Guillaume Saupin

70 CONCEPTS MATHÉMATIQUES

expliqués avec PYTHON

DUNOD

Couverture : Florie Bauduin

Crédits iconographiques pour la couverture:

© Shutterstock / Julia Tim, GoodStudio, Pavlo S, cash1994, VectorHot, vectorwin

© Dunod, 2023
11 rue Paul Bert, 92240 Malakoff
www.dunod.com
ISBN 978-2-10-083613-0

83613 - (I) - OSB 80 - LUM - NRI

DUPLIPRINT
733, rue Saint-Léonard, 53100 MAYENNE
Dépôt légal : février 2023

Imprimé en France

Table des matières

Chapitre 1

Démarrer sur de bonnes bases

1.1 Préambule

Ce livre se destine aux lecteurs désireux d'assimiler les grandes idées mathématiques à travers une approche de type « apprendre en construisant ».

Cette approche, théorisée par Papert Seymour au sein de son laboratoire au MIT, met en avant le bénéfice pédagogique obtenu en construisant les systèmes étudiés.

Cette méthode s'applique à de nombreux domaines, comme par exemple la mécanique automobile, mais peut aussi être mise en œuvre pour des matières plus théoriques comme les mathématiques et la physique. Pour la mécanique, il s'agira par exemple de construire une boîte de vitesses, en imprimant les pièces en 3D.

Pour les mathématiques, le sujet de cet ouvrage, le matériau de construction naturel est la programmation. Les langages de programmation permettent en effet de donner vie aux concepts mathématiques les plus complexes, en permettant non seulement de les matérialiser sous forme graphique, mais aussi de les manipuler et d'expérimenter avec eux.

Tout au long de ce livre, le lecteur verra comment donner vie aux nombreuses et fructueuses idées mathématiques à l'aide de quelques lignes de code.

Motivations

Une image vaut mille mots

La raison principale qui a motivé la rédaction de ce livre est la matérialisation qu'apporte l'informatique aux mathématiques. De la même manière qu'un schéma permet souvent d'expliquer un concept ou une idée complexe rapidement, le code informatique permet de concrétiser rapidement toute idée mathématique. Mieux encore, de la même façon qu'il est possible de raisonner sur une idée en déplaçant des éléments d'un schéma, et en faisant évoluer sa réflexion graphiquement, le code permet cette plongée du monde abstrait des mathématiques dans un médium malléable. Ce livre illustre donc tous les concepts mathématiques à l'aide du nombre minimal de lignes de code permettant de leur donner chair.

Matérialiser les abstractions

Les mathématiques ont cette image d'une science compliquée et abstraite, construite par des esprits rêveurs et imaginatifs. Cette représentation est majoritairement inexacte et ne résiste pas à

une étude biographique et épistémologique des mathématiciens et de leurs découvertes. Gauss s'est fait connaître dans toute l'Europe en calculant la trajectoire de Cérès, et Euler appliquait son immense maîtrise des mathématiques au calcul des pensions de retraite de son institut de recherche.

Ces derniers ont bien souvent développé leurs théories pour répondre à des problématiques concrètes, et offrir des solutions à ces problèmes.

Ce qui crée cette impression de complexité découle de l'enseignement qui en est fait, où deux millénaires de mathématiques sont abordés en peu de temps. Le filtre des siècles a introduit toutes ces abstractions, qui ont l'avantage d'accélérer les calculs et les raisonnements, mais qui cachent la mécanique, la matérialité des concepts sur lesquels elles reposent. Implémenter ces concepts en Python permet de leur rendre chair.

Mécanisation

Une autre perspective qu'apporte la programmation aux mathématiques, c'est la mécanisation qui en découle. En effet, dès lors qu'un concept est implémenté et exécuté sur un ordinateur, il se réduit au final en une succession d'opérations simples, mécaniquement appliquées. Cette représentation est très fructueuse, surtout lorsqu'il est question d'intelligence artificielle.

Un formidable accélérateur

La conséquence majeure de l'utilisation de l'informatique, pour découvrir les mathématiques, est l'accélération formidable qu'elle apporte. Tous les points cités ci-dessus contribuent à réduire le temps d'acquisition des notions mathématiques, tout en renforçant la mémorisation. L'approche suivie dans cet ouvrage permettra donc au lecteur d'acquérir rapidement les nombreux concepts mathématiques présentés.

