TP1a - Bases de Python

1 Instructions élémentaires

Un programme Python est une suite d'**instructions**. Ce sont des ordres élémentaires que l'ordinateur est capable d'exécuter. Par défaut, les instructions sont exécutées les unes après les autres dans l'ordre.
Pour nous, il y a deux instructions de bases à connaître et à maitriser :

- **L'affectation** :

```
variable = expression
```

L'affectation s'écrit avec le signe =. À gauche du signe, on écrit un nom de **variable** (nouvelle ou déjà existante) dans laquelle on désire stocker le résultat d'un calcul ou d'une opération. À droite, on met une **expression** c'est-à-dire une formule qui a une valeur (qui peut être un nombre, une chaine de caractère ou encore un booléen). Par exemple :

```
x = 2
v = x**2 - 2*x + 5
```

Une affectation peut utiliser une variable déjà existante dans l'expression à droite et modifier ensuite cette même variable. Par exemple :

```
u = 1
u = 2*u - 5
# u vaut alors -3
```

- **La fonction `print` :**

```
print("Hello, World!")
```

La fonction **`print`** est intégré au langage Python. Elle permet d'afficher du texte. Dans son utilisation la plus simple, on place entre parenthèse le texte à afficher :

```
print("Bonjour les ECG2B !")
```

Ce texte est une chaine de caractère et doit donc être écrit entre guillemets. On peut cependant utiliser la conversion automatique de Python pour afficher des nombres et des résultats de calculs :

```
x = 2
print(x) # Affiche "2"
```

Utilisation avancée : `print` accepte une chaîne de caractères. N'importe quelle expression donnant une chaîne de caractères est possible. On peut utiliser la méthode **`format`** sur une chaîne de caractères pour donner des résultats intéressants :

```
r = 42
print("La reponse est {}.".format(r)) # Affiche "La reponse est 42."
```

La fonction **`format`**, qui doit être utilisée comme dans l'exemple, remplace toutes les occurences de {} par les paramètres qui lui sont données entre parenthèses, et ce dans l'ordre.

```
r = 42
prof = "Charles"
print("La reponse est {}. C'est M.{} qui me l'a dit.".format(r,prof))
# Affiche "La reponse est 42. C'est M.Charles qui me l'a dit."
```

Exercice 1

Écrire une suite d'instructions qui calculent la valeur de $x^3 - 2x^2 + 5x - 12$ *pour* $x = 2$ *puis affichent la valeur obtenue à l'écran.*

2 Les fonctions

La langage `Python` permet de définir des fonctions. En informatique et plus particulièrement en `Python`, les fonctions sont des suites d'instructions que l'on rassemble et auxquelles on donne un nom afin de pouvoir les réutiliser facilement.

Une fonction commence par un en-tête de la forme suivante :

```
1 def nom_de_la_fonction(parametres):
```

Le mot **`def`** est un **mot-clef** du langage. Il signale à l'interpréteur `Python` que l'on commence la déclaration d'une fonction.

Après le mot-clef **`def`**, on place le nom que l'on désire donner à la fonction. Ce nom doit un être un identifiant valable (comme les variables) et peut contenir des lettres majuscules ou miniscules (non accentuées), des chiffres ou encore le symbole underscore `_`. Le nom doit commencer par une lettre ou par `_`. Éviter cependant de commencer par `_`, cela est souvent utilisé à des fins particulières.

Un identifiant ne peut pas être un mot-clef, puisque ces mots sont déjà utilisés par le langage. Il peut cependant être aussi long qu'on le souhaite. Inutile d'en mettre des tartines, mais cela vous permet de mettre un nom clair qui explicite le rôle de la fonction.

Enfin, `Python` fait la différence entre les majuscules et les miniscules [1]. Les fonctions `ma_fonction` et `Ma_Fonction` ne sont donc pas les mêmes.

Après le nom de la fonction, on met entre parenthèses une liste de paramètres de la fonction, séparés par des virgules. Ce sont des noms de variables qui vont être disponibles au sein de la fonction pour exécuter ses instructions.

Finalement, la ligne se finit par deux points `:`. Ce symbole indique le début d'un bloc de code qui **doit** être indenté. Cela signifie que toutes les instructions de la fonction doivent être décalées vers la droite avec un nombre constant d'espaces ou de tabulations. C'est comme cela que `Python` sait quelles instructions font partie de la fonction ou non. Lorsque vous arrêtez d'indenter votre code, `Python` comprendra cela comme la fin de la fonction.

Il y a une instruction spéciale pour les fonctions, l'instruction **`return`** qui s'utilise ainsi :

```
1 return expression
```

Lorsque `Python` rencontre l'instruction **`return`**, l'exécution de la fonction s'**arrête** et le résultat de l'expression est renvoyé à l'appelant. Cela permet donc d'utiliser les fonctions comme dans des expressions. Par exemple :

```
1 def delta_degre2(a,b,c):
    delta = b**2 - 4*a*c
    return delta

5
r = delta_degre2(1,0,1)
# r vaut -4
```

Exercice 2

Écrire une fonction qui prend en paramètres a, b, c *et* x *et qui calcule la valeur de* $P(x)$ *avec* $P(X) = aX^2 + bX + c$.

