Représentation des objets en mémoire

Le but de ce chapitre est de préciser le fonctionnement et l'organisation des différents types de mémoire en informatique. En cela, il s'agit d'un chapitre assez abstrait, qui la plupart du temps n'a pas de conséquence directe sur l'utilisateur d'un logiciel ou même sur le programmeur de logiciels.
Mais, dans certains cas, le fonctionnement de la mémoire en informatique peut avoir des conséquences directes sur la validité ou la nature des résultats obtenus. C'est notamment le cas pour la représentation en mémoire des nombres flottants et des listes, qui ont des répercussions profondes sur la programmation.
Pour cette raison, ce chapitre est d'une ***importance primordiale***.

I. La mémoire en informatique

I.1. Les différents types de mémoire

Définition 3.1 (Mémoire vive/mémoire morte)

La ***mémoire vive*** est la mémoire utilisée par le processeur (et accessible à travers le processeur par les utilisateurs et les programmeurs) pour traiter les processus en cours. Le contenu d'une mémoire vive peut être ***lu, supprimé et modifié*** rapidement.
La ***mémoire morte*** est une mémoire dont le contenu est fixé lors de sa construction et qui assure le fonctionnement minimal d'une machine numérique. Le contenu d'une mémoire morte peut être ***lu rapidement***, mais est ***difficile ou impossible à modifier et supprimer***.

Généralement, la mémoire morte a une très faible capacité. Elle contient par exemple le ***BIOS*** et d'une manière générale l'ensemble des routines de base nécessaires au fonctionnement d'un ordinateur.

Définition 3.2 (Mémoire rémanente/mémoire volatile)

On dit d'une mémoire informatique qu'elle est ***rémanente*** lorsqu'une coupure de l'alimentation électrique ***n'altère pas*** le contenu de cette mémoire. ***La mémoire morte est toujours de type rémanent***.
On dit d'une mémoire informatique qu'elle est ***volatile*** lorsqu'une coupure de l'alimentation électrique ***efface totalement*** le contenu de cette mémoire. ***La mémoire vive utilise l'ensemble de la mémoire volatile, mais aussi éventuellement une partie de la mémoire rémanente***.

I.2. Organisation générale de la mémoire

Définition 3.3 (Référence et valeur)

Une variable Python ne contient pas directement l'objet qu'on lui a affecté. Ce qu'elle contient véritablement c'est ***son adresse mémoire*** : on dit que la variable est une ***référence vers un objet*** ou qu'elle ***pointe vers cet objet***. L'objet en lui-même est appelé ***valeur*** de la variable.

Le code suivant par exemple effectue trois choses :

```
>>> ma_variable=-2
>>> a=ma_variable
>>> b=7
```

- Comme l'objet référencé par `ma_variable` existe déjà, Python se contente de faire pointer `a` vers le même objet.
- En revanche, 7 n'est pour l'instant pas encore contenu en mémoire. La commande `b=7` fait donc deux choses :
  - ⋆ elle crée une plage mémoire dans laquelle elle met la valeur 7 ;
  - ⋆ elle fait pointer `b` vers l'adresse de cette plage mémoire.

La figure 3.1 résume ce qui en résulte.

FIGURE 3.1 – Variables Python et valeurs en mémoire

```
>>> id(a)
140715874165040
>>> id(ma_variable)
140715874165040
>>> c=-2
>>> id(c)
140715874165040
>>> id(-2)
140715874165040
>>> id(b)
140715874165328
>>> id(7)
140715874165328
```

La fonction `id` permet d'obtenir l'adresse en mémoire d'un objet.
Le code ci-contre illustre la façon dont les variables et la mémoire sont gérées en Python.
Les variables `a,c,ma_variable` pointent toutes les trois vers la même plage mémoire où se trouve la valeur -2. `c` a pourtant été créé indépendamment des variables `a,ma_variable` : Python a repéré que la valeur -2 était déjà contenue dans une case mémoire et a fait pointer `c` vers son adresse.
`b` au contraire pointe vers une seconde plage mémoire où se trouve la valeur 7.

Ce fonctionnement de Python a de gros avantages en termes d'efficacité de la gestion mémoire. Mais il peut avoir dans certaines situations des inconvénients (voir chapitre 2).

