Chapitre 1 : Affectation, fonctions, booléens, branchements conditionnels

Instructions introduites dans ce cours : `+,-,*,/,**,print, ##, #, =, del, def, return, ==, !=, <=, >=, and, or, not, if, else,elif.`

1 Introduction

1 A) Généralités

Python est un langage de programmation créé en 1989 par Guido Van Rossum, dont l'usage s'est répandu depuis (YouTube est par exemple écrit en Python). Citons ses caractéristiques principales, dont certaines seront explicitées au cours de l'année :

- Python est un langage gratuit ;
- la syntaxe de Python est simple et dépouillée, ce qui en fait un langage adapté à la découverte de la programmation (en moyenne, on considère qu'un code Python est de 3 à 5 fois plus court qu'un code C,C++ ou Java correspondant) ;
- Python est un langage multi-plateforme (il fonctionne sous les principaux systèmes d'exploitation) ;
- Python est un langage de programmation multi-paradigme, particulièrement bien adapté à la programmation impérative et orientée objet ;
- Python possède un typage dynamique et fort ;
- Python est un langage semi-compilé, dont l'interpréteur est écrit en C.

1 B) Pyzo

La version de Python disponible au lycée est la version 3.5 (attention, il y a des différences entre les syntaxes des versions 2.x et 3.x). Plus précisemment, les machines du lycée sont équipées de *Pyzo* : il s'agit d'une distribution Python qui contient également une sélection de modules (*packages* en anglais) complémentaires, ainsi qu'un environnement de développement intégré (IDE pour *internal developpement environnement* en anglais) appelé IEP.

Un environnement de développement est un ensemble d'outils permettant de simplifier et d'accélérer l'utilisation d'un langage de programmation. Celui utilisé au lycée comporte un éditeur de texte destiné à l'écriture de programmes (ou *scripts*), des raccourcis permettant de démarrer le compilateur, ainsi qu'un débogueur.

Nous vous conseillons d'installer Python à votre domicile. Le plus simple est alors d'installer la même version que celle sur laquelle vous travaillerez au lycée : allez pour cela à l'adresse `http://www.pyzo.org`, et télécharger la version compatible avec le système d'exploitation que vous utilisez (Linux, Windows ou OS X). Vous pourrez alors utiliser Python via IDE sans même avoir besoin d'installer le programme, celui-ci fonctionnant en live (vous pouvez en particulier copier le dossier sur une clé USB, puis utiliser Python à partir de n'importe quel ordinateur sur lequel vous brancherez cette clé).

1 C) Terminologie usuelle de l'informatique et notations

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.

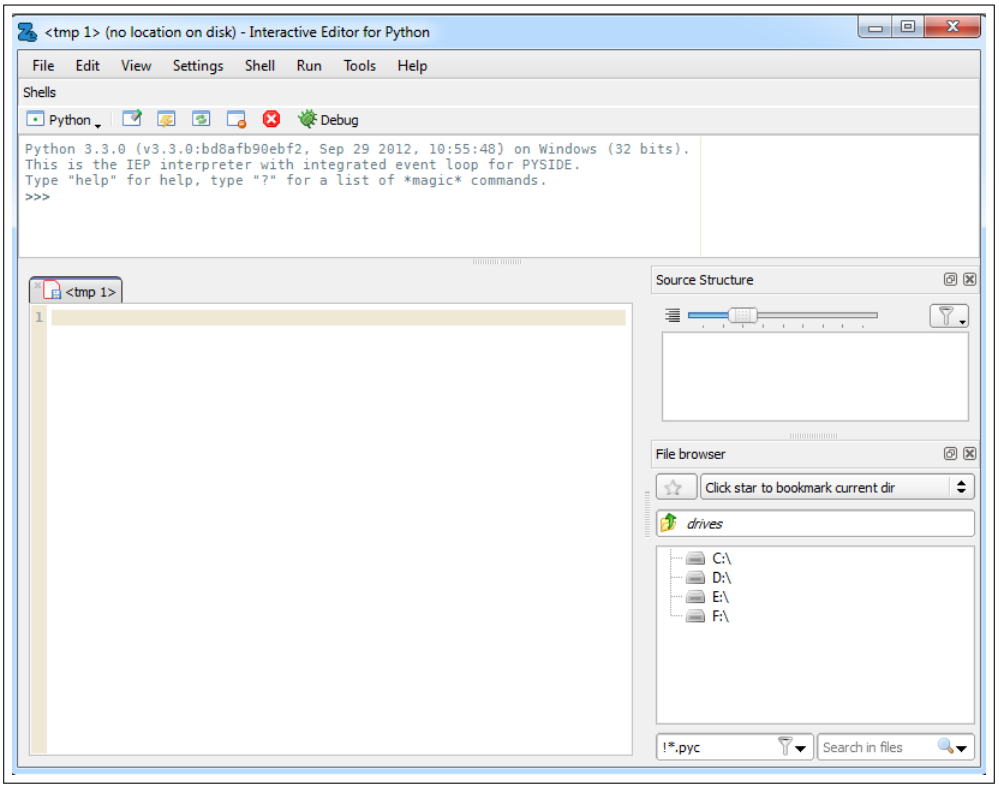
Chaque objet de ce type (ligne de code, primitive ou instruction) sera dénoté par un mot écrit dans `cette police`.

