Les bases de la programmation en Python

I/ Variables

Pour stocker une information, il faut à la fois enregistrer cette information en mémoire et lui attribuer un nom afin de pouvoir y accéder à nouveau. En Python, ces deux opérations s'effectuent simultanément et on dit qu'on ***affecte une valeur à une variable*** ; la ***valeur*** est l'information et la ***variable*** est le nom qu'on lui attribue.

1. Affectation simple

Pour affecter la valeur 2 à la variable A, la syntaxe est :

```
1 A = 2
```

Remarques :

- Le symbole `=` n'est pas symétrique. La variable est à gauche ; la valeur est à droite.
- Le fait d'affecter une valeur à une variable ne provoque aucun affichage.

2. Affectations simultanées

Pour affecter la valeur 2 à la variable A et en même temps la valeur 4 à la variable B, la syntaxe est :

```
1 A , B = 2 , 4
```

Remarques :

- On utilise une virgule pour séparer les variables et les valeurs qu'on leur affecte. En Python, la virgule sert exclusivement à séparer des informations. Pour écrire un nombre décimal, on utilise un point.
- On peut également utiliser des parenthèses. Celles-ci sont facultatives et ne sont utilisées que si elles facilitent la lecture.

3. Echange de variables

L'affectation simultanée permet d'échanger facilement les valeurs affectées à deux variables. Par exemple, pour échanger les valeurs affectées aux deux variables A et B, la syntaxe est :

```
1 A , B = B , A
```

Il faut comprendre qu'à la variable A on affecte la valeur anciennement affectée à la variable B et que, simultanément, on affecte à la variable B la valeur anciennement affectée à la variable A.

4. Types

Toute variable a un ***type***, qui est déterminé automatiquement par Python et qui sert à décrire la nature de la variable. Le type d'une variable conditionne les opérations que l'on peut effectuer avec cette variable. Pour connaître le type d'une variable, on utilise la commande `type` .

A ce stade de l'année, les types à connaître sont : ***int*** (pour les nombres entiers), ***float*** (pour les nombres réels), ***bool*** (pour les booléens, c'est-à-dire des assertions) et ***tuple*** (pour les p-listes).

- Type int et float

Commande	Signification	Remarques
x + y	Somme de x et y	
x - y	Différence de x et y	
x * y	Produit de x et y	Il faut **toujours** écrire l'étoile de multiplication
x / y	Division de x par y	Le résultat d'une division est **toujours** de type float
x**y	x puissance y	
x % y	Reste de la division euclidienne de x par y	
x // y	Quotient de la division euclidienne de x par y	

- Type bool

Pour Python, l'assertion vraie est notée `True` et l'assertion fausse est notée `False` .
Les opérations mathématiques sur les assertions se traduisant en Python de la façon suivante :

Commande	Signification
not P	Négation de l'assertion P
P and Q	P et Q
P or Q	P ou Q

Par ailleurs, lorsque l'on effectue une comparaison entre deux variables, Python renvoie une réponse de type bool. Les comparaisons s'effectuent comme suit :

Commande	Renvoie True si ... et False sinon
x == y	les variables x et y ont la même valeur
x != y	les variables x et y n'ont pas la même valeur
x < y	la variable x a une valeur strictement inférieure à celle de y
x <= y	variable x a une valeur inférieure ou égale à celle de y
x > y	la variable x a une valeur strictement supérieure à celle de y
x >= y	la variable x a une valeur supérieure ou égale à celle de y

- Type tuple

Pour définir un tuple, la syntaxe est la suivante :

```
u = (10 , 12 , 13 , 22)
```

Les éléments d'un tuple sont numérotés à partir de 0 ; le numéro d'un élément est appelé sont ***indice***. Pour accéder à un élément, la syntaxe est la suivante :

```
u[0] # pour acceder au 10
u[1] # pour acceder au 12
u[2] # pour acceder au 13
u[3] # pour acceder au 22
```

II/ Fonctions

En informatique, une ***fonction*** est une suite finie d'instructions qui reçoit des ***arguments*** (aussi appelés ***paramètres***) et ***renvoie*** un ***résultat***.

1. Définir une fonction avec Python

Pour définir une fonction avec Python, la syntaxe est la suivante :

```
'''Ligne de commentaire qui explique le role de la fonction'''

def fonction(arg1,arg2,...,argn):
    instruction 1 # Commentaire facultatif
    instruction 2
    ...
    instruction p
    return resultat
```

Remarques

- La première ligne de la définition d'une fonction commence par le mot clé **def** et se termine par deux points **:** .
- Toutes les instructions sont ***indentées*** par rapport à la première ligne : elles forment ce qu'on appelle un ***bloc***.
- Entre deux instructions, on passe à la ligne.
- Le mot clé **return** sert à renvoyer le résultat de la fonction. Après return, la fonction ne lit plus rien dans le bloc ; toutes les instructions indentées qui suivent sont ignorées.

2. Complément : valeur par défaut

Lorsque l'on crée une fonction, on peut fixer des valeurs par défaut pour ses arguments. Dans ce cas, lorsque l'on fait appel à la fonction, si l'on ne spécifie pas la valeur à donner à ces arguments, ces arguments prennent leurs valeurs par défaut. Pour la syntaxe, on pourra se reporter à l'exemple ci-dessous :

```
def somme(i,j=10) :
    r = i + j
    return r
```

Dans cet exemple :

- L'instruction "somme(1,4)" renvoie 5.
- L'instruction "somme(1)" renvoie 11.

