : Fonction

Exercice 11 : Foncions et dessins

Le module Turtle permet de dessiner à l'écran des figures. Afin de pouvoir uiliser ce module, il faut ajouter, en début de programme, l'instruction : `from turtle import *`.

Attention : votre fichier ne doit pas s'appeler lui-même turtle!!!

On peut alors utiliser les instructions suivantes pour dessiner :

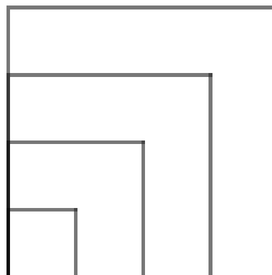
- `goto(x,y)` : aller à l'endroit de coordonnées x et y,
- `forward(distance)` : avancer d'une distance donnée,
- `up()` : relever le crayon (pour pouvoir avancer sans dessiner),
- `down()` : abaisser le crayon (pour pouvoir recommencer à dessiner),
- `left(angle)` : tourner à gauche d'un angle donné (exprimé en degré),
- `right(angle)` : tourner à droite.

À titre d'exemple, regarder ce que donne le code :

```
from turtle import *
forward(100)
left(90)
forward(50)
goto(0, 0)
```

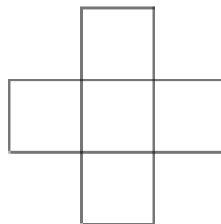
Question 11.1 : Écrire une fonction permettant à la tortue de dessiner un carré de coté n.

Question 11.2 : La tortue est positionnée au coin inférieur gauche et regarde vers le haut. Écrire une fonction qui fera dessiner à la tortue la figure suivante. Les paramètres de la fonction sont la



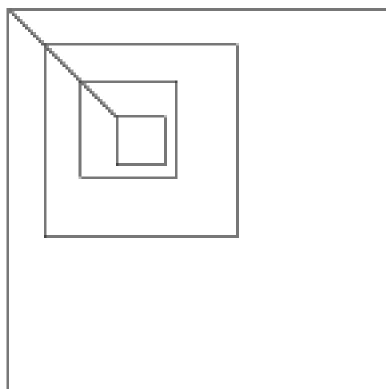
longueur du coté du plus petit carré et la différence entre les longueurs des cotés des différents carrés.

Question 11.3 : Écrire une fonction permettant de dessiner un rectangle. Utiliser cette fonction pour dessiner la figure suivante avec la tortue. Les longueurs des cotés des rectangles seront choisies



par l'utilisateur.

Question 11.4 : Carrés emboîtés.
Donner un programme qui réalise, grâce à la tortue, le dessin suivant :



sachant que
- la tortue est placée dans le coin supérieur gauche du carré interne et qu'elle regarde vers le haut.
- le nombre de carrés à tracer est demandé à l'utilisateur (4 dans l'exemple ci-dessus)
- la longueur du coté du carré le plus petit (à l'intérieur) est demandé à l'utilisateur et la longueur du coté de chaque autre carré est égale au double de la longueur du coté du carré juste en dessous. On utilisera pour dessiner les carrés la fonction définie à l'Exercice 11.
- le segment qui relie chaque carré est de longueur fixe et égale au coté du plus petit carré et forme un morceau de la diagonale de chaque carré.

Question 11.5 : Carré de carrés.

1. Écrire un fonction permettant de dessiner p carrés de coté n les uns à la suite des autres. On utilisera pour dessiner chaque petit carré la fonction définie dans l'Exercice 11
2. Écrire un programme qui permet de dessiner avec la tortue la figure suivante :

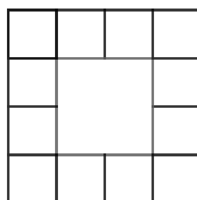


Figure 5.1 – suite de carrés

   - La longueur du coté des petits carrés est demandée à l'utilisateur.
   - Le nombre de petits carrés nécessaires pour former un coté du grand carré est demandé également à l'utilisateur (4 dans l'exemple ci-dessus).
   - La tortue est positionnée au coin inférieur gauche et regarde vers le haut.
3. Écrire un programme qui fait dessiner à la tortue un damier dont le còté est formé de p cases dont le côté est demnadé à l'utilisateur.

Exercice 12 : Modularité du code : calcul de la moyenne et de la variance sur un tableau

Question 12.1 : Écrire une fonction `moyenne( tab )` d'un argument de type tableau, contenant des valeurs numériques (`int` ou `float`). Cette fonction renvoie une valeur réelle (`float`) corres-

pondant à la moyenne des éléments du tableau. Vous pourrez vous aider du code vu lors des TD sur les tableaux.

Question 12.2 : Définissez une fonction `tabCarre(tab)` d'un argument de type tableau, contenant des valeurs numériques (`int` ou `float`). Le tableau initial ne sera pas modifié par la fonction. La fonction renverra le tableau des carrés des éléments du tableau intial. Vous pourrez initialiser un tableau vide, puis, au moyen de la concaténation remplir le tableau avec les carrés. Pour élever un nombre au carré vous utiliserez une fonction `carre()` que vous définirez. Les deux fonctions `carre()` et `tabCarre()` devront donc être définies dans le même fichier.

Question 12.3 : Définissez une fonction `variance( tab )` permettant de calculer la variance des valeurs du tableau. On rappelle que la formule de la variance pour une distribution $X = \{x_i\}$ est :

$$V = \frac{\sum_{i=1}^{N}(x_i - \bar{x})^2}{N} = \frac{\sum_{i=1}^{N}(x_i)^2}{N} - \bar{x}^2,$$

où $\bar{x}$ est la moyenne de la distribution $X = \{x_i\}$.

La variance est donc égale à la moyenne des carrés moins le carré de la moyenne. Vous utiliserez les fonctions `carre()`, `moyenne()` et `tabCarre()` définies précédemment. Les 4 fonctions devront donc être écrites dans le même fichier.

Exercice 13 : Modularité du code : formatage de chaînes de caractères

Dans cet exercice, les chaînes de caractères seront manipulées comme des tableaux de caractères.

Question 13.1 : Définissez la fonction de deux paramètres `ajouteEspaces( chaine , n )` qui renvoie la chaîne de caractères `chaine` après lui avoir ajouté `n` espace(s) après son dernier caractère :

Question 13.2 : Définissez la fonction de deux paramètres `tronquer( chaine , l )` qui renvoie le préfice de longueur l (les l premiers caractères) de la chaîne de caractères `chaine` :

Question 13.3 : Définissez la fonction de deux arguments `padRight( chaine , size )` telle que :
- L'argument `chaine` est une chaîne de caractères,
- L'argument `size` est un entier positif,
- La fonction renvoie une chaîne de caractères de longueur `size` :
  - Si la longueur de la chaîne initiale est supérieure à `size`, la chaîne initiale est tronquée.
  - Si la longueur de la chaîne initiale est inférieure à `size`, des caractères 'espace' sont rajoutés à la fin de la chaîne en nombre suffisant pour que la chaîne renvoyée soit de longueur `size`.

Question 13.4 : En utilisant la fonction `padRight(...)`, proposez une fonction d'un argument `afficheTableMult( u, max )` permettant d'afficher la table de mutliplication de la valeur `u` pour un multiplicateur variant de 0 à `max`. L'affichage devra être "propre", c'est à dire que les caractères `'*'` et `'='` de la table devront être alignés sur une même colonne. Par exemple, l'appel de `afficheTableMult( 7 , 100 )` devra prendre la forme suivante :

```
7   = 1   * 7
14  = 2   * 7
...
70  = 10  * 7
...
700 = 100 * 7
```

Exercice 14 : Fonctions de plusieurs variables et modification des valeurs d'un tableau

Afin de réaliser cet exercice nous aurons besoin de certaines fonctions définies dans la librairie `math`. Pour avoir accès à ces fonctions, vous devez spécifier que cette librairie doit être utilisée, en écrivant l'instruction suivante au début de votre programme : `from math import *`.

Cette librairie définit entre autres choses les fonctions cosinus et sinus, respectivement nommées : `cos()` et `sin()`, et la constante π, nommée `pi`. Les équations paramétriques définissant un cercle de centre (a, b) et de rayon R, sont les suivantes :

$$\begin{cases} x &= a + R * cos(\theta) \\ y &= b + R * sin(\theta) \end{cases}$$

Question 14.1 : Définissez une fonction de deux paramètres `pointCercle(angle,r)`, renvoyant les coordonnées d'un point du cercle de centre $(0, 0)$ et de rayon `r`. Les coordonnées seront renvoyées dans un tableau de 2 éléments.

La translation d'un point de coordonnées $M = (a, b)$ par un vecteur $\overrightarrow{v} = (x, y)$ suit les règles de calcul suivantes :

$$M' = \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} a \\ b \end{pmatrix} + \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a + x \\ b + y \end{pmatrix}$$

Question 14.2 : Définissez une fonction de deux paramètres `translation( point , vecteur )` permettant d'appliquer une translation au `point`. Les coordonnées du point et du vecteur seront définies comme des tableaux de 2 nombres. La fonction `translation( point , vecteur )` ne reverra aucune valeur. Les coordonnées du `point` seront modifiées directement par la fonction.

Question 14.3 : Proposez une fonction de 3 paramètres `pointCercleGeneralise( angle , r , c )` permettant de renvoyer les coordonnées d'un cercle de rayon (r) et de centre `c`. Les coordonnées du centre et du point renvoyé seront données dans des tableaux de 2 éléments.

Pour définir cette fonction, vous utiliserez les fonctions précédemment définies : `pointCercle()` et `translation()`.

Chapitre 6

Boucles imbriquées

Le but de ce chapitre est d'apprendre à analyser un problème contenant des répétitions imbriquées et à utiliser des boucles imbriquées pour traiter ces configurations.

6.1 Piqûre de rappel sur les boucles simples

Nous avons déjà vu que toute séquence d'instruction(s) peut être imbriquée dans une autre instruction structurée et nous avons déjà rencontré l'exemple d'un choix imbriqué dans un choix ou d'un choix imbriqué dans une boucle. Nous allons examiner ici des situations d'imbrications de boucles. Pour cela il existe 2 itératives(en réalité 3) :

- la boucle `while`, dite *boucle conditionnelle* ; elle est utilisée lorsque le nombre d'itération(s) **n'est pas connu à l'avance et peut être de 0**, Par exemple les instructions :

```
1 print('entrez oui ou non')
2 choix=raw_input()
3 while choix!='oui' or choix!='non' :
4        print('entrez oui ou non')
5        choix=raw_input()
6 print('Vous avez choisi ' + choix)
```

- la boucle `for`, dite *boucle inconditionnelle* ; elle est utilisée lorsque le nombre d'itération(s) est connu à l'avance. Par exemple les instructions :

```
1 x=2
2 for i in range(0,2) :
3    print(x)
```

Ecrivent deux fois la valeur de x.
Pour allez plus loin, il existe une troisième forme de boucle : La boucle `do while`, dite *boucle conditionnelle* ; elle est utilisée lorsque le nombre d'itération(s) **n'est pas connu à l'avance et est au moins de 1**. Cette boucle **n'est pas implémentée en Python** mais existe dans de nombreux langages.
Voici une version en **langage C** de cette boucle :

```
1 #include <stdio.h>
2 #include <string.h>
3 int main(void){
4 char choix[20]; /* permet de stocker une suite de caractère(s) */
5 do {
6   printf("entrez oui ou non"); /* c'est le print du C */
7   scanf("%s", choix); /* c'est le raw_input() du C */
8 } while (strcmp(choix,"oui")!=0 || strcmp(choix,"oui")!=0);
```

```
9  /* ecriture en C de while choix!='oui' or choix!='non' */
10 
11 printf("Vous avez choisi %s",choix); /* Affiche la valeur contenue da la variable choix */
12 return 0;
13 }
```

Comparez avec la forme `while` (en faisant abstraction de la synthaxe du C)... Et vous verrez l'intérêt de la boucle `do while`.
Sachez maîtriser dèjà les deux première forme de boucles, ce sera déjà très bien !

Remarque : La boucle `for` en Python est un peu particulière, car elle utilise l'opérateur `in` qui permet de parcourir un tableau (*"pour chaque élément de ce tableau..."*) et c'est la taille de ce dernier qui fixe le nombre d'itération(s) (la taille du tableau). Dans beaucoup d'autres langages "historiques" comme le *C*, l'opérateur `in` n'existe pas, et c'est l'utilisation d'un compteur "classique" qui fixe ce nombre d'itération(s). Algorithmiquement, la définition d'une boucle inconditionnelle se rapproche de la vision historique.

6.2 Les boucles imbriquées

6.2.1 Boucles imbriquées ? C'est quoi ça ?

Pourquoi imbriquer des boucles ? Pour la même raison qu'on imbrique des alternatives. Une alternative permet de traiter une situation, un cas de figure.

Par exemple : *"est-ce un homme ou une femme ?"* peut très bien se subdiviser en d'autres cas *"a-t-il plus ou moins de 18 ans ?"*.
De même, une boucle, c'est un traitement systématique, un examen d'une série d'éléments **un par un**, par exemple, *"prenons tous les employés de l'entreprise un par un"*. Eh bien, on peut imaginer que pour chaque élément ainsi considéré, *pour chaque employé par exemple*, on doive procéder à un examen systématique d'autre chose, *prenons chacune des commandes que cet employé a traité.* Voilà un exemple typique de boucles imbriquées : on devra programmer une boucle principale (celle qui prend les employés **un par un**) et à l'intérieur, une boucle secondaire (celle qui prend les commandes de cet employé **une par une**). Dans la pratique de la programmation, la maîtrise des boucles imbriquées est nécessaire, même si elle n'est pas suffisante.

En conclusion, imbriquer des boucles c'est écrire une boucle à l'intérieure d'une autre boucle. Dans la suite nous allons étudier quelques exemples de boucles imbriquées sachant qu'il existe une multitude de combinaisons.

