Chapitre I : Programmation en python

I - Introduction

a) Les variables

Les variables sont des mots (ou identifiants) qui désignent des objets. Elles peuvent être de différents types : int (pour les entiers), float (pour les nombres en virgule flottante - nombres réels non entiers), str (pour les chaînes de caractères), bool (pour les booléens `True` ou `False`) et d'autres types qui seront étudiés au cours de l'année.
Pour connaître le type de la variable `a`, on utilise dans python la syntaxe `type(a)`.

Le nom d'une variable commence toujours par une lettre (minuscule le plus souvent, la majuscule étant réservée par convention à des objets particuliers) suivie éventuellement de caractères parmi les lettres, les chiffres ou le _ (underscore).

Remarques :

- il est vraiment préférable d'éviter les caractères spéciaux ou accentués dans un nom de variable ;
- il ne faut pas choisir un mot clé utilisé par python (en général l'éditeur l'écrit d'une autre couleur ce qui nous évite l'erreur).

Parcourir un algorithme (écrit dans n'importe quel langage) consiste à déterminer les valeurs successives prises par chaque variable. Cela se présente facilement sous la forme d'un tableau :

Exemple : parcourir le programme ci-dessous :

```
1 a=12
2 b=a-3
3 a=2*a
4 b=a-b
5 a=(a-1)//2
```

a	b
12	9
24	15
11	

À la fin de ce programme, `a` vaut 11 et `b` vaut 15.

Remarque : en python, `//` donne le quotient entier dans le cas d'une division euclidienne ; avec des entiers positifs, `a//b` revient au même que `int(a/b)`.

b) Fonctions de base

Nous avons déjà vu la fonction `type`, mais python possède de très nombreuses fonctions qui seront vues au fil de l'année. En voici deux nouvelles :

- `print` : elle permet d'afficher dans la console tout ce qui sera listé entre parenthèses (du texte, une ou des variables séparées par une virgule).

  ```
  1 a=5
  2 b=3
  3 print("Le produit vaut", a*b)
  ```

  Ce programme affichera le texte « Le produit vaut 15 ».

  Cette fonction possède deux arguments optionnels qui doivent être nommés pour être utilisés :

  - `sep` : définit un séparateur dans l'affichage de plusieurs éléments (par défaut, il vaut `" "`) ;
  - `end` : définit le caractère utilisé à la fin du texte affiché, ce qui serait utile pour des affichages successifs (par défaut, il vaut `"\n"` ce qui représente un retour à la ligne).

  `print("Le produit vaut", a*b, sep=" ", end="\n")` donnera le même résultat.

- `input` : elle permet une interaction avec l'utilisateur. L'appel de cette fonction provoque l'interruption du programme ; l'utilisateur est invité à saisir quelque chose au clavier et le programme reprend au clic sur le bouton "Entrer". On stocke en principe ce qui est saisi dans une variable :

  ```
  saisie=input("Entrer un caractère:")
  ```

  Attention : il est courant de demander la saisie d'un nombre, or input ne permet de récupérer qu'une chaîne de caractères ; il faudra donc la convertir en entier pour poursuivre le traitement :

  ```
  nombre=int(input("Entrer un nombre entier:"))
  ```

Exercice : Quel affichage obtient-on à la fin de chacun de ces quatre programmes :

Programme 1

```
a=5
b=2*a
a=b-2
print(a,b)
```

Réponse : 8 10

Programme 2

```
a=-2
a=3*a+4
b=a**2
print("réponse:", b)
```

Réponse : réponse 2

Programme 3

```
mot1="ta"
mot2=mot1*3
mot3=mot1+"re"
print(mot2, mot2[3], mot3[-1])
```

Réponse : tatata a e

Programme 4

```
c=5
c+=1
c*=3
print(c)
```

Réponse : 18

c) Importer un module

Nous aurons besoin au fil de l'année de fonctions qui ne sont pas présentes dans la version de base de python. Elles sont présentes dans des bibliothèques appelées **modules**. En voici quelques exemples :

- math : pour les fonctions mathématiques (comme sqrt qui permet de calculer la racine carrée d'un nombre positif) ;
- random : permet de générer des nombres aléatoires ;
- matplotlib : permet de construire des graphiques 2D ;
- PIL : permet de manipuler des images.

L'utilisation d'une bibliothèque peut se faire de 3 façons (j'ai pris le mot générique `module` qui pourra être remplacé par `math`, `random`, ...) :

- `import module` : le code d'appel de la fonction est alors sous la forme : `module.fonction()` ;
- `from module import fonction` : le code d'appel de la fonction est alors sous la forme : `fonction()` ;
- `from module import *` : le code d'appel de la fonction est alors sous la forme : `fonction()`.