I.3. Constituants élémentaires de la mémoires

Définition 3.4 (Bits/octets/bytes)

On appelle ***bit*** (abrégée b) le composant élémentaire d'une mémoire informatique : il s'agit d'une case mémoire ***ne pouvant contenir que deux valeurs***, 0 ou 1. L'utilisateur ou le programmeur n'ont pas accès aux bits individuellement.
On appelle ***octet*** un ensemble de ***8 bits*** contigus en mémoire. C'était à l'origine le regroupement minimal de bits accessibles par l'utilisateur ou le programmeur. C'est aujourd'hui une notion en grande partie obsolète.
On appelle ***byte*** (abrégée B) un ensemble de 2^k ***bits*** contigus constituant le regroupement minimal de bits accessibles par l'utilisateur ou le programmeur. C'est l'unité logique de mesure d'une mémoire.

Un ordinateur ***32 bits*** (ou un système d'exploitation ***32 bits***) est un ordinateur dans lequel $1B = 32b$ (***un byte vaut 32 bits***).
Un ordinateur ***64 bits*** est un ordinateur dans lequel $1B = 64b$ (***un byte vaut 64 bits***).
La mémoire d'une unité de stockage est généralement mesurée en ***kilobytes*** (kB), ***megabytes*** (MB), ***gigabytes*** (GB) ou ***terabytes*** (TB) pour ***les puissances de 10*** bytes. Les correspondances sont les suivantes :

- $1kB = 1000B$;
- $1MB = 10^6B = 1\,000\,000B$;
- $1GB = 10^9B = 1\,000\,000\,000B$;
- $1TB = 10^{12}B = 1\,000\,000\,000\,000B$.

Elle est mesurée en ***kibibytes*** (KiB), ***mebibytes*** (MiB), ***gibibytes*** (GiB) ou ***tebibytes*** (TiB) pour ***les puissances de 2*** bytes. Les correspondances sont les suivantes :

- $1KiB = 2^{10}B = 1024B$;
- $1MiB = 2^{20}B = 1\,048\,576B$;
- $1GiB = 2^{30}B = 1\,073\,741\,824B$;
- $1TiB = 2^{40}B = 1\,099\,511\,627\,776B$.

Les constructeurs informatiques jouent parfois avec l'ambiguïté de ces définitions.

II. Représentation des entiers et des flottants en informatique

II.1. Représentation des nombres entiers en mémoire

La mémoire est constituée de composants élémentaires ne pouvant prendre que deux valeurs 0 ou 1 et appelés ..... Pour représenter un objet en mémoire, il faut donc parvenir à l'écrire uniquement à l'aide de 0 ou de 1.
Python possède une fonction permettant d'obtenir cette représentation pour les nombres entiers (c'est-à-dire les objets de type ...) : c'est la fonction `bin` dont la syntaxe est `bin(NombreEntier)`. Cette représentation est appelée ***représentation binaire*** des entiers.
Voyons quelle est la représentation binaire de `a` :

```
>>> bin(a)
'-0b10'
```

Le `-` au début de la représentation binaire signifie que le nombre est négatif !
Le `0b` qui suit signifie qu'il s'agit d'une représentation binaire.
Enfin le reste est la représentation binaire elle-même : $\underline{10}_2$. Elle se traduit ainsi : $\underline{10}_2 = 1\times2^1+0\times2^0 = 2$.
En effet, les chiffres de la représentation binaire d'un nombre entier représentent les puissances de 2 successives en partant de $2^0 = 1$ pour le ***chiffre le plus à droite***, puis en augmentant l'exposant de 1 à chaque chiffre.
Ex. 3.1 Pour la variable `b`, un appel de `bin(b)` renvoie `'0b111'`. Justifier cette valeur de retour.

Cor. 3.1

Ex. 3.2 Pour une variable `x` inconnue, un appel de `bin(x)` renvoie `'0b10100011'`. Combien vaut `x` ?

Cor. 3.2

Ex. 3.3 Quel sera le résultat de la commande `bin(-21)` ?