Entrer une ligne de code signifie *l'écrire dans la console puis appuyer sur la touche « entrée »* .

2 L'environnement

Une fois que vous aurez cliqué sur le lien vers Pyzo présent sur le bureau de votre machine, vous verrez apparaître la fenêtre ci-dessous. Celle-ci se décompose en plusieurs éléments, dont deux sont fondamentaux :

- la fenêtre du haut, où la dernière ligne commence par le symbole >>>, qui est appelée la *console* (ou *shell* en anglais) ;
- la fenêtre en bas à gauche, encore vierge au départ et portant par défaut le nom <tmp 1>, qui est l'éditeur de texte.



2 A) La console

Le symbole >>> est appelé le *signal d'invite* ou le *prompteur*, et indique que Python est prêt à mettre en œuvre une commande (on parlera dans la suite de *l'évaluation* d'une commande). Par exemple, on peut utiliser de suite l'interpréteur comme une simple calculatrice de bureau, chaque ligne de calcul se validant avec la touche « entrée » .

Exemple 2.1.

```
>>> 1+2
3
>>> 3/3-4*(1/4)
0.0
>>> 2/3
0.6666666666666666
```

```
>>> ln(1)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
NameError: name 'ln' is not defined
```

Commentaires :

2 B) Utiliser l'éditeur de texte

- L'éditeur de texte permet également d'effectuer des calculs simples, la différence résidant dans le fait que lorsque l'on appuie sur la touche « entrée » , le curseur se contente d'aller à la ligne. Pour exécuter les instructions écrites, on clique alors sur l'onglet « Exécuter » , puis sur « Exécuter le fichier » . Mais on voit alors que, si la console précise bien qu'elle a exécuté le fichier, elle n'affiche aucun résultat ! Pour voir apparaître le résultat, il faut ainsi entrer `print(2+3)` plutôt que `2+3`. L'instruction `print` demande en effet à la machine d'afficher le résultat dans la console.
- Il faut encore noter que la commande « Exécuter la sélection » permet de n'exécuter que les lignes de code mises en surbrillance dans l'éditeur de texte. D'autre part, on délimite des *cellules* dans l'éditeur de texte en utilisant le symbole `##`, et on peut ensuite choisir de n'exécuter que l'une de ces cellules.
- Le symbole `#` permet d'insérer des commentaires dans l'éditeur de texte, qui ne seront pas exécutés dans la console.
- L'éditeur de texte offre ainsi les mêmes possibilités que la console, mais il est plus pratique dès que l'on a besoin d'écrire des programmes de longueur supérieure à une ligne.
- Enfin, notons que l'on peut sauvegarder le contenu de l'éditeur de texte dans un fichier (dont l'extension est *.py*) en cliquant sur l'onglet « Fichier » puis sur « Enregistrer sous... » . On peut ensuite ouvrir ce fichier avec l'IDE de son choix.

Remarque 2.2.
Lors de l'utilisation de Pyzo, bien que l'utilisation des menus avec la souris soit très pratique, il est important de se familiariser avec les raccourcis clavier pour travailler rapidement en TP.

Dans la console :

- les flèches « haut » et « bas » permettent d'afficher les lignes tapées précédemment ;
- la touche « Entrée » permet d'exécuter la ligne de code.

