Programmer avec Python
Cours

Séance n° 1	**Révisions d'algorithmique**

L'objectif de cette première séance est de réviser les notions abordées l'an dernier dans le module Programmer avec Python et plus précisément aujourd'hui les notions de variables, de conditions, les boucles et fonctions. Commençons par revoir les outils nécessaires pour écrire un programme en Python.

1 Utiliser Python, à l'école ou chez vous

1.1 Environnement de programmation

Un environnement de programmation, Integrated Development Environment (IDE) en anglais, est un logiciel qui permet de gérer tous les aspects du développement d'un programme : édition du code source, compilation si nécessaire, exécution. Pour programmer avec Python, de nombreux IDE sont disponibles. Nous allons nous utiliser l'IDE Spyder disponible sur les ordinateurs de l'école dans la distribution Anaconda.

1.2 Démarrage d'Anaconda et Spyder à l'école

Pour démarrez l'IDE Spyder, double cliquez sur le raccourci d'Anaconda sur le bureau puis lancez le logiciel Spyder.

1.3 Chez vous

Pour vous entrainer chez vous, il existe des IDE en ligne qui vous permettront d'exécuter du code Python sans rien installer sur votre ordinateur. Par exemple : https://replit.com/languages/python3. Vous pouvez aussi installer la distribution Anaconda sur votre ordinateur comme à l'école : https://www.anaconda.com/products/individual-d.

1.4 Ressources

Si vous voulez approfondir ce cours ou si vous souhaitez un autre éclairage, vous pouvez consulter *Apprendre à programmer avec Python 3* de Gérard Swinnen, librement accessible en PDF à cette adresse : https://inforef.be/swi/download/apprendre_python3_5.pdf.

Vous pouvez également vous aider de la fiche de syntaxe de 1A disponible sur l'ENT.

1.5 Conseils !

Enregistrez bien votre travail au fur et à mesure pendant les séances et organisez vos programmes clairement, séance par séance, avec des répertoires, afin de savoir vous repérer facilement. Nous ne vous demanderons pas de déposer votre travail sur l'ENT en fin de séance cette année, à vous d'être rigoureux.

2 Les variables

On utilise une variable dans un programme pour y stocker des informations. Une variable porte plusieurs caractéristiques :

- Son **nom**, qu'on appelle aussi **identificateur**
- Son **adresse** dans la mémoire de l'ordinateur
- Sa **valeur**
- Son **type** qui est déterminé par le type des valeurs stockées dans cette variable (entier : `int`, réel : `float`, chaîne de caractères : `str`)

On utilise le signe d'affectation = pour attribuer une valeur à une variable ou modifier cette valeur ensuite. Pour afficher le contenu d'une variable, on utilise la fonction **print**. Un exemple :

```
nombre = 5 # La variable nombre vaut 5. Elle est donc de type
    entier
print(nombre) # J'affiche à l'écran la valeur de la variable nombre

nombre = 10 + 3 # Je modifie la valeur de la variable nombre

nombre = nombre + 1 # J'ajoute encore 1 à ma variable
print(nombre) # J'affiche à l'écran la valeur de la variable nombre
```

Ce programme affichera d'abord 5, puis 14 !

Python permet aussi de taper des valeurs au clavier et de les affecter à une variable avec la fonction **input**.

```
prenom = input("Tapez votre prénom : ") # La valeur saisie est
    stockée dans prenom
print("Bonjour ", prenom) # On l'affiche. Attention aux guillemets.
effectif = int(input("Tapez l'effectif : ")) # Je saisis un nombre,
    donc je dois convertir le type avec int !
print("Vous avez", effectif, "élèves.")
```

3 Les conditions

L'exécution d'un programme peut dépendre de conditions qui permettront si elles sont vraies d'exécuter une instruction. En Python, ces tests sont réalisables avec les structures **if**, **elif** et **else**. Le type permettant de représenter les valeurs Vrai ou Faux est le type booléen et les expressions qui sont testées sont ainsi appelées expressions booléennes. Exemple :

```
moyenne = float(input("Moyenne générale ?"))
if moyenne >= 10: # Si la moyenne est supérieure ou égale à 10
    print("Bienvenue à l'école !")
elif moyenne >= 9: # Ou sinon si la moyenne est supérieure ou égale
    à 9
    print("Il ne vous manque que", 10-moyenne, "point(s), vous
        allez au rattrapage !")
else: # Sinon dans tous les autres cas
    print("Vous redoublez, vous ferez mieux l'an prochain !")
```