Public

Basé essentiellement sur le programme de Licence de mathématiques, cet ouvrage en parcourt la plupart des notions. Il a donc été conçu pour pouvoir être abordé par un élève de premier cycle universitaire.

Cependant, il peut aussi profiter à tout élève en Master ou au-delà, évoluant dans le domaine des mathématiques, mais aussi des sciences de l'ingénieur ou de l'informatique, qui chercherait un récapitulatif synthétique des grandes notions mathématiques avec un prisme opérationnel.

Enfin, toute personne curieuse de comprendre en finesse les mathématiques, leur usage en science informatique, et finalement tous les outils et services qui peuplent nos ordinateurs, nos téléphones et nos vies trouvera ici matière à réflexion.

Notions

Les notions abordées tout au long de cet ouvrage sont essentiellement tirées du programme de Licence de mathématiques. Le champ des mathématiques est très large, et il a bien sûr fallu opérer un choix.

Ce dernier s'est fait en partie suivant les goûts de l'auteur, mais surtout sur la base de son expérience de près de deux décennies dans la recherche en mathématiques appliquées, en intelligence artificielle au sein du CEA mais aussi de start-ups.

Toutes les notions présentées ici seront donc utiles à tout ingénieur travaillant dans le domaine des sciences, de l'informatique, de la biologie...

Pour faciliter la navigation parmi ces concepts, ils ont été regroupés dans les grandes sections suivantes :

— Théorie des nombres
— Analyse
— Algèbre linéaire
— Statistique
— Géométrie
— Intelligence artificielle / *Machine Learning*
— Physique

Applications

Pour donner à voir le plein potentiel de ces notions et comment elles peuvent être mises en œuvre, plusieurs applications poussées de ces notions sont présentées en fin d'ouvrage.

Il s'agira par exemple de voir comment la programmation peut aider à comprendre la physique sous sa forme variationnelle, ou encore coder l'une des méthodes de *Machine Learning* les plus fructueuses de ces dernières années : le *Gradient Boosting* pour les arbres de décisions.

Prérequis

Les diverses notions introduites dans ce livre le sont progressivement et ne requièrent pas de connaissances préalables au-delà du niveau baccalauréat en mathématique. L'utilisation systématique de code informatique pour illustrer chaque notion abordée simplifie grandement leur abord, en les démystifiant et en leur donnant corps.

Le chapitre suivant introduit les principales notions utiles pour comprendre le langage Python. C'est un langage simple d'abord, dont la prise en main est rapide, et qui permet naturellement d'exprimer des concepts mathématiques.

C'est par ailleurs un langage largement répandu dans le monde professionnel, et apprendre à le maîtriser ne peut être que bénéfique.

Remerciements

L'auteur tient à remercier en tout premier lieu ses parents, qui ont su développer son goût pour les sciences en général et les mathématiques en particulier, et qui lui ont permis de plonger dans le monde de l'informatique très tôt.

Il n'oublie pas non plus sa femme et ses deux filles, pour l'avoir encouragé dans cette expérience solitaire qu'est la rédaction d'un livre. Il espère que ses deux filles seront rapidement deux lectrices intéressées par cet ouvrage.

Enfin, que soit remerciée l'éditrice qui a bien voulu croire en ce projet et a permis sa réalisation.

1.2 Mathématique, programmation et langage

L'objet de ce livre est d'explorer les nombreux concepts mathématiques avec le support de la programmation, afin de disposer d'un laboratoire et des instruments utiles à cette exploration.

Langages de programmation

Depuis l'émergence de l'informatique dans les années 1950, de nombreux langages de programmation ont été développés. Le site *github*, l'une des deux grandes plateformes d'hébergement de code open source en compte plus d'une cinquantaine.

Parmi les plus répandus se trouvent Python, Javascript, Java, TypeScript, Go, C++, Rust, Lisp...

Domaines de prédilection

Chacun de ces langages a son domaine de prédilection, même si techniquement ils sont substituables l'un à l'autre. Le langage Python est par exemple beaucoup utilisé pour assurer les fonctionnalités *back-end* des applications web, tandis que Javascript se cantonne plutôt aux aspects interface web.

Python bénéficie aussi d'un écosystème très vivace dans le domaine de la Data Science.