6.2.2 Boucle inconditionnelle dans une autre boucle inconditionnelle

"for à l'intérieur d'un for"
Tout comme les tests, il est tout à fait possible d'imbriquer les boucles, c'est à dire de mettre une boucle dans une autre boucle, sur autant de niveaux que vous le souhaitez. Vous pouvez envisager 1, 2, 3, n niveaux, mais ça risque de devenir difficilement lisible. Voici un exemple à deux niveaux. Il s'agit tout simplement de calculer et d'afficher toutes les tables de multiplication de 1 à 9. Chaque table se présentera comme suit :

```
TABLE de 4
4x1=4
4x2=8
4x3=12
4x4=16
4x5=20
4x6=24
4x7=28
4x8=32
```

```
4x9=36
4x10=40
```

Ici encore, plutôt que d'essayer d'écrire directement le détail du programme, il peut être préférable de procéder par étape. Par exemple, nous pouvons dire que, globalement, notre programme doit écrire les 9 tables de multiplication de 1 à 9 et qu'il doit donc se présenter ainsi :

```
for i in range(1,10) : # i va varier de 1 à 9
      # écrire la table de i
```

Attention ! Comme `i` donne le numéro de table, il ne faut pas le modifier à l'intérieur du corps de cette boucle. Le contenu de la répétition reste à préciser et, pour l'instant, nous l'avons simplement mentionné sous la forme d'un commentaire. Pour écrire la table i, nous pouvons procéder ainsi :

```
print('TABLE de ' + str(i))
for j in range(1,11) : # j varie de 1 à 10
      // écrire la ligne j de la table i
```

D'où une ébauche plus élaborée de notre programme :

```
for i in range(1,10) : # i va varier de 1 à 9
      print('TABLE de ' + str(i))
      for j in range(1,11) : # j varie de 1 à 10
            // écrire la ligne j de la table i
```

Il ne nous reste plus qu'à préciser comment écrire une ligne d'une table, ce qui peut se formuler ainsi :

```
print(str(i)+'x'+str(j)+'='+str(i*j))
```

Nous aboutissons à l'algorithme complet :

```
for i in range(1,10) : # i va varier de 1 à 9
      print('TABLE de ' + str(i))
      for j in range(1,11) : # j varie de 1 à 10
            print(str(i)+'x'+str(j)+'='+str(i*j))
```

La démarche utilisée est dite *démarche descendante* : elle consiste à décomposer le problème posé en sous-problèmes plus faciles à résoudre, puis à décomposer à son tour chaque sous-problème... et ceci jusqu'à ce que l'on arrive à une solution entièrement formulée (nous reviendrons sur ce point à la fin de ce cours).

6.2.3 Boucle inconditonnelle dans une boucle conditionnelle

"for à l'intérieur d'un while"

```
print('donnez un entier :')
x = int(raw_input())
while x!=0 :
      for i in range(0,2) :
            print('merci pour ' + str(x))
      print('donnez un entier :')
      x = int(raw_input())
```

- Le `while` permet de répéter la séquence qui en dépend et cela autant de fois que nécessaire... Jusqu'à ce que l'utilisateur entre une valeur pour `x` égale à 0,
- le corps de la boucle permet d'afficher deux fois la valeur saisie par l'utilisation d'une boucle inconditionnelle (boucle `for`), puis de saisir la prochaine valeur pour `x`.

Si nous exécutons ces instructions avec les valeurs 3, 5 et 0, nous obtenons ceci :

```
donnez un entier : 3
merci pour 3
merci pour 3
donnez un entier : 5
merci pour 5
merci pour 5
donnez un entier : 0
```

6.2.4 Boucle conditionnelle dans une boucle inconditionnelle

"while à l'intérieur d'un for"
On souhaite écrire un programme qui calcule les moyennes de 25 élèves. Pour chaque élève, le programme lira ses notes (nombres réels) qui pourront être en nombre quelconque ; on conviendra que l'utilisateur fournira une valeur négative pour signaler qu'il n'y a plus de notes pour un élève. Commençons tout d'abord par un exemple d'exécution du programme souhaité :

```
donnez les notes de l'élève numéro 1 (-1 pour finir)
12
15
9
-1
moyenne des 3 notes : 12.00
donnez les notes de l'élève numéro 2 (-1 pour finir)
.....
donnez les notes de l'élève numéro 25 (-1 pour finir)
10
-1
moyenne des 1 notes : 10.00
```

Compte tenu de la complexité du programme, nous pouvons chercher, dans un premier temps, à écrire les seules instructions de calcul de la moyenne d'un élève, en supposant que son *numéro* figure dans une variable nommée i :

```
print('donnez les notes de l'élève numéro ' +str(i) + '(-1 pour finir)')
somme = 0
nb = 0
note = float(raw_input())
while note >=0 :
       somme = somme + note
       nb=nb+1
       note = float(raw_input())

if nb > 0 :
       print('moyenne des ' + str(nb)+' notes : ' + str(somme/nb)
```

Pour obtenir le programme désiré, il nous suffit maintenant de répéter les instructions précédentes, en utilisant la variable i comme compteur, variant de 1 à 25.
Voici le programme complet :

```
for i in range(1,25): # i varie de 1 à 25
       print('donnez les notes de l'élève numéro ' +str(i) + '(-1 pour finir)')
       somme = 0
       nb = 0
       note = float(raw_input())
       while note >=0 :
```

```
7               somme = somme + note
8               nb=nb+1
9               note = float(raw_input())
10          if nb > 0 :
11              print('moyenne des ' + str(nb)+' notes : ' + str(somme/nb)
```

Remarque : Faites bien attention à la place des deux initialisations `somme=0.0` et `nb=0`. Elles doivent figurer dans la boucle inconditionnelle gouvernée par le compteur i et avant la boucle conditionnelle de prise en compte des différentes notes.

6.2.5 Boucle conditionnelle dans une boucle conditionnelle

"while à l'intérieur d'un while"
Considérons une fonction `verifiePresenceRoi()` qui reçoit deux paramètres entiers et renvoie `True` si la fonction a trouvé un roi sur l'échiquier et `False` sinon.

Voici le cartouche de la fonction `verifiePresenceRoi()` :

```
1 #########################################################################
2 # verifiePresenceRoi(i,j)
3 #i : désigne l'indice de la ligne de l'échiquier (indice commence à 0)
4 #j : désigne  l'indice de la colonne de l'échiquier (indice commence à 0)
5 #valeur retournée : True si un roi a été trouvé, False sinon
6 #########################################################################
```

Remarque : Utilisez des cartouches pour documenter vos fonctions.

Le cahier des charges : Nous souhaitons trouver l'indice de la ligne et l'indice de la colonne du premier roi trouvé.

1. il est clair qu'il **faut parcourir chaque ligne de l'échiquier... et de s'arrêter dès qu'on trouve un roi !** : *"tant que je n'ai pas parcouru toutes les lignes ET que je n'ai pas trouvé de roi ! "*,
2. pour chaque ligne, il faut **parcourir les cases... et s'arrêter dès qu'on trouve un roi !** : *"tant que je n'ai pas parcouru tous les éléments de la ligne courante ET que je n'ai pas trouvé de roi !"*,

On abouti à quelque chose comme suit :

```
1  i=0     #parcours des lignes
2  j=0     # parcours des colonnes
3
4  trouve = False
5   # tant que je n'ai pas parcouru toutes les  les lignes ET que je n'ai pas trouvé de roi !
6  while i<10 and not(trouve) :
7          #tant que je n'ai pas parcouru tous les éléments de la ligne courante
8          # ET que je n'ai pas trouvé de roi !
9          while j<10 and not(trouve) :
10                 #je cherche
11                 #et j'avance d'une colonne
12         # et j'avance d'une ligne
```

– le commentaire *je cherche* doit être substitué par une instruction qui consulte l'espace à deux dimensions et qui indique si l'information cherchée est trouvée ou pas pour la ligne d'indice `i` et la colonne d'indice `j`.

```
1 trouve = verifiePresenceRoi(i,j)
```

– pour avancer d'une colonne, il suffit d'incrémenter `j`,

```
j = j +1
```

– pour avancer d'une ligne, il suffit d'incrémenter `i`,

```
i = i +1
```

Nous obtenons finalement l'algorithme suivant :

```
i=0     #parcours des lignes
j=0     # parcours des colonnes

trouve = False
# tant que je n'ai pas parcouru toutes les  les lignes ET que je n'ai pas trouvé !
while i<10 and not(trouve) :
        #tant que je n'ai pas parcouru tous les éléments de la ligne courante
        # ET que je n'ai pas trouvé de roi !
        while j<10 and not(trouve) :
                trouve = verifiePresenceRoi(i,j)
                j = j +1
        i = i +1
```

Dès que que la variable `trouve` devient `True`, on sort "naturellement" des deux boucles imbriquées car la condition booléenne `trouve==False` devient `False`.

6.2.6 Un peu plus compliqué... 3 boucles imbriquées... le vertige !

Il existe une multitude d'autrse exemples de combinaisons d'imbrications de boucles (avec les boucles `for` et `while`). Il existe aussi une "infinité" de niveaux d'imbrications. Prenons un dernier exemple pour illustrer notre propos. Le but de cet anodin algorithme est de trouver pour quelles valeurs de A, B et C, $A*100+B*10+C = A^3+B^3+C^3$ (ABC représentant un nombre décimal). La recherche sera limitée pour chaque valeur entière comprise entre 1 et 10 (bien entendu, vous pouvez augmenter l'intervalle). L'algorithme nécessite trois boucles pour chacune des valeurs. C'est bien entendu au sein de la dernière boucle que les valeurs sont calculées et les résultats affichés en cas d'égalité.

```
for  a in range(1,11):
        for  b in range(1,11):
                for  c in range(1,11):
                        nb1= a*100+b*10+c
                        nb2 = a*a*a+b*b*b+c*c*c
                        if  nb1==nb2:
                                print ('a='+str(a) + 'b='+str(b) + 'c='+str(c))
```

6.2.7 Revenons sur l'exemple des tables de multiplication

Nous vous rappelons l'algortihme obtenu :

```
for  i in range(1,10) : # i va varier de 1 à 9
        print('TABLE de ' + str(i))
        for j in range(1,11) : # j varie de 1 à 10
                print(str(i)+'x'+str(j)+'='+str(i*j))
```

Si nous nous concentrons sur les fonctionnalités des deux deux boucles imbriquées on pourrait écrire : *"je parcours les tables de 1 à 9 (compteur i) et pour chaque table i, je l'affiche"*.

On pourrait définir une **fonction** qui permettrait d'afficher n'importe qu'elle table k sans préjuger de qui l'utilisera :

```
1 # k est un paramètre formel qui désigne la table à afficher
2 # affiche la table de multiplication de k
3 def afficheTable(k) :
4         for i in range(1,11) : # i varie de 1 à 10
5                 print(str(k)+'x'+str(i)+'='+str(k*i))
```

Finalement, l'algoritme initial devient :

```
1 for i in range(1,10) : # i va varier de 1 à 9
2         afficheTable(i)
```

Remarque :

- Bienvenue dans le monde de la décomposition fonctionnelle (démarche descendante) qui consiste à découper un problème en fonction(s) ; la définition de la fonction `afficheTable()` permet de simplifier l'algorithme initial mais surtout permet la réutilisation de cette fonction dans d'autres contextes,
- vous remarquez que la fonction `afficheTable()` masque l'imbrication des boucles ayant pour résultat une fois de plus de simplifier un algorithme,
- souvenez-vous, lors de la séance 1, nous avions affirmé qu'un problème peut être résumé par une séquence ordonnançant des actions plus ou moins complexe à entreprendre (l'ordre d'appel de fonctions), c'est l'essence même de la décomposition fonctionnelle d'un cahier des charges.

Remarque : Une règle de "bonne" programmation consiste à ne jamais écrire un traitement de plus d'une page ; dans ce cas, le décomposer en appels de fonctions dont on sait ce que chacune fait.

TD7 : Boucles imbriquées

Exercice 1 : Puissance d'un circuit

Écrire un algorithme qui permet d'afficher la puissance d'un circuit en fonction de la tension u variant par pas de 0.5V et de l'intensité i i variant par pas de 0.05A sachant que les valeurs initiales et finales de u et i seront demandées à l'utilisateur.

Exemple d'affichage : (pour u variant de 0 à 2V et i variant de 0 à 0.15A)

```
u=0 i=0 p=0
u=0 i=0.05 p=0
u=0 i=0.10 p=0
u=0 i=0.15 p=0

u=0.5 i=0    p=0
u=0.5 i=0.05 p=0.025
u=0.5 i=0.10 p=0.050
u=0.5 i=0.15 p=0.075
********************
```

Exercice 2 : Dessins d'étoiles

1. Écrire une fonction qui affiche un triangle rectangle dont la hauteur et la base dépendent d'un entier `n` (passé en paramètre).

   Exemples :

   pour `n`=1

   ```
   *
   ```

   pour `n`=2

   ```
   *
   **
   ```

   pour `n`=3

   ```
   *
   **
   ***
   ```

   pour `n`=4

   ```
   *
   **
   ***
   ****
   ```

2. Ecrire une fonction qui prendra comme paramètre la hauteur et qui affiche la figure suivante. Les symboles utilisés sont des étoiles (*) et des traits(-).

   Exemple. La hauteur est 5

   ```
   *----
   **---
   ***--
   ****-
   ******
   ```

3. Écrire une fonction qui affiche un triangle rectangle dont la hauteur et la base dépendent d'un entier `n` (passé en paramètre).

   Exemples :

   pour `n`=1

   ```
   *
   ```

   ```
    *
   **
   ```

   pour `n`=2

   pour `n`=3

   ```
     *
   ```

```
 **                              *          ***
***          pour n=4           **          ****
```

4. Écrire une fonction qui affiche un triangle isocèle dont la hauteur dépend d'un entier `n` (passé en paramètre).

   Exemples :

```
pour n=1        pour n=2        pour n=3        pour n=4

*                *                *                *
                ***              ***              ***
                                *****            *****
                                                *******
```

5. Écrire une fonction qui affiche un losange dont la longueur des diagonales (nombre impair) dépend d'un entier `n` (passé en paramètre). Dans le cas où `n` est pair, la fonction affichera un message d'erreur.