Dans l'éditeur de texte, ce sont principalement des combinaisons avec la touche « Ctrl » (en bas à gauche et en bas à droite du clavier) qui seront utiles :

- la touche « Entrée » permet de sauter une ligne ;
- la combinaison « Ctrl + Z/Y » permet d'annuler/refaire la dernière action ;
- la combinaison « Ctrl + S » permet de sauvegarder ce qui a été écrit dans l'éditeur (si aucun nom de fichier n'a été précisé, une fenêtre de sauvegarde s'ouvre) ;
- la combinaison « Ctrl + X/C/V » permet de couper/copier/coller une ligne préalablement mise en surbrillance ;
- la combinaison « Ctrl + R/T » permet de commenter/décommenter des lignes de code.

Il existe beaucoup d'autres raccourcis clavier que l'on peu consulter en cliquant sur les différent menus de l'IDE comme « Fichier » par exemple.

3 Affectation de variables

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.

Exemple 3.1.
Si l'on souhaite donner la valeur 2 à la variable `x`, 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.

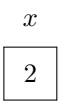
```
>>> x=2
>>> x
2
>>> 1+x
3
```

```
>>> y=x+2
>>> y
4
```

Dans l'exemple ci-dessus, il faut comprendre que la machine évalue l'instruction `y = x+2` de la droite vers la gauche : elle évalue d'abord `x+2`, ce qui donne 4, puis elle mémorise cette nouvelle valeur dans la variable de gauche, ici `y`.

Mais que se passe-t-il dans la machine lorsque l'on réalise une affectation ? Pour comprendre cela, on doit déjà admettre (nous en reparlerons plus tard) que la mémoire d'un ordinateur est composée de différentes cases (appelées *cases mémoires*) dans lesquelles on peut ranger des valeurs (on peut appliquer le même modèle à un CD vierge, dont on remplirait les différentes cases lorsque l'on y graverait par exemple des chansons).

On peut alors donner l'image suivante (dont on verra plus tard qu'elle ne correspond pas exactement à la réalité de la machine) : chaque lettre (par exemple `x`) représente une *boîte*, et la ligne de code `x=2` a pour effet de mettre la valeur `2` dans la boîte portant l'étiquette `x`.



Remarque 3.2.

Exemple 3.3.
Considérons les lignes de codes suivantes :

```
>>> x=1
>>> y=x+1
>>> y
2
```

```
>>> x=2
>>> y
2
```

Commentaires :

Pour supprimer la valeur contenue dans une variable, on peut utiliser la primitive `del`.

```
>>> y=3
>>> y
3
>>> del(y)
```

```
>>> y
Traceback (most recent call last):
  File "<console>", line 1, in <module>
NameError: name 'y' is not defined
```

Commentaires :

Exemple 3.4.

```
>>> alpha12=3
>>> alpha12=alpha12+1
>>> alpha12
4
```

```
>>> if=3
  File "<console>", line 1
    if=3
      ^
SyntaxError: invalid syntax
```

Commentaires :

Exemple 3.5.

```
>>> x=3
>>> 2*x
6
>>> 2x
  File "<console>", line 1
    2x
     ^
SyntaxError: invalid syntax
```

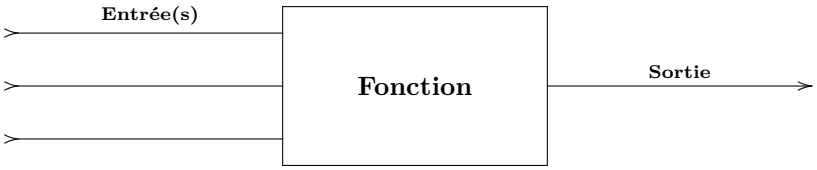
```
>>> n=2
>>> n*x
6
>>> nx
Traceback (most recent call last):
  File "<console>", line 1, in <module>
NameError: name 'nx' is not defined
```

Commentaires :

4 Les fonctions

4 A) Définition d'une fonction avec l'instruction return

On appelle *fonction* tout programme qui, à partir d'un ou plusieurs paramètre(s) appelés *entrée(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.

On peut définir dans l'éditeur de texte des fonctions en utilisant une syntaxe appropriée, et celles-ci pourront ensuite être appelées grâce à leur nom, sans que l'on ait besoin à chaque appel de redéfinir la fonction.

