Numérique et Science Informatique Première

Alain Busser

19 février 2026

Chapitre 1

Expressions et Instructions

1.1 Expressions

1.1.1 Types

Des expressions (constantes ou variables) ont une *valeur* qui a elle-même un *type*. Les types de base au programme sont :

- les booléens (`False` et `True`)
- les entiers
- les flottants
- les chaînes de caractères.

1.1.2 Variables

Une variable est formée :

- d'un nom (une chaîne de caractères sans guillemets)
- d'une valeur (dont le type est le type de la variable).

Une variable est une expression, obtenue en remplaçant son nom par sa valeur.

1.1.3 Expressions complexes

Si E est une expression et f une fonction alors f(E) est une expression. La somme de deux expressions est une expression de même type. Par exemple si `x` vaut 3 alors les expressions suivantes ont même valeur :

```
x+2
3+2
5
25//5
20//4
8-x
```

1.1.4 Expressions booléennes

`2+2==4` est une expression. Sa valeur est `True` : elle est booléenne. L'expression booléenne la plus fréquente est `x > 0` comme dans `while x>0:` mais il y a aussi des expressions plus complexes comme `2 <= x <= 5` ou `x>10 or x==2`.
Les mots-clés `if`, `elif` et `while` sont obligatoirement suivis d'expressions booléennes.

1.2 Instructions

1.2.1 Définition

Une instruction est un ordre donné à la machine.
Une expression a vocation à être *évaluée*, alors qu'une instruction a vocation à être *exécutée*. Une fois une instruction exécutée, l'environnement est en général modifié.

1.2.2 Exemples

`return E` où E est une expression, est une instruction.
`def f():` où f est une chaîne de caractères (le nom de la fonction) est une instruction.
`print(E)` où E est une expression, est une instruction.
`assert E`, où E est une expression booléenne, est une instruction.
`pass` est une instruction.
Dans le module `turtle`, les appels aux fonctions de déplacement (`forward(n)`, `backward(n)`, `left(n)`, `right(n)` etc) sont des instructions.

1.2.3 Variables

L'instruction la plus fréquente est l'affectation de variables. Elle s'écrit `nom = valeur` où `nom` est le nom de la variable (une chaîne de caractères sans guillemets) et `valeur` est une expression (la nouvelle valeur de la variable).

1.3 Programmes

1.3.1 Définition

Un programme (ou script) est une suite d'instructions.

1.3.2 Fonction

Une fonction est formée de trois parties :

- l'initialisation de certaines variables (les *arguments* ou *paramètres* de la fonction)
- un programme
- l'instruction `return` suivie d'une expression à renvoyer.

1.4 POO

Python manipule des *objets*. Chaque objet possède des *attributs* (variables propres à l'objet) dont certains sont des *méthodes* (des fonctions propres à l'objet). La fonction `dir` renvoie les méthodes d'un objet.

1.4.1 mutabilité

Un objet est *mutable* si son contenu peut être modifié en place (exemples : listes et dictionnaires), *immuable* sinon. Un objet immuable possède une méthode `__hash__`.

1.4.2 suscriptabilité

Un objet possédant une méthode `__getitem__` est dit *suscriptable*. On peut utiliser la notation des crochets avec un tel objet (liste, tuple ou dictionnaire).

1.4.3 appelabilité

Un objet possédant une méthode `__call__` est dit *appelable* (exemple : les fonctions sont appelables). Pour appeler un objet appelable, on utilise la notation des parenthèses.

Chapitre 2

Algorithmique

Dans ce chapitre, on considère un algorithme dont la donnée en entrée est de taille n.

2.1 Coût

Le *coût en espace* est une fonction donnant la quantité de mémoire utilisée par l'algorithme, en fonction de n. Le *coût en temps* est une fonction donnant la durée du calcul par l'algorithme, en fonction de n. Quand on parle de coût, on sous-entend *coût en temps dans le pire cas*.

2.1.1 Coût constant

Lorsqu'un coût est majoré par une constante (exemple accès à un attribut d'un objet comme `type`), on dit que l'algorithme est de coût constant.