   Exemples :

```
pour n=1        pour n=3        pour n=5        pour n=7

*                *                *
                ***              ***                 *
                 *              *****               ***
                                 ***               *****
                                  *               *******
                                                   *****
                                                    ***
                                                     *
```

Exercice 3 : Calcul de moyennes

Ecrire un algorithme qui permet de calculer la moyenne de chaque étudiant d'une promotion. On suppose que le nombre d'étudiants n'est pas connu à l'avance, c'est l'utilisateur qui décide de mettre fin à l'exécution du programme lorsqu'il n'y a plus d'étudiant dont il souhaite calculer la moyenne. Le nombre de notes de chaque étudiant est fixé 5.

Exercice 4 : Calculer des puissances entières avec des additions

Écrire un algorithme permettant de calculer x^y où x et y sont des entiers en utilisant seulement l'addition.

Exercice 5 : Comptage de diviseurs

Définissez une fonction `nombreDeDiviseurs( max )` qui renvoie le nombre d'entiers compris entre 0 et et la valeur passée en paramètre `max` qui ont exactement 7 diviseurs.
On rappelle que `n` est un diviseur de `x`, si le reste de la division entière de `x` par `n` est nulle : `x%n=0`.
Par exemple, le nombre d'entiers inférieurs ou égaux à 30 ayant exactement 7 diviseurs est égal à 2. Ce sont les entiers 24 et 30 :

- les 7 diviseurs de 24 = { 1, 2, 3, 4, 6, 8, 12 }
- les 7 diviseurs de 30 = { 1, 2, 3, 5, 6, 10, 15 }

Pour réaliser cette fonction, vous suivrez les spécifications suivantes :

- la boucle principale parcourra les entiers à tester (entre 0 et `max`)
- une boucle imbriquée testera le nombre de diviseurs de l'entier testé.

Exercice 6 : Compression de données - RLE

L'algorithme de compression RLE (Run Length Encoding) est un algorithme utilisé dans de nombreux formats de fichiers (BMP, TIFF,...). Il est basé sur l'identification de la répétition consécutive de mêmes éléments.
Le principe consiste à parcourir la séquence de données à compresser et de donner séquenciellement le nombre de répétitions consécutives d'un élément donné suivie de la description de cet élément.
La séquence de données `AAAAAACCCCAABAAAAAADDDD` soumise à l'algorithme RLE identifie :

- une séquence de 6 caractères `A` → `6A`,
- une séquence de 4 caractères `C` → `4C`,
- une séquence de 2 caractères `A` → `2A`,
- une séquence de 1 caractère `B` → `1B`,
- une séquence de 6 caractères `A` → `6A`,
- une séquence de 4 caractères `D` → `4D`

et donne la séquence compressée suivante : `6A4C2A1B6A4D`.
Définissez une fonction `RLEcompress(tab)` admettant pour argument un tableau de caractères, et retournant la version compressée de ce tableau.
Proposez ensuite la fonction de décompression `RLEuncompress()`.

Chapitre 7

Tableaux à deux dimensions

Dans le chapitre sur les tableaux à une dimension, nous avons vus comment traiter des tableaux à une dimension (vecteur, liste). Pour la définition de matrices ou de tableaux à 2 dimension nous avons besoin de tableaux comme celui-ci :

4	22	55	32
5	23	34	4
2	2.4	2	12
23	44	55	77

Ce tableau contient 4 lignes et 4 colonnes, donc c'est un tableau à 2 dimensions. Pour accéder à un élément il faut spécifier un couple d'indices (deux dimensions, donc deux indices) :

– Le premier indice indique le numéro de la ligne

0,0	0,1	0,2	0,3
4	22	55	32
5	23	34	4
2	2.4	2	12
23	44	55	77

– Le deuxième indice indique la colonne

0,1	4	22	55	32
1,1	5	23	34	4
2,1	2	2.4	2	12
3,1	23	44	55	77

Il est important de retenir que pour un tableau de dimension $n \times m$, les indices des lignes et des colonnes varient respectivement de 0 à $n-1$ et de 0 à $m-1$.

Comme les tableaux à une dimension, un tableau multidimensionnel est défini par les symboles `[` et `]`. Entre ces crochets figurent les indices qui permettent de parcourir le tableau. Chaque ligne du tableau peut être vue comme un tableau à une dimension. En reprenant les notations du cours sur les tableaux à une dimension, nous pouvons écrire chaque ligne du tableau comme une liste Python, par exemple :

4	22	55	32

5	23	34	4

2	2.4	2	12

23	44	55	77

Puis, il faut créer une nouvelle liste Python contenant les lignes qui ont été définies

```
Ligne0 = [4, 22, 55, 32]
Ligne1 = [5, 23, 34, 4 ]
Ligne2=[2, 2.4, 2, 12]
Ligne3=[23, 44, 55, 77]
Tableau = [ Ligne0, Ligne1, Ligne2, Ligne3 ]
```

Evidement, on peut faire la déclaration du tableau en une seule fois :

```
Tableau = [[4, 22, 55, 32], [5, 23, 34, 4 ], [2, 2.4, 2, 12], [23, 44, 55, 77]]
```

Ainsi, en Python, un tableau à plusieurs dimensions peut être implanté comme une liste de listes. Un tableau à deux dimensions est une liste de lignes. On peut accéder à chaque ligne séparément :

```
print(Tableau[0])
print(Tableau[2])
```

on obtient l'affichage

```
[4, 22, 55, 32]
[2, 2.4, 2, 12]
```

Chaque ligne est une liste de cellules individuelles. Cela nous permet d'utiliser la notation Tableau [i] [j] qui signifie que nous choisissons la ligne **i**, et seulement la colonne **j** de cette ligne.

```
print(Tableau[0][0])
print(Tableau[1][2])
```

on obtient l'affichage

```
4
34
```

Il est possible de manipuler des tableaux contenant des chaînes de caractères ou même des valeurs de différents types (`string, integer, float`).

```

s = ['david', 'Mohamed', '32']
s2 = ['Williams', 'Rudy', 'Omar', '18']
s3 = [s, s2]
print s3[1][2]
```

on obtient l'affichage suivant

```
Omar
```

7.1 Parcours et affichage d'un tableau

Parcourir un tableau à deux dimensions peut être fait naturellement avec deux boucles imbriquées : une sur les lignes et une sur les colonnes. Il est possible d'utiliser la commande `range()` pour parcourir successivement les indices de chaque cellule du tableau. L'exemple suivant permet de parcourir les éléments du tableau en utilisant deux boucles `while` :

```
i = 0
while i < len(Tableau):
    j = 0
    while j < len(Tableau[i]):
        print(Tableau[i][j])
        j += 1
    i += 1
```

L'exemple suivant permet de parcourir les éléments du tableau en les mettant à zéro :

```
for r in range(len(Tableau)):
    for c in range(len(Tableau[r])):
        Tableau[r][c]= 0

print(Tableau)
```

On obtient donc l'affichage :

```
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

L'affichage du tableau sous forme d'une liste de listes est un peu difficile à lire. La boucle suivante permet d'afficher la table dans une forme plus lisible.

```
for ligne in Tableau:
    print (ligne)
```

On obtient l'affichage suivant :

```
[4, 22, 55, 32]
[5, 23, 34, 4]
[2, 2.4, 2, 12]
[23, 44, 55, 77]
```

7.2 Initialisation automatique d'un tableau

En reprenant le programme précédent et en manipulant une nouvelle variable `matrice` :

```
for r in range(3):
    for c in range(3):
        matrice[r][c]= r+c

print (matrice)
```

Nous obtenons ce message d'erreur

```
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
NameError: name 'matrice' is not defined
```

Comme pour un tableau à une dimension, l'interpréteur ne peut pas écrire dans une case qui n'a pas été créée à l'initialisation de `matrice`. Nous avons plusieurs possibilités pour écrire cette initialisation :

– définir explicitement la matrice comme une liste de listes d'élément.

```
matrice = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

– En utilisant l'opérateur de répétition `*` et la boucle `for`, nous pouvons construire une liste de listes.

```
A = [0]*3 # crée les A[i] initialisés à 0
for i in range(3):
     A[i] = [0] * 3 # remplace chaque A[i] par un tableau
```

– **En Python**, nous pouvons construire un tableau en utilisant la fonction `range()` et la boucle `for` à l'intérieur des crochets (on appelle cela une liste définie *en compréhension*). L'exemple suivant crée une table comme une liste de listes et en même temps remplit chaque cellule de la table.

```
matrice= [ [ 0 for i in range(3) ] for j in range(3) ]
print (matrice)
```

Cette instruction doit être lue à partir de l'intérieur vers l'extérieur, comme une expression ordinaire. La liste intérieure, `[0 for i in range (3)]`, crée une simple liste de trois zéros. La liste extérieure, `[[...] for j in range (3)]` crée trois exemplaires de ces listes internes. Dans des cas plus compliqués, au lieu du 0, on peut utiliser une expression qui dépend de i et j ...
Il y a bien sûr d'autres écritures possibles en combinant autrement les 3 méthodes ci-dessus.

7.3 Remarques sur les tableaux

Comment détecter un tableau à 2 dimensions dans un énoncé ?

– L'énoncé fait directement référence à une donnée de type grille (l'emploi du temps de la semaine, tableau des ventes des articles par mois, ...)
– Les données ont deux paramètres : les températures moyennes relevées dans toutes les capitales d'Europe par mois de l'année. Une température est une propriété d'une ville d'Europe pour un mois de l'année, ceci implique une relation à deux dimensions donc un tableau à deux dimensions.

De façon plus générale, un tableau à N dimensions est décrit par N paramètres. Le nombre total d'éléments d'un tableau à N dimensions est le produit des tailles de chaque dimension.

TD8 : Tableaux à deux dimensions

Exercice 1 : Manipulations de tableaux

Soit un tableau à deux dimensions de 5 par 4 entiers complètement initialisé :

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

Question 1.1 : Effectuer les actions suivantes :
- afficher le tableau
- afficher le sous-tableau de la 2e ligne à la 3e ligne comprises
- afficher le sous-tableau du début à la 2e ligne comprise
- afficher le sous-tableau de la 3e ligne comprise à la fin du tableau
- afficher le tableau constitué uniquement des lignes d'indice pair
- afficher le tableau constitué uniquement des éléments d'indice impair de chaque ligne (ceci correspond à afficher uniquement les colonnes d'indice impair de la matrice)

Question 1.2 : Écrire un algorithme qui permet premièrement d'afficher la diagonale (de gauche à droite), puis la 2e diagonale

Question 1.3 : Tester si les nombres qui apparaissent dans les deux diagonales sont tous égaux à une seule et même valeur. Le programme affichera "OUI" ou "Non" en fonction du résulat.

Question 1.4 : Soit un tableau à deux dimensions initialisé avec des nombres entiers.

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

Donner un algorithme qui permet de calculer la moyenne par colonne. Les valeurs des moyennes seront sauvegardées dans une liste. Ensuite, indiquer l'indice de la colonne correspondant à la plus grande moyenne.

Exercice 2 : Échange de triangles

Écrire l'algorithme qui échange le triangle inférieur avec le triangle supérieur dans un tableau à deux dimensions. C'est donc le tableau obtenu en faisant une symétrie par rapport à la diagonale principale. Donnez le programme en utilisant la boucle `while` puis la boucle `for`.

Exemple :

```
10 11 45 78                   10 23 56 47
23 44 12 56    ---\           11 44 90 78
56 90 67 89    ---/           45 12 67 55
47 78 55 34                   78 56 89 34
```

Exercice 3 : Base de données étudiante

Dans la base de données des étudiants du département informatique, nous disposons d'une table "étudiant" contenant le numéro, le nom et le prénom des étudiants.

Question 3.1 : Écrire un programme qui permet de saisir toutes ces informations au clavier et de les stocker dans un tableau.

Question 3.2 : Écrire le programme qui permet d'afficher tous les étudiants dont le nom commence par une lettre saisie au clavier (en tenant compte de la casse).

TP8 : Tableaux à deux dimensions

Exercice 4 : Liste de listes

En Python, un tableau à 2 dimensions est défini comme une liste de listes. Écrire le programme qui permet de vérifier que les éléments d'une liste sont des listes de même longueur. Par exemple les éléments de la liste `A = [[1, 2], [2, 4, 3]]` n'ont pas la même longueur (`len([1, 2])!=len([2, 4, 3])`).

Exercice 5 : Manipulation des matrices

Question 5.1 : Écrire un programme qui permet de construire la matrice nulle de taille $m \times n$.

Question 5.2 : Modifier le programme pour construire la matrice carrée identité I (tous les éléments de la diagonale sont égaux à 1, les autres valent 0).

Question 5.3 : Écrire une fonction qui calcule la trace d'une matrice carrée, c'est-à-dire la somme des coefficients diagonaux de cette matrice.

Question 5.4 : Écrire une fonction qui teste si un élément x appartient à une matrice A.

Question 5.5 : Écrire une fonction qui permute deux lignes d'une matrice.

Question 5.6 : Écrire une fonction qui remplace dans une matrice A toutes les occurrences de x par y.

Exercice 6 : Addition de matrices

Ecrire aussi une fonction qui permet de faire l'addition de deux matrices de même dimension.

Exercice 7 : Carré magique

Un carré (tableau d'entiers de N lignes et N colonnes initialisées) est dit magique lorsque la somme d'une ligne, d'une colonne ou d'une diagonale quelconque est toujours égale au même nombre. Voici un exemple de carré magique :

15	8	1	24	17
16	14	7	5	23
22	20	13	6	4
3	21	19	12	10
9	2	25	18	11

Question 7.1 : Concevoir une fonction créant une matrice carrée dont les coefficients sont des entiers positif saisi par l'utilisateur.

Question 7.2 : Concevoir une fonction qui vérifie si un carré est magique.

Chapitre 8

Les chaînes de caractères

8.1 Introduction

À quoi servent les chaînes de caractères ?

On appelle chaîne de caractères des fragments de texte utilisés par un programme. Il n'est pas nécessaire que le texte ait un sens pour un humain : il suffit qu'il soit composé d'une suite de caractères (un caractère n'est pas obligatoirement une lettre de l'alphabet : 'A', '9', ' ', '.' sont des caractères).