Cor. 3.3

II.2. Obtention de la décomposition d'un entier en base 2

Comme nous venons de le voir, la décomposition d'un entier $n \in \mathbb{N}$ en base 2 conduit à l'écriture suivante

$$n = \underline{\epsilon_p \epsilon_{p-1} ... \epsilon_1 \epsilon_0}_2 = \sum_{i=0}^{p} \epsilon_i 2^i = \epsilon_0 + 2 \times \epsilon_1 + ... + 2^p \times \epsilon_p$$

Étant donné l'entier n, deux méthodes permettent d'obtenir les chiffres ϵ_i de son écriture en base 2 : l'une fournit les chiffres de la droite vers la gauche (de ϵ_0 jusqu'à ϵ_p), l'autre de la gauche vers la droite.

Méthode : Obtention de la décomposition en base 2 par divisions successives

On remarque que $n = \epsilon_0 + 2 \times (\epsilon_1 + ... + 2^{p-1}\epsilon_p)$: ϵ_0 est donc le reste de la division euclidienne de n par 2 et $\epsilon_1 + ... + 2^{p-1}\epsilon_p$ son quotient. On obtient donc ϵ_1 comme reste de la division euclidienne par 2 du quotient de la première division effectuée et ainsi de suite. Ceci conduit à l'algorithme suivant :

- Initialisation : on note n l'entier à décomposer.
- Propagation : on effectue la division euclidienne de n par 2, on retient le reste et on remplace n par le quotient obtenu. On recommence cette étape jusqu'à ce que la condition d'arrêt soit satisfaite.
- Condition d'arrêt : l'algorithme se termine lorsque le quotient obtenu dans la division euclidienne par 2 est nul.

Les restes obtenus lors des étapes de propagation de l'algorithme sont les chiffres de la décomposition en base 2 de n, donnés de la droite vers la gauche (de ϵ_0 jusqu'à ϵ_p).

Ex. 3.4 Donner la décomposition en base 2 de 59 et 832.

Cor. 3.4

Méthode : Obtention de la décomposition en base 2 par soustraction de la plus grande puissance de 2 possible

On remarque que $2^p \leqslant n = \sum_{i=0}^{p} \epsilon_i 2^i \leqslant \sum_{i=0}^{p} 2^i = 2^{p+1} - 1$: 2^p est donc la plus grande puissance de 2 inférieure ou égale à n. Ceci conduit à l'algorithme suivant :

- Initialisation : on note n l'entier à décomposer.
- Propagation : on obtient 2^p, la plus grande puissance de 2 inférieure ou égale à n. On la retient et on remplace n par $n - 2^p$. On recommence cette étape jusqu'à ce que la condition d'arrêt soit satisfaite.
- Condition d'arrêt : l'algorithme se termine lorsque la nouvelle valeur de n est nulle.

Les puissances obtenues lors des étapes de propagation de l'algorithme sont les ***chiffres 1*** de la décomposition en base 2 de n, donnés de la gauche vers la droite. Les puissances de 2 qui n'ont pas été rencontrées lors de la propagation correspondent à des ***chiffres 0*** dans la décomposition en base 2 de n.

Ex. 3.5 Donner la décomposition en base 2 de 65 et 1023.

Cor. 3.5

II.3. Opérations bit-à-bit

Il existe des opérateurs arithmétiques opérant directement sur la représentation binaire des opérandes :