2.1.2 Coût linéaire

Si un coût est majoré par une fonction affine, il est aussi majoré (à partir d'une valeur de n) par une fonction linéaire. Par exemple la fonction affine $n \mapsto n+3$ est majorée par la fonction linéaire $n \mapsto 2 \times n$ dès que $n \geqslant 3$:

n	0	1	2	3	4	5	6	7	8
$n+3$	3	4	5	6	7	8	9	10	11
$2 \times n$	0	2	4	6	8	10	12	14	16

Généralement, on néglige les termes constants quand on évalue un coût.

2.1.3 Coût quadratique

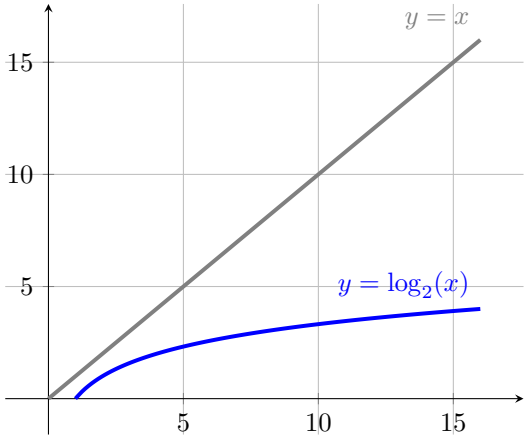
Un coût est dit *quadratique* s'il est majoré par une fonction du type $C \times n^2$ où C est une constante positive.

2.1.4 Coût cubique

Un coût est dit *cubique* s'il est majoré par une fonction du type $C \times n^3$ où C est une constante positive.

2.1.5 Coût logarithmique

Un *logarithme* est une fonction qui transforme les multiplications en additions. Par exemple comme $255^2 = 65025$ il suffit de 2 octets pour stocker le produit de deux nombres inférieurs à 256. Plus généralement il faut $a + b$ bits pour multiplier un nombre de a bits par un nombre de b bits. En notant $\log_2(x) + 1$ le nombre de chiffres binaires de l'écriture de x, la fonction $\log_2$ est un logarithme.



Comme $\log(x) < x$, un coût en $n \times \log(n)$ est meilleur qu'un coût quadratique.

2.1.6 Coût exponentiel

Un coût exponentiel est de la forme C^n où C est une constante supérieure à 1.
Dans l'ordre de performances décroissantes on trouve :

1. le coût constant,
2. le coût logarithmique,
3. le coût linéaire,
4. le coût en $n \times \log(n)$,
5. le coût quadratique,
6. le coût cubique,
7. le coût exponentiel.

2.2 Variants et invariants

2.2.1 Variants

Quand on ne peut pas évaluer le coût, on utilise un *variant* pour prouver la terminaison du calcul : un variant est une expression à valeurs entières qui décroît strictement à chaque passage dans une boucle. Par exemple dans

```
while x<8:
```

l'expression `8-x` est un variant, et dans

```
for i in range(8):
```

l'expression `8-i` est un variant.

2.2.2 Invariants

Un *invariant* est une expression booléenne qui ne peut pas passer de `True` à `False`. Par exemple dans

```
while x<8:
```

l'expression `x<8` est un invariant de boucle : elle ne devient fausse que lorsqu'on a quitté la boucle. En Python on utilise l'instruction `assert` pour implémenter un invariant. Les invariants sont utilisés pour prouver l'absence de bugs.

Chapitre 3

Les booléens

3.1 Type booléen

3.1.1 En Python

Il n'existe que deux booléens : `False` et `True`. Leur type est `bool`.

3.1.2 Origine

George Boole 1854
Aristote environ -340

3.1.3 Conversion

`int(False)` et `int(True)` rappellent que Boole représentait les booléens par les nombres 0 et 1.

3.2 Opérations booléennes

3.2.1 négation

`not False` et `not True` montrent que la négation est la fonction $x \longmapsto 1 - x$.
La négation de `==` est `!=`.
La négation de `>` est `<=`.
La négation est matérialisée par un relais inverseur.