La syntaxe permettant de déclarer une fonction en Python dans l'éditeur de texte est la suivante.

```
def f(x):
        #Calcul d'une valeur rangee dans la variable y
        return(y)
```

Exemples 4.1.
Voilà quelques exemples de fonction. Notez au passage qu'on peut indifféremment définir des fonctions d'une ou plusieurs variables, ou utiliser une fonction déjà définie pour en définir une nouvelle : ci-dessous, on utilise la fonction `f` pour définir la fonction `g`.

```
def f(n):
    a=n+1
    return(a)
##
def g(x,y):
    b=x*y
    return(b)
##
def h(x,y,n):
    c=f(n)+g(x,y)
    return(c)
```

```
>>> f(4)
5
```

```
>>> g(2,3)
6
```

```
>>> h(2,3,4)
11
```

Il est important ici de noter que, lorsque nous sommes allés à la ligne après le code `def f(x):` l'éditeur de texte a immédiatement inséré des espaces (en l'occurrence 4) de sorte que la seconde ligne (et les suivantes) ont été décalées vers la droite. Ce décalage, appelé *indentation*, est nécessaire : s'il est omis, l'exécution du programme renverra un message d'erreur.

On voit ici que les indentations font partie intégrante de la syntaxe, cela peut sembler contraignant pour le moment, mais on verra rapidement que cela permet de lire plus facilement les codes Python. Plus généralement, les indentations sont utilisées dans tous les blocs d'instructions (branchement conditionnel (voir la suite), boucles (voir les cours suivants)).

L'indentation est réalisée automatiquement par l'éditeur de texte, mais en cas de besoin on peut la modifier à la main : une indentation correspond à 4 espaces, ou à une seule tabulation (la touche « tabulation » est celle se trouvant au-dessus de la touche « verrouillage numérique »).

Exercice 4.2.
Écrivons des déclarations plus courtes des fonctions `f,g,h` ?

Remarque 4.3.
Il est possible de définir une fonction directement dans la console ; c'est alors le symbole `:` qui permet d'aller à la ligne sans évaluer la ligne déjà tapée (c'est grâce à ce symbole que la machine comprend que nous allons définir un bloc d'instructions). On peut alors poursuivre la définition de la fonction en travaillant comme dans l'éditeur de texte, à ceci près que dans la console il faut entrer les indentations à la main (la machine ne les gère pas d'elle-même).

4 B) Variables globales et locales

Observons attentivement l'exemple suivant.

Exemples 4.4.

```
def f(y):
    m=y+1
    return(m)
```

```
>>> m=0
>>> f(1)
2
>>> m
0
```

On voit ainsi que la valeur de `m` n'est pas modifiée lors de l'appel de `f`, alors que pourtant le code de `f` semble bien modifier `m` ! Cela s'explique par le fait qu'il y a deux variables différentes portant le nom `m` :

- la première est celle qui a été définie dans la console, et dont la valeur est `0`, que l'on qualifie de *variable globale*,
- la seconde est celle définie dans la fonction `f`, et dont la valeur est `y+1`, que l'on qualifie de *variable locale*.

L'exemple ci-dessus montre que ces deux variables coexistent sans se chevaucher : Python est capable de faire la différence entre les deux. Lorsque la fonction est appelée, un ensemble de variables locales appelé *espace local* est créé, de sorte que cet espace soit disjoint de l'*espace global* (formé de toutes les variables globales), et toutes les affectations relatives à l'évaluation de la fonction auront lieu dans l'espace local : il n'y a donc pas de chevauchement possible.

Ce comportement peut vous sembler étrange, mais il est nécessaire pour permettre à des programmeurs de travailler en équipe : ceux-ci peuvent *in fine* mutualiser leurs fonctions sans se préoccuper des variables qu'ils ont chacun utilisés à l'intérieur de leurs fonctions respectives.
Observons enfin l'exemple suivant.

Exemples 4.5.

```
def aire(R):
    a=pi*R**2
    return(a)
```

```
>>> pi=3.14
>>> aire(1)
3.14
```

```
>>> aire(1)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
    def f(x):
  File "<tmp 1>", line 22, in aire
NameError: global name 'pi' is not defined
```

Commentaires :

Exemple 4.6.
Dans la fonction `h` déclarée à l'exemple 4.1, les variables `f` et `g` étaient déjà des variables globales.

