Programmation

Objectif : Les objectifs de programmation en première NSI sont traités après une introduction rappelant les principes de base qui ont, en principe, été étudiés en classe de seconde :

- Introduction aux fonctionnalités de base de Python version 3 : affectations, variables, séquences d'instructions, instructions conditionnelles, boucles bornées et non bornées, fonctions.
- Données de types construits : p-uplets (tuples), listes, listes de listes, dictionnaires par clés et valeurs, ensembles, fichiers TXT et CSV.

Plan du cours :

- Introduction à la programmation en Python
- Séquences conditionnelles et boucles
- Types de données construits
- Fonctions itératives et récursives

Utilisées dans tout ce chapitre, les fonctions sont plus spécifiquement traitées à la fin pour introduire certains autres aspects de la programmation : valeurs de retour, portée des variables, jeux de test. Le traitement des données en table est seulement abordé ici ; il est traité plus complètement dans le chapitre d'algorithmique (chapitre 3).

1. Introduction à la programmation en Python

Aperçu historique et généralités :

Le langage Python doit tout à Guido van Rossum (1956-), son principal auteur et régulateur, considéré comme Benevolent Dictator For Life dans la communauté pythonienne jusqu'en 2018, date à laquelle il se retire officiellement du projet. Le nom Python vient de la troupe des Monty Python. La 1re version de Python date de février 1991 (version 0.9.0), viennent ensuite la version 1.0 (janvier 1994), la version 2.0 (octobre 2000) et la version 3.0 (décembre 2008). En juillet 2019, les développeurs travaillent sur la version 3.8.0.
Python est un langage de programmation interprété[1] *multiplateformes (Windows, MacOS, Linux, etc.), facile d'accès, riche de potentialités et open source (libre de droit et gratuit) qui est utilisé autant dans l'industrie que dans la formation (universitaire et lycées).*

Le nom de l'éditeur IDLE[2] *– Integrated DeveLopment Environment selon Guido van Rossum – pourrait être une référence à Eric Idle, un membre fondateur de la troupe. Il s'agit d'un environnement de développement intégré pour Python, particulièrement simple, qui accompagne ce langage depuis 1998 et que j'utiliserai dans ce cours. Il existe de nombreux autres environnements de développement pour Python comme Pyzo ou Spyder, plus élaborés que IDLE qui peuvent lui être préférés.*

1. Contrairement à un langage compilé, un langage interprété est traduit en langage machine lors de son exécution
2. Tout savoir sur IDLE à l'adresse `https://docs.python.org/3/library/idle.html`

a. Mode interactif

Le *shell* de Python (la console) se présente avec une invite de commande à trois chevrons `>>>`.
Les opérations usuelles peuvent être effectuées dans ce mode, comme avec une calculatrice :

- Addition, soustraction : `>>> 2+3` retourne `5` (un entier),
  `>>> 4.5-0.5` retourne `4.0` (un flottant, repéré par le point décimal)
- Multiplication : `>>> 10*5` retourne `50`, `>>> 10*0.5` retourne `5.0`
- Division : `>>> 10/5` retourne `2`, `>>> 10/4` retourne `2.5`,
  `>>> 10/3` retourne `3.3333333333333335` (petit problème d'arrondi...)
- Division entière (partie entière de la division : `>>> 10//4` retourne `2`, `>>> 10//3` retourne `3`, avec un quotient négatif, `>>> -10//4` retourne `-3` (la valeur entière plancher) et non `-2`.
- Reste de la division entière : `>>> 10%5` retourne `0`, `>>> 10%3` retourne `1`,
  `>>> 5.3%2` retourne `1.2999999999999998`...
- Puissance : `>>> 5**2` retourne `25`, `>>> 5**-1` retourne `0.2` (inverse de 5),
  `>>> 5**0.5` retourne `2.23606797749979` (racine carrée de 5)

Définition 2.1 (Variables) Une variable est un nom donné à une valeur.
La déclaration du nom et la 1re affectation (initialisation) se font simultanément en Python.
Une fois qu'une variable a été initialisée, elle peut être réutilisée dans un calcul.

Remarques :

- Un identificateur de variable dénote un emplacement dans la mémoire dans lequel une valeur est stockée. Les noms de variables sont sensibles à la casse : les noms `Eff` et `eff` désignent deux variables différentes. Il faut choisir des noms significatifs (préférer `best_player` à `bp`) en évitant les accents et les caractères spéciaux. Minuscules, majuscules, chiffres (mais pas au début du nom) et le tiret bas `_` uniquement !
- L'opération de base pour modifier la valeur d'une variable est l'affectation et s'écrit avec le signe `=` (attention à ne pas le confondre avec l'opérateur d'égalité (dans un test) qui s'écrit avec un double signe égal `==`).
- Un exemple : `>>> a=3` ne retourne rien (c'est une initialisation),
  `>>> a=a+1` ne retourne rien (c'est une affectation : on ajoute 1 à la valeur actuelle de `a` et le résultat remplace cette valeur). Pour connaître la valeur stockée actuellement dans la variable, il suffit de taper son nom dans la console (et valider) : `>>> a` retourne 4.

En python, il est possible d'utiliser un raccourci pour effectuer un opération simple sur une variable :

- Incrémentation ou décrémentation : `a=a+1` peut s'écrire `a+=1` et `a=a-2` peut s'écrire `a-=2`.
- Le même principe est valable pour les autres opérations : `a=a*3` peut s'écrire `a*=3`,
  `a=a//4` peut s'écrire `a//=4`, `a=a%5` peut s'écrire `a%=5` et `a=a+" !"` peut s'écrire `a+=" !"`
  (cela ne donnera pas d'erreur seulement si `a` est de type *string*).

Définition 2.2 (Types) Les valeurs manipulées par Python sont typées.
Les types de base sont :

- `int` : entiers relatifs sur lesquels portent les opérateurs `+`, `-`, `*`, `//`, `%`, `**`, etc.
- `float` : flottants sur lesquels portent les opérateurs `+`, `-`, `*`, `/`, etc.
- `bool` : booléens `True` et `False` sur lesquels portent les opérateurs `and`, `or`, `not` et qui peuvent être retournés par des opérateurs de comparaison `<`, `<=`, `>`, `>=`, `==`, `!=`.
- `str` : chaînes de caractères constituées d'une séquence de caractères entre apostrophes ou guillemets, sur lesquelles s'applique l'opérateur `+` de concaténation.

Remarques :

- Le typage est dynamique : il peut être modifié en cours d'utilisation. Cela arrive lorsqu'on divise une variable entière : le résultat est flottant. On peut forcer le transtypage en utilisant une fonction de *cast* : `>>> int(5.0)` par exemple retourne `5` et `>>> int(5)` retourne `5.0`. Par

contre `>>> int(5.2)` retourne `5` aussi (la fonction `int()` transforme tout nombre en sa partie entière). On peut transformer une chaîne de caractères contenant un nombre (`>>> int("5")` retourne `5`), mais pas une chaîne contenant une lettre.

- L'utilisation du type flottant pour représenter les nombres réels n'est pas sans poser quelques problèmes de précision (voir le chapitre 1). Le type entier est le plus sûr, en ce qui concerne les nombres, d'autant plus qu'il n'y a pas de limite physique à la valeur de l'entier en Python (contrairement à d'autres langages). Pour le type *string*, Python utilise le système d'encodage UTF-8 : on n'est pas obligé de se limiter aux caractères du code ASCII, même si parfois cette limitation subsiste (par exemple sur le module Python des calculatrices Numworks).
- Pour connaître le type d'une variable, il suffit d'utiliser la fonction `type()` : `>>> type("5")` retourne le message `<class 'str'>` tandis que `>>> type(5.)` retourne le message `<class 'float'>` (le point derrière un entier suffit pour obtenir le type flottant). Un test du type (par exemple `type("5")==str` retourne `true`) permet d'éviter parfois une opération qui déclencherait une erreur : `b=a+" !"` déclenchera une erreur si la variable `a` n'est pas du type `str`. Il faudra, dans ce cas, tester le type de `a` (`if type(a)==str : b=a+" !"`) ou bien forcer le transtypage (`b=str(a)+" !"`).

b. Mode de programmation

Pour passer en mode programmation sur IDLE, il suffit de sélectionner `New File` (nouveau fichier) dans le menu `File`. Une nouvelle fenêtre s'ouvre dans laquelle on peut écrire le programme.

Celui-ci, une fois terminé et enregistré avec l'extension `py` (créer un dossier spécifique pour les programmes dans le répertoire `Documents`), sera exécuté en sélectionnant `Run Module` dans le menu `Run`.

Affichage avec `print`

Le(s) résultat(s) de l'exécution s'affiche(nt) dans la console si le programme contient au moins une instruction utilisant la fonction `print()`.

Le programme suivant va provoquer une écriture sur la console :

```
a=3
b=a**2
print("le carré de",a,"est",b)
```

programme test.py

```
>>>
le carré de 3 est 9
```

affichage console

Syntaxe de l'argument de `print()` : plusieurs variables ou constantes (pas nécessairement du même type) peuvent être affichées à la suite les unes des autres. Python intercale un espace entre ces valeurs et place à la fin par défaut, un caractère de retour à la ligne (`\n`).
Si on veut supprimer ce retour à la ligne, on doit utiliser l'option `end=""`, placée à la fin de l'argument (`end=" "` a pour effet d'insérer un espace à la place de `\n`).
Voici un 2e programme qui exploite cette possibilité, et qui montre qu'une opération sur les variables peut être effectuée juste au moment précédent l'affichage :

```
a=3
b=a**2
print("le carré de",a,"est",b,end=" ")
print("et son cube est",a*b)
```

programme test.py

```
>>>
le carré de 3 est 9 et son cube est 27
```

affichage console

Une autre possibilité pour formater un affichage : utiliser la méthode `format()` de la classe `string` en insérant un couple d'accolades à la place d'une variable au sein d'une chaîne de caractères, la variable étant passée en argument de la méthode `format()`. Cela parait compliqué mais en réalité, c'est simple et proche de l'écriture réelle. Constatez le par vous-même sur ce 3e exemple :

```
a=3
b=a**2
print("le carré de {} est {} et son cube est {}".format(a,b,a*b))
```

programme test.py

```
>>>
le carré de 3 est 9 et son cube est 27
```

affichage console

Les modules avec `import`

Les fonctions Python utilisées jusque là sont présentes par défaut dans l'interpréteur : ce sont des fonctions *built-in*. La liste de ces fonctions est assez grande, puisqu'il y en a plus de soixante :

abs	all	any	ascii	bin	bool	bytearray	bytes	callable
chr	compile	complex	delattr	dict	dir	divmod	enumerate	eval
exec	filter	float	format	frozenset	getattr	globals	hasattr	hash
help	id	input	int	isinstance	issubclass	iter	len	list
locals	map	max	min	next	object	oct	open	ord
pow	print	property	range	repr	reversed	round	set	setattr
slice	sorted	str	sum	super	tuple	type	vars	zip

Bien sûr, nous n'allons pas ici les étudier toutes[3].
Aussi nombreuses que soient ces fonctions, il en manque pour de nombreuses applications : les fonctions utiles en mathématiques sont regroupées dans une bibliothèque spéciale – un module – que l'on doit importer si nécessaire.

La fonction `import` est spéciale : on l'écrit généralement en début de programme pour importer tout ou partie du module (l'importation est alors effectuée pour toute la durée d'utilisation du programme) mais on peut écrire l'importation n'importe où dans le programme (l'importation ne concerne pas la partie du programme située en avant).
Si on a besoin d'une seule fonction du module, par exemple de la fonction `sqrt` (racine carrée) du module `math`, on peut écrire `from math import sqrt` et utiliser cette fonction telle quelle : par exemple `a=sqrt(2)`.

Si on veut utiliser deux fonctions du même module, ou davantage, on peut importer tout le module en écrivant `from math import *` mais cela peut être gênant car certaines fonctions du module peuvent avoir des noms qu'on souhaite redéfinir. Le risque de conflits entre les fonctions importées et les fonctions que l'on va créer ne doit pas être négligé. Généralement, on n'importe que les fonctions nécessaires, en les séparant par une virgule. Supposons qu'on utilise dans un programme les fonctions `sin` et `cos` : il suffit d'écrire `from math import sin,cos`. Notez qu'il est possible d'importer le module en écrivant `import math` et d'utiliser les fonctions nécessaires en les préfixant, par exemple `math.cos(0)` retourne `1`.

Donnons un exemple, en important les fonctions du module `math` au moyen de l'instruction `from math import *` et en définissant dans le programme la fonction `factorial`. Cette fonction étant déjà définie dans le module, selon la place de l'instruction d'importation il va y a voir un écrasement de notre fonction `factorial` (ce qui n'est pas souhaitable) par celle du module ou bien le contraire (notre fonction écrase celle du module, ce qui n'est pas forcément souhaitable non plus).

```
def factorial(n):
    if n%2==1 : return n
    else : return n//2
from math import *
a=6
print("factorial({})={}".format(a,factorial(a)))
```

math.factorial l'emporte

```
>>>
factorial(6)=720
```

```
from math import *
def factorial(n):
    if n%2==1 : return n
    else : return n//2
a=6
print("factorial({})={}".format(a,factorial(a)))
```

factorial l'emporte

```
>>>
factorial(6)=3
```

```
def factorial(n):
    if n%2==1 : return n
    else : return n//2
import math
a=6
print("factorial({})={}".format(a,math.factorial(a)))
print("factorial({})={}".format(a,factorial(a)))
```

math.factorial et factorial n'ont pas le même nom

```
>>>
factorial(6)=720
factorial(6)=3
```

Remarque : on peut obtenir une aide pour un module importé en utilisant le *built-in* `help`.

