TP n° 1 – Introduction à la programmation

Tout d'abord, quelques conseils avant de débuter votre premier TP :

— au début vous passerez beaucoup de temps à corriger vos erreurs de syntaxes, qui en général proviennent de l'oubli d'une parenthèse ou du symbole `:` en fin de ligne après un mot clé, ou encore d'un espacement placé au mauvais endroit. Soyez donc vigilant et lisez bien le message envoyé par le débogueur de Python dans la console ;

— si vous avez un doute concernant l'utilisation d'une commande `truc`, n'hésitez pas à utiliser l'aide en tapant `help(truc)` dans la console ;

— habituez-vous à utiliser un maximum les raccourcis clavier de l'éditeur pour aller vite ;

— enregistrez dès le début du TP votre travail avec un nom de la forme `TPX.py` (évitez les accents et caractères spéciaux).

1 B.A.–BA

Toute phrase compréhensible par l'interpréteur Python sera appelée une ligne de code ou un code. On appelle primitive, commande ou instruction une fonction préprogrammée dans Python. Chaque objet de ce type (ligne de code, primitive ou instruction) sera dénoté par un mot écrit dans `cette police`. Entrer une ligne de code signifie l'écrire dans la console à la suite de l'invite de commande `>>>` puis appuyer sur la touche « entrée ». Le symbole `>>>` est appelé le signal d'invite ou le prompteur, et indique que Python est prêt à évaluer une commande.

Dans un environnement de développement, comme Pyzo, on trouve aussi un éditeur de texte dans lequel il est possible de saisir des commandes. Pour les évaluer, il est nécessaire d'exécuter le fichier (ou une cellule du fichier, entre 2 jeux de `##`). Le symbole `#` permet d'insérer des commentaires. Pour afficher le résultat d'une commande dans la console, il est nécessaire d'utiliser la fonction `print`.

1.1 Affectation

Le principe de l'affectation est simple : on souhaite donner une valeur particulière (par exemple 2) à une variable donnée (par exemple `x`). Cette opération est fréquemment utilisée en programmation, par exemple pour stocker des résultats intermédiaires lors d'un calcul. On emploiera dans la syntaxe Python la ligne de code `x=2`. On peut alors utiliser la variable `x` pour effectuer des calculs : Python traitera celle-ci comme si c'était le chiffre 2.

Pour supprimer la valeur contenue dans une variable, on peut utiliser la fonction `del`. La variable donnée en argument disparait du répertoire local que l'on peut explorer avec la fonction `dir()`.

```
>>> x=2
>>> x
2
>>> y=x+2
>>> y
4
>>> del(x)
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
>>> y
4
```

La séquence des commandes a son importance !

Exercice 1.1
On considère le problème de l'interversion de deux variables : partant de deux variables `a,b` possédant chacune une valeur (par exemple `a=1` et `b=2`), on cherche à échanger les valeurs contenus par celles-ci (on veut obtenir `a=2` et `b=1`).

1. Entrer `a=1` et `b=2` dans la console. Entrer ensuite successivement les lignes de code `a=b` et `b=a`. Est-ce que cela permet d'intervertir les valeurs de `a` et `b` ? Pourquoi ?
2. Entrer de nouveau `a=1` et `b=2` dans la console. En utilisant une troisième variable `c`, intervertissez les valeurs de `a` et de `b` (on s'abstiendra d'entrer des lignes tels que `a=2` et `b=1`, sinon l'exercice n'a aucun intérêt !).

1.2 Types de variables

En programmation, on distinguera plusieurs types de variables. Dans une premier temps, on se limitera aux nombres entiers (représentation exacte de $\mathbb{Z}$, type `int`), à une approximation des nombres réels limitée à $\left[-1{,}7\cdot 10^{308}; 1{,}7\cdot 10^{308}\right]$) dits *flottants* (type `float`) et aux booléens (type `bool`, valeurs `True` et `False`).

