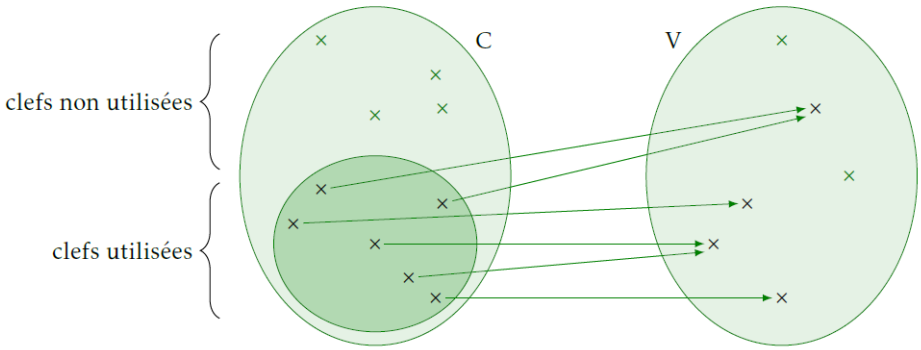
CHAPITRE 4

QUELQUES ÉLÉMENTS SUR L'ORGANISATION DE L'INFORMATION

I - Dictionnaires et fonctions de hachage

I.1 - Description

Un dictionnaire est un ensemble constitué de couples «clé : valeur». Il n'est pas ordonné. Les clés sont uniques et à chacune de ces clés est associée une valeur. Les clés doivent être des objets non mutables - en général on utilise des entiers, des chaînes de caractères ou des tuples, les valeurs peuvent être totalement quelconques.
La structure de données «dictionnaire» répond à la problématique suivante : comment rechercher efficacement de l'information dans un ensemble géré de façon dynamique (c'est à dire dont le contenu est susceptible d'évoluer au cours du temps).
Un dictionnaire (ou *table d'association*, ou *table de hachage*, *hash table*) est un type de données associant un ensemble de clefs à un ensemble de valeurs.
Plus formellement, si C désigne l'ensemble des clefs et V l'ensemble des valeurs, un dictionnaire est un sous-ensemble T de $C \times V$ tel que pour toute clef $c \in C$ il existe au plus un élément $v \in V$ tel que $(c, v) \in T$. Les éléments de T sont appelés des associations.



Un dictionnaire supporte en général les opérations suivantes :

- ajout d'une nouvelle association $(c, v) \in C \times V$ dans T ;
- suppression d'une association (c, v) de T ;
- recherche de l'existence d'une association (c, v) dans T pour une clef $c \in C$ donnée ;
- lecture de la valeur v associée à une clef c présente dans T.

En Python, la création d'un dictionnaire se réalise en suivant la syntaxe `{c1:v1,...,cn:vn}` où $c_1, \ldots, c_n$ sont des clefs (nécessairement deux-à-deux distinctes) et $v_1, \ldots, v_n$ les valeurs qui leur sont associées.
On obtient un dictionnaire vide avec `{}` ou `dict()`.

Si `D` est un dictionnaire et `c` une clef :

- `D[c]=v` ajoute une nouvelle association si la clef n'est pas présente dans le dictionnaire, et modifie l'association précédente sinon ;
- `del D[c]` supprime une association si la clef est présente dans le dictionnaire, et déclenche l'exception KeyError sinon ;
- l'expression `c in D` renvoie un booléen indiquant si la clef est présente ou non dans le dictionnaire ;
- `D[c]` renvoie la valeur associée à la clef si celle-ci est présente dans le dictionnaire, et déclenche l'exception KeyError sinon ;
- `D.keys()` est l'ensemble des clés du dictionnaire (objet itérable) ;
- `D.values()` est l'ensemble des valeurs du dictionnaire (objet itérable) ;
- `D.items()` est l'ensemble des couples clés/valeurs du dictionnaire (objet itérable) ;
- `len(D)` est la taille du dictionnaire (nombre de clés).

Les clefs d'un même dictionnaire doivent impérativement appartenir à un type immuable.