3. Complément : variables locales et variables globales

A l'intérieur d'une fonction, Python peut utiliser une variable qui a été définie de manière globale. En revanche, une variable définie à l'intérieur d'une fonction n'a de sens qu'à l'intérieur de cette fonction.

```
A = 10

def fonction(x,y) :
    s = x + y
    d = x - y
    a = A * x * y
    return s , d , a
```

Dans cet exemple :

- L'instruction "fonction(1,2)" renvoie (3 , -1 , 20).
- L'instruction "print(A)" affiche 10.
- L'instruction "print(s)" renvoie un message d'erreur.

III/ Instructions conditionnelles

1. Test simple

La syntaxe de l'instruction conditionnelle en Python reprend exactement la structure conditionnelle mathématique.

En mathématiques	En Python
Si *condition*, Alors *instructions si la condition est vraie* Sinon *instructions si la condition est fausse*	**if** condition **:** instructions si la condition est vraie **else :** instructions si la condition est fausse

Le mot clé **if ... :** teste une condition. Si cette condition est vraie, le bloc suivant est exécuté ; sinon c'est le bloc après le mot-clé **else :** qui est exécuté.

C'est l'indentation qui détermine les blocs ; tous les éléments au même niveau d'indentation font partie du même bloc et le bloc s'arrête lorsqu'on retourne à une niveau d'indentation inférieur.

La partie **else :** n'est pas obligatoire ; il peut ne pas y avoir de "sinon".

Pour la syntaxe, on pourra se reporter à l'exemple ci-dessous :

```
1 s = input("Entrer un nombre entier :")
2 n = int(s)
3
4 if n%2 == 0 :
5     print("L'entier est pair.")
6 else :
7     print("L'entier est impair.")
```

2. Tests imbriqués

Il est possible en Python d'imbriquer des instructions conditionnelles les unes dans les autres. Ce faisant, il faut faire attention à bien respecter l'indentation des blocs pour que les instructions soient correctement exécutées. On pourra se référer à l'exemple ci-dessous :

```
1 def test_diviseurs(n):
2     if n%2 == 0 :
3         if n%5 == 0:
4             print("L'entier est divisible par 10.")
5         else :
6             print("L'entier est divisible par 2 mais pas par 5.")
7     else :
8         print("L'entier n'est pas divisible par 2.")
```

3. Elif

Lorsque l'on écrit une instruction conditionnelle, il est possible de proposer plusieurs alternatives via le mot-clé **elif**. Cette commande se traduit par "Sinon, si...". La syntaxe est la suivante :

if condition 1 **:**
    instructions
elif condition 2 **:**
    instructions
elif condition 3 **:**
    instructions
else :
    instructions
suite du programme en dehors de la structure conditionnelle

Dès que l'une des condition est vérifiée (ce qui correspond à une condition qui prend la valeur *True*), le bloc correspondant est exécuté puis le programme sort de la structure conditionnelle sans tester si les conditions suivantes sont vraies.

Si aucune des conditions n'est vérifiée, alors le programme exécute le bloc correspondant au **else**.

Cette syntaxe a l'avantage d'éviter d'imbriquer trop de structures conditionnelles les unes dans les autres.

IV/ Boucles while

1. Syntaxe

La syntaxe de la boucle conditionnelle en Python reprend exactement la structure de l'algorithme associé.

Algorithme	En Python
Tant que *condition*	**while** condition **:**
Faire *instructions*	instructions si la condition est vraie
Fin	
Suite de l'algorithme	suite du programme

Le mot clé **while ... :** teste une condition. Tant que cette condition est vraie, le bloc suivant est exécuté ; lorsque la condition devient fausse, Python sort du bloc et exécute la suite.

C'est l'indentation qui détermine le bloc d'instructions à exécuter lors d'une itération de la boucle.

2. Exemple 1

Que fait le programme ci-dessous ?

```
1 def compter(n) :
2     k = 0
3     while k <= n :
4         print(k)
5         k += 1
```

3. Boucles infinies

Pour que le programme se termine, il faut que la condition testée devienne fausse après un nombre fini d'itérations de la boucle. Il faut donc que la condition évolue à chaque itération par le biais d'une variable que l'on modifie. Il faut toujours faire attention à écrire des boucles while qui se terminent.

4. Exemple 2

Ecrire une fonction Python **reste_division_euclidienne** qui prend en arguments deux entiers `a` et `b` et qui renvoie le reste dans la division euclidienne de `a` par `b`.

On s'interdira d'utiliser la commande Python %.

V/ Boucles for

1. Syntaxe

La syntaxe de la boucle for en Python reprend la structure de l'algorithme associé.

Algorithme	En Python
Pour k allant de *début* à *fin* Faire *instructions* Fin Suite de l'algorithme	**for** k **in range**(début,fin+1) **:** instructions suite du programme

Le mot clé **for ... :** indique le début de la boucle for.
C'est l'indentation qui détermine le bloc d'instructions à exécuter lors d'une itération de la boucle.

2. Concernant la commande "in range"

Il faut faire attention à utiliser la commande **in range** correctement. Celle-ci peut être utilisée de plusieurs manières, en spécifiant ou non le début, et en prenant éventuellement en compte un pas. Mais, dans tous les cas, il faut toujours faire attention aux bornes : la première est incluse et l'autre est exclue. Les cas d'utilisations sont résumés dans le tableau ci-dessous :

Syntaxe Python	Interprétation
for i in range(n)	i va de 0 (inclus) à n (exclu)
for i in range(p,n)	i va de p (inclus) à n (exclu)
for i in range(p,n,k)	i va de p (inclus) à n (exclu) avec un pas de k

3. Exemple

1. Ecrire une fonction Python **compter_2** qui prend en argument un nombre entier n et qui affiche tous les entiers compris au sens large entre 0 et n.

2. Ecrire une fonction Python **compter_3** qui prend en argument un nombre entier n et qui affiche tous les entiers pairs compris au sens large entre 0 et n.