La chaîne de caractères est un type de base pour beaucoup de langages de programmation et il existe de nombreuses bibliothèques de fonctions pour les manipuler.
Comprendre et savoir traiter les chaînes de caractères est indispensable pour un informaticien tant elles sont omniprésentes dans tous les domaines de l'informatique :

- pour le Web : les pages XHTML, les scripts PHP par exemple,
- les bases de données : SGBDR,
- les réseaux : dans la constitution des trames ethernets,
- les systèmes d'exploitation : écriture de scripts sous linux,
- programmation : sources de programmes, fichiers textes etc.

Le but de cette séance est de comprendre et concevoir des algorithmes de traitements de chaînes. Ces algorithmes seront de plus en plus élaborés ; cela vous permettra d'entrevoir la puissance des chaînes de caractères au travers des algorithmes qui les traitent.

Rappels...

Une chaîne de caractères est une suite ordonnée de caractères. Cette suite est délimitée par deux apostrophes `'`. Une chaîne de caractères n'est pas modifiable.

Exemples de chaînes littérales :

- *'bonjour'* est une chaînes de caractères contenant 7 caractères.
- '' est la chaîne de caractères vide contenant 0 caractère.

Initialisation d'une variable à partir d'une chaîne de caractères littérale. Nous pouvons initialiser une variable avec une chaîne de caractères littérale, cette variable sera de type chaîne de caractères.

Exemple :

```
1 message='L\'hiver approche, vite vite, il faut se couvrir des morsures du froid.'
2 print(message)
3 #affiche : L'hiver approche, vite vite, il faut se couvrir des morsures du froid.
```

Noter le caractère \ avant l'apostrophe de *L'hiver* qui permet d'*échapper* le caractère ' pour qu'il ne soit pas interprété comme un délimiteur de chaînes de caractères. Pour éviter l'utilisation de caractères d'échappement vous pouvez utiliser les guillemets " pour délimiter la chaîne de caractères.

Exemple :

```
message="L'hiver approche, vite vite, il faut se couvrir des morsures du froid."
print(message)
#affiche : L'hiver approche, vite vite, il faut se couvrir des morsures du froid.
```

Initialisation d'une variable à partir d'une autre variable. Nous pouvons affecter une variable à partir d'une autre variable de type chaîne de caractères.

```
message='L\'hiver approche, vite vite, il faut se couvrir des morsures du froid.'
messagBis=message
print(messageBis)
#affiche : L'hiver approche, vite vite, il faut se couvrir des morsures du froid.
```

Concaténation de chaînes de caractères. L'opérateur + permet de concaténer des chaînes de caractères.

```
message='L\'hiver approche, vite vite, il faut se couvrir des morsures du froid.'
messageBis=' Couvrons nous!'
messageTierce=' Oui, couvrons nos oreilles!'
superMessage = message + messageBis + '\n'+ messageTierce
print(superMessage)

#affiche :
#L'hiver approche, vite vite, il faut se couvrir des morsures du froid. Couvrons nous!
# Oui, couvrons nos oreilles!
```

Les chaînes de caractères vues comme des tableaux. Les chaînes de caractères sont analogues aux tableaux de caractères, à ceci près qu'elles ne sont plus modifiables après leur initialisation : seule la lecture des caractères de la chaîne de caractères est autorisée.

Note : Même si les chaînes de caractères peuvent être soumises à une partie des opérations prévues pour les tableaux, elle ne sont pas de *"type"* tableau, c'est-à-dire que 'abc' est différent de ['a', 'b', 'c'] : le second est modifiable, pas le premier.

```
message='L\'hiver approche, vite vite, il faut se couvrir des morsures du froid.'
print(message[4])     # affichage : v
print(message[3] + message[6]) # affichage : ir
message[1] = 'c'
# resultat :
#  File "<pyshell#6>", line 1, in <module>
#    message[1] = 'c'
# TypeError: 'str' object does not support item assignment
```

Il faut alors bien distinguer les opérations qui *recopient* le tableau de celles qui le *modifient*. Par exemple

```
mot='abc'
mot=mot + 'd' # correct parce que + crée une nouvelle chaîne
list.insert(mot, 0, 'e') # erreur parce que list.insert() modifie son premier argument
```

▶ Sur ce thème : **Exercices 1, 2 et 3, TD8**

8.2 Algorithmes avancés de traitement de chaînes de caractères

Les chaînes de caractères comme un ensemble de sous-chaînes

Beaucoup de traitements de chaînes de caractères consistent à découper des chaînes de caractères en informations élémentaires (*tokens*). En effet, une ligne (se terminant par un retour chariot) peut être considérée comme un tout (*conteneur*) pour le transfert de l'information (dans un réseau notamment sous forme de trames). À la réception de la chaîne, il est souvent nécessaire de désagréger les tokens, c'est-à-dire de les séparer afin de les traiter individuellement.

Exemple : *'Enseignant%informatique#Bouchaib ;Khafif 0149403124'*

Dans la chaîne de caractères précédente, on peut identifier 5 informations élémentaires séparées par des caractères appelés *séparateurs* : `fonction%domaine#prénom;nom numéroProfessionnel`

Terminologie : La chaîne *'#% ; '* (noter un espace après le ;) est appelée chaîne des séparateurs, elle contient les caractères qui jouent le rôle de séparateurs ; ils ne sont donc pas autorisées dans les tokens. De plus, leurs ordres d'apparitions dans la chaîne des séparateurs n'est pas important.

▶ Sur ce thème : **EXERCICES 4, 5 ET 6, TD 8**

TD9 : Chaînes

Pour tous les exercices qui suivent, vous prévoirez des jeux d'essais probants qui permettront de tester les fonctions écrites.

Exercice 1 : Recherche d'un caractère à l'intérieur d'une chaîne de caractères

Un des traitements les plus courants est la recherche d'un (unique) caractère à l'intérieur d'une chaîne de caractères.

Écrire une fonction `indexOfCar()` qui prend comme paramètres une chaîne de caractères `chaîne` et un (unique) caractère `car` et qui retourne l'indice de la première occurrence de ce caractère dans la chaîne. Si `car` n'apparît pas dans `chaîne` la fonction retournera `-1`.

Par exemple, les instructions suivantes

```
chaine="il fait beau ici ; n'est ce pas ? oui c'est vrai"
print(indexOfCar(chaine, 'e'))
print(indexOfCar(chaine, 'x'))
```

conduiront à l'affichage suivant :

```
9
-1
```

Exercice 2 : Recherche d'une sous-chaîne à l'intérieur d'une chaîne de caractères

On peut généraliser le problème précédent en recherchant, non pas un unique caractère, mais une sous-chaîne dans une chaîne. Par exemple, dans la chaîne *"il fait beau aujourd'hui"*, la sous-chaîne *'eau'* apparatît à la position *9*, mais on ne trouve pas la sous-chaîne *'uae'*.

Écrire une fonction `indexOfStr()` qui recherche la présence d'une sous-chaîne `sStr` dans la chaîne de caractères `chaine`, toutes deux passées en paramètres. La fonction retournera la postion de la première occurrence de `sStr` dans la chaîne de caractères `chaine` ; si celle-ci n'existe pas, la fonction retournera `-1`.

Par exemple, les instructions suivantes

```
chaine="il fait beau ici, n'est ce pas ? oui c'est vrai"
print(indexOfStr(chaine, 'bea'))
print(indexOfStr(chaine, 'aeb'))
```

conduiront à l'affichage suivant :

```
8
-1
```

Exercice 3 : Comptage du nombre de voyelle(s) dans une chaîne de caractères

On rappelle que les voyelles sont les caractères *'a', 'e', 'i', o', 'u','y'*.

On souhaite concevoir une fonction qui retourne le nombre de voyelle(s) (indifféremment majuscule(s) ou minuscule(s)) dans une chaîne de caractères. Pour cela,

1. identifier les paramètres et les valeurs retournées,
2. écrire la fonction,
3. proposer un jeu de tests.

Exercice 4 : Repérage des tokens dans une chaîne de caractères

On souhaite concevoir une fonction `indexOfSep(chaine, sep)` qui reçoit deux paramètres en entrées :

- `chaine`, une chaîne de caractères à traiter, c'est-à-dire dans laquelle il faut repérer la position d'un séparateur apparaissant dans la chaîne de caractères des séparateurs `sep` (voir item suivant),
- `sep`, une seconde chaîne de caractères contenant l'ensemble des séparateurs utilisés, par exemple la chaîne `'*; :'`.

Le but de cette fonction est de retourner l'indice du premier séparateur trouvé dans la chaîne. La valeur `-1` est retournée dans le cas où aucun séparateur n'est présent dans la chaîne.

Par exemple, si l'on prend comme chaîne *"Le printemps ;est pour bientôt mais :il faut attendre l'hiver"* et comme ensemble de séparateurs *'* :;'*. L'indice retourné par la fonction sera `12`, il correspond au séparateur *';'* qui a été trouvé (en premier) dans la chaîne `chaine`. L'ordre des séparateurs dans la chaîne de caractères `sep` n'a strictement aucune importance.

Écrire la fonction `indexOfSep()`.

Exercice 5 : Découpage d'une chaîne de caractères en tokens

On souhaite concevoir une fonction `stringTokenizer()` qui permet de découper une chaîne de caractères en une suite de tokens ; pour cela elle s'appuiera sur un ensemble de séparateurs. Cette fonction recevra deux paramètres :

- Une chaîne de caractères à découper,
- une chaîne de caractères contenant les séparateurs,

La fonction renverra un tableau de chaînes de caractères contenant les tokens.

Question 5.1 : *Analyse préliminaire*

1. Soit la chaîne `"il fait*beau ici;n'est ce pas:oui c'est vrai"` et la liste de séparateurs : `'*;:'`. Donnez la liste des tokens.
2. En considérant la même chaîne de caractères et la liste de séparateurs : `' ' (espace)`, donnez la liste des tokens.

Question 5.2 : *Fonction utile*

Écrire une fonction `sousCh()` prenant en paramètre une chaîne de caractères `ch` et deux entiers `i` et `j`, et retournant la chaîne composée de tous les caractères de `ch` compris entre l'indice `i` (inclus) et l'indice `j` (exclu). Ainsi, l'exécution de la fonction `sousCh('abcdefg',2,5)` renvoie la chaîne `'cde'`.

Question 5.3 : *Fonction `stringTokenizer()`*

1. Écrire la fonction,
2. proposez d'autres jeux de tests.

Exercice 6 : Reconstitution d'une chaîne de caractères à partir de tokens et d'un séparateur

L'objectif de cet exercice est d'écrire un fonction `implodeStrToken()` faisant le traitement inverse de la fonction précédente. Cette fonction doit, à partir d'un tableau de tokens (tableau de chaînes de caractères) et d'un séparateur, créer et retourner une chaîne de caractères contenant la concaténation des tokens séparés par le séparateur.

Par exemple, si nous transmettons à la fonction `implodeStrToken()` :

- un tableau `tab` contenant les tokens suivants : `['il', 'fait', 'beau', 'ici', "n'est", 'ce', 'pas', 'oui', "c'est", 'vrai']`,
- le séparateur `';'`,

alors la fonction `implodeStrToken()` **retournera** la chaîne :

`'il;fait;beau;ici;n'est;ce;pas;oui;c'est;vrai'`

Écrire la fonction `implodeStrToken()` et proposer d'autres jeux de tests.

TP9 : Chaines

Exercice 7 : Insertions de symboles

Écrire une fonction qui reçoit une chaîne en argument et qui renvoie une chaîne dans laquelle ont été insérées des tirets bas ('_') entre chaque lettre de la chaîne d'origine. Par exemple, si on lui passe la chaîne *'bonjour'*, la fonction retourne la chaîne *'b_ o_ n_ j_ o_ u_ r'*.

Exercice 8 : Présentation rapide du langage html

Les pages Web sont écrites dans un langage appelé HTML (HyperText Markup Language/ langage de marquage hypertexte). C'est un langage de description de documents qui permet de d'écrire la structure du document et son contenu. HTML est un langage qui utilise des balises (ou marqueurs, ou tags). Toute structure de texte est encadrée par une paire de balises. Le canevas de base d'un document HTML est le suivant :

```
<html>
  <head>
     <title> Exemple pour le cours </title>
  </head>
  <body>
   <p> C'est ici que l'on écrit le texte de la page </p>
  </body>
</html>
```

Chaque balise est reconnaissable car entourée des caractères $< >$

Certaines balises vont par 2 : <html></html>, <head></head>, <p></p>. Elles indiquent comment le texte qui est entre la paire de balises doit être structuré. La deuxième balise (la balise de fin) est construite à partir de la première (balise de début) en lui ajoutant devant le caractère /. D'autres balises sont "seules" : <br/> indique un passage à la ligne,<img/> indique l'insertion d'une image. On supposera que les noms des balises HTML, c'est-à-dire la chaîne comprises entre < (ou </) et >, est uniquement composée de lettres minuscules. Vous allez définir différentes fonctions concernant les documents HTML. On supposera que le document est représenté par une chaîne de caractères.

1. Écrire une fonction `estBaliseDebut()` qui vérifie si une chaîne `s`, reçue en argument est une balise de début. Une balise de début est entourée des caractères < et> et entre les deux, elle n'est composée que de **caractères alphabétiques minuscules**.
2. Écrire une fonction `construitBaliseFin(balDeb)` qui retourne deux valeurs, un booléen et une chaîne :
   - retourne en première valeur `True` dans le cas ou la balise `balDeb` est une balise de début, auquel cas, la deuxième valeur retournée représente la balise de fin correspondant à `balDeb`,
   - retourne `False` dans le cas ou la balise `balDeb` n'est pas conforme au langage HTML, auquel cas, la deuxième valeur retournée est une chaîne vide.

