Partie VI - La programmation orientée objets

par Yves Bailly

Date de publication :

Dernière mise à jour : 22/11/2005

Les programmes que nous avons créés jusqu'ici déclaraient des variables, définissaient des fonctions agissant sur ces variables, tout cela un peu pèle-mèle. Si c'est acceptable pour un petit programme, cela devient rapidement ingérable dès que l'on s'attaque à une application d'envergure.

I - Le principe de l'objet

Le besoin d'organiser les choses, de regrouper ensemble les éléments liés logiquement, s'est rapidement imposé aux programmeurs.

Plusieurs techniques ont été élaborées pour permettre cette organisation. Par exemple, regrouper les fonctions en *modules* cohérents, auxquels un programme principal peut faire appel. Nous verrons plus tard cette notion de modules. Ici, nous allons étudier une autre technique, qui prend ses racines en 1967 avec le langage Simula, puis un peu plus tard avec Smalltalk : la *programmation par objets*.

Ce paradigme de programmation s'est véritablement imposé dans les années 1980, avec l'avènement du langage C++, extension du langage C pour introduire dans ce dernier les principes de la programmation objet. D'autres langages ont également été étendus pour fournir les outils objets : Ada, Pascal, même Basic sont des exemples. Des langages objet dès leur création ont également été conçus, comme Java, Ruby, ou notre ami Python. Aujourd'hui la programmation par objets est le paradigme de programmation le plus largement utilisé dans le monde.

Mais, finalement, qu'est-ce qu'un objet ? Il s'agit avant tout d'un mécanisme d'*abstraction*, qui repose sur un principe fondamental. La chose manipulée par le programme, qu'il s'agisse de la modélisation d'un objet physique comme une voiture ou d'un objet abstrait comme une fonction mathématique, est plus que la simple collection de ses caractéristiques. Pour modéliser correctement cette chose, il faut également prendre en compte les *méthodes*, les règles de son utilisation. Une voiture ne se réduit pas à une vitesse et une quantité d'essence : c'est également un accélérateur par lequel on agit sur ces deux quantités.

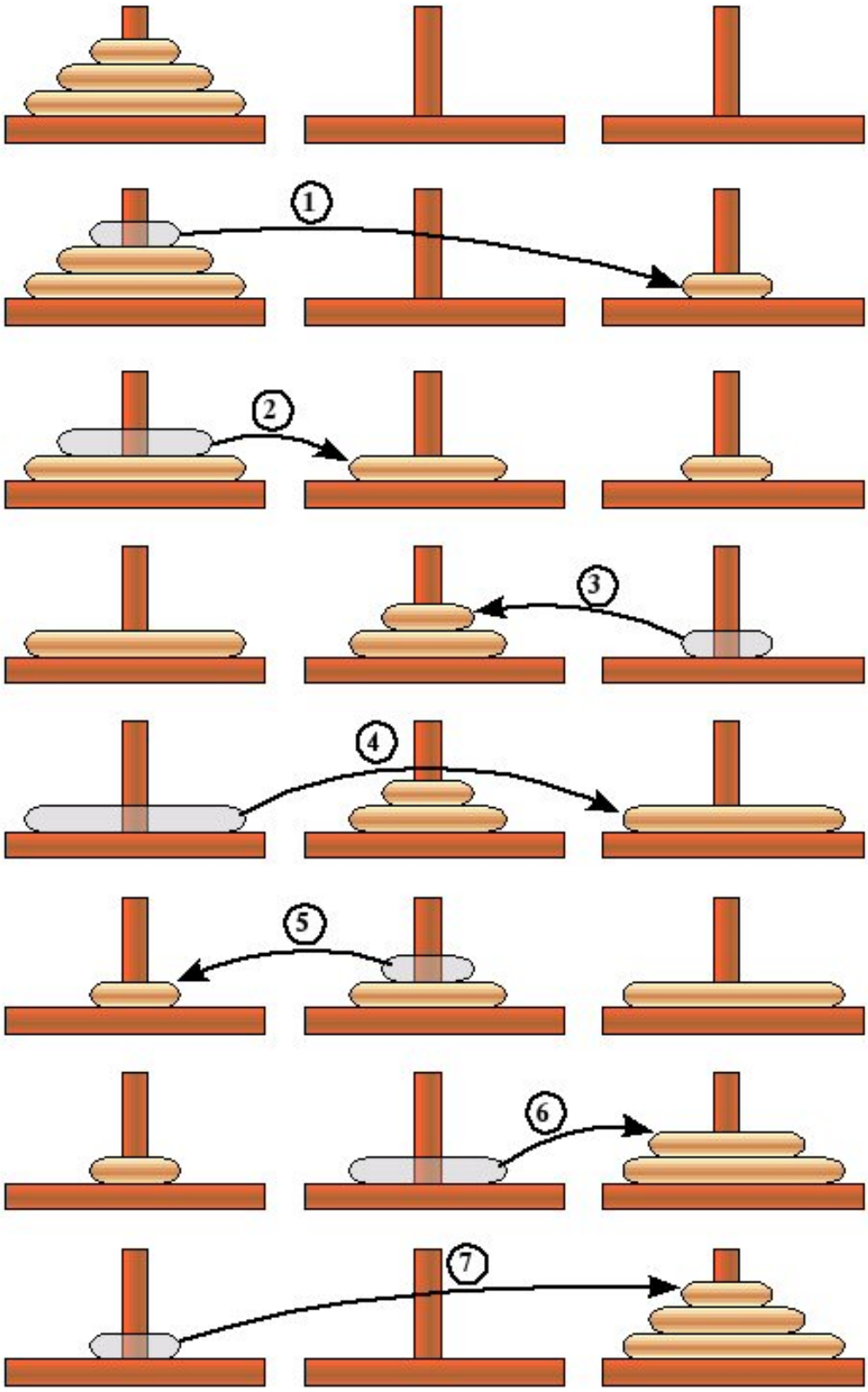
L'objet vise donc à fournir une représentation plus proche de la réalité de ce que l'on manipule : contenir en une même entité informatique, non seulement les *données* associées à la chose, mais aussi les moyens, les *méthodes* par lesquelles on peut agir sur cette chose.