3. Fonctionnalités des *built-in* Python sur `https://docs.python.org/3.3/library/functions.html`

Si je tape dans la console `import math`, le module étant importé, je peux obtenir l'aide sur ce module en tapant `help(math)` : cela affiche un texte assez long qui décrit toutes les fonctions du module et qui commence par :

```
Help on built-in module math:
NAME math
DESCRIPTION This module is always available. It provides access to the mathematical
functions defined by the C standard.
FUNCTIONS acos(...)
...
```

Voici la liste des fonctions du module `math` :

acos	acosh	asin	asinh	atan	atanh	atan2	ceil	copysign
cos	cosh	degrees	erf	erfc	exp	expm1	fabs	factorial
floor	fmod	frexp	fsum	gamma	hypot	isfinite	isinf	isnan
ldexp	lgamma	log	log10	log1p	log2	modf	pow	radians
sin	sinh	sqrt	tan	tanh	trunc	e	pi	

Les modules présents dans la distribution Python de votre ordinateur sont certainement nombreux [4]. Citons le module `random` qui contient, entre autres, des fonctions générant des nombres aléatoires (`random`, `randint`, `randrange`, ...) ou permettant d'obtenir un élément tiré au hasard dans une liste (`choice`, ...).
Certains modules permettent de dessiner (modules `turtle` ou `tkinter`), d'autres donnent des informations sur les dates ou les durées (modules `datetime` ou `calendar`), etc. Dans 99,9% des cas, quelle que soit l'utilisation que l'on veut faire de Python, il y a un module spécialisé qui existe déjà.

Il est possible cependant, et même très facile, de créer de nouveaux modules. Si vous avez de nombreuses fonctions qui sont utilisées par plusieurs programmes, il est intéressant d'enregistrer ces fonctions à part dans un programme « utilitaire » (c'est-à-dire un module) et de faire appel à lui dans d'autres programmes, grâce à la fonction `import`.
Si j'ai enregistré, entre autres, une fonction `longueur()` dans un fichier `dimension.py`, je peux faire appel à cette fonction dans un autre programme (situé dans le même dossier que `dimension.py`), en l'important dans mon programme par l'instruction `from dimension import longueur`. J'accède alors à cette fonction comme si elle était dans mon programme.

Les entrées avec `input`

L'interaction entre l'utilisateur et le programme ne se limite pas à l'affichage sur la console : il est possible d'entrer des données à l'intérieur d'un programme en cours d'exécution par l'intermédiaire de la fonction *built-in* `input`. Cette fonction a pour syntaxe `input(<chaine>)` où `<chaine>` est une suite de caractères entrée au clavier (l'entrée standard). Si un programme contient l'instruction `t=input(<chaine>)`, lors de l'exécution il va y avoir `chaine` qui s'affiche à la console, le programme est en attente d'une entrée au clavier. La saisie est terminée lorsque l'utilisateur appuie sur la touche `Enter`, le programme continue alors son exécution et affecte ici la chaîne de caractères saisis à la variable `t`.
`input(<chaine>)` retourne toujours une chaîne de caractères, même si les caractères sont des chiffres : c'est à l'utilisateur de convertir la saisie en entier ou en flottant, selon les attentes du programme. On peut tester si la saisie est un nombre entier avec la méthode de la classe *string* `isnumeric()` : par exemple `"123".isnumeric()` retourne `True` alors que `"1.3".isnumeric()` retourne `False`.
Pour convertir la saisie en entier ou en flottant, utiliser l'opérateur de *cast* adapté `int()` ou `float()`.

Les commentaires

Des commentaires peuvent être ajoutés en fin de ligne, après le caractère `#`.
Si on veut écrire un commentaire sur plusieurs lignes, il vaut mieux utiliser la syntaxe
`""" <commentaire sur plusieurs lignes> """`.

4. La bibliothèque de modules standards est décrite dans la documentation `https://docs.python.org/fr/3/library/index.html`

Les commentaires font partie des bonnes pratiques de programmation, au même titre que des choix explicites pour les noms de variables : si le programmeur veut espérer se comprendre lui-même et être compris par les autres, il doit absolument respecter ces recommandations. Dans l'exemple ci-dessous j'utilise un long commentaire pour dire comment s'appelle le programme, ce qu'il fait, ce qu'il prend en entrée, etc. On peut ajouter d'autres informations comme le nom de l'auteur, la date de création, un historique des différentes version... Cela est surtout important pour les grands projets impliquants plusieurs personnes et de nombreux programmes dont le développement et la maintenance s'étale sur plusieurs années.

Pendant la phase d'écriture d'un programme, on peut aussi « mettre en commentaire » une séquence d'instructions afin d'en isoler une partie à tester. On peut commenter aussi des parties obsolètes ou utiles seulement dans certains cas. Par exemple, on peut être amené à afficher le contenu de certaines variables pour tester le bon fonctionnement d'un programme. Quand ce besoin ne se fait plus sentir, on peut mettre l'instruction d'affichage en commentaire. Attention cependant, dans la phase finale, à supprimer tous les commentaires devenus superflus.

Exemple 1 – Écrivons un programme qui demande une vitesse donnée en km/h et qui l'affiche en m/s après l'avoir arrondi au centimètre près.
Il y a un petit traitement mathématique pour connaître la réponse : on doit multiplier par 1000 et diviser par 3600. Le résultat étant en m, il suffit de l'arrondi à 2 chiffres après la virgule pour obtenir l'arrondi au centimètre.
La fonction d'arrondi est une des fonctions *built-in*, `round()`, qui prend deux valeurs en argument : le nombre à arrondir et le nombre de chiffres après la virgule, séparés par une virgule.
Comme un peu plus haut, ce programme utilise un test pour afficher la bonne réponse si un entier est entré et un message indiquant l'erreur si un autre caractère qu'un chiffre est entré. L'étude de la syntaxe Python traduisant la tournure « si ... alors ... sinon ... » fait l'objet de la partie suivante.

```
""" Conversion d'une vitesse : km/h >> m/s
    Entrée : un nombre entier
    Sortie : un nombre entier (arrondi à l'entier le plus proche) """
vitesse=input("Entrer une vitesse en km/h : ")
if vitesse.isnumeric() :
    vitesse=round(int(vitesse)/3.6,2)
    print("Cette vitesse est égale à {}m/s".format(vitesse))
else : print("Entrer une valeur entière !")
```

```
>>>
Entrer une vitesse en km/h : 60
Cette vitesse est égale à 16.67m/s
```

```
>>>
Entrer une vitesse en km/h : 2.5
Entrer une valeur entière !
```

2. Séquences conditionnelles et boucles

Les outils de programmation de cette partie sont indispensables et existent dans tous les langages de programmation. Il y a des différences de syntaxe cependant entre les langages. Ici, comme dans tout le chapitre, nous nous limiterons à Python.

a. Séquences conditionnelles

Les instructions conditionnelles utilisent des tests, donc des booléens (voir chapitre 1).
Une instruction conditionnelle permet de programmer ce type de situation :

- si un test est vrai (le booléen vaut `True`) alors on effectue un bloc d'instructions,
- sinon (le booléen vaut `False`) on en effectue un autre,
- dans tous les cas, le programme continue avec ce qui suit ces blocs d'instructions.

Une instruction conditionnelle s'écrit, dans le langage Python :

```
if <test> :
    <instructions 1>
else :
    <instructions 2>
```

La partie `else : ...` est une option. Elle peut ne pas être présente.
Il est possible d'imbriquer une instruction conditionnelle dans une autre, ad libitum.

S'il n'y a qu'une seule instruction dans un bloc, on peut la mettre juste après les deux-points, sinon il faut mettre le bloc d'instructions sur d'autres lignes, en respectant l'indentation (décalage régulier d'une tabulation vers la droite) car c'est l'indentation d'un bloc qui permet au programme de savoir où se situe la fin du bloc.

Si on est amené à écrire un nouveau test, juste après un `else` (`else : if <test> ...`), Python dispose de l'instruction `elif <test> : ...` qui permet de simplifier l'écriture car une nouvelle indentation n'est pas requise. On peut enchaîner plusieurs instructions `elif <test> : ...`, nous verrons dans quel but plus loin. Les instructions `elif <test> : ...` peuvent toujours être suivies (ou pas) d'une instruction `else : ...`.

Dans certains langages (C, java, javascript, etc.), il existe une forme d'instruction conditionnelle qui examine la valeur d'une variable et qui exécute le bloc d'instructions correspondant à la valeur.
Généralement, le mot-clé utilisé alors est `switch` et s'utilise ainsi :

```
switch <variable>:
case(valeur1): ... break
case(valeur2): ... break
default: ...
```

Python n'a pas implémenté cette fonctionnalité.
La raison officielle est « *You can do this easily enough with a sequence of if... elif... elif... else* ».

Exemple 2 – Écrivons un programme qui demande trois nombres a, b et c.
Considérant que ces nombres sont les coefficients de l'équation $ax^2+bx+c=0$, il affiche « l'équation n'a pas de solution » si le nombre (appelé discriminant) b^2-4ac est négatif et sinon, il affiche les solutions qui sont $\frac{-b+\sqrt{b^2-4ac}}{2a}$ et $\frac{-b-\sqrt{b^2-4ac}}{2a}$.

```
from math import sqrt
a=int(input("coefficient a : "))
b=int(input("coefficient b : "))
c=int(input("coefficient c : "))
delta=b**2-4*a*c
if delta<0 :
    print("l'équation n'a pas de solution")
else :
    x1=(-b+sqrt(delta))/(2*a)
    x2=(-b-sqrt(delta))/(2*a)
    print("Les solutions de l'équation sont {} et {}".format(x1,x2))
```

```
>>>
coefficient a : 1
coefficient b : 2
coefficient c : 1
Les solutions de l'équation sont -1.0 et -1.0
>>>
coefficient a : 2
coefficient b : 1
coefficient c : 1
l'équation n'a pas de solution
```

On remarque que le cas où $b^2-4ac=0$ conduit aux deux mêmes solutions (par exemple avec $a=1$, $b=2$ et $c=1$). On peut alors vouloir améliorer le programme en ajoutant le traitement de ce cas. C'est une bonne occasion d'utiliser le mot-clé `elif` et de montrer une séquence `if... elif... else` qui aurait pu être traduite dans un autre langage que Python, par une séquence `switch... case...`.

avec elif

```
...
if delta<0 :
    print("l'équation n'a pas de solution")
elif delta==0 :
    print("l'équation a une solution :",(-b+sqrt(delta))/(2*a))
else :
    x1=(-b+sqrt(delta))/(2*a)
    x2=(-b-sqrt(delta))/(2*a)
    print("Les solutions de l'équation sont {} et {}".format(x1,x2))
```

sans elif

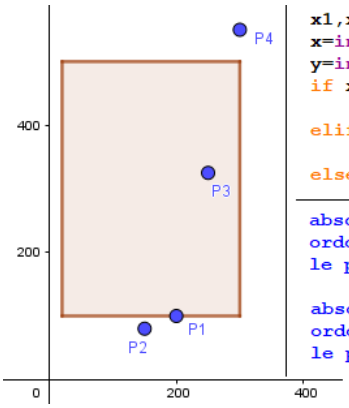
```
...
if delta<0 :
    print("l'équation n'a pas de solution")
else :
    if delta==0 :
        print("l'équation a une solution :",(-b+sqrt(delta))/(2*a))
    else :
        x1=(-b+sqrt(delta))/(2*a)
        x2=(-b-sqrt(delta))/(2*a)
        print("Les solutions de l'équation sont {} et {}".format(x1,x2))
```

```
>>>
coefficient a : 1
coefficient b : 2
coefficient c : 1
l'équation a une solution : -1.0
>>>
coefficient a : 1
coefficient b : 1
coefficient c : 1
l'équation n'a pas de solution
>>>
coefficient a : -1
coefficient b : 1
coefficient c : 1
Les solutions de l'équation sont
-0.6180339887498949
et 1.618033988749895
```

Un test ne se limite pas aux seules comparaisons `<`, `<=`, `>`, `>=`, `==` et `!=`.

On peut utiliser l'opérateur disjonctif `or` et l'opérateur conjonctif `and`, ainsi que la négation `not`, pour réaliser des tests complexes. Pour clarifier l'écriture de ce type de test, les parenthèses sont autorisées, mais il faut savoir l'ordre des priorités décroissantes [5] : comparaison, `not`, `and`, `or`.
On peut ainsi éviter les paires de parenthèses qui alourdissent l'écriture.

Exemple 3 – Écrivons un programme qui teste si un point de coordonnées (x ;y) appartient à l'extérieur du rectangle défini par les points de coordonnées (x1 ;y1), (x1 ;y2), (x2 ;y1), (x2 ;y2). On aimerait aussi distinguer les cas où le point appartient à l'intérieur strict des cas où il appartient au bord de ce rectangle. Il faut donc distinguer au moins trois cas à l'aide de combinaisons de tests.
Le moins évident des tests est celui qui identifie un point du bord : il y a deux types de bord (horizontal ou vertical) et pour chacun, une des coordonnées est fixée, l'autre doit appartenir à un intervalle.
Les parenthèses utilisées dans les tests du programme ci-dessous sont obligatoires : sans elles, la priorité de `and` sur `or` conduirait à des erreurs de logique dans certains cas.




```
x1,x2,y1,y2=20,300,100,500
x=int(input("abscisse x = "))
y=int(input("ordonnée y = "))
if x<x1 or x>x2 or y<y1 or y>y2:
    print("le point est en dehors du rectangle")
elif ((x==x1 or x==x2) and y1<=y<=y2) or ((y==y1 or y==y2) and x1<=x<=x2):
    print("le point est sur les bords du rectangle")
else : print("le point est dans le rectangle")
```

```
abscisse x = 200
ordonnée y = 100
le point est sur les bords du rectangle

abscisse x = 150
ordonnée y = 80
le point est en dehors du rectangle
```

```
abscisse x = 250
ordonnée y = 325
le point est dans le rectangle

abscisse x = 300
ordonnée y = 550
le point est en dehors du rectangle
```

On peut voir sur cet exemple que Python autorise d'écrire un test d'appartenance à un intervalle sous une forme condensée `y1<=y<=y2`, plutôt que la forme plus lourde `y1<=y and y<=y2` qui est parfois, dans d'autres langages que Python, la seule autorisée.

b. Boucles

Définition 2.3 (Boucle conditionnelle) Une boucle « tant que » – boucle *while* – permet de répéter une suite d'instructions tant qu'une condition est vraie.

