Cours d'informatique 2
Quelques notions de bas niveau

I Représentation des entiers

I.1 Le binaire

Du point de vue électronique, l'ordinateur ne sait manipuler qu'une suite de chiffres 0 ou 1, qu'on appelle des **bits**. Un seul bit peut donc coder uniquement deux valeurs différentes, deux bits permettent de former toutes les combinaisons 00, 01, 10, 11. Avec trois bits on peut arriver à huit combinaisons : ces quatre là précédées de 0, ou bien de 1.

000	001	010	011	100	101	110	111
0	1	2	3	4	5	6	7

Plus généralement (cf. cours sur la combinatoire) :

Proposition 1. *Avec n bits, on peut coder 2^n valeurs différentes.*

Les bits sont groupés par blocs de huit et forment un **octet**, qui permet de stocker $2^8 = 256$ valeurs. Les octets eux-mêmes sont souvent groupés par blocs de 2 (16 bits, avec $2^{16} = 65\,536$), par 4 (32 bits, avec $2^{32} = 4\,294\,967\,296$ soit environ quatre milliards) voire par 8 (64 bits, avec $2^{64} \approx 18 \cdot 10^{18}$).

I.2 Compter en binaire

En associant à chaque combinaison de n bits un nombre entier entre 0 et $2^n - 1$ on peut **compter en binaire**, comme ci-dessus de 0 à 7.
En fait, un nombre écrit en binaire comme une liste de chiffres $\underline{a_{n-1}a_{n-2}\dots a_1a_0}$ valant chacun 1 ou 0 correspond au nombre

$$N = a_{n-1} \times 2^{n-1} + a_{n-2} \times 2^{n-2} + \cdots + a_1 \times 2 + a_0$$

et on peut montrer que tout nombre entier naturel s'écrit de façon unique sous cette forme ! Quand on énumère proprement les nombres comme ci-dessus, on passe d'un nombre au suivant en ajoutant 1 (le bit le plus à droite alterne à chaque fois, le deuxième à droite alterne une fois sur deux, etc).

Remarque 1. Dans l'écriture binaire ci-dessus alors $n-1$ est le nombre tel que

$$2^{n-1} \leqslant N < 2^n$$

et donc

$$(n-1)\log(2) \leqslant \log(N) < n\log(2)$$

d'où en divisant par $\log(2)$ on trouve

$$n - 1 = \left\lfloor \frac{\log(N)}{\log(2)} \right\rfloor$$

Le nombre de bits nécessaires pour stocker N est donc $1 + \left\lfloor \frac{\log(N)}{\log(2)} \right\rfloor$.

I.3 Les entiers relatifs

Comment représenter des nombres négatifs en binaire ?
Une idée naturelle serait, dans un bloc de n bits, de réserver un bit qui indiquerait par exemple avec 0 que le nombre suivant est positif, ou avec 1 que le nombre suivant est négatif. On pourrait alors stocker les valeurs de -2^{n-1} à $+2^{n-1}$ avec les $n-1$ bits restants. Mais cela complique les calculs et nécessite des disjonctions de cas. Cela crée deux nombres zéros, l'un positif et l'autre négatif.
Une meilleure idée procède comme suit : partant de 0 on ajoute 1 à chaque fois pour énumérer les nombres positifs puis on arrive au plus grand nombre positif possible (ce sera $+2^{n-1} - 1$) puis en ajoutant encore 1 on saute directement au plus petit nombre négatif (qui sera -2^{n-1}). Sur 3 bits par exemple on obtient le codage suivant :

000	001	010	011	100	101	110	111
0	1	2	3	− 4	− 3	− 2	− 1

On remarque qu'en ajoutant 1 au dernier nombre on trouve $\underline{1000}$, mais puisqu'on raisonne sur 3 bits il faut tronquer ce premier 1 et c'est en fait le nombre $\underline{000}$ qui est l'écriture binaire de 0.
Cette méthode s'appelle **complément à deux**. On peut vérifier que l'opposé d'un nombre N est obtenu un renversant tous les bits (échanger les 0 et les 1) de N puis en ajoutant 1. Et que l'addition se fait « comme d'habitude », que le nombre représenté soit positif ou négatif : ci-dessus $(-3) + (-3)$ correspond à l'addition binaire $\underline{101} + \underline{101}$ qui, en tronquant à 3 bits, donne $\underline{010}$ et c'est le codage de 2… qui est bien la même chose que -6 ici, car sur 3 bits on raisonne modulo 8.