   Exemple **à compléter** :

```
balDebut = '<head>'
flag,balFin = construitBaliseFin(balDebut)
print(str(flag)+' '+ balFin)

balDebut = '<he67d>'
flag,balFin = construitBaliseFin(balDebut)
print(str(flag)+' '+ balFin)
```

```
balDebut = 'head>'
flag,balFin = construitBaliseFin(balDebut)
print(str(flag)+' '+ balFin)
```

Prévoyez les affichages qui doivent être obtenus

3. Écrire une fonction `paireBalises()` qui, prenant deux balises `b1`, `b2` en paramètre, retourne `True` si la balise `b2` est la balise de fin correspondant à la balise de début `b1`, retourne `False` sinon.

   Donnez les jeux d'essai nécessaires.

Exercice 9 :

Écrire une fonction qui indique si une chaîne de caractères représentant une expression parenthèsée est syntaxiquement correcte du point de vue des parenthèses ou pas. La fonction renvoie *-1* si l'expression est correcte et la position de la première erreur si l'expression est incorrecte. Exemple :

```
pour "(a.(b))"  elle retourne  -1
pour "(()())" elle retourne -1
pour "a.(b))(" elle retourne 5
pour "(()(()a.(b))" elle retourne 12
pour ")(" elle retourne 0
```

Chapitre 9

Dictionnaires

9.1 Introduction

9.1.1 Présentation

Les dictionnaires sont des structures de données qui, comme les tableaux, permettent de stocker des collections de valeurs. Ils sont utiles dans de nombreuses situations où les tableaux n'offrent plus assez de souplesse ou d'expressivité pour structurer un jeu de données. La particularité principale qui distingue les dictionnaires des tableaux, est que les données sont stockées sous forme de couples (clef-valeur).

9.1.2 Différentes façons de stocker et donc d'accéder aux valeurs

On rappelle que dans les tableaux les valeurs sont dans l'ordre. On peut donc accéder à une valeur en indiquant sa position au moyen d'un indice :

`tab_astro` :

indice	*0*	*1*	*2*
valeur	'Terre'	'Lune'	'Soleil'

```
print( tab_astro[1] )
'Lune'
```

Dans un dictionnaire **il n'y a pas de notion d'ordre et donc pas de position**. Pour accéder à une valeur il faut un autre moyen. Ce moyen, c'est une clef unique associée à chaque valeur lors de la définition du dictionnaire.

`dico_astro` :

clef	'Planete'	'Satellite'	'Etoile'
valeur	'Terre'	'Lune'	'Soleil'

```
print( dico_astro['Satellite'] )
'Lune'
```

Les dictionnaires sont aussi appelés tableaux associatifs dans le sens où ils associent des valeurs à des clefs (et non à des positions). Dans le dictionnaire `dico_astro` précédent, trois couples `clef-valeur` sont définis : `'Planete'-'Terre'`, `'Satellite'-'Lune'` et `'Etoile','Soleil'`.

On accède donc différemment aux valeurs stockées dans un tableau et dans un dictionnaire. La façon dont on veut accéder aux valeurs stockées va conditionner le choix entre une structure de données ou l'autre.

Pour certains jeux de données, l'utilisation d'un tableau est naturelle car il y a une correspondance simple entre la valeur et sa position. Pour d'autres jeux de données, cette correspondance n'est pas évidente. Dans de nombreux cas il sera avantageux d'utiliser des dictionnaires.

9.2 Définition d'un dictionnaire

Il existe deux façons de définir un dictionnaire.
La première consiste à donner littéralement l'ensemble des couples `clef-valeur` lors de sa déclaration. La collection de couples `clef-valeur` est donnée entre accolades (symboles `{` et `}`), et chaque couple est séparé par une virgule. Un couple `clef-valeur` est spécifié littéralement en donnant la clef suivie de la valeur qui lui est associée, les deux étant séparés par deux points (le symbole de ponctuation :), soit

`dico = { clef_1:valeur_1 , clef_2:valeur_2 , ...}`

```
dico_astro = {'Planete':'Terre', 'Satellite':'Lune', 'Etoile':'Soleil'}
```

La deuxième façon de définir un dictionnaire, consiste à déclarer un dictionnaire vide, puis de le remplir. La déclaration d'un dictionnaire vide est similaire à celle de la déclaration d'un tableau vide. Les symboles du tableau vide `[]` sont remplacés par les symboles d'un dictionnaire vide `{}` :

```
dico_astro2 = {}
```

L'ajout de nouveaux couples `clef-valeur` au dictionnaire `dico_astro2` ainsi initialisé se fait au moyen de la syntaxe suivante : `dico_astro2[clef] = valeur`

```
dico_astro2['Planete']       = 'Terre'
dico_astro2['Satellite']     = 'Lune'
dico_astro2['Etoile']        = 'Soleil'
print( str( dico_astro2 ) )
{'Planete': 'Terre', 'Satellite': 'Lune', 'Etoile': 'Soleil'}
```

► Sur ce thème : **Exercice 1, TD**

9.3 Manipulation d'un dictionnaire

Nous avons vu dans l'introduction que les clefs des dictionnaires jouent en quelque sorte le rôle des indices dans les tableaux :

- les clefs permettent d'accéder à une valeur stockée dans le dictionnaire,
- les clefs permettent de modifier une valeur stockée dans le dictionnaire.

Pour illustrer ces cas nous utiliserons les tableaux et le dictionnaires décrits précédemment dans l'introduction et dont la syntaxe de définition littérale est la suivante :

```
tab_astro = ['Terre','Lune','Soleil']

dico_astro = {'Planete':'Terre', 'Satellite':'Lune', 'Etoile':'Soleil'}
```

9.3.1 Accéder à une valeur

Dans un tableau la valeur correspondant à l'indice de position i est accessible en faisant suivre le nom du tableau de l'indice donné entre crochets.

```
print( str( tab_astro[1] ) )
'Lune'
```

Dans un dictionnaire c'est la même chose, à la différence que l'indice de position est remplacé par la clef.

```
print( str( dico_astro['Satellite'] ) )
'Lune'
```

Lorsqu'on appelle une valeur d'un tableau avec un indice qui dépasse sa taille, l'interpréteur affiche un message d'erreur.

```
tab_astro[100]
Traceback (innermost last):
 File "<stdin>", line 1, in ?
 IndexError: list index out of range
```

Il en est de même si on utilise une clef non contenue dans le dictionnaire lors d'un appel de valeur.

```
dico_astro['Galaxie']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Galaxie'
```

9.3.2 Modifier une valeur

Pour modifier une valeur dans un tableau il suffit d'affecter une nouvelle valeur à une certaine position en spécifiant l'indice de position.

```
print( str( tab_astro ) )
['Terre','Lune','Soleil']

tab_astro[0] = 'Mars'

print( str( tab_astro ) )
['Mars','Lune','Soleil']
```

De la même façon, dans un dictionnaire il suffit d'affecter une nouvelle valeur à une clef :

```
print( str( dico_astro ) )
{'Planete':'Terre', 'Satellite':'Lune', 'Etoile':'Soleil'}

dico_astro['Planete'] = 'Mars'

print( str( dico_astro ) )
{'Planete':'Mars', 'Satellite':'Lune', 'Etoile':'Soleil'}
```

9.3.3 Ajouter une valeur

Pour ajouter une valeur dans un tableau il est possible d'utiliser la concatenation de tableau. L'ajout d'une valeur modifie alors la taille du tableau.

```
1 print( str( tab_astro ) )
2 ['Terre','Lune','Soleil']
3
4 tab_astro = tab_astro + [ 'Voie Lactee' ]
5
6 print( str( tab_astro ) )
7 ['Mars','Lune','Soleil','Voie Lactee' ]
```

Pour ajouter une valeur dans un dictionnaire il suffit d'affecter une valeur à une clef non encore utilisée :

```
1 print( str( dico_astro ) )
2 {'Planete':'Terre', 'Satellite':'Lune', 'Etoile':'Soleil'}
3
4 dico_astro['Galaxie'] = 'Voie Lactee'
5
6 print( str( dico_astro ) )
7 {'Planete': 'Terre', 'Satellite': 'Lune', 'Etoile': 'Soleil', 'Galaxie': 'Voie Lactee'}
```

Il faut remarquer ici que la syntaxe utilisée pour ajouter une valeur à un dictionnaire est la même que celle employée pour modifier la valeur associée à une clef existante. Pour ajouter une valeur (ou plus exactement ajouter un couple `clef-valeur`), il faut impérativement que la clef ne soit pas encore utilisée dans le dictionnaire. En d'autres termes,

- si on utilise une clef existante, la syntaxe conduit à changer la valeur associée à la clef, et le dictionnaire ne change pas de taille,
- si on utilise une nouvelle clef, la syntaxe ajoute un couple clef-valeur au dictionnaire et augmente sa taille.

9.4 Quelques fonctions liées à l'utilisation des dictionnaires

9.4.1 Accéder à la liste des clefs

Dans certains cas il peut être utile d'avoir accès à la liste des clefs d'un dictionnaire. Cela est réalisé par la fonction `dict.keys(dico)` qui renvoie un tableau dont les éléments sont les clefs du dictionnaire `dico`.

```
1 dico_astro = {'Planete':'Terre', 'Satellite':'Lune', 'Etoile':'Soleil'}
2
3 print( str( dict.keys( dico_astro ) ) )
4 ['Planete', 'Satellite', 'Etoile']
```

Ici, les trois clefs du dictionnaire sont renvoyées dans le tableau `['Planete', 'Satellite', 'Etoile']`.

ATTENTION :

On rappelle encore qu'il n'y pas de notion d'ordre dans les dictionnaires. Le tableau retourné par la fonction `dict.values()` ne traduit pas l'ordre dans lequel les couple `clef-valeur` ont été définis...

9.4.2 Accéder à la liste des valeurs

Dans d'autres cas on peut souhaiter parcourir toutes les valeurs d'un dictionnaire indépendamment des clefs auxquelles elles sont associées. Cela est possible en utilisant la fonction `dict.values(dico)` qui renvoie un tableau contenant les valeurs du dictionnaire `dico`.

```
1 dico_astro = {'Planete':'Terre', 'Satellite':'Lune', 'Etoile':'Soleil'}
2
3 print( str( dict.values( dico_astro ) ) )
4 ['Terre', 'Lune', 'Soleil']
```

Ici, les valeurs correspondant aux 3 clefs du dictionnaire sont renvoyées dans la liste `['Terre', 'Lune', 'Soleil']`.

Attention (encore) :
On rappelle qu'il n'y pas de notion d'ordre dans les dictionnaires. Le tableau retourné par la fonction `dict.keys()` ne traduit pas l'ordre dans lequel les couple `clef-valeur` ont été définis...

9.4.3 Tester si une clef existe

`dict.has_key(mon_dico, cl)` : Est une fonction booléenne qui renvoie `True` si le dictionnaire `mon_dico` a la clef `cl` et `False` sinon.

```
1 print( str( dict.has_key( dico_astro , 'Etoile' ) ) )
2 True
3
4 print( str( dict.has_key( dico_astro , 'Galaxie' ) ) )
5 False
```

Dans le premier cas, la fonction renvoie `True` car le dictionnaire comporte la clef 'Etoile' et `False` dans le second cas car la clef 'Galaxie' est absente du dictionnaire.

▶ Sur ce thème : **Exercice 2, TD**
▶ Sur ce thème : **Exercice 3, TD**
▶ Sur ce thème : **Exercice 4, TD**

9.4.4 Supprimer une entrée (`clef-valeur`)

`del(mon_dico[cl])` : Supprime du dictionnaire la clef `cl` et la valeur qui lui est associée.

```
1 del( dico_astro['Planete'] )
2
3 print( str( dico_astro ) )
4 {'Satellite': 'Lune', 'Etoile': 'Soleil'}
```

Ici, le couple `clef-valeur` correspondant à la clef `'Planete'` est supprimé.

9.5 Les clefs des dictionnaires

9.5.1 Utiliser une variable comme clef

Comme pour les tableaux, il est possible d'utiliser comme clef d'un dictionnaire la valeur affectée à une variable.

```
c = 'Etoile'

print( str( dico_astro[ c ] ) )
'Soleil'
```

Ici, la clef `c` contient la valeur `'Etoile'`. La commande `dico_astro[ c ]` sera donc interprétée comme la commande `dico_astro[ 'Etoile' ]`.

9.5.2 Pour aller plus loin : Les clefs doivent être des constantes

Un même dictionnaire peut admettre des clefs de différents types :

```
semaine = {}

semaine[1] = 'lundi'   # Integer

semaine['Couleur'] = 'rouge'  # String

print( str( semaine ) )
{'Couleur': 'rouge', 1: 'lundi'}
```

Ici, le dictionnaire `semaine` admet la clef `1` de type entier et la clef `'Couleur'` de type chaîne de caractères.