3.2.2 Conjonction

Table

a	b	a and b
0	0	0
0	1	0
1	0	0
1	1	1

La conjonction `and` est une multiplication de nombres à un bit.
On matérialise une conjonction en branchant deux relais en série.

3.2.3 Disjonction

Table

a	b	a or b
0	0	0
0	1	1
1	0	1
1	1	1

La disjonction inclusive `or` n'est pas une addition : chez Boole `1+1 == 1`.
On matérialise une disjonction en branchant deux relais en parallèle.

3.2.4 Disjonction exclusive

table

a	b	not a	not b	(not a and b)	(a and not b)	(not a and b) or (a and not b)
0	0	1	1	0	0	0
0	1	1	0	1	0	1
1	0	0	1	0	1	1
1	1	0	0	0	0	0

```
for a in [False,True]:
    for b in [False,True]:
        print(a,b,(not a and b) or (a and not b))
```

Le ou exclusif est une addition sans retenue (Claude Shannon 1935).

3.3 Utilisation en Python

3.3.1 Tests

Le mot `if` est suivi d'une expression booléenne puis d'un double-point.
Si `EB` est une expression booléenne,

```
if EB:
    traitement1()
else:
    traitement2()
```

fait la même chose que

```
if not EB:
    traitement2()
else:
    traitement1()
```

3.3.2 boucles

`while` est suivi d'une expression booléenne variable puis d'un double-point.

```
i = 0
while i<8:
    instructions()
    i += 1
```

fait la même chose que

```
for i in range(0,8):
    instructions()
```

3.3.3 Évaluation séquentielle

En Python, `a and b` et `b and a` ne sont pas pareils, surtout lorsque `a` ou `b` n'est pas booléen.

```
if x!=0 and y==1/x:
```

permet par exemple d'éviter les divisions par 0.

- `False and b` s'évalue à `False`
- `True and b` s'évalue à `b`
- `False or b` s'évalue à `b`
- `True or b` s'évalue à `True`

Chapitre 4

Les tableaux

4.1 Construction des tableaux

4.1.1 En compréhension

```
carres = [i**2 for i in range(6)]
```

4.2 Utilisation

4.2.1 Accès à un élément

```
seize = carre[4]
```

```
carre[2] = 5
```

On peut modifier `carre[2]` : une liste est *mutable*. Pour créer une copie d'une liste il ne faut pas faire `nouvelle_liste = liste` mais `nouvelle_liste = liste.copy()` ou une compréhension.

4.2.2 Parcours

```
sc = 0
for i in range(len(carres)):
    sc += carres[i]
```

a le même effet que

```
sc = 0
for c in carres:
    sc += c
```

4.3 Algorithmes

4.3.1 Maximum

```
def maximum(tab:list)->int:
    assert len(tab)>0
    m = tab[0]
    for x in tab:
        if x>m:
            m = x
    return m
```

Cet algorithme est de coût *linéaire* : dans le pire cas il faut parcourir tout le tableau.

Chapitre 5

Numération

5.1 Binaire

5.1.1 Représentation

```
def binaire(foo):
    assert foo >= 0
    assert type(foo) == int
    if foo == 0:
        return [0]
    tab = []
    while foo > 0:
        tab.insert(0,foo%2)
        foo //= 2
    return tab

def decimal(bar):
    for elt in bar:
        assert elt in [0,1]
    accu = 0
    poids = 1
    bar.reverse()
    for bit in bar:
        accu += bit*poids
        poids *= 2
    return accu
```

5.1.2 Addition

(Leibniz 1703)

```
  110
+ 111
-----
 1101

   101
+ 1011
------
 10000
```

5.1.3 Soustraction

```
  1101
-  111
------
   110
```

```
  10000
-  1011
-------
    101
```

5.1.4 Multiplication

```
    11
x   11
------
    11
   11
------
  1001
```

```
   101
x   11
------
   101
  101
------
  1111
```

```
   101
x  101
------
   101
 101
------
 11001
```