4 C) Définition d'une procédure

La définition d'une fonction peut se terminer par l'utilisation de la primitive `print` plutôt que `return` ; cela donne alors la syntaxe suivante :

```
def f(x):
        #Calcul d'une valeur rangee dans la variable y
        print(y)
```

La différence est que dans ce cas la fonction ne renverra plus une valeur `y` mais se contentera de l'afficher à l'écran. Pour comprendre ce que cela change, observons l'exemple suivant :

Exemples 4.7.

```
def f(n):
    return(n+1)
```

```
>>> f(1)
2
>>> f(1)+1
3
```

```
def g(n):
    print(n+1)
```

```
>>> g(1)
2
>>> g(1)+1
2
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: unsupported operand type(s)
for + : 'NoneType' and 'int'
```

On voit ainsi que si ces deux fonctions calculent effectivement la même chose, la seconde `g` ne pourra pas être utilisée à l'intérieur d'un calcul, car son résultat n'est pas accessible : il ne fait qu'apparaître à l'écran.

Les programmes qui, comme la fonction `g` ci-dessus, ne renvoient rien sont appelés des *procédures*. Si en général on aura plutôt tendance à déclarer des fonctions que des procédures, on rencontrera plus tard de nombreuses procédures parmi les primitives de Python.

5 Conditions booléennes

En l'honneur du logicien anglais *Georges Boole*, les deux valeurs logiques *Vrai* et *Faux* sont appelées des *booléens*.

5 A) Les booléens

Dans le langage de Python, le code `x=2` est une instruction (intuitivement il s'agit d'un ordre donné à la machine : « mets la valeur `2` dans la variable `x` ») ; par contre, le code `x==2` correspond à une *valeur booléenne*, qui est *True* ou *False* suivant la valeur contenue dans la variable `x`.

Exemples 5.1.

```
>>> x=1
>>> x==1
True
```

```
>>> x==0
False
```

Avec Python, on peut manipuler les opérateurs de comparaison usuels : les instructions associées sont listées dans le tableau suivant.

==	égal à
!=	différent de
<	strictement inférieur à
<=	inférieur ou égal à
>	strictement supérieur à
>=	supérieur ou égal à

Exemples 5.2.

```
>>> a=1
>>> a!=2
True
```

```
>>> a<=1
True
>>> a>1
False
```

5 B) Les connecteurs logiques

Python connaît également les connecteurs logiques : les instructions associées sont listées dans le tableau suivant.

`and`	et
`or`	ou
`not`	négation

Exemples 5.3.

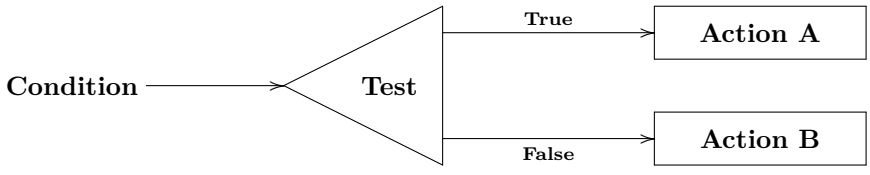
```
>>> 0==1 and 1==1
False
>>> 0==0 and 1==1
True
```

```
>>> 0==1 or 1==1
True
>>> not 0==1
True
```

Notez au passage que ces fonctions ne nécessitent pas de parenthèses (mais il est possible d'en mettre).

6 Branchement conditionnel

Les instructions élémentaires nécessaires à la programmation sont bien entendu disponibles sous Python, comme le branchement conditionnel *si..alors..sinon* (*if..then..else* en anglais) qui peut être schématisé comme suit :



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 l'action A ;
- sinon elle effectue l'action B.

Voilà la syntaxe Python correspondant à cette instruction :

```
if Condition:
    Action A
else:
    Action B
```

La spécificité de la syntaxe Python est que l'instruction *then* n'apparaît pas : elle est remplacée par une indentation. De manière analogue à ce que l'on avait vu avec `def`, on a affaire à un bloc d'instructions : il est donc nécessaire de respecter l'indentation telle qu'elle est représentée ci-dessus, sinon la console renverra un message d'erreur à l'exécution du programme.

Si on souhaite inclure cette structure dans un programme plus grand, il faudra après `Action B` revenir à la ligne sans indentation pour indiquer à la machine que le branchement conditionnel est terminé, et que l'on souhaite continuer l'écriture du programme.

Implémenter une fonction mathématique f, c'est écrire un programme qui permet de calculer effectivement la valeur de $f(x)$ pour tout x.