Notez bien l'usage des " :" à la fin des conditions. Vous avez également un exemple ici pour vous souvenir de l'importance de l'indentation en Python. L'indentation désigne l'espacement de la ligne à partir de la gauche. C'est l'indentation qui définit un bloc de code et son exécution.

```
nombre = 5
if nombre > 10:
    print("Le nombre est grand")
    print("Bonjour") # Les deux print sont exécutés seulement si la
        condition est vraie
```

```
if nombre > 100:
    print("Le nombre est très grand")
print("Bonjour") # Cette ligne ne fait pas partie du bloc de
   condition, elle sera toujours exécutée !
```

4 Les boucles

Dans un programme, on souhaite souvent répéter une action un certain nombre de fois. Pensez au programme qui calcule les moyennes de tous les élèves de la promo ! Ce serait une grosse perte de temps d'écrire le calcul pour chacun des 200 élèves. Pour accélérer la tâche, les langages de programmation proposent des structures qu'on appelle boucles. En Python, il existe deux instructions pour construire ces boucles :

— L'instruction **while**
— L'instruction **for**

4.1 L'instruction `while`

Cette instruction permet de répéter continuellement un bloc d'instructions tant qu'une condition est respectée. Exemple :

```
a = 0
while a < 5:
    a = a + 1
    print(a)
```

Comme dans les conditions, l'instruction while amorce un autre bloc d'instructions, le bloc qui sera répété. Le double point à la fin de la ligne introduit le bloc d'instructions à répéter, lequel doit obligatoirement être indenté.

Cette boucle **while** indique donc qu'il faut répéter continuellement le bloc indenté, tant que la condition `a < 5` est vraie. Notez bien que la variable qui est évaluée dans la condition, ici `a` doit exister avant l'évaluation de la boucle. On doit lui avoir affecté une valeur au préalable. Mais comment ça fonctionne ? Python va commencer par évaluer la validité de la condition fournie, ici `a < 5` puis

— Si la condition est fausse alors tout le bloc indenté qui suit est ignoré et l'exécution se termine
— Si la condition est vraie alors Python exécute le bloc indenté, le corps de la boucle :
  — L'instruction `a = a + 1` augmente d'une unité le contenu de la variable `a` (on dit incrémentée). On peut écrire aussi plus rapidement `a += 1`.
  — La valeur de `a` est affichée avec la fonction **print**
— Une fois ces deux lignes exécutées, l'exécution reprend à la ligne du while. La condition est à nouveau vérifiée et ainsi de suite, le bouclage se poursuit.

Dans cette boucle, la valeur de `a` est donc augmentée d'une unité à chaque répétition.

À la cinquième itération, `a` vaut 5. Lorsque la condition `a < 5` est à nouveau vérifiée, elle est donc fausse, et la boucle s'arrête. Le programme aura ainsi affiché 1, puis 2, puis 3, puis 4, puis 5 et s'arrête. Vous pouvez le tester vous-même en recopiant ce code dans Spyder.

Notez que si la condition est fausse au départ, la boucle ne sera jamais exécutée. Au contraire, si la condition demeure toujours vraie, alors la boucle sera exécutée indéfiniment !

4.2 L'instruction `for`

4.3 Parcours d'une séquence

La boucle **for** a le même objectif que la boucle while : répéter une série d'opérations un certain nombre de fois. La boucle **for** est spécialisée dans une opération très fréquente en programmation : le parcours d'une séquence qui peut être un mot, une liste, une suite de nombres.

Parcours d'un mot

```
mot = "bonjour" # On définit la variable mot
for lettre in mot: # Pour chaque lettre dans mot, successivement
    print(lettre) # On affiche la lettre
```

La boucle parcourt ainsi chaque lettre du mot.