Cependant, une limitation importante concernant les clefs des dictionnaires est qu'elles doivent correspondre à des types constants. Par exemple, un entier, un flottant, un booléen, une chaîne de caractères sont des constantes et peuvent être utilisés comme une clef de dictionnaire.

```
dico = {}
dico[12735]='Diameter'        # Integer
dico[ 6367.5] = 'Radius'      # Float
dico[True]='Vrai'             # Boolean
dico['Nom'] = 'Gaia'          # String

print( str( dico ) )
{6367.5: 'Radius', True: 'Vrai', 12735: 'Diameter', 'Nom': 'Gaia'}
```

Par contre, les tableaux ou les dictionnaires ne sont pas des constantes puisque leur contenu est susceptible d'être modifié. Les utiliser comme clef de dictionnaire provoquera alors un message d'erreur.

```
distance = [ 'Terre', 'Lune' ]

planete= {}

planete[ distance ] = 388272
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Ceci explique l'intérêt des tuples. On rappelle que les tuples sont des sortes de tableaux constants, non-modifiables. Il est donc possible de les utiliser comme clef dans un dictionnaire :

```
distance = ( 'Terre', 'Lune' )

planete= {}
```

```
4
5 planete [  distance  ] = 388272
```

9.6 Les valeurs des dictionnaires

Comme pour les tableaux, les valeurs stockées dans un dictionnaire peuvent être de types différents, sans aucune contrainte. Une valeur d'un dictionnaire peut donc être un nombre entier ou flottant, une chaîne de caractères, un booléen mais aussi des collections contenant d'autres valeurs comme un tableau ou un autre dictionnaire. Par exemple, le dictionnaire suivant contient des données hétérogènes :

clef	'Satellites'	'Rayon'	'Tellurique'	'Oceans'	'Superficie'
valeur	1	6378.137	True	*indice* / *valeur*: *0* 'Pacifique' *1* 'Atlantique' *2* 'Indien'	*clef* / *valeur*: 'Pacifique' 49.7 'Atlantique' 29.5 'Indien' 20.4
type	`<int>`	`<float>`	`<bool>`	`<list>`	`<dict>`

Il peut être codé de la façon suivante :

```
terre= {}
terre['Satellites']    = 1
terre['Rayon']         = 6378.137
terre['Tellurique']    = True
terre['Oceans']        = ['Pacifique', 'Atlantique', 'Indien']
terre['Superficie']    = {'Pacifique':49.7, 'Atlantique':29.5,'Indien':20.4}

print( str( terre ) )
{'Satellites': 1,
 'Superficie': {'Pacifique': 49.700000000000003, 'Atlantique': 29.5, 'Indien': 20.399999999999999},
  'Rayon': 6378.1369999999997,
  'Oceans': ['Pacifique', 'Atlantique', 'Indien'],
  'Tellurique': True}
```

Dans le cas où une valeur du dictionnaire est elle même une collection, il doit être possible d'accéder à une valeur de cette collection. La syntaxe est la même que pour accéder à une valeur dans un tableau multi-dimensionnel.
Ainsi, à partir du tableau précédent, pour accéder à la superficie de l'océan indien on utilise la syntaxe suivante :

```
print( str( planete['Superficie']['Indien'] ) )
20.4
```

En effet, le dictionnaire `planete` admet pour clef `'Superficie'` qui à pour valeur un dictionnaire. L'expression `planete['Superficie']` est donc un dictionnaire qui à lui-même pour clef la chaîne de caractère `'Indien'` et dont la valeur est `20.4`.
De façon analogue, pour accéder au nom du premier océan, on utilise la syntaxe suivante :

```
print( str( planete['Oceans'][0] ) )
'Pacifique'
```

En effet, le dictionnaire `planete` admet pour clef `'Oceans'` qui à pour valeur un tableau. L'expression `planete['Ocean']` est donc un tableau dont le premier élément est 'Pacifique'.

► Sur ce thème : **Exercice 5, TD**

9.7 Récapitulatif et Comparaison Dictionnaire/Tableau

Opération	Tableau	Dictionnaire
Définition d'un conteneur vide	t = [ ]	d = { }
Définition d'un littéral	t = [val_1, val_2, ...]	d = { clef_1 :val_1 , clef_2 :val_2 , ...}
Appel d'une valeur	t[indice]	d[clef]
Ajout d'un élément	t = t + [nlle_val]	d[nlle_clef] = nlle_val
Modification d'une valeur	t[indice] = nlle_val	d[clef] = nlle_val
Nombre d'éléments	len(t)	len(d)
Suppression d'un élément	del(t[indice])	del(d[clef])

TD10 : Dictionnaires

Exercice 1 : Définition d'un dictionnaire

Définissez le dictionnaire `lunes` représenté ci-dessous. Ce dictionnaire stocke le nombre de satellites de chaque planètes. Dans un premier temps vous définirez de façon littérale le dictionnaire pour les 4 premiers couples `clef-valeur` puis vous ajouterez successivement les 4 derniers couples.

clef	'Mercure'	'Venus'	'Terre'	'Mars'	'Jupiter'	'Saturne'	'Uranus'	'Neptune'
valeur	0	0	1	2	63	61	27	11

Proposez un jeu de tests permettant de vérifier que le dictionnaire est correctement défini.

Exercice 2 : Exploration d'un dictionnaire

À partir du dictionnaire défini dans l'exercice précédent, proposez un programme permettant de :

- modifier le dictionnaire précédent pour corriger une erreur qui s'est glissée dans l'énoncé. La planète Neptune comporte en fait 13 satellites.
- afficher le nombre de lunes de la Terre,
- afficher la liste des planètes,
- afficher le tableau des planètes et du nombre de leurs lunes,
- afficher le nombre total de lunes.

Exercice 3 : Nombre d'occurrences des caractères d'un texte

Proposez une fonction `occurrences(texte)` prenant comme paramètre une chaîne de caractères et renvoyant un dictionnaire contenant le nombre d'occurrences de chaque caractère présent dans un texte dont :

- les clefs sont les caractères rencontrés dans le texte (et seulement ceux-là) et
- les valeurs le nombre de leurs occurrences.

Pour implémenter cela, vous aurez besoin :

- de la fonction `dict.has_key( dico , clef )` qui permet de tester si une clef existe dans le dictionnaire. En effet, lors de l'analyse du texte, pour chaque caractère lu, vous devrez tester si il a déjà été vu. Si il a déjà été vu vous incrémenterez son nombre d'occurrences, Si il n'a jamais été vu, vous créerez une nouvelle clef dans le dictionnaire.
- de la fonction `dict.keys( dico )` qui renvoie la liste des clefs du dictionnaire `dict`. Cette fonction vous permettra de passer en revue le contenu du dictionnaire pour l'affichage final.

Vous donnerez un script de test qui affichera le tableau des occurrences.

Exercice 4 : Comptage des points à la Belote Coinchée

La belote coinchée est un jeu qui se joue à 4 joueurs par équipe de 2 et un jeu de 32 cartes. Il se distingue de la Belote par un système d'annonce. Tout à tour, avant de commencer à poser les cartes chaque joueur "passe" ou "annonce" une couleur d'atout et le nombre de points qu'il pense faire en choisissant cet atout. Celui qui propose l'annonce la plus forte définit l'atout pour la partie. Lors du comptage des points en fin de partie, chaque carte remportée par une équipe est convertie en points. Le score d'une équipe correspond au nombre de points associés aux plis qu'elle a remporté. Les règles de calcul sont les suivantes :

Figure	Nb de points à l'Atout	Nb de points Autre Couleur
As	11	11
Roi	4	4
Dame	3	3
Valet	20	2
Dix	10	10
Neuf	14	0
Huit	0	0
Sept	0	0

1. Définissez 2 dictionnaires `Atout` et `Autre` dont les couples (clef,valeur) sont respectivement la figure portée par la carte et le nombre de points associés à cette figure lorsqu'elle est de la couleur de l'atout ou d'une autre couleur.
2. Définissez une fonction `comptage()` prenant comme paramètre les 2 dictionnaires, une couleur d'atout et un ensemble de cartes. La fonction renvoie le nombre de points associés à cet ensemble de cartes. Les cartes seront passées comme une liste de tuples dont
   – la première valeur correspond à la figure de la carte (As, Roi, Dame, Valet, Dix, Neuf, Huit, ou Sept) et
   – la deuxième valeur correspond à la couleur de la carte (Pic, Coeur, Carreau, Trefle)
3. Calculez le nombre de points que les joueurs se partagent au cours d'une partie sachant que 10 points sont accordés à l'équipe qui remporte le dernier tour de jeu,
4. Définissez une fonction `affiche_carte_point` qui prend comme argument les 2 dictionnaires et la couleur d'atout et affiche l'ensemble des 32 cartes et leur valeur.

Exercice 5 :

Les ARN sont formés d'une séquence de "bases" prises parmi un alphabet de 4 lettres (A, U, C, G).
Les protéines sont forment une séquence d'acides aminés pris parmi un alphabet de 20 lettres.
On considérera que chaque ARN peut être traduit en protéine si l'on connait le code de conversion appelé "code génétique".
Pour traduire un ARN en protéine, on découpe la séquence d'ARN en groupe de 3 bases (un codon). A chaque codon correspond un acide aminé unique. Traduire une séquence d'ARN en protéine et vice versa est donc un exercice de ré-écriture.
Par exemple, si l'on considère la séquence `AUG GUU AGG UUG UGA` d'ARN, celle-ci est traduite par la séquence d'acide aminée suivante : `MVRL-`
Le détail de la traduction est donné par le tableau suivant :

ARN	code à 3 lettres	codon1 AUG	codon2 GUU	codon3 AGG	codon4 UUG	codon5 UGA
Code génétique		↓	↓	↓	↓	↓
Acide Aminés	code à 1 lettre	M	V	R	L	-

Par exemple, le premier codon d'ARN `AUG`, code pour l'acide aminé Méthionine, dont le nom abrégé à une lettre est `M`.

Afin de pouvoir travailler sur un vrai code génétique, vous allez devoir télécharger un fichier contenant le code et le lire dans Python. Le fichier qui vous est donné à la forme suivante :

```
AAA	K
AAC	N
AAG	K
AAT	N
ACA	T
```

```
6  ...
7  UUA        L
8  UUC        F
9  UUG        L
10 UUU        F
```

- chaque ligne correspond à un codon (3 lettres de l'ARN)
- La première colonne donne les 3 lettres formant le codon d'ARN.
- La deuxième colonne donne la lettre (le nom) de l'acide aminé (protéine) correspondant à ce codon,
- Chaque colonne est séparée par une tabulation.

Le fichier `CodeGenetique.tsv` téléchargeable à l'adresse :

`http://www-lipn.univ-paris13.fr/~santini/data/CodeGenetique.tsv`

Il doit être enregistré dans votre répertoire de travail.

Question 5.1 : Définissez une fonction `lire_code_genetique( 'fichier' )` qui prend en paramètre le nom d'un fichier de code génétique (son chemin) et qui renvoie un dictionnaire dont les clefs sont les codons (3 lettres de l'ARN) et les valeurs la lettre de l'acide aminé (protéine) correspondant.

Pour lire un fichier dans Python il faut connaître les opérations suivantes :

- `f_in = open('nom_du_fichier' , 'r' )` ouvre le fichier en mode lecture(`'r'`) dans la variable `f_in` et place un curseur de lecture au début de la première ligne,
- `raw_line = file.readline(f_in)` lit une seule ligne du fichier, stocke la ligne dans la variable `raw_line` sous la forme d'une chaîne de caractères et place le curseur de lecture au début de la ligne suivante,
- Lorsqu'on arrive à la fin du fichier la fonction `file.readline(f_in)` renvoie la chaîne de caractères vide ''.

Vous pourrez utiliser la fonction `stringTokenizer()` vue dans un TD/TP précédent pour décomposer la ligne lue.

Question 5.2 : On rappelle que plusieurs codons (3 lettres de l'ARN) peuvent coder pour un même acide aminé (protéine).
Définissez une fonction `reverse_genetic_code( code_genetique )` qui prend pour paramètre le dictionnaire produit la fonction précédente `lire_code_genetique()` et qui renvoie un dictionnaire dont les clefs sont les lettres de acides aminés et les valeur la liste (tableau) des codons (3 lettres de l'ARN) codant pour cet acide aminé.
Il s'agit en quelque sorte "d'inverser" le dictionnaire sans perdre d'information. A titre d'exemple les évaluations suivantes donnent les résultats indiqués :

```
1  CodeGenetique = lire_code_genetique( 'CodeGenetique.tsv' )
2  Codon = reverse_genetic_code( CodeGenetique )
3
4  print( str( Codon['R'] ) )
5  ['AGG', 'AGA', 'CGA', 'CGC', 'CGG', 'CGU']
6
7  print( str( Codon['E'] ) )
8  ['GAA', 'GAG']
```

Question 5.3 : Proposez une fonction `ARN_to_Proteine( seq_ARN , code_genetique )` qui prend pour paramètre une séquence d'ARN et le dictionnaire du code génétique et renvoie la séquence de la protéine correspondante. A titre d'exemple les évaluations suivantes donnent les résultats indiqués :

```
1 print( ARN_to_Proteine( 'AUGGUCCUAAAACGAAGAACCUGGCCGUU' ,CodeGenetique) )
2 'MVLKRRTWP'
```

Question 5.4 : Proposez une fonction `nombre_de_sequences( seq_proteines , codons )` qui prend pour paramètre une séquence de protéine et le dictionnaire des codons et qui renvoie le nombre de séquences possibles d'ARN codant pour la proteine. A titre d'exemple les évaluations suivantes donnent les résultats indiqués :

```
1 print( nombre_de_sequences( 'MVLKRRTWP', Codon) )
2 27648
```

Exercice 6 : Dictionnaire de dictionnaires

Nous voulons ici définir un programme permettant de gérer un stock d'articles. Pour cela, nous voulons être capable :

- d'ajouter une référence (prix, quantité, et couleur) dans le stock,
- de calculer le prix total du stock.

Pour cela :

Question 6.1 : Créez une fonction `ajout_reference()` sans paramètre qui renvoie un dictionnaire dont les couples `clef-valeur` sont :

Clef	Valeur
"Prix"	<float>
"Quantité"	<int>
"Couleur"	<string>

Pour paramétrer le dictionnaire renvoyé, la fonction demandera à l'utilisateur de saisir itérativement au clavier, les différentes valeurs.

Question 6.2 : Définissez le programme principal permettant, de saisir plusieurs références, puis en fin de saisie, d'afficher le montant total du stock.

Question 6.3 : Pour aller plus loin, vous pourrez enrichir le programme principal de fonctionnalités telles que :

- supprimer une référence du stock,
- modifier la quantité d'une référence,
- Afficher l'état du stock,
- ...

Chapitre 10

Tri

10.1 Tri d'un tableau

Trier un tableau, c'est le modifier pour que ses éléments soient rangés dans l'ordre croissant. Par exemple, le tableau

```
[3, 9, 5, 6, 1]
```

devient, ue fois trié :

```
[1, 3, 5, 6, 9]
```

L'ordre entre les chaînes de caractères est alphabétique, si bien qu'un tableau comme

```
['Hakim', 'Yaniss', 'Joel', 'Bastien', 'Loic', 'Remy', 'Deborah', 'Laurent', 'Ayoub']
```

devient une fois trié :