On affichera le type d'une variable `x` dans la console avec la commande `type(x)`. On peut tester le type d'une variable, par exemple si c'est un entier avec la commande `type(a)==int` qui renvoie un booléen.

```
>>> a=1; type(a)
<class 'int'>
>>> a=2.34; type(b)
<class 'float'>
>>> c=True; type(c)
<class 'bool'>
>>> type(a)==int
True
```

Mémo – **Opérations élémentaires sur les nombres**

`x + y`, `x - y` .... somme, différence
`x * y`, `x / y` .... produit, division
`x**y` ou `pow(x,y)` .... puissance
`x // y`, `x % y` .... quotient et reste de la division euclidienne
`a+=` $\iff$ `a=a+` .... fonctionne avec `-`, `*` et `/`

Exercice 1.2

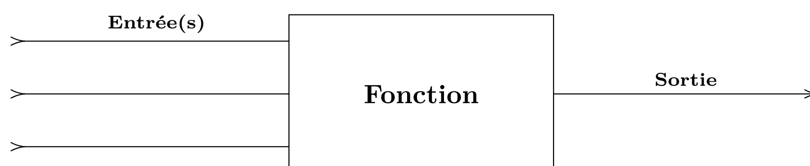
1. Définir dans la console le code suivant :

```
a,b,c,d = 1, 2.34, True, False
```

   puis prévoir et vérifier le type des résultats des opérations suivantes : `a*b`, `a*c`, `a*d`, `b*c`, `b*d` et `c*d`.
2. En déduire une hierarchisation des ensembles de nombres en Python.

1.3 Fonction

On appelle *fonction* tout programme qui, à partir d'un ou plusieurs paramètre(s) appelés *entrée(s)* ou *argument(s)*, calcule une valeur dépendant de ces paramètres appelée *sortie*.



Déclarer une fonction c'est la définir quelles que soient les valeurs des paramètres, et *appeler une fonction* c'est l'appliquer à des paramètres dont les valeurs ont été fixées. Par exemple, une implémentation de $f : x \mapsto x^2$ peut être

```
def f(x):
    return(x**2)
```

Les arguments d'une fonction sont ordonnés et séparés par une virgule. Une fonction peut renvoyer plusieurs objets, eux aussi séparés par des virgules. Pour calculer 3^2, il suffit alors de faire :

```
>>> f(3)
9
```

alors que

```
>>> f
<function f at 0x7f8760b8a310>
```

vous dit seulement que `f` est une fonction.

Exercice 1.3

1. Définir une implémentation de la fonction $f : x \mapsto x+1$. Vérifier les images $f(2)$, $f(3)$.
2. Définir une implémentation de la fonction $g : (x, y) \mapsto xy$. Vérifier les images $g(2, 3)$, $g(4, 3)$.
3. Définir une implémentation de la fonction $h : (x, y, z) \mapsto 1 + x + yz$. Vérifier les images $h(1, 2, 3)$, $h(4, 3, 5)$.

1.4 Variable locale, variable globale

Toute variable qui est définie dans une fonction est appelée *variable locale* et appartient à un espace local, disjoint de l'espace global où sont stockées toutes les *variables globales*. Lors de l'évaluation d'une fonction, c'est l'espace local qui est scruté en premier puis l'espace global. Par défaut, une fonction ne peut pas modifier de variable globale sans que ce soit précisé avec le mot clé `global`.

Exercice 1.4
Prévoir et vérifier ce que renvoie le code suivant :

```
a=2
def f(x):
    return(a*x)
def g(x):
    a=3
    return(a*x)
def h(x):
    global a
    a=3
    return(a*x)
```

avec la série de commandes :

```
print(f(3),a)
print(g(3),a)
print(h(3),a)
print(f(3),a)
```

Dans le cas où une fonction ne possède pas d'instruction `return`, on parle de procédure. À aucun moment, une instruction `print` ne remplace une instruction `return` — même si le comportement apparent dans la console semble identique. Par exemple :

```
>>> def f(x):
...     print(x)
...
>>> a=f(2)
2
>>> type(a)
<class 'NoneType'>
```

définit un objet `a` de type `none`.

1.5 Expressions logiques et branchements conditionnels

En l'honneur du logicien anglais Georges Boole, les deux valeurs logiques Vrai et Faux sont appelées des booléens et sont respectivement notées `True` et `False`. La fonction `bool` renvoie `True` pour tout argument entier non nul, `False` si l'argument est nul. Une *expression logique* est une expression de type booléen, c'est-à-dire une expression pouvant prendre la valeur vrai ou faux. C'est par exemple le cas du résultat d'un test réalisé avec un des opérateurs de comparaison. Il est évidemment possible d'utiliser les connecteurs logiques *et*, *ou* et *non* entre des booléens.