II - Les tours de Hanoï

Pour découvrir et mettre en oeuvre les principes de la programmation par objets, ainsi que bien d'autres aspects de la programmation, je vous propose de nous lancer dans la réalisation d'un petit jeu : le jeu des tours de Hanoï. Ce jeu est inspiré d'une très ancienne légende.

Voilà une bien belle légende. Nous allons tenter de réaliser un programme permettant à un joueur de déplacer des disques d'une tour vers l'autre, en respectant les règles énoncées plus haut. Et cerise sur le gâteau, le programme devra proposer une solution, si le joueur ne la trouve pas - le programme devra donc être capable de résoudre le problème, et d'afficher cette résolution.

Première étape : comprendre à peu près de quoi il retourne. La légende nous parle de soixante-quatre disques, ce qui est assez considérable (nous verrons dans un instant pourquoi). La puissance des matériels disponible ne cessant de croître, il est fréquent d'envisager des programmes destinés à manipuler de grandes quantités de données, ou du moins nécessitant des quantités astronomiques de calculs. Dans ces situations, on commence par étudier le problème sur un ensemble plus petit. Pour notre exemple, prenons simplement trois disques. Saurez-vous trouver le bon enchaînement de mouvements ? L'illustration suivante donne la solution.



Nous avons dû effectuer sept mouvements. Et c'est la meilleure solution : il est impossible de résoudre le problème avec moins de mouvements. En fait, on peut démontrer (ce que nous ne ferons pas ici) que pour *n* disques, il faut effectuer au minimum 2*n*-1 mouvements. Question : combien les moines devront-ils effectuer de mouvements ? Nous pouvons demander cela à Python. Lancez l'interpréteur, et effectuez l'opération :

```
>>> 2**64-1
18446744073709551615L
```

C'est un très grand nombre ! En supposant que les moines effectuent un déplacement par seconde, combien leur faudra-t-il de temps, en années ? Facile, en considérant qu'une année est constituée de 365 jours :

```
>>> (2**64-1)/(3600*24*365)
584942417355L
```

C'est-à-dire un peu moins de 585 *milliards* d'années... ouf, la fin du monde n'est pas pour demain ! Incidemment, cet exemple montre que pour puissantes qu'elles soient, nos machines ne sont pas forcément capables de résoudre un problème simple : un petit programme écrit en C, assez bien optimisé, atteint péniblement 20 millions de mouvements par seconde sur mon Athlon 1Ghz. Il me faudrait donc environ 30000 ans pour obtenir la solution complète... Mais revenons à la programmation.

