Objectifs :

- ⇨ Comprendre comment sont gérées les variables en python
- ⇨ Comprendre le fonctionnement des boucles (condition logique, itérateurs, …)

I – Les variables

1) Notion de variable

Les programmes ont besoin lors de leur exécution de mémoriser certaines informations qui sont susceptibles de varier. Pour ce faire les langages de programmation offrent la notion de variable.
Une variable est un élément de mémorisation qui permet à un programme de stocker une valeur. Les variables d'un programme ont trois caractéristiques importantes :

- Ils ont une durée de vie limitée et ne stockent l'information que pendant la durée du programme qui a créé la variable.
- Une variable n'est accessible que par le programme qui l'a créé (nous préciserons cela avec la *portée des variables* dans le prochain chapitre).
- La valeur d'une variable peut varier : les variables peuvent stocker différentes valeurs au cours de leur vie (la nouvelle valeur remplaçant l'ancienne).

2) Nature d'une variable

Les variables peuvent être de différents types. Des plus simples (Vrai/Faux, nombre entier) aux plus complexes (Annuaire téléphonique, modèle 3D d'un objet, …).
Au cours de l'année, nous verrons en détail les types de base de python ainsi que quelques types construits, mais il peut être utile de présenter dès maintenant les principaux types de base :

Type	Nom en python	Exemple
Booléen (valeur binaire)		Il n'y a que deux valeurs possibles : `True` et `False` (attention à la majuscule au début)
Entier relatif		145 -998766
Flottant (nombres à virgule)		-5.6 2. Le point (.) indique qu'il s'agit d'un float et non d'un entier 6.2e-5 vaut $6{,}2 \times 10^{-5}$
Chaîne de caractère		"Joliot-Curie" 'A' On peut utiliser les simples ou doubles cotes (mais il faut démarrer et finir la chaîne avec le même symbole).

On peut convertir un type de variable en un autre en utilisant la fonction de conversion qui porte le nom du type de destination. Ex : pour convertir en entier `int(3.14)` donne `3` ; pour convertir en chaîne de caractère `str(-7)` donne `"-7"`.

3) Manipulation des variables

Pour utiliser une variable, on passe généralement par trois étapes :

1. La déclaration : Permet de nommer la variable et de donner son type
2. L'allocation : Le système alloue une zone de la mémoire où stocker la valeur de la variable
3. L'initialisation : On fixe une certaine valeur de départ à la variable.

Python étant un langage à typage dynamique, les 3 étapes se font en une seule fois lors de la première affectation de la variable et il n'est généralement pas nécessaire de déclarer son type car python le devine à la façon dont est écrite la donnée.
L'opérateur d'affectation en python est « = ». Il permet **d'affecter la valeur à droite du signe « = » à la variable à gauche du signe « = »**. Si la variable n'existe pas, elle est créée et initialisée par cette affectation.

Ainsi le signe « = » de python n'a pas du tout la même signification que le signe « = » des mathématiques. Il correspond plutôt au signe « ← » utilisé en algorithmique.

```
1 x = 6   # Création de la variable x
2 y = 3   # Création de la variable y
3 y = x + y  # y vaut maintenant 9
4 x = "toto"  # y change de type et devient
5             # une chaîne de caractères
6 w = z = 5 # Création de w et z valant 5
7 z = 8*8   # z vaut 64, mais w reste à 5
```

4) Nom d'une variable et donnée

La façon de gérer les variables change d'un langage à un autre. Python lui fait une séparation entre les noms des variables et les données.
Les noms des variables sont stockés dans l'..... du programme, tandis que les données (qui sont enregistrées comme des « objets »), elles, sont stockées dans un espace mémoire séparé appelé « espace des objets ».
Voyons comment fonctionne ce mécanisme en détaillant ce qui se passe dans l'exemple de programme précédant :