MÉMO – **Tests, connecteurs logiques**

`==`, `!=`	égal, différent
`<`, `>`	strictement inférieur, supérieur
`<=`, `>=`	inférieur/supérieur ou égal
`not(a)`	NON ($\neg a$)
`a and b`, `a & b`	ET logique ($a \wedge b$)
`a or b`, `a	b`	OU logique ($a \vee b$)
`a xor b`, `a ^ b`	ou exclusif ($a \oplus b$)

Comme toutes opérations, il y a des priorités sur les opérations booléennes : le *et* est prioritaire sur le *ou*, et le *non* est prioritaire sur les autres opérations. Lorsque Python évalue une expression booléenne, il le fait de façon paresseuse ; c'est-à-dire que si la partie gauche d'un `or` est vraie, il n'évalue pas la partie droite. De même si la partie gauche d'un `and` est fausse, la partie droite n'est pas évaluée.

Les instructions élémentaires nécessaires à la programmation sont bien entendues disponibles en Python, comme le branchement conditionnel *si..alors..sinon* (*if..then..else* en anglais) que l'on notera :

```
if condition:
    bloc instructions A
else:
    bloc instructions B
```

Le comportement de cette instruction est le suivant. Lorsque la machine évalue la ligne de code *if Condition then A else B*, elle commence par déterminer si `condition` prend la valeur `True` ou `False` :

— si `condition` est `True`, alors elle effectue le `bloc instructions A` ;
— sinon elle effectue le `bloc instructions B`.

On notera qu'il n'y a pas d'instruction *then* : elle est remplacée par une indentation. Le retour à la ligne induit une sortie du bloc d'instruction.

Dans le cas où l'on souhaite distinguer plus de 2 cas, il est possible d'utiliser l'instruction `elif` qui signifie *else-if* et qui s'écrit :

```
if condition1:
    bloc instructions 1
elif condition2:
    bloc instructions 1
...
elif conditionK:
    bloc instructions K
else:
    bloc instructions K+1
```

Exercice 1.5

On considère la fonction définie par $f(x,y) = \left\{ \begin{array}{ll} 1 & \text{si } x = y \\ 0 & \text{sinon.} \end{array} \right.$

1. Implémentez f, puis testez votre programme avec $x = 1$ et $y = 1$, puis $x = 1$ et $y = 2$.
2. Que se passe-t-il si vous appliquez f à $x =$`True` et $y =$`False` ? Qu'en pensez-vous ?

Exercice 1.6

On souhaite implémenter la fonction définie par $f(a,b) = \left\{ \begin{array}{ll} (a,b) & \text{si } a > 0 \\ (a,0) & \text{si } a \leqslant 0. \end{array} \right.$

On considère pour cela les deux déclarations ci-dessous :

```
def f1(a,b):
    if a>0:
        x=a
        y=b
    else:
        x=a
    y=0
    return(x,y)
```

```
def f2(a,b):
    if a>0:
        x=a
        y=b
    else:
        x=a
        y=0
    return(x,y)
```

1. L'une des deux fonctions `f1`, `f2` est une implémentation de `f`. À votre avis, de laquelle s'agit-il ? Vérifiez votre réponse à la question précédente en déclarant `f1` et `f2` dans l'éditeur de texte, puis en évaluant ces fonctions en $(a,b) = (1,2)$ et $(a,b) = (-1,2)$.
2. On considère celle des deux fonctions `f1`, `f2` qui n'est pas une implémentation de `f`. Pouvez-vous expliquer (en moins d'une ligne) ce que fait cette fonction ?