III - Modéliser la tour

Nous verrons plus tard comment résoudre ce problème, pour un nombre de disques raisonnables. Pour l'heure, l'objectif est de permettre à un utilisateur de jouer, pour trouver lui-même une solution.

A priori, le jeu comporte au moins deux objets : le disque et la tour. Pour simplifier un peu les choses, nous pouvons décider qu'un disque est simplement représenté par un entier, dont la valeur est sa taille. Dans l'exemple de trois disques précédent, au tout début du jeu, le petit disque au sommet sera donc représenté par l'entier 1, et le grand tout en bas par l'entier 3.

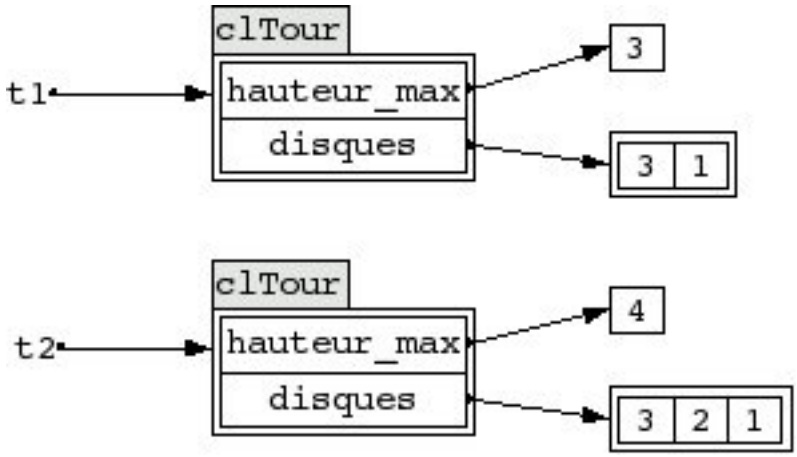
Cela étant posé, il nous faut représenter informatiquement une tour qui porte des disques. Nous allons pour cela définir un nouveau type, que nous nommerons **clTour**. De même que le type entier permet de représenter un nombre entier, ou le type chaîne de caractères permet de représenter une chaîne de caractères (du texte), notre type **clTour** permettra de représenter une tour dans le jeu. Dans le jargon de la programmation objet, dans cette situation de création d'un nouveau type, on parle plutôt de *classe* : nous allons donc définir une classe nommée **clTour**. Cette classe contiendra des données, comme sa hauteur maximale et la liste des disques qu'elle contient, ainsi que des fonctions permettant d'ajouter ou retirer un disque. Dans le contexte d'une classe, on désigne les fonctions par le terme de *méthodes*. Voici le début de sa déclaration, placée par exemple dans un fichier **hanoi.py** :

```
1 # -*- coding: iso8859-1 -*-
2 class clTour :
```

La déclaration d'une classe, c'est-à-dire d'un nouveau type objet, est introduite par le mot-clef **class**. Vient ensuite le nom du type, **clTour** dans notre exemple, puis les deux-points qui signalent le début d'un bloc. Ce bloc va contenir les déclarations des données de l'objet, ainsi que celles des méthodes (fonctions) propres à l'objet.

Pour fixer les idées, disons dès maintenant que la hauteur maximale d'une tour sera contenue dans une donnée nommée **hauteur_max**, tandis que la liste des disques empilés à un moment donné sur la tour sera contenue dans une donnée nommée **disques**.

Imaginons maintenant que nous ayons deux variables **t1** et **t2**, qui seraient du type **clTour** que nous avons défini. La première aurait une hauteur maximale de 3 disques, et contiendraient les disques de tailles 3 et 1. La deuxième, **t2**, pourraient contenir jusqu'à quatre disque, et contiendrait actuellement les disques de tailles 3, 2 et 1. On pourrait représenter la situation en mémoire ainsi :



Poursuivons le programme.

```
3   def __init__ (self, hauteur) :
4     self.hauteur_max = hauteur
5     self.disques = []
```

Voici la première méthode que nous rencontrons. Sa définition ressemble fort à celle d'une fonction, dont le nom serait ici **__init__**, et prenant deux paramètres nommés **self** et **hauteur**. Cette fonction est un peu particulière : elle est exécutée dès qu'une instance de la classe est créée. Nous reviendrons un peu plus loin sur ce sujet. Pour l'instant, retenez simplement que le paramètre **self** doit toujours apparaître en premier lorsque l'on définit une fonction dans un objet. Remarquez également l'indentation : nous sommes toujours dans le bloc annoncé par les deux-points de la déclaration de la classe **clTour**. C'est pourquoi la déclaration de la méthode est indentée, le code qu'elle contient étant encore plus décalé vers la droite.