I.2 - Mise en œuvre d'un dictionnaire

Comparaison listes et dictionnaires : Les points à retenir (en simplifiant un peu) et en notant n la taille de la liste/dictionnaire :

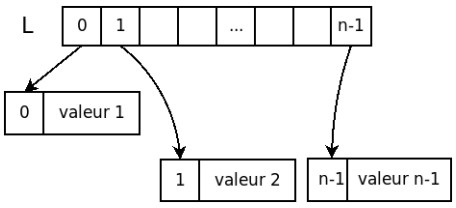
- une liste est intéressante lorsque l'ordre des éléments est important, ou que l'indexation par des entiers est importante - à l'opposé, une dictionnaire permet d'avoir un ensemble de clés totalement quelconque,
- le test d'appartenance est en O(1) pour un dictionnaire, alors qu'il peut être en O(n) pour une liste
- les temps d'accès à un élément donné est en O(1) pour les deux (un peu plus rapide pour les listes),
- le temps de suppression d'un élément est en O(1) pour un dictionnaire et en O(n) pour une liste (sauf si c'est le dernier élément qu'on retire avec pop())

Pour une comparaison plus précise des complexités, on pourra consulter :

`https ://wiki.python.org/moin/TimeComplexity`

I.3 - Principe

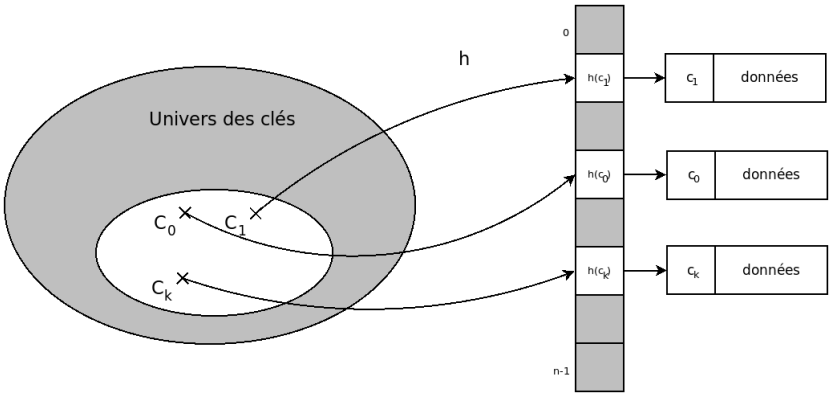
Si les clefs étaient des entiers compris entre 0 et $m-1$, le problème serait simple : il suffirait d'utiliser une liste dynamique Python de taille m. Elles sont gérées par adressage direct. L'univers des clés est un ensemble d'entiers de 0 à $m-1$ et on a accés directement à la case k, puis à sa donnée correspondante :



On pourrait alors avoir une structure analogue mais «avec des trous», lorsqu'on a un ensemble fini de clés possibles mais qu'elles ne sont pas toutes utilisées.

Cela nécessiterait toutefois de réserver un tableau d'adresse très grand (par exemple avec des clés qui sont des chaînes de 5 caractères, on réserverait 26^5, soit presque 12 millions de cases) pour n'en utiliser qu'une petite partie.

L'idée est donc de réduire l'ensemble des clés par une **fonction de hachage** , c'est-à-dire une fonction $f : \mathrm{C} \to [\![0, m-1]\!]$ associant aux différentes clefs $c \in \mathrm{C}$ utilisées un entier dans $[\![0, m-1]\!]$.



Cependant, le nombre de clefs possibles étant très important, le cardinal de C est beaucoup plus grand que m et une telle fonction ne sera pas injective.
Il existera donc des couples (c_1, c_2) dans C tels que $c_1 \neq c_2$ et $f(c_1) = f(c_2)$. Un tel couple (c_1, c_2) est appelé une **collision**, et il faut trouver une solution pour gérer ces collisions lorsqu'elles se produisent.

I.4 - Fonction de hachage

Pour définir la fonction f, on utilise une fonction h, appelée fonction de hachage, associant un entier à une clef de C, et on définit f en posant :

$$f(c) = h(c) \bmod m$$

Pour conduire à une implémentation efficiente, une fonction de hachage doit :

- être facile à calculer ;
- avoir une distribution la plus uniforme possible.

Cette dernière condition est motivée par le souhait de minimiser le nombre de collisions. Pour que cette fonction assure une bonne répartition des clefs dans les différents emplacements du tableau, il faut, de manière informelle, qu'étant donné un entier $i \in [\![0, m-1]\!]$, la probabilité que $h(c) \bmod m$ soit égal à i soit de l'ordre de $1/m$.