Proposition 2. *Sur n bits on peut coder en complément à deux tous les nombres de -2^{n-1} à $+2^{n-1} - 1$.*

I.4 Dans les langages de programmation

Dans les langages de programmation existant au début de l'informatique, et qui sont très proches du fonctionnement de l'ordinateur, avant d'utiliser une variable il faut déclarer sur combien d'octets elle sera enregistrée et si elle est sous forme sans signe (sur n bits, entre 0 et $2^n - 1$) ou avec signe. Les types se nomment `int32` (entier, avec signe, 32 bits), `uint32` (idem mais sans signe), `int64`, `uint64` etc. Ces types existent aussi dans les tableaux `numpy` : par exemple sur un tableau contenant des millions de nombres, le tableau sera quatre fois plus gros s'il utilise des entiers 32 bits alors que des entiers 8 bits non signés auraient suffi (si toutes les valeurs sont entre 0 et 255), et donc il peut être important de contrôler la taille des données.
Le langage Python permet de ne pas se préoccuper du tout de cela : les entiers de type `int` sont enregistrés avec signe, et leur taille en octets est *automatiquement étendue* au fur et à mesure des besoins ! Cela signifie que par derrière, le langage Python doit estimer quelle va être la bonne taille des nombres, et parfois les étendre.

II Représentation des nombres à virgule flottante

II.1 L'écriture scientifique

Les nombres à virgule flottante sont représentés par une écriture scientifique sous la forme

$$x = s \times m \times 2^e$$

où

1. s est le signe, $+$ ou $-$,
2. m est un nombre entier sans signe, appelé **mantisse**,
3. e est un nombre entier avec signe, appelé **exposant**.

Cela ne permet pas de représenter de façon exacte tous les nombres rationnels, mais seulement ceux dont le dénominateur est une puissance de 2 (soit l'exact analogue des nombres décimaux, mais en binaire).

II.2 Représentation interne

Sur les flottants les plus courants, dits *flottants à double précision au standard IEEE 754*, la place est de 64 bits (8 octets) utilisés comme suit :

1. 1 bit sert à donner le signe.
2. 11 bits servent à stocker l'exposant, avec signe. Cela permet d'avoir un exposant dans l'intervalle $\pm 2^{10}$, ce qui dans l'écriture scientifique décimale permet d'aller à $10^{\pm 308}$. C'est bien suffisant !
3. 52 bits servent à stocker la mantisse. Cela correspond à une précision d'environ 15 ou 16 chiffres significatifs décimaux.

Il n'y a pas unicité de l'écriture (par exemple $2 = 2 \times 2^0 = 1 \times 2^1$ a pour mantisse 2 et exposant 0, ou bien aussi mantisse 1 et exposant 1) : en fait il y a quelques contraintes, toutes les combinaisons possibles de bits ne vont pas représenter un nombre valide. Il y a ainsi la place de stocker quelques nombres spéciaux, résultant par exemple d'une division par zéro : `+inf`, `-inf` ainsi que `NaN` (Not a Number).

II.3 Dans les langages de programmation

Les nombres flottants Python, et dans de nombreux langages, sont bien les flottants double précision standards.
Il faut être conscient que les nombres décimaux ne peuvent pas toujours être représentés de façon exacte avec cette écriture scientifique, et qu'elle pose de nombreux problèmes d'arrondis. Le problème suivant est courant et connu :

```
>>> 0.1 + 0.2 == 0.3
False
```

En conséquence **ce n'est pas une bonne pratique de tester l'égalité de deux flottants**. Il faut se débrouiller autrement.
Par exemple : plutôt que de tester si les nombres rationnels $\frac{a}{b}$ et $\frac{c}{d}$ sont égaux, ne pas faire

```
if a/b == c/d:
    ...
```

qui calcule la division en nombres flottants, mais tester l'égalité de nombres entiers $a \times d = b \times c$:

```
if a*d == b*c:
    ...
```

De même, si on veut parcourir l'intervalle $[0, 1]$ avec des sauts de $\frac{1}{n}$, plutôt que

```
i = 0
while i <= 1:
    ...
    i = i + 1/n
```

il est préférable d'écrire

```
k = 0
while k <= n:
    i = k/n
    ...
    k = k + 1
```

En effet dans le premier cas on cumule des erreurs d'arrondis à chaque fois, alors que dans le second cas on est certain d'avoir bien n valeurs.
Pour créer un tableau de n valeurs bien espacées entre a et b, la fonction `numpy.linspace(a, b, n)` est tout à fait adaptée !
Enfin, on peut avoir besoin de manipuler des nombres flottants de précision arbitraire, dans lesquels l'exposant et la mantisse sont agrandis automatiquement au fur et à mesure des besoins. Pour cela on utilise des bibliothèques, qui ne font pas partie du langage Python de base, telles que `decimal` (fournie avec Python), `mpmath`, `gmpy2`. Chacune a ses propres spécificités techniques, car les questions de performance peuvent être cruciales !
On peut aussi manipuler des fractions sans passer par les virgules flottantes avec la librairie `fractions` : le numérateur et le dénominateurs sont tous les deux des nombres entiers de taille arbitraires, pouvant être agrandis automatiquement.