5.1.5 Division

```
 1111 | 11
      |-----
-11   | 101
 --   |
   1  |
   11 |
  -11 |
   -- |
    0 |
```

5.2 Hexadécimal

décimal	binaire	hexadécimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	a
11	1011	b
12	1100	c
13	1101	d
14	1110	e
15	1111	f

Chapitre 6

Langage machine

6.1 Architecture

6.1.1 Circuits calculateurs

Shannon 1935, Von Neumann 1945
Avec des relais (ou tubes à vide ou transistors) on peut réaliser

- des portes `and` (retenue de l'addition)
- des portes `xor` (addition sans retenue)

ce qui permet de réaliser un *additionneur 1 bit*.
Une cascade d'additionneurs 1 bit donne un additionneur 8 bits.
Pour multiplier par 2 on décale les bits vers la gauche, en ajoutant un 0 à la fin. Exemple, le double de `1101` est `11010`.
Pour diviser par 2 on décale les bits vers la droite en écrasant le dernier bit. Exemple la moitié de `1101` est `110`.

6.1.2 ALU

Une unité arithmétique et logique est un circuit comprenant un additionneur, des registres à décalage etc, et un registre de contrôle permettant d'aiguiller les données vers un de ces circuits.

6.1.3 CPU

Un *microprocessur* est un circuit comprenant une ALU, un registre d'instructions permettant de configurer l'ALU, des registres de données et un *compteur de programme* contenant l'adresse de la prochaine instruction.
Un CPU se programme en *assembleur*. Pour le Z80 par exemple

- additionner le contenu du registre B au registre A s'écrit `ADD A, B` (`10000000`)
- doubler le contenu du registre B s'écrit `SLA B` (`00100000`)
- diviser par 2 le contenu du registre A s'écrit `SRA A` (`00110101`)

6.2 Mémoire

6.2.1 RAM

La `Random Access Memory` est formée de registres, elle s'efface lorsqu'on éteint l'ordinateur.

6.2.2 ROM

La `Read Only Memory` est ineffaçable. La mémoire flash aussi.

Chapitre 7

Représentation des nombres

7.1 Représentation des entiers relatifs

Le plus petit entier naturel représentable sur un octet est 0 (`00000000` ou `00`) et le plus grand est 255 (`11111111` ou `ff`). Pour représenter des entiers relatifs il faut ajouter un *bit de signe*.
Les entiers naturels représentables sur 7 bits vont de 0 à 127 (`01111111` ou`7f`). En ajoutant un bit de signe on va de -127 à 127. Mais 0 est alors codé de deux manières :

- +0 (`00000000`)
- -0 (`10000000`

Les entiers positifs sont représentés avec un bit de signe égal à 0, de 0 (`00000000`) à 127 (`01111111`) mais les entiers négatifs sont représentés en *complément à 2*.
Pour représenter -13 sur un octet on fait :

- représentation de 13 sur 7 bits : `0001101`
- inversion des bits : `1110010`
- addition de 1 : `1110011`
- bit de signe à 1 : `11110011`

Cette représentation permet d'additionner les entiers relatifs avec un additionneur d'octets.
Pour effectuer 20-13 = 20+(-13) on fait

```
  00010100
+ 11110011
----------
 100000111
```

En omettant la retenue on obtient `111` soit 20-13=7.

7.2 Représentation des flottants

7.2.1 codage binaire

Sur 32 bits on a :

- 1 bit de signe (1 pour les nombres négatifs, 0 pour les nombres positifs ou nuls)
- 8 bits d'exposant (puissance de 2, ajouté à 127)
- 23 bits de mantisse.

7.2.2 Flottants

Sur 32 bits, le quotient de 1 par 10 est `0 01111011 00110011001100110011001` qui représente le flottant 0,099999994039535522460937 5 : 0,1 n'est pas représentable exactement en machine.