Le rôle essentiel de cette méthode est de créer et de donner une valeur initiale « correcte » aux données que nous avons prévues de placer dans la classe.

Remarquez que les données définies plus tôt dans notre objet sont toutes précédées de « **self**. » : c'est une nécessité imposée par Python. *self* signifie *soi-même*. Si l'on reprend l'exemple des tours **t1** et **t2** plus haut, ce mot **self** permet à Python de « savoir » à quelle donnée on fait référence. Comment faire en effet pour désigner les disques de la tour t1 ou de la tour t2 ? Ce paramètre **self** permet de lever l'indécision : il est en quelque sorte l'*instance* (la variable) à laquelle on fait référence, celle que l'on veut manipuler. L'utilisation conjointe de **self** et de la *notation pointée* (qui doit son nom au point qui sépare deux mots, celui de droite étant « contenu » dans celui de gauche), permet de *qualifier* complètement une variable, c'est-à-dire en fait d'une zone mémoire. Par exemple, l'écriture **t2.hauteur_max** correspond sans ambiguïtée à la zone mémoire contenant la valeur 4, qui est la hauteur maximale attribuée à **t2**.

```
6   def remplir(self) :
7     self.disques = range(1, self.hauteur_max+1)
8     self.disques.reverse()
```

Nous définissons ici une méthode nommée **remplir()**, dont le rôle est de remplir la tour de tous les disques disponibles, jusqu'à sa hauteur maximale. Lorsque nous commencerons à réaliser effectivement le jeu, cette méthode sera utile pour créer la première tour. Pour utiliser cette méthode, on a également recours à la notation pointée, par exemple pour remplir la tour **t2** :

```
t2.remplir()
```

Aucun paramètre n'est donné à la fonction : implicitement, le pseudo-paramètre **self**, durant l'exécution de **remplir()**, devient en quelque sort un synonyme de **t2**. Donc, lors de l'exécution de **t2.remplir()**, l'écriture **self.disques** fait bien référence aux disques de **t2**. Cette petite gymnastique permet d'appliquer la même méthode à autant d'instance qu'on le souhaite : **t1.remplir()** va agir sur les disques de **t1** (car alors, **self** sera un synonyme de **t1**).

Concernant le contenu même de cette méthode, la liste disques est remplie en utilisant une fonction spéciale de Python, la fonction **range()**. Celle-ci retourne une liste d'entiers compris entre le premier paramètre et le dernier, celui-ci étant exclu. Par exemple, **range(2, 5)** retourne la liste **[2, 3, 4]**. Donc si nous avons une tour **t1** de trois disques (**t1.hauteur_max** vaut 3), **range(1, 4)** va créer la liste **[1, 2, 3]**, cette liste étant stockée dans **t1.disques**.

Ensuite nous devons décider dans quel ordre nous mémorisons les disques : le premier élément de la liste est-il le disque le plus bas, ou celui du sommet ? Par commodité, j'ai décidé que le premier élément de la liste est le disque le plus bas. Il nous faut donc « retourner » la liste donnée par **range()**, puisque le disque le plus bas est forcément le plus grand. On pourrait effectuer cette opération par une boucle (c'est d'ailleurs un bon exercice), mais par chance l'objet liste propose une méthode nommée **reverse()** qui fait justement cela. Hé oui, voilà une information que j'avais gardée secrète dans l'article précédent : les listes de Python sont en fait des objets ! Lorsque vous exécutez des instructions comme les suivantes, pour déclarer deux variables de type liste (la première vide, la deuxième contenant trois entiers) :

```
l1 = []
l2 = [1, 3, 5]
```

...en fait vous créez deux instances d'un type objet nommé **list**. Ce type propose des méthodes, comme **append()** que nous avons rencontré le mois dernier. Ici, nous allons utiliser **reverse()**, qui inverse l'ordre des éléments d'une liste. Si vous pensez que ce choix de l'ordre dans la liste n'est pas le plus pertinent, je vous invite à essayer de modifier le programme en décidant que le premier élément de la liste est le disque au sommet de la tour. Cela n'est pas particulièrement compliqué, il faut juste faire attention aux indices.

```
 9   def sommet(self) :
10     if ( len(self.disques) > 0 ) :
11       return self.disques[-1]
12     else :
13       return 0
```