Le Go et le C++, étant de plus bas niveau et offrant de bien meilleures performances en temps de calcul se retrouvent plus généralement dans des applications gourmandes en temps de calcul. Rust met lui l'accent sur la sécurité et notamment la gestion de la mémoire. Il s'est ainsi imposé dans le développement des applications systèmes, dont il est crucial d'assurer la fiabilité.

Critères de choix

Parmi ce large choix de langages, il a fallu en choisir un qui soit adapté à l'ambition de ce livre : plonger dans l'univers des mathématiques et en expérimenter les concepts les plus intéressants.

Simplicité de mise en œuvre

Le langage retenu devait satisfaire plusieurs contraintes, à commencer par une simplicité d'utilisation. En effet, cet ouvrage se destinant à un public pas forcément encore aguerri à la programmation, le langage ne devait pas être un frein à la compréhension du propos.

Math friendly

Il devait aussi être *math friendly*, c'est-à-dire offrir nativement la possibilité de manipuler simplement les constituants de base des mathématiques : nombres, fonctions, ensemble... Les librairies disponibles sur étagère devaient aussi être nombreuses dans le domaine des mathématiques.

Interactivité

Pour permettre de manipuler aisément les divers notions et objets mathématiques, il était indispensable de s'appuyer sur un langage doté d'une REPL (*Read Eval Print Loop*), c'est-à-dire d'une console interactive. Cette dernière permet d'interagir dynamiquement avec le code. Tous les langages n'en disposent pas, et notamment les langages compilés comme le C++ ou le Go.

Capitalisation

Enfin, il était important que ce langage soit utile au lecteur et que les compétences acquises en mathématiques et en programmation ici puissent être transposées immédiatement dans d'autres domaines.

Pourquoi Python ?

Peu de langages répondent finalement à ces exigences. Afin d'incarner les divers concepts mathématiques abordés dans ce livre, le langage retenu est donc le Python.

Plusieurs raisons ont gouverné ce choix :

— La simplicité du langage : Python dispose d'une syntaxe facilement assimilable et ne s'embarrasse pas avec le typage des objets.

— L'existence d'une console : cette dernière permet de manipuler de manière dynamique le code écrit en interagissant de manière interactive.

— L'existence d'un écosystème très vigoureux : de nombreuses librairies sont disponibles librement en ligne ce qui offre énormément de possibilités.

Les bases de Python

Afin de pouvoir tirer parti du formidable laboratoire expérimental que constitue la programmation pour les mathématiques, il faut au préalable maîtriser un langage de programmation.

La maîtrise d'un langage informatique peut sembler délicate à première vue, et cela peut-être le cas, suivant le langage étudié. Fort heureusement, le langage retenu pour cet ouvrage est unanimement reconnu comme étant l'un des plus simples à maîtriser.

Cette simplicité découle de plusieurs points :

— la syntaxe est simple : en Python, pas de parenthèse, ni d'accolades. Les blocs de code sont identifiés visuellement suivant leur indentation.

— Le niveau d'abstraction vis-à-vis de la machine est important : inutile en Python de se soucier de l'allocation de la mémoire, de sa libération... Tout est fait automatiquement.

— La richesse des types natifs : entier, flottant, chaîne de caractères, liste, ensemble, dictionnaire... beaucoup d'objets informatiques sont supportés nativement.

Afin de permettre au lecteur de se familiariser avec le langage Python ou de rafraîchir ses souvenirs, les sections suivantes reviennent sur les bases de ce langage.

Les types

Cette rapide présentation du langage Python commence par la présentation des principaux types. Dans un langage de programmation, un type définit un objet particulier qui va pouvoir être manipulé par le langage. *A minima*, presque tous les langages de programmation supportent deux types : les entiers et les nombres réels.

Python est beaucoup plus complet et étoffe cette liste avec les booléens, les chaînes de caractères, les listes, les ensembles ou encore les dictionnaires. Le listing ci-dessous explique comment créer chacun de ces types natifs.