Syntaxe Python de la boucle *while* :

```
while <test> :
    <instructions>
```

C'est au programmeur de prévoir la sortie de boucle, sans quoi le programme pourrait se prolonger indéfiniment, ce qui se produit avec l'instruction `while True : A=A+1` quelle que soit la valeur initiale donnée à la variable `A`. Cette boucle infinie incrémente la variable `A` mais le test n'ayant jamais la valeur `False`, le programme ne s'arrête pas.
Pour sortie d'une boucle *while*, on peut écrire un test qui, lorsqu'il est vrai, permet d'entrer dans la boucle et, lorsqu'il devient faux, permet d'en sortir. Avec l'exemple précédent, si on initialise la variable `A` à zéro, et que l'on écrit `while A<100 : A=A+1`, on crée une boucle qui s'achève puisque le test porte sur une variable qui est incrémentée dans la boucle.

```
A=0
while A<100 :
    A=A+1
print(A)
```

5. Priorités avec Python : `https://docs.python.org/3/reference/expressions.html#operator-precedence`

De cette façon, on sortira de la boucle après 100 passages (le programme affiche `100`).

Lorsqu'on peut déterminer à l'avance le nombre de passages dans une boucle, il est préférable d'écrire une boucle *for* (voir plus loin). Il faut réserver l'utilisation des boucles *while* aux algorithmes pour lesquels on ne connait pas à l'avance le nombre de tours de boucle nécessaire.

Exemple 4 – L'algorithme d'Euclide permet de déterminer le PGCD de deux nombres entiers (PGCD : Plus Grand Commun Diviseur). Le PGCD de 60 et 18, par exemple est 6 (ces deux nombres sont divisibles par 6 et ne sont pas divisibles par un nombre supérieur à 6), ce qu'on note PGCD(60 ;18)=6.
Voici une version soustractive de cet algorithme qui détermine PGCD(a ;b).

```
a=int(input("Entrer a : "))
b=int(input("Entrer b : "))
texte="PGCD({};{})=".format(a,b)
while (a!=b) :
    if a>b : a=a-b
    else : b=b-a
print(texte+str(a))
```

```
>>>
Entrer a : 60
Entrer b : 18
PGCD(60;18)=6
>>>
Entrer a : 158674923
Entrer b : 985326417
PGCD(158674923;985326417)=27
```

Dans cet exemple, comme on ne peut déterminer à l'avance le nombre de passages dans la boucle, l'utilisation d'une boucle *while* est justifiée.

Il est possible d'imbriquer des boucles à l'intérieur les unes des autres.
Signalons l'existence d'une instruction `break` (un des mot-clés de Python) qui permet de sortir de la boucle qui la contient. Le programme ci-dessous affiche `100` après être passé 100 fois dans la boucle mais cette écriture n'est pas propre. Il faut lui préférer celle qui contient un vrai test (elle a été donnée plus haut). Réservons le mot-clé `break` aux cas où il est indispensable.

```
A=0
while True : #boucle potentiellement infinie
     A=A+1
     if A>99 : break #sortie de boucle
print(A)
```

Exemple 5 – Écrivons un programme qui réalise le jeu « devine un nombre » : on demande au joueur de deviner un nombre aléatoirement choisi parmi les entiers inférieurs ou égaux à une valeur `Max` choisie à l'avance. Le joueur peut saisir plusieurs valeurs jusqu'à trouver la bonne, guidé par les indications du programme (« c'est plus ! » ou « c'est moins ! »). Le programme lui indiquera, à la fin, le nombre d'étapes utilisées et proposera de rejouer.

```
from random import randint
jeu,Max="o",1000
while jeu=="o":
    secret,essai,etape=randint(0,Max),-1,0
    while essai!=secret:
        essai=int(input("Entrez un nombre inférieur à {}: ".format(Max)))
        if(essai<secret) :
            print("C'est plus, recommencez!")
        elif(essai>secret) :
            print("C'est moins, recommencez!")
        etape+=1
    print("Bravo! Vous avez trouvé après ",etape," étapes.")
    jeu=input("Voulez-vous rejouer (oui:o, non:n)? ")
```

```
>>>
Entrez un nombre inférieur à 1000: 500
C'est plus, recommencez!
Entrez un nombre inférieur à 1000: 700
C'est plus, recommencez!
Entrez un nombre inférieur à 1000: 850
C'est moins, recommencez!
Entrez un nombre inférieur à 1000: 799
C'est plus, recommencez!
Entrez un nombre inférieur à 1000: 820
Bravo! Vous avez trouvé après  5  étapes.
Voulez-vous rejouer (oui:o, non:n)? n
```

J'ai utilisé ici un test qui compare les deux valeurs (la valeur `secret` qu'il faut découvrir et la valeur `essai` que le joueur propose). Pour entrer dans la boucle au début, j'initialise `essai` à `-1`, une valeur qui ne peut être égale à `secret` puisque ce nombre est supérieur ou égal à 0. On reste ainsi dans la boucle intérieure tant que les deux valeurs sont différentes. Quand elles sont égales, on sort de la boucle intérieure, on affiche le message de félicitation et on demande au joueur s'il veut continuer. S'il entre le caractère `o`, on rentre à nouveau dans la boucle intérieure, sinon on sort de la boucle extérieure et le programme s'arrête.

Ajoutons une fonctionnalité à ce jeu qui autorise le joueur à s'arrêter en cours de route : il lui suffira de taper 0 pour arrêter (0 n'est plus une solution possible). Pour réaliser cela, j'ai introduit un `break` conditionnel dans la boucle intérieure. Pour sortir de la boucle extérieure, il faut un 2e `break`.
J'ai modifié les affichages pour qu'ils rendent compte de cette nouvelle dynamique de jeu.

```
from random import randint
jeu,Max="o",1000
total_etapes,nbr_jeux=0,0
while jeu=="o":
    nbr_jeux+=1
    secret,essai,etape=randint(1,Max),-1,0
    print("Un nombre entre 1 et {} a été choisi".format(Max))
    print("(entrer 0 arrête le jeu)")
    while essai!=secret:
        essai=int(input("Entrez un nombre : "))
        if essai==0 : break
        if(essai<secret) :
            print("C'est plus, recommencez!")
        elif(essai>secret) :
            print("C'est moins, recommencez!")
        etape+=1
    if essai==0 : break
    else :
        print("Bravo! Vous avez trouvé après ",etape," étapes.")
        total_etapes+=etape
    jeu=input("Voulez-vous rejouer (oui:o, non:n)? ")
if essai==0 : print("Partie interrompue. A bientôt ?")
else :
    print("Vous avez joué {} fois. Merci !".format(nbr_jeux))
    print("Score moyen : {}".format(total_etapes/nbr_jeux))
```

```
...
Entrez un nombre : 822
Bravo! Vous avez trouvé après  11  étapes.
Voulez-vous rejouer (oui:o, non:n)? o
Un nombre entre 1 et 1000 a été choisi
(entrer 0 arrête le jeu)
Entrez un nombre : 550
C'est moins, recommencez!
Entrez un nombre : 420
C'est moins, recommencez!
Entrez un nombre : 310
C'est moins, recommencez!
Entrez un nombre : 250
C'est plus, recommencez!
Entrez un nombre : 280
C'est plus, recommencez!
Entrez un nombre : 299
C'est plus, recommencez!
Entrez un nombre : 305
C'est moins, recommencez!
Entrez un nombre : 302
Bravo! Vous avez trouvé après  8  étapes.
Voulez-vous rejouer (oui:o, non:n)? n
Vous avez joué 4 fois. Merci !
Score moyen : 10.0
```

Notons qu'il est possible de court-circuiter la partie finale du bloc d'instructions d'une boucle *while*. Il faut pour cela utiliser le mot-clé `continue`. Contrairement à `break` qui déclenche une sortie de boucle, ce mot-clé précipite le début du tour de boucle suivant en provoquant un saut vers la condition testée pour rester dans la boucle.
Un exemple d'application de `continue` est donné un peu plus loin.
Ces mot-clés `break` et `continue` doivent rester exceptionnels : on s'en passe tant qu'on le peut.

DÉFINITION 2.4 (BOUCLE BORNÉE) Une boucle « pour » – boucle *for* – permet de répéter une suite d'instructions un nombre prédéfini de fois.

La syntaxe Python d'une boucle *for* utilise une variable locale `i` (sa valeur n'est pas connue en dehors de la boucle) et une variable `n` qui a été initialisée avant d'entrer dans la boucle.
Les noms des variables `i` et `n` peuvent, bien sûr, être changés :

```
for i in range(n) :
    <instructions>
```

La variable `i` prendra alors `n` valeurs (de 0 à `n-1`) et les instructions seront donc exécutées `n` fois.
Il est possible et parfois très utile d'utiliser, dans ces instructions, la valeur de la variable `i` qui compte les tours (quand `i=0` on est dans le 1er tour, quand `i=1` on est dans le 2e tour, etc.)

La fonction `range(<debut>,<fin>,<pas>)` est un *built-in* Python qui génère une séquence de nombres allant de `debut` (inclus) à `fin` (exclu) avec un incrément régulier égal à `pas`.
Cette séquence comporte `(fin-debut)` nombres quand `pas=1`.
Certains des arguments de `range` sont optionnels :

- Écrire `range(n)` équivaut à écrire `range(0,n,1)` (par défaut : `debut=0`, `pas=1`).
- Si on veut la séquence allant de 1 à `n`, il faut taper `range(1,n+1)`
- Si on veut les nombres allant de `n` à 0, il faut taper `range(n,-1,-1)`
- Si tape `range(1,15,3)`, la séquence contient les nombres 1, 4, 7, 10 et 13.

De la même façon qu'on peut imbriquer des boucles *while* à l'intérieur les unes des autres, on peut imbriquer des boucles *for*. Toutes les combinaisons sont d'ailleurs possibles : il suffit de respecter l'indentation.
Seule l'indentation permet de savoir exactement à quel bloc appartient telle ou telle instruction.

Exemple 6 – Écrivons un programme qui tire au hasard `n=100` couples de points au hasard dans un carré de côté 1 et qui détermine la moyenne des distances entre les points de chaque couple. Les abscisses x et ordonnées y sont des nombres aléatoires compris entre 0 (inclus) et 1 (exclu), ce que produit la fonction `random` du module `random` ; la distance entre les points de coordonnées $(x1; y1)$ et $(x2; y2)$ est donnée par la formule issue du théorème de Pythagore $d = \sqrt{(x1-x2)^2 + (y1-y2)^2}$.

```
from random import random
from math import sqrt
N=1000
Total=0
for I in range(N):
    distance=0
    for J in range(2):
        P=random()
        Q=random()
        distance+=(P-Q)**2
    distance=sqrt(distance)
    Total+=distance
print("Moyenne des distances = {} ".format(Total/N))
```

```
>>>
Moyenne des distances = 0.5283094807432922
```

Pour simplifier la programmation de cet algorithme, j'ai réalisé

- les tirages des deux abscisses d'abord (`(P,Q)` correspond à $(x1; x2)$ quand `J` vaut 0)
- les tirages des deux ordonnées ensuite (`(P,Q)` correspond à $(y1; y2)$ quand `J` vaut 1)

Vérifiez que la distance entre les deux points est bien calculée selon la formule rappelée ci-dessus, et en particulier, observez que ce n'est qu'en sortant de la boucle intérieure (par réduction de l'indentation) qu'on prend la racine carrée de la variable `distance`.
Le résultat est une estimation de la véritable valeur de cette distance moyenne qui vaut :

$$\frac{2+\sqrt{2}}{15} + \frac{\ln(1+\sqrt{2})}{3} \approx 0,521405433$$

Exemple 7 – Construisons un solutionneur pour ce problème inspiré par Sam Loyd[6] : déterminer la valeur des chiffres a, b, c, d, e, f, g, h, i et j pour que la somme $abcde + fghij$ fasse 181341. Bien sûr, les nombres sont en base dix dans cet énoncé et on suppose que les dix lettres désignent des chiffres différents donc les dix chiffres de la base dix.
La programmation de ce problème peut rester rudimentaire : j'utilise ci-dessous des boucles *for* imbriquées avec des mot-clés `continue` pour éviter de traiter les cas où deux lettres différentes représenteraient le même chiffre. On peut optimiser cette recherche, ne serait-ce qu'en limitant a et f aux valeurs 8 et 9 mais ce serait dommage de perdre la généralité de ce programme qui s'applique à d'autres nombres que 181341 : l'essayer avec `n=22221` (384 solutions) ou avec `n=22770` (16 solutions).

```
n=181341
solutions=0
for a in range(10) :
    for b in range(10)  :
        if b==a : continue
        for c in range(10)  :
            if c==b or c==a : continue
            for d in range(10)  :
                if d==c or d==b or d==a : continue
                for e in range(10)  :
                    if e==d or e==c or e==b or e==a : continue
                    for f in range(10)  :
                        if f==e or f==d or f==c or f==b or f==a : continue
                        for g in range(10)  :
                            if g==f or g==e or g==d or g==c or g==b or g==a : continue
                            for h in range(10)  :
                                if h==g or h==f or h==e or h==d or h==c or h==b or h==a : continue
                                for i in range(10)  :
                                    if i==h or i==g or i==f or i==e or i==d or i==c or i==b or i==a : continue
                                    j=45-(a+b+c+d+e+f+g+h+i)
                                    num1=a*10**4+b*10**3+c*10**2+d*10**1+e
                                    num2=f*10**4+g*10**3+h*10**2+i*10+j
                                    if num1+num2==n and num1<num2 :
                                        solutions+=1
                                        print("solution {} : {} + {}= {} ".format(solutions,num1,num2,n))
if solutions==0 : print("pas de solution")
```

```
>>>
solution 1 : 84015 + 97326= 181341
solution 2 : 84016 + 97325= 181341
solution 3 : 84025 + 97316= 181341
solution 4 : 84026 + 97315= 181341
solution 5 : 84105 + 97236= 181341
 ...
solution 60 : 87236 + 94105= 181341
solution 61 : 87315 + 94026= 181341
solution 62 : 87316 + 94025= 181341
solution 63 : 87325 + 94016= 181341
solution 64 : 87326 + 94015= 181341
```

Je trouve ainsi 64 solutions différentes au problème.