La méthode précédente n'apporte rien en terme de fonctionnalité à l'objet, elle permet seulement de l'interroger : obtenir la taille du disque au sommet de la tour. Nous verrons par la suite qu'elle nous sera pourtant utile. En effet, donner la taille du disque au sommet d'une tour vide n'a pas de sens : c'est pourquoi il est nécessaire de tester la longueur de la liste, pour vérifier qu'elle n'est pas vide. Si elle n'est pas vide, on retourne la valeur du dernier élément (selon la convention d'ordre adoptée), sinon en renvoit zéro. Implicitement, cela permet également de savoir si la tour est vide ou non.

```
14   def disque(self, etage) :
15     if ( etage <= len(self.disques) ) :
16       return self.disques[etage-1]
17     else :
18       return 0
```

Encore une méthode d'interrogation : elle permet de connaître la taille du disque à un certain étage de la tour. Encore une fois, nous devons établir une convention. Le plus naturel est peut-être de décider que le disque le plus bas est au premier étage, celui au-dessus au deuxième étage, et ainsi de suite. Cette méthode suppose donc que

l'étage demandé se situe entre **1** et la longueur de la liste, inclus. Dans ce cas, on renvoit la valeur de l'élément de la liste à l'indice **etage-1** : en effet, rappelez-vous, les éléments d'une liste sont numérotés à partir de **0**, et non pas de **1**. Sinon, on renvoit **0**.

```
19   def nbDisques(self) :
20     return len(self.disques)
```

Simple petite méthode pour connaître le nombre de disques présents sur la tour. Vous pourriez objecter que cette méthode étant particulièrement triviale, elle n'a pas vraiment sa raison d'être... Ce n'est pas faux. Simplement, pensez à l'utilisation ultérieure de cette classe. Il est plus naturel d'utiliser une méthode dont le nom est « parlant », que de demander la longueur d'une liste. Par ailleurs, nul ne sachant de quoi l'avenir sera fait, peut-être sera-t-il nécessaire un jour de changer radicalement la manière dont sont stockés les disques. Peut-être alors faudra-t-il changer la manière dont est déterminé le nombre de disques. Le fait de fournir une méthode qui répond à cette question, permet de garantir que les autres parties du programme n'auront pas à être modifiées, même si le c#ur de **clTour** est modifié : les changements seront limités à quelques méthodes.

```
21   def transferer(self, vers_tour) :
22     if ( len(self.disques) > 0 ) :
23       if ( (self.sommet() < vers_tour.sommet()) or \
24            (vers_tour.nbDisques() == 0) ) :
25         vers_tour.disques.append(self.sommet())
26         del self.disques[-1]
27       else :
28         print "Impossible d'empiler un disque sur un plus petit"
29     else :
30       print "Tour vide, rien à transférer"
```

Voici probablement la méthode la plus importante : celle qui réalise le transfert d'un disque d'une tour vers une autre, en respectant les règles du jeu. La méthode attend en paramètre la tour vers laquelle on veut transférer un disque, dans le paramètre **vers_tour**. Comme vous pouvez le constater, elle contient plusieurs blocs imbriqués : examinons-les.

Tout d'abord, il convient de vérifier que l'on a quelque chose à transférer : si la tour de départ est vide, inutile d'aller plus loin. C'est le sens du premier test, ligne 22, que l'on pourrait traduire ainsi :

Descendons d'un cran, dans le bloc symbolisé par **[...]** dans la traduction précédente : nous sommes maintenant dans une situation où nous avons des disques. Le transfert ne peut s'effectuer que si l'une des deux conditions est vérifiée :

1. soit la tour cible (vers laquelle on transfert) contient des disques et le disque à son sommet est plus grand que le disque que l'on s'apprête à transférer ;
2. soit la tour cible ne contient aucun disque.