Remarque : la dernière syntaxe n'est pas très judicieuse pour des projets relativement importants car on importe toutes les fonctions du modules, sans connaître la liste de ces fonctions. Il pourrait ainsi arriver qu'on redéfinisse une fonction portant le nom de l'une d'elles ce qui pourrait générer un bug très difficile à déceler.

d) Instruction conditionnelle

Une instruction conditionnelle permet de n'exécuter une instruction que si une condition est remplie. Elle peut se présenter sous trois formes :

```
if condition :
    instructions
```

```
if condition :
    instructions1
else :
    instructions2
```

```
if condition1 :
    instructions1
elif condition2 :
    instructions2
else :
    instructions3
```

Dans la première forme, rien ne se passe si la condition n'est pas vérifiée. La troisième forme permet d'enchaîner différents traitements suivant les conditions à vérifier (il peut y en avoir plus de deux).

IMPORTANT : il ne faut jamais oublier les deux-points à la fin des lignes du if, elif ou else ainsi que l'indentation (décalage vers la droite) qui indique toutes les instructions à effectuer.

Exemple : Qu'effectue le programme suivant :

```
import random
de = random.randint(1,6)
if de == 6:
   print("Gagné")
else :
   print("Perdu")
```

Cette fonction réalise un tirage aléatoire d'un entier compris entre 1 et 6 (pour simuler un dé cubique bien équilibré). Elle affiche "Gagné" si le dé tombe sur 6, "Perdu" sinon.

Remarque : le test d'une égalité s'effectue avec le symbole "==", le signe "=" traduisant exclusivement une affectation.

II - Créer ses propres fonctions

Les fonctions sont des portions de code auxquelles on a donné un nom. On utilise ensuite ce code juste avec son nom. Cela présente deux avantages :

- rendre le code plus lisible : en nommant correctement une fonction (et en la commentant), on comprend son utilité et le code dans lequel on l'utilise devient plus court ce qui facilite sa lecture ;
- utiliser plusieurs fois un même code, la fonction pouvant être appelée plusieurs fois sans avoir besoin de copier plusieurs lignes.

Définition : Définir une fonction revient à lui donner un nom, zéro, un ou plusieurs arguments puis donner le code de la fonction avec le plus souvent une ou plusieurs valeurs qui seront renvoyées.

Remarques :

- la documentation d'une fonction peut paraître fastidieuse mais elle permet une meilleure compréhension de son utilité, y compris pour son auteur qui souhaiterait la réutiliser plus tard ;
- il est possible d'indiquer le type des arguments ainsi que celui des valeurs renvoyées ;
- il est possible de fournir des tests vérifiant le bon fonctionnement de la fonction.

Exemple : On crée une fonction permettant de calculer le prix TTC d'un produit dont on connaît la valeur hors taxe et le taux de TVA :

```
def prix(ht, tva):
   ttc = ht*(1+tva/100)
   return ttc
```

Cette fonction répond bien au problème, le nom de la fonction ainsi que celui des variables utilisées (y compris les arguments) facilitant la compréhension.
On peut l'utiliser sous la forme :
`achat = prix(16, 5.5)` qui donne à la variable `achat` la valeur 16.88
ou bien plusieurs fois :
`achats = prix(30, 5.5) + prix(12, 20)` qui donne à la variable `achats` la valeur 46.05.

Nous allons toutefois la compléter pour qu'elle soit plus conforme à ce qu'on attend :

- la documentation : elle doit être placée au début de la fonction entre des triples guillemets. Par exemple :

  ```
  """cette fonction calcule le prix TTC d'un objet de valeur ht auquel on applique un taux tva"""
  ```

- le typage des arguments : cette partie est optionnelle, python ne vérifiant pas si le typage est respecté lors de l'appel.

  Dans cet exemple, les arguments et la valeur renvoyée sont du type flottant, on peut le préciser ainsi :

  ```
  def prix(ht: float , tva: float ) -> float :
  ```

– les tests qui peuvent être placés au début pour vérifier que les arguments sont acceptables (on appelle ceci les pré-conditions) ou en dehors de la fonction pour tester les valeurs renvoyées (c'est ce que appelle les post-conditions).

Cela se fait en python avec le mot clé `assert`. Par exemple, au début de la fonction, on peut ajouter :

```
assert tva>=0, "Le taux de TVA doit être positif"
```

Si lors de l'appel de la fonction `prix`, le deuxième argument est négatif, alors python renverra une erreur avec le message écrit entre guillemets.

Pour les post-conditions, on peut proposer un test en dehors de la fonction :

```
assert prix(20, 3)==20.6, "Le résultat renvoyé n'est pas le bon"
```

Cette ligne ne renverra rien si le test est correct.