6. Voir le problème « The missing number » dans le livre Cyclopedia of Puzzles de Sam Loyd (1841-1911) à l'adresse `http://cyclopediaofpuzzles.com/pages/94`

3. Types de données construits

DÉFINITION 2.5 (P-UPLET) Un p-uplet – « tuple » en Python – est une suite de valeurs séparées par des virgules et délimitées par des parenthèses (ce délimiteur est facultatif).

La déclaration du *tuple* vide `t1` s'effectue en écrivant `t1=()` (ce genre de déclaration est utile quand on doit déclarer le tuple alors qu'on ne connait pas les valeurs à y mettre, par exemple avant une boucle où elles sont définies).

Le tuple `t2` déclaré par l'instruction `t2=(-1,1)` contient deux valeurs.
On accède en lecture à la 1re valeur du tuple `t2` en écrivant `t2[0]`.
L'instruction `print(t2[0])`, par exemple, affiche `-1`.
De même, la 2e valeur de `t2` étant `t2[1]`, je peux l'affecter à une variable `b=t2[1]`.
On peut parcourir la liste des indices à l'envers, `t[-1]` retourne le dernier élément du tuple `t`.

On ne peut pas modifier les valeurs d'un tuple, c'est une liste non-mutable. Par contre, on peut réaffecter un tuple : je peux écrire par exemple `t1=(0,1)` alors que `t1` était le tuple vide. Mais l'instruction `t2[0]=0` conduirait à l'erreur :

```
'tuple' object does not support item assignment
```

On peut omettre les parenthèses qui sont facultatives.
Le tuple `t3` déclaré par l'instruction `t3=-1,0,1` est tout-à-fait valide.
Il contient trois valeurs qui peuvent être identifiées par les notations `t3[0]`, `t3[1]` et `t3[2]`.

Si un tuple contient n valeurs, on peut affecter n variables en les disposant comme dans un tuple.
On peut par exemple, affecter trois variables avec le tuple `t3` précédent, en écrivant `a,b,c=t3`.
Dans ce cas, `a` contient `t3[0]`, `b` contient `t3[1]` et `c` contient `t3[2]`.
Cette particularité a déjà été utilisée dans les programmes de ce chapitre :
Quand j'ai écrit `x1,x2,y1,y2=20,300,100,500` pour initialiser quatre variables sur une seule ligne (exercice 3), il s'agissait techniquement d'un tuple de variables initialisé avec un tuple de valeurs.

Cette technique est à retenir car elle fait gagner du temps.
Supposez que vous vouliez échanger le contenu de deux variables `a` et `b` :

- En procédant sans tuple, il faut utiliser une variable tampon `c=a`, puis `a=b` et enfin `b=c`.
- Avec un tuple, il suffit d'écrire `a,b=b,a`.

C'est mieux, non ?

On retiendra aussi la structure de tuple quand on veut enregistrer les données variées concernant un objet, constituer une sorte de base de données. Supposons que l'on crée une sorte de carnet d'adresse sous la forme d'une liste (voir plus loin) d'enregistrements où l'enregistrement correspondant à une personne contient : son nom, son prénom, son téléphone, son adresse mail. L'utilisation d'un tuple `registre` pour les enregistrements permet de récupérer chacune des informations avec l'instruction `nom,prenom,tel,mail=registre`.

Certaines méthodes peuvent être utilisées avec des tuples :

- On peut connaître le nombre d'éléments d'un tuple `t` à l'aide de la fonction `len`.
  En tapant `len(t3)` par exemple, j'obtiendrai `3` puisque `t3` contient trois éléments.
- On peut savoir si un élément est présent dans un tuple à l'aide du mot-clé `in`.
  L'instruction `-1 in t3` par exemple retourne `True` alors que `"-1" in t3` retourne `False`

Une chaîne de caractères présente des similarités avec un tuple : on peut accéder à n'importe lequel des caractères en désignant son rang.
Si la variable `s` contient la chaîne `"Mr X"` alors `s[0]` contient `"M"`, `s[2]` contient `" "`, etc.
On ne peut pas modifier un caractère d'une chaîne directement, le type *string* étant non mutable :
L'affectation `s[1]="e"` provoque l'erreur :

```
'str' object does not support item assignment
```

Définition 2.6 (Liste) Un tableau – « liste » en Python – est une suite de valeurs séparées par des virgules et délimitées par des crochets.

La déclaration de la liste vide `L1` s'effectue en écrivant `L1=[]` ou `L1=list()`.

La liste `L2` déclarée par l'instruction `L2=[-1,1]` contient deux valeurs.
Comme pour un tuple, on accède à la valeur `i` de la liste `L` en écrivant `L[i]`, mais, contrairement au tuple, cet accès est autorisé en écriture : l'instruction `L2[0]=0`, par exemple, est tout-à-fait licite.

On peut aussi déclarer et initialiser une liste contenant `n` éléments égaux à 0 en écrivant `L=n*[0]`.
L'instruction `L=5*[0]` est donc équivalente à `L=[0,0,0,0,0]`.

La fonction `range()` permet d'initialiser une liste : `L=list(range(1,10))` conduit à la liste `L=[1,2,3,4,5,6,7,8,9]`. On peut aussi écrire `L=[n for n in range(1,10)]`.

Pour cette dernière construction de liste, dite « en compréhension », la variable `n` est muette. N'étant définie qu'entre les crochets de la définition, ce nom de variable (ici la lettre `n`) n'a pas d'importance.
La syntaxe de la construction en compréhension permet de faire un tri dans les valeurs sélectionnées et permet également d'appliquer une fonction à cette variable muette. On n'est d'ailleurs pas obligé d'utiliser `range`, les valeurs pouvant être sélectionnées dans une autre liste.
Voici quelques exemples de constructions en compréhension de la liste `L` :

- `L=[2*n for n in range(1,10)]` construit la liste `L=[2,4,6,8,10,12,14,16,18]`
- `L=[n**2 for n in range(1,10)]` construit la liste `L=[1,4,9,16,25,36,49,64,81]`
- `L=[-n for n in range(1,10) if n%2==0]` construit la liste `L=[-2,-4,-6,-8]`
- `L=[n for n in range(1,10) if n>3 and n%3==0]` construit la liste `L=[6,9]`
- `L=["m"+c for c in ["a","i","u","e","o"]]` donne `L=["ma","mi","mu","me","mo"]`
- avec `T=[1,2,3,4,5]` et `L=[n-5 for n in T]` on obtient `L=[-4,-3,-2,-1,0]`

De nombreuses méthodes peuvent être utilisées avec les listes :

- Comme pour un tuple, le nombre d'éléments d'une liste `L` est donné par `len(L)`.
  Les indices d'une liste (ou d'un tuple) vont donc de `0` à `len(L)-1`.
- Le mot-clé `in` permet de tester, comme pour un tuple, la présence d'un élément dans une liste : `e in L` retourne `True` ou `False` selon les cas.
  L'instruction `L.count(e)` permet de connaître le nombre d'occurrences d'un élément `e`.
- On peut accéder au 1er rang d'un élément `e` présent dans une liste avec `L.index(e)`.
  Attention : si il n'y a pas l'élément cette instruction provoque une erreur !
- On peut extraire une liste `T` à partir d'une liste `L` en précisant les rangs `debut` et `fin` en écrivant `T=L[debut:fin]`. Si `debut` est omis `debut=0` et si `fin` est omis `fin=len(L)`.
  Avec `L=[12,5,7]`, `L[1:2]` retourne `[5]`, `L[:2]` retourne `[12,5]` et `L[1:]` retourne `[5,7]`.
- On peut concaténer deux listes `L1` et `L2` en écrivant `L=L1+L2` : `L` contient les éléments des deux listes. On peut également dupliquer une liste en écrivant `L=L*n` où `n` est un entier.
  Avec `L=[12,5,7]`, `L*2` ou `L+L` retourne `[12,5,7,12,5,7]` et `L*0` retourne `[]`.
- La méthode `sort()` ordonne les éléments d'une liste.
  Avec `L=[12,5,7]`, l'instruction `L.sort()` ne retourne rien mais la liste a été ordonnée : il s'agit désormais de `[5,7,12]`. L'instruction `sorted(L)` par contre ne modifie pas la liste `L` : elle retourne une liste ordonnée, construite sur les éléments de `L`, mais indépendante.

Des indices négatifs peuvent être employés :
`L[-1]` est le dernier élément de la liste `L` et `L[-len(L)]` en est le premier.
Les valeurs admises pour les indices d'une liste `L` vont donc de `-len(L)` à `len(L)-1` :
en parcourant cette séquence, on obtient deux fois chaque élément de la liste.

Le choix d'un élément aléatoire dans une liste est une possibilité offerte par la fonction `choice` du module `random`. Après avoir importé la fonction avec `from random import choice`, il suffit d'écrire `choice(L)` pour obtenir un élément tiré au hasard dans la liste `L`.
`choice(["pierre","feuille","ciseaux"])` renvoie aléatoirement un de ces trois éléments.
L'instruction `shuffle(L)` issue du même module, produit un mélange aléatoire de `L`.

Les listes sont mutables, on peut en changer chaque élément. On peut aussi ajouter ou supprimer des éléments d'une liste, par concaténation ou par duplication de listes, mais pas seulement :

- On peut ajouter un élément `e` en dernière position dans une liste `L` en écrivant `L.append(e)`. Avec `L=[]`, l'instruction `L.append("a")` conduit à avoir `L=["a"]`.
  Si on écrit encore `L.append("b")`, on aura `L=["a","b"]`.
- La méthode `insert(i,e)` insère l'élément `e` au rang `i`, en décalant tous les indices suivants. Avec `L=[1,2,3]`, l'instruction `L.insert(2,4)` insère `4` au rang `2` ce qui donne `L=[1,2,4,3]`.
- Pour supprimer le dernier élément d'une liste `L`, écrire `L.pop()`. Cette instruction renvoie l'élément supprimé qui peut alors être utilisé. `L.pop(i)` supprime et renvoie `L[i]`.
- On peut aussi, avec `L.remove(e)`, enlever la 1re occurrence de l'élément `e` de la liste `L`. L'instruction `del L[i]` supprime `L[i]` sans le renvoyer, l'élément de rang `i` de la liste `L`.

Attention : si on affecte une liste à une variable, on ne crée pas une 2e liste indépendante de la 1re. Si on a `L=[1,2,3]` et qu'on fait `T=L`, la liste `T` contient bien `[1,2,3]` comme `L`, mais il s'agit en fait de deux noms (deux adresses mémoires) pour un seul et même objet. Modifier l'une modifie également l'autre : après `T[1]=5`, les deux listes contiennent `[1,5,3]`.
Pour faire une copie indépendante d'une liste, on peut procéder de plusieurs façons :

- `T=L[:]`
- `T=[e for e in L]`
- `T=[]`, puis `for e in L: T.append(e)`
- `T=[0]*len(L)`, puis `for i in range(len(L)): T[i]=L[i]`

Ce phénomène de fausse copie se produit également lorsqu'on transmet une liste à une fonction ou lorsqu'une fonction renvoie une liste après l'avoir modifiée : ce sont les adresses qui sont transmises et non les valeurs. Nous reparlerons de cela plus loin.

On peut transformer une liste `L` ne contenant que des éléments de type `string` en texte avec la méthode `join(L)` appliquée à un caractère de séparation :
Avec `L=["Bonjour,","comment","vas","tu","?"]`, l'instruction `print(" ".join(L))` conduit à l'affichage de `Bonjour, comment vas tu ?`
Inversement, on peut créer une liste à partir d'une chaîne de caractères en utilisant la fonction `list()`. L'instruction `L=list("abcdef")` conduit à la liste `["a","b","c","d","e","f"]`.