Dans ces conditions, si k désigne le nombre de clefs distinctes de T, la probabilité pour qu'il y ait au moins une collision est égale à :

$$1 - \frac{m!}{m^k (m-k)!}$$

Par exemple, pour $m = 1000000$, la probabilité pour qu'il y ait collision dépasse $0,5$ lorsque le nombre de clefs dépasse 1200 ; pour $k = 2500$ la probabilité qu'il y ait collision dépasse $0,95$. Les collisions sont donc rapidement inévitables.

Choix de la fonction de hachage
C'est bien évidemment un problème très complexe (que nous n'aborderons pas). Sachez seulement que Python propose une fonction de hachage : si x est un objet immuable (`int, str, float`, ..) alors `hash(x)` renvoie un entier répondant à peu près aux exigences d'une fonction de hachage :

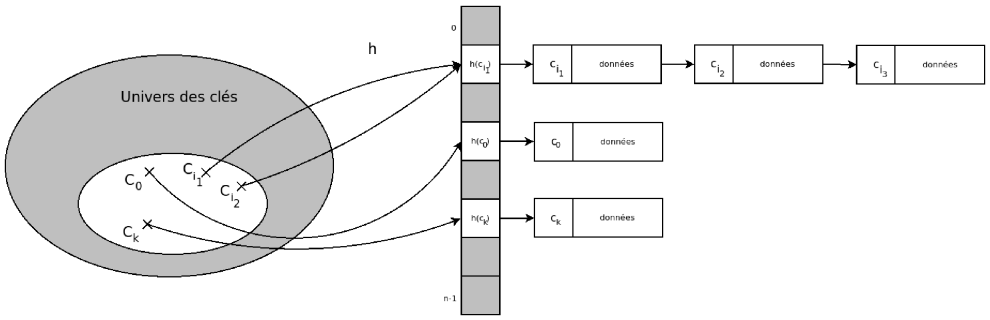
```
>>> hash(12345)
12345
>>> hash("12345")
-5394429019530837313
>>> hash ((1,2,3,4,5)) # un tuple est immuable
-5659871693760987716
>>> hash ([1,2,3,4,5]) # une liste est mutable
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Remarque
Une fonction de hachage a d'autres usages. Par exemple, lorsque l'on choisit un mot de passe sur un site sécurisé, ce dernier ne va pas stocker le mot de passe en clair mais le résultat de l'application d'une fonction de hachage h à ce mot de passe. Sachant qu'il est très difficile de trouver les antécédents d'un entier par la fonction h, pirater le site ne mettra pas en cause la sécurité du mot de passe.

I.5 - Résolution des collisions

I.5.a - Résolution des collisions par chaînage

Une solution possible pour résoudre les collisions consiste à stocker les valeurs dont les clefs sont rentrées en collision dans un même "paquet".



Pour deux clefs c_1 et c_2 entrant en collision, on stocke donc les associations (c_1, v_1) et (c_2, v_2) dans un même paquet (une liste par exemple).
Pour trouver la valeur associée à une clef c présente dans le dictionnaire, on procède alors en deux temps :

- on calcule $f(c) = h(c) \bmod m$ pour déterminer l'emplacement du tableau où se trouve le paquet contenant l'association recherchée ;
- on procède à une recherche naïve dans le paquet pour trouver cette association.

Cette méthode présente l'avantage de pouvoir contenir un nombre k de clefs plus grand que le nombre m de cases du tableau.
Si on note $\alpha = \frac{k}{m}$ le taux de remplissage du dictionnaire, les paquets auront une taille moyenne de α (sous une hypothèse de hachage uniforme) et la recherche d'une association dans le dictionnaire utilisera un nombre de comparaison en moyenne de l'ordre de α.
Si on considère que le calcul de $f(c)$ se réalise en temps constant et que chaque paquet est de petite taille, les quatre opérations se réalisent effectivement avec une complexité temporelle constante.

I.5.b - Adressage ouvert

Autre solution : en cas de collision, on cherche un emplacement libre dans lequel déposer la nouvelle association. La recherche d'un emplacement libre porte le nom de *sondage* (On dit que l'on *sonde* la table pour trouver une case libre).

- le sondage *linéaire* consiste à partir de $i = h(c)$ et à chercher une place libre en testant successivement les cases d'indices $(i+1) \bmod m, (i+2) \bmod m, (i+3) \bmod m, \ldots$ jusqu'à trouver un emplacement libre.
- le sondage *quadratique* procède de même mais en sondant les cases d'indices $(i+1) \bmod m, (i+1+2) \bmod m$, $(i+1+2+3) \bmod m, \ldots$

