10 De Java vers Python

Cette annexe a été rédigée pour le lecteur connaissant le langage Java et souhaitant s'initier à Python, mais aussi pour le programmeur Python souhaitant utiliser le système multimédia Processing en mode Python avec Processing.py, alors qu'il fonctionne pour l'essentiel en mode Java. Il vise notamment les utilisateurs Python de la librairie audio **Beads** *programmée en Java, dans laquelle nous mettrons le pied, ou plutôt l'oreille, au § 11.136.*

10.1 Le typage : fort ou dynamique ?

Le cas de Python

En Python, les valeurs sont typées. Par exemple `2021` et `-23` sont de type `int`, `-56.894` ou `23.0` sont de type `float`, `True` est de type `bool`, `[4,-5,True,0.5]` est de type `list`, la valeur du mot `abs` est de type *built-in function*, etc. Cependant les variables comme `x`, `y` ne sont pas typées au sens où leur valeur peut changer de type au cours de l'exécution. Par exemple, il est légal d'écrire :

```
>>> x = 2021            # x est un int
>>> x = 2021 + 0.5      # x est devenu un float
```

faisant passer la variable `x` d'une valeur entière à une valeur flottante. On peut improprement dire qu'elle était de type entier, puis est devenue de type flottant. Les programmeurs disent plutôt qu'elle est **dynamiquement typée** : son type est celui de sa valeur et peut changer dynamiquement. Pire que cela, on peut détruire une variable avec le mot-clé `del` :

```
>>> x = 2021            # x est une variable globale
>>> del x               # suppression de l'espace de noms courant
>>> x                   # la variable globale x n'existe plus
NameError: name 'x' is not defined
```

Du coup, les fonctions sont elles aussi dynamiquement typées. C'est au programmeur de s'assurer que le code de sa fonction respecte son domaine de définition.

```
def f(x,y):
    return x+2*y
```

On ne peut pas déduire du texte de cette fonction `f` sa **signature** $E \to F$, c'est-à-dire son ensemble de départ E et celui d'arrivée F. Si x et y sont entiers, elle retourne un entier. Si l'un des deux est flottant, elle retourne un flottant. Si les deux sont des chaînes de caractères, elle retourne une nouvelle chaîne. À la lecture du texte de `f`, rien ne garantit que `x,y` soient des nombres ! Il se peut même que `x` et `y` soient des objets d'une classe (comme `str`) qui redéfinit les opérateurs `+` et `*` à travers des méthodes spéciales `__add__` et `__mul__`.

Une **conversion de type** en Python consiste à prendre une expression, à l'évaluer et à produire soit une erreur (conversion impossible), soit une valeur d'un autre type, c'est-à-dire d'une autre classe. Par exemple, la conversion d'un entier en flottant se ferait par `float(23)` qui produirait `23.0`, et inversement par `int(23.8)` qui produirait `23`. Ou encore la conversion d'une liste `L` en tuple avec `tuple(L)`. Une conversion s'écrit en général sous la forme `typ(expr)` si `typ` est le nom de la classe d'arrivée.

Conclusion sur le typage dynamique, qui vaut ce qu'elle vaut : c'est plutôt *agréable*. MAIS : nous risquons de faire des erreurs de typage qui ne se verront (ou pas) qu'au moment de l'exécution. Sans compter une perte d'efficacité par-rapport aux langages industriels plus fortement typés (comme Java ou C) disposant de compilateurs capables de profiter des annotations de type pour produire du code machine très rapide. Du coup, les langages dynamiques comme Python disposent si besoin de nombreuses bibliothèques écrites en C (par exemple *Pymunk*), voire Java en ce qui concerne Processing (exemple : *Beads*). Nous avons même rencontré le dialecte Cython qui sait profiter du compilateur C.

Le cas de Java

En Java, les variables sont **fortement typées**. Une variable doit être **déclarée** d'un certain type, et elle ne pourra plus en changer. Si `x` est un entier, il sera toujours un entier :

```
int x;                  // je déclare une variable entière en C/Java
x = 2021                // bien typé
x = x + 12;             // bien typé
float y = x + 0.5;      // bien typé !
x = y + 1;              // non : erreur de typage...
```

Remarque — À la quatrième ligne, Java a procédé à une **conversion automatique de type** : l'entier `x` doit pouvoir être considéré comme flottant lorsque c'est nécessaire ! Sa valeur `2033` est lue comme `2033.0`.

Du coup, une fonction doit obligatoirement **déclarer** sa signature. La notation mathématique $f : x \in \mathbb{Z} \mapsto f(x) \in \mathbb{R}$ s'écrit dans le Java de Processing (qui prend pour réels les `float` en simple précision) :

```
float f(int x) { corps de la fonction }
```

Par exemple la fonction $f : (x \in \mathbb{R}, n \in \mathbb{N}) \mapsto x^n + 1 \in \mathbb{R}$ s'écrira :

```
float f(float x, int n) {
      return pow(x, n) + 1;      // pow(x,n) pour x^n en Processing
}

void setup() {
     println(f(2, 10));          // affiche 1025.0
}
```

Vous avez remarqué dans `setup` que le paramètre `x` de `f` a reçu la valeur `2` qui est un `int` et non un `float`. Comme plus haut, Java a procédé à une **conversion automatique de type**. On peut écrire `float x = 2;` mais ce `2` sera compris comme `2.0` (`float` par défaut en Processing). De même un caractère[1] comme `'A'` sera automatiquement converti en entier dans une expression arithmétique, l'expression `'A' + 1` vaut 66.