Exemple 8 – Simulons une collection d'objets choisis au hasard parmi `n` et numérotés de 1 à `n` :
On souhaite obtenir le nombre total `t` d'objets à tirer pour obtenir la collection complète ainsi que le nombre d'occurrences de chaque objet.
Pour fixer les idées, tirons un dé à `n=6` faces jusqu'à avoir obtenu les 6 chiffres possibles, au moins une fois chacun. Le programme affichera la liste `L` contenant le nombre d'occurrences de chaque objet.

```
from random import randint
n=6 #nombre d'objets dans la collection
t=0 #nombre de tirages effectués
L0,L1,L2,L3=[],n*[0],n*[0],n*[0]
while L1.count(0)>0 :
    t+=1
    r=randint(0,n-1)
    L0.append(r)          # L0 contient tous les tirages, dans l'ordre
    if L1[r]==0 : L1[r]=t # L1 contient le rang de la premiere occurence
    L2[r]=t               # L2 contient le rang de la derniere occurence
    L3[r]+=1              # L3 contient le nombre d'occurences
print("{} objets obtenus en {} tirages".format(n,t))
print("|objets|premier|dernier|nombre|")
for i in range(n):
    print("|{:6}|{:7}|{:7}|{:6}|".format(i+1,L1[i],L2[i],L3[i]))
print("Liste des tirages:",[i+1 for i in L0])
```

```
>>>
6 objets obtenus en 16 tirages
|objets|premier|dernier|nombre|
|     1|      1|     10|     1|
|     2|      2|     14|     3|
|     3|      7|     15|     5|
|     4|      1|     13|     5|
|     5|      9|      9|     1|
|     6|     16|     16|     1|
Liste des tirages: [4, 2, 2, 4, 4,
  4, 3, 3, 5, 1, 3, 3, 4, 2, 3, 6]
>>> L0
[3, 1, 1, 3, 3, 3, 2, 2, 4, 0, 2, 2,
                              3, 1, 2, 5]
>>> L1
[10, 2, 7, 1, 9, 16]
```

Ce programme nécessite juste une liste (la liste `L3`) ; j'ai ajouté les listes `L1` et `L2` pour enregistrer, respectivement : les rangs de la 1re et de la dernière occurrence d'un objet. Et j'ai aussi ajoutée la liste `L0` pour enregistrer les objets tirés dans l'ordre où ils ont été obtenus.
J'ai voulu soigner l'affichage en présentant les résultats dans un tableau :
Pour cela, l'option de la méthode `format` utilisée pour insérer la valeur des variables dans la chaîne de caractères (syntaxe `{:large}`) permet d'avoir une zone de largeur constante.
Remarquer que l'affichage du numéro d'un objet est `i+1` quand l'indice de l'objet est `i` (deux dernières lignes du programme). J'ai affiché `L0` et `L1` pour mieux voir cela.

Définition 2.7 (Liste de listes) Un tableau de `l` lignes de `c` colonnes est réalisé en Python en définissant une liste de `l` éléments, chaque élément de cette liste étant une liste de `c` éléments.

Exemple `L=[[1,2,3,4,5],[6,7,8,9,10]]` est un tableau de 2 lignes de 5 colonnes.
Si je tape `L[0]`, j'obtiens la liste `[1,2,3,4,5]` tandis que `L[1]` contient la liste `[6,7,8,9,10]`.
Pour obtenir le 1er élément de `L[0]`, je dois taper `L[0][0]` et, pour obtenir la valeur `10`, le dernier élément de la dernière ligne, je dois taper `L[1][4]`.

Le nombre de ligne peut être obtenu avec `len(L)` et le nombre de colonnes avec `len(L[0])` (ou `len(L[1])` puisque les lignes ont toutes le même nombre de colonnes).
De cette façon, à la place de `L[1][4]` pour obtenir le dernier élément de la dernière ligne, je peux taper `L[len(L)-1][len(L[1])-1]` (c'est moins facile à écrire mais d'utilisation plus générale).

Remarque : cette fonctionnalité n'est pas propre aux tableaux de `l` lignes de `c` colonnes : comme on peut mettre n'importe quel type de donnée dans une liste, on peut y mettre des listes. Les objets d'une liste ne sont pas forcément homogènes. La particularité examinée ici permet de fabriquer un objet utile autant en mathématiques qu'en informatique : une matrice. J'ai montré un exemple de matrice de dimension deux (lignes et colonnes), mais on peut manipuler des listes en dimension trois (lignes, colonnes et plans) ou davantage.
Ce sera moins fréquent cette année, mais voici la déclaration d'une liste contenant deux plans de trois lignes de trois colonnes : `L=[[[1,2,3],[4,5,6],[7,8,9]],[[9,8,7],[6,5,4],[3,2,1]]`

Pour initialiser un tableau de `l` lignes de `c` colonnes ne contenant que des zéros, on peut utiliser la construction suivante : `L=[[0]*c for i in range(l)]` (la variable muette `i` n'est pas utilisée dans ce contexte). En effet `[0]*c` crée une ligne de `c` zéros et, la séquence `range(l)` contenant `l` valeurs, on crée ainsi `l` lignes de `c` zéros.
Ne pas réaliser cette initialisation en tapant `L=[[0]*c]*l` car ainsi, comme nous l'avons souligné précédemment, nous dupliquerions `l` fois *la même* ligne. Du coup, le tableau obtenu n'aurait pas le comportement souhaité (en changeant la valeur d'une case, on change simultanément les valeurs de toutes les lignes pour la même colonne que cette case).

Exemple 9 – En guise d'exemple, construisons un tableau à deux dimensions contenant les tables de multiplication par $2, 3, 4, \cdots, 9$ des nombres allant de `n1` à `n2` (avec `0<n1<n2`).
La valeur contenue dans `L[l][c]` donne le produit de `l+n1` par `c+2`.

```
n1=int(input("premier nombre : "))
n2=int(input("premier nombre : "))
L=[[i*j for j in range(2,10)] for i in range(n1,n2+1)]
print(L)                        # affichage brut
print("")                       # affichage lisible
for i in range(len(L[0])): print("  ",i+2,end="")#ligne en-tête
print("")
for i in range(len(L)): print(i+n1,L[i])          #lignes
print("")                       # affichage soigné
print("|\u00D7| 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |")#ligne entête
print("-"*(9*4+1))
for i in range(n2-n1+1):
    print("|{:3}|".format(i+n1),end="")           #en-tête ligne
    for j in range(10-2):
        print("{:3}|".format(L[i][j]),end="")     #cellule ligne
    print("")
```

```
>>>
premier nombre : 11
premier nombre : 15
[[22, 33, 44, 55, 66, 77, 88, 99], [24, 36, 48
, 60, 72, 84, 96, 108], [26, 39, 52, 65, 78, 9
1, 104, 117], [28, 42, 56, 70, 84, 98, 112, 12
6], [30, 45, 60, 75, 90, 105, 120, 135]]

   2   3   4   5   6   7   8   9
11 [22, 33, 44, 55, 66, 77, 88, 99]
12 [24, 36, 48, 60, 72, 84, 96, 108]
13 [26, 39, 52, 65, 78, 91, 104, 117]
14 [28, 42, 56, 70, 84, 98, 112, 126]
15 [30, 45, 60, 75, 90, 105, 120, 135]

| × | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
-------------------------------------
| 11| 22| 33| 44| 55| 66| 77| 88| 99|
| 12| 24| 36| 48| 60| 72| 84| 96|108|
| 13| 26| 39| 52| 65| 78| 91|104|117|
| 14| 28| 42| 56| 70| 84| 98|112|126|
| 15| 30| 45| 60| 75| 90|105|120|135|
```

J'ai donné le tableau demandé sous trois formes :

- sa forme brute, peu lisible pour un humain.
- sous une forme plus lisible, en séparant les lignes.
- sous la forme la plus appropriée pour une utilisation éventuelle : avec les nombres concernés dans une légende pour les lignes et les colonnes.

DÉFINITION 2.8 (DICTIONNAIRES) Un dictionnaire en Python est une suite de paires (clé : valeurs) séparées par des virgules et délimitées par des accolades.

Un dictionnaire est une sorte de tableau associatif où les clés ne sont, à priori, pas des indices comme dans les tableaux, mais des chaînes de caractères.

Exemple :
`Dic={"Nom":"Modiano", "Prénom":"Patrick", "Profession":"écrivain", "Promo":"1962"}`.
Pour accéder au nom `Modiano`, il faut taper `Dic["Nom"]`.
Pour le prénom `Patrick`, il faut taper `Dic["Prénom"]`.

Un dictionnaire peut être déclarée vide : `Dic={}`.
Les valeurs peuvent être entrées dans n'importe quel ordre, par exemple `Dic["Prénom"]="Patrick"`, `Dic["Nom"]="Modiano"`, `Dic["Promo"]="1962"`, etc. Une fois qu'un dictionnaire est constitué, on peut en modifier une valeur de la même façon (ex : `Dic["Promo"]="1961"`) ou supprimer une entrée avec le mot-clé `del` (ex : `del Dic["Promo"]`).
On peut connaître le nombre de clés d'un dictionnaire avec la fonction `len` (`len(Dic)` renvoie ici `4`).

Un dictionnaire peut contenir des clés numériques ainsi que des valeurs numériques.
Dans ce cas, les guillemets sont inutiles.
Exemple `dé={1:1, 2:3, 3:5, 4:5, 5:1, 6:1}` (j'ai repris les résultats de l'exercice 8 où la clé correspond au numéro de la face du dé et la valeur au nombre d'occurrences de la face lors du tirage).
J'obtiens le nombre d'occurrences de la face 2 en écrivant `dé[2]` (réponse : `3`).
Les valeurs numériques ne sont pas forcément entières :
`exp={1.1:1, 1.2:5, 1.3:10, 1.4:5, 1.5:3, 1.6:1, 1.7:1}`
`exp` pourrait être un dictionnaire donnant les statistiques d'une expérience.
Toutes les combinaisons nombres ou chaînes de caractères sont possibles pour un dictionnaire (`livres={"romans":5, "scolaires":12, "magazines":3}` ou `bond={1962:"S.Connery", 1969:"G.Lazenby", 1973:"R.Moore", 1987:"T.Dalton", 1995:"P.Brosnan", 2006:"D.Craig"}`).

L'ensemble des clés est obtenu avec la méthode `keys()`
(`Dic.keys()` renvoie pour le 1er exemple
`dict_keys(['Profession', 'Promo', 'Prénom', 'Nom'])`).
L'ensemble des valeurs est obtenu avec la méthode `values()`
(`Dic.values()` renvoie ici `dict_values(['écrivain', '1962', 'Patrick', 'Modiano'])`).
De même, on peut accéder à tous les contenus sous la forme (clés, valeurs) avec la méthode `item()`
(`Dic.items()` renvoie ici `dict_items([('Profession', 'écrivain'), ...])`).
Mais c'est plus utile d'obtenir de vraies listes :

- pour la liste des clés taper `list(Dic.keys())` ou plus simplement `list(Dic)`
- pour la liste des valeurs taper `list(Dic.values())`
- pour la liste des contenus sous la forme (clés, valeurs) taper `list(Dic.items())`

On peut tester si un dictionnaire contient une clé avec le mot-clé `in` (`"Nom" in Dic` renvoie `True`).
Pour tester si un dictionnaire contient une valeur, il faut utiliser `in Dic.values()`
(`'écrivain' in Dic.values()` renvoie `True` alors que `'écrivain' in Dic` renvoie `False`).

On parcourt toutes les clés d'un dictionnaire `Dic` avec la syntaxe `for x in Dic :`
Cela place dans la variable `x` le nom de la clé.
Il ne reste plus qu'à accéder aux valeurs, avec la syntaxe `Dic[x]`.

EXEMPLE 10 – Cherchons à déterminer la fréquence des lettres d'un texte quelconque.
S'il n'y a aucun signe de ponctuation, aucun accent et aucune majuscule, même pas d'apostrophe, juste des espaces entre les mots, c'est facile. Appelons ce genre de texte un « texte simple » et implémentons ce cas. Dans un 2e temps on convertira un texte ordinaire en texte simple.
J'utilise un dictionnaire pour enregistrer les occurrences d'une lettre : si la lettre n'est pas encore dans le dictionnaire, on l'ajoute en initialisant le compteur d'occurrences à 1, sinon on incrémente ce compteur.

Pour tester mon programme, je choisis le 1er paragraphe de Alice's Adventures in Wonderland de Lewis Carroll, expurgé de sa ponctuation : « Alice was beginning to get very tired of sitting by her sister on the bank and of having nothing to do once or twice she had peeped into the book her sister was reading but it had no pictures or conversations in it and what is the use of a book thought Alice without pictures or conversation ».

version non triée

```
texte="alice was beginning to get very tired of sitting...
lettres={}
for c in texte:
    if c==" " : continue
    if c in lettres:
        lettres[c]+=1
    else:
        lettres[c]=1
for c in lettres:
    print("Nombre de {} : {}".format(c,lettres[c]))
```

```
>>>
Nombre de h : 14
Nombre de e : 25
Nombre de d : 8
Nombre de g : 8
Nombre de a : 15
Nombre de y : 2
Nombre de l : 2
Nombre de f : 3
Nombre de r : 14
Nombre de v : 4
Nombre de b : 6
Nombre de c : 8
Nombre de t : 26
Nombre de p : 4
Nombre de w : 5
Nombre de i : 23
Nombre de u : 6
Nombre de s : 15
Nombre de n : 20
Nombre de o : 24
Nombre de k : 3
```

version triée

```
alphabet="abcdefghijklmnopqrstuvwxyz"
for c in alphabet:
    if c in lettres:
        print("Nombre de {} : {}".format(c,lettres[c]))
```

```
>>>
Nombre de a : 15
Nombre de b : 6
Nombre de c : 8
Nombre de d : 8
Nombre de e : 25
Nombre de f : 3
Nombre de g : 8
Nombre de h : 14
Nombre de i : 23
Nombre de k : 3
Nombre de l : 2
Nombre de n : 20
Nombre de o : 24
Nombre de p : 4
Nombre de r : 14
Nombre de s : 15
Nombre de t : 26
Nombre de u : 6
Nombre de v : 4
Nombre de w : 5
Nombre de y : 2
```

L'inconvénient majeur de ce programme, outre le fait qu'il ne traite que des textes simples, est le tri alphabétique qui n'est pas effectué.
Pour obtenir l'affichage des lettres dans cet ordre qui nous est familier, il suffit d'ajouter une boucle qui examine les clés dans le bon ordre.
Pour traiter un texte ordinaire :

- je commence par enlever les majuscules avec la méthode de la classe *string* `lower()`
- ensuite, j'ai construit une liste des signes de ponctuation usuels (sans doute incomplète) qui vont tout simplement être ignorés (ne seront pas enregistrés ni comptés dans le total)
- pour les accents, je crée un dictionnaire répertoriant toutes les lettres accentuées (en français, sans doute incomplet) et qui permet de comptabiliser les lettres accentuées avec celles qui n'ont pas d'accent.