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

En Python les flottants ont le type `float`.
éviter de comparer des flottants

Chapitre 8

Algorithmes de tri

8.1 Tri par sélection

```
def tri_selection(tab):
    t = [c for c in tab]
    lt = []
    while len(t)>0:
        carte = min(t)
        t.remove(carte)
        lt.append(carte)
    return lt
```

8.2 Tri par insertion

```
def tri_insertion(tab):
    t = [c for c in tab]
    lt = []
    while len(t)>0:
        carte = t.pop()
        i = 0
        while i<len(lt) and lt[i]<carte:
            i += 1
        lt.insert(i,carte)
    return lt
```

8.3 Coût

`len(t)` est un variant.
lt est triée est un invariant.
Pour ces deux algorithmes, le coût dans le pire cas est *quadratique* : le nombre de comparaisons dans le pire cas est $\frac{n\times(n+1)}{2}$ (`https://oeis.org/A000217`).

Chapitre 9

Représentation d'un texte en machine

9.1 Ascii

9.1.1 American Standard Code for Information Interchange

Dans le code Ascii chacune des lettres de l'alphabet latin est représentée par un mot de 7 bits.
Il y a 128 mots de 7 bits dont 26 lettres majuscules, 26 lettres minuscules, 10 chiffres et 32 signes de ponctuation et autres (opérations mathématiques etc). Le reste est des caractères spéciaux.
Par exemple la lettre `P` est codée en Ascii par le mot `01010000`. Or le nombre 80 est aussi codé par `01010000`. On dit que le code Ascii de la lettre `P` est 80.

```
>>> ord('P')
80
>>> chr(80)
'P'
```

9.2 Unicode

Unicode permet de coder les caractères du monde entier. On utilise *utf-8* qui découpe le code en octets. Il y a de 1 à 4 octets donnant des points code.

- 1 octet commençant par 0 code les 128 caractères Ascii.
- 2 octets commençant respectivement par `110` et `10` codent 2048 caractères.
- 3 octets commençant respectivement par `1110`, `10` et `10` codent 65536 caractères.
- 4 octets commençant respectivement par `1110`, `10`, `10` et `10` codent 4 194 304 caractères.

9.3 Les chaînes de caractères en Python

9.3.1 Le type str

En Python, les chaînes de caractères ont le type `str`. On peut additionner des chaînes de caractères, lire la lettre à l'indice 2 avec `mot[2]` et la longueur d'un mot est `len(mot)`. Les chaînes de caractères sont immuables.

9.3.2 Unicode

- L'expression `chr(n)` renvoie le caractère de code `n`.
- L'expression `ord(c)` renvoie le code du caractère (c'est-à-dire la chaîne de caractères de longueur 1) `c`.

Chapitre 10

Les tuples

10.1 Construction

10.1.1 Notation

```
couple = (4,16)
```

10.1.2 Noms

- couple (longueur 2)
- triplet (longueur 3)
- quadruplet
- quintuplet
- ...
- n-uplet (en anglais t-uple abrégé en `tuple`)

10.2 Propriétés

10.2.1 Accès

Un tuple possède une méthode `__getitem__()` donc le premier élément d'un tuple `t` sera `t[0]`.

10.2.2 Modification

Un tuple ne possède *pas* de méthode `__setitem__(elt)` donc on ne peut *pas* modifier le premier élément d'un tuple : un tuple est *immuable*.

10.2.3 Fonctions

`return (a,b)` peut s'écrire `return a,b`

Chapitre 11

Réseaux

11.1 Adresses IPv4

Une adresse IP est formée de 4 octets, écrits en notation décimale, séparés par des points.
Exemple `243.1.2.5`
Il y a donc 2^{32} adresses IP possibles.

11.2 Masquage

11.2.1 Principe

Un routeur permet d'envoyer des paquets vers plusieurs machines d'un réseau local, par exemple avec les adresses IP `243.1.2.1`, `243.1.2.2`, `243.1.2.3`, `243.1.2.4`, `243.1.2.5` etc.
Toutes ces machines ont la même adresse réseau qui est `243.1.2.0`.
Pour obtenir l'adresse du réseau auquel appartient une machine, on effectue une opération `and` (multiplication) bit à bit, entre l'adresse IP de la machine, et une adresse IP appelée *masque de réseau*. Par exemple un `and` entre `243.1.2.5` et `255.255.255.0` donne `243.1.2.0`.
La notation CIDR (*Classless Inter-Domain Routing* fait suivre l'adresse IP, du nombre de bits à 1 dans le masque de réseau. Par exemple `243.1.2.5/24`.