Ligne 1
Ligne 2
Ligne 3
Ligne 4 et 5
Ligne 6
Ligne 7

Espace de nommage	Espace des objets

5) Suppression d'une variable

On peut supprimer une variable de l'espace de nommage grâce à l'instruction du langage `del` qui prend en argument le nom de la variable à supprimer.
Cette instruction supprime le nom dans l'espace de nommage, mais pas l'objet vers lequel elle pointe.

```
1  a = b = "toto"
2  del(b)
3  print(b)
```

Espace de nommage	Espace des objets

II – Les boucles

Dans un programme il est souvent nécessaire de faire plusieurs fois une certaine opération (dessiner plusieurs fois une maison, examiner chaque élément d'une liste de contacts, …). Cela tombe bien car répéter c'est ce qu'un ordinateur fait de mieux. Et lui ne se lasse jamais.

On a vu en TP les deux types de boucles que propose python et nous allons maintenant les détailler.

1) La boucle while

Il s'agit d'une boucle dont le nombre d'itérations (de répétitions) ..... .
Sa structure est la suivante :

```
while condition logique :
      Bloc
      d'instruction
      à
      répéter
suite du programme
```

La boucle va s'exécuter tant que la condition logique est vérifiée (qu'elle vaut `True`). Mais qu'est-ce qu'une condition logique ?

a. Condition logique

Une condition logique est une expression mathématique contenant des **opérateurs logiques**. Comme toutes les expressions, celle-ci va être évaluée par python et le résultat sera de type booléen, c'est-à-dire `True` ou `False`.

Ces expressions peuvent être très simples comme par exemple a < 3 ou plus complexes en utilisant des opérateurs moins évidents mais tout de même relativement intuitif récapitulés dans le tableau de référence ci-après :

Les opérateurs logiques et de test :	Exemple
Supérieur à : > supérieur ou égal : >= *idem pour inférieur*	`7 < 3` vaut **`False`** et `24 >= (22+2)` vaut **`True`**
Test d'égalité : == Test de différence : !=	`2 == (1+1)` vaut **`True`** et `8 != (9-1)` vaut **`False`**
Test d'appartenance : `a` **`in`** `b` (vrai si a est dans b) *A utiliser sur des listes ou des chaînes de caractères*	`'i' in 'chimie'` vaut **`True`** `13 in [1, 3.4, 12, 9]` vaut **`False`**
Et logique : **`and`** `a` **`and`** `b` est vrai si a et b sont tous les deux vrais	`(4 > 8) and (5 == (2+3))` vaut **`False`** `(7 != 9) and (8 > 6)` vaut **`True`**
Ou logique : **`or`** `a` **`or`** `b` est vrai si a ou b est vrais (ou qu'ils sont tous 2 vrais)	`(4 > 8) or (5 == (2+3))` vaut **`True`** `(7 == 9) or (8 <= 6)` vaut **`False`**
Négation : **`not`** (permet d'inverser une condition) **`not`** `a` est *vrai* si a est faux et *faux* si a est vrai	**`not`** `(2 == (1+1))` vaut **`False`** **`not`** `(6 < 3)` vaut **`True`**

Remarque importante :
Le test d'égalité s'écrit bien « `==` » et non simplement « `=` » qui est l'opérateur d'affectation. Ainsi l'instruction `a = 3` affecte la valeur 3 à la variable a et ne renvoie rien tandis que `a == 3` teste si la variable a vaut 3 et renvoie `True` si c'est le cas et `False` sinon sans modifier le contenu de la variable a.

Ces opérateurs de test ont une priorité plus faible que les opérateurs de calcul (**, /, *, +, -) et sont donc évalués après les calculs. La priorité entre ces opérateurs est celle des lignes du tableau (le plus prioritaire en haut, le dernier évalué en bas).

Application 1 :
Pour chacune des expressions logiques suivantes, déterminer si elle vaut `True` ou `False` :

On précise l'état des variables : `a` vaut -4, `b` vaut 7, `c` vaut "abcdef" et `d` vaut False.