J'ai ajouté le calcul et l'affichage de la fréquence des lettres, ce qui permet de comparer les textes.
Pour tester mon programme, j'ai pris la traduction française du 1er paragraphe de Alice au Pays des Merveilles : « Alice, assise auprès de sa sœur sur le gazon, commençait à s'ennuyer de rester là à ne rien faire ; une ou deux fois elle avait jeté les yeux sur le livre que lisait sa sœur ; mais quoi ! pas d'images, pas de dialogues ! « La belle avance, » pensait Alice, « qu'un livre sans images, sans causeries ! » ».

```
texte="Alice, assise auprès de sa sœur sur le gazon, commençait à s'ennuyer ...
accents={"é":"e","è":"e","ë":"e","ê":"e","à":"a","â":"a","ü":"u","û":"u","ô":"o"}
ponctuation=[" ","!","?",".",";",",","'",":","\"","'","-","«","»"]
alphabet="abcdefghijklmnopqrstuvwxyz"
texte=texte.lower()       # pour transformer les majuscules en minuscules
lettres={}
total=0
for c in texte :
    if c in ponctuation :# pour ne pas tenir compte de la ponctuation
        continue
    if c in accents :     # pour enlever les accents
        c=accents[c]
    if c=="œ" :           # pour comptabiliser le œ par o et e
        c="o"
        texte+="e"
    if c in lettres:      # pour ajouter une occurrence sur une clé
        lettres[c]+=1
    else:                 # pour débuter une nouvelle clé (lettre)
        lettres[c]=1
    total+=1
for c in alphabet:        # pour afficher les résultats dans l'ordre
    if c in lettres:
        print("Nombre de {} : {} (soit {}%)".format(c,lettres[c],round(lettres[c]/total*100,1)))
```

```
>>>
Nombre de a : 28 (soit 12.4%)
Nombre de b : 1 (soit 0.4%)
Nombre de c : 5 (soit 2.2%)
Nombre de d : 6 (soit 2.7%)
Nombre de e : 37 (soit 16.4%)
Nombre de f : 2 (soit 0.9%)
Nombre de g : 4 (soit 1.8%)
Nombre de i : 19 (soit 8.4%)
Nombre de j : 1 (soit 0.4%)
Nombre de l : 15 (soit 6.6%)
Nombre de m : 5 (soit 2.2%)
Nombre de n : 12 (soit 5.3%)
Nombre de o : 8 (soit 3.5%)
Nombre de p : 4 (soit 1.8%)
Nombre de q : 3 (soit 1.3%)
Nombre de r : 13 (soit 5.8%)
Nombre de s : 28 (soit 12.4%)
Nombre de t : 6 (soit 2.7%)
Nombre de u : 16 (soit 7.1%)
Nombre de v : 4 (soit 1.8%)
Nombre de x : 2 (soit 0.9%)
```

DÉFINITION 2.9 (ENSEMBLES) Un ensemble – « set » en Python – contient des éléments séparés par des virgules, le tout étant délimité par des accolades.

Un ensemble contient des éléments uniques : il ne peut y avoir de doublons dans un *set* contrairement à ce qui peut arriver dans une liste. L'autre grande différence avec les listes est qu'il n'y a pas d'ordre dans les éléments d'un ensemble permettant l'usage d'indices.

On peut construire un ensemble à partie d'un ensemble vide. Mais la construction `E={}` ne définit pas un ensemble (elle définit un dictionnaire). Pour définir un ensemble vide `E`, il faut écrire `E=set()`.
Pour ajouter des éléments dans un ensemble, on peut utiliser la méthode `add()` de la classe *set*.
Après avoir défini l'ensemble vide `E`, si on écrit `E.add("Alain")`, puis `E.add("Béatrice")`, l'ensemble `E` peut s'afficher `{"Alain","Béatrice"}` ou `{"Béatrice","Alain"}`, il s'agit dans tous les cas du même ensemble.
On peut, par contre, définir directement l'ensemble non vide `E={"Alain","Béatrice"}`.

Les opérations ensemblistes sont possibles entre deux ensembles :

- l'union `E|F` (notée `E`∪`F` en mathématiques) contient les éléments qui sont dans `E` ou dans `F` (on réunit tous les éléments en éliminant les doublons éventuels).
  Si on a `E={"Alain","Béatrice"}` et `F={"Chloé","Béatrice"}`, alors l'instruction `G=E|F` conduit à `G={"Chloé","Béatrice","Alain"}`.
- l'intersection `E&F` (notée `E`∩`F` en mathématiques) contient les éléments qui sont dans `E` et dans `F`. Avec les mêmes ensembles `E` et `F`, l'instruction `H=E&F` conduit à `H={"Béatrice"}`.
- la différence `E-F` (notée `E`\`F` en mathématiques) contient les éléments qui sont dans `E` privés de ceux qui sont dans `F`. Avec les mêmes ensembles `E` et `F`, l'instruction `I=E-F` conduit à `I={"Alain"}`.
- le ou exclusif `E^F` (notée `E`Δ`F` en mathématiques) contient les éléments qui sont dans `E` ou dans `F` mais pas dans les deux ensembles à la fois. Avec les mêmes ensembles `E` et `F`, l'instruction `J=E^F` conduit à `J={"Alain","Chloé"}`.

Il est possible de définir un ensemble à partir d'une liste (cela permet d'en expurger les doublons éventuels). Si `L=["Zoé","Yannis","Xavier","Zoé"]` alors l'instruction `E=set(L)` conduit à `E={"Xavier","Yannis","Zoé"}` (le doublon a disparu et l'ordre n'a pas été conservé).
En fait, la fonction `set` permet la construction d'un ensemble à partir de n'importe quelle valeur itérable : une liste, la fonction `range()` ou une chaîne de caractère.
L'instruction `F=set("anticonstitutionnellement")` conduit à l'ensemble `F={"t","e","c","l","n","a","i","u","o","s","m"}`.

Une autre possibilité de construction d'ensemble est la définition en compréhension, comme pour les listes : `E={"Z"+c for c in "aeiou"}` qui conduit à `E={"Za","Zi","Zu","Ze","Zo"}`.
Cette construction élimine les doublons éventuels, comme avec `F={n**2 for n in range(-5,5)}` qui conduit à `F={0,1,4,9,16,25}`.

Un ensemble ne peut pas contenir n'importe quel type de données. Il ne peut contenir que des types simples (entiers, flottants, booléens) ou des structures non mutables (chaînes, tuples). Il ne peut donc contenir ni listes, ni ensembles.

Pour enlever un élément d'un ensemble, on peut utiliser la méthode `remove()` de la classe *set*.
Cette méthode provoque une erreur si l'élément demandé n'existe pas.
Avec l'ensemble `F` précédent, l'instruction `F.remove(4)` conduit à `F={0,1,9,16,25}`, mais `F.remove(5)` conduit au message d'erreur : `KeyError: 5`.

Pour compter les éléments d'un ensemble, on peut toujours utiliser la fonction `len()` qui fonctionne donc avec les chaînes de caractères, les tuples, les listes, les dictionnaires et les ensembles.

Le mot-clé `in` permet de tester la présence d'un élément dans un ensemble (`if e in E : ...`) ou de parcourir un ensemble (`for e in E : ...`).
La méthode `issubset()` de la classe *set* permet de tester si un ensemble est inclus dans un autre.
Si `E={0,1,4,9,16,25}` et `F={0,4,16}` alors l'instruction `F.issubset(E)` retourne `True`.

Exemple 11 – Écrivons un utilitaire qui construit l'ensemble des mots contenus dans un texte.
On pourrait utiliser un dictionnaire pour indiquer le nombre d'occurrences de tel ou tel mot, mais ici, comme on ne veut que les mots présents, on va utiliser la structure de *set*.
Pour découper le texte en mot, on va utiliser la méthode `split()` de la classe *string*.
Appliquée à la chaîne `t="Bonjour, comment vas-tu?"`, l'instruction `L=t.split()` conduit à `L=["Bonjour,","comment","vas-tu?"]`, ce n'est donc pas tout-à-fait bien séparés en mots : il reste les signes de ponctuation à éliminer, les tirets et les apostrophes (il n'y en a pas dans mon exemple, mais elles unissent deux mots différents comme les tirets).

```
texte="Alice, assise auprès de sa sœur sur le gazon, commençait à
texte=texte.lower()
ponctuation=[" ","!","?",".",";",",",":","\"","'","-","«","»"]
cesure=["'","-"]
mots=texte.split()                    # découper grossièrement en mots
for i in range(len(mots)) :           # traitement des mots
    if mots[i][-1:] in ponctuation :
        mots[i]=mots[i][:-1]          # enlever la ponctuation
    for c in cesure :
        if c in mots[i] :
            deuxMots=mots[i].split(c)# liste de 2 mots liés
            mots[i]=deuxMots[0]      # remplacer par le premier mot
            mots.append(deuxMots[1]) # ajouter le 2ème mot lié
mots=set(mots)
print("Nombre de mots du texte : ",len(mots))
print("Liste des mots du texte : ",mots)
```

```
>>>
Nombre de mots du texte :  45
Liste des mots du texte :  {'faire', '',
'ne', 'dialogues', 'que', 'pas', 'belle',
'les', 'sur', 'assise', 'alice', 'pensait',
'la', 'quoi', 'sœur', 'de', 'avance',
'sa', 'rester', 'mais', 'ennuyer', 'd',
's', 'gazon', 'jeté', 'images', 'là',
'elle', 'auprès', 'le'  'rien', 'lisait',
'sans', 'qu', 'causeries', 'livre', 'une',
'un', 'à', 'ou','deux', 'commençait',
'yeux', 'fois', 'avait'}
```

Le résultat n'étant pas très lisible, je le mets en forme en triant les mots selon leur nombre de caractères. D'autre part, les mots qui ont subi une éludation du fait de l'apostrophe apparaissent peu lisibles et sont parfois redondants (`s` et `se` désignent le même mot). J'ai donc ajouté aussi un morceau de code pour corriger la liste des mots obtenus.

```
mots=set(mots)                        ############## Rectification
if "" in mots : mots.remove("")       # supprimer un mot vide
if "d" in mots  :
    mots.remove("d")
    mots.add("de")                    # remplacer d par de
if "s" in mots  :
    mots.remove("s")
    mots.add("se")                    # remplacer s par se
if "qu" in mots :
    mots.remove("qu")
    mots.add("que")                   # remplacer qu par que
if "l" in mots and ("le" in mots or "la" in mots):
    mots.remove("l")                  # supprimer l quand redondant
longMax=0                             ############## Affichage
for mot in mots :                     # détermination du plus grand mot
    if len(mot)>longMax:
        longMax=len(mot)
for long in range(1,longMax+1):       # affichage mots par longueur
    print("\n Mots de longueur",long,end=" : ")
    for mot in mots :
        if len(mot)==long:
            print(mot,end=" ")
```

```
>>>
Mots de longueur  1 : à
Mots de longueur  2 : là ou sa un se de la le ne
Mots de longueur  3 : pas une que les sur
Mots de longueur  4 : elle rien jeté sans quoi deux sœur mais fois yeux
Mots de longueur  5 : avait belle faire livre alice gazon
Mots de longueur  6 : assise lisait auprès images avance rester
Mots de longueur  7 : pensait ennuyer
Mots de longueur  8 :
Mots de longueur  9 : causeries dialogues
Mots de longueur  10 : commençait
```

Ce programme peut maintenant être utilisé sur un plus grand texte. Je l'essaie sur le 1er chapitre de Alice au Pays des Merveilles [7]. Cela m'indique qu'il y a encore quelques erreurs : un signe de ponctuation a été oublié (un tiret différent du signe de la soustraction) et les parenthèses n'ont pas été traitées correctement. Cet extrait de l'affichage final le montre :
`Mots de longueur 10 : probable.) étonnement tristement conduisait commençait paraissait pénètrerai rapetisser aveuglette brillantes impossible maintenant profondeur clairement remarquera ténébreuse agenouilla.`

7. Livre disponible sur `https://fr.wikisource.org/wiki/Alice_au_pays_des_merveilles/Texte_entier`

DÉFINITION 2.10 (FICHIERS CSV) Un fichier CSV (*Comma Separated Values*) contient des lignes de données structurées où les valeurs sont séparées par un caractère particulier (virgule, espace, tabulation, point-virgule ou deux-points)

Les fichiers d'extension csv sont des fichiers textes ordinaires : ils peuvent être créés avec un simple éditeur de texte, un programme ou bien aussi avec un tableur.
Quand on demande au tableur *calc* d'OpenOffice d'enregistrer au format csv le tableau ci-dessous (à gauche) qui contient des éléments du calendrier 2019, il propose d'utiliser la virgule comme séparateur et de mettre des guillemets droits (") autour des valeurs ; je choisis de ne pas ajouter de guillemets. Ce fichier csv, ouvert avec un éditeur de texte est montré à droite.