L'inconvénient d'un sondage linéaire est qu'il y a un risque de former des «agrégats», autrement dit de longues successions de cases contigües occupées, qui nuisent à la répartition uniforme recherchée.

Remarque

D'autres solutions plus complexes existent, comme par exemple sonder les cases $i + h'(c) \bmod m, i + 2h'(c) \bmod m, i + 3h'(c) \bmod m, \ldots$ où h' est une autre fonction de hachage, mais aucune ne résout complètement le problème de la création d'agrégats.

Évidemment, un adressage ouvert exige que le nombre k de clefs soit inférieur à la taille de la table m. En outre, on imagine aisément que lorsque le rapport $\alpha = \frac{k}{m}$ se rapproche de 1 , il devient de plus en plus difficile de trouver un emplacement vide : si $\alpha = 0,5$ un ajout nécessite en moyenne deux sondages, contre dix lorsque $\alpha = 0,9$. Aussi, lorsque α devient trop grand il est nécessaire de créer une table plus grande (la taille est en général doublée) pour préserver les performances.

II - Représentation des nombres

II.1 - Introduction

Tout dans un ordinateur est stocké sous la forme de 0 ou de 1 qui correspondent à un état électrique haut ou bas en un point du circuit (*i.e.* à une tension nulle ou de $+n$ Volts). Ce 0 ou 1 qui est la plus petite unité d'information gérable par la machine, constitue un ***bit*** (de *binary digit*). Afin d'accélérer le traitement, ces bits sont regroupés par 8, formant ainsi un ***octet*** auquel on attribue un numéro (son adresse mémoire) permettant de le repérer dans la mémoire.
Un ordinateur calcule donc en base 2, tout nombre étant représenté par une suite de 0 ou de 1. Par exemple :

$$5 = 1.2^2 + 0.2^1 + 1.2^0 = \underline{101}_2$$

$$255 = 2^7 + 2^6 + \cdots + 2^0 = \underline{11111111}_2$$

$$65535 = 2^{15} + 2^{14} + 2^{13} + \cdots + 2^0 = \underline{1111111111111111}_2$$

Un octet peut représenter un nombre entre 0 et $255 = 2^8$. Deux octets mis bout à bout (donc 16 bits) représentent un nombre entier compris entre 0 et 65535. Certains registres contiennent 2 octets (on parle de registre 16 bits), d'autres 8 (registres 64 bits). Une machine disposant de registres 64 bits est plus rapide qu'une machine 16 bits car elle dispose de circuits capable de traiter directement des opérations sur 8 octets à la fois au lieu de décomposer ces opérations par blocs de 2 octets.

Nom	Symbole	Nombre d'octets
kilooctet	ko	10^3
mégaoctet	Mo	10^6
gigaoctet	Go	10^9
téraoctet	To	10^{12}
pétaoctet	Po	10^{15}

TABLE 4.1 – Multiples de l'octet : préfixes SI

Nom	Symbole	Nombre d'octets
kibioctet	Kio	2^{10}
mébioctet	Mio	2^{20}
gibioctet	Gio	2^{30}
tébioctet	Tio	2^{40}
pébioctet	Pio	2^{50}

TABLE 4.2 – Multiples de l'octet : préfixes binaires

II.2 - Représentation des entiers

II.2.a - Bases de numération

Si b est un entier avec $b \geqslant 2$ alors tout entier naturel N s'écrit de façon unique sous la forme :

$$\mathrm{N} = a_{n-1}b^{n-1} + \cdots + a_1 b^1 + a_0 = \sum_{k=0}^{n-1} a_k b^k$$

où les a_i sont dans l'intervalle $[\![0, b-1]\!]$.
Par exemple, l'écriture en base 2 d'un entier naturel N est son écriture sous la forme :

$$\mathrm{N} = a_{n-1}2^{n-1} + \cdots + a_1 2 + a_0 = \sum_{k=0}^{n-1} a_k 2^k$$

où les a_i valent 0 ou 1. On peut noter : $\mathrm{N} = \underline{a_{n-1}\cdots a_1 a_0}_2$.

Exemples

