TP 2 : fonctions, entrées et sorties

1 Fonctions

Supposons que nous cherchions à calculer l'image d'un nombre par une fonction polynomiale donnée. Si la fonction est un peu longue à saisir, par exemple pour $f : x \mapsto x^7 - 6x^6 + 15x^4 - 12x^3 + 3x - 6$, il sera fastidieux de retaper toute cette fonction à chaque fois que l'on veut calculer une image par f. On peut dans un premier temps utiliser l'historique de l'intepréteur (grâce à la flèche du haut). Par exemple :

```
>>> x = 2
>>> x**7 - 6*x**6 + 15*x**4 - 12*x**3 + 3*x - 6
-112
>>> x = 3
>>> x**7 - 6*x**6 + 15*x**4 - 12*x**3 + 3*x - 6   # grace a fleche du haut deux fois
-1293
>>> x = -10
>>> x**7 - 6*x**6 + 15*x**4 - 12*x**3 + 3*x - 6   # grace a fleche du haut deux fois
-15838036
```

On peut bien entendu faire beaucoup mieux grâce à Python. On va définir une **fonction Python** qui ressemble à une fonction mathématique. La **syntaxe** (la manière précise d'écrire ce type d'instruction) est alors la suivante :

```
>>> def f(x) :
...     return x**7 - 6*x**6 + 15*x**4 - 12*x**3 + 3*x - 6
...
>>>f(2), f(3), f(-10)
(-112, -1293, -15838036)
```

Dans cette définition vient d'abord la déclaration d'une nouvelle fonction par le mot-clé `def`. Vient ensuite le **nom** de la fonction, ici `f`, suivi du **paramètre formel** de la fonction, placé entre parenthèses

Le paramètre formel est l'équivalent de la variable muette x dans la déclaration mathématique $f : x \mapsto x^7 - 6x^6 + 15x^4 - 12x^3 + 3x - 6$.

On conclut cette première ligne par deux-points. Une fois cette ligne saisie, on appuie sur Entrée, et les chevrons sont alors remplacés par trois points. Cela signifie que l'interpréteur attend la suite des instructions. Il faut saisir **quatre espaces** (on dit qu'on fait une **indentation** du code) afin d'indiquer que l'on va désormais préciser le contenu de la fonction. Une fois le contenu de la fonction terminé, on fait Entrée, les trois points apparaissent à nouveau, et si l'on fait Entrée sans rien taper sur cette ligne cela signale la fin de la définition de la fonction. Dans notre exemple, la fonction ne contenait qu'une ligne, la suivante contient donc seulement trois points et la définition de la fonction est alors terminée. Voici quelques erreurs à éviter :

```
>>> def f(x)                # erreur 1 : oubli des deux points
  File "<stdin>", line 1
    def f(x)
           ^
SyntaxError : invalid syntax

>>> def f(x) :              # erreur 2 : non respect de l'indentation
... return x**7 - 6*x**6 + 15*x**4 - 12*x**3 + 3*x - 6
       ^
IndentationError : expected an indented block
```

```
>>> def f(x) :          # erreur 3 : oubli du mot-cle return
...       x**7 - 6*x**6 + 15*x**4 - 12*x**3 + 3*x - 6
...
>>> f(2), f(3), f(-10)
(None, None, None)
```

Dans la troisième erreur, il n'y a pas de message d'erreur, mais notre fonction ne donne aucun résultat : l'expression $x**7-6*x**6+15*x**4-12*x**3+3*x-6$ est calculée en remplaçant x successivement par $2, 3$ et -10, mais l'interpréteur n'a pas reçu l'instruction de renvoyer le résultat. Il reste muet et se contente de renvoyer comme valeur l'objet `None`. On pourrait être tenté de remplacer `return` par l'instruction `print` que nous avons recontrée plus haut :

```
>>> def f(x) :
...     print(x**7 - 6*x**6 + 15*x**4 - 12*x**3 + 3*x - 6)
...
>>> f(2), f(3), f(-10)
-112
-1293
-15838036
(None, None, None)
```