1. On dit que `Python` est sensible à la casse.

3 Instructions de contrôle

Jusqu'à présent, nous avons regardé une exécution **linéaire** (dans l'ordre) des instructions données. Mais ce n'est évidemment pas très utile en pratique sauf peut-être pour faciliter des calculs. On aimerait en effet que l'ordinateur puisse prendre des décisions, qu'il puisse sauter des instructions ou en répéter certaines. C'est à ça que servent les instructions de **contrôle**.

Une instruction de contrôle est une instruction spéciale qui permet d'éviter l'exécution ou au contraire qui permet de forcer l'exécution d'un bloc d'instructions sous certaines conditions. Il y en a trois à retenir :

- **L'instruction conditionnelle `if` :**

```
1 if condition:
    # instructions si la condition est vraie
  else:
    # instructions si la condition est fausse
```

La partie **`else`** est optionnelle.

L'instruction **`if`** permet de conditionner l'exécution d'un bloc d'instructions à la réalisation d'une condition précise. Typiquement, la condition est une comparaison entre deux nombres. On peut utiliser les opérateurs `==` (égal), `!=` (différent), `<` (strictement plus petit), `>` (strictement plus grand), `<=` (plus petit ou égal), `>=` (plus grand ou égal). Par exemple :

```
1 if delta >= 0:
    print("Il y a au moins une solution.")
  else:
    print("Il n'y a pas de solution.")
```

Les blocs sont, comme pour les fonctions, introduits par deux points `:`. Ils doivent être indentés correctement.

Exercice 3

Écrire une instruction conditionnelle permettant d'afficher l'unique solution à $ax^2 + bx + c = 0$ lorsque cette solution existe et est bien unique.

Utilisation avancée : Techniquement, la condition doit être une expression dont le résultat est un booléen (`True` ou `False`). On peut donc écrire des choses du genre :

```
1 def fonction_compliquee():
    # Plein d'instructions compliquees
    return True

5 if fonction_compliquee():
    print("La fonction compliquee a reussi.")
```

Ces instructions afficheront `"La fonction compliquee a reussi."` dès lors que `fonction_compliquee` renvoie `True`.

- **La boucle `for` :**

```
1 for k in sequence:
    # instructions a repeter
```

La boucle **`for`** permet de répéter un bloc d'instructions. Le bloc est répété autant de fois qu'il y a d'éléments dans `sequence`. À chaque itération, la variable indiquée, ici `k`, contient la valeur d'un élément de la séquence. Ainsi `k` va prendre toutes les valeurs de la séquence successivement.

On peut utiliser pleins de types d'objets en tant que séquences (des chaînes de caractères, des listes, des tableaux `numpy`), mais en pratique on utilisera les séquences qui sortent de la fonction **`range`**. **`range`** s'utilise sous 3 formes principales :

- Pour rapidement répéter n fois : **range(n)** renvoie la séquence de n nombres de 0 à $n-1$. Ce code affiche les nombres 0 à 9 :

```
1 for k in range(10):
    print(k)
```

- Pour parcourir les nombres entre deux bornes : **range(i,n)** renvoie la séquence de $n-i$ nombres de i à $n-1$. Ce code affiche les nombres 2 à 15 :

```
1 for k in range(2,16):
    print(k)
```

- Pour parcourir des nombres avec un pas défini : **range(i,n,s)** renvoie la séquence de $n-i$ nombres de i à $n-1$ en avançant avec des pas de s. Ce code affiche les nombres impairs de 1 à 17 :

```
1 for k in range(1,18,2):
    print(k)
```

Exercice 4

Écrire une boucle qui permet de lister l'ensemble des carrés de $1 = 1^2$ *à* $100 = 10^2$.

- **La boucle while :**

```
1 while condition:
    # instructions a repeter tant que condition est verifiee
```

Comme la boucle **for**, la boucle **while** permet de répéter un bloc d'instructions. Mais contrairement à la boucle **for**, et de manière semblable à **if**, le nombre de répétitions n'est pas définie à l'avance mais est soumis à une condition. Tant que la condition est remplie, le bloc s'exécute en boucle. La condition est vérifiée à chaque fois avant le bloc, et si la condition est vraie, alors le bloc est exécuté en entier. Puis la condition est de nouveau vérifiée et ainsi de suite.

Exemple :

```
1 n = 1
  while n**2 <= 100:
    n = n + 1
  print("Le premier carre apres 100 est {}.".format(n))
```

Exercice 5

Écrire une boucle qui permet d'afficher la plus grande solution entière à $x^3 - 5x^2 - 35 \leq 0$.