1 ► Supposons que N s'écrive $\mathrm{N} = \underline{a_{n-1}\cdots a_1 a_0}_2$ en base 2.
Comment s'écrit 2N ? 4N ? Que peut-on dire du quotient et du reste de la division de N par 2 ?

2 ► Représentons les entiers sur (des mots de) 8 bits. Par exemple, l'entier N = 101 s'écrit sur 8 bits :

$$0\ 1\ 1\ 0\ 0\ 1\ 0\ 1.$$

Quels entiers sont ainsi représentés ?

Quel est le résultat de l'addition 203 + 121 ?

II.2.b - Les entiers relatifs

La représentation des *entiers signés* sur n bits repose sur le principe suivant :

▷ Si $N \geqslant 0$ alors le bit de gauche est nul et les autres bits servent à coder N sur $n-1$ bits.

▷ Si $N < 0$ alors le bit de gauche vaut 1 et les autres bits codent $2^{n-1} + N$ sur $n-1$ bits.

Exemples

1 ► Quels sont les codages sur 8 bits de de 13 ? de -1 ? de -3 ?

2 ► Quels sont les entiers représentables sur 8 bits ?

3 ► Est-ce que la somme de deux entiers signés représentés sur 8 bits est toujours «cohérente» ?

Remarques

1 ► On différencie facilement les entiers positifs des négatifs par le bit de gauche : 0 pour les $N \geqslant 0$ et 1 pour les $N < 0$.

2 ► En fait, pour coder un entier négatif N, la méthode revient à coder $-N$ puis à inverser tous les bits et à ajouter 1 (on dit que c'est la *méthode du complément à 2*). Par exemple, pour coder $N = -103$ sur 8 bits, on commence par coder 103 :

$$103 = \underline{1100111}_2 \text{ est codé par } 0|1100111$$

puis on inverse les bits et on ajoute 1 :

$$1|0011000 \quad \text{puis} \quad 1|0011001.$$

3 ► Connaissant la représentation de N, on obtient facilement celle de $-N$ en inversant tous les bits de N puis en ajoutant 1.

4 ► Python est un langage informatique sophistiqué (langage «de haut niveau») qui manipule par défaut des entiers de taille non spécifiée (*entiers multi-précision*), en adaptant le mode de stockage de façon dynamique aux besoins.

II.3 - Réels, décimaux, dyadiques et flottants

II.3.a - Principe de la représentation en virgule flottante

Un *nombre dyadique* est le produit d'un entier par une puissance de 2 et s'écrit sous la forme d'un «nombre à virgule» :

$$x = \pm a_p a_{p-1} \dots a_0, a_{-1} a_{-2} \dots a_{-q}$$

où les a_i valent 0 ou 1 ; cela signifie :

$$x = \pm \sum_{k=-q}^{p} a_k 2^k = \pm \left(a_p 2^p + \dots + a_1 2^1 + a_0 2^0 + a_{-1} \frac{1}{2^1} + \dots + a_{-q} \frac{1}{2^q} \right).$$

Par exemple $5,75 = 4 + 1 + 0,5 + 0,25$ admet pour développement dyadique $(101,11)_2$.

On appelle ***représentation en virgule flottante*** d'un nombre réel x, toute écriture de la forme :

$$x = s\ m\ b^e$$

où $s \in \{-1, +1\}$ est le signe, b est la base choisie (on ne considère que 2 et 10), e l'*exposant* (un entier relatif) et m un réel appelé la *mantisse*. Tout nombre n'admet pas une telle écriture (mais c'est le cas des décimaux en base 10 et des dyadiques en base 2) et elle n'est pas non plus unique ; par exemple :

$$1,35.10^0 = 135.10^{-2} = 0,135.10^1.$$

Pour palier ce problème de non unicité, on décide (pour $x \neq 0$) de normaliser l'écriture en imposant $m \in [1, b[$ (par exemple $6,02.10^{23}$).
On décide de représenter les réels avec une telle écriture (mais nécessairement un nombre fini de bits). La norme IEEE-754 fixe les convention pour le représentation sur 32 ou 64 bits. Sur 32 bits, on a (tous les b_i valent désormais 0 ou 1) :

$$x = \pm 1, b_1 \cdots b_{23}\ 2^e$$