Pour mieux comprendre ce qui vient de se passer voici un autre exemple. Supposons que nous cherchions à calculer la somme des images de 2 3 et -10 par f. Comparons les résultats avec `print` et avec `return`.

```
>>> def f(x) :
...     return x**7 - 6*x**6 + 15*x**4 - 12*x**3 + 3*x - 6
...
>>> f(2)+ f(3)+f(-10)
-15839441
>>> def f(x) :
...     print(x**7 - 6*x**6 + 15*x**4 - 12*x**3 + 3*x - 6)
...
>>> f(2)+ f(3)+f(-10)
-112
-1293
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NoneType' and 'NoneType'
```

Dans le deuxième cas, l'interpréteur affiche à l'écran la valeur de l'expression $x^7-6x^6+15x^4-12x^3+3x-6$ lorsqu'il rencontre la fonction `print`, mais ensuite il ne renvoie aucun objet réutilisable ultérieurement pour faire un calcul. Il vaut donc mieux éviter de prendre l'habitude d'utiliser `print` à l'intérieur des fonctions, suaf si c'est explicitement requis. On utilisera surtout la fonction `print` en dehors des fonctions, dans des suites d'expressions écrites dans l'éditeur de texte (appelées **scripts**). Une propriété remarquable de `return`, c'est qu'elle interrompt instantanément l'exécution de la fonction. Inutile donc de placer des instruction après un `return`. Ces instructions ne seront jamais lues, c'est du code mort.

```
>>> def mult_7(x) :
...     return 7 * x
...     print("Ceci ne s'affichera jamais")    # du code mort
...     return 0                 # toujours mort
```

Autre fait crucial : les variables définies à l'intérieur d'une fonction ne sont pas visibles depuis l'extérieur de la fonction : ce sont des variables muettes ! Aussitôt l'exécution de la fonction terminée, elles sont effacées par l'interpréteur. On dit qu'une telle variable est **locale** à la fonction.

```
>>> def f(y) :
...     x = 1
...     return y
...
>>> f(2)
2
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

Si une variable `x` existait déjà avant l'exécution de la fonction, tout se passe comme si, durant l'exécution de `f`, cette variable était masquée momentanément, puis restituée à la fin de l'exécution de la fonction.

```
>>> x = 0
>>> def f(y) :
...     x = 1
...     return y
...
>>> f(2)
2
>>> x
0
```

Finalement, signalons qu'une fonction peut comporter autant de paramètres formels que souhaité (et éventuellement aucun). Par exemple

```
>>> def une_fonction_a_deux_variables(x, y) :
...     return x * 2**y
...
```

Instructions d'entrée et sortie

Nous allons utiliser un nombre de lignes de code de plus en plus important au fur et à mesure de notre approfondissement du langage Python. En utilisant l'interpréteur de commandes, si l'on veut corriger une instruction dans une série d'instructions, il nous faut tout recommencer à chaque essai. De plus, on peut vouloir sauvegarder une suite d'instructions que nous avons testée, afin de la réutiliser dans le futur. Nous allons donc commencer à travailler avec des fichiers texte dans lesquels nous entrerons nos suites d'instructions, ligne après ligne. Nous taperons les lignes d'instructions dans l'éditeur de texte (fenêtre de gauche dans Spyder). Il faudra sauvegarder ce fichier régulièrement en maintenant enfoncée la touche **Ctrl** puis en appuyant sur **s** (on note cela **Ctrl**+**s**). On peut ensuite exécuter cette suite de commandes (on parle de **script** Pyhton pour désigner le fichier contenant la suite de commande) en appuyant sur le triangle vert au dessus de l'éditeur de texte. Python exécute alors les unes après les autres les commandes du script. Les résultats apparaissent au fur et à mesure dans l'interpréteur. Voici un premier exemple (n'oubliez pas que l'absence des balises `>>>` indiquent que vous devez entrer les instructions dans l'éditeur de texte puis les exécuter) :

```
x = 2 ** 8
x
```

Observez le résultat : rien ne s'affiche ! En fait lorsque l'interpréteur lit un script Pyhton, il effectue les calculs les uns après les autres mais n'affiche aucun résultat, sauf si cela lui est explicitement demandé. Cela évite d'encombrer l'interpréteur de résultats : imaginez un code réalisant des dizaines de milliers de calculs afficher CHACUN des résultats intermédiaires ! Si l'on souhaite dans le script précédent faire afficher la valeur de x, on ajoute une instruction `print`.