Syntaxe : Opérateurs bit-à-bit

- ***Négation bit-à-bit*** : pour un entier `a`,
  `~a` renvoie l'entier dont la représentation binaire est obtenue en remplaçant
  ⋆ les 1 de `a` par des 0 ;
  ⋆ les 0 de `a` par des 1.
  Par exemple : pour un entier `n` représenté sur 8 bits,
  si `n`$= 7 = \underline{00000111}_2$ alors `~n`$= \underline{11111000}_2$.
  Calculer `~n+n+1` et en déduire la valeur (décimale) de `~n`.
  .....
  .....
- ***ET bit-à-bit*** : pour deux entiers `a` et `b`,
  `a&b` renvoie l'entier dont chaque chiffre binaire vaut
  ⋆ 1 si les chiffres de `a` et `b` à la même position sont des 1 ;
  ⋆ 0 sinon.
  Par exemple, pour `a`$= 23 = \underline{10111}_2$ et `b`$= 89 = \underline{1011001}_2$, `a&b`$= \underline{10001}_2 = ...$
- ***OU bit-à-bit*** : pour deux entiers `a` et `b`,
  `a|b` renvoie l'entier dont chaque chiffre binaire vaut
  ⋆ 1 si au moins l'un des chiffres de `a` et `b` à la même position vaut 1 ;
  ⋆ 0 les chiffres de `a` et `b` à la même position sont des 0.
  Par exemple, pour `a`$= 23 = \underline{10111}_2$ et `b`$= 89 = \underline{1011001}_2$, `a|b`$= \underline{1011111}_2$ = ...
- ***XOR bit-à-bit*** : pour deux entiers `a` et `b`,
  `a^b` renvoie l'entier dont chaque chiffre binaire vaut
  ⋆ 1 si un seul des chiffres de `a` et `b` à la même position vaut 1 ;
  ⋆ 0 les chiffres de `a` et `b` à la même position sont tous les deux des 0, ou tous les deux des 1.
  Par exemple, pour `a`$= 23 = \underline{10111}_2$ et `b`$= 89 = \underline{1011001}_2$, `a^b`= .....
- ***décalage à gauche*** : pour deux entiers `a` et `b`,
  `a<<b` renvoie l'entier dont les chiffres binaires sont ceux de `a` décalés de `b` chiffres vers la gauche.
  Par exemple, pour `a`$= 23 = \underline{10111}_2$ et `b`$= 3$, `a<<b`$= \underline{10111000}_2$ = ...
  `a<<b` donne donc le même résultat que `a*2**b`.
- ***décalage à droite*** : pour deux entiers `a` et `b`,
  `a>>b` renvoie l'entier dont les chiffres binaires sont ceux de `a` décalés de `b` chiffres vers la droite. Les chiffres « passant à droite de la virgule » sont perdus.
  Par exemple, pour `a`$= 23 = \underline{10111}_2$ et `b`$= 3$, `a>>b`$= \underline{10}_2$ = ...
  `a>>b` donne donc le même résultat que `a//2**b`.

II.4. Taille de la représentation des nombres entiers en mémoire

Python possède une fonction permettant d'obtenir la ***taille*** occupée par des objets en mémoire : c'est la fonction `getsizeof` dont la syntaxe est `getsizeof(objet)`.
Cependant, ce n'est pas une ***fonction standard*** (`builtin_function_or_method`), c'est-à-dire qu'elle n'est pas directement accessible au démarrage de Python, il faut d'abord ***l'importer***.

```
>>> from sys import getsizeof
>>> getsizeof(a)
```

```
28
>>> getsizeof(b)
28
```

Comme on le voit, la taille occupée par les objets référencés par `a` et `b` est la même : 28 octets.
Pour des entiers plus grands (en valeur absolue), le nombre d'octets occupés pourra être supérieur. Essayons par exemple d'obtenir la taille en mémoire de 2^{100} :

```
>>> c=2**100
>>> c
1267650600228229401496703205376
>>> type(c)
<class 'int'>
>>> getsizeof(c)
40
```

Remarque

À retenir.

- Python3 est capable d'effectuer des calculs sur des entiers de taille quelconque. ***Mais plus la taille occupée en mémoire est grande, plus les calculs sont lents*** !
- Pour importer la fonction `getsizeof` du ***module*** `sys`, on utilise la syntaxe
  `from sys import getsizeof`
  Même si nous utiliserons d'autres modules d'ici là, nous parlerons en détail de ce que sont les modules au chapitre 6.

II.5. Limitations de la représentation des objets en mémoire

Parce que la mémoire d'un ordinateur est finie tandis qu'on peut vouloir travailler avec des objets qui ne possèdent pas de représentation finie, il y a des limitations intrinsèques à la représentation des objets en mémoire.
Deux exemples :

- une photo numérique ne peut en aucun cas être une reproduction ***absolument fidèle*** de la scène vue par le photographe. En effet, la photo est stockée en mémoire sous la forme d'un tableau à deux dimensions de points de couleurs appelés ..... Toute l'information présente entre deux points consécutifs a été perdue. On s'aperçoit de cette perte lorsqu'on effectue zoom de forte magnitude sur la photo (voir figure 3.2) ;
- les nombres irrationnels (c'est-à-dire .....) ont un développement décimal ou binaire non seulement infini, mais encore non périodique. Il est donc impossible de les représenter précisément avec une mémoire finie. On ***effectue des approximations***. Voici un code l'illustrant bien :

  ```
  >>> from math import sin,pi
  >>> sin(2*pi)
  -2.4492935982947064e-16
  ```