et on code ce nombre avec :

- le premier bit donne le signe (0 pour positif et 1 pour négatif) ;
- les 8 suivants codent $e + 127$ (mais on ne considère que $e \in [\![-126, 127]\!]$ car les valeurs -127 et 128 sont réservées pour coder les infinis par exemple) ;
- les 23 suivants sont les 23 chiffres après la virgule.

Exemple

1 00110111 10110100000000000000000 correspond à $-1,101101\ 2^{55-127}$.

Sur 64 bits, on code $e + 1023$ sur 11 bits (et $-1022 \leqslant e \leqslant 1023$) et on code 52 chiffres après la virgule. Le cas où tous les bits sauf éventuellement le premier sont nuls correspond à 0^- et 0^+ (selon le premier bit).

Remarques

1 ► Il y a beaucoup d'exceptions, en voici à titre indicatif quelques unes sur 64 bits :

- ▷ exposant 0 (ou -1023 après décalage)
  Comme la mantisse est censée commencer par un 1 implicite (non écrit dans les 52 bits), 0 ne peut être représenté ainsi, d'où
  $$x = 0 \Leftrightarrow \forall (i, j) \in [\![1, 52]\!] \times [\![1, 11]\!],\ m_i = e_j = 0$$
  c'est-à-dire que tous les bits de la mantisse et de l'exposant sont nuls et il reste le signe : il y a deux zéros, un positif et l'autre négatif...
- ▷ exposant 2047 (ou 1024 après décalage)

$$+\infty : 0\ \overbrace{11\dots1}^{\text{11 bits = 1}}\ \overbrace{00\dots0}^{\text{mantisse 52 bits =0}}$$
$$-\infty : 1\ \overbrace{11\dots1}^{\text{11 bits = 1}}\ \overbrace{00\dots0}^{\text{mantisse 52 bits =0}}$$
$$\text{NaN (Not a Number)} : s\ \overbrace{11\dots1}^{\text{11 bits = 1}}\ \overbrace{\dots1\dots}^{\text{mantisse non nulle (au moins un 1)}} \quad \text{avec } s = 0 \text{ ou } 1$$

Les `NaN` dénotent en général une erreur de calcul : division par 0, racine carrée d'un nombre négatif,...

2 ► ***Les dépassements de capacité***

La plupart des langages informatiques stockent les nombres sur un nombre N fixé de bits ce qui engendre certains problèmes de calcul.

Sur les entiers naturels ou relatifs, le produit de 2 nombres écrits sur 8 bits (par exemple) peut facilement dépasser les 8 bits : on a un dépassement arithmétique (*overflow*) :

- soit un message d'erreur ;
- soit une poursuite du calcul en tronquant les bits qui dépassent...(produit négatif sur $\mathbb{Z}$ de deux nombres $\geqslant 0$...)

Avec Python, la seule limite pour les entiers est la mémoire disponible (Python découpe les entiers par paquets et les stocke dans des tableaux).

Sur les flottants, les dépassements arithmétiques (calcul sur les trop grands nombres) donnent :

- $+\infty$ et $-\infty$, dans les opérations qui suivent ceux-ci respectent les règles usuelles de calcul sur les limites ;
- `NaN` quand il y a ambiguïté.

Sur les flottants, pour les petits nombres proches de 0, un dépassement par valeurs inférieures (*soupassement arithmétique* ou *underflow*) peut donner 0 ou une erreur.

3 ► ***Arrondis***

Tous les réels ne sont pas représentables de manière exacte (même dans un intervalle fixé). La norme IEEE 754 fixe 4 modes d'arrondis :

- $-\infty$
- $+\infty$
- 0^+ ou 0^-
- par le nombre représentable le plus proche, s'il y a égalité de distance avec celui juste inférieur et celui juste supérieur, on choisit celui dont la mantisse se termine par 0.

III - Des exemples de sujets

Extrait de Centrale 2020

I Pixels et images

I.A Pixels