Principaux types natifs en Python

```
# a est un entier
a = 12
print(type(a))
#> <class 'int'>
```

```
# x est un réel
x = 3.5
print(type(x))
#> <class 'float'>

# c est un booléen
c = True
print(type(c))
#> <class 'bool'>

# l est une liste
# les listes peuvent être hétérogènes
l = [1, 2, a, x, c]
print(type(l))
#> <class 'list'>

# A est un ensemble
# Contrairement à une liste, un ensemble garantit l'unicité
# des éléments en son sein
A = {1, 1, 2, 3, 4, 5, 6}
print(type(A))
#> <class 'set'>

# s est une chaîne de caractères
s = 'python'
print(type(s))
#> <class 'str'>

# D est un dictionnaire
D = {'key1' : 1, 'key2': 2, 'key3': 3}
print(type(D))
#> <class 'dict'>
```

Disposer nativement de nombreux types est un avantage précieux. Le cœur d'un algorithme est souvent la structure de données sous-jacente. Disposer nativement de types complexes permet de créer rapidement des structures de données complexes, reflétant au mieux la nature d'un problème.

Ce qui est encore plus précieux c'est de pouvoir appliquer des opérations sur ces types. Là encore, le langage Python est riche, comme en témoignent les lignes de code du listing ci-dessous.

Opérations sur les types natifs

```
# les opérations arithmétiques de base sont assurées sur les entiers et les
    réels
a = 1
b = 2
print(a + b)
#> 3
print(a - b)
#> -1
print(a * b)
#> 2
print(a / b)
#> 0.5

# il est possible de sommer entier et flottant
```

```
c = 3.5
print(a - b + c)
#> 2.5

# Les chaînes de caractères sont bien dotées aussi
s = 'python'
print(s + s)
#> pythonpython
print(len(s))
#> 6
# Les strings sont des listes de caractères
print(s[:3])
#> pyt
```

À noter que des opérateurs peuvent avoir des sens différents suivant les types. L'opérateur + par exemple exécute une somme sur des entiers et des réels, mais appliqué sur des tableaux ou des chaînes de caractères, il réalise une concaténation. Appliquer ces opérations à des types non compatibles génère des exceptions.

Typage dynamique

Il est crucial de noter que Python est un langage qui gère les types de manière dynamique. C'est-à-dire que contrairement à des langages comme le C/C++, Java ou encore Go, le type des variables n'est pas fixé définitivement. Ce typage se fait généralement à la compilation, étape qui n'existe pas en Python.

Le code est uniquement interprété et non compilé, c'est-à-dire que lors de l'exécution d'un programme en Python, le code est lu, et seule la validité de la syntaxe est vérifiée.

Ce n'est qu'à l'exécution que la validité du code en termes de type est éprouvée. Cela signifie qu'il est possible d'écrire du code qui somme une chaîne de caractères avec des dictionnaires sans que cela pose souci jusqu'à l'exécution du code.

Typage dynamique en Python

```
# Cette fonction exécute une opération
# impossible entre un entier
# et une chaîne de caractères
def this_codes_fails():
    a = 12
    b = 'rabbit'
    return a + b

# le type d'une variable peut changer dynamiquement au cours du programme
var1 = 12
print(type(var1))
#> <class 'int'>
var1 = 'rabbit'
print(type(var1))
#> <class 'str'>

print('jusque la tout va bien')
#> jusque la tout va bien
# L'interpréteur va échouer ici en levant une exception
print(this_codes_fails())
#> TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

La syntaxe

La syntaxe d'un langage est ce qui définit la manière correcte de l'écrire. Par exemple, une chaîne de caractères se définit entre guillemets, le point est utilisé dans les nombres à virgule, on distingue minuscules et majuscules...

Le listing ci-dessus est assez complet pour ce qui est de la déclaration de valeur littérale, c'est-à-dire ne résultant pas d'un calcul.

À cela s'ajoutent des règles sur le nommage des variables et des fonctions. Ces dernières ne peuvent commencer par un chiffre par exemple, comme le met en évidence le listing ci-dessous.

Nommage des fonctions variables

```
# Le nom de cette variable n'est pas valide
3_not_possible = 6
#> SyntaxError: invalid decimal literal