1	Jour	date	Semaine
2	lundi	09/09/19	B
3	mardi	10/09/19	B
4	mercredi	11/09/19	B
5	jeudi	12/09/19	B
6	vendredi	13/09/19	B
7	samedi	14/09/19	B
8	dimanche	15/09/19	B
9	lundi	16/09/19	A
10	mardi	17/09/19	A
11	mercredi	18/09/19	A
12	jeudi	19/09/19	A
13	vendredi	20/09/19	A
14	samedi	21/09/19	A
15	dimanche	22/09/19	A

```
Jour,date,Semaine
lundi,09/09/19,B
mardi,10/09/19,B
mercredi,11/09/19,B
jeudi,12/09/19,B
vendredi,13/09/19,B
samedi,14/09/19,B
dimanche,15/09/19,B
lundi,16/09/19,A
mardi,17/09/19,A
mercredi,18/09/19,A
jeudi,19/09/19,A
vendredi,20/09/19,A
samedi,21/09/19,A
dimanche,22/09/19,A
```

Pour être lu par un programme Python :

- le fichier csv doit être dans le même dossier que le programme Python (sinon il faut utiliser la fonction `chdir` du module `os`)
- on utilise la fonction `open(<nomFichier>,"r")` (le `r` signifiant *read*, lecture) pour obtenir le fichier dans une variable
- le contenu du fichier est récupéré par la méthode `readlines()` qui retourne une table dont les éléments sont les lignes terminées par le caractère `\n` de fin de ligne
- le fichier est fermé avec la méthode `close()`
- le contenu de la table doit être débarrassé du caractère de fin de ligne (`\n`) avant d'être traité par la méthode `split(",")` à qui on indique le séparateur des champs de la table, qui a pour fonction de partager la ligne en table

```
fichier=open("calendrier.csv","r")
donnees=fichier.readlines()
fichier.close()
calendrier=[]
for ligne in donnees :
    jour=ligne[:-1].split(",")
    calendrier.append(jour)
for c in calendrier:
    print(c[0],c[1],c[2])
```

```
Jour date Semaine
lundi 09/09/19 B
mardi 10/09/19 B
mercredi 11/09/19 B
jeudi 12/09/19 B
vendredi 13/09/19 B
samedi 14/09/19 B
dimanche 15/09/19 B
lundi 16/09/19 A
...
```

Il y a d'autres façons de récupérer les données d'un fichier csv.
On remarque que l'en-tête qui décrit les données est incorporée à celles-ci. On peut utiliser cet en-tête pour construire un dictionnaire dont les clés sont les informations lues dans la 1re ligne du fichier.
Le module `csv` de Python permet cela :

- importer la fonction `DictReader` du module `csv`
- utiliser la fonction `open(<nomFichier>,"r")` comme précédemment
- le contenu du fichier est récupéré par la fonction `DictReader(<variable>,delimiter=',')` qui retourne un objet que l'on récupère dans une variable
- le contenu de l'objet est lu, ligne par ligne, chaque ligne étant un dictionnaire
- le fichier est fermé avec la méthode `close()`

Le programme suivant lit le fichier `"calendrier.csv"` et construit une table de dictionnaires.
Cette table est ensuite lue afin d'en extraire les informations que l'on souhaite.

```
from csv import DictReader
fichier=open("calendrier.csv","r")
donnees=DictReader(fichier,delimiter=',')
calendrier=[]
for ligne in donnees :
    calendrier.append(ligne)
fichier.close()
for c in calendrier:
    if c['Jour']=='lundi':
        print(c['date'],c['Semaine'])
```

```
09/09/19 B
16/09/19 A
23/09/19 B
30/09/19 A
```

Données ouvertes

De nombreuses données sont accessibles via internet sur certains sites spécialisés, le format csv étant alors souvent utilisés pour stocker l'information. La France diffuse des données publiques sur le site data.gouv.fr [8].

Exemple 12 – Téléchargeons le fichier concernant les vacances scolaires en France
`https://www.data.gouv.fr/fr/datasets/le-calendrier-scolaire/`
Ce fichier existe en deux formats : csv et json. Choisissons de télécharger le fichier csv.
On peut prévisualiser le fichier, ce qui permet d'en connaître les descripteurs :
`Description, Population, Date de début, Date de fin, Académies, zones, annee_scolaire`

	Description	Population	Date de début	Date de fin	Académies	zones	annee_scolaire
1	Vacances de Noël		December 23, 2017	January 8, 2018	Rennes	Zone B	2017-2018
2	Vacances d'Hiver		February 24, 2018	March 12, 2018	Rouen	Zone B	2017-2018
3	Grandes Vacances	Enseignants	July 7, 2018	September 3, 2018	Martinique	Martinique	2017-2018
4	Vacances de Noël		December 23, 2017	January 8, 2018	Orléans-Tours	Zone B	2017-2018
5	Vacances d'Hiver		February 24, 2018	March 12, 2018	Nancy-Metz	Zone B	2017-2018

Il y a seulement 627 lignes dans ce fichier que l'on ouvre facilement avec un tableur.
Certains fichiers présents sur ce site sont beaucoup plus gros et ne doivent pas être ouverts avec un tableur (l'opération risque d'être excessivement longue).

Supposons que l'on cherche le calendrier scolaire pour l'académie de Paris.
Comme dans l'exemple précédent, on peut au choix :

- disposer les valeurs dans un dictionnaire et en extraire la partie qui nous intéresse
- traiter les lignes du fichier comme des listes avec la méthode `.split(";")`

Ces deux techniques conduisent aux programmes ci-dessous qui produisent le même affichage :

```
from csv import DictReader
with open("fr-en-calendrier-scolaire.csv", encoding='utf-8', mode='r') as fichier:
    donnees=DictReader(fichier,delimiter=';')
    calendrier=[]
    for ligne in donnees :
        calendrier.append(ligne)
for c in calendrier:
    if c['Académies']=='Paris' and c['annee_scolaire']=='2019-2020':
        print(c['Description'],"du",c['Date de début'],"au",c['Date de fin'])
```

```
with open("fr-en-calendrier-scolaire.csv", encoding='utf-8', mode='r') as fichier:
    donnees=fichier.readlines()
calendrier=[]
for ligne in donnees :
    jour=ligne[:-1].split(";")
    calendrier.append(jour)
for c in calendrier:
    if c[4]=='Paris' and c[6]=='2019-2020':
        print(c[0],"du",c[2],"au",c[3])
```

```
Vacances de Noël du 2019-12-21 au 2020-01-06
Vacances d'Hiver du 2020-02-08 au 2020-02-24
Vacances d'Été du 2020-07-04 au
Vacances de la Toussaint du 2019-10-19 au 2019-11-04
Vacances de Printemps du 2020-04-04 au 2020-04-20
Pont de l'Ascension du 2020-05-20 au 2020-05-25
```

J'ai utilisé ici la construction `with open(<nomFichier>,encoding='utf-8',mode='r') as fichier:` qui s'occupe de l'ouverture et de la fermeture du fichier. Cette construction admet aussi une option d'`encoding` qui permet une lecture correcte des chaînes de caractères du fichier.

8. Données ouvertes françaises sur `https://www.data.gouv.fr/fr/`

Écriture

Un programme Python peut également produire un fichier csv.
Pour écrire, il faut ouvrir le fichier avec l'option `"w"` (pour *write*) ou `"a"` (pour *append*) ; l'option `"w"` écrase le fichier s'il existe, alors que l'option `"a"` enregistre les nouvelles données à la fin du fichier.
Ensuite, l'écriture est réalisée avec la méthode `write(" ...texte... \n")`.

Exemple 13 – Pour l'exemple 6, nous avons tiré aléatoirement les coordonnées de `n` points du plan dont les abscisses varient entre 0 et 1. Si nous voulons réutiliser ces points (pour être traités par un autre programme), il apparait nécessaire d'enregistrer les données. Le programme qui suit réalise les tirages et les enregistrements de ces coordonnées.
Une fois enregistrées, les données peuvent être traitées par des programmes spécifiques. Ici, je montre seulement comment effectuer la lecture de ces données pour extraire la distance moyenne entre deux points.

écriture

```
from random import random
n=1000
Total=0
fichier=open("points.csv","w")
for i in range(n):
    P=random()
    Q=random()
    fichier.write(str(P)+" "+str(Q)+"\n")
fichier.close()
print("Fin de l'enregistrement")
```

```
Fin de l'enregistrement
Moyenne des distances = 0.535194517728271
Nombre de couples de points lus : 500
```

lecture

```
from math import sqrt
Total,nbrLignes,fin=0,0,False
with open("points.csv", mode='r') as fichier:
    while not fin:
        distance=0
        for coord in range(2):
            ligne=fichier.readline()
            if ligne=='':
                fin=True
                break
            nbrLignes+=1
            P,Q=ligne.split(" ")
            distance+=(float(P)-float(Q))**2
        distance=sqrt(distance)
        Total+=distance
print("Moyenne des distances = {} ".format(Total/(nbrLignes//2)))
print("Nombre de couples de points lus : {}".format(nbrLignes//2))
```

Ce programme utilise l'instruction `readline()` qui ne lit qu'une seule ligne du fichier à la fois. Ceci est utilisé ici pour lire successivement deux lignes, correspondant aux coordonnées de deux points.
Pour identifier la fin du fichier, je teste le contenu de la ligne lue, sachant que s'il n'y a plus de ligne à lire, l'instruction `readline()` retourne une chaîne vide.
Noter que les données numériques lues sont des chaînes de caractères qu'il faut convertir en flottant.

Fichiers textes

Les données ne sont pas toujours structurées comme des tableaux.
On peut lire et écrire des fichiers textes simples (l'extension est alors naturellement `.txt`).
Voici quelques méthodes que l'on peut utiliser pour lire un fichier texte :

- La méthode `<monFichier>.read()` renvoie tout le contenu du fichier sous la forme d'une chaîne de caractères.
- La méthode `<monFichier>.readlines()` (déjà utilisées) renvoie une liste contenant toutes les lignes du fichier.
- La méthode `<monFichier>.readline()` lit une ligne du fichier et la renvoie sous forme de chaîne de caractères. Pour lire le fichier ligne par ligne, il faut l'intégrer à une boucle `while` qui détecte la fin du fichier (par exemple avec `while ligne != "":`)
- On peut créer un objet « itérable » sur lequel itérer la lecture :
  avec `with open(<monFichier>, "r") as <monIterable>:`
  Il suffit alors de parcourir le fichier en écrivant :
  `for ligne in <monIterable>:...`
  On obtient successivement dans `ligne` toutes les lignes du fichier (les lignes sont identifiées par le caractère de fin `\n`).

Pour écrire dans un fichier texte, la méthode `.write()` déjà décrite suffit. Elle peut être employée avec des données structurées (avec un séparateur entre les données d'une ligne et un caractère de fin de ligne) comme avec des textes quelconques.

4. Fonctions

a. Les fonctions en Python

DÉFINITION 2.11 (FONCTION) Une fonction en Python est un morceau de programme qui commence par l'instruction `def <nomFonction>(<arg1>,<arg2>,...):` suivie d'un bloc d'instructions indentées.

Remarques :

- Le nom de la fonction, noté `<nomFonction>`, doit être choisi en rapport avec ce que réalise la fonction. Il est obligatoire et ne doit pas être déjà utilisé.
  En effet, il existe de nombreuses fonctions prédéfinies (*built-in*) en Python dont certaines sont toujours présentes (la liste en a été donnée plus haut) et d'autres ont éventuellement été importées. La fonction `abs`, par exemple, est prédéfinie ; il ne faut pas nommer une autre fonction `abs` sous peine de rendre la fonction prédéfinie inopérante (sauf si, volontairement on veut la redéfinir).
- Les arguments d'une fonction, notés `<arg1>,<arg2>,...`, sont optionnels.
  Il peut n'y en avoir aucun, on laisse alors les parenthèses vides.
  Ces arguments sont des variables locales : des variables qui n'existent pas en dehors de la fonction. Le corps de la fonction peut contenir d'autres variables locales, définies pour les besoins de la fonction et dont l'existence ne dépasse pas le cadre de celle-ci.
- Une fonction peut transmettre des valeurs à l'instruction du programme qui l'a appelée : dans ce cas, on utilise le mot-clé `return` suivi de la ou des valeur(s) retournée(s).
  Après une instruction `return <val1>,<val2>,...`, le pointeur du programme sort de la fonction et retourne à l'instruction appelante.
  Il peut y avoir plusieurs `return` dans une fonction, liés aux résultats de tests ; il peut n'y en avoir aucun (la fonction est alors appelée « procédure », mais cette distinction n'est pas importante en Python).

EXEMPLE 14 – Réalisons une fonction qui teste si trois nombres positifs `a,b,c` fournis en arguments peuvent être les côtés d'un triangle. On sait que chacun des côtés doit être inférieur à la somme des deux autres (inégalités triangulaires). La fonction examine donc cette éventualité et retourne le booléen `True` si le triangle existe et `False` s'il n'existe pas.
Le programme qui utilise cette fonction consiste juste en deux instructions qui en teste le bon fonctionnement.
L'ajout d'un commentaire entre triples guillemets `"""..."""` après la ligne de définition fourni à un utilisateur éventuel de la fonction, la possibilité d'une aide directement dans la console (sans entrer dans le programme) avec la syntaxe `help(<nomFonction>)`. Cette aide donne, en principe, l'objet de la fonction et sa valeur de retour sur une 1re ligne, les conditions d'utilisation sur une 2e ligne.

```
def triangle(a,b,c):
    """ Teste si les nombres entrés peuvent être les côtés d'un triangle
        Les nombres entrés (a,b,c) sont supposés être positifs """
    if a>b+c or b>a+c or c>b+a : return False
    return True

print(triangle(3,4,5))
print(triangle(3,4,8))
```

```
>>>
True
False
>>> help(triangle)
Help on function triangle in module __main__:

triangle(a, b, c)
    Teste si les nombres entrés peuvent être les côtés d'un triangle
    Les nombres entrés (a,b,c) sont supposés être positifs
```

Supposons que l'on cherche à savoir également si le triangle éventuel a un angle droit, on doit tester l'égalité de Pythagore $a^2=b^2+c^2$ (si c'est `a` le plus grand des côtés). Je vais confier cette tâche à une autre fonction, `triangleRectangle`, qui sera appelée depuis la fonction `triangle`.