```
x = 2 ** 8
print(x)
```

Revenons maintenant sur l'indentation nécessaire dans une déclaration de fonction. Entrez les deux scripts suivants et testez les, l'un après l'autre (une indentation valide comporte nécessairement quatre espaces. Sur certains éditeurs quatre espaces sont automatiquement réalisés par la touche Tab du clavier, mais il faut se méfier car ceci n'est pas automatique. On obtient alors une erreur particulièrement difficile à détecter)

```
def f():
    print('hello')
print('bonjour')
f()
```

```
def f():
    print('hello')
    print('bonjour')
f()
```

Exercice :

Expliquez la différence de comportement entre les deux fonctions.
Demandez à votre professeur de vérifier que vous avez compris.

Le point crucial à comprendre est qu'une définition de fonction utilisant le mot-clé **def** n'est qu'une **définition**. Avec simplement une instruction **def** l'ordinateur n'exécute PAS les commandes contenues dans le corps de la fonction. Elles ne seront exécutées que lors d'un appel de la fonction : c'est le rôle de la quatrième ligne des deux scripts précédents. Puisque la fonction n'a pas d'argument formel, il y a le nom de la fonction **f**, suivi d'une paire de parenthèses vide.

Venons-en à l'instruction `print` elle-même. Voici quelques exemples d'utilisation de cette fonction (attention dans ce script je n'ai utilisé que des apostrophes, pas de guillemets malgré les apparences).

```
x, y = 3, 100000
z = 3.1416
print(x, y, z)
print(x, y, z, sep='')
print(x, y, z, sep='; ')
print(x, y, z, sep='\n')
print('x=', x, sep='',end='; ')
print('y=', y, sep='',end='; ')
print('z=', z, sep='')
```

Exercice :

Prenez le temps de bien comprendre **chaque** ligne affichée après exécution de ce code.

Nous n'avons pas encore parlé des **chaînes de caractères** bien que nous les ayons déjà rencontrées dans les instructions `print`. Une chaîne de caractères est un objet Python de type **str** (vérifiez en tapant

`type("hello")` dans l'interpréteur), succession de caractères typographiques de longueur quelconque, délimitée par des apostrophes simples ou des guillemets. On peut aussi utiliser des triples guillemets pour revenir à la ligne à l'intérieur d'une chaîne de caractères. Voici des exemples (dans ce code une double apostrophe est en fait un guillemet) :

```
print('-> Ici on peut employer des "guillemets" !!')
print("-> Ici on peut employer des 'apostrophes' !!")
print("""-> Voici un saut ....
    de ligne.""")
print('Autre possibilité pour \npasser à la ligne')
```

Si une chaîne de caractères est un peu trop longue pour être écrite sur une seule ligne dans un script, on peut saisir cette chaîne sur plusieurs lignes : voici un exemple

```
print('-> Voici comment couper une ligne '
    'trop longue dans un script sans que ce'
    " changement ne soit visible dans l'interpreteur" )
```

Il n'y aura dans ce cas pas de saut de ligne à l'affichage : le saut de ligne n'est présent dans le script que pour le confort de lecture du script. Remarquez l'utilisation des guillemets dans la troisième ligne afin de pouvoir afficher l'apostrophe.

La commande **format** permet de contrôler très précisément l'affichage des chaînes de caractères, mais son utilisation est un peu plus avancée. Recherchez la dans l'aide lorsque vous aurez terminé cette feuille de TP.

Après avoir vu comment faire **écrire** des messages par l'ordinateur, voyons comment lui faire **lire** des informations données par **l'utilisateur** (c'est-à-dire la personne qui exécutera le script python que vous codez : cette année ce sera toujours vous-mêmes, mais un code est fait pour être utilisé par d'autres bien sûr). Voici un petit script qui demande à l'utilisateur de fournir un nombre, qui affecte ce nombre à la variable x, et qui calcule puis affiche la valeur du carré de ce nombre.