11.2.2 Adresse de diffusion

Si le routeur envoie un fichier à l'adresse `243.1.2.2` la machine d'adresse `243.1.2.5` ne le reçoit pas.
Si le routeur envoie un fichier à l'adresse `243.1.2.5` seule cette machine le reçoit.
Si le routeur envoie un fichier à l'adresse `243.1.2.255` la machine d'adresse `243.1.2.5` le reçoit : l'adresse `243.1.2.255` permet d'envoyer un fichier à toutes les machines du réseau d'adresse `243.1.2.0`.
L'adresse `243.1.2.255` est l'*adresse de diffusion* (`broadcast`) du réseau local.

11.3 Transmission Control Protocol

Le protocole TCP découpe un fichier en paquets. Les paquets sont envoyés l'un après l'autre et chaque fois qu'un paquet arrive, un paquet est envoyé en retour pour accuser réception.
Le *protocole du bit alterné* consiste à envoyer

- un paquet pair
- un paquet impair
- un paquet pair
- un paquet impair
- etc.

et chaque fois qu'arrive un paquet pair, le récepteur confirme qu'il a reçu un paquet pair, à la réception d'un paquet impair il confirme qu'il a reçu un paquet impair.

Chapitre 12

Recherche dichotomique

12.1 Algorithme

```
def contient(tab,elt):
    b1 = 0
    b2 = len(tab)-1
    while b2-b1 >= 0:
        m = (b1+b2)//2
        if tab[m] == elt:
            return True
        elif tab[m] > elt:
            b2 = m-1
        else:
            b1 = m+1
    return False
```

12.2 Analyse

12.2.1 Terminaison

`b2-b1` est un variant, il prouve que l'algorithme termine en un temps fini.

12.2.2 Correction

Un invariant est *si l'élément est dans le tableau, il est à gauche de `m` s'il est inférieur à `tab[m]` et à droite de `m` sinon.* Il prouve que la fonction renvoie `True` si l'élément est dans la liste triée, `False` sinon.

12.2.3 Coût

L'algorithme de recherche dichotomique est de coût *logarithmique* : il passe au maximum $log_2(n)$ fois dans la boucle, pour une liste de n éléments.

```
def log2(n):
    compteur = 0
    while n>0:
        compteur += 1
        n //= 2
    return compteur
```

Chapitre 13

Fichiers

13.1 Commandes de base

13.1.1 Fichiers

Pour créer un fichier :

```
touch script.py
```

Pour créer une copie du fichier :

```
cp script.py script.bak
```

Pour renommer un fichier :

```
mv script.py exo.py
```

Pour supprimer un fichier :

```
rm script.py
```

13.1.2 Dossiers

Pour créer un dossier :

```
mkdir documents
```

Pour aller dans un sous-dossier :

```
cd documents
```

Pour aller dans le dossier parent :

```
cd ..
```

Pour supprimer un dossier vide :

```
rmdir documents
```

13.2 Droits d'accès

Les droits d'accès d'un fichier sont codés sur 10 bits :

- un bit signalant si le fichier est un dossier
- 3 bits pour le créateur `u` du fichier
- 3 bits pour le groupe `g` auquel appartient le créateur du fichier
- 3 bits pour les autres `o`

Les trois bits sont codés par un entier en base 8 (de 0 à 7) :

- le droit de lire le fichier
- le droit d'écrire dans le fichier
- le droit d'exécuter le fichier

Chapitre 14

Les dictionnaires

14.1 Construction

14.1.1 En compréhension

```
carre = {i:i**2 for i in range(6)}
```

14.2 Utilisation

14.2.1 Accès

```
seize = carre[4]
```

14.2.2 modification

On ne peut pas modifier les clés du dictionnaire, mais leurs valeurs :

```
>>> carre[2] = 5
>>> carres
{0:0, 1:1, 2:5, 3:9, 4:16, 5:25}
```

On peut aussi ajouter une clé avec cette notation :

```
carre = {}
for i in range(6):
    carre[i] = i**2
```