Parcours d'une séquence d'entiers avec `range`

La boucle **`for`** permet également de parcourir des séquences de nombres grâce à la fonction **`range`**. Par exemple, si on veut afficher chaque entier entre 1 et 50 :

```
for entier in range(1, 51): # Pour chaque entier entre 1 et 50
    print(entier) # On affiche l'entier
```

Vous noterez que j'ai écrit **`range`**`(1, 51)` pour générer les entiers entre 1 et 50, 51 est donc exclu ! Par défaut, si on n'indique que la valeur de début et la valeur de fin, la suite de nombres va de 1 en 1 mais nous pouvons préciser le pas, en troisième dans les parenthèses, comme ceci :

```
for entier in range(0, 10, 2):
    print(entier)
```

Les nombres affichés seront : 0, 2, 4, 6, 8.

```
for entier in range(10, 5, -1):
    print(entier)
```

Les nombres affichés seront : 10, 9, 8, 7, 6

Comment choisir entre `while` et `for` ?

On choisira d'utiliser une boucle **`for`** lorsqu'on connaît le nombre d'itérations à effectuer à l'avance. Par exemple, on sait qu'on veut effectuer 10 répétitions, on utilisera donc **`for`** `i` **`in`** **`range`**`(...)` ou on veut parcourir une chaîne que l'on connaît sur toute sa longueur, on utilisera donc **`for`** `i` **`in`** `lachaine`. Lorsque le nombre d'itérations repose sur une condition et qu'on ne sait pas à l'avance combien d'itérations seront nécessaires, on utilisera la boucle **`while`**. Par exemple, on veut faire une boucle qui s'arrête lorsque qu'un nombre calculé dans la boucle devient supérieur à 10000, ou quand un booléen devient True, ou False.

5 Les fonctions

Nous avons déjà revu quelques fonctions intégrées à Python comme **`print`**, **`input`** ou **`range`**. Le rôle d'une fonction en programmation est de réaliser une tâche prédéfinie. Par exemple, la fonction **`print`** affiche un texte à l'écran. Vous aurez bien sûr remarqué que le message en question varie. Si j'écris **`print`**`("Bonjour")`, bonjour s'affichera. Si j'écris **`print`**`("Au revoir")` c'est Au revoir qui s'affichera. Le résultat de la fonction s'adapte donc à la valeur donnée dans les parenthèses. Cette valeur est nommée **paramètre**. Une même fonction peut ainsi être utilisée sans avoir à réécrire le moindre code pour réaliser une même tâche, un même calcul pour des paramètres différents.

Vous savez donc utiliser les fonctions intégrées à Python, voyons maintenant comment écrire vos propres fonctions. Imaginons, vous êtes en train de développer un jeu et vous avez besoin de calculer l'aire de cercles très souvent pour dessiner des structures dans votre jeu. Écrire le calcul de l'aire d'un cercle n'est pas très compliqué, il suffit d'appliquer la formule πr^2 mais après 10-20 fois vous risquez de vous lasser. Écrivons alors une fonction qui fera ce travail à notre place.

```
# Définition de la fonction aireCercle qui calcule l'aire d'un
    cercle de rayon rayon
def aireCercle(rayon):
    return 3.14*rayon*rayon
```

Nous avons ici défini avec le mot-clé **def** une nouvelle fonction appelée `aireCercle`, qui accepte un seul paramètre appelé `rayon`. Cette fonction donnera comme résultat, **retournera**, le résultat du calcul de l'aire d'un cercle. On pourra ainsi utiliser cette fonction pour calculer l'aire de n'importe quel cercle dont on lui donnera la valeur de rayon. Voyons maintenant comment l'utiliser :

```
# Définition de la fonction aireCercle qui calcule l'aire d'un
    cercle de rayon rayon
def aireCercle(rayon):
    return 3.14*rayon*rayon

# Programme principal (toujours après les fonctions)
# Premier cercle
rayon1 = 5
aire1 = aireCercle(rayon1) # On appelle la fonction pour qu'elle
    calcule l'aire avec rayon1 et on stocke le résultat dans la
    variable aire1
print(aire1)

# Un deuxième cercle
rayon2 = 1.5
aire2 = aireCercle(rayon2) # On appelle la fonction pour qu'elle
    calcule l'aire avec rayon2 cette fois et on stocke le ré
    sultat dans la variable aire2
print(aire2)
```

Et on peut ainsi réutiliser la fonction autant de fois que souhaité.

Dans notre exemple, la fonction `aireCercle` n'accepte qu'un seul paramètre. Vous pouvez tout à fait définir des fonctions demandant plusieurs paramètres (ils seront séparés par des virgules) ou même aucun paramètre (ne jamais oublier alors d'écrire les parenthèses vides).