Exercice 1.7 ($\star$)

1. Définir une fonction `maximum2`, qui prend en entrée deux entiers a et b, puis renvoie en sortie le plus grand des deux. Tester votre fonction avec les couples $(1,2)$, $(2,1)$ et $(1,1)$.
2. Définir une fonction `maximum3`, qui prend en entrée trois entiers a, b et c, puis renvoie en sortie le plus grand des trois. Tester votre fonction avec les triplets $(1,2,3)$, $(1,3,2)$, $(3,1,2)$ et $(1,1,2)$.
3. Définir une fonction `maximum4`, qui prend en entrée quatre entiers a, b, c et d, puis renvoie en sortie le plus grand des quatre. Tester votre fonction avec les quadruplets $(1,2,3,4)$, $(1,3,4,2)$, $(4,3,1,2)$ et $(1,4,2,3)$.
4. Prévoir le nombre de tests à réaliser pour trouver le maximum d'un ensemble de $n \geqslant 2$ entiers.
5. Regarder dans l'aide du logiciel à quoi servent les primitives `min` et `max`. À l'avenir on aura toujours le droit d'utiliser celles-ci.

Exercice 1.8 ($\star$)

Écrire trois fonctions qui évaluent respectivement la négation d'une variable booléenne, le *et* et le *ou* logiques de deux variables booléennes, sans utiliser aucune des trois instructions `not`, `and` ou `or`.

Exercice 1.9 (⋆)
Étant donnés trois réels a, b, c, on note $P_{a,b,c}$ la fonction définie par $P_{a,b,c}(x) = ax^2 + bx + c$.

1. On se place dans le cas $a \neq 0$. Déclarer une fonction `nb_racines`, qui prend en entrée trois réels `a`, `b`, `c` tels que `a` contienne une valeur non nulle et qui renvoie :
   — `2` si le trinôme $P_{a,b,c}$ admet deux racines réelles ;
   — `1` si le trinôme $P_{a,b,c}$ admet une racine double réelle ;
   — `0` si le trinôme $P_{a,b,c}$ n'admet aucune racine réelle.

   Vérifier ensuite que le programme fonctionne, en considérant les trinômes $x^2 - 1$, $x^2 - 2x + 1$ et $x^2 + x + 1$.
2. On se place maintenant dans le cas général (c'est-à-dire que a peut être nul). Déclarer une fonction `nb_racines_général` qui possède la même spécification que la précédente, mais qui donne encore le bon résultat si $a = 0$ (on renverra -1 dans le cas où le polynôme possède une infinité de racines). Tester ensuite le programme avec les polynômes $x + 1$, 1 et 0.

Exercice 1.10 (⋆⋆)

1. Tester dans la console les instruction `15%3`, `15//3`, `15//4`, `15%4`. Expliquer le fonctionnement des commandes `//` et `%`.
2. Écrire une fonction `divise` qui prend comme arguments deux entiers relatifs a et $b \neq 0$ et qui renvoie le booléean `True` si b divise a, `False` sinon.
3. En déduire une implémentation d'une fonction `pair` qui teste la parité d'un entier relatif `n`.
4. Définir une fonction `bissextile` qui prend comme argument un entier a et qui renvoie le booléen `True` si l'année a est bissextile, `False` sinon. On rappelle qu'une année est bissextile si elle est divisible par 4 mais non par 100, sauf si elle est divisible par 400.

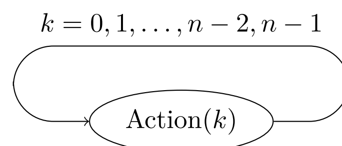
2 Structures itératives

L'une des tâches dont s'acquittent le mieux les machines est l'exécution à grande fréquence et sans erreur de tâches répétitives. Il existe bien entendu différentes méthodes pour programmer l'exécution de ces tâches, et nous allons commencer par la plus courante d'entre elles : la structure itérative ou *boucle for*.

La *structure itérative* est un procédé informatique analogue au raisonnement par récurrence en mathématiques. Elle ordonne à la machine d'opérer à la suite et dans l'ordre n actions, que nous appellerons Action(0), Action(1), . . ., Action($n - 1$). Elle s'exprime intuitivement de la manière suivante :

Pour k allant de 0 à $n - 1$ faire Action(k).

Cette structure est communément appelée une **boucle**, car sa représentation la plus simple est celle d'une boucle (symbolisant le sens d'évaluation du programme) autour d'une bulle figurant les actions considérées.



La syntaxe Python correspondant à l'expression *Pour* k *allant de* 0 *à* $n - 1$ *faire Action*(k) est la suivante :

```
for k in range(n):
    Action(k)
```

Lorsque la machine évalue cette expression, la variable `k` prend successivement les valeurs $0, 1, ..., n-2, n-1$, et pour chacune d'elles la machine exécute Action(k). Les n actions effectuées sont notées de 0 à $n - 1$ et non pas de 1 à n (convention courante en informatique).