III La mémoire

III.1 Introduction

La mémoire de l'ordinateur, dans laquelle sont stockées les données en cours d'utilisation, peut être vue comme une **très** longue suite d'octets consécutifs. Tous sont numérotés à partir de 0, ainsi chaque octet dans la mémoire possède une **adresse**, un peu comme les maisons alignées dans une très longue rue...

Remarque 2. Avec 32 bits, on peut compter les adresses jusqu'à 2^{32} ce qui permet de travailler avec une mémoire d'environ 4 Go (un Giga-octet, Go, est égal à un milliard d'octets). Si cela était suffisant sans problèmes à la fin des années 90 et au début des années 2000, les mémoires actuelles sont plus grandes et c'est une bonne motivation pour travailler avant tout avec des entiers 64 bits !

Le langage Python permet de connaitre l'adresse mémoire d'une variable avec la commande `id()`, considérée comme une *identité unique* de la variable :

```
>>> x = 3
>>> id(x)
140169658804592
# en ce moment, sur mon ordinateur, x est à cette adresse
```

III.2 Les tableaux

Dans son fonctionnement basique, le gestionnaire de mémoire ne sait faire qu'une chose : réserver un bloc de mémoire, de taille fixée à l'avance, pour y stocker ou y lire des octets. Sachant par exemple que l'on a besoin d'une liste `L` (on parle en fait dans tous les cas de **tableau**) de n nombres entiers codés sur 4 octets, le bloc mémoire aura une taille de $4 \times n$ octets. Si le premier de ces nombres a une adresse N, alors le suivant est à l'adresse $N + 4$, et ainsi de suite les nombres sont aux adresses

$$N, \quad N+4, \quad N+8, \quad \ldots, \quad N+4(n-1)$$

et donc en notation Python `L[i]` est à l'adresse `N + 4*i`. Cela justifie d'ailleurs pleinement la convention de numérotation à partir de $i = 0$. Testez d'ailleurs les adresses d'éléments successifs d'une liste :

```
>>> L = [1, 2, 3, 4, 5]
>>> for i in range(5): print(id(L[i]))
140522552162608
140522552162640
140522552162672
140522552162704
140522552162736
```

Les éléments ici apparaissent régulièrement espacés de 32 octets ! C'est plus qu'il n'en faut pour coder un nombre entier (4 ou 8 octets) et il y a certainement une bonne raison pour cela...
Comment faire alors pour concaténer une liste `M` à la suite d'une autre `L` ? Et bien, s'il y a de la place à la fin du premier bloc de `L`, on peut y recopier `M` directement. Sinon, il faut réserver un bloc de mémoire plus gros pour y recopier `L` puis `M` les uns à la suite des autres.
Comment faire pour insérer un élément en plein milieu de la liste `L` ? Pour l'insérer à l'adresse N_0 il faut décaler tous les éléments qui sont après l'adresse N_0 vers la droite, puis insérer notre élément en N_0. Cela nécessitait d'avoir une case de libre au bout du bloc. Si non, il fallait tout recopier dans un bloc mémoire plus grand...
On voit qu'à bas niveau il n'est pas si simple de travailler avec les listes. Cependant le langage Python n'a pas toutes ces contraintes : les listes contiennent des objets qui n'occupent pas tous la même taille mémoire, et n'ont pas une longueur fixe, peuvent être concaténées, on peut ajouter ou supprimer des éléments en plein milieu. Ce qui se passe en fait, c'est que le langage Python **nous fait croire** que tout cela est facile, en nous fournissant plein de fonctions et d'opérations sur les listes qui les rendent très faciles à utiliser, mais **par derrière** il lui arrive de déplacer des blocs de mémoire ou de décaler tous les éléments, silencieusement, sans que l'utilisateur ne puisse le contrôler. Et cela prend parfois beaucoup de temps et de mémoire !

En comparaison, les tableaux `numpy` fonctionnent de façon la plus proche possible de ce qui est décrit là : taille fixée à l'avance, éléments tous de la même taille et du même type. Cela est plus difficile à apprendre à utiliser, mais rend leur manipulation par l'ordinateur bien plus rapide.