Voici le code complet de cette fonction.

```
def prix(ht: float, tva: float) -> float:
    """cette fonction calcule le prix TTC d'un objet de valeur ht auquel on applique un
    taux tva"""
    assert tva>=0, "Le taux de TVA doit être positif"
    ttc = ht*(1+tva/100)
    return ttc

assert prix(20, 3)==20.6, "Le résultat renvoyé n'est pas le bon"
```

III - Les boucles

a) Boucles bornées

Une boucle est la répétition d'une partie du code. Lorsque le nombre maximal de répétitions est connu à l'avance, on peut utiliser une boucle bornée ou boucle `for`.
La syntaxe générale de ce type de boucle en python est de la forme `for elt in` *iterable* :
où `elt` prendra successivement les différentes valeurs de *iterable*.

`range` est un itérable propre à python permettant d'obtenir une suite de nombres entiers. Elle peut s'utiliser avec 1, 2 ou 3 arguments :

- `range(m)` : on obtient tous les entiers de 0 à `m-1` ;
- `range(m, n)` : on obtient tous les entiers de `m` à `n-1` ;
- `range(m, n, p)` : on obtient tous les entiers de `m` à `n-1` avec un pas de `p`.

Exemple : Qu'affichent les programmes suivants ?

```
a=3
for i in range(3):
    a=2*a
print(a)
```

Réponse : 24

```
a="z"
for i in range(2,6):
    b=a*i
    print(b, end=";")
```

Réponse : zz ;zzz ;zzzz ;zzzzz ;

```
c=5
for i in range(5, 21, 3):
    print(c*i, end=" ")
```

Réponse : 25 40 55 70 85 100

Une chaîne de caractères est un autre itérable. On peut donc la parcourir caractère par caractère.
Exemple : Que renvoie la fonction suivante ? La renommer et ajouter des éléments pour la rendre plus conforme à ce qu'on attend (comme dans la partie 2).

```
def fct_mystere(chaine):
    nb=0
    for lettre in chaine:
        if lettre == "a" or lettre == "A":
            nb += 1
    return nb
```

Cette fonction compte le nombre de fois où la lettre "a" apparaît dans la chaîne (que ce soit en minuscule ou en majuscule).
Un nom plus judicieux de cette fonction pourrait être `nombre_de_a`. De plus, dans cette fonction, il manque la documentation et la nature de l'argument ainsi que de la valeur renvoyée.
On peut aussi vérifier que l'argument donné à cette fonction est bien du type str, ce qui peut être fait avec la fonction `isinstance`.
On peut enfin réduire l'instruction conditionnelle en mettant la lettre en majuscule et en vérifiant si elle est égale à "A". Cela se fait à l'aide de la méthode `upper()` (attention à son utilisation car ce n'est pas tout à fait une fonction).

```
def nombre_de_a(chaine: str) -> str:
    """cette fonction compte le nombre de fois que la lettre "a" (minuscule ou majuscule)
    apparaît dans la chaîne"""
    assert isinstance(chaine, str), "il nous faut une chaîne de caractères"
    nb=0
    for lettre in chaine:
        if lettre.upper() == "A":
            nb += 1
    return nb
```

b) Boucle non bornée

Dans certains cas, on ne sait pas combien de fois on aura à répéter une instruction. La boucle `while` permet alors de répéter du code tant qu'une condition reste vérifiée.
La syntaxe générale de ce type de boucle en python est de la forme `while` *condition* :

> Dans l'utilisation d'une boucle `while`, il est très important que la condition ne soit plus vérifiée à un moment, sans quoi le programme générerait une boucle infinie.

Exemple : lancement d'un dé tant qu'on ne tombe pas sur 6.
On compte alors le nombre de coups nécessaires pour obtenir le premier 6. Cela doit rappeler un programme de la partie I - d).

```
import random
def tentatives() -> int:
    """cette fonction renvoie le nombre de tentatives nécessaires pour obtenir un 6 dans
    le cas du jet d'un dé bien équilibré"""
    de = random.randint(1,6)
    nb = 1
    while de != 6:
        de = random.randint(1,6)
        nb += 1
    return nb
```

Même si à chaque lancer, rien ne nous garantit qu'on tombe sur 6, la probabilité de ne pas tomber sur un 6 au bout de n coup devient tellement faible quand n augmente qu'on est convaincu qu'on sortira de la boucle à un moment.

Exemple : Doublement d'un capital.

```
def doublement_capital(capital: float, taux: float) -> int:
    """cette fonction renvoie le nombre d'année nécessaire au doublement d'un capital si
    celui-ci augmente de taux % chaque année"""
    annee = 0
    c = capital
    while c < capital*2:
        c = c*(1+taux/100)
        annee += 1
    return annee
```

`capital` est le capital de départ, `c` sera le capital obtenu au bout d'un certain nombres d'années. Ces deux variables sont indispensables pour effectuer la comparaison.