Nous avons donc une condition qui est composée de deux sous-conditions. La condition globale est vérifiée si au moins l'une des deux sous-conditions est vérifiée. Nous devons pour cela combiner les deux sous-conditions, pour n'en faire qu'une seule. Si nous appelons **c1** la première sous-condition, et **c2** la deuxième, la condition globale peut s'exprimer par :

L'alternative qui détermine si on peut transférer le disque ou non s'exprimerait donc par :

La petite conjonction « ou » se traduit en anglais par « or », et c'est justement ce mot que Python utilise pour

combiner deux conditions. Pour exprimer la première condition, il suffit de comparer les sommets des deux tours. Dans cette méthode, **self** représente la tour de départ : son sommet est obtenu par **self.sommet()**, en appelant une méthode que nous avons définie plus haut. De même, on obtient le disque en haut de la tour d'arrivée par **vers_tours.sommet()**. La condition **c1** peut alors s'exprimer par :

```
self.sommet() < vers_tours.sommet()
```

À l'intérieur de **sommet()**, lors du premier appel, **self** vaut... **self**, c'est-à-dire que nous sommes toujours dans le contexte de la tour de départ. Par contre, au deuxième appel, **self** devient synonyme de **vers_tours** : nous sommes dans le contexte de la tour d'arrivée. Au premier appel, nous utilisons la liste de disques de la tour de départ, au deuxième appel, nous utilisons la liste de disques de la tour d'arrivée.

Pour la deuxième sous-condition, il suffit de vérifier que la tour d'arrivée ne contient aucun disque. Comme nous avons défini une méthode donnant justement l'information du nombre de disques, cela s'exprime par :

```
vers_tours.nbDisques() == 0
```

Enfin, la combinaison des deux conditions donne :

```
(self.sommet() < vers_tours.sommet()) or (vers_tours.nbDisques() == 0)
```

Remarquez que chacune des sous-conditions est encadrée de parenthèses. Ce n'est pas strictement nécessaire, mais je vous recommande vivement de prendre cette habitude : votre code n'en sera que plus aisé à relire par la suite. Par ailleurs, dans l'extrait de code plus haut, vous pouvez voir que cette longue condition a été « étalée » sur deux lignes. D'une manière générale, je vous recommande également de placer chacune des sous-conditions sur une ligne, toujours dans un souci de lisibilité. Lorsque l'on programme en Python, et que l'on étale ainsi une unique instruction sur plusieurs lignes (la condition globale est en effet assimilée à une instruction unique), il est nécessaire d'ajouter une barre oblique inversée (*backslash*, ****) à la fin de chaque ligne, sauf la dernière. Vous pouvez voir ce caractère à la fin de la ligne 24 : il signale que la ligne suivante est en fait la continuation de cette ligne. Tout se passe comme si les deux lignes étaient accolées sur une seule, simplement c'est plus facile à lire pour l'#il humain.

Voyons maintenant le transfert en lui-même. Il faut *ajouter à la fin de la liste de disques de la tour d'arrivée, le disque au sommet de la tour de départ*. Essayez d'écrire vous-même l'instruction, en traduisant cette phrase en instructions Python :

- le disque au sommet de la tour de départ : **self.sommet()** ;
- ajouter un objet **x** à la fin d'une liste **l** : **l.append(x)** ;
- la liste de disques de la tour d'arrivée : **vers_tour.disques**.

Ce qui nous donne exactement :

```
vers_tour.disques.append(self.sommet())
```

Nous appelons la méthode **append()** du type liste, à partir de la variable disques faisant partie de la variable **vers_tours** de type **clTour**. Ouf, tout ça en une ligne !

Mais ceci ne réalise pas un transfert : strictement parlant, nous *dupliquons* le disque, pour en placer une *copie* à la

fin de la liste cible. Pour terminer le transfert, nous devons le retirer de la tour de départ. Pour supprimer un élément d'une liste, Python propose l'instruction **del**. Comme nous voulons supprimer le dernier élément de la liste de départ, il suffit d'exécuter :

```
    del self.disques[-1]
```

En réalité, **del** permet de supprimer bien d'autres choses, mais cela nous suffit pour l'instant.

Voilà, vous venez de créer votre premier type objet ! Il est maintenant temps de l'utiliser.