Plutôt que de chercher le plus grand côté, je vais tester les trois égalités de Pythagore : si l'une d'entre elle est vraie, le triangle a un angle droit.
J'ai été obligé de modifier ma fonction `triangle` initiale car il y a maintenant trois valeurs de retour, pour les trois cas envisagés : les booléens `True` et `False` ne suffisaient plus.

```
def triangleRectangle(a,b,c):
    """ Teste si les nombres entrés sont les côtés d'un triangle rectangle
        Les nombres entrés (a,b,c) sont supposés être les côtés d'un triangle """
    if a**2==b**2+c**2 or b**2==a**2+c**2 or c**2==b**2+a**2 :
        return True
    return False

def triangle(a,b,c):
    """ Teste si les nombres entrés peuvent être les côtés d'un triangle
        Les nombres entrés (a,b,c) sont supposés être positifs          """
    if a>b+c or b>a+c or c>b+a : return "pas un triangle"
    if triangleRectangle(a,b,c): return "triangle rectangle"
    return "triangle non rectangle"

print(triangle(3,4,5))
print(triangle(3,4,6))
print(triangle(3,4,8))
```

```
>>>
triangle rectangle
triangle non rectangle
pas un triangle
```

Utilisation des valeurs de sortie

On a vu qu'une fonction pouvait ne pas avoir de valeur de sortie.
Lorsqu'il y a un `return`, la valeur retournée peut être utilisée dans une instruction d'affichage, un `print()` comme dans l'exemple ci-dessus. Si il n'y a qu'une seule valeur, elle peut être affectée à une variable ou intégrée à un calcul.
Un `return` peut être suivi de plusieurs valeurs de retour : il s'agit alors d'un tuple. Les valeurs contenues dans le tuple peuvent être récupérées chacune dans une variable séparée ou bien dans un tuple de réception.

Exemple 15 – Écrivons un programme qui interroge l'utilisateur sur les tables de multiplication : une fonction sans argument `table()` génère deux nombres aléatoires entre 2 et 9, et renvoie un texte mis en forme pour l'interrogation ainsi que la réponse attendue.
Le programme principal qui appelle la fonction `table()` récupère les deux valeurs du tuple de retour dans des variables séparées. La première sert à interroger l'utilisateur, l'autre à vérifier la réponse.
J'ai ajouté deux compteurs pour afficher également le score. La dynamique de l'interrogation ne doit pas laisser de côté la nécessité de pouvoir sortir du mode interrogation : il suffit de taper 0 pour cela (0 ne pouvant pas être la réponse à une question).

```
from random import randint
def table() :
    a=randint(2,9)
    b=randint(2,9)
    c=a*b
    return str(a)+'\u00D7'+str(b)+'=',c

score,nbr=0,0
while True:
    nbr+=1
    question,bonneReponse=table()
    reponse=int(input(question))
    if reponse==0 : break
    elif reponse==bonneReponse :
        score+=1
        print("Bravo! Score:{}/{}".format(score,nbr))
    else : print("Non : "+question+str(bonneReponse))
```

```
>>>
7×9=63
Bravo! Score:1/1
8×3=24
Bravo! Score:2/2
7×5=30
Non : 7×5=35
4×9=36
Bravo! Score:3/4
2×5=0
```

Altération d'un tableau

Il a déjà été mentionné plus haut (partie 2.7) qu'un tableau n'est pas transmis à une fonction en valeurs ; c'est l'adresse du tableau en mémoire qui est transmise. Autrement dit, lorsqu'on transmet un tableau à une fonction qui l'attend en argument, le tableau initial peut être modifié par la fonction. Cela n'est pas le cas avec d'autres types de données : un nombre passé en argument ne sera pas modifié par la fonction car c'est une copie en valeur du nombre qui est transmise.

Exemple 16 – Examinons cette situation où l'on effectue le même traitement aux éléments d'un tableau d'entiers : on divise par deux les nombres pairs et on prend le successeur du triple pour les nombres impairs. Selon la façon de passer (ou pas) la liste originale et selon la façon de procéder, la liste initiale est modifiée (ou non) :

- La fonction `traitement1` récupère la liste en argument et la modifie. La liste initiale n'a pas le même nom que la liste récupérée mais pourtant la liste initiale est modifiée.
- La fonction `traitement2` récupère la liste en argument et la recopie dans une autre liste qu'elle modifie. La liste initiale n'est pas modifiée (elle a juste été recopiée), la nouvelle liste peut être transmise au programme appelant. Ainsi les listes initiale et modifiée peuvent cohabiter.
- La fonction `traitement3` ne récupère pas la liste en argument. Par contre, comme il s'agit d'une variable globale, celle-ci peut être utilisée dans une fonction et modifiée. Remarquer alors l'emploi du nom de la liste globale dans la fonction.

```
def traitement1(L):
    for i in range(len(L)):
        if L[i]%2==0 : L[i]=L[i]//2
        else : L[i]=3*L[i]+1

Liste=list(range(1,9))
print(Liste)
traitement1(Liste)
print(Liste)
```

```
[1, 2, 3, 4, 5, 6, 7, 8]
[4, 1, 10, 2, 16, 3, 22, 4]
```

```
def traitement2(L):
    M=L[:]
    for i in range(len(M)):
        if M[i]%2==0 : M[i]=M[i]//2
        else : M[i]=3*M[i]+1
    return M

Liste1=list(range(1,9))
print(Liste1)
Liste2=traitement2(Liste1)
print(Liste1)
print(Liste2)
```

```
[1, 2, 3, 4, 5, 6, 7, 8]
[1, 2, 3, 4, 5, 6, 7, 8]
[4, 1, 10, 2, 16, 3, 22, 4]
```

```
def traitement3():
    for i in range(len(L)):
        if L[i]%2==0 : L[i]=L[i]//2
        else : L[i]=3*L[i]+1

L=list(range(1,9))
print(L)
traitement3()
print(L)
```

```
[1, 2, 3, 4, 5, 6, 7, 8]
[4, 1, 10, 2, 16, 3, 22, 4]
```

Le fonctionnement est différent pour les variables de types numériques. Je donne ci-dessous un exemple avec un nombre transmis (ou pas) en argument à une fonction. Le nombre transmis n'est jamais modifié : on peut en utiliser la copie qui est transmise en valeur, mais il ne s'agit pas de la même variable. La fonction `traitement3` échoue car elle tente de modifier une variable globale. Le message d'erreur dit que la variable locale `A` est utilisée avant d'être initialisée car, malgré les noms identiques utilisés dans le programme et dans la fonction, il s'agit de variables différentes.
On peut utiliser une variable globale dans une fonction, sans la passer en argument, à condition de ne pas chercher à la modifier. C'est ce que fait la fonction `traitement4`.
Pour avoir l'autorisation de modifier une variable globale, il faut la déclarer avec le mot-clé `global`, ce que fait la fonction `traitement5`.

```
def traitement1(A):
    if A%2==0 : A=A//2
    else : A=3*A+1

A=120
print(A)
traitement1(A)
print(A)
```

```
120
120
```

```
def traitement2(A):
    B=A
    if B%2==0 : B=B//2
    else : B=3*B+1
    return B

A=120
print(A)
B=traitement2(A)
print(A)
print(B)
```

```
120
120
60
```

```
def traitement3():
    if A%2==0 : A=A//2
    else : A=3*A+1

A=120
print(A)
traitement3()
print(A)
```

```
120
Traceback  ...
UnboundLocalError:
local variable 'A'
referenced before
assignment
```

```
def traitement4():
    B=A
    if B%2==0 : B=B//2
    else : B=3*B+1
    return B

A=120
print(A)
A=traitement4()
print(A)
```

```
120
60
```

```
def traitement5():
    global A
    if A%2==0 : A=A//2
    else : A=3*A+1

A=120
print(A)
traitement5()
print(A)
```

```
120
60
```

Jeux de tests

Jusqu'à présent, les tests des fonctions présentés consistaient en un affichage de la valeur de retour. Cette pratique n'est pas mauvaise en soi, mais il faut que l'utilisateur effectue lui-même la comparaison avec le résultat attendu et, une fois la fonction validée, il lui faut supprimer le jeu de tests (on ne veut plus en afficher le résultat) ou le passer en commentaire.
On peut confier ces deux opérations (comparer au résultat attendu - suppression après validation) à la fonction `assert`. La syntaxe de cette fonction est :

```
assert <valeurRetour>==<valeurAttendue>
```

Si le test est `True`, la fonction `assert` ne fait rien (pas d'affichage, pas d'interruption de programme).
Si le test est `False`, la fonction `assert` interrompt le programme et affiche un message d'erreur.

Exemple 17 – Voici un exemple de fonction qui est supposée supprimer les doublons d'une liste. J'ai mis l'instruction `assert L==[1,2,3,5]` car je sais que la liste `L` expurgée de ses doublons contient ces éléments. Mais cette instruction interrompt le programme et affiche `AssertionError`.

```
def doublon():
    for elem in L:
        if L.count(elem)>1 :
            L.remove(elem) #supprime le 1er élément elem

L=[1,2,3,5,2,1,3,1]
doublon()
assert L==[1,2,3,5],"Liste correcte triée"
```

```
Traceback (...
AssertionError
```

En réalité, la liste `L` modifiée par la fonction `doublon()` est `[5,2,3,1]`. C'est le principe de ma fonction `doublon()` qui conduit à enlever le 1er élément en double (en non le dernier).
Mon test est mal rédigé : il ne me permet pas de constater que la liste est bien expurgée de ses doublons (ma fonction est correcte). Pour supprimer ce faux négatif, je dois comparer des listes triées.
Voici une assertion qui n'interrompt pas le programme, signe de sa correction.

```
L=[1,2,3,5,2,1,3,1]
doublon()
assert sorted(L)==sorted([1,2,3,5])
print("test 1 passé")
assert sorted(L)==[1,2,3,5]
print("test 2 passé")
assert L==[1,2,3,5]
print("test 3 passé")
```

```
test 1 passé
test 2 passé
Traceback (...
AssertionError
```

```
L=[1,-2,3,5,-2,1,3,1]
doublon()
assert sorted(L)==sorted([1,-2,3,5])
print("test 1 passé")
assert sorted(L)==[1,-2,3,5]
print("test 2 passé")
assert L==[1,-2,3,5]
print("test 3 passé")
```

```
test 1 passé
Traceback (...
AssertionError
```

En modifiant la liste sur laquelle s'effectue le test, on peut tester différentes configurations.
La spécification de notre fonction est qu'elle supprime les doublons : fonctionne t-elle correctement avec une liste vide ? avec une liste ne contenant que des doublons ? avec une liste contenant autre chose que des nombres ?
Dans l'état actuel, le test d'une ligne s'effectue en trois temps : je déclare une liste, je lance la fonction, je teste si la liste modifiée est bien celle attendue. 'est long à écrire, surtout si j'ai plusieurs cas de figures à tester.
Je modifie donc légèrement ma fonction pour qu'elle prenne une liste en argument et qu'elle renvoie la liste modifiée ; ceci, afin de pouvoir écrire ma batterie de tests sur une seule ligne.
Voici finalement ma fonction `doublon()` avec sa batterie de tests validée.
Le `print` final n'est pas nécessaire : je l'ai mis juste pour montrer que le programme est passé à travers la batterie de tests (les quatre instructions `assert ...` sans être interrompu.

```
def doublon(L):
    for elem in L:
        if L.count(elem)>1 :
            L.remove(elem) #supprime le 1er élément elem
    return L
assert sorted(doublon([]))==sorted([])
assert sorted(doublon([2,2,2]))==sorted([2])
assert sorted(doublon([1,1,1,-1]))==sorted([1,-1])
assert sorted(doublon(["a","b","a","a"]))==sorted(["b","a"])
print("tests passés")
```

```
tests passés
```

Je serai bien avisé de ne pas supprimer cette batterie de tests si j'étais amené, par la suite, à modifier ma fonction `doublon()`. Car alors il ne faut pas négliger de retester tous les cas de figure.

Si une fonction n'effectue pas ce qu'elle devrait, on doit s'interroger sur les moyens de la corriger : Pour détecter une erreur, on peut utiliser des instructions d'affichage, à l'intérieur de la fonction, en des endroits stratégiques, pour inspecter différents endroits du programme que l'on estime cruciaux :

- Le programme entre t-il dans telle ou telle partie du programme ?
  Des tests incorrects sont peut-être à l'origine d'une erreur d'aiguillage... Il peut être intéressant d'écrire, comme je l'ai fait un peu plus haut `print("test 1 passé")`, `print("test 2 passé")`, etc.
- Combien vaut telle ou telle variable, à tel ou tel moment ?
  On peut afficher l'étiquette et la valeur d'une ou plusieurs variables, par exemple en écrivant `print("var1=",var1,"var2=",var2)`, ceci en différents endroits du programme.

Une source d'erreur fréquente est une mauvaise initialisation des variables associées à une boucle (le compteur de boucle en particulier) ou une erreur d'indice à la sortie d'une boucle. Ce type d'erreur peut être détecté par un affichage des variables comme il a été dit plus haut, mais si le programme passe de très nombreuses fois dans la boucle testée, il peut être difficile de s'y retrouver dans tous les affichages.
Certains environnement de développement permette de dérouler pas à pas les instructions du programme, avec un affichage du contenu de toutes les variables déclarées. Cela peut se révéler fastidieux d'examiner chacune des étapes mais parfois ce n'est qu'à ce prix qu'on découvre l'incongruité qui empêche le bon déroulement des opérations.