```
x = input('Entrez une valeur pour la variable x : ')
print(x,x**2)
```

À l'exécution de l'instruction **input**, Python affiche le message passé en argument de `input`, puis attend une saise au clavier. On saisit un nombre puis on valide avec la touche Entrée. Malheureusement ce code produit une erreur. La commande **input** a pourtant bien récupéré la valeur donnée au clavier. Mais le message d'erreur nous met sur la piste du problème : la valeur récupérée par `input` n'est pas l'entier 3 de type **int**, mais la chaîne de caractères (de longueur 1 !) "3". Or le programme demande d'afficher le carré de x, mais Python ne sait pas faire le carré d'une chaîne de caractère. C'est notre première rencontre avec un problème de **typage**.

Sur la notion de type en informatique :

*Toute information est stockée sous la forme de 0 et de 1 (les bits) dans la mémoire de l'ordinateur (c'est la fameuse machine de Turing). Même en regroupant les bits par groupe de 8 (les bytes), tout ce que l'ordinateur voit à priori ce sont des nombres. Si l'on veut pouvoir coder d'autres types d'informations (chaînes de caractères, nombres à virgule par exemple), il est nécessaire d'ajouter à ces données quelques bits qui permettent à l'ordinateur de savoir à quoi il a à faire. C'est à cela que servent les **types**. Nous avons rencontré pour l'instant deux types : **int** et **str**. Vous pouvez constater en tapant `type(3.1416)` dans*

*l'interpréteur que le type **float** est celui qui permet de coder des nombres à virgule (nous détaillerons plus tard dans l'année ce type).*

On peut corriger ce problème en forçant Python à donner une valeur entière à la saisie par la commande `input`, en utilisant la commande **eval**, comme suit

```
x = eval(input('Entrez une valeur pour la variable x : '))
print(x,x**2)
```

Instruction conditionnelle : if

Définissons ici la fonction max, qui à un couple (a, b) de réels, associe la plus grande valeur entre a et b. Mathématiquement cela se représenterait ainsi :

$$\max \ : \ (a,b) \mapsto \begin{cases} a & \text{si } a \geqslant b \\ b & \text{sinon} \end{cases}$$

De sorte que $\max(-1,3) = 3$, $\max(\pi,\pi) = \pi$, et $\max(-1,-\sqrt{2}) = -2$. Comment faire pour coder cette fonction à l'aide de Python ? Il nous faut pouvoir faire effectuer un **test** à l'ordinateur. Cela se fait à l'aide du mot clé **if**, avec la syntaxe suivante :

```
def max(a,b):
    if a >= b:
        return a
    else:
        return b
```

Fonction que vous pouvez tester avec

```
>>> max(-3,-2)
-2
>>> max(1,1)
1
```

Observons la commande `if` utilisée dans notre fonction max. Le mot-clé `if` est suivi d'une **condition de choix**. Voyons sur quelques exemples ce que l'interpréteur Python renvoie lorsqu'on entre ce genre de conditions :

```
>>> 2 >= 0
True
>>> -1 < -2
False
>>> b = 1 > 3
>>> b, id(b), type(b)
(False,9474016,bool)
```

Une condition est donc d'un nouveau type : le type **booléen** (mot clé Python `bool`). Une variable booléenne ne peut prendre que deux valeurs : vraie (`True`) ou faux (`False`). La condition de choix est vérifiée quand l'évaluation de cette condition renvoie le booléen `True`. L'interpréteur exécute alors la suite d'instructions qui se trouve dans le premier bloc d'instructions (rappelez vous que l'indentation détermine où commencent et finissent les blocs d'instructions), il n'exécute ensuite **pas** le deuxième bloc d'instructions précédé du mot clé `else` et saute directement à la fin de ce bloc. Si par contre la condition de choix n'est pas vérifiée (on obtient le booléen `False` en l'évaluant), alors l'interpréteur n'exécute **pas** le premier bloc et saute à la fin de celui-ci, pour n'exécuter que le deuxième bloc.

Rôle crucial de l'indentation en Python :

En python contrairement à d'autres langages, et dans un souci de légèreté, il n'y a pas de parenthèses pour délimiter des blocs d'instructions. L'indentation permet de délimiter les blocs, et les deux points : indiquent

l'ouverture d'un bloc. Il est crucial de respecter scrupuleusement cette indentation (quatre espaces) sous peine de comportements imprévisibles de vos programmes.

Revenons aux Booléens. Pour en définir, on a recours aux opérateurs de comparaison usuels : les comparaisons, l'égalité, etc. Voici une liste des différents opérateurs acceptés en Python (n'exécutez pas ces lignes de code, elles produiraient des erreurs, voyez-vous pourquoi ?)