II.6. Représentation des nombres flottants en mémoire

Nous venons de le voir, lorsqu'ils sont irrationnels ou parfois même rationnels, il est impossible de représenter de façon exacte les nombres réels. Python travaille donc avec des représentations approximatives des nombres réels appelées ***nombres à virgule flottante*** ou encore ***float***.

FIGURE 3.2 – Une photo et un zoom ($\sim 20\times$)

Cette représentation doit être binaire puisque l'ordinateur lui-même stocke les objets à l'aide de bits égaux à 0 ou à 1. De plus, il faut pouvoir représenter des nombres très petits mais aussi très grands et pouvoir faire des opérations sur ces nombres sans perdre trop de précision.
La solution choisie est l'analogue binaire de la représentation scientifique des nombre décimaux : tout nombre réel x va être représenté par trois entiers (S, M, E) tels que $x \approx (-1)^S \times M \times 2^{E-1075}$.

- S code le ***signe*** de x par un unique bit : 0 si $x \geqslant 0$, 1 si $x < 0$.
- M code les ***chiffres binaires significatifs*** sur 52 bits. Il s'agit d'un nombre entier compris entre ....
- E code un analogue de l'exposant de l'écriture scientifique, binaire, sur 11 bits. Il s'agit donc d'un nombre entier compris entre .... Suivant que E sera supérieur ou inférieur à 1075, $|x|$ sera respectivement .... à M.

Ex. 3.6 Quel est le plus grand float x possible ? Quel est le plus petit float y strictement positif possible ?

Cor. 3.6

II.7. Déclaration des `float` avec Python

Prenons l'exemple suivant :

```
>>> a=2
>>> b=2.
>>> a**1024
1797693134862315907729305190789024733617976978942306572734300811577326758055009631327084773
>>> b**1024
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: (34, 'Result too large')
```

Que s'est-il passé ? `a` et `b` ont la même valeur, mais Python calcule bien a^{1024} (un nombre de plus de 300 chiffres décimaux, qui ne tient pas sur la largeur de la page) tandis qu'il met un code d'erreur pour b^{1024} :
`Overflow : Result too large`
La raison en est que `a` est un entier (nous avons déjà vu que Python permet les calculs avec une précision quelconque sur les entiers) tandis que `b` est un nombre à virgule flottante et l'opération b^{1024} a engendré un ***dépassement de capacité mémoire*** ou ***overflow*** pour la représentation d'un `float`.

```
>>> type(2)
<class 'int'>
>>> type(2.)
<class 'float'>
```

Toute la différence se situe dans le `.` placé après le `2` : en mettant un point (l'équivalent de la virgule pour les anglo-saxons), on indique à Python que `b` est un `float`.

II.8. Approximations cachées

Encore un autre exemple :

```
>>> a=0.1
>>> a
0.1
```

On pourrait croire au premier abord que la valeur de `a` est exactement `0.1`. ***Or c'est faux !***
En effet, l'écriture binaire de `0.1` est ***infinie***. Python a donc été obligé de faire une approximation au moment de la création de l'objet `0.1`.
On peut s'en rendre compte à l'aide de la commande `format` qui permet de modifier le format d'affichage des objets :

```
>>> format(a,'.60f')
'0.1000000000000000055511151231257827021181583404541015625000000'
>>> format(10*a,'.60f')
'1.0000000000000000000000000000000000000000000000000000000000'
>>> format(pi,'.60f')
'3.1415926535897931159979634685441851615905761718750000000000000'
```

`'3.14159265358979323846264338327950288419716939937510582097494`4'` étant les véritables 60 premières décimales de π.
En fait, étant donnée la précision avec laquelle les `float` sont stockés en mémoire, on peut considérer que seules les 16 premières décimales sont significatives.

```
>>> format(a,'.16f')
'0.1000000000000000'
```