```
['Ayoub', 'Bastien', 'Deborah', 'Hakim', 'Joel', 'Laurent', 'Loic', 'Remy', 'Yaniss']
```

Il existe de nombreux algorithmes différents pour trier des algorithmes. Les algorithmes sont plus ou moins difficiles à comprendre, et plus ou moins efficaces. Parmi les algorithmes facilement compréhensibles, mais peu efficaces dès que la taille du tableau à trier augmente, notons :

- Tri à bulles
- Tri par insertion

Parmi les algorithmes avancés et efficaces avec des tableaux de grande taille, notons :

- Tri fusion (merge sort)
- Tri rapide (quick sort)
- Tri par tas (heap sort)

10.2 Complexité des algorithmes

Les performances de différents algorithmes dépendent de la nature des données à trier : déjà presque triées, ordre inverse ou presque, répartition quelconque etc. Certains algoritmes peuvent être très rapides dans certains cas, et moins dans d'autres. Avoir une idée de la configuration des tableaux à trier permet d'utiliser les algorithmes les plus rapides.

Pour trier un tableau, quelle que soit la méthode employée, les algorithmes seront amenés à effectuer un certain nombre de tests (pour savoir si une valeur est plus grande qu'une autre) et surtout un certain nombre de modifications des valeurs du tableau. Ce nombre d'opérations à effectuer pour trier un tableau croît avec la taille des tableaux. Par exemple, trier un tableau de 100 000 éléments dans un ordre quelconque demandera plus d'opérations que d'en trier un de 100 éléments, mais pas nécessairement 1 000 de plus.

La relation entre le nombre d'opérations à effectuer — donc le temps de calcul nécessaire pour le tri — et la taille du tableau n'est pas nécessairement linéaire (*i.e.*, égale à $constante \times tailledutableau$). Si n est la taille du tableau à trier,

- Pour les algorithmes peu efficaces cités plus haut, le nombre d'opérations nécessaires est de l'ordre de n^2, n étant la taille du tableau. Autrement dit, le temps de calcul croît de manière quadratique par rapport à la taille du tableau à trier.
- Pour les algorithmes efficaces mentionnés précedemment, la complexité est parfois de l'ordre de $n \times \log(n)$. Autrement dit, le surcoût à l'augmentation de la taille du tableau est moindre.

Il est possible d'analyser de manière théorique la plupart des algorithmes, et d'en déduire son ordre de complexité (n^2, $n \times \log^2(n)$ etc) : on procède alors à une analyse de complexité. C'est absolument nécessaire dès qu'on en vient à traiter de gros volumes de données. Ce n'est pourtant pas ce qui sera abordé dans la suite de ce cours. Nous nous livrerons ici plutôt à une étude empirique des temps de calcul de différents algorithmes de tri.

10.3 Tri par insertion

Parmi les algorithmes de tri faciles à mettre en oeuvre, le tri par insertion est le plus efficace avec des tableaux de petite taille. L'idée est simple :

- On considère chaque élément du tableau, en partant du premier. Ceci constitue une première boucle sur les éléments du tableau.
- A chaque étape, l'objectif est d'insérer l'élément considéré à sa place parmi ceux qui précèdent. Il faut pour cela trouver à quel endroit l'élément doit être inséré en le comparant aux autres, puis décaler les éléments afin de pouvoir effectuer l'insertion. En pratique, ces deux actions sont fréquemment effectuées en une passe, qui consiste à faire « remonter »l'élément au fur et à mesure jusqu'à rencontrer un élément plus petit.

Considérons par exemple le tableau `[9, 6, 1, 4, 8]`. A la première étape, on considère l'élément 6 et on le place à sa place parmi les éléments précédents, c'est à dire en toute première position :

9 6 1 4 8 $\longrightarrow$ 6 9 1 4 8

A la seconde étape, c'est l'élément 1 que l'on considère. Pour le faire « remonter »jusqu'à sa place, on décale le 9 puis le 6 vers la droite, puis on met le 1 en première position :

6 9 1 4 8 $\longrightarrow$ 1 6 9 4 8

A la troisième étape, c'est le 4 que l'on considère. Le 9 puis le 6 sont décalés, mais pas le 1 car sa valeur est inférieure. La place du 4 est donc à l'ancienne position du 6. On l'insère ici :

1 6 9 4 8 $\longrightarrow$ 1 4 6 9 8

La dernière étape permet de terminer le tri du tableau :

1 6 9 4 8 $\longrightarrow$ 1 4 6 9 8

Le tri par insertion n'est pas efficace dès que les tableaux sont grands. En revanche, il reste très efficace pour de petits tableaux et avec des tableaux plus grands, mais presque ordonnés.

▶ Sur ce thème : **Exercice 1, TP 10**

10.4 Fonction `range()` et tranches de tableaux

10.4.1 Rappel sur la fonction `range()`

Dans le chapitre sur les tableaux, nous avons introduit la fonction `range()` qui permet de créer des tableaux d'entiers (et d'entiers uniquement) de manière simple et rapide. Elle fonctionne sur le

modèle : `range([début,] fin[, pas])`. Elle crée alors un tableau contenant tous les entiers entre `début` compris et `fin` non compris, selon le pas `pas`. Les arguments entre crochets sont optionnels. Si l'argument `début` n'est pas spécifié, il est alors considéré comme égal à 0. Si le pas `pas` n'est pas spécifié, il est alors considéré comme égal à 1. Attention, si seulement deux arguments sont spécifiés, ils correspondent alors aux arguments `début` et `fin`. En exécutant le code

```
print(range(0,10))
print(range(10))
print(range(15,21))
print(range(0,1000,100))
print(range(2,-5,-1))
```

on obtient l'affichage

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[15, 16, 17, 18, 19, 20]
[0, 100, 200, 300, 400, 500, 600, 700, 800, 900]
[2, 1, 0, -1, -2, -3, -4]
```

10.4.2 Tranches de tableaux

Avec une syntaxe proche de celle de la fonction `range()` on peut extraire un sous-tableau d'un tableau. Plus précisément, si `tab` est un tableau alors `tab[[début]: fin[: pas]]` est le sous-tableau de `tab` formés des éléments compris entre `tab[début]` compris et `tab[fin]` non compris, selon le pas `pas`. Les arguments entre crochets sont optionnels. Si l'argument `début` n'est pas spécifié, il est alors considéré comme égal à 0. Si le pas `pas` n'est pas spécifié, il est alors considéré comme égal à 1. Attention, si seulement deux arguments sont spécifiés, ils correspondent alors aux arguments `début` et `fin`. En exécutant le code

```
tab=range(0,10)
print(tab)
print(tab[5:8])
print(tab[1:10:2])
```

on obtient l'affichage

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

10.5 Temps d'exécution d'une fonction

En Pyhton, on peut utiliser le module time pour déterminer avec précision le temps d'exécution d'un programme. Pour ce faire, il faut placer `import time` en début de programme. On utilisera la fonction `time.clock()` qui retourne un nombre flottant (`float` exprimant le nombre de secondes écoulées depuis une origine dépendant du système d'exploitation utilisé).

Mais cette origine importe peu quand on veut mesurer le temps d'exécution d'une fonction : il suffira d'enregistrer ce qui est renvoyé par `time.clock()` avant l'exécution, puis après, et enfin de faire la différence des deux temps. Pour afficher le nombre de secondes nécessaires à l'exécution de la fonction `test()`, par exemple, on peut écrire

```
t1=time.clock()
test()
t2=time.colck()
print(t2-t1)
```

Ce faisant, on affiche un nombre de secondes avec de nombreux chiffres après la virgule (l'échelle de précision de `time.clock()` est de l'ordre de la micro-seconde). La plupart les fonctions s'effectuant en quelques millisecondes, on préfère souvent afficher un nombre millisecondes.

A cet effet, rappelons la possibilité de formater l'affichage des nombres flottants dans la fonction `print()`.

TP10 : Etude empirique d'algorithmes de tri

Exercice 1 : Test du tri par insertion

Question 1.1 : Proposez une fonction de tri par insertion `triTableauInsertion(tab)`.

Question 1.2 : Testez la fonction `triTableauInsertion(tab)` de tri par insertion avec des tableaux déjà triés, triés en ordre inverse (le pire des cas) ou bien initialisés avec des valeurs aléatoires. Vous pourrez utiliser les fonctions ci-dessous pour initialiser les tableaux.

```
def construireTableauCroissant ( taille ) :
    return range(10,11+taille)
```

```
def construireTableauDecroissant ( taille ) :
    return range(11+taille,10, -1)
```

```
def construireTableauAleatoire ( taille ) :
    tab=[]
    for i in range (0,taille) :
        tab.append(random.randint(0,taille*2))
    return tab
```

Question 1.3 : Exécutez des tris par insertion de tableaux aléatoires de longueurs différentes. En 1 minute à peu près, combien d'éléments est-il possible de trier ?

Exercice 2 : Test de la fonction `list.sort`

Question 2.1 : Testez la fonction de tri `list.sort` de Python avec des tableaux aléatoires de tailles différentes. En 1 minute à peu près, combien d'éléments est-il possible de trier ?

Question 2.2 : Mesurez précisément le temps d'exécution du tri rapide en utilisant les fonctions clock() du module time décrites dans le cours.

Question 2.3 : Avec des tableaux aléatoires de même taille, selon les nombres tirés, le temps d'exécution peut varier de manière significative. Modifiez le programme principal pour que la fonction de tri soit appelée plusieurs fois de manière à afficher une moyenne pour les temps d'exécution.

Exercice 3 : Comparaison empirique entre la fonction `list.sort` et le tri par insertion

Question 3.1 : Mesurez le temps d'exécution moyen pour `list.sort` de tableaux aléatoires de 50, 100, 200, 500 et 1000 éléments. Reportez ces données sur un graphique avec :

- En abscisse le nombre d'éléments à trier
- En ordonnée le temps d'exécution

Question 3.2 : Que signifie le fait que la courbe ne soit pas une droite ?

Question 3.3 : Sur le même graphique, reportez les temps d'exécution moyens pour le tri par insertion de tableaux aléatoires de 50, 100, 200, 500 et 1000 éléments. Pour des tableaux de petite taille, quelle méthode de tri vaut-il mieux utiliser ? A partir de quelle taille de tableaux vaut-il mieux employer `list.sort` ?

Exercice 4 : Tri à bulles (Pour les plus rapides)

L'algorithme du tri à bulles parcourt le tableau, et compare les couples d'éléments successifs. Lorsque deux éléments successifs ne sont pas dans l'ordre croissant, ils sont échangés. Après chaque

parcours complet du tableau, l'algorithme recommence l'opération. Lorsqu'aucun échange n'a lieu pendant un parcours, cela signifie que le tableau est trié. On arrête alors l'algorithme.

Prenons le tableau $[5, 1, 4, 2, 8]$ et trions-le de manière croissante en utilisant l'algorithme de tri à bulles. Pour chaque étape, les éléments comparés sont écrits en gras.

Première étape :

$$[\mathbf{5}, \mathbf{1}, 4, 2, 8] \rightarrow [1, 5, 4, 2, 8]$$

Les éléments 5 et 1 sont comparés, et comme $5 > 1$, l'algorithme les intervertit.

$$[1, \mathbf{5}, \mathbf{4}, 2, 8] \rightarrow [1, 4, 5, 2, 8]$$

$$[1, 4, \mathbf{5}, \mathbf{2}, 8] \rightarrow [1, 4, 2, 5, 8]$$

$$[1, 4, 5, \mathbf{2}, \mathbf{8}] \rightarrow [1, 4, 5, 2, 8]$$

Comme $5 < 8$, les éléments ne sont pas échangés.

Continuez à dérouler l'exemple.

Question 4.1 : Proposez une fonction de tri à bulles `triBulles(tab)`.

Question 4.2 : Quel est le pire cas (celui nécessitant le plus d'échanges d'élément du tableau) ? Quand cela se produit-il ? Combien y a-t-il alors d'échanges, de tests ? Quel est le meilleur cas (celui nécessitant le moins d'échanges d'élément du tableau) ? Quand cela se produit-il ? Combien y a-t-il alors d'échanges, de tests ?

Question 4.3 : Comparez l'efficacité du tri à bulles avec celle des algorithmes précedemment étudies

Chapitre 11

Révisions

Les trois premiers exercices constituaient le sujet du controle long de janvier 2011

Exercice 1 : Index

On se propose de réaliser une sorte d'index. Pour cela, on passera par les étapes suivantes (les questions sont indépendantes).

Question 1.1 : Écrire une fonction `code_a` qui étant donnée une chaine de caractères, calcule la somme des codes ASCII de ses lettres. En Python la fonction `ord` renvoie le code ASCII du caractère passé en argument.

Exemple : `code_a("jussieu")= 106+117+115+115+105+101+117=776`

Question 1.2 : Écrire une fonction `separateur` qui, étant donnée une chaine de caractères, remplace tous les caractères non alphabétiques par un espace. En Python la fonction, `isalpha` renvoie `True` si son argument est constitué de caractères alphabétiques et `False` sinon.

Exemple : Il s'en allait ruminant son bonheur, comme ceux qui machent encore, apres son diner, le gout des truffes qu'ils digerent.

Après l'appel de la fonction séparateur :

Il s en allait ruminant son bonheur comme ceux qui machent encore apres son diner le gout des truffes qu ils digerent.

Question 1.3 : On suppose que la fonction `code_a` ne renvoie jamais la même valeur pour deux mots différents. On définit un tableau de comptage : c'est-à-dire un tableau d'entiers dont les indices varient entre 1 et M, où M est l'entier maximum que l'on peut obtenir par la fonction `code_a`. Autrement dit, chaque indice correspond à la représentation d'un mot par la valeur que renvoie la fonction `code_a`. Le contenu de chaque case de ce tableau représente le nombre d'occurrences du mot correspondant ;

Écrire un programme de comptage : étant donné un tableau `t` de mots (chaines de caractère) supposé rempli et représentant un texte. Ce programme remplit un tableau de comptage `C`, en indiquant pour chaque mot son nombre d'occurrences dans le texte.

Exemple : Si le mot "bonheur" apparait 3 fois dans le tableau `t` et `code_a("bonheur")=755`. La case `C[755]` doit avoir la valeur 3.