La **conversion explicite de type** (en anglais *type cast*) consiste à convertir une valeur d'un type donné en une valeur d'un autre type requis par le contexte de calcul. En Java, la conversion du caractère `'A'` en code numérique `65` se fait par l'expression `(int)'A'`, que Python rédige `int('A')`. De manière générale, l'expression `(typ)expr` retourne la valeur de l'expression `expr` convertie en une valeur de type `typ`, lorsque cette conversion est autorisée. Sauf cas exceptionnel, le contexte permet de comprendre ce que le programmeur avait dans la tête.

10.2 Les boucles

Java propose trois sortes de boucles : `for`, `while` et `do...while`. Commençons par la boucle `for` qui accepte aussi la forme `for (x : L) {...}` à la place du `for x in L:...` de Python (les `ArrayList` de Java sont les listes de Python).

Java	Python

`for(int i=10; i<20; i++) {`
`  System.out.println(i);`
`}  // i n'existe plus`
	`for i in range(10,20):`
	`  print(i)`
	`# i vaut 19 en sortie de boucle`

Remarque — La variable `i` est **locale à la boucle** en Java, mais pas en Python.

1. Java possède un type caractère `char` à part entière contrairement à Python.

La boucle `while` de Java présente une simple différence cosmétique par-rapport à celle de Python.

Java	Python

`while (x < N) {`
`  System.out.println(f(x));`
`  x++;`
`}`
	`while x < N:`
	`  print(f(x))`
	`  x += 1`

Enfin, la boucle `do...while...` de Java n'existe pas en Python, mais on peut la simuler. Elle débute par une action alors que la boucle `while` débute par un test.

Java	Python

`do {`
`  System.out.println(f(x));`
`  x++;`
`} until (x > N);`
	`while True:`
	`  print(f(x))`
	`  x += 1`
	`  if x > N: break`

10.3 Les listes

La classe `ArrayList` de Java fournit des fonctionnalités similaires à la classe `list` de Python. Il ne s'agit pas de *listes chaînées* à la Lisp, mais de tableaux extensibles à la demande. À une telle liste est dévolu, de manière transparente, un certain espace mémoire. Une fois cet espace mémoire rempli, un nouvel espace mémoire plus grand est alloué à la liste afin d'ajouter de nouveaux éléments (en queue de liste). Le coût d'un ajout en queue est donc en $\mathcal{O}(1)$ *amorti* sur un grand nombre d'ajouts. Java note `L.add(x)` ce que Python écrit `L.append(x)`.

Bien que l'on puisse stocker des éléments de types distincts dans une `ArrayList` (ils seront tous de classe `Object`), il est usuel de n'y placer que des valeurs de même type, par exemple des entiers. Pour cela, la classe enveloppante `Integer` a été introduite car les entiers ne forment pas une classe en Java (ici Python marque un point pour la simplicité).

Java

`ArrayList<Integer> L = new ArrayList<Integer>();`
`for(int i=0; i<10; i++) {`
`  L.add(i*i);` *//passage automatique de* `i*i` *à* `Integer(i*i)`
`}`
`for(int x : L) {` *//et inversement*
`  System.out.print(x + " ");`
`}`

`0 1 4 9 16 25 36 49 64 81`

Python

`L = []`
`for i in range(10):`
`  L.append(i*i)`
`for x in L:`
`  print(x,end=' ')`

`0 1 4 9 16 25 36 49 64 81`

10.4 Les classes d'objets

Si l'on reste à un niveau élémentaire, les modèles Python et Java sont assez proches. Le code Java se comprend si l'on connaît la structure d'une classe Python. La notation pointée `objet.attribut` ou `objet.methode(...)` est similaire.

Voici quelques différences loin d'être exhaustives, j'en dirai le minimum vital pour être capable de lire les *Lessons* fournies en Java comme exemples de la bibliothèque audio *Beads*, et qu'il s'agira de traduire en Python au § 11.137.

Objet ou pas objet ?

En Python tout est objet, même les nombres et les fonctions : les types sont les classes. Java distingue par contre les classes et les *types primitifs* comme les nombres (`int`, `float`, *etc.*) et les booléens (`boolean`). À chaque type primitif est associé une classe enveloppante : `Integer`, `Float`, `Boolean`, *etc.*

Les fonctions Java sont des *méthodes* (d'instance ou de classe) définies obligatoirement dans le texte d'une classe. Pour passer une fonction en argument à une autre fonction, sans utiliser les nouveautés de Java 8, il faut la penser comme un objet (voir les classes anonymes plus bas).