Exemples 6.1.

Le programme ci-dessous implémente la fonction mathématique définie par $f(x) = \begin{cases} 0 & \text{si } x = 0 \\ 1 & \text{si } x \neq 0 \end{cases}$.

```
def f(x):
    if x==0:
        y=0
    else:
        y=1
    return(y)
```

```
>>> f(0)
0
>>> f(2)
1
```

Observez bien la différence entre `=` et `==` dans l'exemple ci-dessus : `y=1` est une affectation, c'est-à-dire un ordre donné à la machine, alors que `x==0` est un booléen (soit Vrai soit Faux).

Remarque 6.2.
Il est important d'avoir du recul concernant les indentations effectuées par l'éditeur de texte : celui-ci ne détecte pas le moment où vous souhaitez passer de *Instructions A* à *else :*, et vous proposera donc une indentation de trop qu'il faudra absolument supprimer (sinon l'exécution du programme vous renverra un message d'erreur) !

Bien entendu, il existe une version simplifiée de cette syntaxe, pour le cas où l'on n'a pas besoin d'effectuer une action particulière si la condition n'est pas vérifiée : on peut alors simplement se passer de l'instruction `else` et des instructions qui la suivent.

```
if Condition:
    Action
```

Exemple 6.3.
Le programme ci-dessous est une seconde implémentation de la fonction f définie à l'exemple 6.1.

```
def f(x):
    y=1
    if x==0:
        y=0
    return(y)
```

Les choses se compliquent lorsque l'on cherche à *composer* (on dira plutôt *imbriquer*) plusieurs *if*.

Exercice 6.4.

Dans cet exercice, on cherche à implémenter la fonction définie par $g(x) = \begin{cases} 0 & \text{si } x = 0 \\ 1 & \text{si } x > 0 \\ -1 & \text{si } x < 0 \end{cases}$.

1. Écrire un programme convenant qui comporte trois fois l'instruction `if`.
2. Écrire un programme convenant qui comporte deux branchements conditionnels imbriqués (on prendra garde à l'indentation).

Il y a enfin une version plus complexe du branchement conditionnel, pour le cas où on souhaite distinguer plus de deux cas : cela permet d'éviter le type de problème décrit dans l'exemple ci-dessus.

```
if Condition1:
    Action1
elif Condition2:
    Action2
.
.
.
elif ConditionK:
    ActionK
else:
    Action(K+1)
```

Dans le schéma ci-dessus, il faut comprendre l'instruction `elif` comme signifiant *else-if*. Ainsi, le code

```
if C1:
    A1
elif C2:
    A2
else:
    A3
```

s'évalue de la manière suivante :

- si la condition `C1` est vraie, alors c'est l'action `A1` qui est évaluée ;
- si la condition `C1` est fausse et la condition `C2` est vraie, alors c'est l'action `A2` qui est évaluée ;
- si les conditions `C1` et `C2` sont fausses, alors c'est l'action `A3` qui est évaluée.

Exemple 6.5.

Voilà une nouvelle implémentation de la fonction définie par $g(x) = \begin{cases} 0 & \text{si } x = 0 \\ 1 & \text{si } x > 0 \\ -1 & \text{si } x < 0 \end{cases}$.

```
def g(x):
    if x<0:
        y=-1
    elif x==0:
        y=0
    else:
        y=1
    return(y)
```

Remarque 6.6.
La méthode qui en général induit le moins en erreur est celle consistant à utiliser « le plus de `if` et le moins de `else` possible », c'est-à-dire celle vue dans la question 1 de l'exercice 6.4. Dans un premier temps, c'est donc celle que je vous conseillerais d'utiliser.

Il reste encore une dernière chose à voir, dans la classe de « ce qu'il ne faut pas faire. »

Exemple 6.7.
Considérons que, étant données deux fonctions g, h on veut implémenter une fonction f définie par :

$$f(x) = \begin{cases} g(x) & \text{si } x \geqslant 0 \\ h(x) & \text{si } x < 0 \end{cases}.$$

Les deux programmes suivants conviennent :

```
def f1(x):
    if x>=0:
        y=g(x)
    else:
        y=h(x)
    return(y)
```

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

Commentaires :

Il est important de prendre dès le départ de bonnes habitudes de programmation, et de garder à l'esprit la question de la vitesse d'évaluation du programme.