IV - Tester un peu le modèle

Nous allons faire cela à la fin du fichier **hanoi.py**, qui donc contient la définition de notre type. Ajoutez pour commencer ces trois lignes :

```
32 t1 = clTour(5)
33 t1.remplir()
34 print "t1 =", t1.disques
```

La première crée une tour dans une variable **t1**, pouvant accepter jusqu'à 5 disques. Remarquez que cela ressemble beaucoup à un appel de fonction... ce qui est assez proche de la réalité. En fait, cette ligne, en plus de créer une instance de **clTour**, va provoquer l'exécution d'une méthode particulière, nommée **__init__()** : la première que nous avons rencontrée dans la définition de notre classe **clTour**. En langage Python, toutes les classes (donc, tous les types objet) possèdent une méthode **__init__()** qui est exécutée dès qu'une instance est créée. Cette méthode permet d'initialiser l'instance de l'objet, notamment de donner des valeurs cohérentes aux données qu'il contient. Comme on le voit ici, on peut passer des paramètres à cette méthode : dans notre exemple, on donne la taille maximale de la tour.

Cette méthode spéciale est appelée le *constructeur* (parfois le *créateur*) de la classe : sont rôle est de construire l'instance de l'objet, de façon à ce qu'il soit dans un état cohérent pour être utilisé. Tous les langages objets proposent un mécanisme analogue.

La ligne suivante remplit la tour, en exécutant la méthode **remplir()** définie plus haut. Ensuite nous affichons le contenu de la liste de disques, ce qui doit donner :

```
t1 = [5, 4, 3, 2, 1]
```

Nous avons donc bien cinq disques, dont les tailles vont de 1 à 5, rangés du plus bas au plus haut. Créons une deuxième tour :

```
35 t2 = clTour(5)
36 print "t2 =", t2.disques
37 t2.transferer(t1)
```

La création est comme la précédente. L'affichage de la liste donne simplement :

```
t2 = []
```

En effet, nous n'avons pas rempli la tour, la liste est donc vide. Ligne suivante, nous tentons de transférer un disque de **t2** vers **t1**. Que va-t-il se passer ? Comme **t2** est vide, il n'y a rien à transférer, on obtient donc le message d'erreur que nous avions prévu dans la méthode **transfert()** :

```
Tour vide, rien à transférer
```

Poursuivons nos essais :

```
38 t1.transferer(t2)
39 print "t1 =", t1.disques, "t2 =", t2.disques
```

Cette fois, pas de message d'erreur. L'affichage des listes donne :

```
t1 = [5, 4, 3, 2] t2 = [1]
```

t2 a bien récupérer le disque au sommet de **t1**, et **t1** a bien « perdu » un disque. Que se passe-t-il si on recommence ?

```
40 t1.transferer(t2)
```

Cela revient à vouloir transférer le disque au sommet de **t1**, de taille 2, sur **t2**, dont le disque au sommet est de taille 1. Nous obtenons naturellement un message d'erreur :

```
Impossible d'empiler un disque sur un plus petit
```

Tout va bien, notre objet fonctionne !

V - Conclusion

Vous l'aurez compris, nous ne sommes qu'au début de notre réalisation ! Nous n'avons modélisé qu'une partie de notre jeu, mais j'espère que cela vous fait déjà entre apercevoir l'intérêt de la programmation par objet. Comme exercice, vous pouvez essayer de réaliser un programme équivalent, sans mettre en oeuvre le moindre objet. C'est parfaitement possible, naturellement, mais l'écriture vous paraîtra sans doute moins « naturelle ».

La prochaine fois, nous nous attaquerons à la modélisation effective du jeu, afin de permettre à un utilisateur d'essayer de résoudre le problème. L'objectif étant, d'ici quelque temps, d'obtenir un programme graphique manipulable à la souris.

VI - Téléchargement

Vous pouvez télécharger ce cours sous format **pdf**

Langue	Mise à jour	Pages	Taille	Mode FTP	Mode HTTP de secours
	2005.11.22	17	138 Ko	YB06.pdf	YB06.pdf

Vous pouvez également télécharger les sources de ce cours: hanoi.py

VII - Glossaire

classe

Type de donnée, au même titre qu'un entier ou une chaîne de caractères, défini par les moyens de la programmation orientée objet.

méthode

Fonction définie à l'intérieure d'une classe, agissant sur les données de celle-ci.

instance

Variable dont le type est une classe, comme **10** est une instance du type entier ou "**bonjour**" est une instance du type chaîne de caractères.

constructeur

Méthode spéciale dans une classe, exécutée lorsque l'on créé une instance de cette classe. En Python cette méthode est nommée **__init__()**.