# Ces deux variables sont bien distinctes,
# en raison de la casse (majuscule/minuscule) qui n'est pas la même
variable_1 = 1
vAriable_1 = 2

print(variable_1)
# > 1
print(vAriable_1)
# > 2

# Le nom de cette fonction n'est pas valide non plus
def 3_not_possible_function():
    return 3
```

La syntaxe est un élément généralement déroutant pour les personnes qui découvrent la programmation et qui se retrouvent à écrire du code qui n'est pas compris par l'interpréteur ou le compilateur. Il faut bien comprendre que ces règles doivent être impérativement respectées, sans quoi la machine n'est pas capable de comprendre le programme.

Fort heureusement, dans le cas du Python, pour tout programme non syntaxiquement correct, des erreurs sont affichées afin d'aider le développeur à traquer son erreur. De plus, les éditeurs de code modernes (et même en vérité anciens comme *emacs*) mettent en avant de manière visuelle les erreurs.

Ce n'est pas le cas en revanche pour l'identification des blocs, qui jouent un rôle central dans un programme, et qui, en Python, sont identifiés uniquement suivant l'indentation, c'est-à-dire suivant leur alignement.

Un bloc de code définit un ensemble de lignes qui font partie d'une unité logique, comme une fonction, une boucle... Ce sont donc des lignes qui sont exécutées un même nombre de fois. Le Python est assez unique dans sa façon de traiter le sujet, dans la mesure où les blocs sont identifiés selon leur alignement, ou indentation.

Dans la plupart des autres codes, cette identification se fait à l'aide de parenthèses ou d'accolades. Quelques exemples sont donnés dans le listing ci-dessous.

Quelques exemples d'indentation

```
# Cette ligne ne va pas être comprise
# car elle commence par un espace.
# L'indentation est incorrecte
 variable_2 = 2
```

```
#> IndentationError: unexpected indent

# Cette boucle calcule la somme des nombres de 1 à 100
# de manière brute force, contrairement à ce qu'avait fait
# Gauss étant jeune étudiant
sum = 0
for i in range(1, 101):
    # ce bloc, identifié par l'indentation de 4 caractères
    # est exécuté autant de fois que la boucle l'impose
    sum += i

print(sum)
# > 5050

# Le corps d'une fonction se définit
# toujours dans un bloc en retrait
# d'une indentation
def gauss_sum():
    sum = 0
    for i in range(1, 101):
        # ce bloc, identifié par l'indentation de 4 caractères
        # est exécuté autant de fois que la boucle l'impose
        sum += i
    return sum
# le bloc de la fonction se termine ici

print(gauss_sum())
# > 5050
```

La syntaxe se complète toujours d'une grammaire qui précise comment combiner correctement les éléments syntaxiques.

Les structures de contrôle

Afin de contrôler le flot d'exécution du code et de pouvoir appliquer des opérations différentes pour des conditions différentes, les langages de programmation impératifs tels que Python utilisent des structures de contrôle.

Ce sont des mots-clefs du langage qui permettent, suivant le résultat de tests, d'exécuter tel ou tel code. Ils autorisent ainsi l'existence de plusieurs branches de code qui vont être exécutées suivant les données en entrée.

En Python, il en existe finalement peu par rapport à d'autres langages. Il s'agit essentiellement du *if* et du *else*, comme le montre le listing ci-dessous.

Structures de contrôle

```
import random

# la condition est le résultat d'un test,
# c'est-à-dire que c'est un objet qui peut être interprété
# comme un booleen. Deux valeurs sont donc possibles : vrai ou faux
condition = True

# La structure de contrôle de base en Python
# prend la forme suivante
if condition:
```

```
    # alors on exécute le code dans le bloc qui suit
    print("la condition est vraie")
#> la condition est vraie

# Il est aussi possible d'avoir une alternative
# dans le cas où la condition serait fausse.
# C'est ce que permet le mot-clef else
if condition:
    # alors on exécute le code dans le bloc qui suit
    print("la condition est vraie")
else:
    # sinon ce bloc s'exécute
    print("la condition est fausse")
#> la condition est vraie

# Il est possible de mettre le code de la condition
# directement sur la ligne du if
x = random.random()
if x < 0.5:
    # alors on exécute le code dans le bloc qui suit
    print(f"{x} est strictement inferieur a 0.5")
else:
    # sinon ce bloc s'exécute
    print(f"{x} est superieur a 0.5")
#> résultat suivant la valeur de x