```
x == y   # x est egal à y
x != y   # x est different de y
x > y    # x est strictement superieur à y
x < y    # x est strictement inferieur à y
x >= y   # x est superieur ou egal à y
x <= y   # x est inferieur ou egal à y
x is y   # x et y pointent au meme endroit dans la memoire
   autrement dit id(x)=id(y)
x in y   # x appartient a y (voir plus loin)
```

Il est crucial de ne pas confondre l'opérateur d'égalité « == »de l'opérateur d'affectation « = ». Ce dernier, dois-je le rappeler **évalue ce qui est à droite du signe égal et affecte le résultat dans la variable qui est à gauche du signe égal**. Le premier quant à lui teste l'égalité entre les deux membres de droite et de gauche, et renvoie `True` si il y a égalité et `False` sinon.

Pour exprimer des conditions plus complexes, on dispose, comme en logique mathématique, des opérateurs de négation (`not`), de conjonction (`and`) et de disjonction (`or`). Python accepte aussi la syntaxe `1 < a < 2` qu'il traduit par `1 < a and a < 2`.

Exercice :

Devinez les résultats des commandes ci-dessous **avant** de les exécuter dans l'interpréteur.

```
>>> x = -1
>>> ( x > -2 ) and ( x**2 < 5 )
>>> ( x < -1 ) or ( x**2 >= 2 )
>>> not( x > -2 )
>>> -2 < x <= 0
```

True
False
False
True

Avec les règles de priorité (les opérateurs non booléens sont prioritaires sur les opérateurs de comparaison, qui sont prioritaires sur les opérateurs logiques), on peut parfois écrire des expressions en se passant de parenthèses, mais il vaut mieux les écrire tout de même pour des raisons de lisibilité du code.

Exercice :

Voici deux exemples de fonctions utilisant les booléens et l'instruction `if`.
À vous de dire ce qu'elles codent. Vérifiez en les testant vous-mêmes.

```
def fonction1(x,y,z):
    if x <= y <= z:
        return True
    else:
        return False
```

```
def fonction2(x) :
    if x >= 0:
        return x
    else:
        return -x
```

Remarquez la syntaxe de la deuxième fonction, utilisable quand les instructions conditionnelles sont très courtes et que chaque bloc ne contient qu'une instruction. La première fonction quand à elle prend en argument des nombres et renvoie un booléen.

Exercice :

1. Codez (dans l'éditeur de texte) une fonction nommée **trinome** prenant comme arguments trois nombres a,b et c et qui affiche une phrase indiquant le nombre de solutions de l'équation $ax^2 + bx + c = 0$. Votre fonction testera d'abord si $a = 0$, puis dans le cas $a \neq 0$, séparera les cas $\Delta < 0$, $\Delta = 0$ et $\Delta > 0$. Pour ce faire il est possible d'imbriquer les instructions conditionnelles de la manière suivante :

```
if condition1:
    ............
    ............
else:
    if condition2:
        ............
        ............
    else:
        if condition3:
            ............
            ............
        else:
            ............
            ............
```

2. Cette fonction peut être codée de manière plus lisible de la manière suivante :

```
def trinome_v2(a,b,c):
    delta=b**2-4*a*c
    if a==0:
        print('Equation du
    premier degre')
    elif delta<0:
        print('Pas de
    solution')
    elif delta==0:
        print('Unique
    solution')
    else:
        print('Deux
    solutions')
```

Ici `elif` est un raccourci pour `else if`. Cela allège la syntaxe et permet de mettre toutes les indentations au même niveau.

Exercice :

Construire une fonction `max3` qui prend en argument trois nombres a, b et c et renvoie le maximum des trois nombres. On s'aidera de la fonction `max` codée plus haut.

Exercice :

1. Les années bissextiles reviennent tous les 4 ans, sauf les années séculaires (multiples de 100) si celles-ci ne sont pas multiples de 400. Ainsi l'année 1900 n'était pas bissextile, alors que l'année 2000 l'était. Codez une fonction `bissextile(n)` qui prend en argument l'année n et qui renvoie `True` si l'année est bissextile et `False` sinon.
2. Faire de même mais en utilisant une seule ligne dans le corps de la fonction et aucune instruction conditionnelle, seulement des opérateurs logiques.