Un pixel (contraction de l'anglais picture element) est un élément de couleur homogène utilisé pour représenter une image sous forme numérique. La teinte d'un pixel peut être représentée de plusieurs façons. Une méthode courante, basée sur la synthèse additive, consiste à la décomposer en trois composantes qui correspondent aux couleurs rouge, vert et bleu. On parle de représentation RGB (pour red, green et blue). Chacune des trois composantes donne l'intensité de la couleur correspondante dans la teinte finale, 0 indiquant l'absence de cette couleur. Ainsi, le triplet $(0,0,0)$ désigne un pixel noir.

Q 1. On suppose que chacune des trois composantes RGB d'un pixel est représentée par un nombre entier positif ou nul, codé sur 8 bits. Combien de couleurs différentes peut-on représenter avec un tel pixel ?

Dans la suite, on représente un pixel par un vecteur (tableau numpy à une dimension) d'entiers de type `np.uint8` (entier non signé codé sur 8 bits) à trois éléments, correspondant respectivement à chacune des composantes RGB du pixel ; on utilise dans toute la suite le type pixel pour désigner un tel vecteur.

Q 2. Donner une instruction permettant de créer un vecteur correspondant à un pixel blanc.

Il est rappelé qu'en Python, comme dans beaucoup de langages de programmation, les opérations d'addition, soustraction, multiplication, division entière, modulo et élévation à la puissance (opérateurs $+, -, *, //, \%, **$) appliquées à deux opérandes de même type fournissent un résultat du type de leurs opérandes. Cela peut conduire à un dépassement de capacité et à une erreur de calcul car, les dépassements de capacité étant par défaut « silencieux », ils ne produisent pas d'erreur lors de l'exécution du programme.

L'opérateur division (/) entre deux entiers produit toujours un résultat sous forme de nombre à virgule flottante même si la division est exacte ($12/2 \rightarrow 6.0$). Il en est de même pour toute fonction faisant implicitement appel à cet opérateur comme `np.mean`.

Q 3. On pose `a=np.uint8(280)` et `b=npuint8(240)`. Que valent `a`, `b`, `a+b`, `a-b`, `a//b` et `a/b` ?

Les fonctions numpy qui effectuent de manière répétitive des opérations élémentaires, si elles ne garantissent pas l'absence de dépassement de capacité, prennent la précaution d'utiliser pour leurs calculs intermédiaires et leur résultat un type compatible avec le type de base de la plus grande capacité possible. Par exemple le résultat de `np.sum (np.array([100, 200], np.uint8))` est de type `np.uint64` (entier non signé codé sur 64 bits) et vaut bien 300 .

Pour représenter une image en niveau de gris, on peut se contenter d'une valeur par pixel, représentant l'intensité du gris entre le noir et le blanc. Pour convertir une image en couleurs en niveaux de gris, on peut remplacer chaque pixel par un seul entier, dont la valeur correspond à la meilleure approximation entière de la moyenne des trois composantes RGB du pixel.

Q 4. Écrire une fonction d'entête `def gris(p :pixel) -> np.uint8 :` qui calcule le niveau de gris correspondant au pixel `p`.

Extrait de Mines 2019 (autour des nombres premiers)

1. Dans un programme Python on souhaite pouvoir faire appel aux fonctions log, sqrt, floor, ceil du module math (round est disponible par défaut). Écrire des instructions permettant d'avoir accès à ces fonctions et d'afficher le logarithme népérien de 0.5.

2. On donne le code suivant :