Expression logique	Valeur et commentaire éventuel
`b == 13`	
`a + 3 < 2`	
`'de' in c and 7 >= b`	
`True and not a < b`	
`b`[1] `or 'x' in c or a != 13`	

b. Principe de fonctionnement

Lorsque python rencontre une instruction `while`, il commence par évaluer la condition logique (qu'on appelle aussi *condition de poursuite de boucle*). Si elle vaut `False`, alors il saute tout le bloc d'instruction qui suit les '`:`' et continu le programme. Si elle vaut `True`, il exécute le bloc une première fois, puis il évalue à nouveau la condition de poursuite de boucle. Si elle vaut `False` il saute le bloc et continu l'exécution, si elle vaut `True`, il exécute à nouveau le bloc puis ré-évalue la condition, etc…

Remarque : Cela signifie que le bloc d'instruction du `while` peut éventuellement ne jamais être exécuté si la condition de poursuite est fausse dès le début.

Application 2 :

Exécuter le programme suivant en mode pas à pas.

1) Combien de fois la boucle est-elle exécutée ? .....
2) Changer la comparaison de la ligne 2 pour faire `i < 3`. Qu'est-ce qui est modifié ?
   .....
3) Même question avec `i >= 3`.
   .....

```
1 i = 0
2 while i <= 3:
3     print(i)
4     i = i + 1
5 print("Valeur finale :",i)
6
7
```

Remarque importante :

Ce type de boucle est très dangereux, car si les instructions à l'intérieur du bloc ne changent pas les termes de la condition de poursuite, on peut se retrouver avec le cauchemar du programmeur : une boucle infinie !

2) La boucle for

Ce type de boucle permet de répéter un bloc d'instruction ..... .

a. Principe de fonctionnement

Plus précisément, la boucle va s'effectuer pour chaque valeur d'une liste. Le nombre de répétitions est donc égal à la taille de la liste).
En python on parle de boucle sur un **itérateur** (l'itérateur fournit la liste que l'on va parcourir). L'itérateur le plus simple est l'itérateur `range` qui est intégré au langage.

`range(n)` génère une liste de tous les entiers de 0 à n-1 (il y a donc **n** valeurs et n doit être un entier)

Exemple : `range(5)` génère la liste `[0, 1, 2, 3, 4]`.

[1] En python un nombre peut être ramené lors d'un test à une valeur logique : il vaut `True` s'il est égal à 1 et `False` dans tous les autres cas.

La structure d'une boucle for est la suivante :

```
for element in iterateur :
      Bloc
      d'instruction
      à
      répéter
suite du programme
```

On remarque que le bloc à répéter est, là aussi, indenté par rapport au reste du code.
`element` est le nom de la variable qui va recevoir les valeurs successives renvoyées par l'`itérateur`. On essaye de choisir un nom de variable assez explicite, mais les programmeurs utilisent aussi assez souvent le nom de variable `i` (comme **i**ndex) car il est assez court.
`itérateur` est le nom de l'itérateur qui contient les différentes valeurs sur lesquelles on va boucler.

Application 3 :

Exécuter le programme suivant en mode pas à pas.

```
1 for i in range(3):
2     print(i)
```

1) Combien de fois la boucle est-elle exécutée ? .....
2) Changer la valeur de l'argument de `range` à la ligne 1. Qu'est-ce qui est modifié ?
   .....
3) Pour quelles valeurs de l'argument de range la boucle n'est-elle jamais exécutée ?
   .....

b. Précision sur les itérateurs

L'itérateur `range` prend ici un seul argument, mais il peut en prendre deux ou trois pour un contrôle plus précis de la liste générée :

- Avec deux arguments : le premier argument est la valeur de départ, le deuxième l'entier avant lequel s'arrêter.
  Exemple : `range(-2,4)` génère la liste `[-2, -1, 0, 1, 2, 3]`.
- Avec trois arguments : le premier argument est la valeur de départ, le deuxième l'entier avant lequel s'arrêter et le troisième argument donne le pas (l'incrément) entre deux valeurs successives.
  Exemple : `range(1,10,2)` génère la liste `[1, 3, 5, 7, 9]`.
  On peut utiliser une valeur négative pour le pas, ce qui générera la liste à l'envers. Dans ce cas, il faut bien sûr en premier argument donner la valeur la plus grande :
  `range(7,3,-1)` génère la liste `[7, 6, 5, 4]`
  `range(9,0,-2)` génère la liste `[9, 7, 5, 3, 1]`

Si les arguments sont mal choisis, la liste peut être vide : `range(3,3)` génère une liste vide (`[]`). Dans ce cas la boucle n'est jamais exécutée.