14.3 Parcours

14.3.1 Clés

```
>>> list(carre.keys())
[0,1,2,3,4,5]
```

14.3.2 valeurs

```
>>> list(carre.values())
[0,1,4,9,16,25]
```

14.3.3 parcours

```
for k in carre:
    print(k,carre[k])
```

a le même effet que

```
for k in carre.keys():
    print(k,carres[k])
```

Chapitre 15

Bases de données

15.1 n-uplets nommés

```
nombre,carré
0,0
1,1
2,4
3,9
4,16
5,25
```

La première ligne représente les *descripteurs*. Le couple (4,16) est alors *nommé* : 4 s'appelle `nombre` et 16 s'appelle `carré`.

15.2 SQL

15.2.1 Tables

En SQL les données sont regroupées en *relations* (ou *tables*).

nombre	carré
0	0
1	1
2	4
3	9
4	16
5	25

```
CREATE TABLE Carres (nombre INT, carre INT);
```

15.2.2 Enregistrements

Chaque ligne autre que celle des descripteurs est un n-uplet nommé appelé *enregistrement*.

```
INSERT INTO Carres VALUES (4,16);
```

15.2.3 Attributs

Pour extraire une colonne, on donne son attribut :

```
SELECT carre FROM Carres;
```

15.2.4 Utilisation

Pour calculer une racine carrée :

```
SELECT nombre FROM Carres WHERE carre=16;
```

Pour additionner les carrés :

```
SELECT SUM(carre) FROM Carres;
```

Chapitre 16

Algorithmes gloutons

16.1 Définition

On considère un problème d'optimisation (pour lequel il y a une relation d'ordre). Un algorithme résolvant ce problème est dit *glouton* s'il consiste à commencer par un optimum local.

16.2 Exemple

En Atlantide, il n'existe que trois sortes de pièces d'or :

- des pièces de 1 drachme atlante,
- des pièces de 3 drachmes atlantes,
- des pièces de 4 drachmes atlantes.

On cherche comment on peut rendre la monnaie de n drachmes atlantes.
L'algorithme glouton consiste à essayer de rendre d'abord des pièces de 4 drachmes atlantes tant que c'est possible, et ensuite seulement des pièces de valeur plus petite :

```
def monnaie(n:int)->dict:
    bourse = {k:0 for k in (1,3,4)}
    while n>=4:
        n -= 4
        bourse[4] += 1
    while n>=3:
        n -= 3
        bourse[3] += 1
    while n>=1:
        n -= 1
        bourse[1] += 1
    return bourse
```

16.3 Minimum local

Un minimum local peut être différent du minimum global. Par exemple l'algorithme glouton ci-dessus donne 3 pièces pour 6 drachmes atlantes $(4+1+1)$ alors qu'on peut faire avec 2 pièces seulement $(3+3)$.

Chapitre 17

K plus proches voisins

17.1 Distance

Un point du plan est représenté par un tuple de flottants (abscisse et ordonnée). Alors la distance entre A et B peut se calculer par

```
from math import sqrt

def distance(A,B):
    return sqrt((A[0]-B[0])**2+(A[1]-B[1])**2)
```

ou mieux :

```
from math import hypot

def distance(L1,L2):
    return hypot(A[0]-B[0],A[1]-B[1])
```

On considère un dictionnaire dont les clés sont des points du plan et les valeurs sont `'rouge'` et `'bleu'`. On cherche un algorithme permettant de deviner la couleur d'un point donné.

17.2 Algorithme des 3 plus proches voisins

```
def couleur(point:tuple)->str:
    L = []
    for key in nuage:
        L.append((distance(key,point),nuage[key]))
    L.sort()
    liste3 = [t[1] for t in L[:3]]
    liste3.sort()
    return liste3[1]
```

17.3 Algorithme des 5 plus proches voisins

```
def couleur(point:tuple)->str:
    L = []
    for key in nuage:
        L.append((distance(key,point),nuage[key]))
    L.sort()
    liste5 = [t[1] for t in L[:5]]
    liste5.sort()
    return liste5[2]
```