Pour définir un ensembles d'entiers ordonnés régulier, on utilisera la commande `range`. Étant donné trois entiers relatifs a, b et h :

— la commande `for k in range(a)` permet de faire une boucle avec k allant de 0 jusqu'à $a - 1$;
— la commande `for k in range(a,b)` permet de faire une boucle avec k allant de a jusqu'à $b - 1$;
— la commande `for k in range(a,b,h)` permet de faire une boucle avec k allant de a à $b - 1$, mais en avançant par pas de longueur h (c'est-à-dire que k prend les valeurs $a, a + h, a + 2h, a + 3h, \ldots$ et ce en allant au plus jusqu'à $b - 1$).

Notez bien que a, b et h doivent nécessairement être des entiers relatifs de type `int`. En particulier, `range(5.0)` renvoie une erreur.

Exercice 1.11

1. Écrire une fonction `sommation` qui prend en entrée une fonction g ainsi qu'un entier n, puis qui renvoie la valeur de $\sum_{k=0}^{n-1} g(k)$.
2. En déduire la valeur de la somme des 3 premiers nombres impairs, puis des 100 premiers nombres impairs. Qu'est-ce que ces valeurs nous incitent à conjecturer ?
3. En déduire la valeur de $u_n = \sum_{k=0}^{n-1} \frac{4(-1)^k}{2k+1}$, pour $n = 100$ puis $n = 10^5$. Qu'est-ce que ces valeurs nous incitent à conjecturer ?

Lorsqu'on a une boucle for, mais qu'on souhaite la terminer avant la fin de son exécution, on peut utiliser la commande `break`.

Exercice 1.12

1. Écrire un programme qui calcule les 50 premiers termes de la table de multiplication de 13, mais n'affiche que ceux qui sont des multiples de 7.
2. Écrire un programme qui affiche à l'écran les 50 premiers termes de la table de multiplication de 13 qui sont des multiples de 7.

3 Boucle conditionnelle `while`

Tout comme la boucle `for`, la boucle `while` est une structure itérative. Elle permet donc de répéter un certain nombre de fois une (ou plusieurs) action. Dans une boucle for, le nombre de répétitions est fixé à l'avance alors que dans une boucle `while`, l'action est répétée tant qu'une certaine condition est vérifiée. La boucle `while` s'exprime donc intuitivement de la manière suivante :

Tant que *Condition* Faire *Action* .

La syntaxe Python pour les boucles while est donc la suivante :

```
while Condition:
    Action
```

La syntaxe d'une boucle while est analogue à celle d'une boucle for : il faut utiliser les deux-points et l'indentation pour le bloc d'instructions à répéter.

Exercice 1.13

Un nombre $n \geqslant 2$ est dit premier s'il n'admet que deux diviseurs positifs : 1 et n. On peut montrer qu'un entier naturel n n'est pas premier si, et seulement si, il admet un diviseur inférieur ou égal à $\sqrt{n}$. Définir une fonction `EstPremier` qui prend comme argument un nombre entier `n` et qui renvoie `True` si c'est un nombre premier, `False` sinon.

Exercice 1.14

1. Définir une fonction `PGCD` qui prend comme arguments deux entiers `a` et `b` et qui renvoie leur PGCD, c'est-à-dire le plus grand entier qui les divise simultanément, avec une boucle `for`.
2. Définir une fonction `PGCD` qui prend comme arguments deux entiers `a` et `b` et qui renvoie leur PGCD déterminé avec l'algorithme d'Euclide :

   Soient $a, b \in \mathbb{N}^*$ avec $a \leqslant b$. On définit la suite des restes (r_n) par récurrence :

   $$\begin{cases} r_0 = a, \ r_1 = b \\ \text{pour tout } n \in \mathbb{N}, \ r_{n+2} \text{ est le reste de la division euclidienne de } r_n \text{ par } r_{n+1} \end{cases}$$

   Il existe un rang p à partir duquel la suite est nulle et dans ce cas, le PGCD de a et b est le dernier reste non nul : r_{p-1} est le PGCD de a et b.

* *
*