La fonction `list()` permet de générer une liste et ainsi de visualiser la suite générée par l'itérateur range. Dans la console python essayer `list(range(10))`. Vous pouvez ensuite essayer différents arguments pour la fonction range et visualiser directement le résultat.

De nombreux autres objets peuvent être des itérateurs en python. C'est le cas notamment des chaînes de caractères dont nous avons déjà parlé ou des listes que nous verrons plus tard.

```
1 chaine = "Dammarie-les-Lys"
2 for lettre in chaine:
3     print(lettre,'*', end="")
4 print()
```

3) Utilisation d'un accumulateur

Application 4 :

1) Ecrire un programme qui calcule la somme des 10 premiers entiers non-nuls. Vérifier qu'il donne bien le résultat attendu (55).
2) Modifier le programme pour qu'il calcule la somme des nombres entiers pairs jusqu'à 100 inclus. Vérifier qu'il donne bien le résultat attendu (2550).

Pour réaliser ce programme, on a du faire appel à une variable que l'on modifie à l'intérieur de la boucle à chaque répétition. Une telle variable est appelée **accumulateur** car elle accumule à chaque itération les calculs effectués.

III – Exercices d'application

Exercice 1 :

1) Ecrire un programme qui compte jusqu'à 10 en affichant à chaque fois les nombres comme indiqué ci-contre.

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

Aide : Les arguments `sep` *et* `end` *de la fonction* `print` *pourront être utilisés pour réaliser l'affichage demandé.*

2) Modifier le programme pour qu'il simule cette fois un compte à rebours de 10 jusqu'à 0 avec une seconde de pause entre chaque nombre.

Aide : La bibliothèque `time` *de python possède une fonction* `sleep()` *pour faire une pause dans le programme.*

Exercice 2 :

Ecrire un programme qui demande à l'utilisateur un nombre n de notes, puis demande de saisir les n notes et enfin affiche la moyenne de ces notes.

Exercice 3 :

Ecrire un programme avec la bibliothèque `turtle` qui dessine une spirale ayant l'allure ci-contre, avec un nombre de tours défini par l'utilisateur.

Pour être facilement dessinée tout en restant harmonieuse, cette spirale est constituée de demi-cercles dont les dimensions augmentent régulièrement : chaque demi-cercle a une épaisseur de un supérieur à l'épaisseur du précédent et un diamètre qui est le carré de cette valeur.

Aide :

La largeur du trait peut être modifiée dans `turtle` *avec la fonction* `width()` *:* `width(2)` *donne un trait d'épaisseur 2.*

On peut tracer facilement des arcs de cercle avec la fonction `circle(d, a)` *où* `d` *est le diamètre du cercle et* `a` *est l'angle de l'arc de cercle en degré (360 pour un cercle complet).*

Exercice 4 :

Ecrire un programme avec la bibliothèque `turtle` demandant un entier n à l'utilisateur et traçant un dégradé du noir au blanc de n niveaux de gris (noir et blanc compris).

Aide :

La couleur de trait et de remplissage peut être donnée en utilisant les composantes rouge, vert et bleue : `color(rouge, vert, bleu)` *où rouge, vert et bleu sont des nombres entre 0 (intensité nulle) et 1 (intensité maximale).*

Exemple : `color(0.3,0.3,0.3))` *choisit une couleur gris foncé (la même valeur sur les trois composantes donne du gris).* `color(0,0,0)` *donne du noir et* `color(1,1,1)` *donne du blanc.*