III.3 Les chaines de caractères

Chaque **caractère** individuel est codé par un nombre : il existe donc des tables donnant les valeurs en nombre de chacun des caractères. La plus connue et universelle est la table ASCII (American Standard Code for Information Interchange) qui donne un nombre aux caractères les plus courants qu'on trouve sur un clavier anglais-américain (lettres minuscules et majuscules, signes de ponctuation, chiffres, mais pas les accents), sur 7 bits seulement, soit 128 caractères. Par exemple, la lettre `a` est codée par le nombre 97 (et `b` par 98 etc) alors que `A` est codée par 65. On accède au code par la fonction Python `ord()` :

```
>>> ord("a")
97
>>> ord("?")
63
```

et son inverse est la fonction `chr()` :

```
>>> chr(246)
'ö'
# utile en allemand ! sehr schön !
```

La table ASCII contient aussi des caractères spéciaux comme les espaces et sauts de ligne.
Ensuite, une chaine de caractères est enregistrée dans la mémoire comme un tableau de caractères individuels consécutifs. Certains langages de programmation contiennent d'ailleurs un vrai type « caractère seul » et les chaines de caractères n'y sont vraiment rien de plus que des tableaux de caractères individuels.
En fait, un octet entier et ses 256 codes possibles n'est pas suffisant pour coder tous les caractères dont on peut avoir besoin, avec les nombreux accents et signes de ponctuation dans de nombreuses langues différentes. La solution qui est maintenant largement diffusée est le codage UTF-8 (Universal Character Set Transformation Format) où les caractères sont codés sur 1, 2, 3 ou 4 octets, cela est variable ! Mais avec plusieurs milliards de caractères ainsi possibles, cela permet d'écrire largement tous les alphabets utilisés sur Terre.
Le codage UTF-8 complique nettement le traitement des chaines de caractères, puisque si les caractères n'occupent pas tous le même nombre d'octets alors la longueur en nombre de caractères n'est plus égale à la longueur en nombre d'octets... Fort heureusement, le langage Python et les systèmes d'exploitation Linux gèrent les chaines en UTF-8 au plus profond de leur fonctionnement et depuis bien longtemps (et bien avant Windows...) et le programmeur n'a pas vraiment à s'en préoccuper.

IV Conclusion : à retenir

Il y a une opposition entre le fonctionnement **bas niveau** de l'ordinateur, et le langage Python qui fonctionne à **haut (ou très haut) niveau**. Il faut être conscient que certaines syntaxes Python sont très faciles à utiliser, mais font de nombreux calculs ou modifications de mémoire par derrière car elles sont éloignées du vrai fonctionnement de l'ordinateur. C'est le cas notamment des manipulations de listes.
C'est ce caractère haut niveau qui rend le langage Python particulièrement adapté à l'enseignement : on peut se concentrer directement sur la méthode de programmation et sur les algorithmes, alors que dans un langage comme C il faudrait déjà des dizaines de lignes rien que pour mettre en place des listes (calculer la place mémoire nécessaire, demander à réserver la place, vérifier si la place est bien libre...) Mais ceci rend en même temps Python fondamentalement plus lent, pour des programmes de calcul scientifique sur des grandes données où la vitesse de calcul et l'occupation mémoire sont des enjeux importants que l'on peut vouloir contrôler finement.
Cependant, un certain nombre de bibliothèques scientifiques (comme `numpy`) sont créées à bas niveau dans le but de fonctionner très efficacement, et sont utilisables en Python *via* les syntaxes habituelles et sympathiques (chargement avec `import`, aide interactive avec `help()`, affichage d'un seul coup avec `print()` etc). C'est cela qui fait le succès de Python dans les milieux scientifiques : n'importe quel scientifique peut en quelques lignes de code Python charger des données de grande taille depuis un fichier ou depuis le réseau et les envoyer à traiter par des bibliothèques de calcul optimisées ultra-rapides, rendre compte des résultats graphiquement, sauvegarder tout cela dans un fichier.
Pour être un bon programmeur Python, il faut donc être conscient des processus se passant par derrière, des syntaxes Python qui sont efficaces et de celles qui ont l'air sympathiques mais qui ne sont pas du tout efficaces. Et de même pour utiliser correctement `numpy` et en tirer le meilleur il faut savoir un minimum ce qu'il fait par derrière et pourquoi certaines pratiques sont meilleures que d'autres ! Cela nécessite un certain temps d'apprentissage à écrire et ré-écrire ses programmes du mieux possible.