#Il est aussi possible d'enchaîner plusieurs tests avec elif
if x < 0.5:
    # alors on exécute le code dans le bloc qui suit
    print(f"{x} est strictement inferieur a 0.5")
elif x < 0.6:
    print(f"{x} est entre 0.5 et 0.6")
else:
    # sinon ce bloc s'exécute
    print(f"{x} est superieur a 0.5")
#> résultat suivant la valeur de x
```

Il est important de noter qu'il est possible d'utiliser autre chose que des booléens comme valeurs pour les tests. Dans ce cas, la valeur fausse correspond généralement à l'élément nul du type. Pour les listes, il s'agit par exemple de la liste vide $[]$. Pour les entiers et les réels, il s'agit naturellement du zéro. Pour les dictionnaires et les ensembles c'est l'ensemble vide $\{\}$.

Toutes les autres valeurs sont considérées comme vraies, comme le montre le listing ci-dessous.

Structures de contrôle sans booléen

```
import random

# La condition est ici une liste vide.
# Elle est considérée comme fausse
condition = []

if not condition:
    print("la condition est vraie")
else:
    print("la condition n'est pas vraie")
```

```
#> la condition n'est pas vraie

# Ici, la condition contient un élément
# Elle est considérée comme vraie
condition = {1}

if not condition:
    print("la condition est vraie")
else:
    print("la condition n'est pas vraie")
#> la condition n'est pas vraie
```

À noter qu'il est aussi possible d'utiliser le couple *if / else* sur une seule ligne. Cela ne change rien aux performances du code, mais cela accroît parfois la lisibilité. Voir le listing ci-dessous.

Opérateur ternaire

```
condition = False
# L'opérateur ternaire est utilisé ici
# afin de condenser le traitement sur une ligne.
# La lisibilité est accrue
var = 12 if condition else 24
print(var)
#> 24
```

Cette utilisation sur une seule ligne est souvent nommée opérateur ternaire, car elle contient trois termes : la condition, la valeur dans le cas vrai, la valeur dans le cas faux.

Les structures itératives

Python étant un langage impératif, il offre à ses utilisateurs les outils nécessaires pour itérer sur des données. À cet effet, les mots-clefs du langage *while et for* sont utilisés.

Le mot-clef *while* permet d'exécuter indéfiniment un bloc de code, jusqu'à ce qu'une condition soit atteinte. *for* de son côté itère sur une collection d'objets.

Dans les deux cas, il est possible de sortir de ces deux boucles d'itérations à l'aide du mot-clef *break*. Passer directement à l'itération suivante se fait par l'appel du mot-clef *continue*.

Structures itératives *for* et *while*

```
# La forme la plus classique d'une boucle d'itération est la suivante :
# il s'agit de répéter n fois le même bloc de code.
n = 5
for i in range(0, n):
    print(i)
#> 0
#> 1
#> 2
#> 3
#> 4

# L'utilisation de while permet d'arriver au même résultat
# avec cependant une formulation un peu moins lisible.
i = 0
while i < 5:
    print(i)
    i += 1
```

```
#> 0
#> 1
#> 2
#> 3
#> 4

# Le Python permet d'itérer directement sur
# des structures de données itérables
# comme les listes.
data = [0, 1, 2, 3, 4]
for i in data:
    print(i)
#> 0
#> 1
#> 2
#> 3
#> 4

# Le mot-clef continue permet de sauter
# directement à l'itération suivante.
# Dans l'exemple suivant, il est utilisé
# pour n'afficher que les nombres pairs.
for i in range(0, 5):
    if i % 2 == 1:
        # pour les nombres impairs, la boucle s'arrête ici
        continue
    print(i)
#> 0
#> 2
#> 4

# Le mot-clef break permet quant à lui
# d'interrompre complètement une boucle.
# Ci-dessus, on recherche un nombre dans un tableau
# une fois celui-ci trouvé, la boucle s'arrête.
data = [6, 2, 1, 3, 0, 4, 5]
for d in data:
    if d == 0:
        print("found 0")
        break
# > found 0
```

Comprehension list et *Dictionary*

Dans certains cas, il est possible d'écrire de manière plus compacte des boucles. C'est le cas en particulier des structures de données de type liste ou dictionnaire. Pour ces dernières, il est possible d'embarquer dans leur définition toute la mécanique d'itération. Elles sont alors nommées *comprehension list* ou *comprehension dictionary*.

Leur utilisation permet de construire rapidement des dictionnaires et des listes depuis d'autres listes ou dictionnaires. Il est aussi possible de procéder à des filtrages ou à des traitements conditionnels comme le fait apparaître le listing ci-dessous.