Question 1.4 : On souhaite pouvoir savoir quel mot du tableau `t` correspond à chacune des valeurs prises par la fonction `code_a`. À cette fin, on définit un tableau de décodage de la même taille que le tableau de comptage, chaque indice correspond à la valeur prise par la fonction `code_a` et la case correspondante contient la chaîne de caractères qui a ce `code_a` s'il y en a une dans `t`, la chaîne vide sinon.

Écrire un programme de décodage : étant donné un tableau `t` de mots (chaines de caractère) supposé rempli et représentant un texte. Ce programme remplit un tableau de décodage `D`, en plaçant chaque mot de `t` dans la case d'indice son `code_a` et en s'assurant que les autres cases

contiennent la chaîne vide.

Exercice 2 : Programme mystère

Question 2.1 : Qu'affiche le programme suivant ? (remarque : une chaîne de caractères est considérée comme supérieure à une autre lorsqu'elle est plus loin dans l'ordre alphabétique)

```
1  def grandAbsorbe(x, y):
2      if (x>y) :
3          y=x
4      else :
5          x=y
6      return x
7
8  def petitAbsorbe(x, y):
9      if (x<y) :
10         y=x
11     else :
12         x=y
13     return (x, y)
14
15 def grandRemplace(tab):
16     for i in range (0, len(tab)-1):
17         if tab[i]>tab[i+1] :
18             tab[i+1]=tab[i]
19     return tab
20
21 print("debut")
22 str1="abc"
23 str2="def"
24 str3=grandAbsorbe(str1, str2)
25 print(str1)
26 print(str2)
27 print(str3)
28
29 str1="abc"
30 str2="def"
31 (str1, str2)=petitAbsorbe(str1, str2)
32 print(str1)
33 print(str2)
34
35 t1=[2, 1, 4, 3]
36 print(t1)
37 t2=grandRemplace(t1)
38 print(t1)
39 print(t2)
```

Question 2.2 : Écrire `grandRemplace2` qui renvoie le nombre d'éléments modifiés par `grandRemplace` dans `tab`. Quels sont les tableaux `tab` pour lesquels `grandRemplace2` renvoie 0 ?

Question 2.3 : Écrire une fonction `grandRemplace3` qui renvoie le même résultat que `grandRemplace` mais qui ne modifie pas son argument.

Exercice 3 : Manipulation des éléments d'un tableau

Dans cet exercice, les questions sont indépendantes. On supposera pour chaque fonction que le tableau passé en paramètre est correct.

Lors d'une enquête de satisfaction pour un produit, des clients étaient invités à donner une note entière entre 0 et 20. Les 500 réponses ont été collectées dans le tableau d'entiers donné

ci-dessous.

```
resultat =[3, 7, 9, 0, 18, 5, 4, 19, 8, 3, 10, 7, 14, 13, 0, 18, 17, 5,
16, 4, 5, 1, 20, 6, 16, 10, 13, 12, 6, 5, 20, 8, 9, 15, 3, 17, 12, 10,
19, 3, 13, 4, 13, 14 , 17, 17, 15, 14, 18, 6, 17, 9, 9, 6, 15, 17, 11,
8, 8, 0, 19, 19, 13, 18, 11, 1, 16, 5, 16, 7, 18, 20, 3, 16, 3, 9, 1,
3, 20, 11, 6, 19, 0, 13, 10, 14, 13, 5, 1, 16, 2, 19, 10, 2, 14, 6, 8,
15, 4, 6, 8, 20, 4, 2, 0, 18, 19, 20, 13, 1, 13, 16, 12, 12, 7, 19,
13, 11, 3, 18, 10, 7, 20, 0, 4, 4, 16, 14, 19, 0, 0, 16, 4, 0, 7, 18,
15, 8, 8, 6, 1, 12, 17, 9, 1, 4, 13, 0, 8, 15, 19, 4, 3, 1, 3, 1, 1,
1, 6, 10, 15, 8, 14, 17, 14, 3, 5, 0, 19, 6, 8, 13, 14, 2, 1, 12, 4,
1, 10, 17, 11, 7, 11, 12, 6, 10, 8, 18, 13, 15, 12, 14, 8, 2, 2, 3,
17, 6, 9, 17, 9, 19, 16, 8, 18, 0, 10, 15, 2, 17, 2, 3, 13, 19, 13,
12, 12, 2, 2, 4, 17, 17, 17, 2, 11, 19, 13, 8, 0, 9, 8, 5, 4, 13, 9,
9, 19, 15, 19, 1, 19, 10, 20, 1, 11, 8, 13, 0, 10, 6, 5, 1, 11, 0, 8,
13, 5, 0, 15, 9, 7, 0, 10, 0, 1, 12, 12, 10, 19, 17, 0, 6, 16, 12, 8,
12, 14, 13, 15, 20, 7, 17, 1, 18, 15, 13, 6, 14, 18, 8, 13, 14, 8, 14,
7, 16, 2, 19, 14, 6, 17, 8, 11, 14, 16, 11, 19, 2, 20, 16, 2, 13, 14,
20, 6, 13, 20, 9, 3, 19, 3, 0, 7, 0, 16, 7, 4, 15, 16, 17, 2, 14, 3,
15, 12, 16, 18, 1, 5, 9, 2, 1, 5, 9, 5, 13, 10, 11, 16, 6, 16, 8, 13,
11, 6, 8, 2, 5, 2, 5, 9, 1, 7, 20, 19, 4, 14, 2, 8, 9, 5, 9, 9, 5, 15,
12, 3, 7, 15, 17, 6, 0, 7, 2, 17, 19, 20, 15, 1, 11, 11, 18, 9, 7, 1,
1, 17, 1, 17, 10, 2, 5, 1, 14, 7, 2, 3, 0, 4, 2, 9, 17, 3, 6, 7, 18,
14, 14, 4, 10, 3, 15, 16, 17, 6, 15, 12, 17, 9, 5, 14, 14, 6, 4, 4, 4,
17, 10, 10, 11, 13, 16, 3, 0, 2, 18, 3, 20, 5, 16, 6, 0, 8, 3, 3, 10,
2, 0, 6, 11, 12, 9, 18, 8, 6, 2, 0, 0, 19, 12, 17, 5, 19, 12, 17, 4,
0, 20, 8, 3, 7, 6, 10, 0, 3, 4, 4, 6, 11, 0, 13, 14, 17, 20, 12, 3,
18, 3, 13, 14]
```

Question 3.1 : On souhaite connaître la répartition précise des notes qui ont été données, c'est-à-dire le nombre de fois que chaque note a été donnée. Écrire une fonction qui retourne un tableau de 21 éléments indicés de 0 à 20 dont la n-ième composante contiendra le nombre de fois où la note n a été donnée.

Question 3.2 : On souhaite connaître les notes qui ont été données le plus et moins fréquemment. Écrire une fonction qui à partir d'un tableau de notes affiche les notes qui ont été données le plus et le moins souvent ainsi que le nombre de fois qu'elles ont été données. Utiliser cette fonction pour afficher les notes données le plus et le moins fréquemment, ainsi que le nombre de fois qu'elles ont été utilisées dans le cas de l'enquête mentionnée ci-dessus.

L'exemple a été choisi afin que le maximum et le minimum des fréquences soient uniques.

Question 3.3 : Afin d'avoir une vision plus précise, on souhaite obtenir le pourcentage de personnes étant très satisfaites (note entre 16 et 20), plutôt satisfaites (note entre 11 et 15), plutôt insatisfaites (note entre 6 et 10) et pas du tout satisfaites (note entre 0 et 5). Pour cela, créer une fonction prenant en paramètre le tableau retourné par la fonction de la question 1 et retournant un tableau de 4 cases, chaque case contenant le pourcentage de satisfaction.

Question 3.4 : Pour simplifier la lecture des résultats, écrire les pourcentage de satisfaction sous la forme de graphique selon :

```
Tres satisfaites        : *******************************
Plutot satisfaites      : ***********************
Plutot instatisfaites   : **********************
Tres insatisfaites      : **********************
```

où le nombre d'étoiles de chaque ligne correspond à la partie entière inférieure du pourcentage de satisfaction associé.

Pour cela, créer une fonction prenant en paramètre le tableau de pourcentage et affichant sous forme graphique les résultats.

On rappelle que quand le paramètre passé à la fonction `int` est un nombre réel (à virgule flottante) x , la fonction retourne la partie entière de x.

Exercice 4 : Divisible ou pas ?

Écrire une fonction prenant comme paramètres deux entiers min et max qui vérifie d'abord que $min \leq max$ et qui affiche les nombres entiers, compris entre les valeurs min (inclus) et max (inclus), et qui sont pairs. De plus, si le cas échéant, le nombre (en plus d'être divisible par 2) est divisible par 3, le message "de plus ce nombre est divisible par 3". On écrira un programme principal où les valeurs min et max seront demandées à l'utilisateur et la fonction est appelée.

On vous rappelle que l'opérateur % (modulo) permet de calculer le reste de la division euclidienne.

Exemple : Si $min = 3$ et $max = 7$

Affichage obtenu

```
4 est divisible par 2
6 est divisible par 2 de plus ce nombre est divisible par 3
```

Exercice 5 : Calcul du de la clé d'un numéro de SIREN

(Source Wikipédia) En France, SIREN (Système d'Identification du Répertoire des ENtreprises) est un code INSEE unique qui sert à identifier une entreprise française. Il existe au sein d'un répertoire géré par l'INSEE : SIRENE. Il est national, invariable et dure le temps de la vie de l'entreprise. L'INSEE attribue un identifiant à toute personne juridique, physique ou morale, introduite dans le répertoire SIRENE sur demande des organismes habilités, en général le Centre de formalités des entreprises (CFE). Il correspond donc au numéro d'identification au répertoire (NIR) des personnes physiques. Il est composé de neuf chiffres, les huit premiers sont attribués séquentiellement, sauf pour les organismes publics commençant par 1 ou 2, le neuvième est une clé de contrôle.

Le numéro SIREN est composé de 8 chiffres, plus un chiffre de contrôle qui permet de vérifier la validité du numéro.

C'est ce chiffre de contrôle que nous voulons engendrer. La clé de contrôle utilisée pour vérifier l'exactitude d'un identifiant est une clé suivant l'algorithme de Luhn. ipe est le suivant : on multiplie les chiffres de rang impair à partir de la droite par 1, ceux de rang pair par 2 ; la somme des chiffres obtenus doit être congrue à zéro modulo 10, c'est-à-dire qu'elle doit être multiple de 10.

Exemple : 1 3 9 2 8 2 3 7 ? La somme des chiffres suivants : $7+6+2+16+2+18+3+2 = 56$ n'est pas divisible par 10. Pour que cette valeur le soit, il faut rajouter 4. $56 + 4 = 60$ qui est divisible par 10. Le numero : 1 3 9 2 8 2 3 7 4 est donc un numero de SIREN correct.

Nous représenterons ce numéro SIREN par un tableau d'entiers ; et le nombre La case d'indice 8 représentera la clé à calculer..

Votre programme devra :

Question 5.1 : Demander à l'utilisateur les huit premiers chiffres du numéro de SIREN

Question 5.2 : Calculer la clé de contrôle et compléter le tableau dans une fonction `calcul_cle`

Question 5.3 : Faire un affichage qui permet de valider cette clé, c'est-à-dire d'afficher la somme vue précédemment (en tenant compte de la clé) et vérifier que le nombre est bien divisible par 10 en affichant par exemple le reste de la division par 10.

Exercice 6 : Programme mystère

Qu'affiche le programme suivant ?

```
def augmente1(x):
    x+=1

def augmente2(x):
```

```
5       return x+1
6
7   def augmente3(x):
8       x+=1
9       return x
10
11  def diminue1(tab, x):
12      for i in range(0, len(tab)) :
13          tab[i]=tab[i]-x
14
15  def diminue2(tab, x):
16      for elem in tab :
17          elem-=x
18
19  def diminue3(tab, x):
20      res=[]
21      for elem in tab :
22          res=res+[elem-1]
23      return res
24
25  def diminue4(tab, x):
26      copie=[]
27      for elem in tab :
28          copie+=[elem]
29      tab=diminue3(copie, x)
30
31  print("debut")
32  y=1
33  augmente1(y)
34  print(y) # 1er print
35  x=1
36  augmente1(x)
37  print(x) # 2e print
38  augmente2(y)
39  print(y) # 3e print
40  x=augmente2(x)
41  print(x) # 4e print
42  y=augmente3(x)
43  print(x) # 5e print
44  print(y) # 6e print
45
46  print("diminue1")
47  t=range(5, 10)
48  tab=t
49  print(t) # 7e print
50  print(tab) # 8e print
51  diminue1(t, x)
52  print(t) # 9e print
53
54  print("diminue2")
55  diminue2(t, x)
56  print(tab) # 10e print
57
58  print("diminue3")
59  t=diminue3(t, x)
60  print(t) # 11e print
61  print(tab) # 12e print
62
```

```
63 print("diminue4")
64 diminue4(tab, x)
65 print(tab) # 13e print
```

Exercice 7 : Recherche dichotomique

Dans le cas général, la recherche d'un élément dans un tableau se fait en comparant l'élément que l'on recherche à chaque case du tableau. Cependant, si le tableau est déjà trié, on peut utiliser cette propriété pour effectuer une recherche sans comparer l'élément à toutes les cases du tableau. Une telle recherche est appelée recherche dichotomique.

Question 7.1 : Si l'on compare l'élément à une case quelconque du tableau, que peut-on conclure de cette comparaison ? Déterminer alors l'idée générale de la recherche dichotomique.

Question 7.2 : Écrire une fonction de recherche dichotomique et appliquez-la à un tableau de chaînes de caractères.

Exercice 8 : Ajout d'un élément dans un tableau trié

Lorsqu'un tableau est trié, il est important, lorsque l'on lui ajoute un élément, de faire en sorte que le tableau reste trié. Il est donc impératif d'insérer l'élément au bon endroit dans le tableau. Écrivez une fonction permettant d'ajouter un élément dans un tableau trié de manière à ce que le tableau reste trié après l'ajout de l'élément.