```
pas = 1e-5

x2 = 0
for i in range(100000):
    x1 = (i + 1) * pas
    x2 = x2 + pas

print("x1:", x1)
print("x2:", x2)
```

L'exécution de ce code produit le résultat :

```
x1: 1.0
x2: 0.9999999999980838
```

Commenter.

3. Le crible d'Eratosthène est un algorithme qui permet de déterminer la liste des nombres premiers compris entre 1 et n. A la fin de l'algorithme, pour une valeur N on renvoie une liste de N booléens liste_bool. Si un élément de liste_bool vaut Vrai alors le nombre codé par l'indice considéré est premier. Par exemple pour N = 4 on renvoie [False, True, True, False].

Sachant que le langage Python traite les booléns comme une liste d'éléments de 32 bits, quel est (approximativement) la valeur maximale de N pour laquelle liste_bool est stockable dans une mémoire vive de 4 Go ?

4. Quel facteur peut-on gagner sur la valeur maximale de N en utilisant une bibliothèque permettant de coder les booléens non pas sur 32 bits mais dans le plus petit espace mémoire possible pour ce type de données (on demande de le préciser) ?

Extrait Centrale 2019 (Elasticité d'un brin d'ADN)

1. Soit ϕ une fonction de classe $\mathscr{C}^2$ sur $\mathbb{R}$ présentant un minimum local.

On rappelle que :

$$\frac{\phi(x(1+h)) - \phi(x(1-h))}{2xh} \tag{III.2}$$

est une expression approchée d'ordre 2 de la dérivée de ϕ en x (notée $\phi'(x)$).

On suppose que l'ordinateur utilisé représente les nombres flottants sur 64 bits avec un bit de signe, 11 bits d'exposant et 52 bits de mantisse.

Calculer le nombre de chiffres significatifs décimaux donnés par ce codage.

2. Justifier que les valeurs `h` = 1 et `h` = 10^{-16} ne permettent pas obtenir une bonne approximation du nombre dérivé ϕ'(x). Proposer alors une valeur adaptée de `h`.

Extrait Centrale 2016 (Système d'alerte de trafic et d'évitement de collision)

La règlementation actuelle impose aux avions de ligne d'être équipé d'un système embarqué d'évitement de collision en vol, ou TCAS pour *traffic collision avoidance system*. Nous nous intéressons au fonctionnement du TCAS vu d'un avion particulier que nous appelons *avion propre*. Les avions qui volent à proximité de l'avion propre sont qualifiés d'*intrus*.

III.A Acquisition et stockage des données

Chaque avion est équipé d'un émetteur radio spécialisé, appelé *transpondeur*, qui fournit automatiquement, en réponse à l'interrogation d'une station au sol ou d'un autre avion, des informations sur l'avion dans lequel il est installé. La fonction `acquerir_intrus` utilise les données du système de navigation de l'avion propre, les données fournies par le transpondeur de l'intrus, le relèvement de son émission et les informations fournies par le système de contrôle aérien au sol.

III.A.1) Les transpondeurs utilisent tous la même fréquence radio. Afin d'éviter la saturation de cette fréquence, en particulier dans les zones à fort trafic, chaque émission ne doit pas durer plus de 128 µ s. Le débit binaire utilisé est de 10^6 bits par seconde ; chaque message commence par une marque de début de 6 bits et se termine par 4 bits de contrôle et une marque de fin de 6 bits.

$$\underbrace{d\,d\,d\,d\,d\,d}_{\text{début}}\underbrace{x\,x\,x\ldots x\,x\,x}_{\text{données}}\underbrace{e\,e\,e\,e}_{\text{contrôle}}\underbrace{f\,f\,f\,f\,f\,f}_{\text{fin}}$$

Déterminer le nombre maximum de bits de données dans une émission de transpondeur.

III.A.2) Le système TCAS souhaite récupérer l'altitude et la vitesse ascensionnelle de chaque intrus en interrogeant son transpondeur. La réponse du transpondeur contient systématiquement un numéro d'identification de l'avion sur 24 bits. Les autres informations sont des entiers codés en binaire qui peuvent varier dans les intervalles suivants :

- altitude de 2 000 à 66 000 pieds ;
- vitesse ascensionnelle de −5 000 à 5 000 pieds par minute.

La taille d'un message de transpondeur est-elle suffisante pour obtenir ces informations en une seule fois ?

III.A.3) Après avoir récupéré les informations nécessaires, la fonction `acquerir_intrus` calcule la position et la vitesse de l'intrus par rapport à l'avion propre et renvoie une liste de huit nombres :

```
[id, x, y, z, vx, vy, vz, t0]
```

où

- `id` est le numéro d'identification de l'intrus ;
- `x`, `y`, `z` les coordonnées (en mètres) de l'intrus dans un repère orthonormé $\mathscr{R}_0$ lié à l'avion propre ;
- `vx`, `vy`, `vz` la vitesse (en mètres par seconde) de l'intrus dans ce même repère ;
- `t0` le moment de la mesure (en secondes depuis un instant de référence).

À des fins d'analyse une fois l'avion revenu au sol, la fonction acquerir_intrus conserve chaque résultat obtenu. Chaque nombre est stocké sur 4 octets. En supposant que cette fonction est appelé au maximum 100 fois par seconde, quel est le volume de mémoire nécessaire pour conserver les données de 100 heures de fonctionnement du TCAS ?

Ce volume de stockage représente-t-il une contrainte technique forte ? Nous nous intéressons au fonctionnement du TCAS vu d'un avion particulier que nous appelons *avion propre*. Les avions qui volent à proximité de l'avion propre sont qualifiés d'*intrus*.