Structure d'une classe

En Java, les attributs d'instance d'un objet sont déclarés en tête de classe puis initialisés dans le constructeur dont le nom est celui de la classe, alors qu'en Python ils sont déclarés et initialisés en général dans la méthode `__init__`.

L'objet courant `self` de Python se nomme `this` en Java. Ce dernier n'est PAS en premier paramètre dans le constructeur ni dans les méthodes d'instance en Java, il est passé implicitement et l'accès à un attribut `rayon` d'un objet, au sein d'une méthode d'instance, peut s'écrire simplement `rayon` au lieu de `this.rayon`, alors qu'en Python l'usage de `self` dans `self.rayon` est obligatoire !

Python est beaucoup moins paranoïaque que Java sur la sécurité. Les attributs sont en Python généralement publics, il n'y a pas besoin de méthodes `get/set` pour y accéder ou les modifier. La notation `objet.attribut` suffit en général. Sauf peut-être lorsqu'on utilise une librairie écrite en Java !

Le droit d'accès à un attribut ou méthode en Java peut être explicitement `public` (ouvert à tous), `private` (restreint à la classe), ou `protected` (autorisé aux sous-classes). En l'absence de déclaration explicite, l'accès est réduit au paquetage courant dans lequel est plongé la classe.

Java	Python

`class Cercle {`
`  static int nbCercles = 0;`
`  private float r;`
 	
`  Cercle (float r) {`
`  this.r = r;`
`  nbCercles++;`
`  }`
 	
`  public String toString() {`
`  return "Cercle("+r+")";`
`  }`
 	
`  float getR() {`
`  return r;`
`  }`
`}`
	`class Cercle:`
	`  nb_cercles = 0`
	
	`  def __init__(self,r):`
	`  self.r = r`
	`  Cercle.nb_cercles +=1`
	
	`  def __repr__(self):`
	`  return 'Cercle('+str(self.r)+')'`
	
	`# accès au rayon du cercle c par c.r`

Dans les deux cas, la variable de classe contenant le nombre de cercles (`nbCercles` pour Java, et `nb_cercles` pour Python, notez les conventions d'écriture) est obtenue depuis l'extérieur de la classe en la préfixant par le nom `Cercle` de la classe. Pour exprimer que `Oiseau` est une sous-classe de `Animal`, on écrit en Python :

```
class Oiseau(Animal):
```

et en Java :

```
class Oiseau extends Animal {
```

Les sous-classes anonymes

Le langage Java comporte un concept intéressant, celui de **sous-classe anonyme**. Illustrons-le avec le moyen traditionnel [2] de considérer une fonction comme un objet (que l'on pourra passer en argument).

Au fond, *qu'est-ce qu'une fonction sinon un objet capable de calculer sa valeur en un point* ? Transformons cette phrase en une classe abstraite et pour être complet, ajoutons un attribut protégé `x`.

2. Nous nous bornons à ce moyen simple qui était utilisé avant la version Java 8 qui offre une classe `Function`, moyen que l'on trouve souvent dans les fichiers d'exemples en Java de la bibliothèque audio *Beads* qui contient une classe `Function.java` qui n'est pas celle de Java 8.

```
/* voir le sketch 'fonction_java' en Processing parmi les fichiers
de ce livre */
public abstract class Fonction {
  protected float x = 10;         //  accessible aux sous-classes en Java
  public abstract float calcule(float t);  // à implémenter plus tard
}
```

Une « fonction » concrète $t \mapsto t^2+x$ sera vue comme un objet de la classe `Fonction` *sauf que* sa méthode d'instance `calcule` sera particularisée.

```
Fonction f = new Fonction() {              // f est une Fonction
  public float calcule(float t) {          // sauf qu'elle calcule ainsi...
    return t*t + x;                 // x est visible depuis la classe mère !
  }
};
System.out.println(f.calcule(2))           // affiche 14.0
```

C'était la solution adoptée en Java avant que ce dernier ne (re)découvre dans sa version 8 les lambda-fonctions de Lisp (1960). Il n'est jamais trop tard...

Mais Python (Jython) ne possède pas de manière simple les classes anonymes [3], alors comment faire ? Il suffit de briser l'anonymat, et de nommer explicitement la sous-classe. La bibliothèque audio *Beads* contient un fichier `Function.java` définissant *grosso modo* la classe `Fonction` ci-dessus, où l'attribut protégé `x` représente une liste d'unités génératices (UGens). La traduction en Python du sketch `Lesson03FMSynthesis` des exemples de *Beads* fait l'objet d'une question au § 11.137.

Remarque — Dernière minute. L'attribut `x` est passé du statut `protected` à celui de `public` dans le fichier source `Function.java` livré avec la bibliothèque *Beads* pour Processing.py. Ceci n'enlève rien à ce qui a été dit ci-dessus pour Java, mais a une importance pour Processing.py à cause d'un comportement anormal des attributs `protected` en Jython. Voir la note au bas de la page 312.

3. En fait, la documentation de la primitive `type` de Python montre qu'il existe une syntaxe adaptée à la construction dynamique d'un type, c'est-à-dire d'une classe.